UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

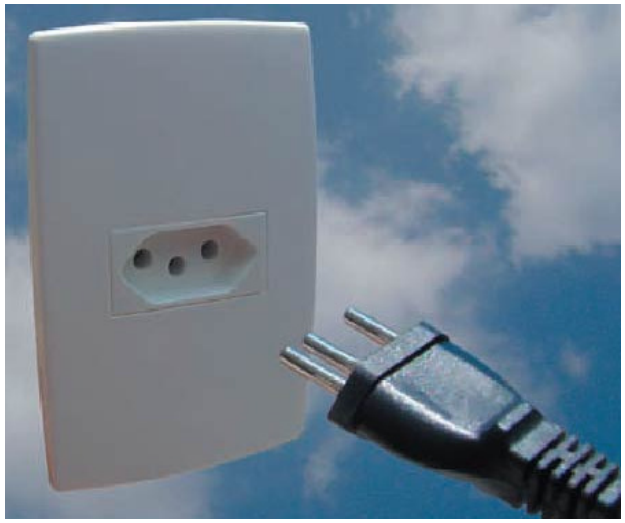
SOCKETS UDP, TCP E MULTICAST

Prof. Cesar Augusto Tacla
http://www.dainf.ct.utfpr.edu.br/~tacla

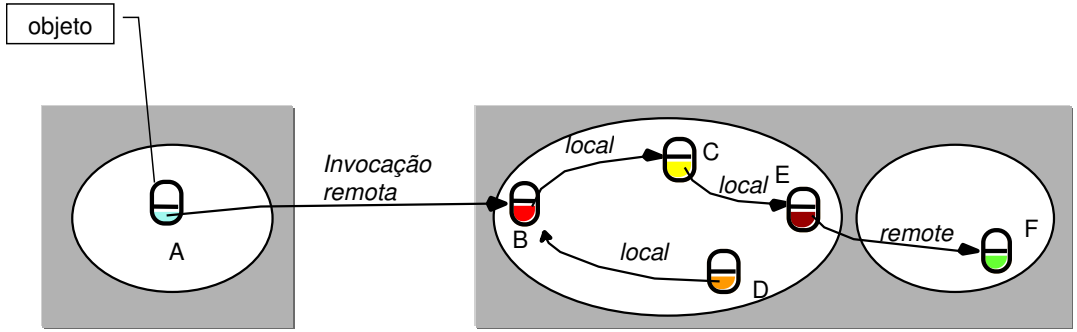
Sumário

CONTEXTO: Comunicação Inter-processos

- **Sockets**
  - Datagrama
  - *Stream* (fluxo)



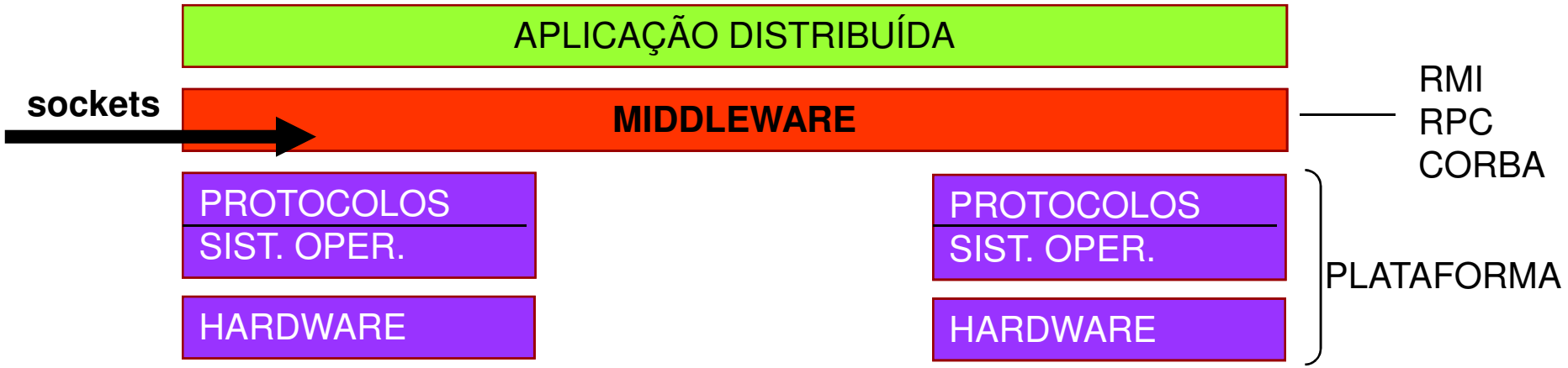
- **RMI**
  - *Remote method invokation*
  - Invocação remota de métodos



