Sockets

Sistemas de informação – Univas

Roberto Ribeiro Rocha

rrocha.roberto@gmail.com

Março de 2015

Sumário

1 Objetivos

Este material provê informações básicas para o aluno:

- entender os conceitos fundamentais de comunicação de computadores;
- entender *sockets* e portas;
- entender algumas especificações do pacote java.net;
- programar usando Java *Sockets*;
- criar aplicações em rede usando *sockets*;
- utilizar recursos multi-*thread*.

2 Introdução

Este material mostra os conceitos chave da comunicação entre processos executados em diferentes máquinas na rede. Também é introduzido os elementos de programação em rede e conceitos de *sockets*, usando como aplicação prática, algumas classes do pacote java.net necessárias para criar *sockets* e fazer a comunicação entre os programas que se comunicam pela rede.

3 *Sockets*

A internet é importante meio de comunicação e mudou o modo de evolução do conhecimento e tecnologias. A vida atual está cada vez mais dependente da internet para efetuar suas atividades.

Para tirar proveito das oportunidades apresentadas pela internet, as empresas estão em constante busca de novos meios para oferecer seus serviços. Isto criou uma grande demanda por software que se adequasse neste ambiente, disponibilizando aplicações e portando sistemas legados para a internet. Um elemento chave para desenvolver aplicações em rede é o bom entendimento envolvido em implementar sistemas distribuídos e conhecimentos dos modelos fundamentais de *network programming*.

Definição: *Socket* é uma extremidade de um *link* de comunicação bidirecional na rede, utilizada por dois programas em execução. Ele é vinculado a um número de porta, tal que a camada TCP possa identificar a aplicação associada a cada canal de comunicação.

O *socket* é criado e usado por um programa ou mais programas de acordo com as necessidades do programador, permitindo que o mesmo envie e receba bytes através da conexão.

Além de IP e porta, o *socket* também possui o protocolo atrelado, que pode ser TCP ou UDP.

3.1 *Sockets* TCP

Os *sockets* TCP permitem a comunicação entre aplicações através de uma conexão. Suas principais características são:

- orientado a conexão: é necessário estabelecer uma conexão entre o emissor e receptor antes de enviar um pacote TCP;
- confiável: o pacote é retransmitido em caso de perdas;
- entrega os pacotes em ordem;
- provê um canal bidirecional;
- possui controle de fluxo;
- é mais complexo e consome mais recursos computacionais.

Em um servidor, normalmente existe um *socket* vinculado a um número de porta específica. O servidor somente espera, escutando o *socket*, por um cliente fazer um pedido de conexão.

No Cliente, deve ser conhecido o IP e a porta que o servidor está escutando. O cliente também usa uma porta de conexão vinculando também seu *socket* a uma porta local, que será usada na conexão (usualmente atribuída pelo sistema operacional).

A Figura 1 ilustra o processo de pedido e estabelecimento da conexão via *socket* entre um cliente e um servidor:

