Faculdade de Engenharia da Universidade do Porto

Mestrado em Engenharia Informática e Computação

2º Trabalho Prático de Sistemas Distribuídos

Relatório

Realizado por:

Carolina Centeio Jorge - up201403090

Inês Proença - up201404228

Nuno Miguel Castro - up201406990

Índice

Introdução

No âmbito da disciplina da cadeira de Sistemas Distribuídos, do 3º ano do Mestrado Integrado em Engenharia Informática e Computação, realizámos uma aplicação distribuída baseada em REST (Representational State Transfer) e TCP com servidor próprio. Esta aplicação pretende simular um serviço de um museu/casa de arte/casa interativa e tem a vertente de servidor e cliente. Assim, dependendo de um dado de input do Cliente, como o nome da sala onde se encontra (noutro nível de abstração, a localização propriamente dita), a aplicação descarrega o ficheiro correspondente, que se encontra no servidor. Quando um Cliente dá "like" nalguma sala, os outros recebem a notificação e o sistema atualiza e regista esse like.

Arquitetura

Cliente

O Cliente deve usar o comando “Client <server ip> <username> ”, onde <server ip> é o IP da máquina onde está a correr o Servidor. Depois, o cliente pode digitar o código da sala onde se encontra (ex: “sala1”), “like” na sala onde está e, por último, “sair”, para encerrar a aplicação.

Servidor

O Servidor deve usar o comando “Server <port>”, onde <port> é o porto da ligação HTTP. A partir desse momento, o servidor lê os ficheiros com a informação necessária ao bom funcionamento da aplicação, cria uma ligação HTTP para processar os pedidos do cliente e uma ligação SSL para enviar notificações.

O Servidor comunica ao Cliente, através de um SSLSocket, usando o protocolo TCP, uma notificação com a informação de atividade de outros clientes, ou seja, todos os clientes recebem uma notificação sempre que um dá “like” numa sala.

A ligação HTTP utiliza o protocolo com o mesmo nome para responder aos pedidos do cliente. Este protocolo define vários métodos para fazer o envio da informação, Nós utilizamos o método POST com o ip associado ao Server para o qual o cliente manda os pedidos.

Implementação

Client

A classe Client tem username como id e location sempre atualizada, pois dispõe de um **Locator** a correr como **thread** (Client l. 50), que atualiza a localização assim seja introduzida uma nova. A classe tem uma **Queue** de ações (likes) que o utilizador vai realizando e que precisam de ser reportadas ao servidor e um **Executor** (Client l. 63) de processos necessários à realização de um download. O Cliente inicia pela *main* desta classe. Quando tal acontece, é estabelecida uma ligação **TCP (JSSE)** com o Servidor (Client l. 67), para que o Cliente consiga receber as notificações e atualizações do sistema. Em seguida inicia os **threads Locator** e **MessageCS** (este está responsável por mandar as atualizações, localização e/ou ação, do cliente ao Servidor).

MessageCS

Esta classe é um **Runnable** que a cada segundo, usando **ScheduledExecutorService** (MessageCS l. 60), verifica se há novos "likes" ou nova localização a reportar. Quando verifica que há uma atualização a fazer, cria um **JSONObject** (MessageCS l. 28) e envia um **POST** (MessageCS l. 35 e 40) ao servidor, por **HTTP**.

NewLike

A classe NewLike, também **Runnable** espera as **notificações** a receber no Cliente por parte do Servidor (NewLike l.19). Esta conexão cumpre o protocolo **TCP** com segurança (**JSSE**).

Servidor

Server

O servidor implementado pela classe Server recebe como argumento o porto da ligação **HTTP** e imprime o endereço ip que será usado pelas ligações **HTTP e TCP**. Posteriormente, lê os ficheiros de setup com a informação necessária ao seu correto funcionamento. Finalmente, estabelece as ligações **HTTP** e **SSL** em threads distintas.

A informação necessária concretiza-se no mapeamento da localização com o ficheiro correspondente, nos gostos por cliente e na inicialização de sockets **SSL** para notificar os clientes.

A ligação HTTP é estabelecida com recurso à classe **HTTPServer** e associando uma instância da classe **ServerHandler** ao URI path "/SDIS" que processará os pedidos enviados para o caminho "http://ip:port/SDIS". O processamento de vários pedidos concorrentes é feito recorrendo a um executor do tipo **CachedThreadPool** (Server linha 65), que cria threads à medida que estas são necessárias e reutiliza as que estiverem já disponíveis.

```
46      public static void main(String[] args) throws UnknownHostException {
47          if(args.length < 1){
48              System.out.println("Usage Server <port>");
49              return;
50          }
51
52          setup();
53
54          clients = new ArrayList<SSLSocket>();
55          TCPServer tcpserver = new TCPServer();
56          new Thread(tcpserver).start();
57
58          //HTTP Server
59          final String IP = InetAddress.getLocalHost().getHostAddress();
60          final int PORT = Integer.parseInt(args[0]);
61
62          try {
63              InetSocketAddress inet = new InetSocketAddress(IP, PORT);
64              HttpServer server = HttpServer.create(inet, 0);
65              server.createContext("/SDIS", new ServerHandler());
66              server.setExecutor(Executors.newCachedThreadPool());
67              server.start();
68          } catch (IOException e) {
69              e.printStackTrace();
70          }
71
72      }
```

