# Captura de ecrã 2015-12-30, às 18.14.12.png

|  |
| --- |
| UNIVERSIDADE DO MINHO |
| Relatório Final |
| Sistemas Distribuídos |
| Mestrado Integrado em Engenharia Informática |
| C:\Users\André\Desktop\carlos.jpgCaptura de ecrã 2015-12-30, às 18.17.12.png |
| **Carlos Silva(a66553) Vitaliy Parytskyy(a69857) Diogo Constâncio(a70719)** |
|  |

**2015/2016**

# Introdução

Com vista a aplicar os conhecimentos adquiridos na unidade curricular de Sistemas Distribuídos, foi-nos proposto um trabalho prático com o objetivo de desenvolver uma aplicação que suportasse um serviço de táxis, com utilizadores normais e potenciais condutores.

A aplicação é feita sobre Java, TCP/IP, com o intuito de permitir a utilização do serviço por parte de múltiplos utilizadores no mesmo instante.

Neste documento vamos descrever o modelo da aplicação e o protocolo que suporta as suas várias funcionalidades.

# Modelo do Servidor

Qualquer serviço que pretenda servir múltiplos clientes tem de suportar, necessariamente, multi threading, como é o caso deste servidor.

Para permitir a ligação de múltiplos clientes o servidor opera sobre 1 thread principal, que monitoriza uma porta predefinida à espera de um potencial utilizador. Quando um utilizador se liga a esta porta é redirecionado para outra porta e trabalhado noutra thread por forma a libertar a thread e porta principal para quaisquer outros utilizadores.

Este novo utilizador tem agora a sua própria thread e porta, que será usada ao longo da ligação para esperar por comunicação do cliente conforme definido pelo protocolo. Por uma questão de libertar portas o mais cedo possível é criada outra thread e associada outra porta ao cliente para estabelecer e manter um protocolo de heartbeat, desta forma o socket principal pode não ter timeout ativo, dado que o cliente pode demorar muito tempo entre comunicações com o servidor, o que simplifica o funcionamento da aplicação.

Em resposta a certos comandos o servidor cria outro par de thread e sockets para o suporte de notificações PUSH, como por exemplo para notificar o cliente da chegada do condutor sem bloquear a utilização de demais serviços que pudessem a vir ser implementados futuramente.

# Modelo do Cliente

O modelo de funcionamento do cliente é muito semelhante ao do servidor, exceto ao invés de criar sockets e esperar por ligações o cliente recebe como resposta aos pedidos relevantes um número de porta ao qual estabelecer ligação.

Este é o caso, por exemplo, quando o servidor cria uma porta para o serviço de notificações PUSH.

Os detalhes da comunicação esta descritos na secção referente ao protocolo.

# Modelo do Protocolo Cliente-Servidor

Por forma a manter uma via de comunicação consistente um protocolo foi definido de acordo com um conjunto de regras simples. Todas as comunicações principais, ou seja, excluindo heartbeat, têm um prefixo consistente:

username:[user];password:[pw];command:[command];

O objetivo deste prefixo é primeiro, permitir a autenticação quaisquer pedidos, através do username e password, e de definir de forma concreta que lógica aplicar aos demais argumentos que seguem após command, tomando a forma de “argumento1:[argumento1];…”. O servidor pode retornar dados se tal for necessário usando um formato semelhante ao prefixo : [“success” ou “failure” ]:;data1:[data1];…

O protocolo torna trivial estender o leque de funções suportadas pelo protocolo devido à sua flexibilidade, e de momento são suportados 6 destes comandos:

1. login
   1. Argumentos:
   2. Retorno: “success” seguido da classe de utilizador serializada, ou “failure” com uma descrição do erro
   3. Funcionalidade: Tal como o nome indica este comando tem como objetivo autenticar o utilizador por forma a receber os dados que o servidor tem sobre si.
2. register
   1. Argumentos: plate:[matricula];make:[modelo]
   2. Retorno: “success” seguido da classe de utilizador serializada, ou “failure” com uma descrição do erro
   3. Funcionalidade: Este comando tem como objetivo registar o utilizador por forma a permitir posteriores operações. O login é feito automaticamente com os dados fornecidos.
3. request
   1. Argumentos: startx:[coordenada de inicio x]; starty:[starty]; endx:[ coordenada de fim x], endy:[endy];
   2. Retorno: “success:; plate:[matricula do condutor]; make:[modelo do veiculo]; eta:[tempo previso para chegada]; port:[porta para PUSH]”, ou “failure” com uma descrição do erro
   3. Funcionalidade: Este comando tem como objetivo permitir ao utilizador requesitar um táxi na sua localização que o leve até ao ponto especificado. O servidor retorna os dados do veiculo que foi atribuído como táxi.
4. driver
   1. Argumentos: x:[localização x]; y:[starty];
   2. Retorno: “success:;port:[porta para PUSH]”, ou “failure” com uma descrição do erro
   3. Funcionalidade: Este comando tem como objetivo permitir a um utilizador assinalar a sua disponibilidade como condutor assinalando a sua localização. A porta retornada é usada para o agora condutor receber a notificação que lhe foi atribuído um cliente.
5. arrival
   1. Argumentos: x:[localização x]; y:[starty];
   2. Retorno: “success:;port:[porta para PUSH]”, ou “failure” com uma descrição do erro
   3. Funcionalidade: Este comando tem como objetivo permitir a um utilizador assinalar a sua disponibilidade como condutor assinalando a sua localização. A porta retornada é usada para o agora condutor receber a notificação que lhe foi atribuído um cliente.
6. destination
   1. Argumentos: ;
   2. Retorno: “success:;”, ou “failure” com uma descrição do erro
   3. Funcionalidade: Este comando permite ao condutor informar o servidor que levou o cliente ao destino.

# Modelo do Protocolo Servidor-Cliente

O protocolo do servidor para o cliente é mais simples, dado que apenas server para pequenas notificações, no caso do condutor para o informar de um cliente e o seu destino, e para o cliente para o informar do preço a pagar quando chega ao destino.

# Conclusão

Este trabalho prático permitiu-nos perceber e aplicar mecanismos de exclusão mútua como

o uso de ***Locks*** ou ***synchronized*** e as variáveis de condição do Java e ainda o uso de ***Sockets*** para a comunicação entre duas ***Threads*** diferentes.

Tivemos de tomar várias decisões durante a implementação, entre as quais destaca-se:

* O envio dos pedidos do cliente para o servidor são feitos através de mensagens escritas num ***BufferReader*** criado paracada ***Thread*** cliente, sendo estas mensagens depois interpretadas e executadas pelo servidor.
* Decidimos implementar um mecanismo de ***ping***, em que o Servidor a cada 0.5 segundos escreve uma mensagem num ***BufferReader,*** acima citado, sendo que enquanto um cliente estiver conectado este tenta ler a mensagem escrita pelo Servidor, escrevendo também uma mensagem de seguida por forma a indicar que a conexão ainda esta ativa.