



Linguagem de Programação III (LP35A)

Prof^a Eliana Santos

Linguagem de Programação III (LP35A)

Programação paralela: servidor de tarefas

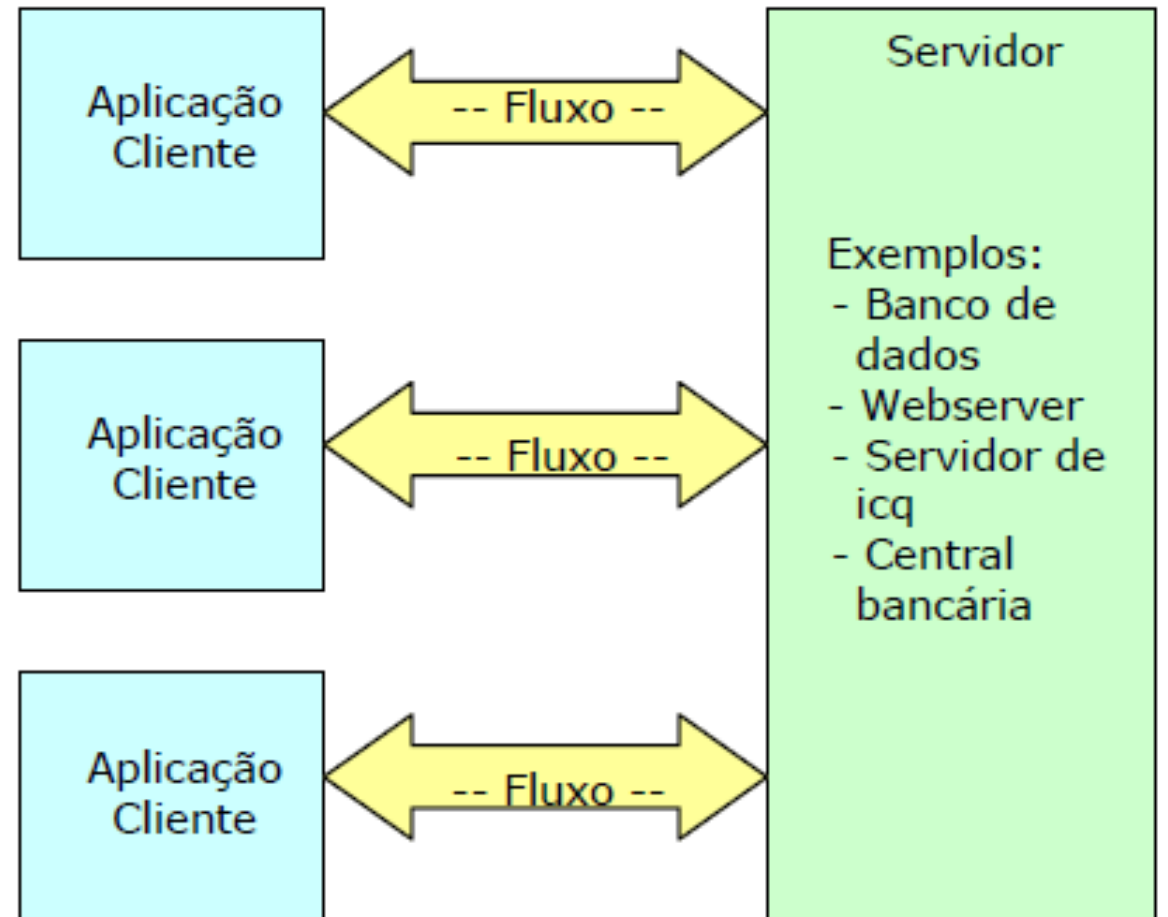
Linguagem de Programação III (LP35A)

