Especialização Avançada em Sistemas Distribuídos - EASD

Módulo X – Programação em Redes com Java

Marcela Santana e
Sérgio Gorender

Universidade Federal da Bahia

Pacote java.net

- A linguagem Java possui uma biblioteca de classes responsáveis pela programação em rede, contendo *sockets unicast* (TCP e UDP) e também *multicast* (UDP). Esta biblioteca é a **java.net**.
- Java oferece os seguintes modos de utilização de **sockets**:
  - o modo orientado a conexão, que funciona sobre o protocolo TCP
  - o modo orientado a **datagrama**, que funciona sobre o protocolo UDP.
- Os dois modos funcionam sobre o protocolo IP.
- Cada um desses modos tem sua aplicabilidade, com vantagens e desvantagens.

Pacote java.net

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

Pacote java.net

- Para que serve um **Socket** ?

"Os sockets são mecanismos de comunicação entre processos (IPC - InterProcess Communication, desenvolvidos por um grupo de pesquisa da Universidade da Califórnia), que podem estar numa mesma máquina ou em máquinas diferentes via rede. São diferentes dos **pipes**, principalmente porque fazem distinção entre cliente e servidor, e a comunicação pode ocorrer nos dois sentidos."

Pacote java.net

Modo Orientado a conexão TCP/IP

O processo de comunicação no modo orientado à conexão ocorre da seguinte forma:

- O servidor escolhe uma determinada porta e fica aguardando conexões nesta porta.
- O cliente deve saber previamente qual a máquina servidora e a porta que o servidor está aguardando conexões.
- O cliente solicita uma conexão em um **host/porta**.
- Se nenhum problema ocorrer o servidor aceita a conexão gerando um **socket** em uma porta qualquer do lado do servidor.
- Assim é criado um canal de comunicação entre o cliente e o servidor.

Em Java a comunicação confiável(modo orientado a conexão) acontece através de **sockets** utilizando duas classes: **Socket** (soquete de dados) e **ServerSocket** (soquete do servidor)

Pacote java.net

ServerSocket

Alguns métodos

```
public ServerSocket(int  port)
public ServerSocket(int  port, int  count)
public Socket accept()
public void close()
public InetAddress getInetAddress()
public int getLocalPort()
```

Pacote java.net

Socket

Alguns métodos

```
public Socket(InetAddress  address, int  port)
public Socket(String  host, int  port)
public void close()
public InetAddress getInetAddress()
public InputStream getInputStream()
public OutputStream getOutputStream()
```

Pacote java.net

- Modelo de comunicação em Java

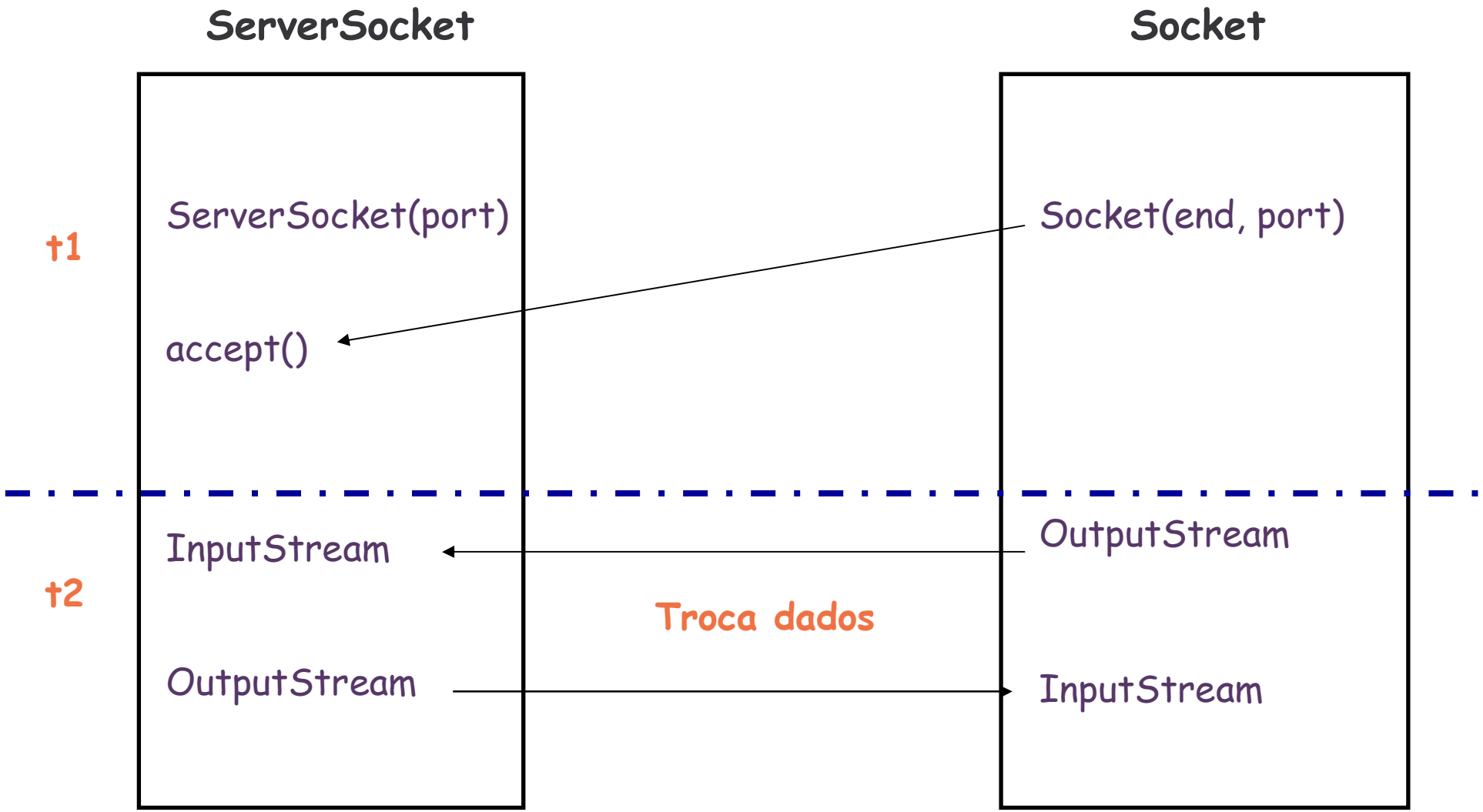
1) O servidor cria um **socket** associado a uma porta