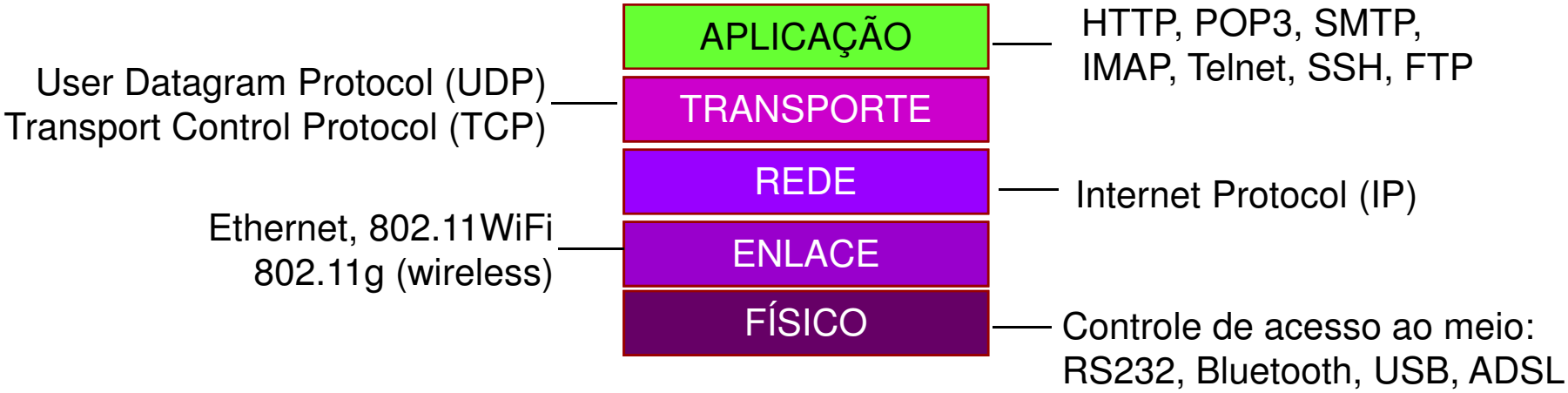
1 b

Conceito de Socket

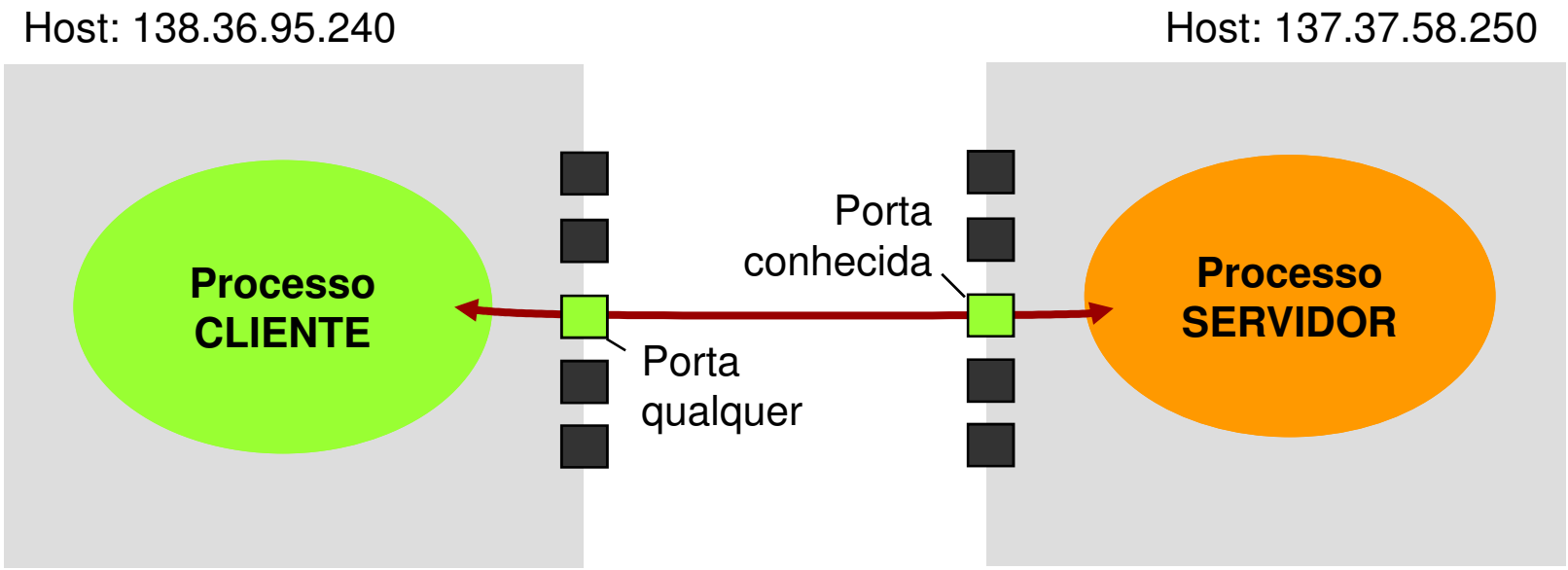
Os **sockets UDP** e **TCP** são a interface provida pelos respectivos protocolos na interface da camada de transporte.



PILHA DE PROTOCOLOS

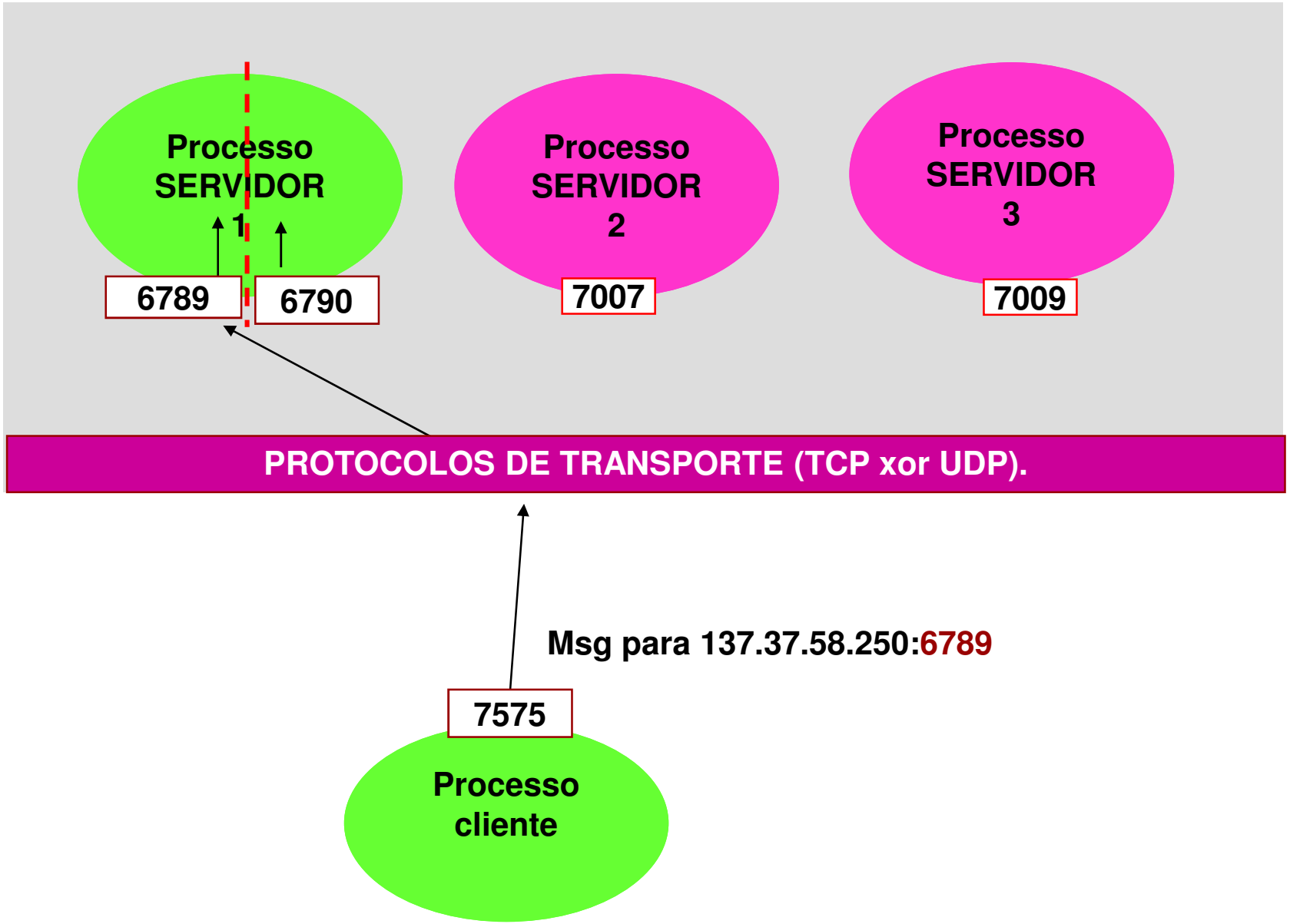


- Utilizado para comunicação interprocessos distribuídos
- Aplicações cliente-servidor



Socket = IP + porta

SOCKETS: CONCEITO



◇ Execute no shell dos: *netstat -na*

Active Connections

Proto	Local Address	Foreign Address	State
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING
TCP	0.0.0.0:2804	0.0.0.0:0	LISTENING
TCP	127.0.0.1:1033	0.0.0.0:0	LISTENING
TCP	127.0.0.1:1044	127.0.0.1:1045	ESTABLISHED
TCP	127.0.0.1:1045	127.0.0.1:1044	ESTABLISHED
TCP	127.0.0.1:1047	127.0.0.1:1048	ESTABLISHED
TCP	127.0.0.1:1048	127.0.0.1:1047	ESTABLISHED
TCP	200.17.96.134:139	0.0.0.0:0	LISTENING
TCP	200.17.96.134:139	200.17.96.175:1209	ESTABLISHED
TCP	200.17.96.134:2169	200.17.96.235:139	ESTABLISHED
UDP	0.0.0.0:445	*:*	
UDP	0.0.0.0:500	*:*	
UDP	0.0.0.0:1039	*:*	

2

SOCKETS UDP

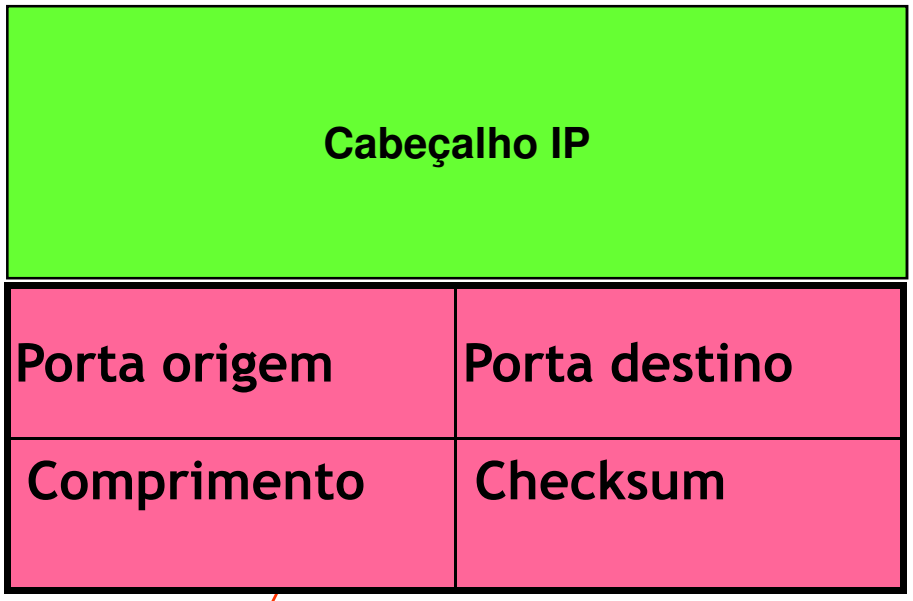
2 a

Protocolo UDP

◇ Sockets UDP: canal não-confiável

- Não garante entrega dos datagramas
- Pode entregar datagramas duplicados
- Não garante ordem de entrega dos datagramas
- Não tem estado de conexão (escuta, estabelecida)