Comunicação entre máquinas: sockets

Protocolo

Conectando-se a máquinas remotas

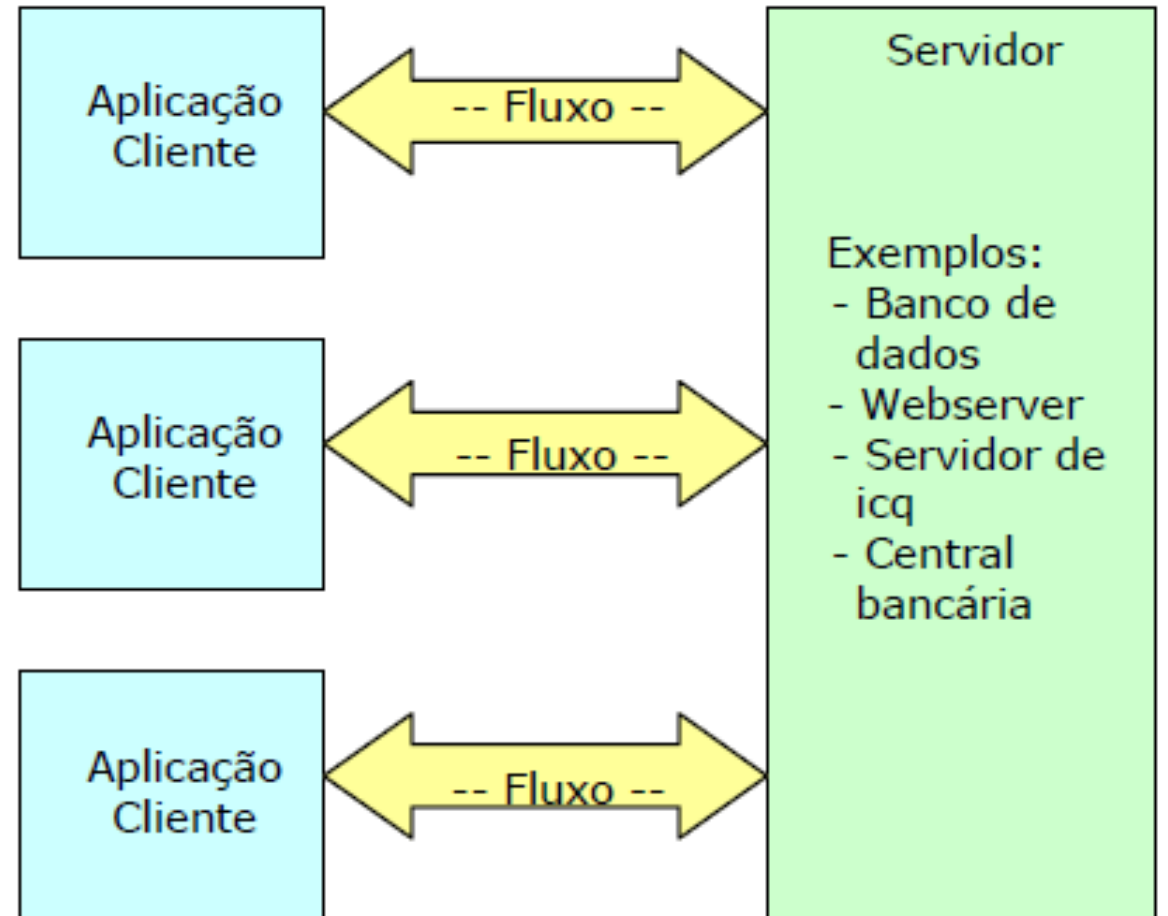
- Da necessidade de dois computadores se comunicarem, surgiram diversos protocolos que permitissem tal troca de informação: o protocolo que iremos estudar aqui é o **TCP** (Transmission Control Protocol).
- Através do **TCP** é possível criar um **fluxo** entre dois computadores como é mostrado no diagrama:



Protocolo

Conectando-se a máquinas remotas

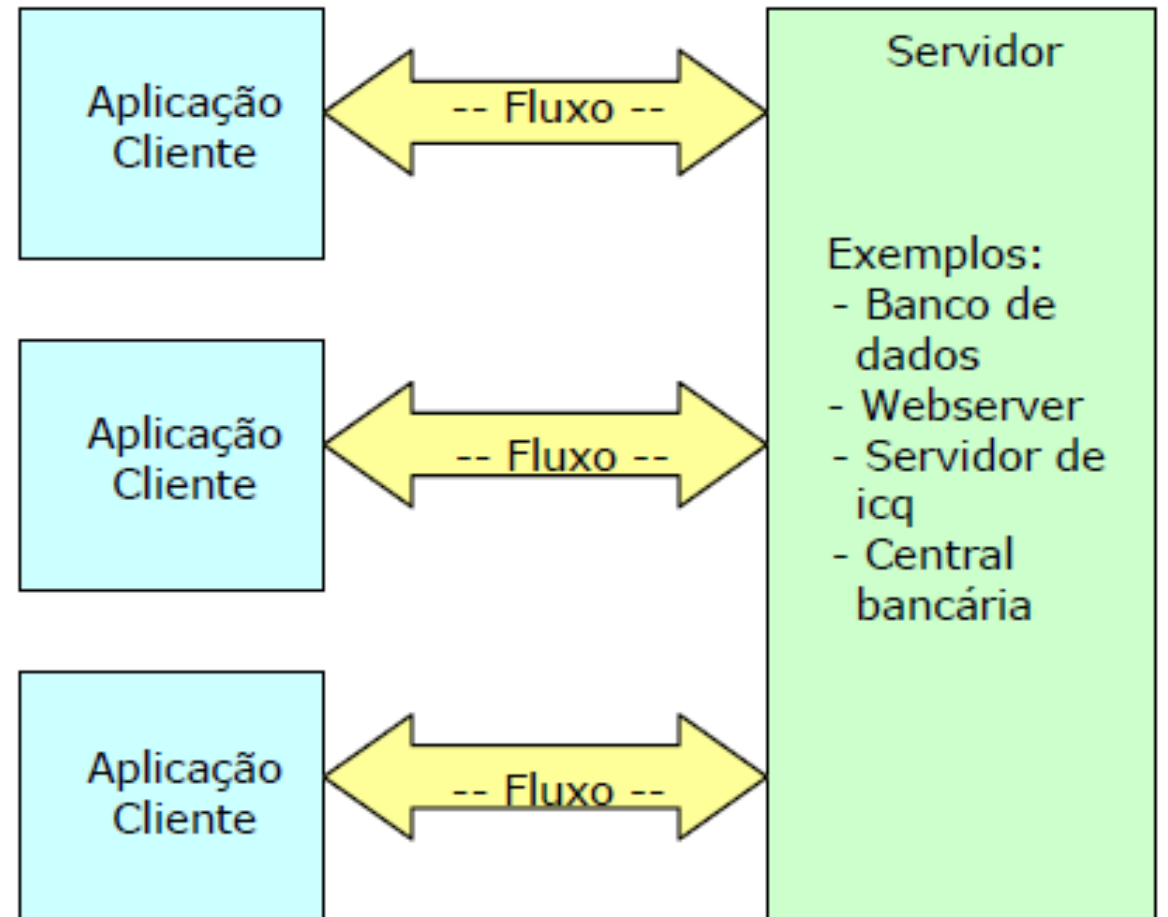
- É possível **conectar mais de um cliente ao mesmo servidor**, como é o caso de diversos banco de dados, webservers etc.
- Ao escrever um **programa em Java que se comunique com outra aplicação**, não é necessário se preocupar com um nível tão baixo quanto o protocolo.
- As classes que trabalham com eles já foram disponibilizadas para serem usadas no pacote java.net



