

# **Programação de Sistemas Distribuídos - Java para Web**

## **2 - Sockets**

### **Citeforma**

Jose Aser Lorenzo, Pedro Nunes, Paulo Jorge Martins

[jose.l.aser@sapo.pt](mailto:jose.l.aser@sapo.pt), [pedro.g.nunes@gmail.com](mailto:pedro.g.nunes@gmail.com),  
[paulojsm@gmail.com](mailto:paulojsm@gmail.com)

Maio 2010

V 1.3

## Sumário

<b>2- Sockets .....</b>	<b>2-3</b>
<b>2.1- Objectivos.....</b>	<b>2-4</b>
<b>2.2- Modelo OSI e arquitectura TCP .....</b>	<b>2-8</b>
2.2.1- Modelo OSI .....	2-9
2.2.2- Arquitectura TCP/IP .....	2-11
2.2.3- Modelo OSI vs Arquitectura TCP/IP .....	2-12
2.2.4- Arquitectura TCP/IP em detalhe .....	2-13
<b>2.3- Sockets .....</b>	<b>2-14</b>
2.3.1- Arquitectura cliente/servidor .....	2-15
2.3.2- Camada de transporte.....	2-16
2.3.3- Para que servem os portos de comunicações? .....	2-17
2.3.4- Como os portos são identificados? .....	2-18
2.3.5- Portos TCP – Os mais conhecidos .....	2-19
2.3.6- Portos UDP – Os mais conhecidos .....	2-20
2.3.7- Socket .....	2-21
2.3.8- Sockets e Java .....	2-23
<b>2.4- TCP e UDP .....</b>	<b>2-24</b>
2.4.1- TCP - Transport Control Protocol .....	2-25
2.4.2- TCP: Conteúdo de um pacote .....	2-26
2.4.3- TCP: Comunicação entre sockets .....	2-27
2.4.4- TCP: Programação em Java .....	2-30
2.4.5- UDP - User Datagram Protocol .....	2-30
2.4.6- UDP: Datagram .....	2-30
2.4.7- UDP: Comunicação entre sockets.....	2-30
2.4.8- UDP: Programação em Java .....	2-30
2.4.9- TCP vs UDP .....	2-30
<b>2.5- Ambiente de trabalho.....</b>	<b>Error! Bookmark not defined.</b>
2.5.1- Criar e configurar um projecto usando JDev .....	<b>Error! Bookmark not defined.</b>
<b>2.6- TCP – Implementar Cliente &amp; Servidor .....</b>	<b>2-30</b>
2.6.1- TCP – Servidor em Java .....	2-30
2.6.2- TCP – Cliente em Java .....	2-30
2.6.3- Exercício – Servidor e Cliente .....	2-30
<b>2.7- UDP – Implementar Cliente &amp; Servidor .....</b>	<b>2-30</b>
2.7.1- UDP – Servidor em Java.....	2-30
2.7.2- UDP - Cliente em Java .....	2-30
2.7.3- Exercício – Servidor e Cliente .....	2-30
<b>2.8- TCP – Multithreaded .....</b>	<b>2-30</b>
2.8.1- Vantagens de usar Multithreaded .....	2-30
2.8.2- TCP – Non-Multithreaded .....	2-30
2.8.3- TCP: Multithreaded .....	2-30
2.8.4- Exercício – Multithreaded .....	2-30
<b>2.9- Multicast .....</b>	<b>2-30</b>
2.9.1- Unicast .....	2-30
2.9.2- Multicast .....	2-30
2.9.3- Exemplos de endereços multicast .....	2-30
2.9.4- Regras Multicast .....	2-30
2.9.5- Multicast: Programação em Java para enviar mensagem .....	2-30
2.9.6- Multicast: Programação em Java para receber mensagem.....	2-30
2.9.7- Exercício – Multicast .....	2-30



# Programação de Sistemas Distribuídos - Java para a Web

---

## Capítulo 2 – *Sockets*

José Aser Lorenzo  
Pedro Nunes  
Paulo Jorge Martins



**Java Web**  
© Citeforma

## 2- Sockets

## Objectivos

- Conhecer o Modelo OSI e a Arquitectura TCP/IP, sendo capaz de identificar as suas semelhanças e diferenças;
- Conhecer a utilidade dos *sockets* e a forma como comunicam entre si;
- Utilizar a linguagem Java para escrever programas que comunicam entre si através de *sockets* e que tiram partido dos protocolos que estão disponíveis;



### 2.1- Objectivos

No fim deste capítulo o formando estará apto a trabalhar com sockets.

## Sumário

- Ambiente de trabalho;
- Modelo OSI e arquitectura TCP;
- *Sockets*;
- TCP e UDP;
- TCP – Implementar Cliente & Servidor;
- UDP – Implementar Cliente & Servidor;
- TCP – Multithreaded;
- UDP – *Multicast*.



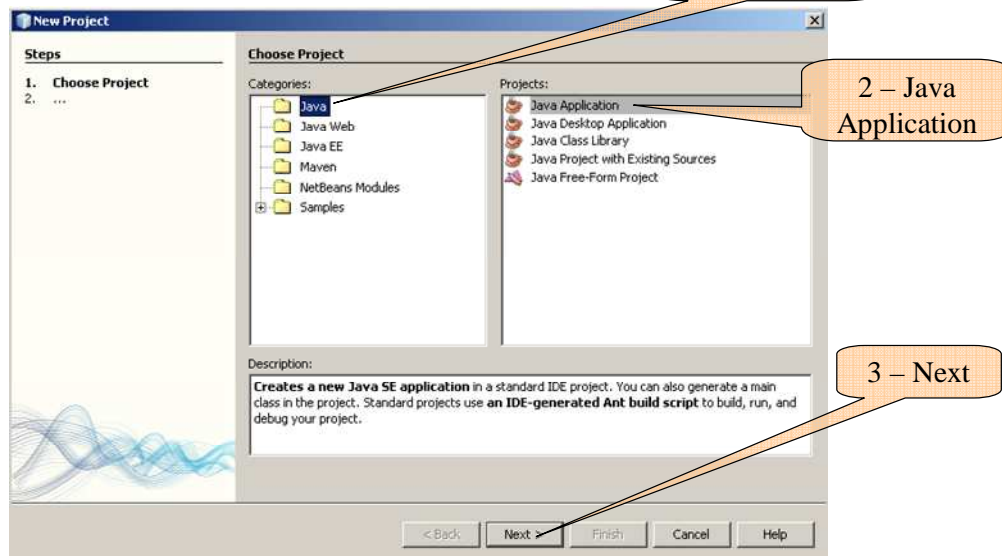
### 2.2- Ambiente de trabalho

Vamos criar um projecto para albergar os exercícios deste capítulo. Para isso siga as instruções dos próximos slides.

## NetBeans - Criar um projecto

#1/3

○ File → New Project



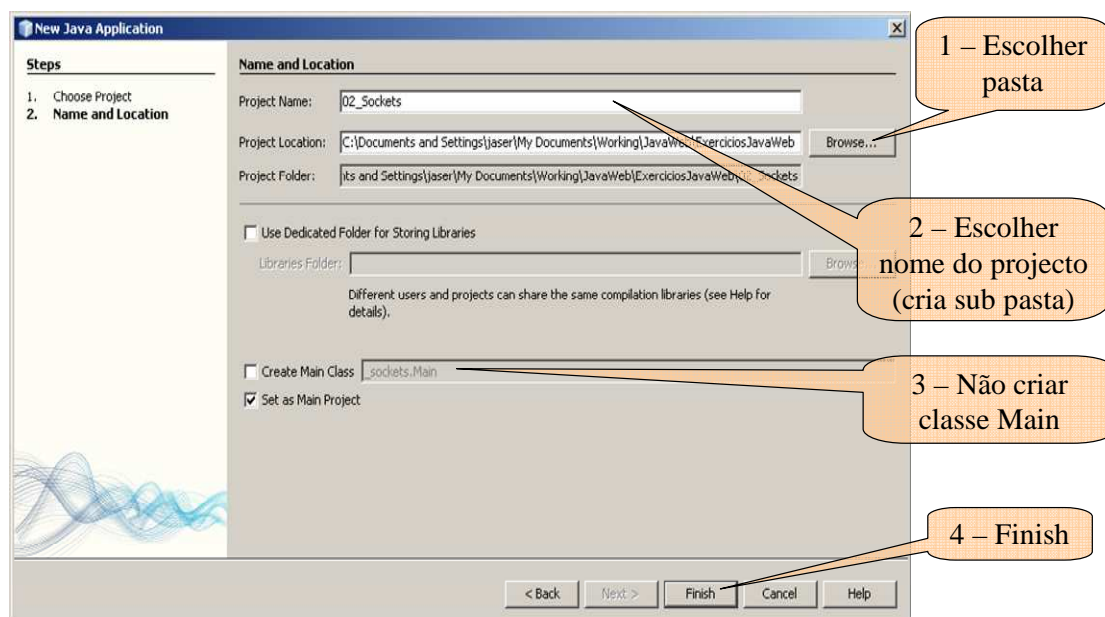
Java Web  
© Citeforma

Capítulo 2 - Sockets

3

## NetBeans - Criar um projecto

#2/3



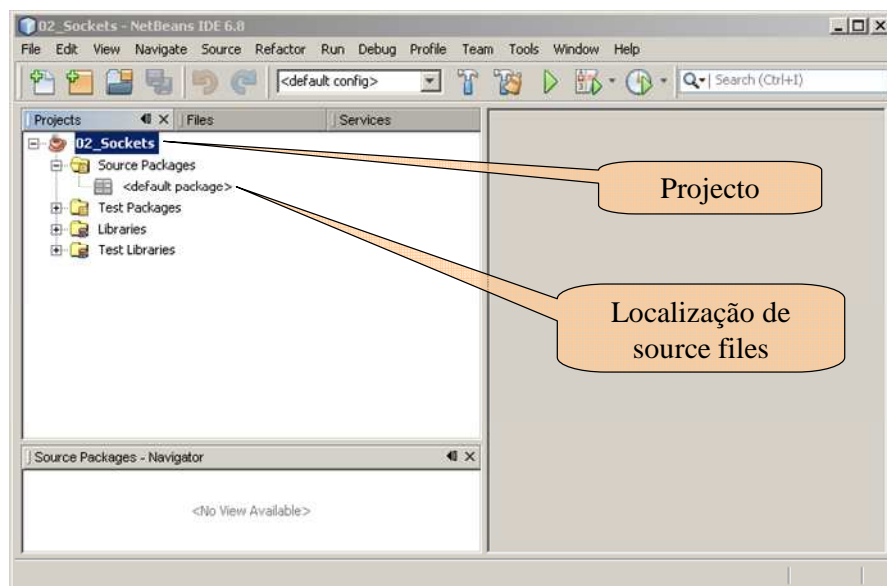
Java Web  
© Citeforma

Capítulo 2 - Sockets

4

## NetBeans - Criar um projecto

#3/3



## Sumário

---

- Ambiente de trabalho;
- Modelo OSI e arquitetura TCP;
- *Sockets*;
- TCP e UDP;
- TCP – Implementar Cliente & Servidor;
- UDP – Implementar Cliente & Servidor;
- TCP – Multithreaded;
- UDP – *Multicast*.



### 2.3- Modelo OSI e arquitetura TCP



## Modelo OSI

#1/3

- É um modelo de referência que divide as redes de computadores em sete camadas;
- Criado em 1984 pela *International Organization for Standardization* (ISO):

### Camadas do Modelo OSI:

1. Camada Física;
2. Camada de Enlace ou Ligação de Dados;
3. Camada de Rede;
4. Camada de Transporte;
5. Camada de Sessão;
6. Camada de Apresentação;
7. Camada de Aplicação;



Java Web  
© Citeforma

Capítulo 2 - Sockets

7

### 2.3.1- Modelo OSI

O slide anterior e os próximos dois mostram as características do modelo OSI.

## Modelo OSI

#2/3

### 1. Camada Física

Ocupa-se das características técnicas dos dispositivos eléctricos (físicos) do sistema (equipamentos e conexões físicas em si);

### 2. Camada de Enlace ou Ligação de Dados

Ocupa-se do direccionamento físico, da topologia da rede, do acesso à rede, da notificação de erros, da distribuição ordenada dos pacotes de dados e do controle de fluxo.

### 3. Camada de Rede

Garante que os dados vão da origem ao destino.

### 4. Camada de Transporte

Aceita os dados enviados pelas camadas superiores e passa-os à camada de rede.



## Modelo OSI

#3/3

### 5. Camada de Sessão

- Controlar a concorrência;
- Controlar as sessões a estabelecer entre o emissor e o receptor;
- Manter pontos de verificação (checkpoints).

### 6. Camada de Apresentação

Encarrega-se da apresentação da informação.

### 7. Camada de Aplicação

Oferece às aplicações a possibilidade de aceder aos serviços de outras camadas e define os protocolos que as aplicações utilizam para a troca de dados.



## Arquitectura TCP/IP

- A *Internet* baseia-se na arquitectura TCP/IP;
- A arquitectura TCP/IP possui apenas **quatro** camadas, que podem ser mapeadas com o modelo OSI.