Datagrama UDP



ENDEREÇO IP

Cabeçalho UDP: PORTAS

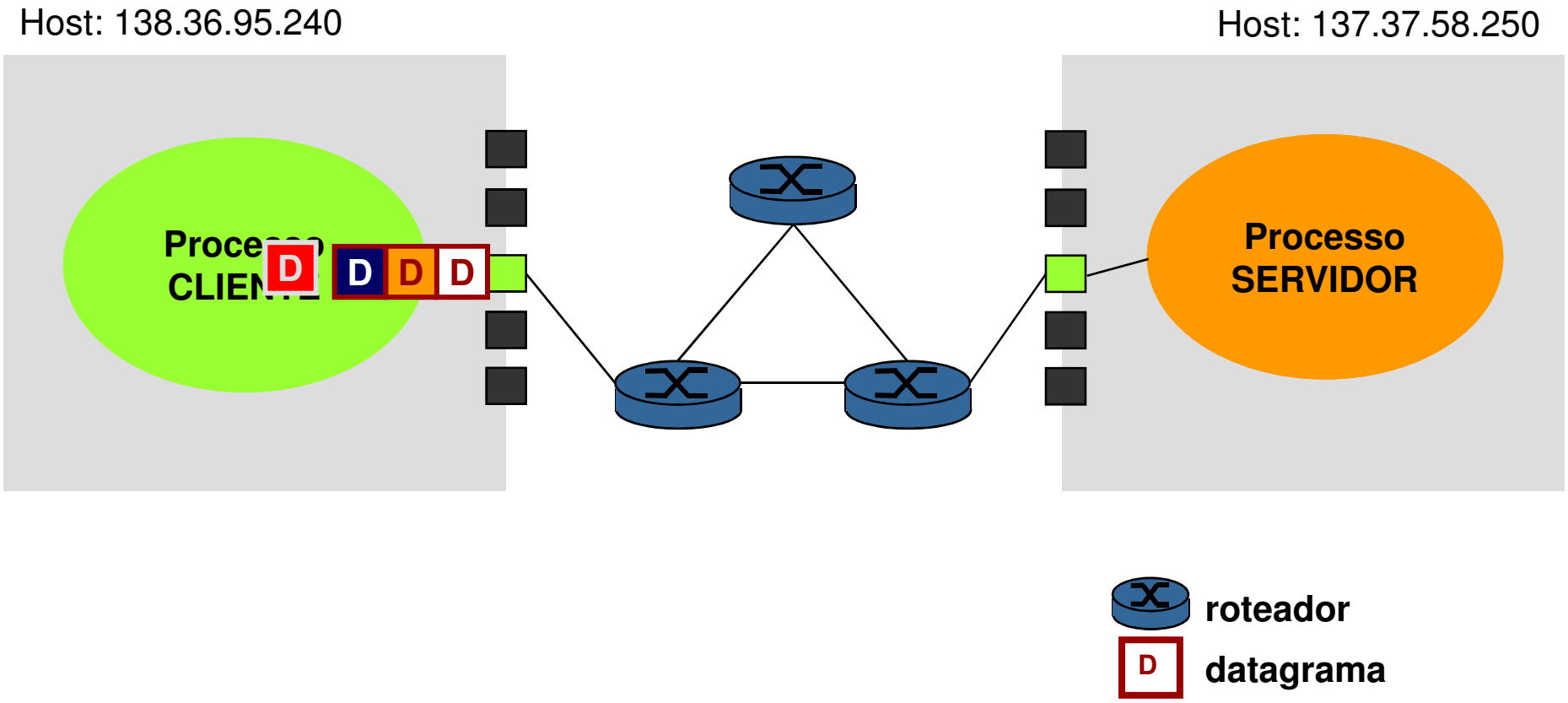
Datagrama

Mensagem auto-contida

Tamanho máximo: limitado pelo protocolo IP v4

2^{16} bytes (cabeçalhos + conteúdo) = 65.536 bytes

- Canal não-confiável



2 b

Programação com sockets UDP

SOCKETS UDP: Comandos básicos

- Criar socket
  - **DatagramSocket s = new DatagramSocket(6789);**
- Receber um datagrama
  - **s.receive(req);**
- Enviar um datagrama
  - **s.send(resp);**
- Fechar um socket
  - **s.close();**
- Montar um datagrama para receber mensagem
  - **new DatagramPacket(buffer, buffer.length);**
- Montar um datagrama para ser enviado
  - **new DatagramPacket(msg, msg.length, inet, porta);**
  - ***Buffer e msg são byte[]***

SOCKETS UDP: SERVIDOR

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

// cria um socket UDP
DatagramSocket s = new DatagramSocket(6789);
byte[] buffer = new byte[1000];
System.out.println("*** Servidor aguardando request");

// cria datagrama para receber request do cliente
DatagramPacket r = new DatagramPacket(buffer, buffer.length);
s.receive(r);
System.out.println("*** Request recebido de: " + r.getAddress());

// envia resposta
DatagramPacket resp = new DatagramPacket(r.getData(), r.getLength(),
                        r.getAddress(), r.getPort());
s.send(resp);
s.close();
```

Servidor de “um-tiro”. Ao receber uma conexão de um cliente, retorna a resposta e encerra a execução.

JAVARepositorio\JSockets\UDPScktCoulouris\src

SOCKETS UDP: CLIENTE

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

```
// cria um socket UDP
s = new DatagramSocket();
System.out.println("* Socket criado na porta: " + s.getLocalPort());
byte[] m = args[0].getBytes(); // transforma arg em bytes

InetAddress serv = InetAddress.getByName(args[1]);
int porta = 6789;
DatagramPacket req = new DatagramPacket(m, args[0].length(), serv, porta);

// envia datagrama contendo a mensagem m
s.send(req);

byte[] buffer = new byte[1000];
DatagramPacket resp = new DatagramPacket(buffer, buffer.length);
s.setSoTimeout(10000); // timeout em ms

// recebe resposta do servidor - fica em wait ateh chegada
s.receive(resp);
System.out.println("* Resposta do servidor:" + new String(resp.getData()));