Portas

Conectando-se a máquinas remotas

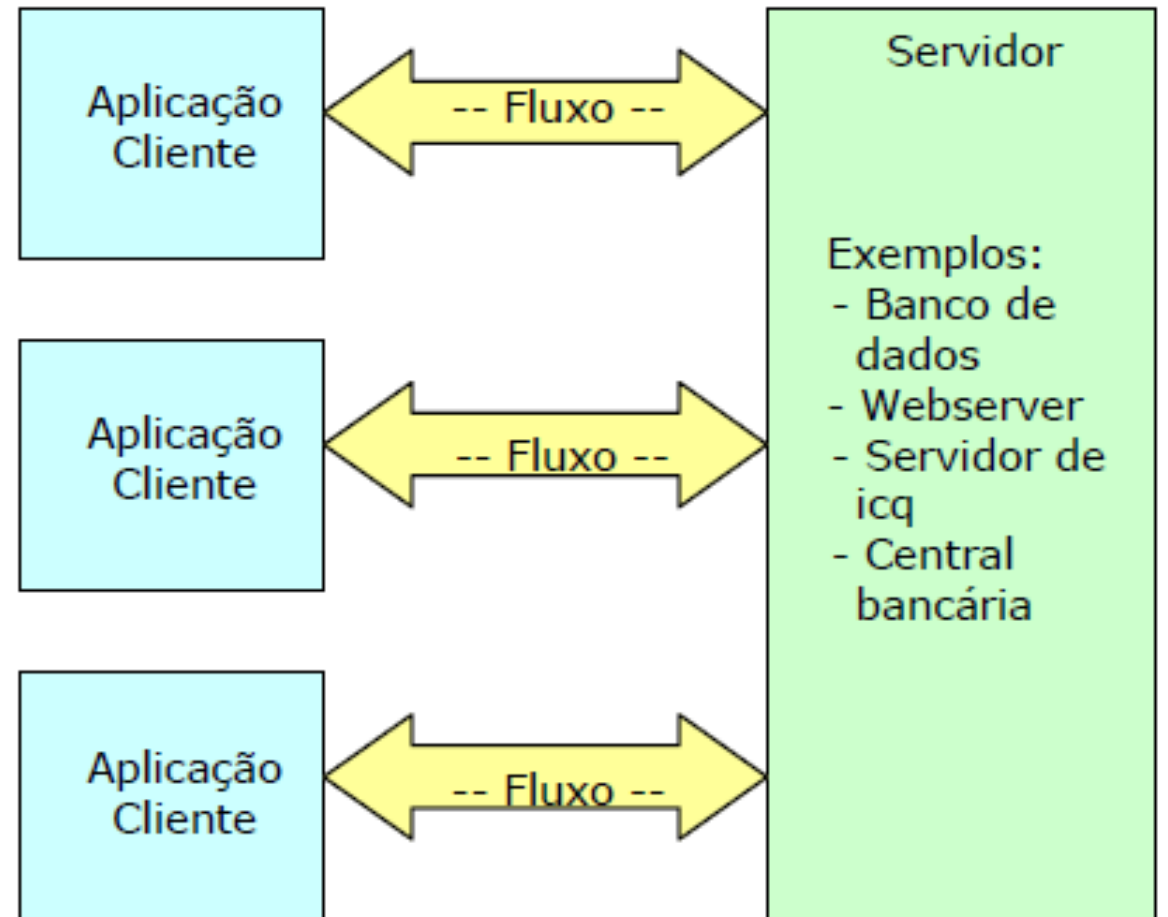
- Na realidade é muito comum encontrar máquinas clientes com uma só conexão física.
- Então como é possível se conectar a dois pontos?
Como é possível ser conectado por diversos pontos?



Portas

Conectando-se a máquinas remotas

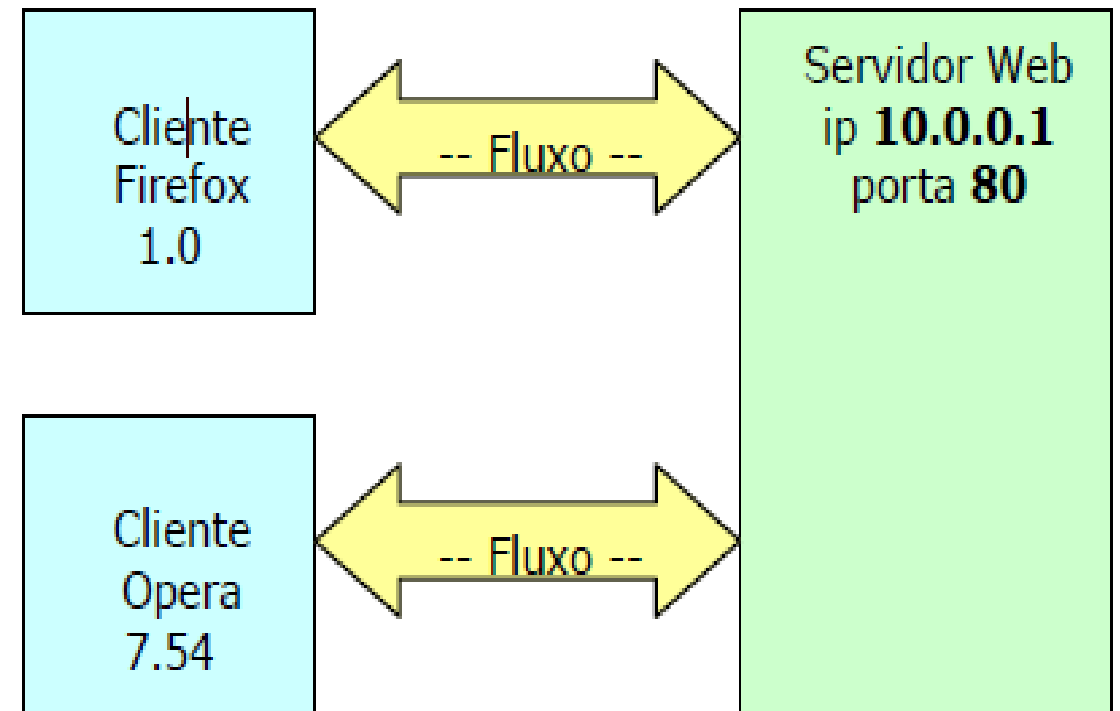
- Todas as aplicações que estão enviando e recebendo dados fazem isso através da mesma conexão física **mas o computador consegue discernir durante a chegada de novos dados quais informações pertencem a qual aplicação**, mas como?