### Camadas da Arquitectura TCP/IP:

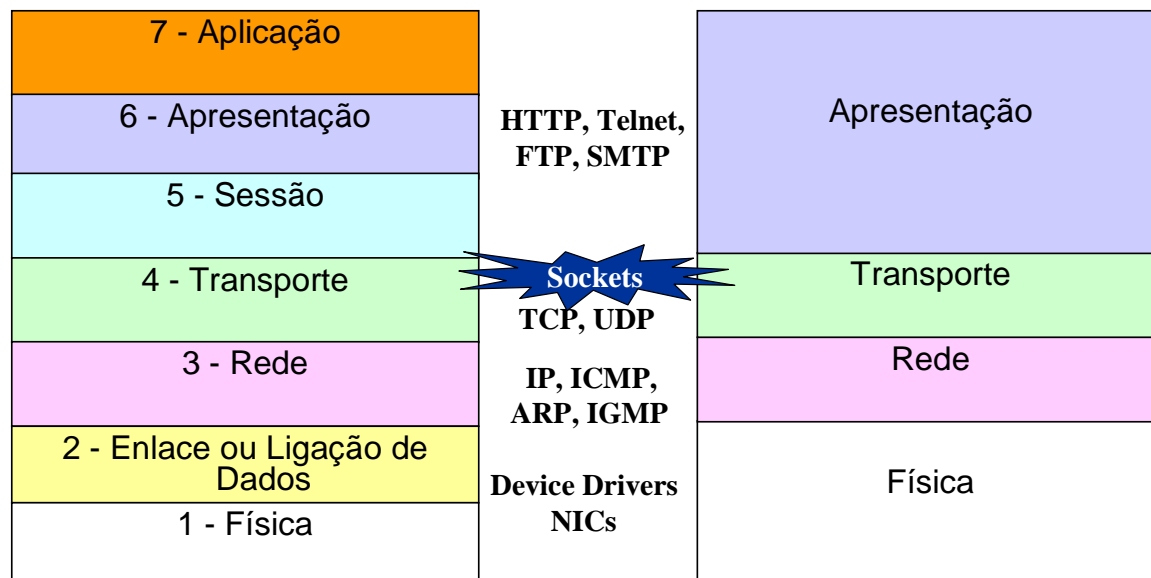
1. Camada Física (OSI – 1 e 2)
2. Camada de Rede (OSI - 3)
3. Camada de Transporte (OSI - 4)
4. Camada de Aplicação (OSI – 5, 6 e 7)



### 2.3.2- Arquitectura TCP/IP

O slide anterior mostra as características da arquitectura TCP/IP.

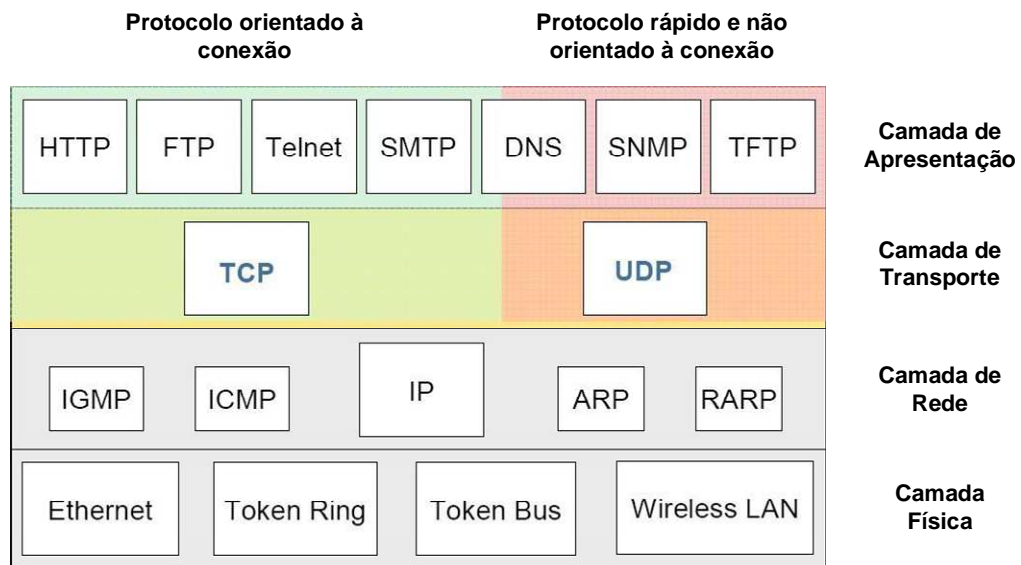
## Modelo OSI vs Arquitectura TCP/IP



### 2.3.3- Modelo OSI vs Arquitectura TCP/IP

O slide anterior compara o modelo OSI com a arquitectura TCP.

## Arquitectura TCP/IP em detalhe



### 2.3.4- Arquitectura TCP/IP em detalhe

O slide anterior mostra a arquitectura TCP/IP em detalhe.

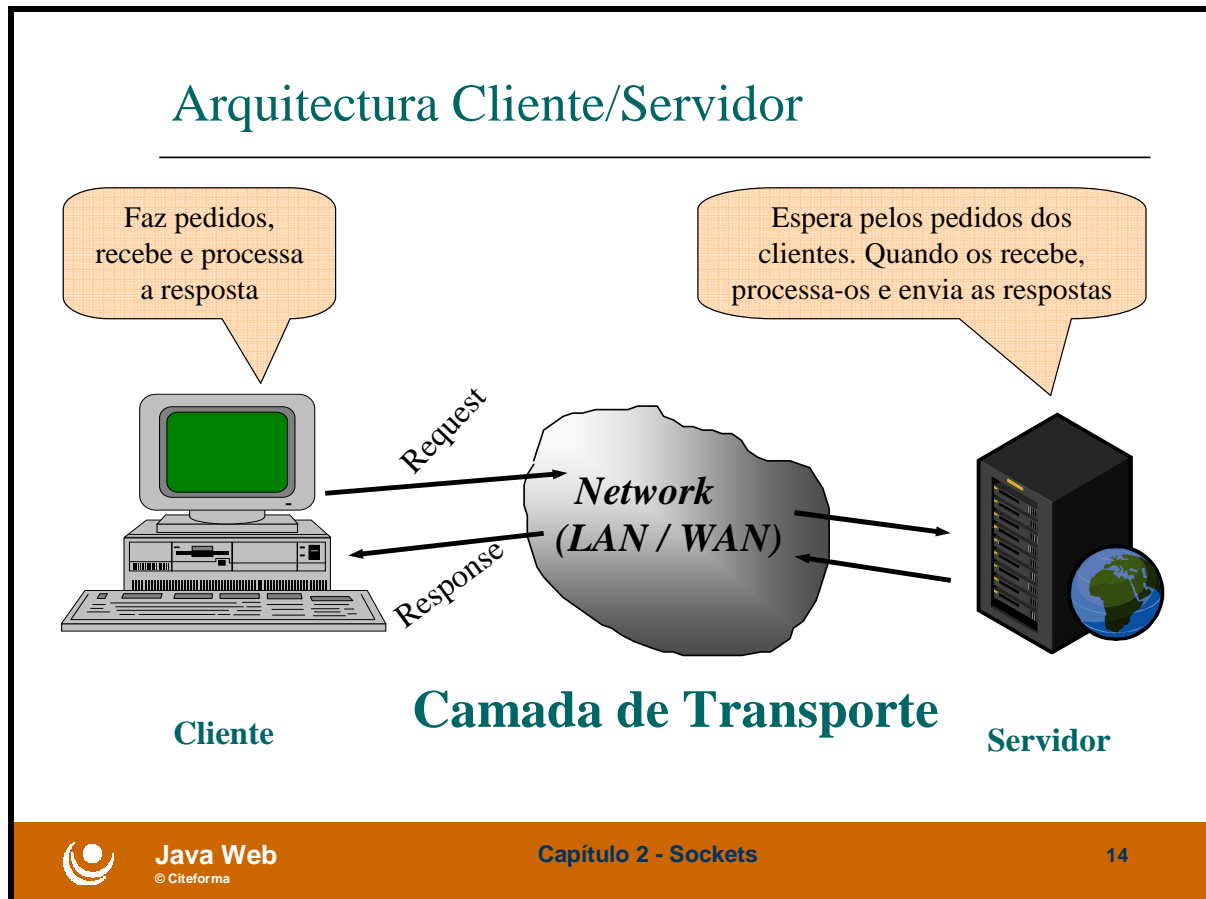
## Sumário

---

- Ambiente de trabalho;
- Modelo OSI e arquitectura TCP;
- *Sockets*;
- TCP e UDP;
- TCP – Implementar Cliente & Servidor;
- UDP – Implementar Cliente & Servidor;
- TCP – Multithreaded;
- UDP – *Multicast*.



### 2.4- Sockets



### 2.4.1- Arquitectura cliente/servidor

Na arquitectura cliente/servidor o servidor presta serviços e o cliente solicita os serviços. Nesta perspectiva, o servidor assume uma atitude reactiva, esperando que o cliente faça o pedido. Depois de receber o pedido o servidor deve satisfaze-lo no menor espaço de tempo possível.

## Camada de Transporte

### ○ Quais os protocolos a utilizar ?

- **TCP** (*Transport Control Protocol*)

É um protocolo de comunicação fiável entre dois computadores.

**Exemplos de Aplicações:** HTTP, Telnet, FTP, ...

- **UDP** (*User Datagram Protocol*)

É um protocolo de comunicação que envia pacotes de dados independentes (*datagrams*), de um computador para outro, sem qualquer garantia de que chegam ao seu destino.

**Exemplos de Aplicações:** ping, clock, chat, ...



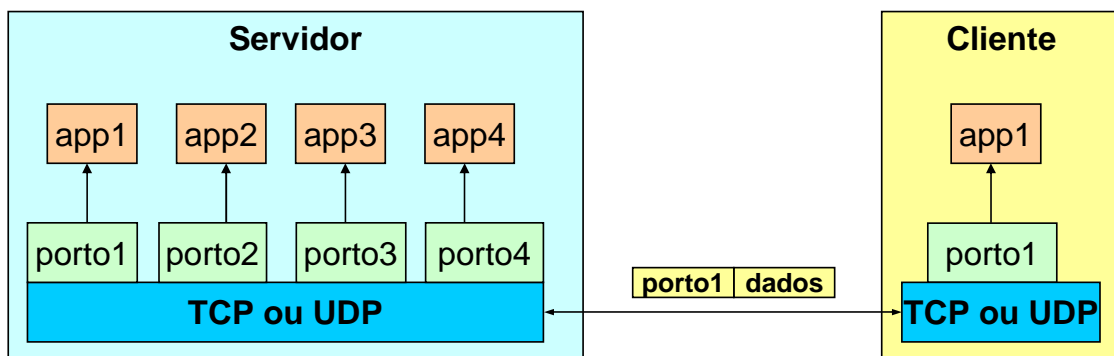
### 2.4.2- Camada de transporte

No transporte de dados entre cliente e servidor são principalmente usados dois protocolos: TCP e UDP. Estes protocolos serão descritos em detalhe mais à frente neste manual.



## Para que servem os portos de comunicações?

- O porto é usado por uma aplicação para ler (escrever) dados que são transferidos de (para) outra aplicação, normalmente a correr noutra máquina;
- O porto é “agarrado” pelo primeiro processo que o requisita, não sendo partilhado por outros processos;



### 2.4.3- Para que servem os portos de comunicações?

Um porto de comunicações é uma porta de entrada saída, por onde circulam bits, em série. Esses bits podem ser agrupados em bytes. Um porto é requisitado por um processo e não pode ser partilhado por dois processos distintos.

## Como os portos são identificados?

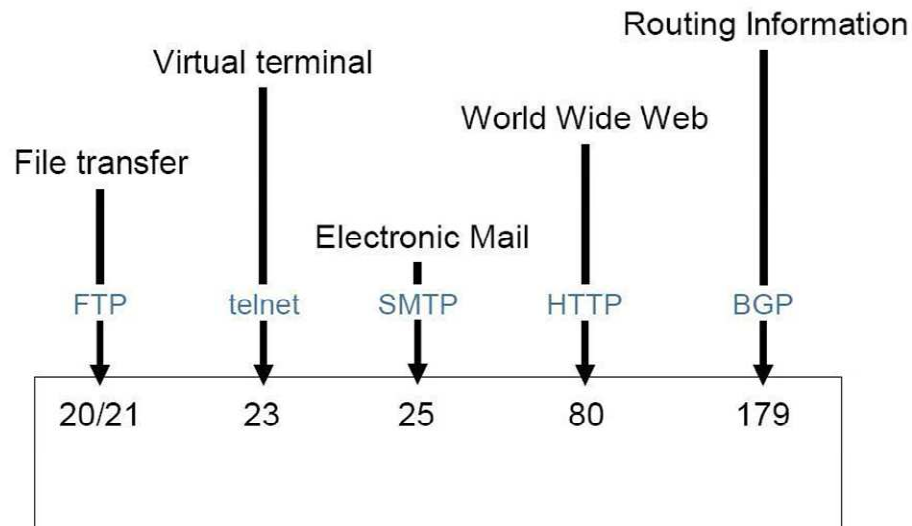
- Um porto de comunicações é identificado por um número inteiro positivo de 16-bit (*integer* até 65535);
- Alguns portos são reservados para suportar **serviços *standard***, normalmente entre o 0 e 255;
- Os serviços ou processos não standard devem usar os portos com número **superior a 1024**;
- Em Windows podemos usar os comandos abaixo para verificar os portos e os serviços que os ocupam:  
*netstat -a* ou *netstat -na*  
*netstat -a -b -p TCP* ou *netstat -a -b -p UDP*



### 2.4.4- Como os portos são identificados?

O slide acima descreve como os portos de comunicações são identificados. Também descreve os comandos de sistema operativo que podem ser executados para verificar os portos que estão ocupados e os processos que os requisitaram.

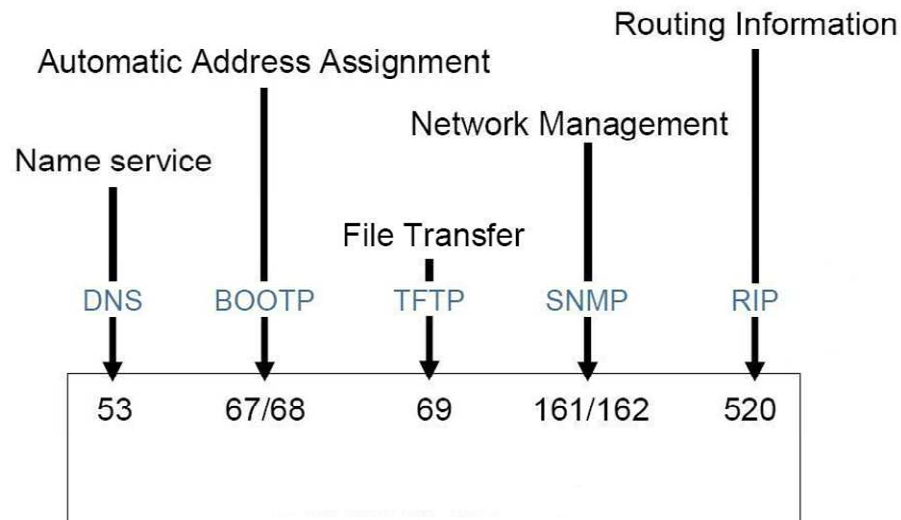
## Portos TCP – Os mais conhecidos...



