Sockets em Java

Leonardo R. Nunes - leonardo@sumersoft.com

1. Introdução

A comunicação entre processos de software tornou-se indispensável nos sistemas atuais. O mecanismo mais utilizado atualmente para possibilitar comunicação entre aplicações é chamado socket. Neste artigo será apresentado o suporte que Java oferece para a utilização desse mecanismo de comunicação.

Java oferece os seguintes modos de utilização de sockets: o modo orientado a conexão, que funciona sobre o protocolo TCP (*Transmission Control Protocol*, ou protocolo de controle de transmissão), e o modo orientado a datagrama, que funciona sobre o protocolo UDP (*User Datagram Protocol*, ou protocolo de datagrama de usuários). Os dois modos funcionam sobre o protocolo IP (*Internet Protocol*).

Cada um desses modos tem sua aplicabilidade, e possuem vantagens e desvantagens em sua utilização.

Modo orientado a conexão TCP/IP	Modo orientado a datagrama UDP/IP

• Serviços confiáveis:
  o Sem perdas de dados na rede;
  o Garantia de ordem dos pacotes;
• Possibilidade de utilização de fluxo de dados (*DataStreams);*
 	
Desvantagens:
• É mais lento que o modo orientado a datagrama;
• Comportamento servidor diferente de comportamento cliente;
	• Serviços não confiáveis:
	  o Mensagens podem ser perdidas;
	  o Ordem das mensagens não é garantida;
	• Cada mensagem é um datagrama: [sender (remetente), receiver (receptor), contents (conteúdo da mensagem)]
	
	Vantagem:
	• É muito mais rápido que o modo orientado a conexão;

2. Sockets TCP/IP

O processo de comunicação no modo orientado à conexão ocorre da seguinte forma: O servidor escolhe uma determinada porta (o termo correto seria porto, em inglês *port*, mas aqui no Brasil o termo utilizado comumente é porta) e fica aguardando conexões nesta porta. O cliente deve saber previamente qual a máquina servidora (host) e a porta que o servidor está aguardando conexões. Então o cliente solicita uma conexão em um host/porta, conforme demonstrado na figura 1.

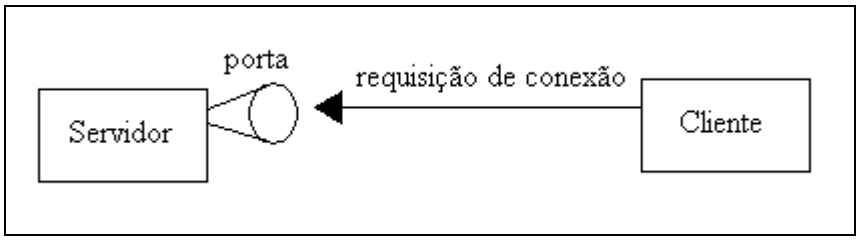

Figura 1

Se nenhum problema ocorrer, o servidor aceita a conexão gerando um socket em uma porta qualquer do lado servidor, criando assim um canal de comunicação entre o cliente e o servidor. A figura 2 demonstra este canal de comunicação.

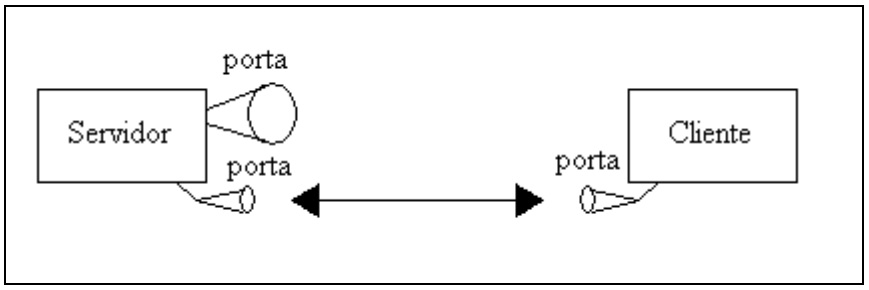

Figura 2

Tipicamente o comportamento do servidor é ficar em um *loop* aguardando novas conexões e gerando sockets para atender as solicitações de clientes.