Portas

Conectando-se a máquinas remotas

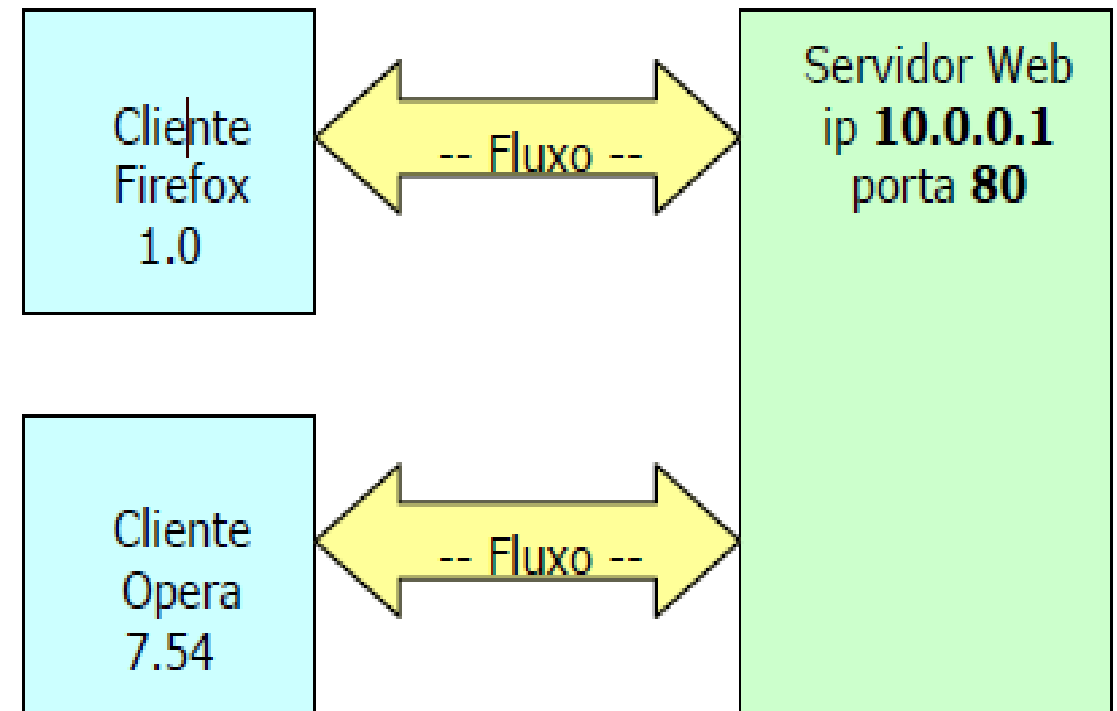
- Assim como existe o IP para indentificar uma máquina, a porta é a solução para indentificar diversas aplicações em uma máquina.
- **Esta porta é um número de 2 bytes, varia de 0 a 65535.**
- Se todas as portas de uma máquina estiverem ocupadas não é possível se conectar a ela enquanto nenhuma for liberada.