### 2.4.5- Portos TCP – Os mais conhecidos

O slide anterior mostra os portos TCP/IP utilizados por aplicações populares.

## Portos UDP – Os mais conhecidos...



### 2.4.6- Portos UDP – Os mais conhecidos

O slide anterior mostra os portos UDP utilizados por aplicações populares.

## Socket

#1/3

- Conhecido por *Internet Socket*, *Berkeley Socket* ou simplesmente *Socket*
- Um socket é identificado univocamente pelo sistema operativo e agrupa 3 ou 5 elementos:
  - Um protocolo (TCP ou UDP);
  - Um endereço IP local;
  - Uma porta de comunicações;
  - Um endereço IP remoto (quando a ligação está estabelecida);
  - Uma porta de comunicações remota (quando a ligação está estabelecida);
- **Socket=Protocol+IP+Port+[remote IP+remote Port]**



### 2.4.7- Socket

O slide anterior e os dois seguintes mostram o que é um socket.

## Socket

#2/3

- Os *sockets* oferecem uma interface que permite **programar** comunicações ao nível da **Camada de Transporte**;
- A comunicação entre dois programas utilizando *sockets* é muito semelhante à gestão de ficheiros (I/O):
  - A **gestão de sockets** é tratada de mesma maneira que a gestão de ficheiros;
  - Os **streams** utilizados numa operação de ficheiros (I/O) são do mesmo tipo que os utilizados em *sockets* (I/O);
- Utilizando *sockets* um programa escrito numa linguagem pode comunicar com outro programa escrito noutra linguagem (**independência da linguagem**);

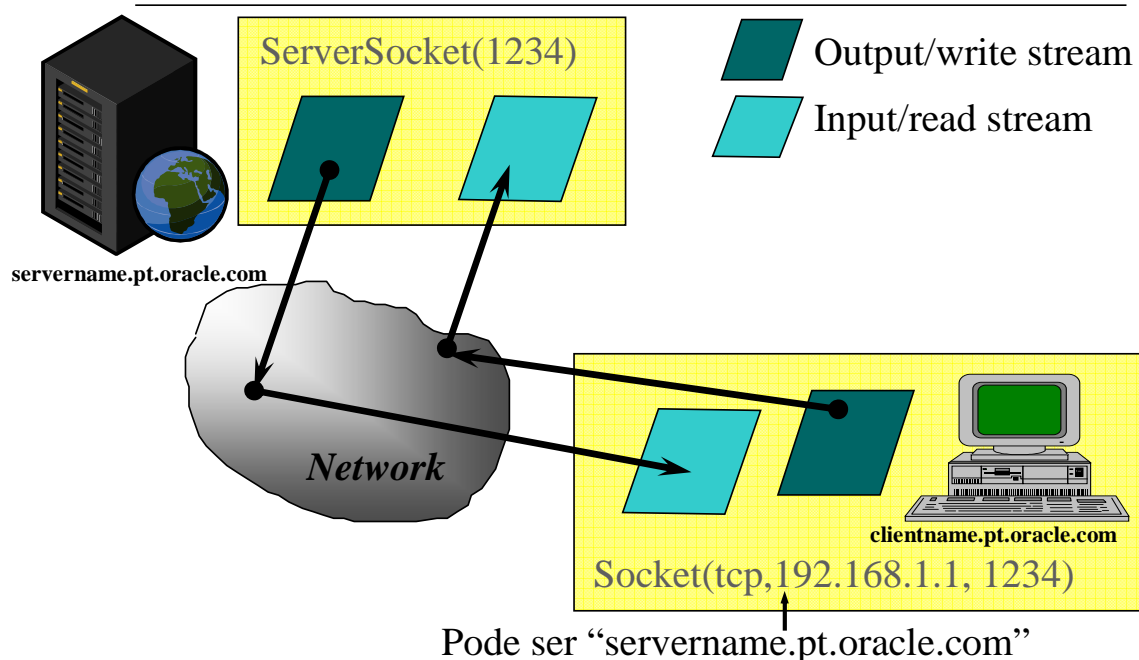
Java Web  
© Citeforma

Capítulo 2 - Sockets

21

## Socket

#3/3

Java Web  
© Citeforma

Capítulo 2 - Sockets

22

## *Sockets e Java*

- Em Java o package *net* contém as seguintes classes:
  - Para o Servidor:
    - ServerSocket - TCP;
  - Para ambos cliente e servidor:
    - Socket → TCP;
    - DatagramSocket --> UDP;



### **2.4.8- Sockets e Java**

O slide anterior mostra as classes Java mais importantes destinadas a lidar com *sockets*.

## Sumário

---

- Ambiente de trabalho;
- Modelo OSI e arquitetura TCP;
- *Sockets*;
- TCP e UDP;
- TCP – Implementar Cliente & Servidor;
- UDP – Implementar Cliente & Servidor;
- TCP – Multithreaded;
- UDP – *Multicast*.

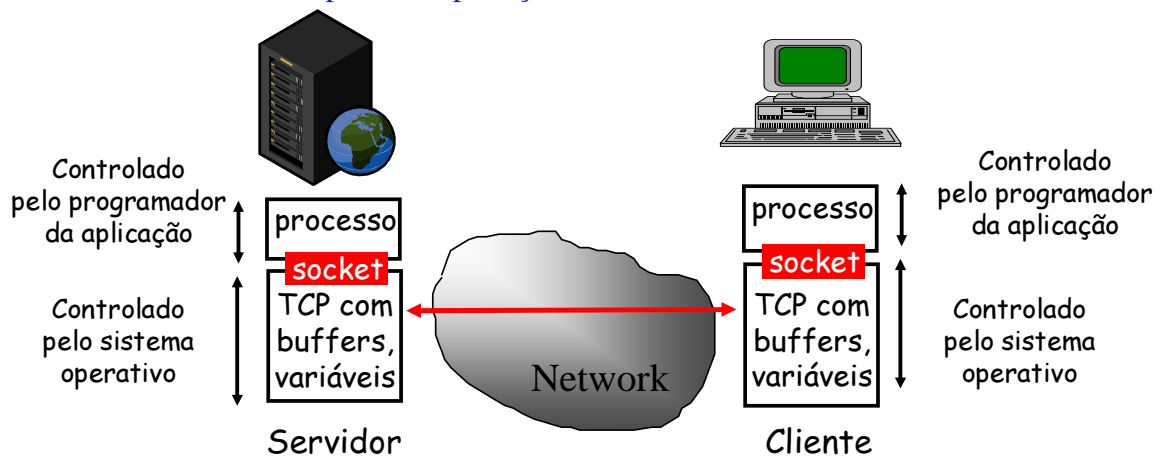


### 2.5- TCP e UDP



## TCP - *Transport Control Protocol*

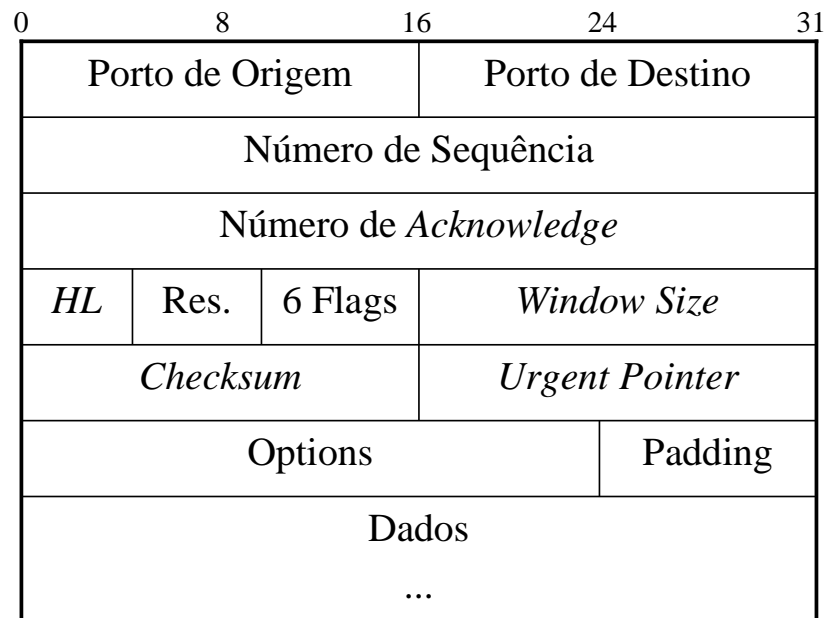
- É um protocolo de comunicação **fiável** entre dois computadores.
  - Exemplos de Aplicações: HTTP, Telnet, FTP, ...



### 2.5.1- TCP - *Transport Control Protocol*

O slide anterior mostra as características mais importantes do protocolo TCP.

## TCP: Conteúdo de um pacote



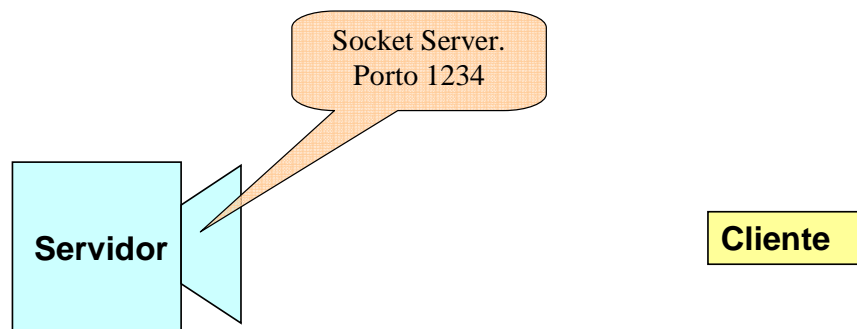
### 2.5.2- TCP: Conteúdo de um pacote

O slide anterior mostra o conteúdo de um pacote TCP.

## TCP: Comunicação entre *sockets*

#1/6

- Um programa do lado do servidor tem um processo associado que está a escutar num determinado porto;
- Este programa está à escuta de pedidos de conexão emitidos pelos clientes, sob o protocolo TCP;



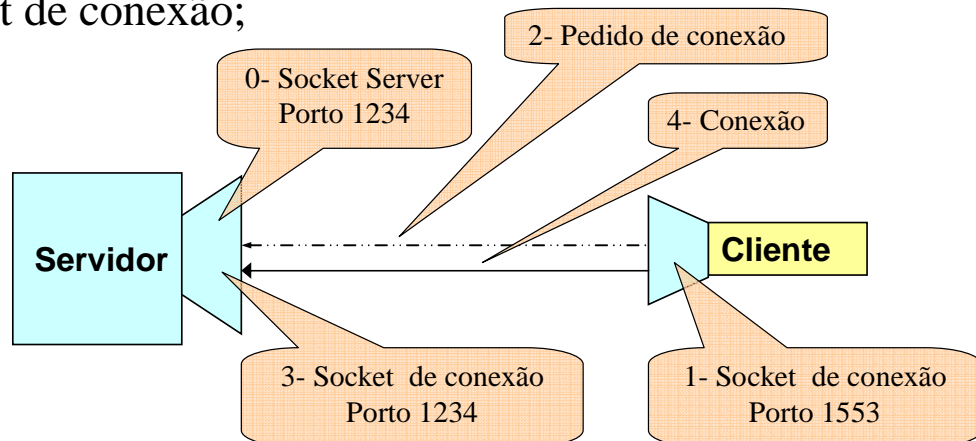
### 2.5.3- TCP: Comunicação entre *sockets*

O slide anterior e os próximos 5 mostram como funciona o protocolo TCP e como os programas que o utilizam tiram partido dos *sockets*.

## TCP: Comunicação entre *sockets*

#2/6

- O cliente abre um socket e faz um pedido de conexão ao servidor;
- O programa no servidor aceita a conexão criando um socket de conexão;

Java Web  
© Citeforma

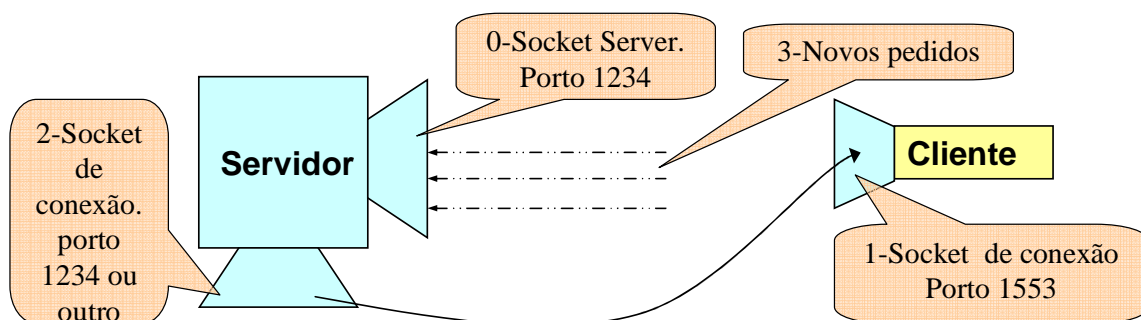
Capítulo 2 - Sockets

28

## TCP: Comunicação entre *sockets*

#3/6

- Para responder ao pedido o servidor abre outro socket e estabelece uma nova ligação dedicada para o cliente;
- O novo socket usa o mesmo porto ou um novo porto;
- O socket inicial continua à escuta de novos pedidos;