// fecha socket
s.close();
```

JAVARepositorio\JSockets\UDPScktCoulouris\src

SOCKETS UDP: Esquema cliente-servidor

- **Cliente**
  1. **Criar socket**: um socket pode ser utilizado para enviar datagramas para qualquer socket servidor
  2. **Montar datagrama** com <servidor:porta> de destino <servidor:porta> de origem
  3. **Enviar datagrama**
  4. **Bloqueia num receive**
     - ...
     - ...
     - ...
  5. **Recebe a resposta**
  6. **Trata a resposta**
  7. **Volta ao item 2**

- **Servidor**
  1. **Aguarda num receive**
     - ...
     - ...
     - ...
     - ...
     - ...
     - ...
     - ...
     - ...
     - ...
  2. **Recebe a mensagem**
  3. **Processa a mensagem**
  4. **Envia resposta ao cliente**
  5. **Volta ao item 1**

EXERCÍCIOS

1. Baseando-se no código dos slides anteriores, fazer um servidor que atenda aos clientes invertendo a string recebida ou fazendo uma modificação qualquer na mensagem recebida (versão 1)

   Fonte em http://www.dainf.ct.utfpr.edu.br/~tacla/JAVARepositorio/JSockets/UDPScktCoulouris/src/
   Solução em JAVARepositorio\JSockets\UDPScktCoulourisInverteStr

2. Alterar o servidor (ex. 1) para tratar solicitações dos clientes de forma concorrente. Cada vez que uma mensagem é recebida, o servidor cria uma thread para atendê-la e volta a esperar por nova solicitação de cliente

   Solução: JAVARepositorio\JSockets\UDPScktServidorMultiThread

3. Desenhar um diagrama de sequência para o exercício 2 para o cenário de duas conexões simultâneas de clientes com o servidor (baixar o código do repositório).

◇ Prática 1: *servidor atende um cliente*

1. Baixar os arquivos do repositório

1. Numa janela DOS, rodar o servidor:
   - ir para o diretório onde se encontra o Servidor.jar
   - java –jar Servidor.jar

2. Verificar a execução do servidor (em outra janela) através de
   - netstat –a –b –p UDP

3. Rodar o cliente passando com argumentos uma mensagem e o endereço do servidor
   - Ex. java –jar Cliente.jar 127.0.0.1 6789 “MSG TST”

Código fonte e .jar disponível em
JAVARepositorio/JSockets/UDPScktCoulouris/dist/Servidor.jar
JAVARepositorio/JSockets/UDPScktCoulouris/dist/Cliente.jar

SOCKETS UDP: PRÁTICA 2

◇ Prática 2: *servidor atende vários clientes (multithread)*

1. Baixar os arquivos do repositório (quadrado em destaque)

2. Numa console DOS, rodar o servidor:
   - ir para o diretório onde salvou o .jar
   - java –jar **Servidor.jar**

3. Rodar o cliente passando com argumentos uma mensagem e o endereço do servidor.
   Duas threads clientes serão executadas.
   - Ex. java –jar **Cliente.jar** 127.0.0.1 6789 “mensagem teste”

Código fonte e .jar disponível em
JAVARepositorio\JSockets**UDPScktCoulourisClienteMultiThread**

EXERCÍCIO UDP

- **Fazer um sistema cliente-servidor para correção e estatística de questionários.**
- **O servidor deve ser capaz de receber várias conexões de clientes simultaneamente.**
- **O cliente envia ao servidor, vários datagramas contendo cada um uma resposta do tipo V ou F ao questionário, no seguinte formato:**
  - <número da questão>;<número alternativas>;<respostas>
  - Exemplo:
    - 1;5;VVFFV
    - 2;4;VVVV
- **O servidor lê a mensagem e calcula o número de acertos e erros devolvendo uma resposta simples:**
  - <número da questão>;<número acertos>;<número erros>
- **O servidor também faz uma estatística de acertos/erros por questão com base em todos as questões recebidas até um certo momento.Estas informações devem ser representadas num objeto compartilhado por todas as threads de atendimento aos clientes.**
  - *Estatística*
  - Questão 1: acertos=5 erros=3
  - Questão 2: acertos=4 erros=4

- **Send não é bloqueante**
- **Receive é bloqueante**
  - A menos que se especifique um *timeout*
  - Servidor pode ter várias *threads*:
    - uma na escuta de novas solicitações
    - outras, servindo os clientes
- **Um socket UDP não é *conectado***
  - Pode receber dados de quaisquer outros sockets
  - Exceto se for "conectado" a outro socket pelo método:
    - public void **connect**(InetAddress address, int port)

3

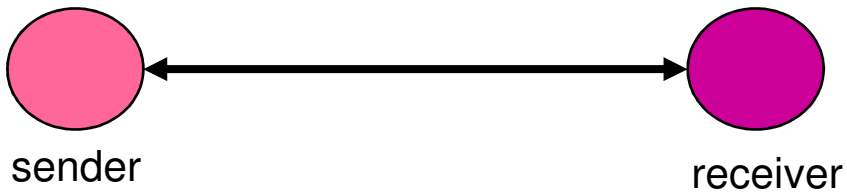
SOCKETS TCP

3 a

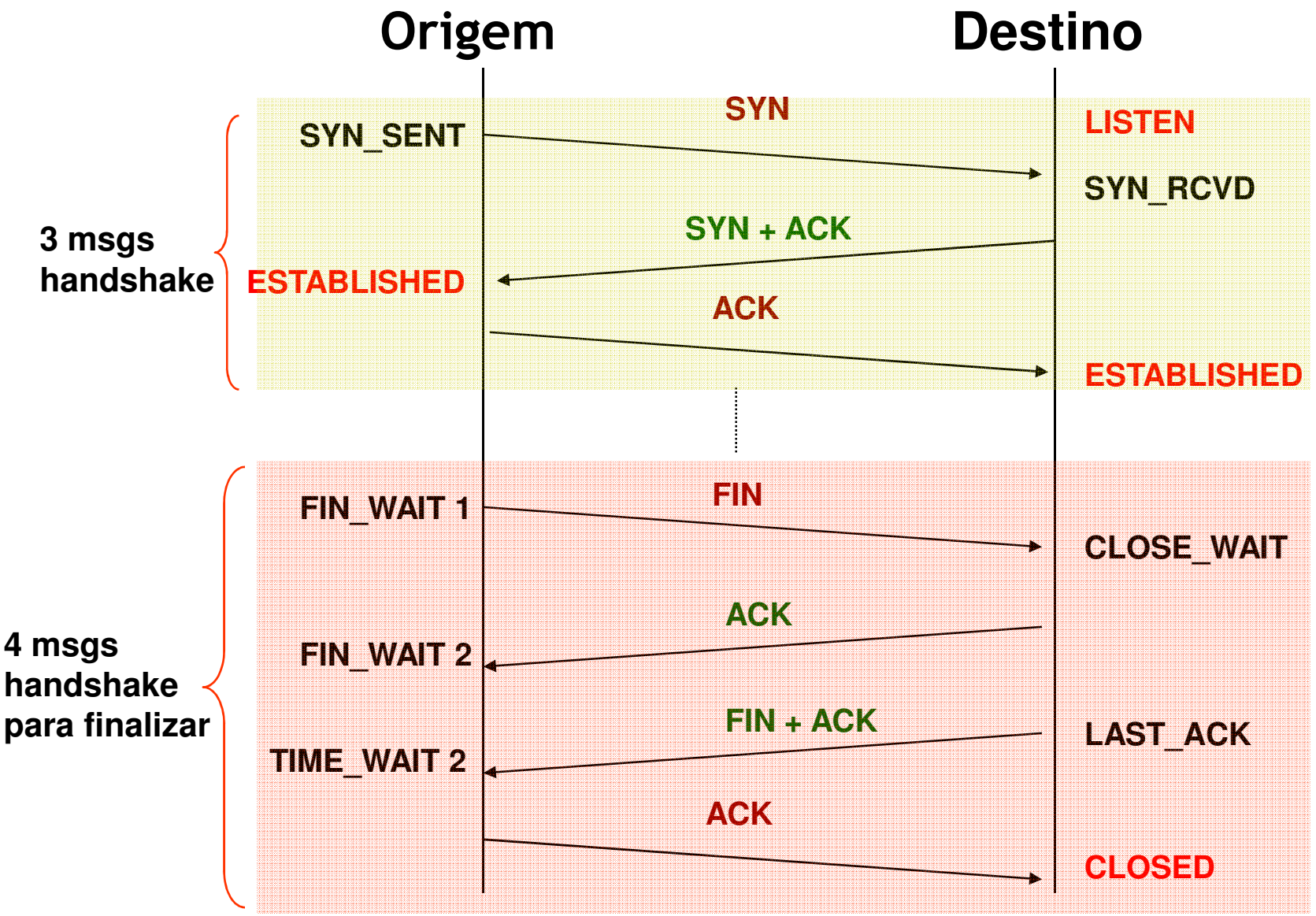
Protocolo TCP

◇ Protocolo TCP implementa um canal confiável

- Do ponto de vista do desenvolvedor: *fluxo contínuo (stream)*
- São fragmentados pelo TCP em *segmentos*
- Garante a **entrega** dos *segmentos*
- Não há **duplicação**
- Garante **ordem** de entrega dos *segmentos*
- Possui **conexão** e, portanto, controla o estado de conexão (escuta, estabelecida, fechada)
- Ponto-a-ponto: ***um sender:um receiver*** – sockets são conectados
- Controle de ***congestionamento***: TCP controla o sender quando a rede congestiona.
- ***Controle de fluxo***: Controla o sender para não sobrecarregar o receiver



SOCKETS TCP: ESTADOS CONEXÃO



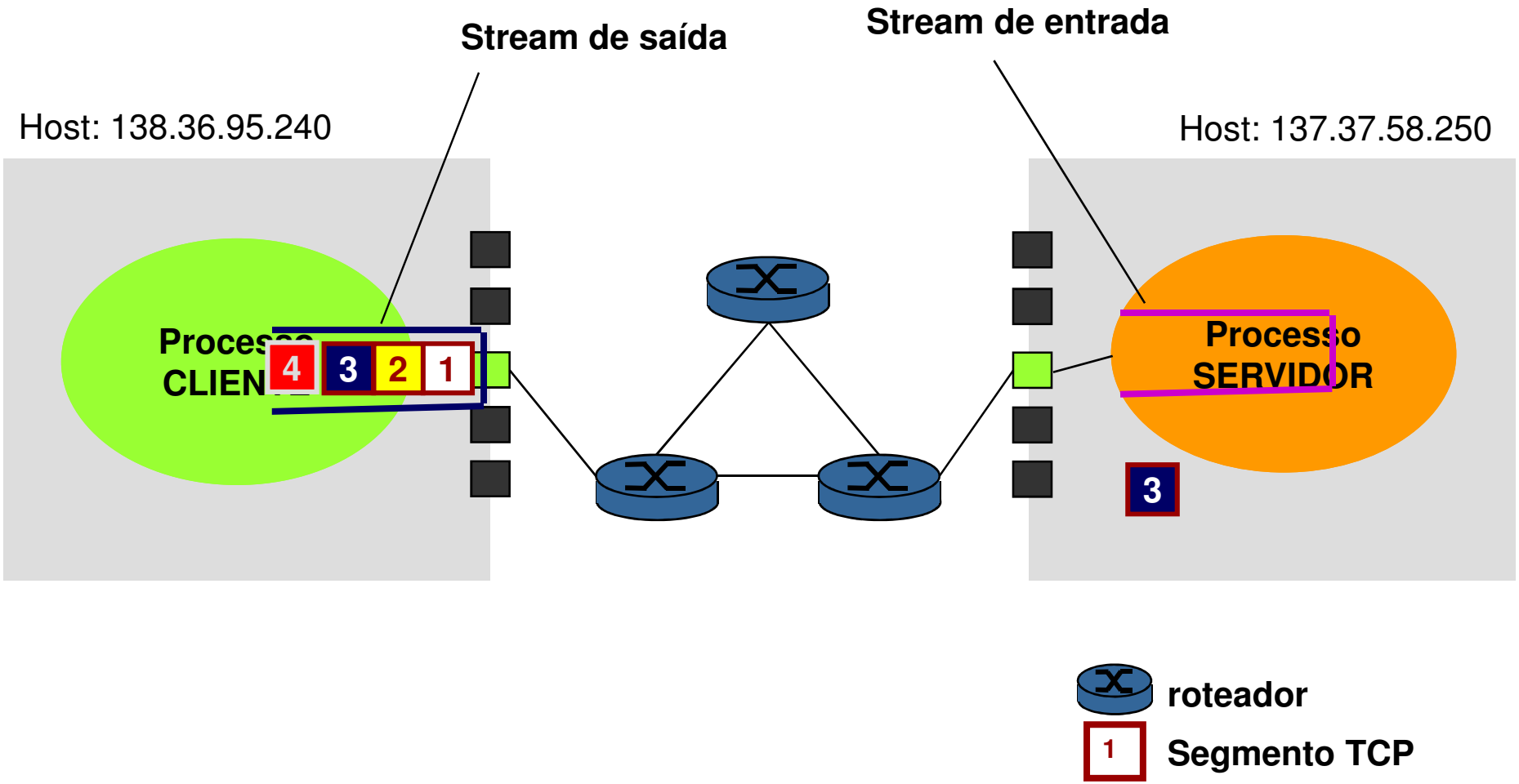
◇ Stream

- Um stream é uma ***seqüência de bytes*** transmitida ou recebida continuamente por um processo

- TCP preocupa-se em ***segmentar o stream***, se necessário, e entregar os segmentos à aplicação na ordem correta.
- Programador pode forçar envio: *flush*

- ***Para o programador de sockets TCP:***
  - basta gravar os dados num buffer de saída para que sejam enviados e
  - ler os dados de chegada num buffer de entrada.

◇ Canal confiável



3 b

Programação com sockets TCP

SOCKETS TCP: PRIMITIVAS DE PROGRAMAÇÃO

- Servidor cria **socket de escuta** **numa porta (ex. 6789)**
  - **ServerSocket ss = new ServerSocket(6789);**
- Servidor aceita uma conexão e cria novo socket para atendê-la
  - **Socket a = ss.accept();**
- **Cliente** cria **socket de conexão**
  - **Socket s = new Socket("localhost", 6789)**
- Fecha o socket
  - **s.close()**

SOCKETS TCP: PRIMITIVAS DE PROGRAMAÇÃO

- Cliente escreve no **stream de saída** do socket
  - *Criar um stream de dados - getOutputStream retorna uma classe abstrata*
  - `DataOutputStream sai = new DataOutputStream(sc.getOutputStream());`
  - `sai.writeUTF("mensagem para o servidor");`

- Cliente **lê o stream de entrada** do socket
  - **`DataInputStream ent = new DataInputStream(sc.getInputStream())`**
  - **`String recebido = ent.readUTF();`**

- *Leitura e escrita são similares no servidor, mas são feitas usualmente no socket retornado pelo método accept*

SOCKETS TCP: EXEMPLO SERVER

