# Java
# Sockets

## Utilização das classes Java para comunicação TCP/IP e UDP/IP

- **Agenda**
  - **Diferenças entre TCP e UDP**
  - **Comunicação utilizando *Streams* - TCP**
  - **Comunicação utilizando *Datagramas* - UDP**

- **Referencias**
  - **SUN - Essentials of the Java Programming Language**
    - http://java.sun.com/developer/onlineTraining/Programming/BasicJava1
    - http://java.sun.com/developer/onlineTraining/Programming/BasicJava2
  - **SUN - The Java Tutorials, Networking**
    - http://java.sun.com/docs/books/tutorial/networking/index.html

# Diferenças Entre TCP e UDP

***TCP (Transmission Control Protocol)***

- **Orientado a conexão**
- **Confiável**
- ***Stream***
- **Controle de fluxo**

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

- **Orientado a datagrama**
- **Não é confiável**
- **Datagramas (pacotes)**
- **Sem controle de fluxo**

# Diferenças Entre TCP e UDP

***TCP (Transmission Control Protocol)***

- **Mensagens ordenadas**
- **Mais lento**

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

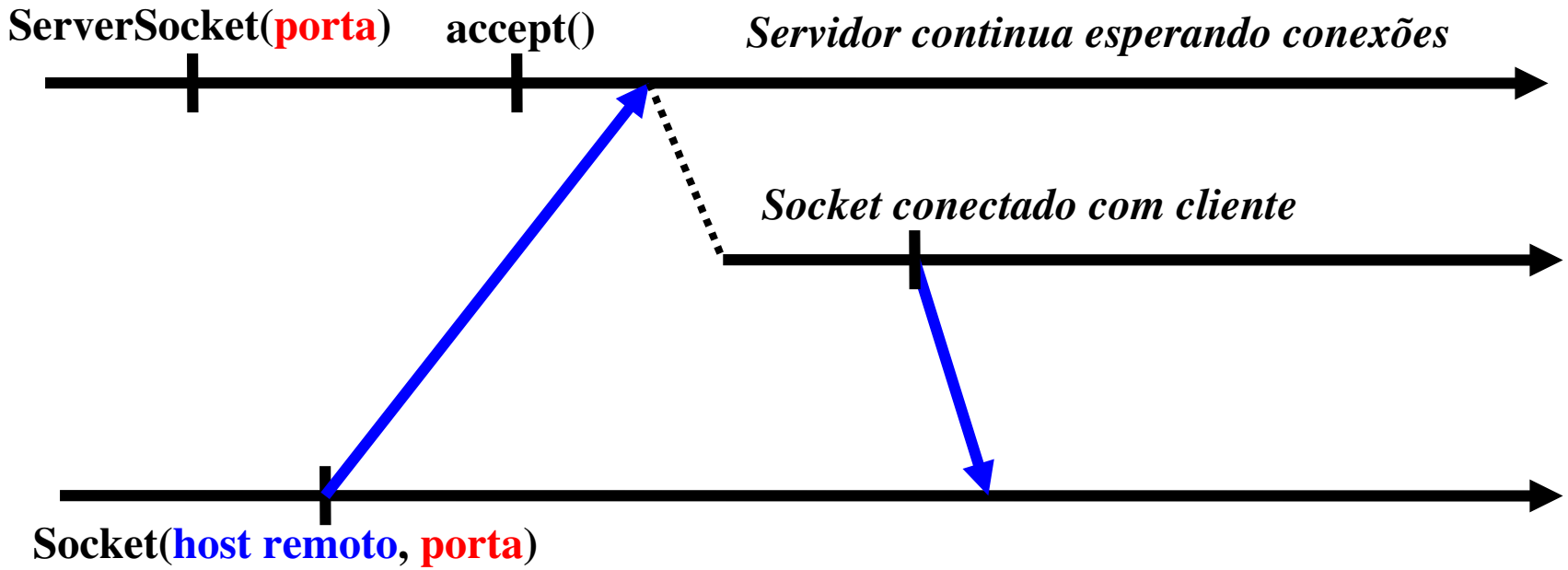
- **Sem garantia de ordem ou de chegada**
- **Menor *overhead***
- **Mais apropriado a broadcast**

- **Conceitos básicos de sockets TCP**
  - **servidor**
    - possui (cria) um socket associado a uma porta
    - espera pedidos de conexões de clientes
    - conexão aceite
      - novo socket é criado para a conexão em nova porta
      - permite aceitar outras conexões na mesma porta enquanto conexões anteriores estejam abertas

- **Conceitos básicos de sockets TCP**
  - **cliente**
    - conhece hostname (IP) da máquina servidora
    - conhece porta do programa servidor
    - pede conexão
    - se a conexão for aceite
      - um socket é criado
      - associado a uma porta na máquina cliente

- **Classes sockets TCP em Java**
  - **no pacote java.net**
  - **escondem detalhes dependentes de plataforma**
  - **código (mais) portável**
  - **classes**
    - ServerSocket
      - usada por servidores
    - Socket
      - usada por clientes e servidores