A seguir serão apresentadas as ações necessárias para implementar comunicação sobre TCP através de um socket cliente e um socket servidor:

2.1. Criando um socket cliente:

- Passo 1 - Abrir a conexão:

```
import java.io.* ;   // streams
import java.net.* ;  // sockets
//Conectar no servidor  java.sun.com na porta 80.
Socket client = new Socket("java.sun.com", 80);
```

- Passo 2 - Obter fluxos (*streams*) de entrada e saída para comunicação com o servidor:

```
//Cria um canal para receber dados.
DataInputStream in=new DataInputStream(client.getInputStream());

//Cria um canal para enviar dados.
DataOutputStream out=new DataOutputStream(client.getOutputStream());
```

- Passo 3 - Realizar comunicação com o servidor:

```
out.writeInt( 3 ); //Envia o inteiro 3.
out.writeUTF( “Hello” ); //Envia a string “Hello”.

int k = in.readInt(); //Aguarda o recebimento de um inteiro.
String s = in.readUTF(); //Aguarda o recebimento de uma string.
```

- Passo 4 - Fechar fluxos (*streams*) e conexão:

```
//Fecha os canais de entrada e saída.
in.close();
out.close();
//Fecha o socket.
client.close();
```

2.2. Criando um socket servidor:

- Passo 1 - Criar o socket servidor:

```
//Cria um socket servidor na porta 80
ServerSocket serverSocket=new ServerSocket(80);
```

- Passo 2 - Aguardar novas conexões:

```
// O metodo accept retorna um socket para comunicação com o próximo
//cliente a conectar.
// A execução do método bloqueia até que algum cliente conecte no servidor.
Socket socket = serverSocket.accept();
```

- Passo 3 - Obter fluxos (*streams*) de entrada e saída para comunicação com o cliente:

```
//Cria um canal para receber dados.
DataInputStream in=new DataInputStream(socket.getInputStream());
//Cria um canal para enviar dados.
DataOutputStream out=new DataOutputStream(socket.getOutputStream());
```

- Passo 4 - Realizar comunicação com o servidor:

```
int k = in.readInt(); //Aguarda o recebimento de um int.
String s = in.readUTF() ; //Aguarda o recebimento de uma string.

out.writeInt(3); //Envia um int.
out.writeUTF("Hello"); //Envia uma string.
```

- Passo 5 - Fechar fluxos (*streams*) e socket cliente:

```
//Fecha os canais in e out do socket que está atendendo o cliente
in.close();
out.close();
//Fecha o socket que está atendendo o cliente.
socket.close();
```

- Passo 6 - Fechar o socket servidor:

```
//Fechando o servidor.
serverSocket.close();
```

A utilização do modo orientado a conexão possibilita algumas funcionalidades interessantes como a utilização de canais unidirecionais, que podem ser obtidos através dos métodos *socket.shutdownInput()* ou *socket.shutdownOutput()*.

Java provê também algumas implementações de alto nível para sockets TCP/IP, como por exemplo, um conector para o protocolo HTTP (`java.net.HttpURLConnection`).

3. Sockets UDP/IP

Sockets UDP/IP são muito mais rápidos que sockets TCP/IP. São mais simples, porém menos confiáveis. Em UDP não temos o estabelecimento de conexão, sendo que a comunicação ocorre apenas com o envio de mensagens.

Uma mensagem é um datagrama, que é composto de um remetente (*sender)*, um destinatário ou receptor (*receiver*), e a mensagem (*content*). Em UDP, caso o destinatário não esteja aguardando uma mensagem, ela é perdida. A figura 3 apresenta o envio de um datagrama de uma suposta máquina (Maq1) para outra (Maq2) em uma rede.