```
public class TCPServidor {
    public static void main(String args[]) {
        try {
            int porta = 6789; // porta do serviço
            if (args.length > 0) porta = Integer.parseInt(args[0]);
            ServerSocket escuta = new ServerSocket(porta);
            System.out.println("*** Servidor ***");
            System.out.println("*** Porta de escuta (listen): " + porta);

            while (true) {
                // accept: bloqueia servidor até que chegue um
                // pedido de conexão de um cliente
                Socket cliente = escuta.accept();
                System.out.println("*** conexao aceita de (remoto): " +
                                   cliente.getRemoteSocketAddress());
                // quando chega, cria nova thread para atender o
                // cliente passando o socket retornado por accept
                Conexao c = new Conexao(cliente);
            }
        } catch (IOException e) {
            System.out.println("Erro na escuta: " + e.getMessage());
        }
    }
}
```

SOCKETS TCP: EXEMPLO CLIENTE

```
public class TCPCliente {
    public static void main(String args[]) {
        Socket s = null;
        try {
            // conecta o socket aa porta remota
            s = new Socket(args[0], Integer.parseInt(args[1]));
            DataInputStream ent = new DataInputStream(s.getInputStream());
            DataOutputStream sai = new DataOutputStream(s.getOutputStream());
            sai.writeUTF(args[2]);
            // le stream de entrada
            String recebido = ent.readUTF();
            System.out.println("*** Recebido do servidor: " + recebido);
         } catch (UnknownHostException e) {
             System.out.println("!!! Servidor desconhecido: " + e.getMessage());
         } catch (EOFException e) {
             System.out.println("!!! Nao ha mais dados de entrada: " + e.getMessage());
         } catch (IOException e) {
             System.out.println("!!! E/S: " + e.getMessage());
         } finally {
             if (s!=null)
                 try {
                     s.close();
                 } catch (IOException e){
                    System.out.println("!!! Encerramento do socket falhou: " +
                                       e.getMessage());
                 }
         }
    }
}
```

EXEMPLO 2: Transferência de objetos