2) O servidor espera uma conexão através do método **accept**

3) O cliente estabelece a conexão usando um **Socket**

4) Ambos se comunicam através de **InputStreams** e **OutputStreams**

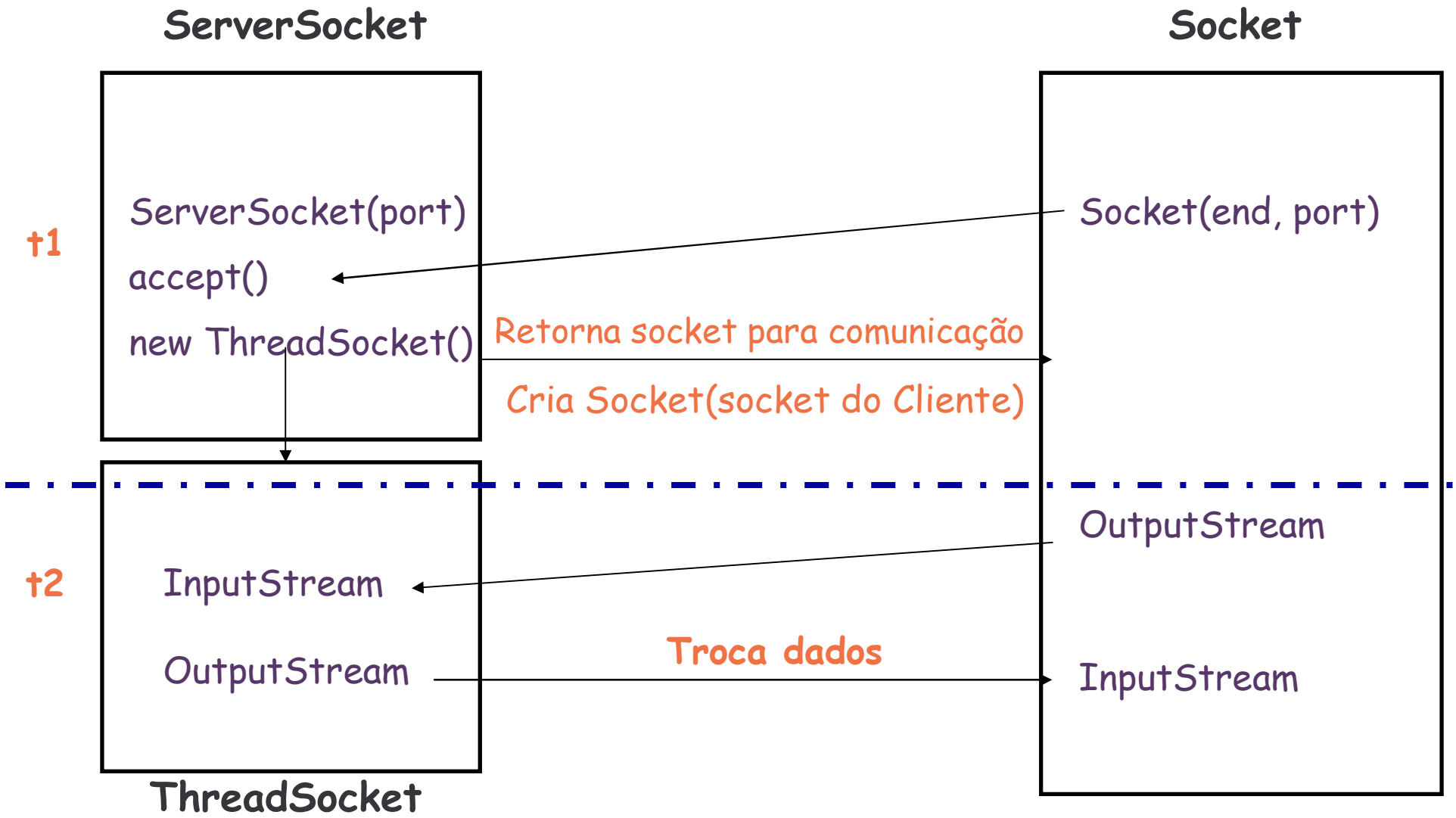
Pacote java.net



Pacote java.net

- Esta solução apresenta um problema grave. Não permite que o servidor atenda simultaneamente a muitos clientes.
- A rejeição de múltiplas conexões permite que qualquer cliente monopolize o serviço, conectando-se com ele por um longo tempo.
- É possível ter um serviço muito mais interessante através de múltiplas linhas de execução.
- Sempre que o servidor estabelece uma nova conexão de soquete, ou seja quando uma chamada de **accept** tem êxito, uma nova linha de execução deve ser ativada para cuidar da conexão entre o servidor e o cliente em questão. O servidor volta então para esperar novas conexões.

Pacote java.net



Exemplo

```
public class Cliente {
  public static void main(String[] args) {
   try{
     Socket conexao = new Socket(InetAddress.getLocalHost(), 8080);
     InputStream in = conexao.getInputStream();
     BufferedReader is = new BufferedReader(new InputStreamReader(in));
     String a= is.readLine();
     while (a!=null){
         System.out.println(a);
         a= is.readLine();
     }
     in.close();
     conexao.close();
    }catch (IOException e){
          e.printStackTrace();
    }
  }
}
```

Solicita conexão

Aguarda resposta do servidor

Exemplo

```
public class Servidor {
   public static void main(String[] args) {
    try{
      Socket cSocket;
      ServerSocket sSocket = new ServerSocket(8080);
      while(true){
           System.out.println("Aceitando solicitacoes");
           cSocket = sSocket.accept();
           ThreadAtendeCliente tc = new ThreadAtendeCliente(cSocket);
           tc.start();
      }
     }catch (IOException e){}
   }
}
```

Cria socket para comunicação

Aceita conexão através do socket

Cria thread para Atender ao cliente

Exemplo

```
class ThreadAtendeCliente extends Thread {
   private Socket cSocket;
   public ThreadAtendeCliente(Socket cSocket) {
      this.cSocket = cSocket;
    }
   public void run(){
     System.out.println("Atendendo a um ciente");