Portas

Conectando-se a máquinas remotas

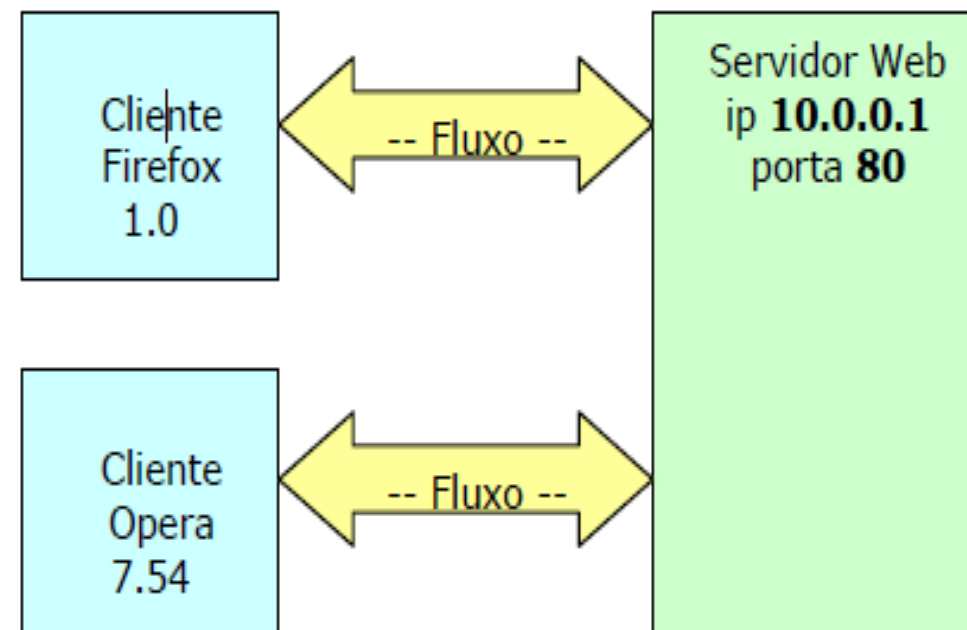
- Ao configurar um servidor para rodar na porta 80 (padrão http), é possível se conectar a esse servidor através dessa porta, que junto com o ip vai formar o endereço da aplicação.
- Por exemplo, o servidor web da xxx.com.br pode ser representado por: xxx.com.br:80



Sockets

Conectando-se a máquinas remotas

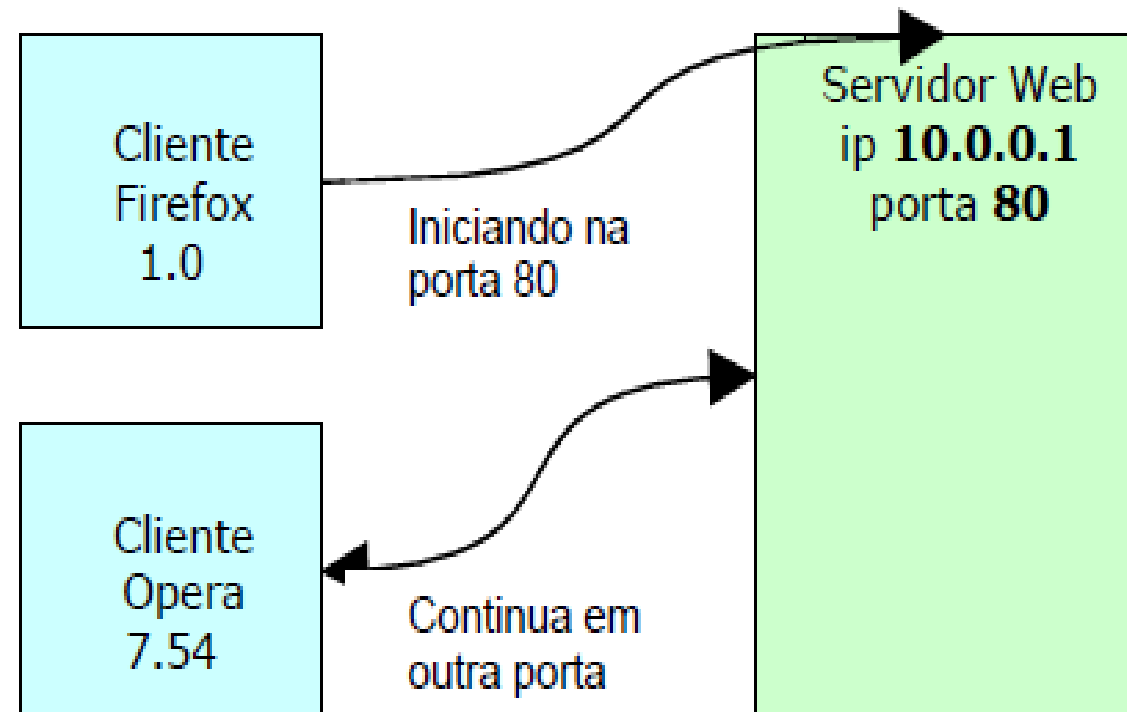
- Mas se um cliente se conecta a um programa rodando na porta 80 de um servidor, enquanto ele não se desconectar dessa porta será impossível que outra pessoa se conecte?