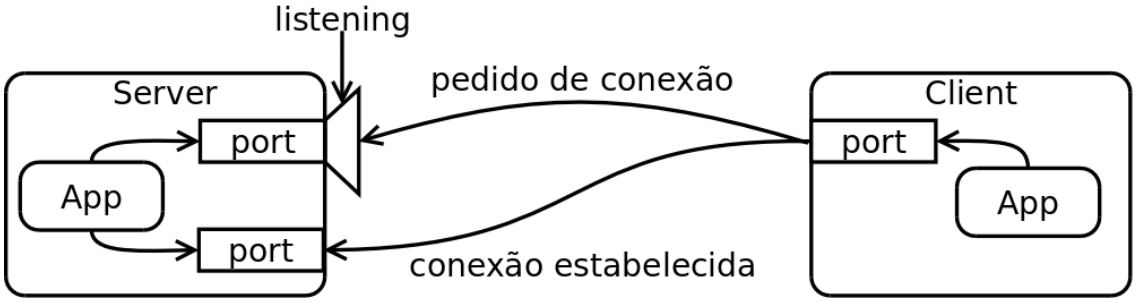

Figura 1: Pedido e estabelecimento da conexão através do *socket*

Em condições normais, após o cliente fazer o pedido de conexão, o servidor aceita a conexão, criando um novo *socket* e vinculando-o com o *socket* do cliente. Assim, o *socket* original fica livre para escutar novos pedidos de conexão, enquanto o servidor comunica com o cliente. Para cada cliente que pede uma nova conexão, é utilizada uma porta para a comunicação.

Do lado do cliente, se a conexão for aceita, um *socket* é criado com sucesso e o cliente pode utilizá-lo. Assim ambos podem ser comunicar escrevendo e lendo do *socket*.

3.2 Operações com *Sockets* TCP

Para trabalhar com *sockets*, existem funções prontas, chamadas de *system calls* disponibilizadas pelo sistema operacional, onde é necessário utilizar para se trabalhar com *sockets*. As principais operações são descritas a seguir:

- ***create***: o sistema operacional cria o *socket* e as informações necessárias para a interação da aplicação com o *socket*;
- ***bind***: utilizado no lado do servidor para associar o *socket* com as estruturas internas de IP e porta;
- ***listen***: utilizado no lado do servidor para deixar o *socket* em estado de *listening* (escutando por uma conexão);
- ***accept***: utilizado no lado do servidor para aceitar uma conexão feita por um cliente;
- ***send/write***: envia dados para o *socket* remoto;
- ***receive/read***: recebe dados de um *socket* remoto;
- ***close***: fecha o *socket*.

Logo, os passos envolvidos na interação entre cliente e servidor, através destas operações, podem ser vistos na Figura 2 e possuem algumas dependências, onde o servidor deve ser iniciado antes que o cliente.

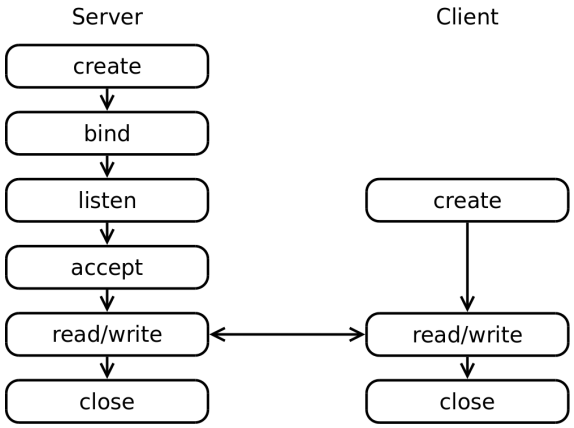

Figura 2: Uso das operações do *socket*

Dependendo da linguagem de programação utilizada as diretivas `create` e `bind` estão encapsuladas e são executados através de uma única chamada de método ou função.

Através destes comandos, um processo servidor poderia ter os seguintes passos:

Algorithm 1: Algoritmo do servidor utilizando *socket* TCP

```
ss ← create socket TCP (port);
listen ss;
sc ← accept from ss;
read or write to sc;
close sc;
```

E um processo cliente poderia ter os seguintes passos:

Algorithm 2: Algoritmo do cliente utilizando *socket* TCP

```
sc ← create socket TCP (IP, port);
read or write to sc;
close sc;
```

3.3 Diretivas bloqueantes

Diretivas bloqueantes são aquelas onde o processo de execução de um programa fica bloqueado, aguardando ocorrer algum evento que ele esteja esperando.

3.4 *Sockets* UDP

Os *sockets* UDP não necessitam de estabelecer uma conexão entre o cliente e o servidor, para efetivar a comunicação. Basta enviar para a rede um datagrama com as informações desejadas. Isto o torna mais simples e rápido, porém não confiável.