     try {
       InputStream in = cSocket.getInputStream();
       PrintWriter os = new PrintWriter(cSocket.getOutputStream());
       String msg = "Servidor "+ cSocket.getInetAddress() + "
            usando a porta " + cSocket.getPort() + " para responder a um cliente" ;
       msg=msg+"\n"+"A data e hora do sistema: " + new Date().toString();
       os.write(msg);
       os.close();
       cSocket.close();
     } catch (IOException e) {
       System.exit(1);
     }
 }
}
```

Exemplo2 – Conexão SMTP

```
public class TesteMail {
   static PrintStream ps = null;
   static DataInputStream dis = null;
public static void send(String str) throws IOException{
   ps.println(str);
   ps.flush();
   System.out.println("Enviando: " + str);
}
public static void receive() throws IOException{
   String readstr = dis.readLine(); // resposta do smtp
   System.out.println("Resposta: " + readstr);
}
public static void main (String args[]){
   String HELO = "HELO ";
   String MAIL_FROM =
            "MAIL FROM: marcela@ufba.br ";
   String RCPT_TO = "RCPT TO: marcela@ufba.br ";
   String SUBJECT = "SUBJECT: Teste em Java
            para envio de e-mail";
   String DATA = "DATA"; // começo da mensagem
   String BODY = "Corpo da mensagem . ";
   Socket smtp = null; // socket SMTP
try {
      smtp = new Socket("smtp.ufba.br", 25);
      OutputStream os = smtp.getOutputStream();
      ps = new PrintStream(os);
      InputStream is = smtp.getInputStream();
      dis = new DataInputStream(is);
   } catch (IOException e){
      System.out.println("Erro conectando: " + e);
   }
   try {
      send(HELO);
      receive(); // resposta SMTP
      send(MAIL_FROM); // envia SMTP
      receive(); // pega resposta SMTP
      send(RCPT_TO);
      receive(); // pega resposta SMTP novamente
      send(DATA); // envia begin de SMTP
      receive(); // pega resposta SMTP
      send(SUBJECT); // envia SUBJECT de SMTP
      receive(); // pega a resposta de SMTP
      send(BODY); // envia o corpo da mensagem
      receive(); // pega a resposta SMTP novamente
      smtp.close();
   } catch (IOException e){
      System.out.println("Erro ao enviar:" + e);
   }
   System.out.println("Mensagem enviada!");
}}
```

