Programação Distribuída – Introdução a Sockets e TCP

Leitura complementar:

- Distributed systems: concepts and design (George Coulouris) cap 3.
- Computer networking: A top-down approach featuring the internet (James F. Kurose).

O objetivo da aula de hoje é iniciar o estudo sobre programação distribuída, mais especificamente sobre sockets TCP. A atividade deverá ser realizada durante o horário da aula, sendo que o prazo para o envio das respostas das questões está especificado na atividade aberta no Moodle. As implementações deverão ser enviadas no formato compactado para o Moodle. Esta esta aula EAD, se corretamente realizada, garante a presenta no dia da aula.

Neste momento, a unidade já está em Programação Distribuída. O objetivo da aula de hoje é iniciar a utilização dos recursos de rede em uma aplicação Java. O primeiro protocolo de transporte a ser utilizado é o TCP, ou seja, aquele que fornece comunicação confiável baseada em conexão.

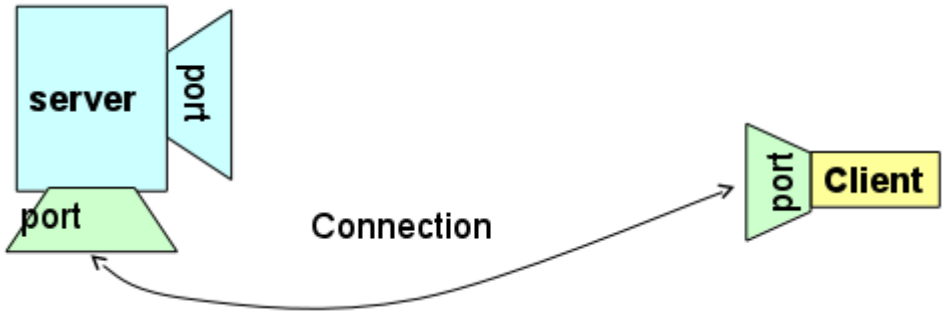
Em Java, bem como em qualquer linguagem de programação, a comunicação básica de rede é feita a partir de sockets. Os sockets fornecem uma interface para utilização das camadas de transporte, sendo que esta comunicação é muito semelhante à execução de E/S em arquivos. Na verdade, o identificador de socket é tratado como identificador de arquivo.

Os fluxos usados na operação de E/S de arquivos (read, write em C e InputStream e OutputStream em Java) também são aplicáveis às E/S baseadas em socket. Um ponto importante é que a comunicação baseada em socket é independente de linguagem de programação. Isso significa que um programa com socket escrito em linguagem Java também pode se comunicar com um programa escrito em C, C++, Python e C#, por exemplo.

O foco da aula de hoje é utilização dos sockets para comunicação TCP. Em uma comunicação TCP, sempre existirão duas entidades: **o servidor** e **o cliente.** O servidor é a entidade passiva que fica aguardando a solicitação de conexão e posterior requisição (em qualquer que seja o protocolo). Esquematicamente o servidor opera “escutando” uma determinada porta:

Se tudo correr bem, o servidor aceita a conexão. Após a aceitação, o **servidor recebe um novo socket** ligado a uma porta possivelmente, mas não necessariamente, diferente. Ele precisa de um novo socket para que ele continue a ouvir o socket original para solicitações de

conexão enquanto serve o cliente conectado. Novamente, têm-se os seguinte cenário, onde existe a comunicação com o cliente e o socket original ainda espera outros clientes:



Sockets em Java

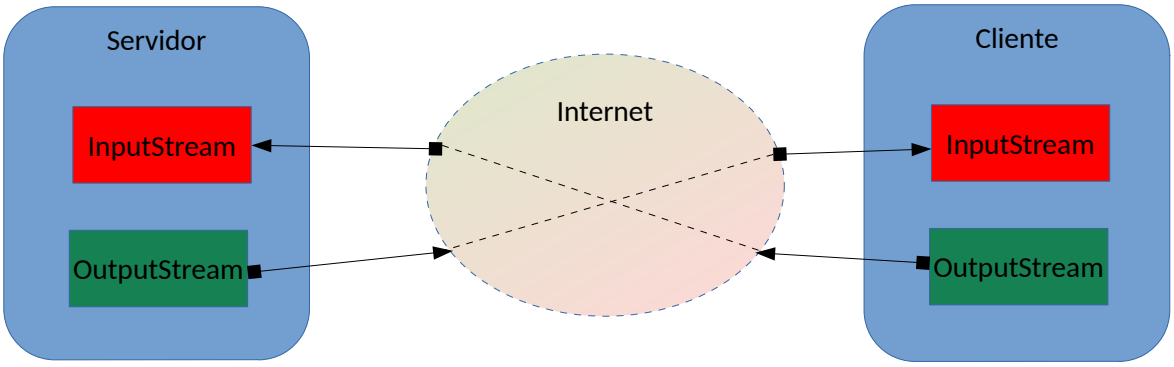
Um socket é um ponto final de um link de comunicação bidirecional entre dois programas em execução na rede. Um socket sempre será vinculado a um número de porta para que a camada TCP/UDP possa identificar o aplicativo que receberá os dados. O pacote .net do Java oferece duas classes:

- **Socket** - para implementar um cliente
- **ServerSocket** - para implementar um servidor

Conforme mencionado, os sockets permitem o estabelecimento de um canal de comunicação (bidirecional) entre o servidor e cliente, sendo este muito similar a E/S em arquivos. Este canal é composto por duas partes:

- **InputStream**: sub canal/descritor/fluxo de recepção de dados.
- **OutputStream**: sub canal/descritor/fluxo de transmissão de dados.

