

Artigo

Invista em você! Saiba como a DevMedia pode ajudar sua carreira. 

Java Sockets: Criando comunicações em Java

Neste artigo veremos como desenvolver aplicações em Java que podem comunicar-se via rede local ou via internet, usando sockets.

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

Neste artigo veremos como desenvolver aplicações em Java que podem comunicar-se via rede local ou via internet, usando sockets. Os sockets são compostos por um conjunto de primitivas do sistema operacional e foram originalmente desenvolvidos para o BSD Unix. Podem ser utilizados nos mais variados sistemas operacionais com recursos de comunicação em rede, sendo suportados pela maioria das linguagens de programação. Sockets são suportados em Java desde o JDK 1.0, para sua utilização devemos fazer uso das classes contidas no pacote **java.net**. Um exemplo interessante da programação de sockets em Java são os drivers JDBC do tipo 4, que usam sockets para comunicar-se diretamente com a API de rede do banco de dados.

Estrutura básica de uma aplicação de rede

Uma aplicação que utiliza sockets normalmente é composta por uma parte servidora e diversos clientes. Um cliente solicita determinado serviço ao servidor, o servidor processa a solicitação e devolve a informação ao cliente (ver **Figura 1**). Muitos serviços podem ser disponibilizados numa mesma máquina, sendo então diferenciados não só pelo endereço IP, mas também por um número de porta.

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar



Fluxo de troca de dados com sockets

Figura 1 Fluxo de troca de dados com sockets.

Como primeiros passos na criação do servidor, é necessário importar o pacote `java.net` e em seguida instanciar um objeto do tipo `ServerSocket`, responsável por atender pedidos via rede e em determinada porta. Após receber uma conexão, um objeto do tipo `Socket` deve ser criado para manter a comunicação entre o cliente e o servidor.

Vejamos um exemplo. A seguinte linha cria o `ServerSocket`, que irá esperar conexões na porta 12345 (caso esta porta já esteja em uso, uma exceção será lançada):

```
1 | ServerSocket server = new ServerSocket(12345);
```

Em seguida criamos um objeto `Socket`, o qual irá tratar da comunicação com o cliente, assim que um pedido de conexão chegar ao servidor e a conexão for aceita:

```
1 | Socket client = server.accept();
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

de 1024 em diante, pois as portas com números abaixo deste são reservados para o uso do sistema (por exemplo a porta 80 é usada pelo protocolo HTTP, 25 pelo SMTP, 110 pelo POP3, entre vários outros serviços).

Endereços IP

Cada máquina conectada a uma rede possui um endereço IP único de maneira que possa ser identificada na rede. A classe InetAdress nos permite obter informações sobre um computador conectado a rede. Os principais métodos desta classe são os seguintes:

getAddress(): Este método retorna um array de bytes contendo o endereço IP. Para isso, o nome do host que se deseja obter o endereço IP é fornecido ao método *getByName* da classe *InetAddress*, veja um exemplo:

```
1 | byte[] b = InetAddress.getByName("localhost").getAddress();
2 | System.out.println(b[0] + "." + b[1] + "." + b[2] + "." + b[3]);
```

getHostAddress(): Este método retorna uma String contendo o endereço IP no formato 999.999.999.999, veja um exemplo:

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

`getHostName()`: Dado um array de bytes contendo o endereço IP de um host, este método retorna uma String com o nome do host, veja um exemplo:

```
1 | byte[] addr = {127, 0, 0, 1};  
2 | System.out.println(InetAddress.getByAddress(addr).getHostName());
```

O protocolo TCP

Quando necessitamos de uma troca confiável de informações, isto é, quando é necessária a confirmação de recebimento da mensagem enviada, devemos utilizar o protocolo **TCP** (Transmission Control Protocol). Este protocolo estabelece uma conexão entre dois pontos interligados. Por exemplo, uma mensagem enviada de um host (o termo host representa uma máquina conectada na rede) a outro é confirmada pelo host receptor indicando o correto recebimento da mensagem. Uma mensagem pode ser enviada em vários pacotes, o TCP cuida para que os pacotes recebidos sejam remontados no host de destino na ordem correta (caso algum pacote não tenha sido recebido, o TCP requisita novamente este pacote). Somente após a montagem de todos os pacotes é que as informações ficam disponíveis para nossas aplicações. A programação do TCP com sockets utiliza streams, o que simplifica muito o processo de leitura e envio de dados pela rede.

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

Streams são objetos Java que permitem obter dados de qualquer fonte de entrada, seja o teclado, um arquivo ou até mesmo um fluxo de bytes recebidos pela rede (o que é o nosso caso). Isto torna a manipulação de dados da rede como se fossem arquivos, ao ler dados enviados é como se estivéssemos lendo um arquivo e ao enviar dados é como se estivéssemos gravando dados em um arquivo.

Um primeiro servidor TCP

Vamos começar agora a trabalhar na prática com sockets. Primeiro vamos montar um servidor TCP que permite a seus clientes solicitarem a data e a hora atuais do servidor. A primeira versão deste servidor tem uma limitação (que mostraremos mais tarde como resolver): apenas um cliente pode ser atendido por vez.

Uma das características importantes do TCP é que os pedidos de conexões dos clientes vão sendo mantidos em uma fila pelo sistema operacional até que o servidor possa atendê-los. Isto evita que o cliente receba uma negação ao seu pedido, pois o servidor pode estar ocupado com outro processo e não conseguir atender o cliente naquele momento.

Cada sistema operacional pode manter em espera um número limitado de conexões até que sejam atendidas. Quando o sistema operacional recebe mais conexões que esse limite, as conexões mais

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

Veja como funciona o nosso primeiro exemplo:

- Ao ser iniciado o servidor fica ouvindo na porta 12345 a espera de conexões de clientes;
- O cliente solicita uma conexão ao servidor;
- O servidor exibe uma mensagem na tela com o endereço IP do cliente conectado;
- O servidor aceita a conexão e envia um objeto Date ao cliente;
- O cliente recebe o objeto do servidor e faz o cast necessário, em seguida exibe na tela as informações de data;
- O servidor encerra a conexão.

Na **Listagem 1** é apresentado o código do nosso primeiro exemplo de servidor e na **Listagem 2** é apresentado o código do cliente que utiliza o nosso servidor.

Listagem 1: Código do servidor TCP básico.

```
1 | public class ServidorTCPBasico {  
2 |     public static void main(String[] args) {  
3 |         try {  
4 |             // Instancia o ServerSocket ouvindo a porta 12345  
5 |             ServerSocket servidor = new ServerSocket(12345);  
6 |         } catch (IOException e) {  
7 |             e.printStackTrace();  
8 |         }  
9 |     }  
10| }
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
8     // o servidor receba um pedido de conexão
9     Socket cliente = servidor.accept();
10    System.out.println("Cliente conectado: " + cliente.getInetAddress().getHostAddress());
11    ObjectOutputStream saida = new ObjectOutputStream(cliente.getOutputStream());
12    saida.flush();
13    saida.writeObject(new Date());
14    saida.close();
15    cliente.close();
16  }
17 }
18 catch(Exception e) {
19   System.out.println("Erro: " + e.getMessage());
20 }
21 finally {...}
22 }
23 }
24 }
```

Listagem 2: Código do cliente TCP básico.