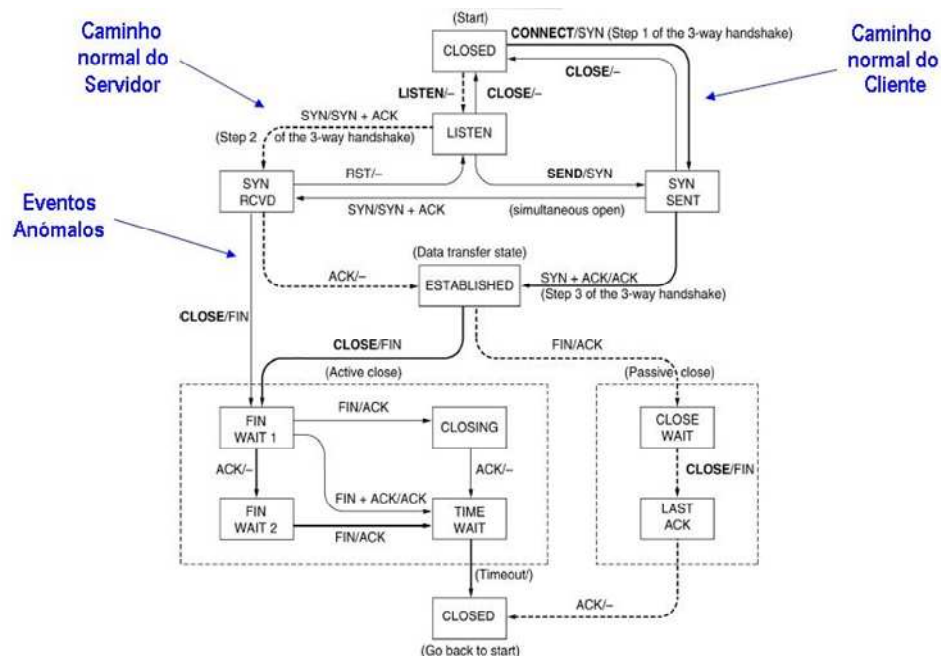
Java Web  
© Citeforma

Capítulo 2 - Sockets

29

TCP: Comunicação entre *sockets*

#4/6

TCP: Comunicação entre *sockets*

#5/6

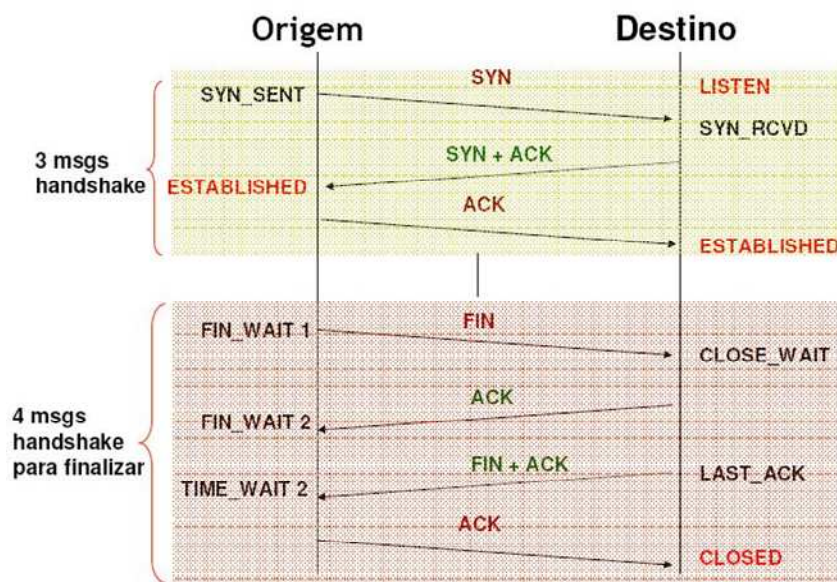
Estado	Descrição
CLOSED	Não existem conexões activas
LISTEN	O servidor espera por um pedido de conexão
SYN RCVD	O pedido de conexão foi recebido e processado; há que esperar pelo LAST ACK para estabelecer a conexão
SYN SENT	A aplicação começou a abrir uma conexão
ESTABLISHED	Conexão estabelecida, podem ser enviados dados
FIN WAIT 1	A aplicação começou a fechar uma conexão
FIN WAIT 2	O outro lado confirma que a conexão foi fechada
TIME WAIT	Está à espera dos últimos pacotes...
CLOSING	Terminar conexão
CLOSE WAIT	O outro lado inicia uma terminação de conexão
LAST ACK	Está à espera dos últimos pacotes...



## TCP: Comunicação entre *sockets*

#6/6

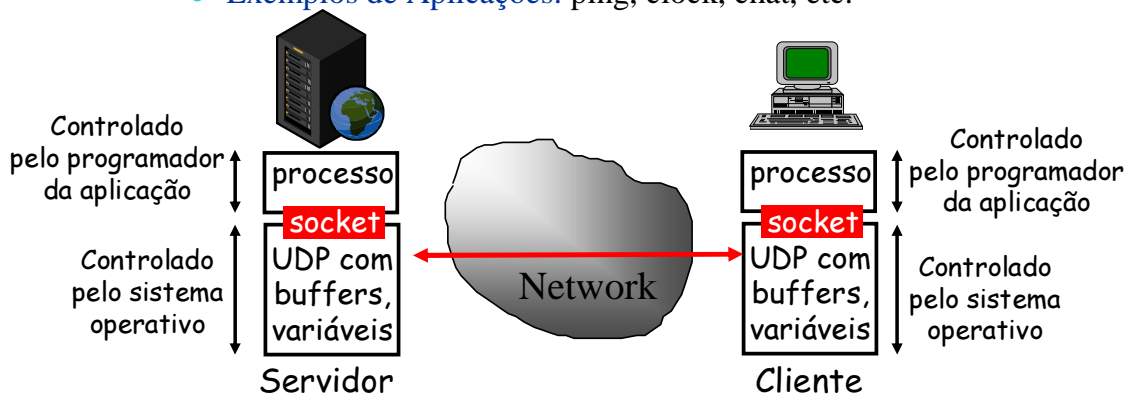
Em Resumo:



## UDP - *User Datagram Protocol*

- É um protocolo de comunicação que envia pacotes de dados independentes (*datagrams*) de um computador para outro, **sem qualquer garantia** de que chegam ao seu destino.

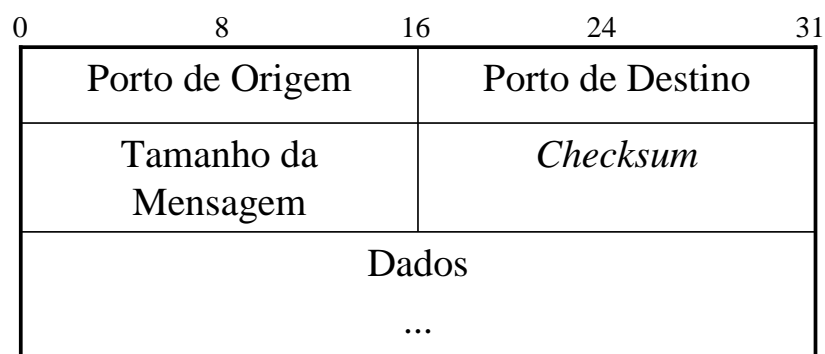
- Exemplos de Aplicações: ping, clock, chat, etc.



### 2.5.4- UDP - *User Datagram Protocol*

O slide anterior mostra as características principais do protocolo UDP.

## UDP: *Datagram*



### 2.5.5- UDP: *Datagram*

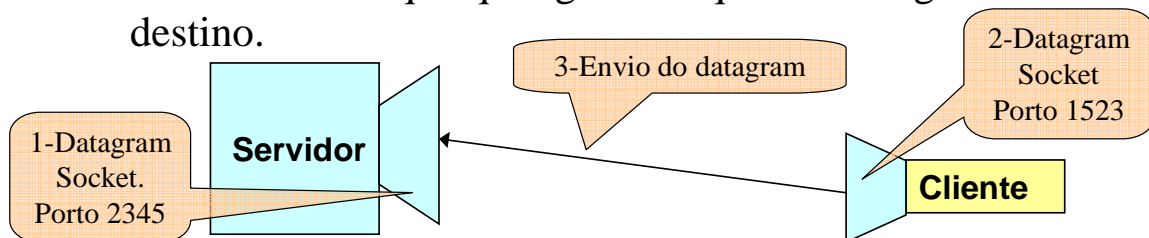
O slide anterior mostra a composição de um *datagram* UDP.



## UDP: Comunicação entre *sockets*

#1/2

- Um programa do lado do servidor tem um processo associado que está a escutar num determinado porto, esperando pedidos UDP;
- O programa fica à escuta que um cliente lhe envie um *datagram*;
- O cliente abre um *socket* para enviar o *datagram* ao servidor – sem qualquer garantia que este chegue ao seu destino.



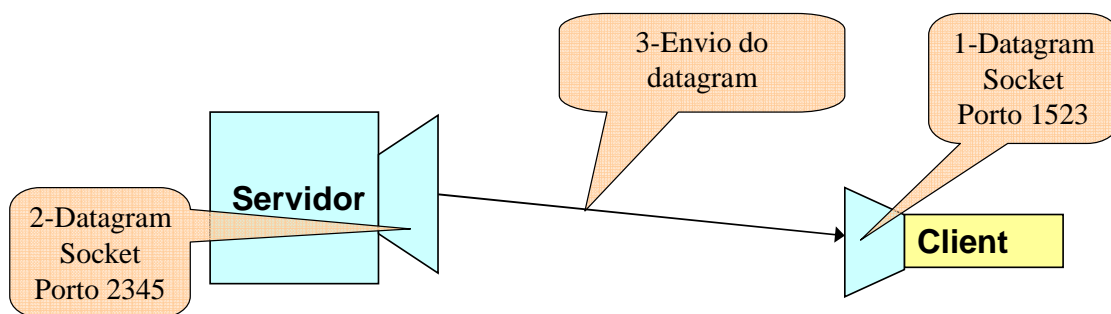
### 2.5.6- UDP: Comunicação entre *sockets*

O slide anterior e o próximo descrevem a sequência de passos envolvidos numa comunicação UDP.

## UDP: Comunicação entre *sockets*

#2/2

- O programa do lado do servidor extrai a informação do *datagram*, altera-a (ou faz o que tem a fazer) e envia a resposta ao cliente;
- A informação sobre o cliente encontra-se dentro do *datagram* recebido. Mais uma vez não existe qualquer garantia de que o cliente receba a resposta;



## TCP vs UDP

### TCP

“Keep it safe!”

#### Vantagens:

- Ligação full-duplex e ponto-a-ponto – conexão fiável entre o servidor e o cliente (*handshake*).
- Não há desordenação e duplicação de mensagens.
- Numa rede onde exista perda de pacotes.

#### Desvantagens:

- Perde-se algum tempo a estabelecer a ligação ponto-a-ponto (*handshake*);
- Podem ficar ligações “penduradas”, caso não sejam fechadas correctamente.

### UDP

“Keep it simple!”

#### Vantagens:

- *Overhead* pequeno : não há *handshake* de conexão/finalização.
- Não há necessidade de salvar estados de transmissão entre o servidor e cliente.
- Diminui o tempo de latência.
- Possível utilização de *multicast*

#### Desvantagens:

- Pode haver perda, desordenação e duplicação de mensagens.
- Existe um limite no tamanho das mensagens.



### 2.5.7- TCP vs UDP

O slide anterior compara os dois protocolos, descrevendo as suas vantagens e inconvenientes.

## Sumário

---

- Modelo OSI e arquitectura TCP;
- *Sockets*;
- TCP e UDP;
- Ambiente de trabalho;
- TCP – Implementar Cliente & Servidor;
- UDP – Implementar Cliente & Servidor;
- TCP – Multithreaded;
- UDP – *Multicast*.



### 2.6- TCP – Implementar Cliente & Servidor

## TCP: Programação em Java

### Servidor

1 - Cria socket no porto=1234 e **espera pedidos**  
`socketServidor = serverSocket(1234);`

3 - Recebe pedido, cria conexão e canais I/O  
`socketConexao = socketServidor.accept();`  
`doCliente = new BufferedReader (...);`  
`paraCliente = new DataOutputStream(...);`

6 - Recebe frase  
`frase = doCliente.readLine();`

7 - Transforma e envia frase  
`paraCliente.writeBytes(fraseAlterada);`

10 - Fecha conexão  
`socketConexao.close();`

### Cliente

2 - Cria socket ligado ao porto do server 1234  
`socketCliente = new Socket("localhost", 1234);`

4 - Cria canais de I/O  
`paraServidor = new DataOutputStream(...);`  
`doServidor = new BufferedReader (...);`

5 - Envia pedido  
`paraServidor.writeBytes(frase + '\n');`

8 - Lê a resposta  
`fraseAlterada = doServidor.readLine();`

9 - Fecha conexão  
`socketCliente.close();`



Java Web  
© Citeforma

Capítulo 2 - Sockets

39

### 2.6.1- TCP: Programação em Java

O slide anterior mostra as principais etapas utilizadas na programação em Java de um Servidor e de um cliente TCP, assim como a sua interação ao longo do tempo. A linha temporal é vertical.