Esquematicamente, quando um cliente e um servidor estão conectados utilizando Sockets TCP, têm-se a seguinte relação entre seus fluxos de entrada e saída (sim, são cruzados como um par RX TX de um cabo de rede):



Como criar um servidor em Java (resumidamente), sendo PORT um valor de porta inteiro:

- Declarando os objetos necessários objetos necessários (passo 1):

```
ServerSocket server;
DataOutputStream os;
DataInputStream is;
```

- Abra o socket e a guarde a solicitação do cliente (passo 2):

```
server = new ServerSocket( PORT );
Socket client = server.accept();
```

- Crie fluxos de E/S para se comunicar com o cliente (passo 3):

```
is = new DataInputStream( client.getInputStream() );
os = new DataOutputStream( client.getOutputStream() );
```

- Execute comunicação com o cliente (passo 4):

```
String line = is.readLine(); // recebendo do cliente
os.writeBytes("Hello\n"); // enviando para cliente
```

- Feche o socket (passo 5): `socket.close();`

Como ficaria um servidor multithread de maneira conceitual (isto será visto mais adiante):

```
while (true) {
     i. aguarde os pedidos dos clientes (passo 2 acima)
     ii. crie uma thread com o soquete "client" como parâmetro (a
          thread cria fluxos (como na etapa (3) e faz a
          comunicação como indicado em (4). Remova a thread
          depois que o serviço for fornecido.
}
```

Agora, como ficaria um cliente?

- Declaração dos objetos necessários (passo 1):

```
Socket client;
DataOutputStream os;
DataInputStream is;
```

- Crie um objeto Socket (passo 2):

```
client = new Socket( server, port_id );
```

- Crie fluxos de E/S para se comunicar com o servidor (passo 3):

```
is = new DataInputStream(client.getInputStream() );
os = new DataOutputStream( client.getOutputStream() );
```

- Execute E/S ou comunicação com o servidor (passo 4):

```
Receba dados do servidor:  String line = is.readLine();
Envie dados para o servidor: os.writeBytes("Hello\n");
```

- Feche o socket quando terminar (passo 5): `client.close();`

Um Exemplo completo:

Considere o seguinte exemplo composto por um cliente e um servidor.

Servidor: o servidor simplesmente envia uma string para o cliente e depois finaliza sua execução. Note que este servidor não é multithread. Veja que o servidor escuta a porta 1234.

```
// SimpleServer.java: a simple server program
import java.net.*;
import java.io.*;
public class SimpleServer {
  public static void main(String args[]) throws IOException {
    // Register service on port 1234
    ServerSocket s = new ServerSocket(1234);
    Socket s1=s.accept(); // Wait and accept a connection
    // Get a communication stream associated with the socket
    OutputStream s1out = s1.getOutputStream();
    DataOutputStream dos = new DataOutputStream (s1out);
    // Send a string!
    dos.writeUTF("Hi there");
    // Close the connection, but not the server socket
    dos.close();
    s1out.close();
    s1.close();
  }
}
```

Cliente: o cliente recebe uma string do servidor, imprime na tela e depois finaliza sua execução. O cliente conecta na porta 1234 do servidor em *localhost*. Poderia ser outro IP aqui.

```
// SimpleClient.java: a simple client program
import java.net.*;
import java.io.*;
public class SimpleClient {
  public static void main(String args[]) throws IOException {
    // Open your connection to a server, at port 1234
    Socket s1 = new Socket("localhost",1234);
    // Get an input file handle from the socket and read the input
    InputStream s1In = s1.getInputStream();
    DataInputStream dis = new DataInputStream(s1In);
    String st = new String (dis.readUTF());
```

```
    System.out.println(st);
    // When done, just close the connection and exit
    dis.close();
    s1In.close();
    s1.close();
  }
}
```

Para executar:

1. Crie um projeto Java na sua IDE.
2. Inclua os arquivos como especificado na página anterior.
3. Execute o servidor[1] e depois o cliente e veja o resultado[2].

Exercício para entregar:

Considerando os exemplos fornecidos, implemente uma aplicação cliente e servidor em que:

- Servidor recebe uma String, a inverte e envia de volta.
- O cliente lê uma String do teclado, envia para o servidor, recebe uma resposta e a imprime na tela.

1 Se uma instância do servidor estiver em execução, outra não poderá ser executada na mesma porta. Antes de executar o servidor, então, verifique se não há nada sendo executado. Uma exceção "java.net.BindException: Address already in use: Cannot bind." é um sinal claro deste tipo de problema.
2 O firewall do sistema pode causar problemas com a utilização de sockets, fique atento!