ServerHandler

Para processar os pedidos **HTTP** utiliza-se uma instância da classe **ServerHandler**, como dito na secção acima. O método handle desta classe espera receber mensagem **POST** no formato **JSON** (ServerHandler l. 33 e 38). As mensagens têm todas um campo obrigatório type que identifica o objetivo da mensagem (informar da localização, buscar chunk do ficheiro ou dar "like"). De acordo com este campo, as mensagens são processadas consultando os ficheiros do servidor e é uma resposta enviada, também no formato **JSON (*library* java-json.jar)**, com o formato e informação necessários (ServerHandler l. 90 e 107)..

TCPServer

Esta classe é um **Runnable** cria um **SSLServerSocket,** usando **JSSE,** para estabelecer ligação **TCP Unidirecional** com o Cliente (TCPServer l. 18 a 58), de forma a comunicar-lhe as novas notificações no **TCPSender**.

TCPSender

.A classe TCPSender é um **Runnable** que recebe um **SSLSocket** e uma mensagem e envia, com PrintWriter para o **SSLSocket**, essa mesma mensagem (TCPSender l. 20).

Assuntos Relevantes

Tolerância a falhas

Esta aplicação possui tolerância a falhas na medida em que, caso o servidor principal falhe, existe sempre um servidor backup com acesso a toda a informação necessária para que a aplicação continue a ser executada sem problemas. Isto acontece caso o cliente não se consiga conectar ao IP do servidor principal, sendo então reencaminhado para o IP do servidor backup.

```
36                  if(mainServerURL == null){
37                          mainServerURL = new URL ("http://"+serverIp+":8000/SDIS");
38                  }
39                  if(backupServerURL == null){
40                          backupServerURL = new URL ("http://"+serverIp+":8080/SDIS");
41                  }
42
43                  HttpURLConnection con = getValidConnection(message, mainServerURL);
44
45                  if(con == null){
46                          //Try backup
47                          System.out.println("Main server down, connecting to backup server");
48                          con = getValidConnection(message, backupServerURL);
49                  }
50
51                  if(con == null){
52                          return new JSONObject("{error: Server not reachable}");
53                  }
54
```

Em termos de memória, esta é guardada em ficheiros do tipo **JSON**. Sempre que há alguma alteração no servidor principal (novo utilizador a utilizar a aplicação ou um novo "like"), essa alteração é guardada em memória **não-volátil** para que, caso o servidor principal falhe, o backup possa continuar com a informação actual.

Escalabilidade

Grande parte da nossa aplicação é escalável na medida em que quase todas as operações do Servidor e do Cliente são feitas em **Threads**, não ficando assim o programa principal bloqueado à espera de finalização destas ações. Em particular, a propagação das notificações, o envio de atualizações do Cliente ao Servidor e a receção de ficheiros (Download e Restore).

```
33                  /*
34                   * Download initiator
35                   *
36                   * If Download protocol chunk is null, then create GetChunk protocol for each of the chunks
37                   * and put it in the queue
38                   */
39                  client.files.put(filename, new HashMap<Integer,Chunk>());
40
41                  for(int i = 0; i < numChunks; i++){
42                          this.client.executor.execute(new GetChunk(client, i, filename, numChunks));
43                  }
44          }
45
46
47  }
```

Este método parte do pressuposto que todos os chunks estão armazenados num **ConcurrentHashMap** chamado *files* em que a chave é o nome do ficheiro e o valor é um **HashMap** com o número do chunk como chave e um objecto **Chunk** como valor. A cada execução desta thread é apagado o ficheiro mais antigo (linha 39) e substituído pelo novo, garantindo assim que o cliente nunca tem ficheiros duplicados. Desta forma, garantindo maior **escalabilidade**.

Segurança

Como referido acima, foi utilizado o protocolo HTTP (Server e Unicast l. 35) para transferência de ficheiros, para que fosse mais segura. Utilizámos, ainda, a ligação SSL em TCP, para garantir autenticação das entidades (TCPServer e Client l.67).

Notificações

É enviada uma notificação a todos Clientes sempre que um dê “like” (Server l. 70), por um canal TCP Unidirecional.

Conclusão

Realizámos o que propusemos, no entanto, gostaríamos de ter tido tempo para melhorar a interface com o utilizador e disponibilizar-lhe os downloads por localização, em vez de introdução de chave de sala.

Encontrámos um problema de conexão com o Servidor quando corrido na consola LINUX. Na consola do IDE Eclipse Neon.3, não houve problema, assim como na consola de Windows.