```
public class TCPServidor {
    public static void main(String args[]) throws
                ClassNotFoundException, IOException {
        int porta = 6789;
        ServerSocket escuta = new ServerSocket(porta);
        System.out.println("*** Servidor ***");
        System.out.println("*** Porta de escuta (listen): " + porta);
        while (true) {
            Socket cliente = escuta.accept();
            System.out.println("*** conexao aceita de (remoto): " +
                    cliente.getRemoteSocketAddress());
            ObjectInputStream ois =
                  new ObjectInputStream(cliente.getInputStream());
            while (true) {
                try {
                    Documento doc = (Documento) ois.readObject();
                    System.out.println(doc.toString());
                } catch (IOException e) {
                    break;
                }
            }
        }
    }
}
```

Servidor desempacota os objetos recebidos

Código fonte e .jar disponível em
http://www.dainf.ct.utfpr.edu.br/~tacla/JAVARepositorio/JSockets/TCPTrocaObjetos/

EXEMPLOS: Transferência de Objetos

```
public class TCPCliente {
    public static void main(String args[]) {
        Socket s = null;
        try {
            s = new Socket("localhost", 6789); // conecta o socket aa porta remota
            ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());
            Documento d1 = new Documento("Divina Comedia", "Dante");
            oos.writeObject(d1);
            Documento d2 = new Documento("Dom Casmurro", "M. de Assis");
            oos.writeObject(d2);
         } catch (UnknownHostException e) {
             System.out.println("!!! Servidor desconhecido: " + e.getMessage());
         } catch (EOFException e) {
             System.out.println("!!! Nao ha mais dados de entrada: " +
                     e.getMessage());
         } catch (IOException e) {
             System.out.println("!!! E/S: " + e.getMessage());
         } finally {
             if (s!=null)
                 try {
                     s.close();
                 } catch (IOException e){
                    System.out.println("!!! Encerramento do socket falhou: " +
                        e.getMessage());
                 }
         }
    }
}
```

◇ Prática 1: request-reply

1. Baixar os arquivos do repositório (ver quadro)

2. Numa console DOS, fazer:
   - ir para o diretório onde se encontra os .jar
   - Executar: java -jar Servidor **ou** *//porta default 6789*
   - Executar: java -jar Servidor <porta>

3. Em outra janela, rodar o cliente passando com argumentos uma mensagem e o endereço do servidor.
   - Ex. java –jar Cliente <SERVIDOR> <PORTA> <MENSAGEM>
   - Ex. java –jar Cliente *localhost* 6789 "MENSAGEM TESTE TCP"

Código fonte e .jar disponível em
http://www.dainf.ct.utfpr.edu.br/~tacla/JAVARepositorio/JSockets/TCPScktRequestReply/

Prática 2: Servidor de arquivo

1. Baixar os arquivos do repositório (ver quadro) – baixe o arquivo .txt da pasta *dist*

2. Numa console DOS, fazer:
   - ir para o diretório onde se encontram os .jar
   - Executar: java –jar Servidor.jar

3. Rodar o cliente passando como argumentos o servidor, porta e seu nome
   - Ex. java –jar Cliente.jar <SERVIDOR> <PORTA> <NOME>
   - Ex. java –jar Cliente "localhost" 6789 CESAR

