Guião N.º 1

Sockets TCP (em Java)

Sistemas Distribuídos

Ricardo Costa
rcosta@estgf.ipp.pt

ESTGF | POLITÉCNICO DO PORTO

Outubro 2017

1 Protocolo TCP

O protocolos TCP (*Transmission Control Protocol*) é um dos protocolos utilizados em redes IP para permitir a comunicação entre equipamentos. É considerado como protocolo da camada de Transporte, pois permite a comunicação extremo-a-extremo, ou seja, a comunicação entre dois processos em execução em equipamentos distintos.

O TCP suporta a fiabilidade da entrega da informação, implementando o conceito de ligação. Uma ligação TCP pressupõe então o prévio set estabelecimento (ver Figura 1) e garante a entrega ordenada da informação enviada para a ligação.

2 Sockets

O termo *sockets* é o termo normalmente atribuído às API que permitem a criação e utilização de ligações TCP e de criação, envio e recepção de datagramas UDP. Assim, um *socket* representa um extremo comunicante de uma ligação entre dois programas em execução na rede. Os *sockets* são tradicionalmente utilizados segundo o modelo cliente-servidor, sendo que em Java estão disponíveis classes distintas para os clientes (classe Socket) e para os servidores (classe ServerSocket).

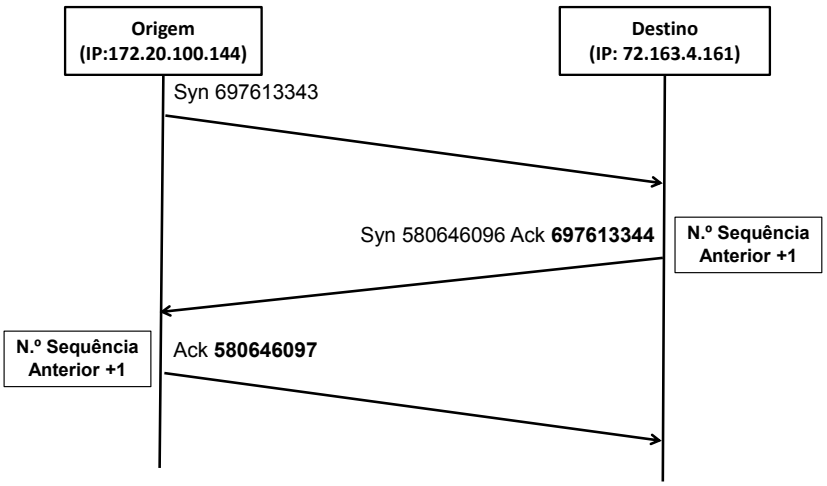

Figura 1: Mensagens trocadas no estabelecimento de uma ligação TCP (*3-way handshake*)

2.1 Exemplo TCP - Serviço *Echo*

O serviço *Echo* é um exemplo tradicional da utilização de ligações TCP. Ao cliente compete o envio de mensagens de texto para o servidor. Já o servidor limita-se a devolver ao cliente as mensagens recebidas deste. O porto tradicional deste serviço é o porto 7.

```
1  import java.io.*;
2  import java.net.*;
3
4  public class EchoClient {
5      public static void main(String[] args) throws IOException {
6
7          Socket echoSocket = null;
8          PrintWriter out = null;
9          BufferedReader in = null;
10
11         try {
12             echoSocket = new Socket("taranis", 7);
13             out = new PrintWriter(echoSocket.getOutputStream(),
                   true);
14             in = new BufferedReader(new InputStreamReader(
15                                     echoSocket.
                                           getInputStream()));
```

```
16          } catch (UnknownHostException e) {
17              System.err.println("Don't know about host: taranis."
                    );
18              System.exit(1);
19          } catch (IOException e) {
20              System.err.println("Couldn't get I/O for "
21                                 + "the connection to: taranis.");
22              System.exit(1);
23          }
24
25    BufferedReader stdIn = new BufferedReader(
26                                       new InputStreamReader(System.
                                           in));
27    String userInput;
28
29    while ((userInput = stdIn.readLine()) != null) {
30        out.println(userInput);
31        System.out.println("echo: " + in.readLine());
32    }
33
34    out.close();
35    in.close();
36    stdIn.close();
37    echoSocket.close();
38    }
39  }
```

Listing 1: Exemplo de um cliente para o serviço *echo*

A Listagem 1 apresenta um exemplo de uma aplicação cliente para o serviço *echo*. É importante notar que a classe apresentada, de nome *Echo-Client*, necessita tanto de enviar como de receber informação. **As linhas 12, 13 e 14 são linhas muito importantes. A linha 12 é usada para se estabelecer a ligação, a linha 13 para se obter um objecto do tipo *PrintWriter* e a linha 14 para se obter um objecto do tipo *BufferedReader*.** A criação da ligação (linha 12) recebe dois parâmetros: "taranis"e 7; sendo o primeiro o endereço do servidor (nome ou endereço IP), e o segundo o porto de destino (Echo usa tradicionalmente o porto 7). O *PrintWriter* será utilizado para enviar informação para o servidor, já o *BufferedReader* será utilizado para obter a informação devolvida pelo servidor.

```
1  ...
2  try {
3      serverSocket = new ServerSocket(7);
4  } catch (IOException e) {
5      System.out.println("Could not listen on port: 7");
6      System.exit(-1);
7  }
8  Socket clientSocket = null;
9  try {
10     clientSocket = serverSocket.accept();
11 } catch (IOException e) {
12     System.out.println("Accept failed!");
13     System.exit(-1);
14 }
15 ...
```

Listing 2: Uso de sockets do lado do servidor

A Listagem 2 apresenta um extracto de código que exemplifica a utilização de *sockets* do lado de um servidor. As diferenças de codificação estão relacionadas com o comportamento esperado por cada interveniente do modelo Cliente-Servidor. Ou seja, ao servidor compete o manter-se activo à espera de pedidos de clientes. **A linha 3 permite a criação do *socket* e a sua associação ao porto aplicacional (no caso o porto 7). Se tudo correr bem, pode-se então iniciar o processamento de pedidos (linha 10). É importante notar que o código que está depois desta linha 10 só será executada quando, e se, surgirem pedidos de clientes. O método *accept* é então bloqueante, já que bloqueia a aplicação até que chegue um pedido de ligação de um cliente, e devolve um *socket* novo que representará a ligação com o cliente que se ligou no momento. Será gerado um novo *socket* para cada cliente/ligação recebida.** De notar também, é o facto de este extracto de código não estar adaptado para o processamento simultâneo de vários clientes.

3 Exercícios

3.1 Serviço Echo

1. Implemente o código de um Servidor Echo capaz de interagir com o cliente demonstrado na Listagem 1. Não se preocupe, nesta fase, com o suporte para múltiplos clientes simultâneos.

2. Adapte o servidor desenvolvido na alínea anterior.
   As mensagens devolvidas pelo servidor devem assumir a seguinte es-

trutura:
IP_SERVIDOR:IPCLIENTE:MENSAGEM

4 Recursos *online*

- The Java Tutorial, Networking
  `http://download.oracle.com/javase/tutorial/networking/`
- The Java Tutorial, Networking, All About Sockets
  `http://download.oracle.com/javase/tutorial/networking/sockets/`