```
1 public class ClienteTCPBasico {
2   public static void main(String[] args) {
3     try {
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
7      JOptionPane.showMessageDialog(null,"Data recebida do servidor:" + data_atua
8      entrada.close();
9      System.out.println("Conexão encerrada");
10 }
11 catch(Exception e) {
12     System.out.println("Erro: " + e.getMessage());
13 }
14 }
15 }
```

Na **Figura 2** podemos ver as mensagens exibidas pelo servidor ao receber conexões e na **Figura 3** é apresentada a data atual do servidor recebida pelo cliente.



Mensagens exibidas pelo servidor ao receber conexões de clientes

Figura 2: Mensagens exibidas pelo servidor ao receber conexões de clientes.



Data e hora recebidas pelo cliente

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

O protocolo UDP

Quando necessitamos de uma troca não confiável de informações podemos usar o protocolo UDP (User Datagram Protocol), pois este protocolo não garante a entrega dos pacotes (o UDP não espera uma mensagem de confirmação do host de destino). É de responsabilidade da aplicação receptora a remontagem dos pacotes na ordem correta e a solicitação de reenvio de pacotes que não foram recebidos. O UDP utiliza datagram sockets para a troca de mensagens. As principais aplicações do UDP são aplicações como transmissões de vídeo, skype, voip, etc... Para exemplificar imagine um serviço de voz sobre IP onde um pacote é perdido enquanto dois usuários conversam, não faz sentido reenviar o pacote pois o usuário da outra ponta precisaria saber que ainda faltam pacotes a receber. Veja uma simulação abaixo da conversa entre dois usuários:

- Erika: Olá, Paulo.
- Paulo: Olá, Erika.
- Erika: Como você está?
- Paulo: Tudo bem e vc? (este pacote foi perdido)

No exemplo acima o TCP enviaria novamente o pacote e o usuário da outra ponta deveria ficar esperando. Não faria nenhum sentido isso tratando-se de uma ligação telefônica. Este é um típico caso

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

Os datagram sockets são mensagens que podem ser enviadas pela rede quando não existe a necessidade de confirmação de entrega, de tempo de entrega e nem mesmo garantia de conteúdo. Datagramas são úteis em aplicações que não necessitam do estabelecimento de uma conexão para o envio da mensagem. Um bom exemplo do seu uso é o envio de mensagens em broadcast para clientes de uma rede (o servidor pode enviar um datagrama para todos os clientes avisando que irá reiniciar, por exemplo). Em Java podemos trabalhar com datagramas utilizando as classes DatagramPacket e DatagramSocket do pacote java.net.

Um servidor UDP básico

Antes de mostrarmos como criar um servidor TCP capaz de receber várias conexões simultâneas, vamos mostrar como criar um servidor UDP. Como vimos, o UDP envia os pacotes sem esperar por uma resposta do receptor. Este protocolo pode ser útil em situações como o envio de pacotes multimídia, por exemplo, ou um serviço de voz sobre ip, o que é muito comum.

Nosso servidor UDP envia mensagens para os clientes de uma determinada rede local. Perceba neste exemplo que no UDP o cliente também aguarda mensagens que poderão ser enviadas pelo servidor, ou seja, mantém um DatagramSocket em uma determinada porta. Por exemplo, o seguinte trecho cria o

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
1 | DatagramSocket serverdgram = new DatagramSocket(12346);
```

Veja na **Listagem 3** o remetente UDP exemplo e na **Listagem 4** o código do receptor UDP, na **Figura 4** você pode ver a mensagem sendo enviada pelo remetente. Na **Figura 5** temos o receptor sendo inicializado para receber as mensagens e na Figura 6 a mensagem recebida pelo receptor. Como estamos usando UDP neste caso o remetente sempre vai mostrar a mensagem indicando que a mensagem foi enviada, porém não existe a confirmação de que a mensagem foi recebida.

Listagem 3. Classe RemetenteUDP.

```
1 | public class RemetenteUDP {  
2 |  
3 |     public static void main(String[] args) {  
4 |  
5 |         if(args.length != 3) {  
6 |             System.out.println("Uso correto: <Nome da maquina> <Porta> <Mensagem>");  
7 |             System.exit(0);  
8 |         }  
9 |     try {  
10 |
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
14     byte[] msg = args[2].getBytes();
15     //Monta o pacote a ser enviado
16     DatagramPacket pkg = new DatagramPacket(msg,msg.length, addr, port);
17     // Cria o DatagramSocket que será responsável por enviar a mensagem
18     DatagramSocket ds = new DatagramSocket();
19     //Envia a mensagem
20     ds.send(pkg);
21     System.out.println("Mensagem enviada para: " + addr.getHostAddress() + "\n"
22                     "Porta: " + port + "\n" + "Mensagem: " + args[2]);
23
24     //Fecha o DatagramSocket
25     ds.close();
26 }
27
28     catch(IOException ioe) {...}
29 }
30 }
```