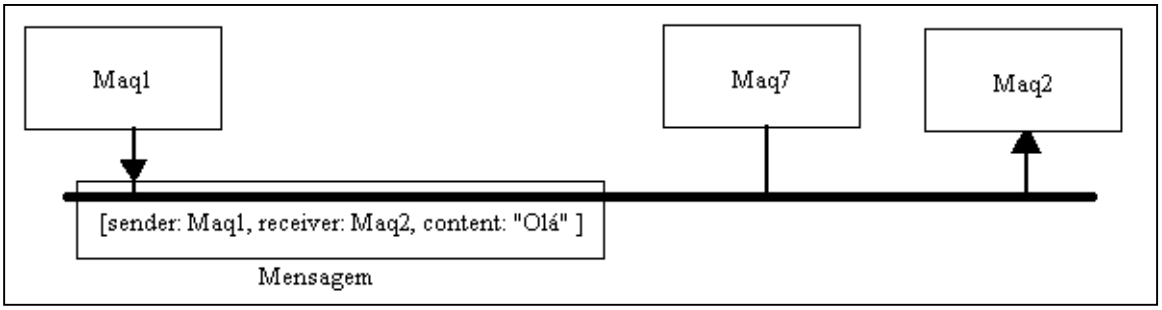

Figura 3

A seguir serão apresentadas as ações necessárias para implementar comunicação utilizando sockets UDP:

3.1. Enviando uma mensagem:

- Passo 1 - Criar o socket:

```
//sender socket não precisa de uma porta em especial.
DatagramSocket clientSocket=new DatagramSocket();
```

- Passo2 - Criando e enviando o datagrama:

```
InetAddress addr=InetAddress.getByName("www.javasoft.com");

String toSend ="PERGUNTA";
byte[] buffer = toSend.getBytes();

//Enviar datagrama para destinatário na porta 4545.
DatagramPacket question = new DatagramPacket(buffer, buffer.length, addr, 4545);
//Envia o datagrama.
clientSocket.send(question);
```

- Passo 3 - Recebendo e abrindo uma resposta:

```
//Passa um buffer e o tamanho do buffer para receber a mensagem.
//Caso o conteúdo da mensagem recebida for maior que o buffer
// o excedente é perdido.
DatagramPacket answer=new DatagramPacket(new byte[512], 512);

clientSocket.receive(answer);
```

- Passo 4 - Fechando o socket:

```
//Fechando o socket.
clientSocket.close();
```

3.2. Recebendo mensagens:

- Passo 1 - Criando um socket servidor:

```
//Escutando na porta 4545.
DatagramSocket serverSocket=new DatagramSocket(4545);
```

- Passo 2 - Recebendo um datagrama:

```
//Cria o datagrama para receber uma mensagem.
DatagramPacket question=new DatagramPacket(new byte[512], 512);
//Aguarda até o recebimento de uma mensagem.
serverSocket.receive(question);
```

- Passo 3 - Enviando uma resposta:

```
String toSend="RESPOSTA";
byte[] buffer = toSend.getBytes();

DatagramPacket answer = new DatagramPacket (buffer, buffer.length,
 question.getAddress()/*sender info*/, question.getPort()/*sender info*/);

serverSocket.send(answer);
```

- Passo 4 - Fechando o servidor:

```
//Fechando o servidor.
serverSocket.close();
```

3.3. Multicast

O protocolo UDP suporta o envio de uma mensagem para um grupo de destinatários ao invés de um único destinatário. Isto é denominado *multicast*. Um grupo *multicast* é especificado por um endereço IP de classe "D" (224.0.0.1 até 239.255.255.255, inclusive) e uma porta UDP. Classes IP definem *ranges* de endereços. O endereço 224.0.0.0 é reservado e não deve ser utilizado. Em Java o suporte a *multicast* é oferecido através da classe `java.net.MulticastSocket`.