Exemplo2 - Resultado

Enviando: HELO work-q7vxr6c89b/201.4.18.222
Resposta: 220 itariri.ufba.br Kerio MailServer 6.0.8 ESMTP ready
Enviando: MAIL FROM: marcela@ufba.br
Resposta: 250 itariri.ufba.br
Enviando: RCPT TO: marcela@ufba.br
Resposta: 250 2.1.0 Sender <marcela@ufba.br> ok
Enviando: DATA
Resposta: 550 5.7.0 Your server IP address is in the **SORBS DNSBL** database, bye
Enviando: SUBJECT: Teste em Java para envio de e-mail
Resposta: null
Enviando: Corpo da mensagem .
Resposta: null
Mensagem enviada com sucesso!

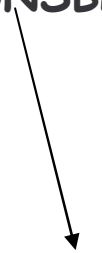


Tabela de Spam

Pacote java.net

Modo Orientado a datagrama 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.
Um 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.
Em Java a comunicação não confiável (modo orientado a **datagrama**) acontece através de **sockets** utilizando duas classes: **DatagramSocket** e **DatagramPacket**.

Pacote java.net

DatagramSocket (Socket para receber ou enviar datagramas)

Alguns métodos

```
public DatagramSocket()
public DatagramSocket(int  port)
public void close()
protected void finalize()
public int getLocalPort()
public void receive(DatagramPacket  p)
public void send(DatagramPacket  p)
```

Pacote java.net

DatagramPacket (Datagramas a serem enviados pela rede)

Alguns métodos

```
public DatagramPacket(byte  ibuf[], int  ilength)
public DatagramPacket(byte  ibuf[], int  ilength,
          InetAddress  iaddr, int iport)
public InetAddress getAddress()
public byte[] getData()
public int getLength()
public int getPort()
```

Exemplo

```
public class ServidorUDP
  public static void main(String[] args) {
       try{
          DatagramSocket servidor = new DatagramSocket(4545);
          DatagramPacket question = new DatagramPacket(new byte[512], 512);
          int i=0;
          while(i<10){
             System.out.println("Aguardando envio de informacoes");
             servidor.receive(question);
             String toSend="Resposta Servidor";
             byte [] buffer = toSend.getBytes();
             DatagramPacket resposta = new DatagramPacket(buffer, buffer.length,
                                  question.getAddress(), question.getPort());
             servidor.send(resposta);
             i++;
          }
          servidor.close();
       }catch (SocketException e){          e.printStackTrace();
       }catch (UnknownHostException o){     o.printStackTrace();
       }catch (IOException a){              a.printStackTrace();
     }
  }
}
```

Exemplo

```
public class ClienteUDP {
   public static void main(String args[]){
      try{
         DatagramSocket cliente = new DatagramSocket();
         InetAddress end = InetAddress.getLocalHost();
         String toSend ="PERGUNTA";
         byte[] bufferSend = toSend.getBytes();
         DatagramPacket pacoteEnviado = new DatagramPacket(bufferSend,
                                        bufferSend.length, end, 4545);
         cliente.send(pacoteEnviado);
         DatagramPacket pacoteRecebido = new DatagramPacket(new byte[512],512 );
         cliente.receive(pacoteRecebido);
         String fraseModificada = new String(pacoteRecebido.getData());
         System.out.println("Do Servidor:" + fraseModificada);
         cliente.close();
         System.out.println("FIM");
      }catch (SocketException e){          e.printStackTrace();
      }catch (UnknownHostException o){     o.printStackTrace();
      }catch (IOException a){              a.printStackTrace();
      }
   }
}
```

Pacote java.net

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) e uma porta UDP.

Classes IP definem intervalos de endereços. O endereço 224.0.0.0 é reservado e não deve ser usado.

Em Java o suporte a **multicast** é oferecido através da classe **java.net.MulticastSocket**

Pacote java.net

Multicast

```
InetAddress grupo = InetAddress.getByName("228.8.7.7");
MulticastSocket s = new MulticastSocket(4543);
s.joinGroup(grupo);


// envia e recebe mensagens via MulticastSocket,
// através de DatagramPacket


s.leaveGroup(group);
```