Listagem 4. Classe ReceptorUDP.

```
1 public class ReceptorUDP {
2 }
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
6         System.exit(0);
7     }
8
9     try {
10        //Converte o argumento recebido para inteiro (numero da porta)
11        int port = Integer.parseInt(args[0]);
12        //Cria o DatagramSocket para aguardar mensagens, neste momento o método t
13        //até o recebimento de uma mensagem
14        DatagramSocket ds = new DatagramSocket(port);
15        System.out.println("Ouvindo a porta: " + port);
16        //Preparando o buffer de recebimento da mensagem
17        byte[] msg = new byte[256];
18        //Prepara o pacote de dados
19        DatagramPacket pkg = new DatagramPacket(msg, msg.length);
20        //Recebimento da mensagem
21        ds.receive(pkg);
22        JOptionPane.showMessageDialog(null,new String(pkg.getData()).trim(),
23            "Mensagem recebida",1);
24        ds.close();
25    }
26
27    catch(IOException ioe) {...}
28 }
29 }
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar



Remetente UDP em ação (enviando mensagem ao receptor)

Figura 4. Remetente UDP em ação (enviando mensagem ao receptor).



Receptor sendo inicializado para receber mensagens na porta 1234

Figura 5. Receptor sendo inicializado para receber mensagens na porta 1234.



Mensagem UDP recebida pelo receptor

Figura 6. Mensagem UDP recebida pelo receptor.

Um servidor TCP com múltiplos threads

Como vimos no nosso primeiro exemplo de servidor TCP, um servidor de produção não deve se limitar a processar uma conexão de cada vez, devendo ser capaz de atender inúmeras conexões simultaneamente. Veremos agora como montar um servidor TCP mais profissional usando threads.

Um caso-prático de uso de sockets em Java

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

Vamos ver agora o exemplo de um cliente que solicita arquivos de imagem ao servidor. O cliente deve informar o caminho do arquivo no servidor, ao receber o pedido o servidor enviará o arquivo ao cliente. A aplicação exemplo solicita um arquivo JPG presente em um diretório do servidor, e ao receber o arquivo, exibe um JLabel o conteúdo do arquivo.

Para cada solicitação de cliente será criada uma nova thread para gerenciar a troca de mensagens, deixando assim o servidor livre para esperar por novas conexões. Vejamos detalhadamente este processo.

Na **Listagem 5** temos o código do servidor de arquivos e na **Listagem 6** o código da thread que trata do pedido específico. Na **Figura 7** temos a aplicação em ação solicitando arquivos ao servidor e na **Figura 8** temos o console de atividades do servidor. Na **Listagem 7** você pode ver o código do botão Buscar da aplicação exemplo.

Listagem 5. Servidor de arquivos multithreaded.

```
1 | public class ServidorArquivo {  
2 |  
3 |     public static void main(String[] args) {
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
7     }
8
9     try {
10        //Converte o parametro recebido para int (número da porta)
11        int port = Integer.parseInt(args[0]);
12        System.out.println("Incializando o servidor...");
13        //Inicia o servidor
14        ServerSocket serv = new ServerSocket(port);
15        System.out.println("Servidor iniciado, ouvindo a porta " + port);
16        //Aguarda conexões
17        while(true) {
18            Socket clie = serv.accept();
19            //Inicia thread do cliente
20            new ThreadCliente(clie).start();
21        }
22    }
23    catch(Exception e) {...}
24 }
25 }
```

Listagem 6. Thread para tratar conexões recebidas pelo servidor.