4. Executar vários clientes em consoles diferentes para verificar que o servidor atende a todos por ser multithread. Cada cliente recebe um stream de dados que está armazenado no arquivo teste.txt no servidor.

Código fonte e .jar disponível em
http://www.dainf.ct.utfpr.edu.br/~tacla/JAVARepositorio/JSockets/TCPScktServidorArquivo/

SOCKETS TCP: EXERCÍCIO

- **Modifique o código do servidor de arquivos para que o cliente possa Solicitar o arquivo que deseja baixar passando o nome o mesmo.**
  - Servidor pode retornar o arquivo (se existir) ou uma mensagem de arquivo não encontrado.

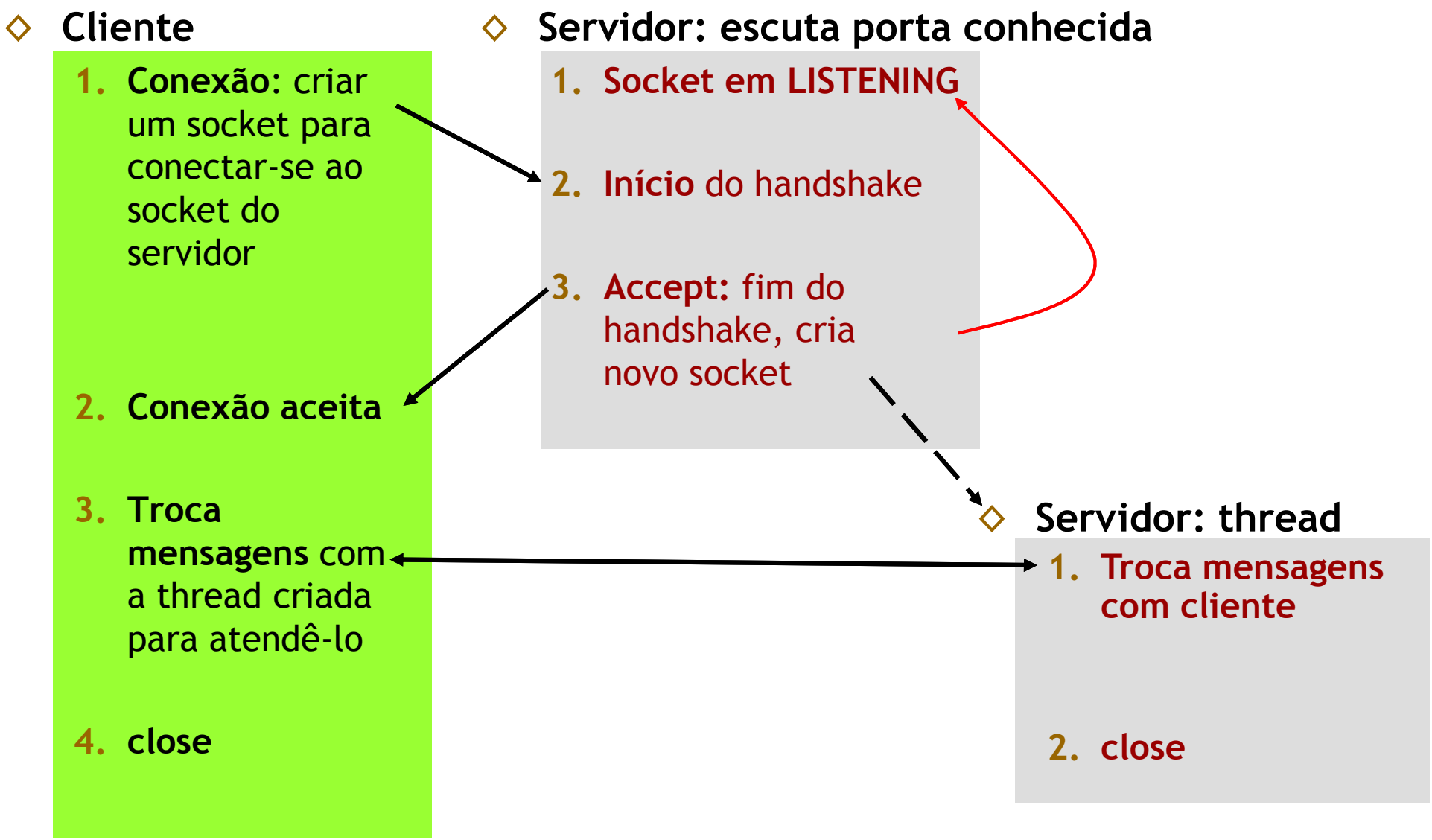
Código fonte e .jar disponível em
http://www.dainf.ct.utfpr.edu.br/~tacla/JAVARepositorio/JSockets/TCPScktServidorArquivo/

Solução: Código fonte e .jar disponível em
---- não disponível ----

EXERCÍCIO

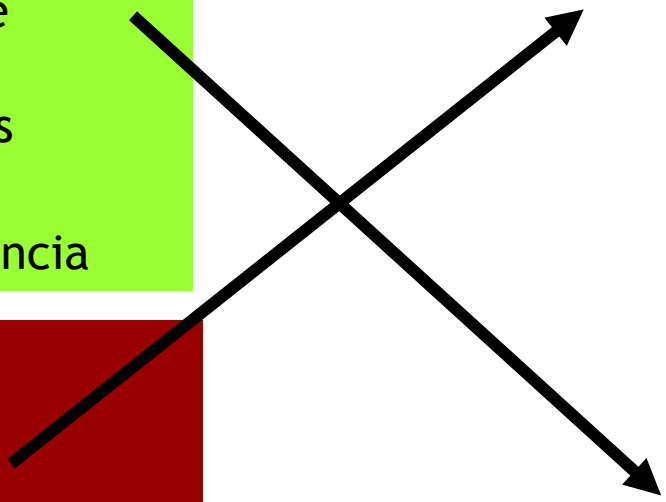
- Fazer um sistema cliente-servidor para correção e estatística de questionários.
- O servidor deve ser capaz de receber várias conexões de clientes simultaneamente.
- O cliente envia ao servidor, um arquivo texto contendo as respostas tipo V ou F ao questionário, no seguinte formato:
  - <número da questão>;<número alternativas>;<respostas>
  - Exemplo:
    - 1;5;VVFFV
    - 2;4;VVVV
- O servidor lê o arquivo e calcula o número de acertos e erros devolvendo uma resposta simples:
  - <número acertos>;<número erros>
- O servidor também faz uma estatística de acertos/erros por questão com base em todos os questionários recebidos até um certo momento.Estas informações devem ser representadas num objeto compartilhado por todas as threads de atendimento aos clientes.
  - Questão 1: acertos=5 erros=3
  - Questão 2: acertos=4 erros=4

SOCKETS TCP: ESQUEMA MULTI-THREAD

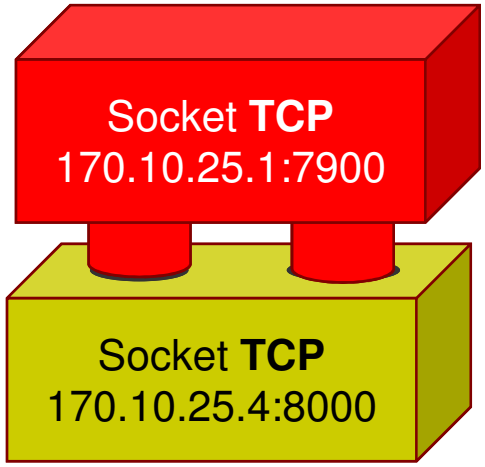


SOCKETS UDP X TCP

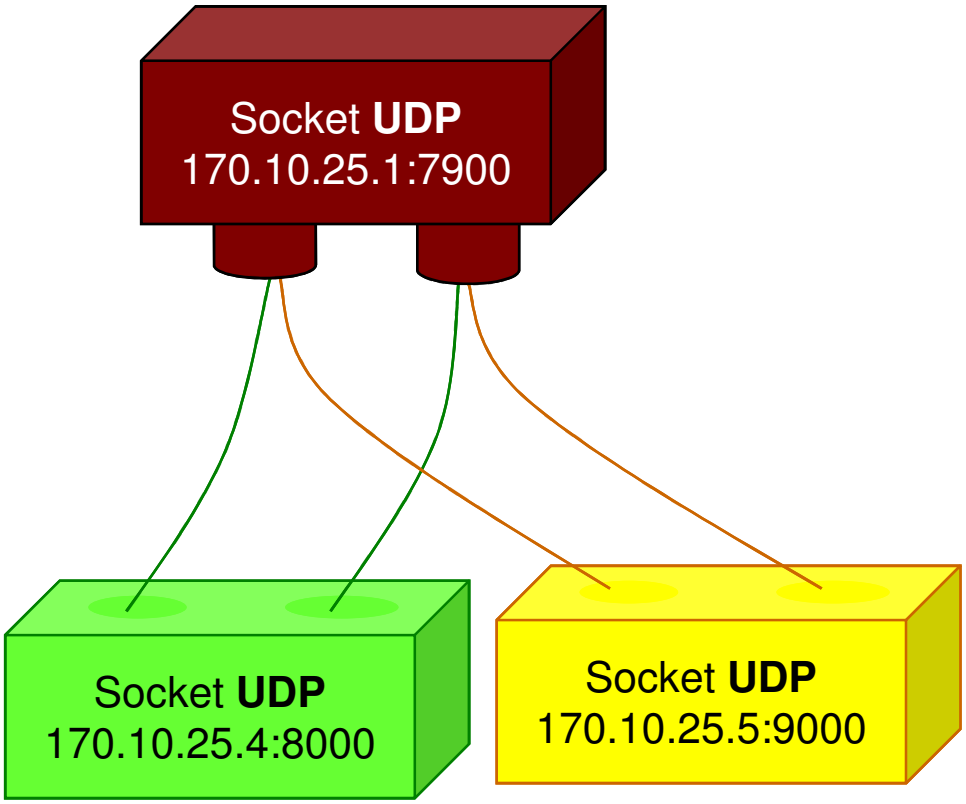
- **UDP**
  - **VANTAGENS**
    - Overhead pequeno: não há handshake de conexão/finalização
    - Não há necessidade de salvar estados de transmissão nas pontas (fonte e destino)
    - Diminui tempo de latência
  - **DESVANTAGENS**
    - Perda de mensagens
    - Não há ordenação
    - Limite de tamanho de mensagens
- **TCP**



SOCKETS UDP X TCP



CONECTADO



PROMÍSCUO

4

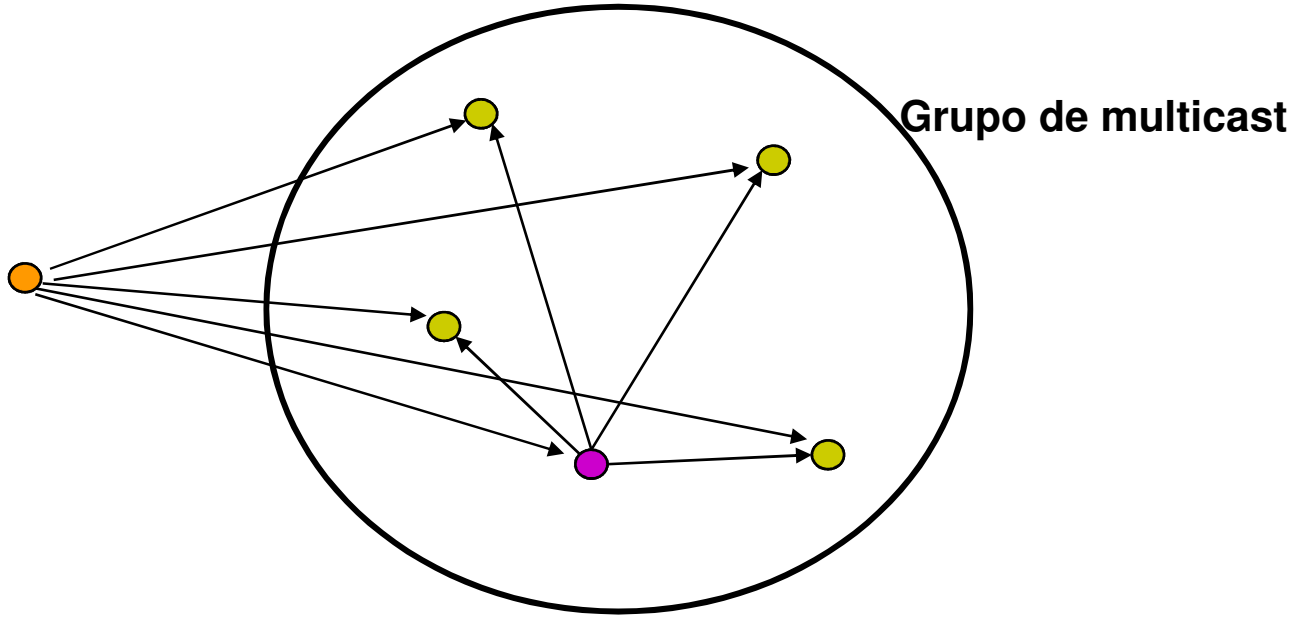
SOCKETS MULTICAST

4 a

Protocolo Multicast

- Um processo envia uma mensagem para um grupo de processos

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



Variações

- Multicast confiável e não-confiável
- Ordenado e não-ordenado
- Ordenado: FIFO, CAUSAL e TOTAL

Aplicações

- Difusões de áudio e vídeo
- Replicação de serviços
- Localização de serviços em redes espontâneas
- Replicação de operações/dados
- Difusão de eventos

MULTICAST: Funcionamento

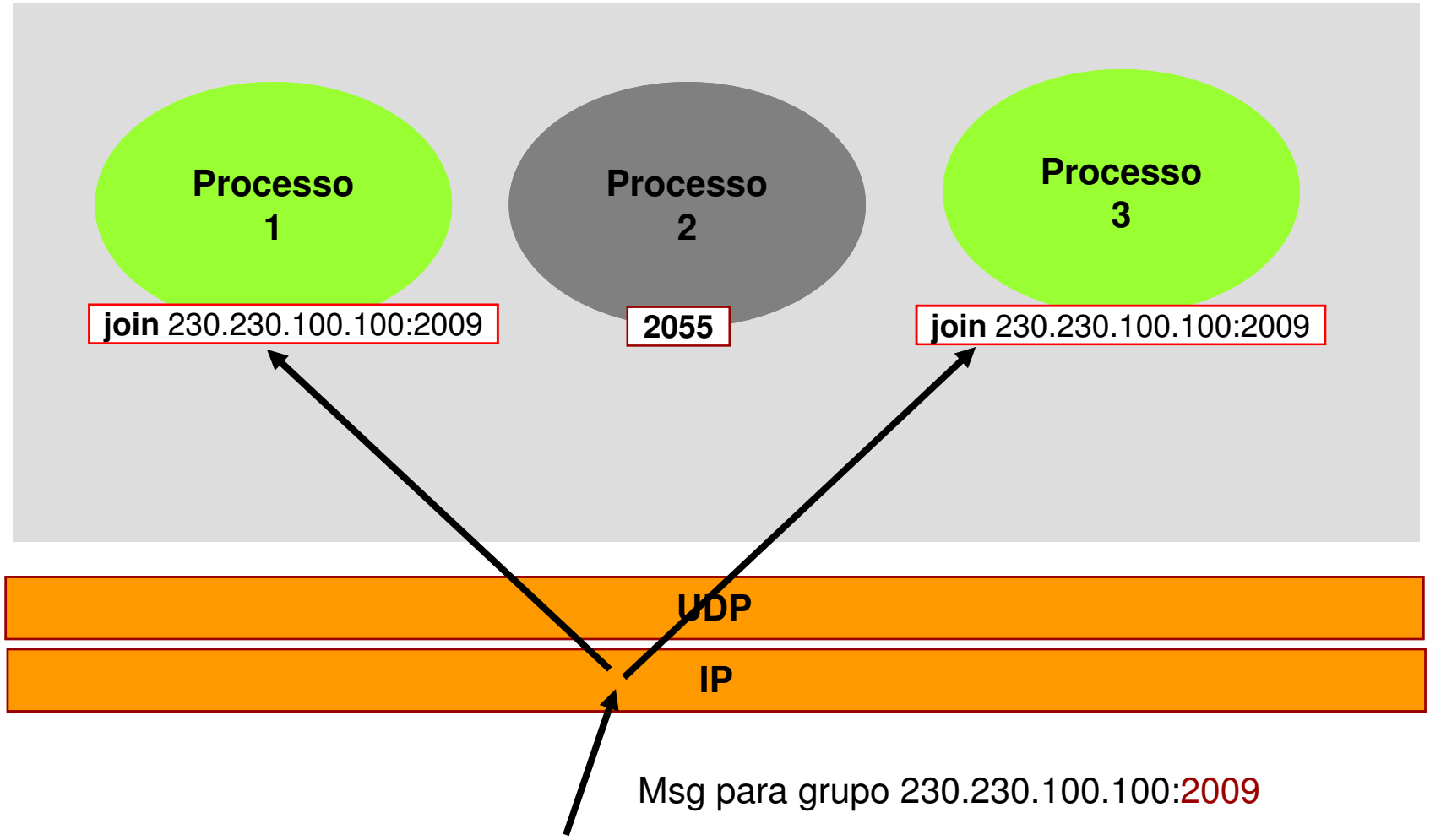
- **Um único pacote IP para um grupo**
- O emissor não conhece:
  - a identidade dos destinatários
  - o tamanho do grupo

- **Endereço de Grupo (IPv4)**
  - Categoria D
  - [224, 239] [0, 255] [0, 255] [0, 255]
  - Exemplo: 230.230.100.100

  - **Reservados**: 224.0.0.1 até 224.0.0.255
  - **Temporários**: todos os demais, existem enquanto houver participantes

◇ Recebimento de pacotes

Host: 137.37.58.250



4 b

Programação sockets Multicast

SOCKETS MULTICAST: PRIMITIVAS PROGRAMAÇÃO

- s = new MulticastSocket(porta)
- s.joinGroup(ipGrupo)
- s.leaveGroup(ipGrupo)

SOCKETS MULTICAST: EXEMPLO ENVIO MSG

