Programação Concorrente

Trabalho Prático
Relatório de desenvolvimento do Gestor de deslocações

Bruno Guedes(68707) Joel Carvalho(68698)

1 de Junho de 2016

Licenciatura em Ciências da Computação

Introdução

Neste relatório do Gestor de deslocações para a cadeira de Programação Concorrente vamos explicar como foi implementado o Servidor e o Cliente, bem como o seu funcionamento. Vamos também demonstrar a comunicação entre eles.

Tal como foi pedido no enunciado o servidor foi criado na linguagem de programação Erlang e o Cliente em Java, estabelecendo comunicação entre eles através de Sockets TCP.

Servidor

- Implementação

São criados inicialmente 4 processos, cada um contendo um map.

O primeiro map guarda os utilizadores criados (como chave), as passwords, uma string (que indica se é cliente ou taxista), uma flag para saber se está online ou não e o pid do utilizador. Este map permite saber que utilizadores estão online e que utilizadores são clientes ou taxistas.

O segundo guarda os taxistas (como chave), as matriculas dos veículos e uma flag que nos permita saber se o taxista está disponível ou ocupado. Este map permite saber que taxistas estão disponíveis.

No terceiro são guardados os utilizadores (como chave) , as suas coordenadas e um tempo (em horas,minutos,e segundos). Este map permite guardar e saber a localização de cada utilizador, e guardar o tempo (/*para que serve o tempo*/).

O quarto e último map guarda uma associação entre clientes e taxistas (sendo este tuplo a chave), 4 flags (sendo estas, flags de controlo quando o cliente chama o taxi), o tempo que demora o taxista a chegar ao cliente, o tempo de viagem até ao destino e o preço. As flags dão a informação de onde está o taxi (se já chegou ao cliente, se já chegou ao destino, ou se a viagem foi cancelada).

- Funcionamento

Inicialmente temos um processo à espera de conexão e que cria um processo novo cada vez que um novo utilizador é conetado.

Cada map tem um processo associado, que decide as tarefas a realizar. Quando um utilizador quer aceder a alguma informação, ou alterar alguma informação dos maps tem que enviar uma mensagem para o processo associado ao map e apenas este acede e/ou altera a informação guardada nos mapas garantindo assim concorrência entre os processos.

Quando o cliente chama por um taxi, o servidor, após responder ao cliente que o taxi vai a caminho, é criado um novo processo onde, através de um sleep(Tempo), esse processo vai demorar dentro do sleep, exactamente o tempo que o taxi demora a chegar ao utilizador. Caso este sleep chegue ao final sem o processo ser interrompido, o servidor comunica que o condutor chegou ao passageiro e entramos num novo sleep(Tempo2), mas agora sendo Tempo2, o tempo que demora a chegar ao destino com o passageiro lá dentro. De notar que o cliente fica em espera caso não existam taxis disponíveis (o mesmo acontece quando o taxista diz que está disponível e não encontra cliente disponíveis).

Quando o cliente comunica um cancelamento do taxi atribuido, tanto seja antes como depois do condutor chegar ao cliente, haverá um exit(pid,kill) que mata o processo onde estão a correr os sleeps procedendo então ao envio de mensagens para ambos os

clientes associados à viagem, sendo que o tempo que fica guardado agora é o tempo que passou até ao cancelamento e não o tempo que demora a chegar ao destino

Cliente

Cada cliente terá uma classe **TaxiSer** que é chamada pela classe **Main**. Dentro da classe **TaxiSer** temos duas classes: ***Message*** e ***InterfaceIO***, sendo que ambas extendem Thread e vão correr em concorrência. A thread ***Message*** trata de fazer a comunicação com o servidor, enviar perguntas e receber as respostas respectivas do servidor para o cliente, enquanto que a classe ***InterfaceIO*** trata da interface da aplicação que vai ser usada pelo cliente, e consequentemente decide qual a informação que a thread da classe ***Message*** vai enviar ao servidor, assim como trata também da informação recebida na mesma pelo servidor.

A sincronização e passagem de informação entre as threads ***Message*** e ***InterfaceIO*** é realizada através da leitura e escrita de informação nas variáveis de uma classe **Client** que é passada como parâmetro no construtor de ambas.

Ainda dentro do cliente, temos uma classe **Interface(**chamada dentro da thread ***InterfaceIO*****)** que contém todos os métodos que imprimem as views dos menus para a aplicação.

Para terminar, nota ainda para as classes que tratam da interface GUI da aplicação em swing chamadas na classe **Interface** (sendo que todas são threads, controladas com wait-notify):

- ***InitialMenu*** - Menu inicial com opções Login, Register, Exit.
- ***LoginMenu*** - Menu de Login com inserção de Username e Password.
- ***RegisterMenu*** - Menu de Registo com inserção de Username, Password, Tipo(Driver/Buyer), Matricula (só no caso do buyer).
- ***DriverMenu*** - Menu do Taxista com opções “I am Available” e “Exit”.
- ***BuyerMenu*** - Menu do Buyer com opções “Call Taxi” e “Exit”.
- ***AvailableMenu*** - Menu de inserção de coordenadas para o Taxista.
- ***LocalizationMenu*** - Menu de inserção de coordenadas de localização para o Buyer.
- ***DestinationMenu*** - Menu de inserção de coordenadas de destino para o Buyer.
- ***AutomaticBMenu*** - Area de texto com as notificaçoes de viagem recebidas pelo Buyer vindas do servidor e botão com a opçao de cancelar uma viagem.
- ***AutomaticDMenu*** - TextArea com as notificações de viagem recebidas pelo Taxista vindas do servidor.
- ***MessageMenu*** - TextAreas com mensagens importantes para o Utilizador ao longo do uso da aplicação (um pouco ao modo das pop-ups).