```
1 | class ThreadCliente extends Thread {
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
5  public ThreadCliente(Socket cliente) {
6      this.cliente = cliente;
7  }
8
9  public void run() {
10     try {
11         //ObjectInputStream para receber o nome do arquivo
12         ObjectInputStream entrada = new ObjectInputStream(cliente.getInputStream());
13         DataOutputStream saida = new DataOutputStream(cliente.getOutputStream());
14         //Recebe o nome do arquivo
15         String arquivo = (String)entrada.readObject();
16         //Buffer de leitura dos bytes do arquivo
17         byte buffer[] = new byte[512];
18         //Leitura do arquivo solicitado
19         FileInputStream file = new FileInputStream(arquivo);
20         //DataInputStream para processar o arquivo solicitado
21         DataInputStream arq = new DataInputStream(file);
22         saida.flush();
23         int leitura = arq.read(buffer);
24         //Lendo os bytes do arquivo e enviando para o socket
25
26         while(leitura != - 1) {
27             if(leitura != - 2) {
28                 saida.write(buffer,0,leitura);
29             }
30             leitura = arq.read(buffer);
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
34         cliente.getRemoteSocketAddress().toString());
35
36         entrada.close();
37         saida.close();
38         cliente.close();
39     }
40
41     catch(Exception e) {
42         System.out.println("Excecao ocorrida na thread: " + e.getMessage());
43         try {
44             cliente.close();
45         }
46
47         catch(Exception ec) {}
48     }
49 }
50 }
```

Listagem 7. Código do botão buscar arquivo presente na aplicação exemplo (Figura 6).

```
1 | private void btnBuscarActionPerformed(java.awt.event.ActionEvent evt) {
2 | }
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
6 //Enviando o nome do arquivo a ser baixado do servidor
7 ObjectOutputStream saida = new ObjectOutputStream(rec.getOutputStream());
8 saida.writeObject(txtArquivo.getText());
9
10 // DataInputStream para processar os bytes recebidos
11 DataInputStream entrada = new DataInputStream(rec.getInputStream());
12 //FileOuputStream para salvar o arquivo recebido
13 FileOutputStream sarq = new FileOutputStream(txtSaida.getText());
14 byte[] br = new byte[512];
15 int leitura = entrada.read(br);
16 while(leitura != -1) {
17     if(leitura != -2) {
18         sarq.write(br, 0, leitura);
19     }
20     leitura = entrada.read(br);
21 }
22
23
24 saida.close();
25 entrada.close();
26 sarq.close();
27 rec.close();
28 ImageIcon img = new ImageIcon(txtSaida.getText());
29 lblImagem.setText("");
30 lblImagem.setIcon(img);
31 }
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
35 }  
36 }
```



Aplicação de exemplo que busca um arquivo JPG no servidor

Figura 7. Aplicação de exemplo que busca um arquivo JPG no servidor.



Mensagens exibidas pela thread do servidor

Figura 8. Mensagens exibidas pela thread do servidor.

Broadcast

Broadcast significa enviar datagram sockets para todos os clientes conectados em uma determinada rede, uma das situações onde o broadcast é aplicado é o envio de mensagens de notificação (um exemplo clássico é um servidor informando a seus clientes que irá reinicializar).

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

Como vimos o broadcast é muito útil porém o fato de enviar mensagens a todos os clientes da rede em algumas situações não é desejado, pois é necessário o envio de mensagens apenas para um grupo de clientes da rede. Para resolver este problema usamos multicast, uma técnica que possibilita enviar datagramas para um grupo de clientes conectados na rede.

Os clientes da rede interessados em receber estas mensagens devem participar de um grupo de multicasting. O multicast é suportado pelo UDP, o que consiste em mais uma diferença em relação ao TCP. Em Java temos a classe MulticastSocket do pacote java.net a qual nos permite trabalhar com multicast. Ela se assemelha bastante com um socket datagram porém com capacidades adicionais como formação de grupos multicast.

Os endereços reservados para o multicast estão entre 224.0.0.0 e 239.255.255.255, sendo os endereços com prefixo 239 reservados para uso em intranets. Veja abaixo um exemplo de cliente assinante de multicast.

```
1 | InetAddress addrgrp = InetAddress.getByName("239.0.0.1");
2 | // Cria o Multicast para a porta 12347 que ira permitir ao cliente participar de
3 | MulticastSocket mcs = new MulticastSocket(12347);
4 | // Após a criação do MulticastSocket é necessário utilizar o método joinGroup(Inet
5 | // seja associado ao endereço de multicasting, o que significa fazer a assinatura
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

Um servidor e um cliente Multicast

Vamos montar agora um servidor e um cliente multicast. Neste exemplo temos a seguinte situação:

- O cliente assina um grupo multicast, no nosso caso “239.0.0.1”, a partir deste momento o cliente aguarda por mensagens do servidor;
- O servidor envia mensagens multicast para o grupo “239.0.0.1”;
- O cliente recebe a mensagem e a exibe na tela.