**Servidor: classe java.net.ServerSocket**




**Cliente: classe java.net.Socket**

- **Sockets TCP e Classes Java**
  - **Primeiro**
    - servidor cria *ServerSocket*
    - espera por mensagens em uma determinada porta (método *accept( )*)
  - **Cliente cria *Socket*, conectando com o servidor**
  - **Servidor**
    - pode criar uma nova *Thread* para atender cliente
    - continua aceitando novas conexões na mesma porta

# Código para Criar Conexão

**SERVIDOR**

```
(...) ServerSocket s = new
    ServerSocket(8189);

while (true){

 Socket conexao = s.accept();

 /* Disparar uma thread que faça
    algo, passando conexão como
    parâmetro */

 (...)

}
```

**CLIENTE**

```
(...)Socket s;

try{

s =new Socket("netlab",8189);

}catch(Exception e) {/*Erro*/

    System.exit(0);

}

/* Socket conectado */
```

- **Como Enviar e Receber Mensagens**
  - **A classe *Socket* não tem *send( )* e *receive( )***
  - **Os métodos *getInputStream( )* e *getOutputStream( )***
    - retornam objetos “fluxos de bytes” (*streams*)
    - que podem ser manipulados como se viessem de arquivos
    - esses métodos pertencem às classes InputStream e OutputStream, e suas derivadas

- **Como Enviar e Receber Mensagens**
  - **Várias classes e métodos para leitura e escrita em *streams***
    - podendo transmitir desde bytes até objetos
  - **Para fechar uma conexão, utilizar *close( )***

# Exemplos de Uso de Streams

**RECEBER**

```
(...)InputStream input;
try {  input = s.getInputStream();
} catch (IOException e) {(...)}
ObjectInputStream objInput;
try {
objInput = new
   ObjectInputStream(input);
String line = (String)
   objInput.readObject( );
}catch (Exception e){(...)}
```

**ENVIAR**

```
(...) OutputStream output;
try {  output =
   s.getOutputStream();
} catch (IOException e) {(...)}
ObjectOutputStream objOutput;
try {
objOutput = new
   ObjectOutputStream(output);
objOutput.writeObject( “Olá!”);
}catch (Exception e){(...)}
```

- **Uso de Streams**
  - **um socket (conexão) pode ser usado ao mesmo tempo para**
    - input stream
    - output stream
  - **após a conexão**
    - tanto cliente quanto servidor podem tomar a iniciativa de trocar mensagens
    - evitar somente deadlocks
      - dois em receive inicialmente

# TCP: limite de conexões

- **Quantidade de conexões**
  - **limite da fila de pedidos de conexão em espera**
    - na versão 1.2: 50 é o default
  - **limite de conexões abertas**
    - na versão 1.2: não encontrado

- **Exemplo Echo**
  - **fonte: tutorial Java da Sun**
    - trail networking, lição Socktes, 1o exemplo
  - **descrição**
    - le string da standard input
    - envia o string ao servidor Echo
    - recebe resposta do servidor Echo
    - imprime resposta

- **Exemplo Echo**
  - **código**

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

public class EchoClient {
    public static void main(String[] args)
                throws IOException {

        Socket echoSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;
```

- **Exemplo Echo**
  - **código**

```
try {
        // cria socket local e conecta ao servidor
        echoSocket = new Socket("taranis", 7);
        // PrintWriter:  1o arg: OutputStream
        //                    2o arg: println com ação flush
        out = new
            PrintWriter(echoSocket.getOutputStream(), true);
        // BufferedReader: arg: Reader
        // InputStreamReader: arg: InputStream
        //                                subclasse de Reader
         in = new BufferedReader(new InputStreamReader(
               echoSocket.getInputStream()));
    }
```

- **Exemplo Echo**
  - **código**