A seguir temos as ações necessárias para a utilização de *multicast* com sockets UDP.

```
InetAddress group = InetAddress.getByName("228.5.6.7");
MulticastSocket s = new MulticastSocket(6789);

// Entra no grupo. A partir deste momento as mensagens
//para 228.5.6.7 serao recebidas em s.
s.joinGroup(group);

// Envia e recebe mensagens UDP conforme apresentado anteriormente...

// Retira-se do grupo. Mensagens para 228.5.6.7
//não mais chegarão até o socket s.
s.leaveGroup(group);
```

4. New I/O API

O novo pacote de I/O (`java.nio`), introduzido na versão J2SE 1.4 traz novas funcionalidades e avanços significativos de desempenho em I/O em JAVA. Dentre as diversas funcionalidades temos uma nova abstração para I/O que são os canais (*channels*). Um canal é uma abstração que representa uma conexão entre entidades que fazem operações de I/O. No caso de sockets temos as classes `java.nio.channels.ServerSocketChannel` e `java.nio.channels.SocketChannel`. A utilização de canais possibilita uma forma mais simples para realizar comunicação, pois basta abrir o canal, escrever e/ou ler, e ao final da execução, fechar o canal. As implementações destas classes de canais de socket utilizam as classes `java.net.ServerSocket` e `java.net.Socket`, já vistas.

Ao final deste artigo será apresentado um exemplo de utilização de canais para acessar um servidor www.

5. Conclusão

Este artigo apresentou as principais formas de *networking* oferecidas por Java, porém o suporte a *networking* oferecido por Java é mais abrangente. Existem algumas configurações que podem ser realizadas na utilização de IP, como a configuração de *timeouts* (tempo de espera máximo para realização de operações), *buffers* (tamanho do buffer dos pacotes), *keep alive* (mensagem automática para verificação de disponibilidade dos pares em uma conexão quando a conexão não está em utilização), além de outras configurações. Estas configurações podem ser verificadas na interface `java.net.SocketOptions`.

Para os entusiastas em *networking*, temos algumas outras funcionalidades do J2SE 1.4 que não mencionamos, porém são muito interessantes. Entre elas podemos destacar o suporte à IPv6, suporte à operações assíncronas, suporte à *Secure Socket Layer* (permite socket seguro), além de outras novidades. Maiores detalhes podem ser encontrados em:

http://java.sun.com/j2se/1.4.2/docs/guide/net/enhancements14.html

Exemplo de conexão em um servidor www utilizando Channels:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.charset.Charset;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class Test {

	static public void main(String[] args) {

		Charset charset=Charset.forName("UTF-8");
		SocketChannel channel=null;

		try {
			InetSocketAddress socketAddr=
				new InetSocketAddress("www.sumersoft.com", 80);
			//Abre a conexão
			channel=SocketChannel.open(socketAddr);
			//Envia dados
			channel.write(charset.encode("GET /index.html\r\n\r\n"));
			ByteBuffer buffer=ByteBuffer.allocate(2048);
			//Enquanto houver dados pega do buffer e imprime.
			while((channel.read(buffer))!=-1) {
				buffer.flip();
				System.err.println(charset.decode(buffer));
				buffer.clear();
			}
		} catch(IOException e) {
			e.printStackTrace();
		} finally {
			if(channel!=null) {
				try {
					channel.close();
				} catch(IOException e) {
					e.printStackTrace();
				}
			}
		}
	}
}
```

Autor:

Leonardo R. Nunes - leonardo@sumersoft.com - Formado em Ciência da Computação pela PUCPR, mestrando em Sistemas Distribuídos na PUCPR, trabalha com desenvolvimento de software orientado a objetos utilizando Java desde 1997. Desenvolveu aplicações Java para a área de telefonia, área industrial e de telecomunicações. Atualmente é Diretor da Sumersoft Tecnologia (http://www.sumersoft.com) e Coordenador do PRoJAVA (http://www.projava.com.br).