## TCP - Servidor em Java

#1/2

```
import java.io.*;
import java.net.*;
public class TCPServidor {
    public static void main(String argv[ ]) throws Exception {
        String frase;    String fraseAlterada;

        ServerSocket socketServidor = new ServerSocket(1234);
        while(true) {
```

Cria um  
ServerSocket à escuta  
no porto 1234

Detecta um pedido  
de um cliente e  
responde criando  
um socket de  
conexão

```
            Socket conexaoSocket = socketServidor.accept();
```

```
            BufferedReader doCliente = new BufferedReader(
                new InputStreamReader(
                    conexaoSocket.getInputStream()));
```

Cria um canal de  
entrada de dados  
agarrado ao socket  
de conexão



Java Web  
© Citeforma

Capítulo 2 - Sockets

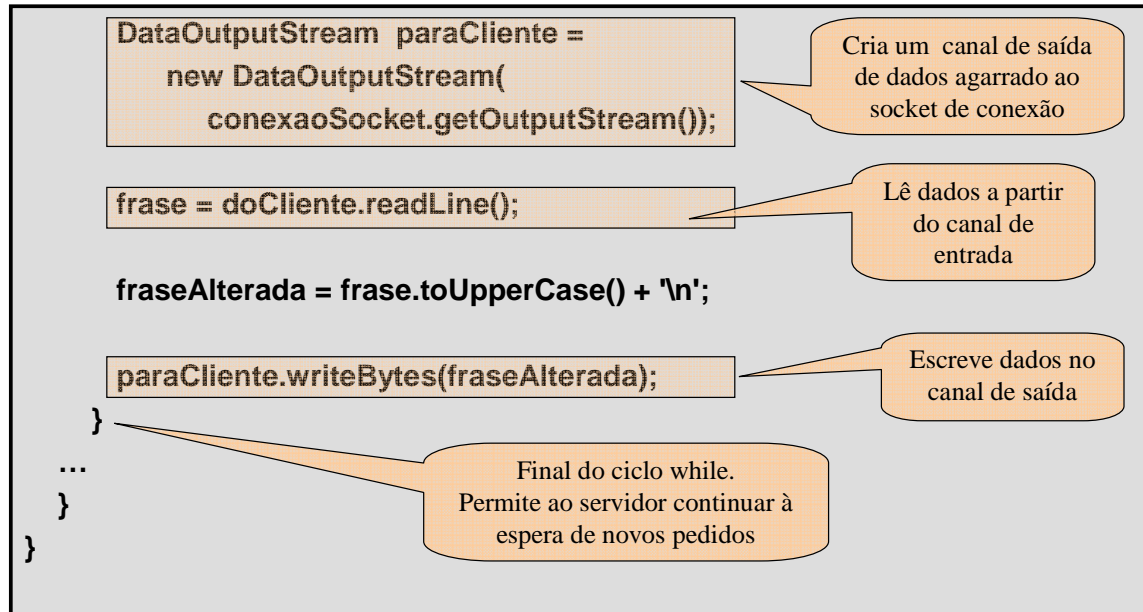
40

### 2.6.2- TCP – Servidor em Java

O slide anterior e o próximo descrevem a classe que implementa um servidor TCP usando a linguagem Java.

## TCP - Servidor em Java

#2/2



## TCP - Cliente em Java

#1/2

```
import java.io.*;
import java.net.*;
public class TCPCliente {
    public static void main(String argv[ ]) {
        String frase;  String fraseAlterada;

        frase = javax.swing.JOptionPane.showInputDialog(
            "Qual a mensagem?");

        Socket socketCliente = new Socket("localhost", 1234);

        DataOutputStream paraServidor =
            new DataOutputStream(socketCliente.getOutputStream());
```

Cria a mensagem a ser enviada

Cria um socket ligado ao servidor

Cria um output-stream agarrado ao socket



Java Web  
© Citeforma

Capítulo 2 - Sockets

42

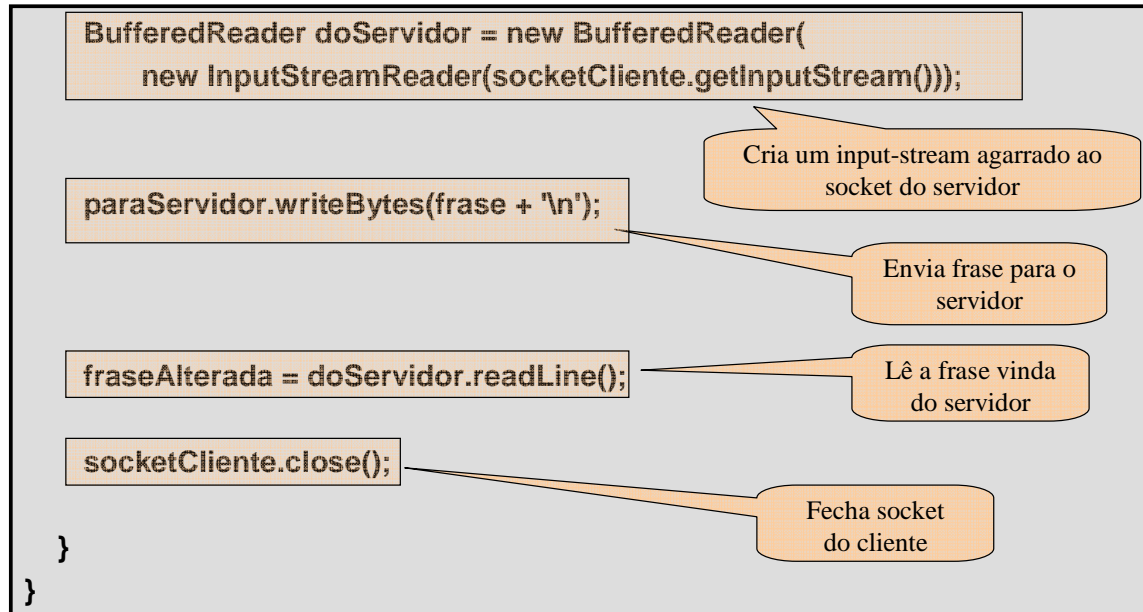
### 2.6.3- TCP – Cliente em Java

O slide anterior e o próximo descrevem a classe que implementa um cliente Java TCP, que vai comunicar com o servidor descrito no passo anterior.



## TCP - Cliente em Java

#2/2



## Ver conexões de rede

- Output de netstat com servidor à espera de pedidos:

```
C:\Documents and Settings\jaser>netstat -a -b -p TCP
Active Connections
  Proto Local Address           Foreign Address         State       PID
  TCP   aser:1234               0.0.0.0:0               LISTENING   2176
  [java.exe]
```

- Output de netstat depois de execução de cliente:

```
C:\Documents and Settings\jaser>netstat -a -b -p TCP
Active Connections
  Proto Local Address           Foreign Address         State       PID
  TCP   aser:1234               0.0.0.0:0               LISTENING   2176
  [java.exe]
  TCP   aser:1234               localhost:1094           FIN_WAIT_2   2176
  [java.exe]
  TCP   aser:1094               localhost:1234           CLOSE_WAIT   844
  [java.exe]
```



### 2.6.4- Ver conexões de rede

Em Windows podemos usar o comando netstat para verificar o estado das conexões TCP/IP. Após iniciar o servidor execute o comando netstat com as opções indicadas no slide e obterá um output semelhante, após eliminar as linhas que não interessam.

A opção `-a` destina-se a listar todos os portos (e não apenas os que têm ligações estabelecidas). A opção `-b` lista o porto de origem e o processo que o reservou. A opção `-p` filtra o protocolo (neste caso queremos apenas TCP).

Podemos verificar que o processo Java com o PID 2176 reservou e está à escuta na porta 1234.

Após arrancar com o cliente verificamos que outro processo Java, com o PID 844 estabeleceu comunicação com o porto 1234, estando a usar o porto 1094. O processo server estabeleceu comunicação com este novo processo, usando o porto 1234, ao mesmo tempo que continua à escuta no porto 1234.

## TCP – Cliente e Servidor

### Exercício:

- Definir a classe TCPServidor que implementa um servidor de pedidos TCP;
- Definir a classe TCPCliente que implementa um cliente TCP;
- Usar comando netstat -a -b -p TCP para ver como evoluem as conexões;



### 2.6.5- Exercício – Servidor e Cliente

O próximo exercício consiste em desenvolver as classes que implementam o servidor e o cliente TCP.

Segue o código da classe TCPServidor:



```
package sockets;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;

import java.net.ServerSocket;
import java.net.Socket;

public class TCPServidor {

    public static void main(String[] argv) {

        // ClienteTCP          ServidorTCP
        // frase                -----> frase
        //                      |
        //                      v
        // fraseAlterada <----- fraseAlterada
        String frase;
        String fraseAlterada;
```

```

System.out.println("SERVIDOR");

try {
    // Criar um socket para comunicacao com os clientes.
    // O servidor tem de estar sempre a escutar no mesmo socket.
    ServerSocket socketServidor = new ServerSocket(1234);

    // Mostrar o porto aberto no servidor associado a este socket
    System.out.println("Porto do Servidor : " +
        socketServidor.getLocalSocketAddress());

    // Ciclo criado para estar sempre 'a escuta dos clientes.
    while (true) {

        // ***** Estabelecimento de uma conexao *****
        //Quando chega um pedido do cliente vamos aceitar a ligacao.
        //Para isso criamos uma conexao aproveitando o socket que ja esta
aberto

        Socket socketConexao = socketServidor.accept();

        // Mostrar o porto de conexao entre servidor e cliente
        System.out.println("Porto de comunicacao entre servidor e cliente
: " +
            socketConexao.getLocalSocketAddress());

        // Criar um canal de input para entrada das mensagens do cliente
        // Exactamente igual ao manuseamento de ficheiros
        BufferedReader doCliente = new BufferedReader
            (new InputStreamReader(socketConexao.getInputStream()));

        // Criar um canal de output para enviar mensagens para o cliente
        // Exactamente igual ao manuseamento de ficheiros
        DataOutputStream paraCliente =
            new DataOutputStream(socketConexao.getOutputStream());

        //-----
        // Esperar pelo cliente
        //-----

        // Ler o que vem do cliente...
        frase = doCliente.readLine();

        // Mostrar a mensagem recebida do cliente...
        System.out.println("Mensagem do Cliente : " + frase);

        // Modificar a mensagem do cliente para a reenviar de volta
        fraseAlterada = frase.toUpperCase() + '\n';

        // Enviar para o cliente o que foi transformado pelo servidor...
        paraCliente.writeBytes(fraseAlterada);

        // Mostrar a mensagem enviada para o cliente...
        System.out.println("Mensagem para o Cliente : " + fraseAlterada);

        // Temos de fechar o socket de conexao, senao temos inumeros
        // sockets a ficarem em CLOSE_WAIT deixados pelo fecho da conexao
        // por parte do cliente.
        // Ao fechar, o socket de conexao fica no estado TIME_WAIT para
        // mais tarde ser fechado pelo sistema operativo:
        // 1. Go to Start | Run, type regedit, and click OK
        // 2. Navigate to: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet
        // \Services\Tcpip\Parameters
        // 3. Highlight the Parameters key, right-click and
        // select New | DWORD Value
        // 4. Type the name TcpTimedWaitDelay
        // 5. Double-click TcpTimedWaitDelay and enter a decimal value of
30

        // By default, this value is 240 seconds (4 minutes).
        socketConexao.close();
    }
}

```

```

        // Fechar a conexao ao Servidor, caso algum cliente diga : "adeus"
        if (frase.equalsIgnoreCase("adeus"))
            break;
    }

    // Fechar o socket Servidor, para libertar recursos
    socketServidor.close();
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

TCPCliente:



```

package sockets;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;

import java.net.Socket;
import java.net.UnknownHostException;

public class TCPCliente {

    public static void main(String[] argv) {

        // ClienteTCP          ServidorTCP
        // frase          -----> frase
        //                  |
        //                  V
        // fraseAlterada <----- fraseAlterada
        String frase;
        String fraseAlterada;

        System.out.println("CLIENTE");

        // Para introduzir a mensagem que ser enviada para o servidor
        frase = javax.swing.JOptionPane.showInputDialog("Qual a mensagem?");

        try {
            // Criar um socket para comunicacao com o servidor
            // O hostname e porto do servidor sao: localhost:1234
            Socket socketCliente = new Socket("localhost", 1234);

            // Criar um canal de output do cliente para o servidor
            // Exactamente igual ao manuseamento de ficheiros
            DataOutputStream paraServidor =
                new DataOutputStream(socketCliente.getOutputStream());

            // Mostrar o porto aberto no cliente para enviar os dados para o
servidor
            System.out.println("TCP - Porto do cliente : " +
                socketCliente.getLocalSocketAddress());

            // Criar um canal de input do servidor para o cliente
            // Exactamente igual ao manuseamento de ficheiros
            BufferedReader doServidor = new BufferedReader
                (new InputStreamReader(socketCliente.getInputStream()));

            // Enviamos a frase para o servidor...
            paraServidor.writeBytes(frase + '\n');

            // Mostrar a mensagem enviada para o servidor...

```

```
        System.out.println("Mensagem para o Servidor : " + frase);

        // -----
        // Aguardar pelo servidor
        // -----

        // Ler a resposta do servidor
        fraseAlterada = doServidor.readLine();

        // Escrevemos para output o que vem do servidor
        System.out.println("Mensagem do Servidor      : " + fraseAlterada);

        System.out.println("Vou parar por 120 segundos antes de fechar o
socket.");
        System.out.println("Veja o output de netstat -a -b -p TCP");
        Thread.sleep(120000);

        // Fechar o socket do cliente...
        socketCliente.close();