```
catch (UnknownHostException e) {
    System.err.println("Don't know about host: taranis.");
    System.exit(1);
} catch (IOException e) {
    System.err.println("Couldn't get I/O for ”
                       + "the connection to: taranis.");
    System.exit(1);
}
```

- **Exemplo Echo**
  - **código**
    - ```
      BufferedReader stdIn = new BufferedReader(
            new InputStreamReader(System.in));
      String userInput;

      while ((userInput = stdIn.readLine()) != null) {
         out.println(userInput);
         System.out.println("echo: " + in.readLine());
      }
      ```
      ```

- **Exemplo Echo**
  - **código**

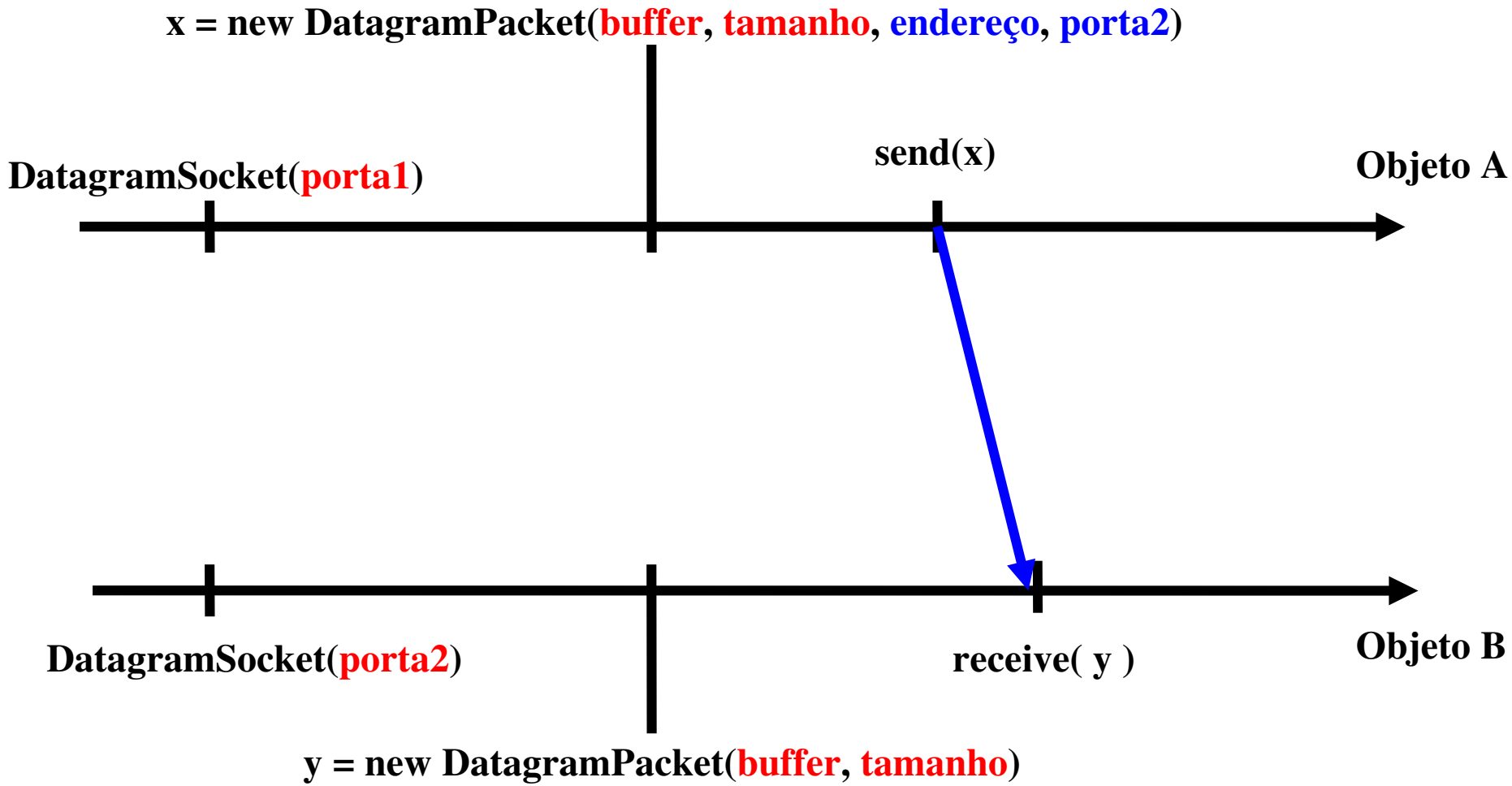
```
out.close();
in.close();
stdIn.close();
echoSocket.close();
}
}
```

# Exercícios Sockets

- **Exercícios**
  - **A) identifique e explique os comandos send e receive no exemplo Echo**
  - **B) faça um esqueleto (pseudo-código) do programa Echo servidor**

- **Sockets UDP**
  - **sumário**
    - esquema de comunicação com classes
    - esqueleto de programa

# Sockets UDP e Classes Java



- **Sockets UDP e Classes Java**
  - **Ambos os lados da conexão criam um novo *DatagramSocket***
    - receptor deve informar a sua porta de recepção
  - **Ambos os objetos criam *DatagramPacket***
    - mas o objeto que vai receber o pacote tem que informar o endereço  e porta do destinatário
  - **Para cada mensagem a ser enviada**
    - criar um novo *DatagramPacket*
    - *infomar mensagem (buffer e tamanho) e destino (endereço e porta)*

- **Sockets UDP e Classes Java**
  - **Utilizar os métodos send e  receive**
    - programador precisa empacotar/desempacotar dados em um buffer
  - **Para terminar a conexão, utilizar close**

# Exemplo de Código para Datagramas

**SEND**

```
(...) DatagramSocket s;

try {

     s= new DatagramSocket( );

}catch(SocketException e)
    { (....) }

byte[] b = {0,1,2,3,4,5,6,7};

DatagramPacket p = new
    DatagramPacket ( b, 8, iaddr,
    2000);

try{

    s.send( p );

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

**RECEIVE**

```
(...) DatagramSocket s;

try {

     s= new
    DatagramSocket( 2000 );

}catch(SocketException e)
    { (....) }

DatagramPacket p = new
    DatagramPacket (new byte[8],
    8);

try{

    s.receive( p );

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

iaddr é o endereço InetAddr do host para onde será mandada a mensagem