```
class Envia {
    public static void main(String args[]) {
        int porta=6868;
        InetAddress ipGrupo=null;
        MulticastSocket s=null;
        String msg="mensagem default";

        // junta-se a um grupo de Multicast
        try {
            ipGrupo = InetAddress.getByName("224.225.226.227");
            s = new MulticastSocket(porta);
            s.joinGroup(ipGrupo);
        } catch (SocketException e) { }

        // envia mensagem
        DatagramPacket dtgrm = new DatagramPacket(msg.getBytes(),
                msg.length(), ipGrupo, porta);
        try {
            s.send(dtgrm);
        } catch (IOException e) {  }

       // sai do grupo e fecha o socket
       try {
            s.leaveGroup(ipGrupo);
            if (s!=null) s.close();
        } catch (IOException e) {  }
    }
}
```

SOCKETS MULTICAST: EXEMPLO RECEPÇÃO MSG

```
class Observador {
    public static void main(String args[]) {
        int porta=6868;
        InetAddress ipGrupo=null;
        MulticastSocket s=null;
        String msg="mensagem default";

        // junta-se a um grupo de Multicast
        try {
            ipGrupo = InetAddress.getByName("224.225.226.227");
            s = new MulticastSocket(porta);
            s.joinGroup(ipGrupo);
        } catch (SocketException e) { }

        // le continuamente as mensagens
        byte[] buf = new byte[1512];
        while (true) {
            DatagramPacket recebido = new DatagramPacket(buf, buf.length);
            try {
                s.setSoTimeout(120000);
                s.receive(recebido);
            } catch (SocketTimeoutException e) {
                break;
            } catch (IOException e) { }
            String str = new String(recebido.getData());
            System.out.println("(" + recebido.getAddress().getHostAddress() +
                    ":" + recebido.getPort() + ") << " + str.trim());
        }
}
```

◇ Prática 1: chat

1. Baixar os arquivos em destaque no quadro

2. Numa console DOS, fazer:
   - ir para o diretório onde se encontra o *.jar*
   - *Executar*: java –jar JMulticastConfererenciav2.jar <IP grupo> <porta> <usr>
   - *Ex*: java –jar JMulticastConferenciav2.jar 231.232.233.234 6789 DOG

3. O programa permite fazer uma espécie de chat
   - Basta teclar uma mensagem seguida de enter, para que todos os membros do grupo a recebam
   - Para encerrar, não teclar nada durante 1 minuto

Solução: Código fonte e .jar disponível em
http://www.dainf.ct.utfpr.edu.br/~tacla/JAVARepositorio/JSockets/JMulticastConferenciav2/

[JAVA]

SOCKETS MULTICAST: EXERCÍCIO

- Inclua no código anterior uma janela de para envio/recepção de mensagens
- Incluir uma opção de sair do chat

Solução: Código fonte e .jar disponível em
--- não disponível ---