    } catch (InterruptedException e) {
        System.out.println("Erro no sleep");
        e.printStackTrace();
        System.exit(1);
    } catch (UnknownHostException e) {
        System.err.println("Erro desconhecido no servidor");
        e.printStackTrace();
        System.exit(1);
    } catch (IOException e) {
        System.err.println("Erro de I/O");
        e.printStackTrace();
        System.exit(1);
    }
}
```



O código contém comentários complementares aos que foram apresentados nos slides.



Após arrancar com o programa servidor e antes e de arrancar com o cliente, execute o comando `netstat -a -b -p TCP` para verificar o estado das conexões TCP. Procure o processo Java que está a reservar o porto 1234 e registo o respectivo PID (Process ID).



Depois de executar o cliente, repita o comando `netstat` anterior e verifique que:

- O processo servidor continua à escuta no porto 1234;
- O processo servidor estabeleceu uma comunicação com o processo cliente;
- O processo cliente estabeleceu comunicação com o servidor;
- Os portos de comunicações usados por cliente e servidor estão cruzados;

## Sumário

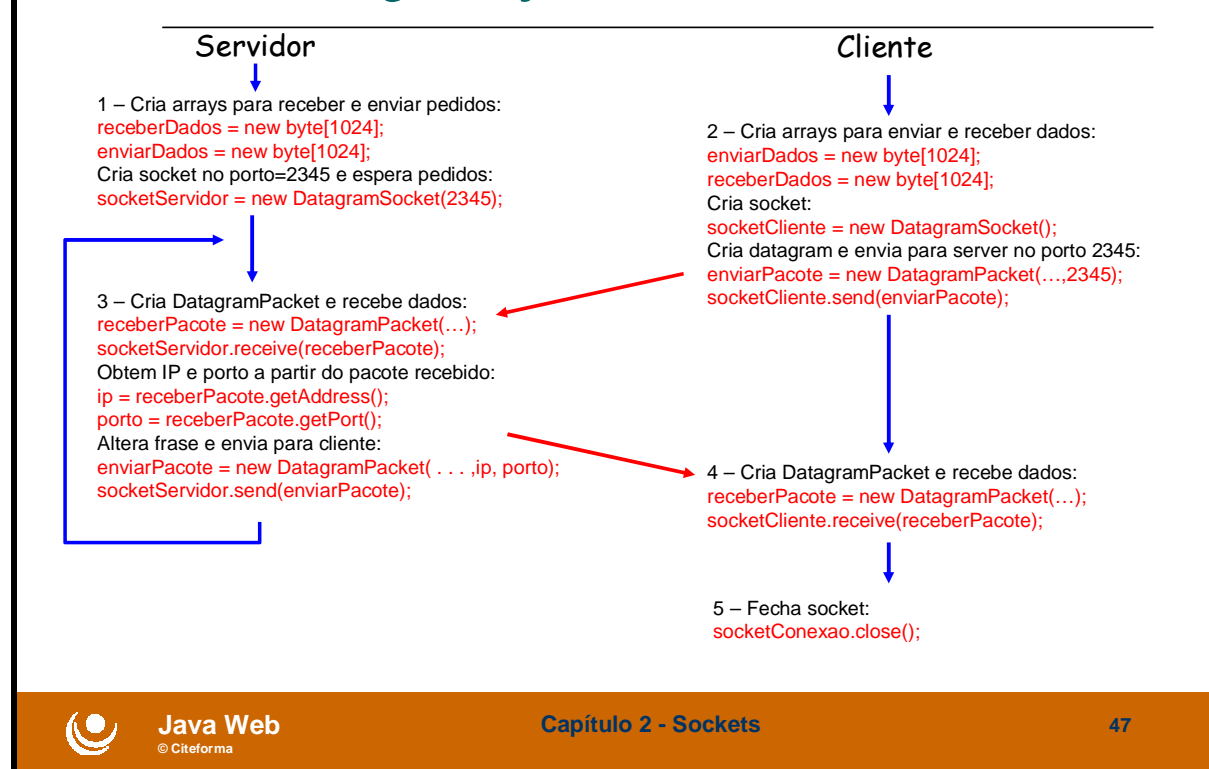
---

- Ambiente de trabalho;
- Modelo OSI e arquitetura TCP;
- *Sockets*;
- TCP e UDP;
- TCP – Implementar Cliente & Servidor;
- UDP – Implementar Cliente & Servidor;
- TCP – Multithreaded;
- UDP – *Multicast*.



### 2.7- UDP – Implementar Cliente & Servidor

## UDP: Programação em Java



### 2.7.1- UDP: Programação em Java

O slide anterior mostra as principais etapas utilizadas na programação em Java de um Servidor e de um cliente UDP, assim como a sua interação ao longo do tempo. A linha temporal é vertical.

Quando comparado com o diagrama que descreve o processo TCP pode verificar que há menos interação entre cliente e servidor, pois o UDP não garante a comunicação entre os pares.



## UDP - Servidor em Java

#1/2

```
import java.io.*; import java.net.*;
public class UDPServidor {
    public static void main(String args[] ) {
        String frase; String fraseAlterada;
        byte[] receberDados = new byte[1024];
        byte[] enviarDados = new byte[1024];
        DatagramSocket socketServidor = new DatagramSocket(2345);

        while(true) {

            DatagramPacket receberPacote = new DatagramPacket(
                receberDados, receberDados.length);

            socketServidor.receive(receberPacote);
```

Dimensiona os arrays que recebem e enviam dados

Cria um DatagramSocket à escuta no porto 2345

Cria espaço para receber um datagram usando o array

Recebe o datagram



Java Web  
© Citeforma

Capítulo 2 - Sockets

48

### 2.7.2- UDP – Servidor em Java

O slide anterior e o próximo descrevem a classe que implementa um servidor UDP em Java.

## UDP - Servidor em Java

#2/2

```
frase = new String(receberPacote.getData());
```

Obter o IP e porto do cliente

```
InetAddress EnderecoIP = receberPacote.getAddress();  
int porto = receberPacote.getPort();
```

```
fraseAlterada = frase.toUpperCase();  
enviarDados = fraseAlterada.getBytes();
```

Criar o datagram para  
enviar ao cliente

```
DatagramPacket enviarPacote = new DatagramPacket(  
    enviarDados, enviarDados.length, EnderecoIP, porto);
```

```
socketServidor.send(enviarPacote);
```

Escrever o datagram  
para o socket

```
}
```

Ciclo While. Fica em ciclo à  
espera de novos datagrams

## UDP - Cliente em Java

#1/2

```
import java.io.*; import java.net.*;
public class UDPCliente {
    public static void main(String args[ ]) throws Exception{
        String frase;    String fraseAlterada;
        frase = javax.swing.JOptionPane.showInputDialog("Qual a mensagem?");

        byte[] enviarDados = new byte[1024];
        byte[] receberDados = new byte[1024];

        DatagramSocket socketCliente = new DatagramSocket();

        InetAddress enderecoIP = InetAddress.getByName("localhost");
```

Frase a enviar ao servidor

Arrays a usar nos datagrams

Cria socket para o cliente

Traduzir o nome do servidor para endereço IP, utilizando o DNS

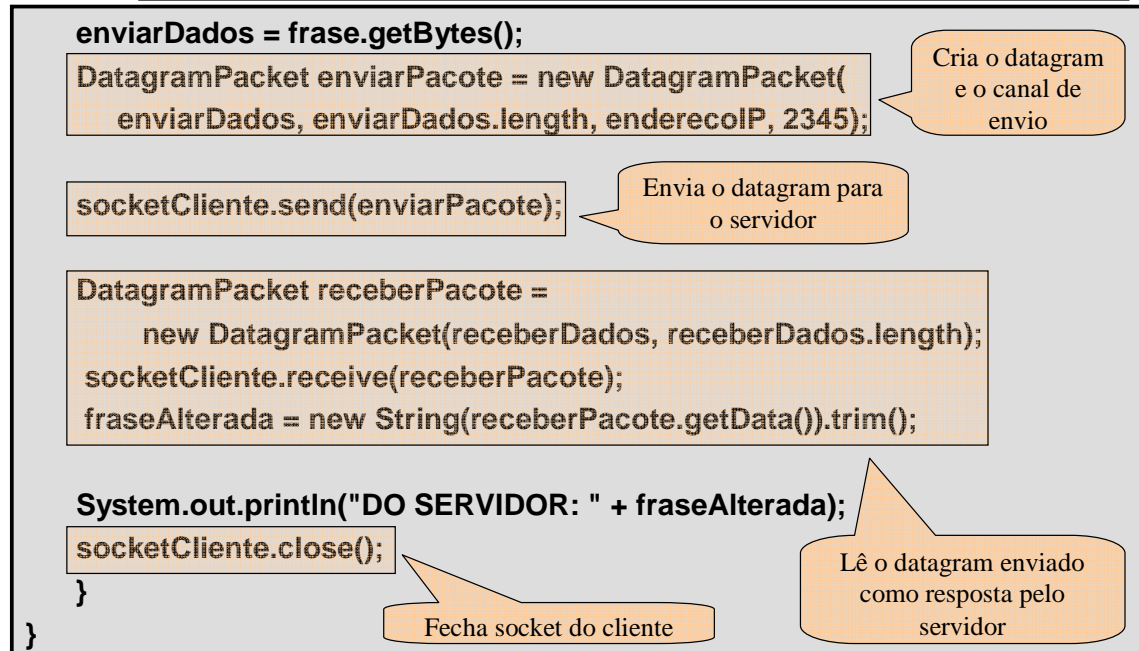


### 2.7.3- UDP - Cliente em Java

O slide anterior e o próximo descrevem a classe que implementa um cliente UDP em Java.

## UDP - Cliente em Java

#2/2



## Ver conexões de rede

- Output de netstat com servidor à espera de pedidos:

```
C:\Documents and Settings\jaser>netstat -a -b -p UDP
Active Connections
Proto Local Address          Foreign Address         State               PID
UDP   aser:2345                *:*                     *                   1092
[java.exe]
```

- Output de netstat depois de execução de cliente:

```
C:\Documents and Settings\jaser>netstat -a -b -p UDP
Active Connections
Proto Local Address          Foreign Address         State               PID
UDP   aser:1149                *:*                     *                   3152
[java.exe]
UDP   aser:2345                *:*                     *                   1092
[java.exe]
```



### 2.7.4- Ver conexões de rede

De forma semelhante com o que fizemos em TCP, vamos agora usar o comando netstat para ver a evolução da comunicação e alocação de portas entre cliente e servidor.

## UDP – Cliente e Servidor

### Exercício:

- Definir a classe UDPServidor que implementa um servidor de pedidos UDP;
- Definir a classe UDPCliente que implementa um cliente UDP;
- Usar comando netstat -a -b -p UDP para ver como evoluem as conexões;



### 2.7.5- Exercício – Servidor e Cliente

O próximo exercício envolve o desenvolvimento de uma classe que implementa um Servidor e um Cliente UDP.

Classe UDPServidor:



```
package sockets;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;

public class UDPServidor {

    public static void main(String[] args) {

        // ClienteUDP                      ServidorUDP
        // frase          -----> frase
        //                |
        //                V
        // fraseAlterada <----- fraseAlterada
        String frase;
        String fraseAlterada;

        System.out.println("SERVIDOR");

        // Criar arrays de bytes para receber e enviar os datagrams
```

```
byte[] receberDados = new byte[1024];
byte[] enviarDados = new byte[1024];

try {
    // Cria um socket para comunicacao com os clientes
    DatagramSocket socketServidor = new DatagramSocket(2345);

    // Mostra o porto aberto no servidor associado a este socket
    System.out.println("UDP - Porto do servidor : " +
        socketServidor.getLocalSocketAddress());

    // Ciclo criado para estar sempre 'a escuta dos clientes
    while (true) {

        // Cria o pacote que vai receber o datagram enviado pelo cliente
        DatagramPacket receberPacote =
            new DatagramPacket(receberDados, receberDados.length);

        // Recebe o datagram enviado do cliente para o servidor
        socketServidor.receive(receberPacote);

        // Converte dados de byte em string
        frase = new String(receberPacote.getData());

        // Mostra os dados que vieram do cliente
        System.out.println("Mensagem do Cliente      : " + frase);

        // Obtem o IP e porto do cliente a partir do pacote que foi enviado
        pelo cliente
        // Isto e' necessario para depois sabermos para onde enviar o
        pacote de resposta
        InetAddress enderecoIP = receberPacote.getAddress();
        int porto = receberPacote.getPort();
        System.out.println("Endereco IP do Cliente: " + enderecoIP);
        System.out.println("Porto do Cliente      : " + porto);

        // Modifica a mensagem do cliente (fazer o trabalho de servidor)
        fraseAlterada = frase.toUpperCase();

        // Converte mensagem a enviar em bytes
        enviarDados = fraseAlterada.getBytes();

        // Cria o pacote a enviar para o cliente
        DatagramPacket enviarPacote = new DatagramPacket( enviarDados,
            enviarDados.length, enderecoIP, porto);

        // Envia o pacote do servidor para o cliente
        socketServidor.send(enviarPacote);

        // Mostra os dados enviados no pacote para o cliente...
        System.out.println("Mensagem para o Cliente : " + fraseAlterada);

        // Fecha o socketServer caso o cliente envie "adeus"
        if (frase.equalsIgnoreCase("adeus"))
            break;
    }

    // Fecha o socketServer
    socketServidor.close();