Como não há a necessidade da conexão, o processo de envio e recebimento torna o código, tanto do cliente quanto do servidor, mais simplificado, mostrados nos algoritmos a seguir:

Algorithm 3: Algoritmo do servidor utilizando *socket* UDP

```
ss ← create socket UDP (port);
read or write to ss datagram(IP, port);
close ss;
```

E o cliente ficaria idêntico à execução TCP:

Algorithm 4: Algoritmo do cliente utilizando *socket* UDP

```
sc ← create socket UDP;
read or write to sc datagram(IP, port);
close sc;
```

3.5 Multicast

Um processo envia uma mensagem (um pacote UDP/IP) para um grupo de processos.

Permite enviar um único pacote IP para um conjunto de processos denominado grupo de multicast.

O multicast é suportado pelo UDP.

Um grupo multicast é especificado por algum endereço de IP da classe D, e por uma porta padrão UDP (ex: 6789).

Aplicações:

- Difusões de áudio e vídeo
- Sistemas Distribuídos/UTFPR Prof. Cesar Augusto Tacla
- Replicação de serviços
- Localização de serviços em redes espontâneas
- Replicação de operações/dados
- Difusão de eventos

Ambos, o cliente e o servidor, devem entrar no grupo, para receber e enviar mensagens.

Exemplo:

```
MulticastSocket ms = new MulticastSocket(porta);
//entra no grupo
ms.joinGroup(ipGrupo);
//...
//cria datagrama
ms.send(datagrama);
//sai do grupo
ms.leaveGroup(ipGrupo);
```

3.6 Broadcast

Envia datagramas (UDP) para todos os clientes em uma mesma rede física, pois os roteadores e *gateways* não propagam estes pacotes.

O endereço de broadcast é o mesmo definido pela rede, ou o 255.255.255.255. Ele depende da máscara de rede de seu *host*.

Principal aplicação: notificar a todos a ocorrência de um evento.

4 Java *Sockets*

A linguagem Java provê um conjunto de classes definidas no pacote `java.net` para permitir o desenvolvimento rápido de aplicações em rede.

4.1 Java *Sockets* através de TCP

As principais classes e interfaces neste pacote são: `InetAddress`, `ServerSocket`, `Socket` e `SocketException`.

A leitura e escrita dos *bytes* são feitas através dos objetos `InputStream` e `OutputStream`.

Com todas estas classes é possível escrever o código Java do servidor da seguinte forma:

```
try {
  System.out.println("Iniciando servidor.");
  int port = 3134;
  byte[] dados = new byte[10]; // buffer de leitura de tamanho fixo
  StringBuffer buffer = new StringBuffer(); //buffer geral

  ServerSocket server = new ServerSocket(port); //cria um socket para ficar escutando

  while (true) {
    System.out.println("Aceitando a conexao.");
    Socket sock = server.accept();

    System.out.println("Lendo os dados.");
    InputStream in = sock.getInputStream(); //le os dados a partir do inputStream

    int qtd = in.read(dados); // le 10 bytes
    while (qtd > 0) {
      buffer.append(new String(dados, 0, qtd));
      qtd = in.available(); // verifica se existe dados para ler
      if(qtd > 0) { //isto evita ficar preso no read bloqueante
        qtd = in.read(dados); // le 10 bytes
      }
    }
    System.out.println("Dados recebidos: " + buffer.toString());

    // processamento qualquer do pedido
    String resposta = "Ola " + buffer.toString();

    OutputStream out = sock.getOutputStream();

    System.out.println("Enviando a resposta: " + resposta);
    out.write(resposta.getBytes()); // envia dados através do outputStream

    System.out.println("Fechando a conexao.");
    sock.close();
  }
  // server.close();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

O código do cliente fica da seguinte forma:

```
System.out.println("Iniciando o cliente.");
int port = 3134;
Socket sock = new Socket("localhost", port); //cria um socket