Nota: A negrito classes simples, a “negrito, itálico e sublinhado” classes que extendem threads.

Comunicação Servidor - Cliente

Tal como foi referido anteriormente estabeleceu-se conexão entre o Servidor e o Cliente via Sockets TCP. Apresentamos agora uma tabela com o protocolo das mensagens. As respostas preenchidas a verde acontecem quando o servidor sucede ao proceder à tarefa pedida, a vermelho é quando o servidor não sucede e manda de volta a razão do porque não conseguiu, enquanto que as azuis representam mensagens de pedido de informação ao servidor e não “ordens”:

Cliente para Servidor	Opção de Resposta 1	Opção de Resposta 2	Opção de Resposta 3
{login, [Username,Password]}	"Utilizador_entrou\n"	"Password_errada\n"	"Utilizador_nao_existe\n"
{logout, [Username]}	"Utilizador_saiu\n"		
{createAccount,[Username,Password]}	"User Created\n"	"Utilizador_ja_existe\n"	
{createAccountDriver,[Username,Password, Matricula]}	"Driver Created\n"	"Utilizador_Condutor_ja_existe\n"	
{closeAccount,[Username,Password]}	"Conta_Removida\n"		
{normalToDriver,[Username, Matricula]}	"TaxiAdicionado_normalToDriver\n"		
{availabletaxi,[Username,X,Y]}	Espera pela ocurrencia de um callTaxi e recebe a mesma resposta que o Buyer		
{callTaxi, [Username,X,Y,X1,Y1]}	"Taxi_a_caminho_das_coordenadas " ++ X ++ "_" ++ Y ++"_para_levar_nas_coordenadas_" ++ X1 ++ Y1 ++ "_demora_"++ L ++ "\n"	Envia a mesma mensagem da opção 1 para o taxista associado	

{cancelUser,[Username]}	"canceled demorou_" ++ U ++ "_o_preco_a_pagar_" ++ Z ++ "\n"		
{cancelDriver,[Taxista]}	"canceled demorou_" ++ U ++ "_o_preco_a_pagar_" ++ Z ++ "\n"	"notCanceled\n"	
{okchegoutaxiDriver,[Taxista]}	"notTaxi\n"	"taxi tempo_de_viagem_" ++ P ++ "_Preco_"++O++"\n"	
{okchegoutaxiUser,[Username]}	"notTaxi\n"	"taxi tempo_de_viagem_" ++ P ++ "_Preco_"++O++"\n"	
{okchegoudestinoDriver, [Taxista]}	"notDestino\n"	"destino Valor_a_pagar_"++ O ++ "\n"	
{okchegoudestinoUser, [Username]}	"notDestino\n"	"destino Valor_a_pagar_"++ O ++ "\n"	
{online}	"Utilizadores_online_recolhidos\n"		

Conclusão

Como conclusão fizemos uma introspectiva ao trabalho realizado, e, salvo erro, acreditamos que cumprimos todos os pontos pedidos no enunciado, sendo que relativamente ao funcionamento da viagem, a nossa aplicação está ligeiramente diferente no seguinte aspecto:

- Enunciado: o condutor deverá informar o servidor quando chegar ao local onde se encontra o outro passageiro e também quando chegar ao final da viagem, e o servidor informa o passageiro.
- Na nossa aplicação: depois de o servidor informar o condutor e o passageiro que a relação entre ambos foi estabelecida, é o servidor que automaticamente faz a estimativa do tempo de chegada, faz um spawn com um sleep desse tempo e quando terminar o sleep informa o condutor que o mesmo já chegou aos checkpoints e a mesma informação é enviada ao passageiro.

A decisão desta mudança teve como base uma má interpretação desse ponto do enunciado, e, quando finalmente o interpretamos tomamos a opção de não alterar esse aspecto da aplicação porque foi das implementações que mais trabalho deu (todo esses spawn()'s, sleep's e kill's de processos no servidor, tal como a sincronização das mensagens com o cliente), e apesar da mudança ser relativamente simples, não quisemos estar a deitar fora tudo o que fizemos.

Fora isso, e mais uma vez, salvo erro, funciona tudo como pedido no enunciado, incluindo o cancelamento da viagem, adaptado obviamente ao que fizemos acima, mas com a mesma implementação que pede no enunciado onde o passageiro manda mensagem ao servidor, este cancela e comunica ao passageiro e ao condutor que o serviço foi cancelado.