} catch (SocketException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Classe UDPCliente:



```
package sockets;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;

public class UDPCliente {

    public static void main(String[] args) {

        // ClienteUDP                ServidorUDP
        // frase            -----> frase
        //                  |
        //                  V
        // fraseAlterada <----- fraseAlterada
        String frase; String fraseAlterada;
        // Para introduzir a mensagem que sera' enviada ao servidor
        frase = javax.swing.JOptionPane.showInputDialog("Qual a mensagem?");

        System.out.println("CLIENTE");

        // Cria arrays de bytes para enviar e receber os datagrams
        byte[] enviarDados = new byte[1024];
        byte[] receberDados = new byte[1024];

        try {
            // Cria um socket para comunicacao com o servidor
            DatagramSocket socketCliente = new DatagramSocket();

            // Obtem o IP do servidor atraves do DNS
            // Neste caso e' o localhost, pois temos o servidor e o cliente na
            // mesma maquina
            InetAddress enderecoIP = InetAddress.getByName("localhost");

            // Converte string em bytes
            enviarDados = frase.getBytes();

            // Cria o datagram que vai ser enviado ao servidor
            DatagramPacket enviarPacote = new DatagramPacket(
                enviarDados, enviarDados.length, enderecoIP, 2345);
            // Mostra o porto aberto no cliente para enviar os dados para o
servidor

            System.out.println("UDP - Porto do Cliente: " +
                socketCliente.getLocalSocketAddress());

            // Envia o pacote do cliente para o servidor
            socketCliente.send(enviarPacote);

            // Mostra os dados enviados no pacote para o servidor
            System.out.println("Mensagem enviada ao servidor: " + frase);

            // -----
            // espera pelo servidor
            // -----

            // Cria o pacote que contem os dados enviados pelo servidor
            DatagramPacket receberPacote = new DatagramPacket(
                receberDados, receberDados.length);

            // Recebe o pacote enviado pelo servidor
```



```
        socketCliente.receive(receberPacote);

        // Converte bytes em String
        fraseAlterada = new String(receberPacote.getData()).trim();

        // Mostra os dados recebidos
        System.out.println("Mensagem enviada pelo Servidor: " + fraseAlterada);

        System.out.println("Vou parar por 120 segundos antes de fechar o
socket.");
        System.out.println("Veja o output de netstat -a -b -p UDP");
        Thread.sleep(120000);

        // Fecha o socket do cliente
        socketCliente.close();

        } catch (SocketException e) {
            e.printStackTrace();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



A explicação deste código está descrita nos slides.



O comando netstat mostra que com UDP são abertos dois portos na origem, não havendo portos de destino, pois a mensagem é broadcasted. Não há protocolo que garanta que a mensagem é recebida.

Aser – Vou aqui

## Sumário

---

- Modelo OSI e arquitetura TCP;
- *Sockets*;
- TCP e UDP;
- Ambiente de trabalho;
- TCP – Implementar Cliente & Servidor;
- UDP – Implementar Cliente & Servidor;
- TCP – Multithreaded;
- UDP – *Multicast*.



### 2.8- TCP – Multithreaded

## Vantagens de usar *Multithreaded*

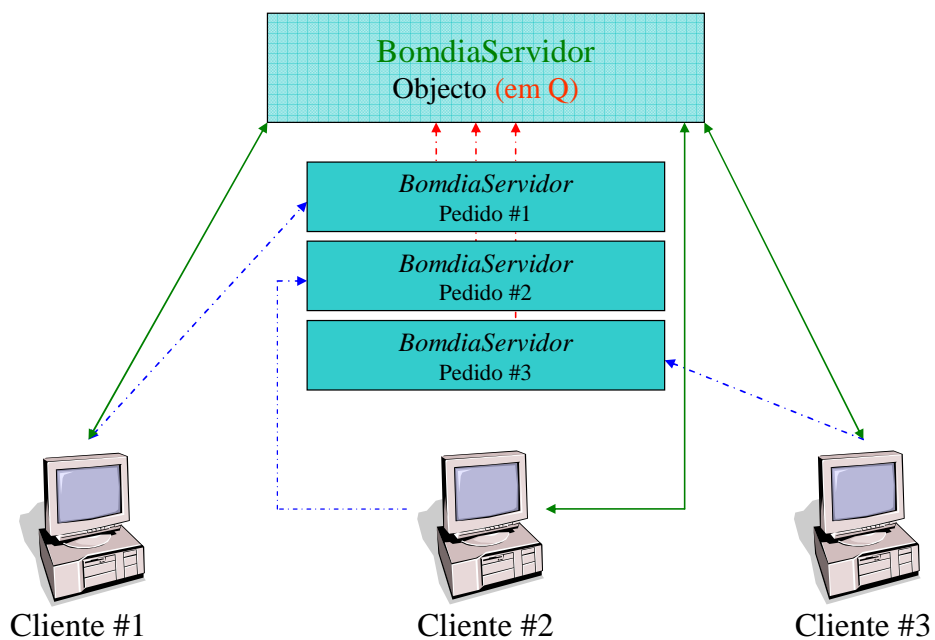
- Um programa TCP simples é desenhado para atender um único pedido de ligação;
- Se houver vários clientes estes podem colocar múltiplos pedidos de ligação ao mesmo *host* e porto (onde está à escuta o *ServerSocket*);
- Os vários pedidos vão ficar numa fila de espera, até que o servidor tenha tempo de os processar;
- O multithreading permite-nos atender estes pedidos quase em paralelo. Para isso temos que criar um server thread por cada pedido de ligação de um cliente.



### 2.8.1- Vantagens de usar *Multithreaded*

O slide anterior descreve as vantagens de usar programação *multithreaded* num servidor TCP.

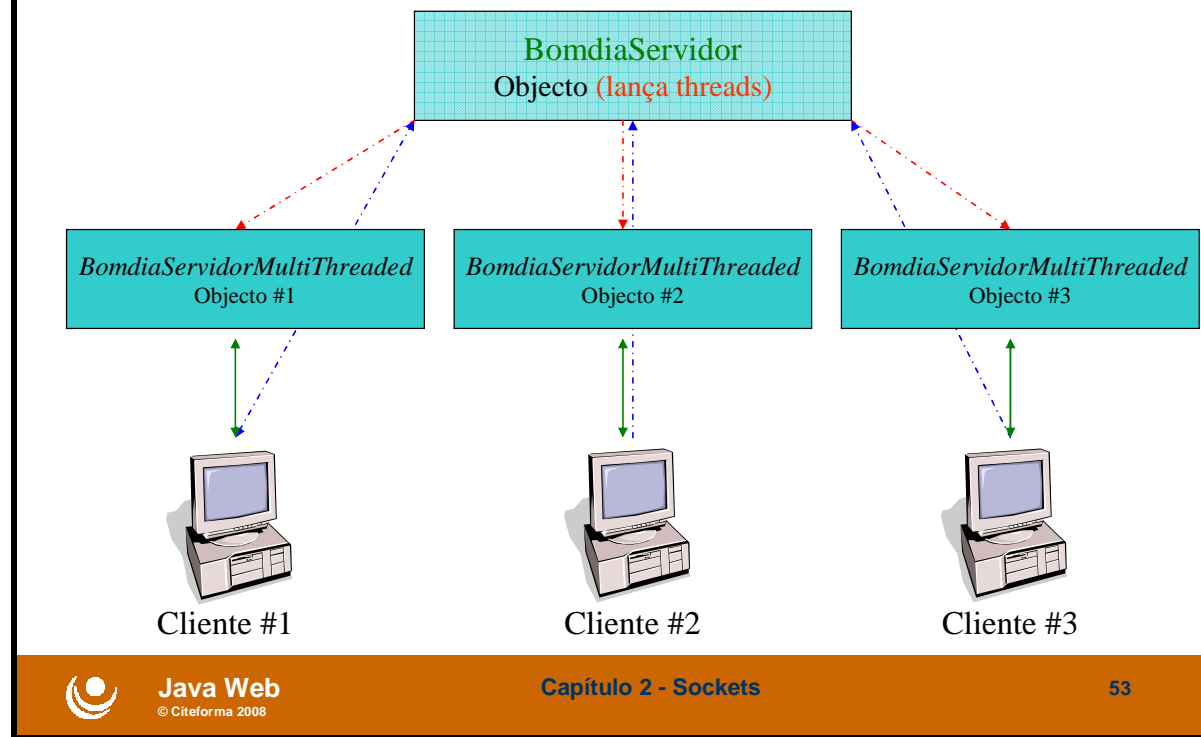
## TCP: *Non-Multithreaded*



### 2.8.2- TCP – *Non-Multithreaded*

O slide anterior mostra a serialização existente quando um servidor TCP recebe vários pedidos em simultâneo.

## TCP: *Multithreaded*



### 2.8.3- TCP: *Multithreaded*

O slide anterior mostra como um servidor *multithreaded* consegue tratar em paralelo as respostas aos vários pedidos que recebe em simultâneo.

## TCP: *Multithreaded*

### Exercício:

- Definir a classe BomdiaProtocolo que define o protocolo a seguir para cada mensagem;
- Definir a classe BomdiaServidorMultiThreaded que define um *thread* para cada pedido do cliente;
- Definir a classe BomdiaServidor que define um servidor *multithread* para que este lance *threads*;
- Definir a classe BomdiaCliente que define um cliente;



### 2.8.4- Exercício – Multithreaded

No próximo exercício vamos desenvolver 4 classes que implementam um servidor *multithreaded* e um cliente.

Classe BomdiaProtocolo:



```
package sockets;

public class BomdiaProtocolo {
    // Estados possíveis do Protocolo Bomdia
    private static final int COMECO = 0;
    private static final int BOMDIA = 1;
    private static final int DENOVO = 2;
    private static final int PISTA = 3;
    private static final int OUTRAVEZ = 4;
    private static final int FRASE = 5;

    private int estado = COMECO;
    private int frase = 0;


    private String[] pistas = { "Servidor", "Thread", "Socket", "Programa" };
    private String[] respostas = {
        "Sou o Servidor de sockets TCP!",
        "Sou um dos threads do Servidor de sockets TCP!",
        "Sou um socket do lado do Servidor de TCP!",
        "Programa de Java sobre sockets TCP em multithreading!"
    };

    public String processarInput(String input) {
```

```
String output = null;

// Começa o protocolo...
// O Servidor vai responder 1º: Bom dia!
// Fica à espera do Cliente responder...
if (estado == COMECO) {
    output = "Bom dia!";
    estado = BOMDIA;
}
// Já passámos a fase do "Bom dia!"...
else if (estado == BOMDIA)
{
    // O Ciente deve responder 1º: Quem é?
    // Responde acertadamente...
    if (input.equalsIgnoreCase("Quem é?")) {
        output = pistas[frase];
        estado = PISTA;
    }
    // Caso o Cliente não tenha respondido : "Quem é?"...
    else
    {
        output = "Deves escrever: \"Quem é?\"! " +
            "Tenta de novo...!";
    }
}
// Já passámos a fase do "Quem é?"...
else if ((estado == PISTA) || (estado == DENOVO))
{
    // O Ciente deve responder 2º: [XXXXXXXXX (pistas)] de?
    // Responde acertadamente...
    if (input.equalsIgnoreCase(pistas[frase] + " de?")) {
        output = respostas[frase] + " Queres tentar de novo? (s/n)";
        estado = OUTRAVEZ;
    }
    // Caso o Cliente não tenha respondido : "[XXXXXXXXX (pistas)] de?"...
    else
    {
        output = "Deves escrever: \"" +
            pistas[frase] +
            " de?\"! Tenta de novo...!";
        estado = DENOVO;
    }
}
// O Ciente deve responder 3º: s/n
else if (estado == OUTRAVEZ)
{
    // Ciente responde 3º: s ou S
    if (input.equalsIgnoreCase("s")) {
        output = "Bom dia!";
        if (frase == (FRASE - 1))
            frase = 0;
        else
            frase++;
        estado = BOMDIA;
    }
    // Ciente responde 3º: n ou N
    else
    {
        // Esta mensagem é a mesma para "destruir" o Cliente:
        // É apanhado no: BomdiaCliente.java
        output = "Adeus! Até já...!";
        estado = COMECO;
    }
}
return output;
}
```

Classe BomdiaServidorMultithreaded:



```
package sockets;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

import java.net.Socket;

public class BomdiaServidorMultiThreaded extends Thread {
    private Socket socketServidor = null;

    public BomdiaServidorMultiThreaded(Socket socket) {
        super("BomdiaServidorMultiThreaded");
        this.socketServidor = socket;
    }

    public void run() {
        try {
            // Os dois streams...
            PrintWriter out =
                new PrintWriter(socketServidor.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(
                    socketServidor.getInputStream()));

            // Processamento de input e output por parte do Servidor...
            String linhaInput, linhaOutput;

            // O processamento vai seguir um determinado protocolo...
            BomdiaProtocolo bdp = new BomdiaProtocolo();

            // Iniciar processamento...
            // Primeiro contacto com o Cliente...
            linhaOutput = bdp.processarInput(null);
            out.println(linhaOutput);

            while ((linhaInput = in.readLine()) != null) {
                linhaOutput = bdp.processarInput(linhaInput);
                out.println(linhaOutput);
                if (linhaOutput.equalsIgnoreCase("Adeus! Até já...!"))
                    break;
            }

            // Fechar os dois streams, para libertar recursos...
            out.close();
            in.close();

            // Fechar o socket Servidor, para libertar recursos...
            socketServidor.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Classe BomdiaServidor:



```
package sockets;
```



```

import java.io.IOException;

import java.net.ServerSocket;

public class BomdiaServidor {
    public static void main(String[] args) {
        ServerSocket socketServidor = null;
        boolean aEscutar = true;

        // Tentativa de escutar no porto 2222...
        // Pode ser que esteja ocupado pelo que vai dar erro.
        try {
            socketServidor = new ServerSocket(2222);
        } catch (IOException e) {
            System.err.println("Não é possível escutar no porto: 2222.");
            System.exit(-1);
        }

        // Sendo possivel escutar no porto 2222, vamos escutar...
        try {
            while (aEscutar) {
                new BomdiaServidorMultiThreaded(socketServidor.accept()).start();
            }

            // Fechar o socket Servidor, para libertar recursos...
            socketServidor.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Classe BomdiaCliente:



```

package sockets;

import java.io.*;
import java.net.*;

public class BomdiaCliente {
    public static void main(String[] args) {

        Socket socketCliente = null;
        PrintWriter out = null;
        BufferedReader in = null;

        // Tentativa de ligação ao Servidor Bomdia...
        try {
            socketCliente = new Socket("localhost", 2222);
            out = new PrintWriter(socketCliente.getOutputStream(), true);
            in = new BufferedReader
                (new InputStreamReader(socketCliente.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Desconheço este servidor: localhost.");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Não consegui um ligação I/O " +
                               "ao servidor: localhost.");
            System.exit(1);
        }

        // BufferedReader stdIn = new BufferedReader
        //                               (new InputStreamReader(System.in));
        String doServidor;
        String doCliente;
    }
}

```

```
// Fica a falar com o Servidor até "Adeus! Até já...!", que provoca
// sair do ciclo... e fechar o Cliente.
try {
    while ((doServidor = in.readLine()) != null) {
        System.out.println("Servidor: " + doServidor);
        // Para "destruir" o Cliente...
        if (doServidor.equals("Adeus! Até já...!"))
            break;

        doCliente = javax.swing.JOptionPane.showInputDialog("Qual a
mensagem (do cliente)?");
        // stdIn.readLine();
        if (doCliente != null) {
            System.out.println("Cliente: " + doCliente);
            out.println(doCliente);
        }
    }

    // Fechar os dois streams, para libertar recursos...
    out.close();
    in.close();
    // stdIn.close();