String mensagem = "Roberto";
```

```
OutputStream out = sock.getOutputStream();
System.out.println("Enviando dados.");
out.write(mensagem.getBytes()); //envia dados
out.flush();

//buffer de leitura de tamanho fixo
byte [] dados = new byte[10];

StringBuffer buffer = new StringBuffer();
//recebe e imprime os dados
InputStream in = sock.getInputStream();
int qtd = in.read(dados); // le 10 bytes
while (qtd > 0) {
  buffer.append(new String(dados, 0, qtd));
  qtd = in.available(); // verifica se existe dados para ler
  if(qtd > 0) {
    qtd = in.read(dados); // le 10 bytes
  }
}
System.out.println("Resposta do servidor: " + buffer.toString());
sock.close();
```

Observação: estes códigos necessitam de tratamento adequado das exceções.

Prática 1 – Criar duas classes, cliente e servidor, e utilizar os dois códigos anteriosres para fazer o teste de envio e recebimento de bytes via *socket*.

Exercício 1 – Implementar um servidor de calculadora usando *socket* TCP, que fornece os resultados para as quatro operações básicas.

O servidor deve fornecer as operações seguindo a seguinte interface:

```
public interface CalculadoraService {
   public int add(int v1, int v2);
   public int sub(int v1, int v2);
   public int div(int v1, int v2);
   public int mul(int v1, int v2);
}
```

Juntamente com os números, é necessário enviar também o operador. Para padronizar a comunicação, deve-se utilizar uma `String` com um formato padrão, por exemplo: `"operador:valor1:valor2"`. Assim o servidor sabe identificar as informações recebidas do cliente e fazer a chamada correspondente da operação desejada.

Para facilitar a implementação, deve-se fazer a conversão de `int` para `String` e vice-versa para transmitir/receber os dados via *socket*.

4.2 Enviando e recebendo objetos Java

A API do Java permite se sejam enviados e recebidos quaisquer objetos através das classes `ObjetcOutputStream` e `ObjetcInputStream`. Este recurso facilita muito a transferência de informações através do *socket*. Porém, para que os objetos sejam

tratados corretamente, é necessário que as classes destes objetos implementem a interface `Serializable`.

Logo os códigos de envio e recebimento, principalmente de recebimento fica bem simplificado, conforme mostrado no fragmento de código:

```
Socket sock = new Socket("localhost", port); //cria um socket

String mensagem = "Roberto";

//envia o objeto via socket
ObjectOutputStream out =
        new ObjectOutputStream(sock.getOutputStream());
out.writeObject(mensagem);

//recebe a resposta do socket
ObjectInputStream in =
        new ObjectInputStream(sock.getInputStream());
String resp = (String) in.readObject();

sock.close();
```

Repare que os objetos `out` e `in` trabalham somente com objetos, logo na linha 13, é necessário fazer um *cast* para `String` para obter a resposta.

A leitura dos dados do teclado pode ser feita através da classe `Scanner`. Assim o código do cliente teria a seguinte forma:

```
...
Scanner sc = new Scanner(System.in);
msg = sc.next() + sc.nextLine();//le um texto do teclado
while (!msg.equals("exit")) {
  //envia a msg para o socket via ObjectOutputStream
  out.writeObject(msg);

  //recebe a resposta do socket via ObjectInputStream
  String resp = (String) in.readObject();
  if (resp != null) {
    System.out.println("Resposta: " + resp);
  }

  //le o proximo texto do teclado
  msg = sc.next() + sc.nextLine();
}
sock.close();
```

Uma outra forma de enviar a `String` lida do teclado é utilizar a classe `Scanner` em conjunto com a classe `PrintStream`, tornando o código bem simples, da seguinte forma:

```
PrintStream saidaSocket = new PrintStream(sock.getOutputStream());
Scanner stdin = new Scanner(System.in);
while(stdin.hasNextLine()) {
  saidaSocket.println(stdin.nextLine());
}
```

Para encapsular o envio e recebimento de objetos, pode-se criar uma classe específica para isto, que neste material será chamada de `IOHelper`, e seu código poderia ter a seguinte forma:

```
import java.io.*;
import java.net.Socket;

public class IOHelper {
  ObjectInputStream in = null;
  ObjectOutputStream out = null;

  public IOHelper(Socket sock) throws IOException {
    in = new ObjectInputStream(sock.getInputStream());
    out = new ObjectOutputStream(sock.getOutputStream());
  }

  public void send(Object objetc) throws IOException {
    out.writeObject(objetc);
  }

  public Object receive() throws IOException, ClassNotFoundException {
    return in.readObject();
  }
}
```

Prática 2 – Criar um código cliente que leia do teclado as mensagens e envie seus objetos para o servidor. Quando o cliente digitar a palavra "exit", ele deve sair do *loop* e fechar a conexão. Altere o código original das Listagens 1 e 2 para que eles utilizem o envio e recebimento de objetos. Utilize a classe `IOHelper` para simplificar o código do cliente e do servidor.

4.3 Tratando várias conexões simultâneas

Os exemplos mostrados até aqui são ineficientes, onde o servidor aceita e trata somente uma conexão de cada vez. Assim é necessário utilizar os recursos das linguagens de programação para aceitar uma conexão e tratá-la de forma independente, fazendo com que o servidor fique livre para executar outras tarefas.

Para isto, pode-se utilizar o recurso de *threads*, para enviar e receber os dados de um cliente, deixando o *main* livre para aceitar novas conexões, fazendo com que a cada nova conexão aceita, o programa cria uma nova *thread*.

Logo, é necessário criar uma estrutura de classes para suportar este novo requisito, conforme a Figura 3.

A *thread* para processar a conexão, representada pela classe `ProcessaConexao` (utilizada pela classe `ServidorSocket`), é facilmente criada e iniciada utilizando o seguinte código:

```
...// apos aceitar a conexao
// cria e executa a thread para processar a conexao recebida
ProcessaConexao pc = new ProcessaConexao(sock);
Thread th = new Thread(pc);
th.start();
```

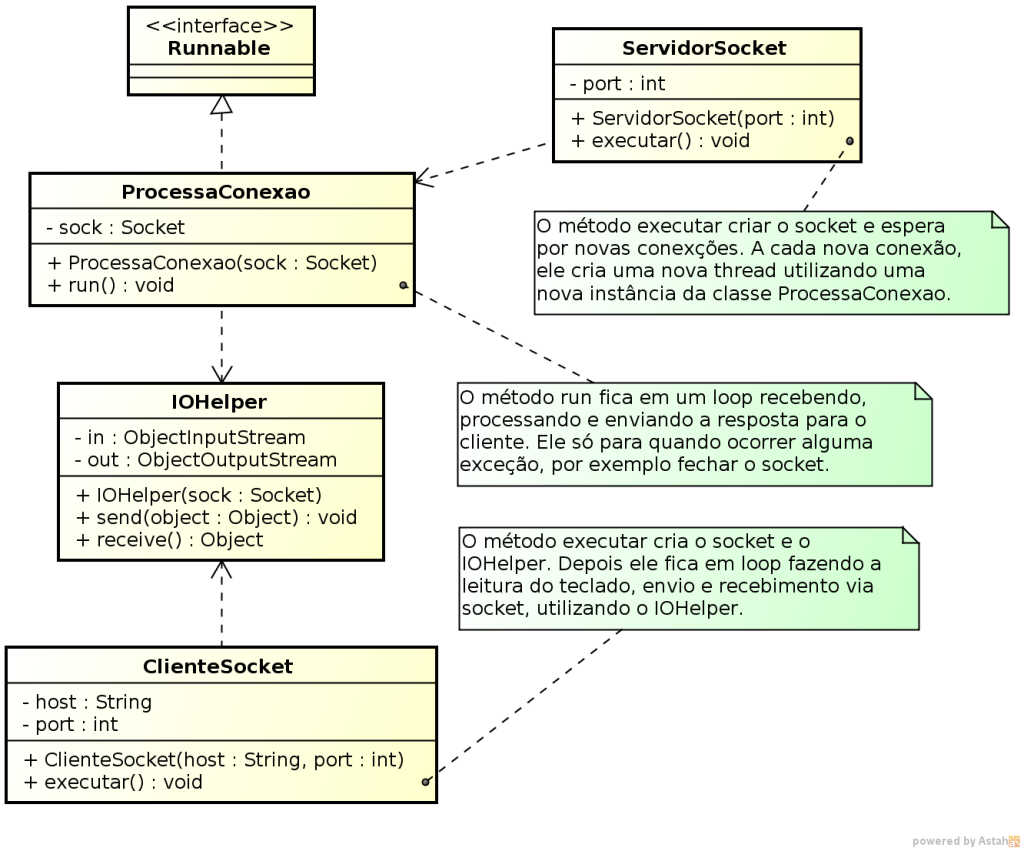

Figura 3: Estrutura de classes para o servidor aceitar várias conexões.

Prática 3 – Implementar o código ilustrado pela Figura 3, utilizando os recursos vistos até este momento para enviar e receber objetos `String` e fazendo o servidor aceitar várias conexões "simultâneas". Para testar o código, peça para que vários colegas executem o cliente conectando em um mesmo servidor e enviando várias mensagens. Esta atividade pode ser feita em grupo.

4.4 Criando um chat utilizando *sockets* TCP

Ver arquivo da prática 4 na página da disciplina.

4.5 Java *Sockets* através de UDP

Para a implementação da comunicação via *sockets* UDP, é necessário utilizar as seguintes classes: `DatagramPacket`, `DatagramSocket` e `InetAddress`.

Assim o código do servidor UDP ficaria da seguinte maneira:

```
byte [] buf = new byte[20];
//cria um socket UDP
DatagramSocket socket = new DatagramSocket(5000);

DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet); //le os bytes do socket

String msg = new String(packet.getData()); //obtem a mensagem
System.out.println("Mensagem recebida: " + msg);
msg = "Ola " + msg.toUpperCase(); //processa a mensagem

InetAddress addrClient = packet.getAddress(); //obtem as informacoes do cliente

//cria o pacote de resposta
DatagramPacket packToSend = new DatagramPacket(msg.getBytes(), msg.length(),
    addrClient, packet.getPort());
socket.send(packToSend); //envia o pacote
```

Para obter as informações de conexão do cliente, pode-se utilizar a classe `InetAddress`:

```
InetAddress addrClient = packet.getAddress();
System.out.println("IP do cliente: " + addrClient.getHostName());
```

E o cliente teria o seguinte código:

```
DatagramSocket socket = new DatagramSocket();
InetAddress address = InetAddress.getByName("labf0207");

String msg = "Roberto";
DatagramPacket packToSend = new DatagramPacket(msg.getBytes(), msg.length(), address,
    5000);
socket.send(packToSend);

byte [] dados = new byte[20];
DatagramPacket packRecv = new DatagramPacket(dados, dados.length);
socket.receive(packRecv);

System.out.println("Mensagem recebida: " + new String(dados));
```

Observação: estes códigos necessitam de tratamento adequado das exceções.

5 Bibliografia

- COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. Distributed Systems - Concepts and Design. 2. ed., 1994
- TANENBAUM, A. S. Distributed Operating Systems. Prentice Hall, 1995.
- `http://docs.oracle.com/javase/tutorial/networking/sockets/definition.html`
- `http://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html`
- `http://devmentor.org/articles/network/Socket%20Programming(v2).pdf`