Na **Listagem 8** temos o código do servidor, o qual envia mensagens para um grupo de clientes e na Listagem 9 temos o código do cliente o qual espera por uma mensagem multicast do servidor. Na Figuras 9 e 10 temos respectivamente as atividades do servidor e do cliente.

Listagem 8. Servidor multicast que envia mensagens para um grupo de clientes.

```
1 | public class ServidorMulticast {  
2 |  
3 |     public static void main(String[] args) {
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
7     System.exit(0);
8 }
9
10 try {
11     byte[] b = args[2].getBytes();
12     InetAddress addr = InetAddress.getByName(args[0]);
13     DatagramSocket ds = new DatagramSocket();
14     DatagramPacket pkg = new DatagramPacket(b, b.length, addr, Integer.parseInt(args[1]));
15     ds.send(pkg);
16 }
17 catch(Exception e) {
18     System.out.println("Nao foi possivel enviar a mensagem");
19 }
20 }
21 }
```

Listagem 9. Cliente multicast que aguarda mensagens do servidor.

```
1 public class ClienteMulticast {
2
3     public static void main(String[] args) {
4 }
```

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

```
8 |         InetAddress grp = InetAddress.getByName("239.0.0.1");
9 |         mcs.joinGroup(grp);
10|         byte rec[] = new byte[256];
11|         DatagramPacket pkg = new DatagramPacket(rec, rec.length);
12|         mcs.receive(pkg);
13|         String data = new String(pkg.getData());
14|         System.out.println("Dados recebidos:" + data);
15|     }
16|     catch(Exception e) {
17|         System.out.println("Erro: " + e.getMessage());
18|     }
19| }
20| }
21| }
```



Servidor multicast enviando mensagens aos clientes do grupo 239.0.0.1

Figura 9. Servidor multicast enviando mensagens aos clientes do grupo 239.0.0.1.



Mensagens de multicast recebidas pelo cliente

Figura 10. Mensagens de multicast recebidas pelo cliente.

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

Conclusões

Desenvolver aplicações que se comunicam em rede local ou internet é hoje uma necessidade crescente. Neste artigo aprendemos a desenvolver este tipo de aplicação usando Java e sockets. Os sockets em Java representam um recurso poderoso para desenvolvimento de aplicações que podem comunicar-se via rede. Apesar de existirem frameworks que facilitam o desenvolvimento de aplicações em rede com Java é importante ao leitor entender o fundamento da comunicação com sockets que é a base para toda e qualquer aplicação que utiliza comunicação em rede.

Referencias

- Java, guia do programador – Peter Jndl Junior, editora Novatec
- Java, como programar – 6^a Edição – H. M. Deitel e P. J. Deitel, editora Prentice Hall
- Java network programming – 3rd Edition – Elliotte Rusty Harold, editora O'REILLY

Tecnologias:

Java

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

Chega de perder tempo!

Comece hoje mesmo a programar de verdade

Apenas 12x de R\$ 54,90

POR QUE A DEVMEDIA?

- Didática focada no iniciante
- Aprenda construindo projetos reais
- Domine as tecnologias mais usadas no mercado

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

Comece agora



Por **Paulo**

Em 2008

RECEBA NOSSAS NOVIDADES

Informe o seu e-mail

Receber Newsletter

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar

Tecnologias

Exercicios

Cursos

Artigos

Revistas

Quem Somos

Fale conosco

Plano para Instituição de ensino

Assinatura para empresas

Assine agora

Tecnologias

Fundamentos

HTML e CSS

Mobile

.NET

JavaScript

Spring

PHP

Node

Arquitetura

Python

React Native

Automação

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar



Hospedagem web por Porta 80 Web Hosting

Utilizamos cookies para fornecer uma melhor experiência para nossos usuários, consulte nossa [política de privacidade](#).

Aceitar