    // Fechar o socket Cliente, para libertar recursos...
    socketCliente.close();
}
catch (IOException e) {
    e.printStackTrace();
}
}
```



A explicação deste código está nos comentários dentro do próprio código.

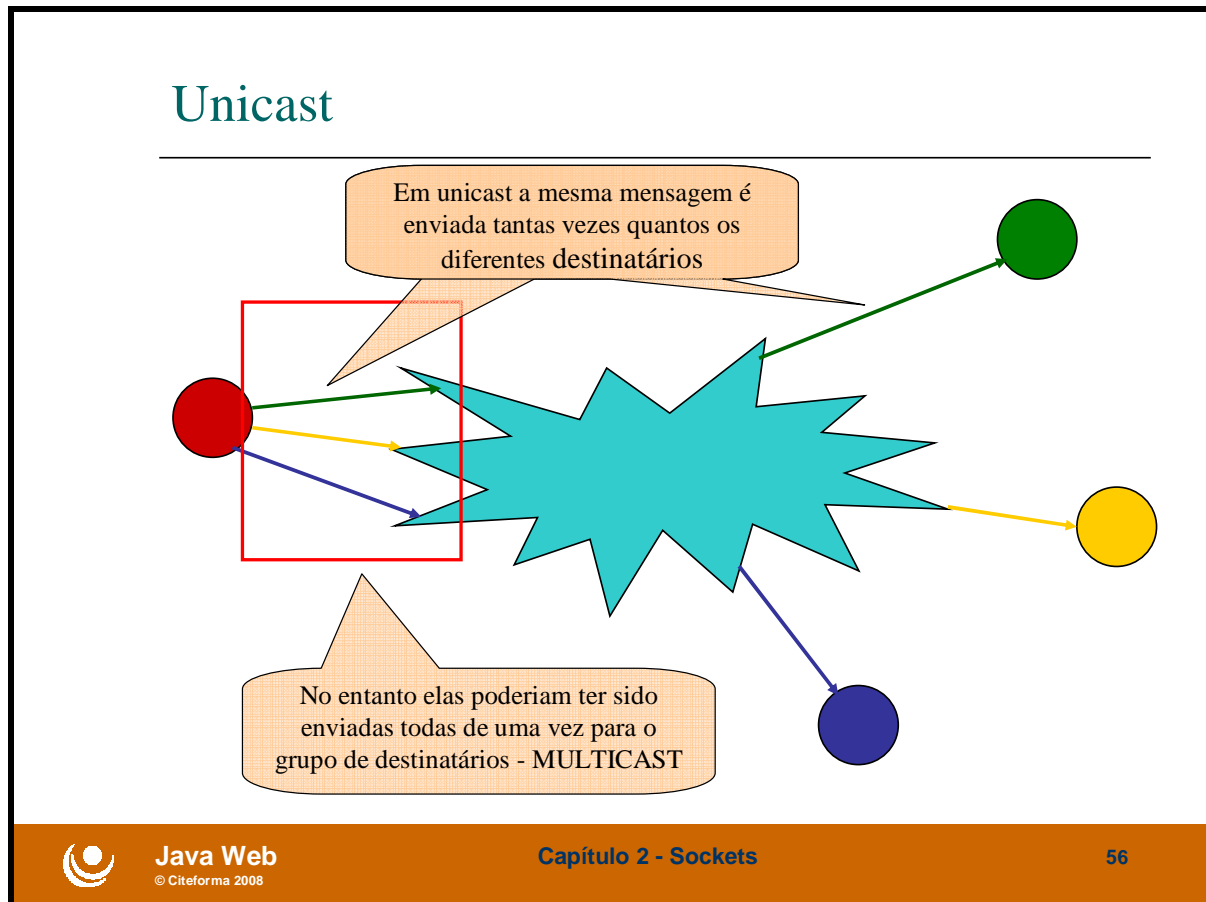
## Sumário

---

- Modelo OSI e arquitetura TCP;
- *Sockets*;
- TCP e UDP;
- Ambiente de trabalho;
- TCP – Implementar Cliente & Servidor;
- UDP – Implementar Cliente & Servidor;
- TCP – Multithreaded;
- UDP – *Multicast*.



### 2.9- Multicast



### 2.9.1- Unicast

Em condições normais as mensagens UDP são enviadas do emissor para o receptor. Se tivermos uma mensagem que se destina a vários destinatários, em condições normais ela teria que ser enviada tantas vezes quantos os diferentes destinatários. Isto é descrito por *UNICAST*, visto que há repetição de envio pois o receptor é único.

No entanto poderíamos definir um grupo de destinatários e enviar a mensagem para o grupo, uma única vez. Isso é *multicast*.

## *Multicast*

- Multicast é baseado no protocolo UDP;
- Nas aplicações baseadas em *multicast* **não existe**:
  - Controlo do esforço efectuado na entrega dos pacotes. As aplicações em *multicast* não são fiáveis;
  - Controlo do tráfego. Falta o controlo/verificação do tamanho do *buffer* do receptor para realizar a conexão, por forma a não se perder nada na comunicação entre ambos;
  - Controlo de pacotes duplicados. Alguns mecanismos do protocolo de *multicast* resultam em geração de pacotes duplicados;
  - Controlo na ordem em que os pacotes chegam ao destino. Alguns mecanismos do protocolo de *multicast* resultam na desordem ou perda de alguns pacotes.



### **2.9.2- Multicast**

O slide anterior descreve as desvantagens de usar *multicast*.

## Exemplos de endereços *Multicast*

### ○ Grupos de Endereços IP:

- Endereços: Class D → 1110 = 224.0.0.0/4
- Entre 224.0.0.0 até 239.255.255.255

IPv4 Multicast Addresses:

<http://www.iana.org/assignments/multicast-addresses>

### ○ Alguns dos mais populares endereços multicast:

- 224.0.0.1            All Systems on this subnet
- 224.0.0.2            All Routers on this subnet
- 224.0.0.13          All PIM v2 Routers
- 224.0.1.32          mtrace

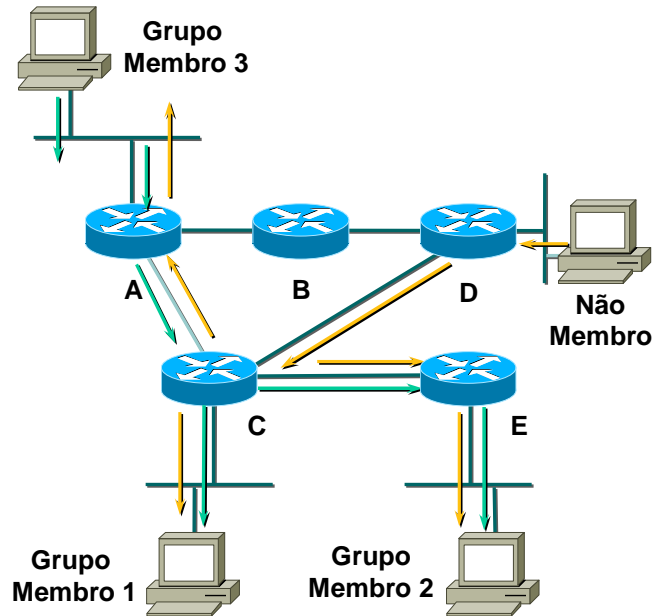


### **2.9.3- Exemplos de endereços multicast**

O slide anterior descreve exemplos de endereços *multicast*.

## Regras *Multicast*

1. Se a mensagem for enviada para um grupo de endereços, todos os seus membros vão recebê-la;
2. Tem de ser membro de um grupo para receber a mensagem;
3. Não tem de ser membro de um grupo para enviar uma mensagem a um grupo;



### 2.9.4- Regras *Multicast*

O slide anterior descreve as regras na comunicação *multicast*.

## *Multicast: Prog. em Java - Enviar*

```
String msg = "Isto é uma mensagem em multicast.";

InetAddress group = InetAddress.getByName("239.1.2.3");
MulticastSocket socket = new MulticastSocket(3456);

socket.joinGroup(group); // opcional

DatagramPacket dpsent =
    new DatagramPacket(msg.getBytes( ),msg.length( ),group, 3456);

socket.send(dpsent);

socket.leaveGroup(group);
```



### **2.9.5- Multicast: Programação em Java para enviar mensagem**

O slide anterior descreve os principais passos usados em Java para programar o envio de mensagens por *multicast*.



## *Multicast: Prog. em Java - Receber*

```
byte[ ] buf = new byte[256];  
InetAddress group = InetAddress.getByName("239.1.2.3");  
MulticastSocket socket = new MulticastSocket(3456);  
socket.joinGroup(group);  
DatagramPacket dprecv = new DatagramPacket(buf, buf.length);  
socket.receive(dprecv);  
socket.leaveGroup(group);
```



### **2.9.6- Multicast: Programação em Java para receber mensagem**

O slide anterior descreve os principais passos usados em Java para programar a recepção de mensagens por *multicast*.

## *Multicast*

### **Exercício:**

- Definir a classe MulticastServidorThreads;
- Definir a classe MulticastServidor que arranca com o servidor de threads;
- Definir a classe MulticastCliente;



### **2.9.7- Exercício – Multicast**

No próximo exercício vamos desenvolver 3 classes que simulam o envio e recepção de mensagens por *multicast*.

Classe MulticastServidorThreads:



```
package sockets;

import java.io.*;
import java.net.*;
import java.util.*;

public class MulticastServidorThreads extends Thread {

    private long ESPERA = 2000;

    protected DatagramSocket socketServidor = null;
    protected BufferedReader in = null;
    protected boolean maisCotacoes = true;
    protected int linhas = 0;

    public MulticastServidorThreads() throws IOException {
        super("MulticastServidorThreads");

        // O Servidor vai estar a escutar no porto 4445...
        socketServidor = new DatagramSocket(4445);

        // Vamos ler um ficheiro de cotações...
        // Ou caso não exista, vamos mostrar o tempo de 2 em 2 segundos...
```

```
try {
    // Ler de um ficheiro as cotações...
    in = new BufferedReader(new FileReader("cotacoes-online.txt"));
}
// Caso não exista o ficheiro de cotações...
catch (FileNotFoundException e)
{
    System.err.println("Nao foi possivel abrir o ficheiro de cotacoes.");
    System.out.println("Vamos mostrar o tempo...");
}

}

public void run() {
    // Enquanto existirem cotações vai mostrando...
    while (maisCotacoes) {
        try {
            // Buffer para a passagem da mensagem dos pacotes...
            byte[] buf = new byte[256];

            // Obter a mensagem: do ficheiro ou do tempo...
            String mensagem = null;
            if (in == null)
                // Caso não exista o ficheiro de cotações...
                // Mostra o tempo...
                mensagem = new Date().toString();
            else
            {
                // Obter a próxima cotação, do ficheiro de cotações...
                mensagem = obterProximaCotacao();

                // Quantas linhas já foram lidas...
                linhas++;
                System.out.println("Leu a linha nº "
                                    + linhas +
                                    " do ficheiro de cotações.");
            }
            // Codifica a String numa sequência de bytes utilizando um
            // determinado charset guardando o seu resultado num byte array.
            buf = mensagem.getBytes();

            // Enviar a mensagem (e o pacote) para o Cliente para o seu
            // endereço e porto...
            // ATENÇÃO :
            // Um grupo multicast é especificado por um endereço IP de
            // classe D, eg, de 224.0.0.1 até 239.255.255.255 e
            // por uma porta UDP.
            // Para enviar para o endereço de multicast...
            InetAddress grupo = InetAddress.getByName("224.0.0.1");
            System.out.println("Endereço mulicast: " + grupo);

            // Criação do pacote...
            DatagramPacket packet =
                new DatagramPacket(buf, buf.length, grupo, 4444);

            // Enviar o pacote do Servidor para o Cliente...
            socketServidor.send(packet);

            // Fechar a conexão ao Servidor, caso não existam mais cotações...
            if (mensagem.equals("Nao existem mais cotacoes... Adeus. "))
                break;

            // Pôr o Servidor a dormir por 2 segundos...
            // (cada vez que lê uma mensagem do ficheiro de cotações ou
            // mostra tempo)
            try {
                sleep(ESPERA);
            }
            // Algum problema na espera, mostrá-o...
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

        }
    }
    // Havendo alguma exceção mostrá-a...
    catch (Exception e) {
        e.printStackTrace();
        maisCotacoes = false;
    }
}

// Fechar o socket Servidor, para libertar recursos...
socketServidor.close();
}

protected String obterProximaCotacao() {
    String obterCotacao = null;
    // Ler mais cotacoes...
    try {
        if ((obterCotacao = in.readLine()) == null) {
            in.close();
            maisCotacoes = false;
            obterCotacao = "Nao existem mais cotacoes... Adeus.";
        }
    } catch (IOException e) {
        obterCotacao = "IOException obtida do lado do Servidor.";
    }
    return obterCotacao;
}
}

```

Classe MulticastServidor:



```

package sockets;

// Tem de estar ligado a uma rede para fazer multicast...
public class MulticastServidor {
    public static void main(String[] args) throws java.io.IOException {
        new MulticastServidorThreads().start();
    }
}

```

Classe MulticastCliente:



```

package sockets;

import java.io.IOException;

import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;
import java.net.UnknownHostException;

public class MulticastCliente {

    public static void main(String[] args) {

        try {
            // O Cliente vai estar a escutar no porto 4444...
            MulticastSocket socketCliente = new MulticastSocket(4444);

            // Para adicionar ao endereço de multicast...
            InetAddress endereco = InetAddress.getByName("224.0.0.1");
            System.out.println("Endereco mulicast: " + endereco);

            // Adicionar o socket Cliente ao grupo de endereços para onde o

```

```
// multicast está a ser efectuado...
socketCliente.joinGroup(endereco);

DatagramPacket pacote;

System.out.println("Cotacao Empresa          Preco");
System.out.println("-----");

// Obter algumas cotações...
for (int i = 0; i < 10; i++) {
    // Buffer para a passagem da mensagem dos pacotes...
    byte[] buf = new byte[256];

    // Criação do pacote...
    pacote = new DatagramPacket(buf, buf.length);

    // Receber o pacote do Servidor para o Cliente...
    socketCliente.receive(pacote);

    // Obter a mensagem incluída no pacote...
    String mensagem = new String(pacote.getData(), 0,
pacote.getLength());

    // Mostrar a mensagem - cotação...
    System.out.println(mensagem);

    // Para fechar a(s) conexão(conexões) do(s) Cliente(s)...
    if (mensagem.equals("Nao existem mais cotacoes... Adeus."))
        break;
}

// Remover o socket Cliente do grupo de endereços para onde o
// multicast está a ser efectuado...
socketCliente.leaveGroup(endereco);

// Fechar o socket Cliente, para libertar recursos...
socketCliente.close();
}
catch (UnknownHostException e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
}
}
```



A explicação deste código está nos comentários dentro do próprio código.

## Sumário

---

- Modelo OSI e arquitectura TCP;
- *Sockets*;
- TCP e UDP;
- Ambiente de trabalho;
- TCP – Implementar Cliente & Servidor;
- UDP – Implementar Cliente & Servidor;
- TCP – Multithreaded;
- UDP – *Multicast*.