Sockets

Conectando-se a máquinas remotas

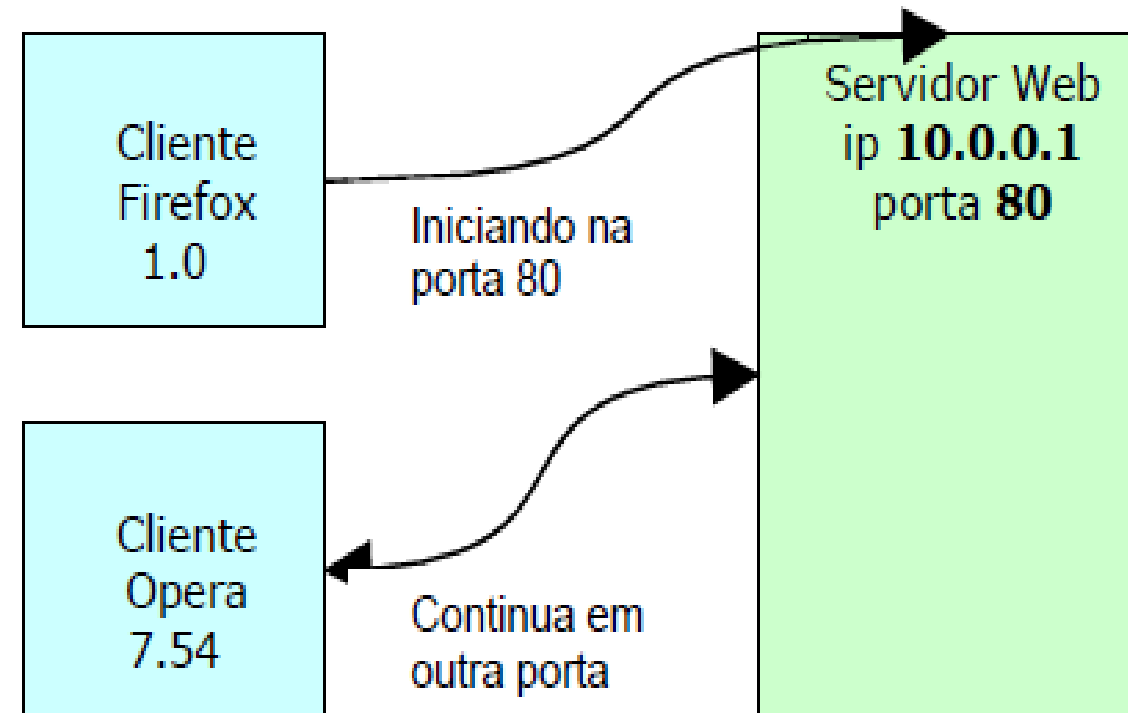
- Acontece que ao efetuar a conexão, ao aceitar a conexão, o servidor redireciona o cliente de uma porta para outra, liberando novamente sua porta inicial e permitindo que outros clientes se conectem novamente



Sockets

Conectando-se a máquinas remotas

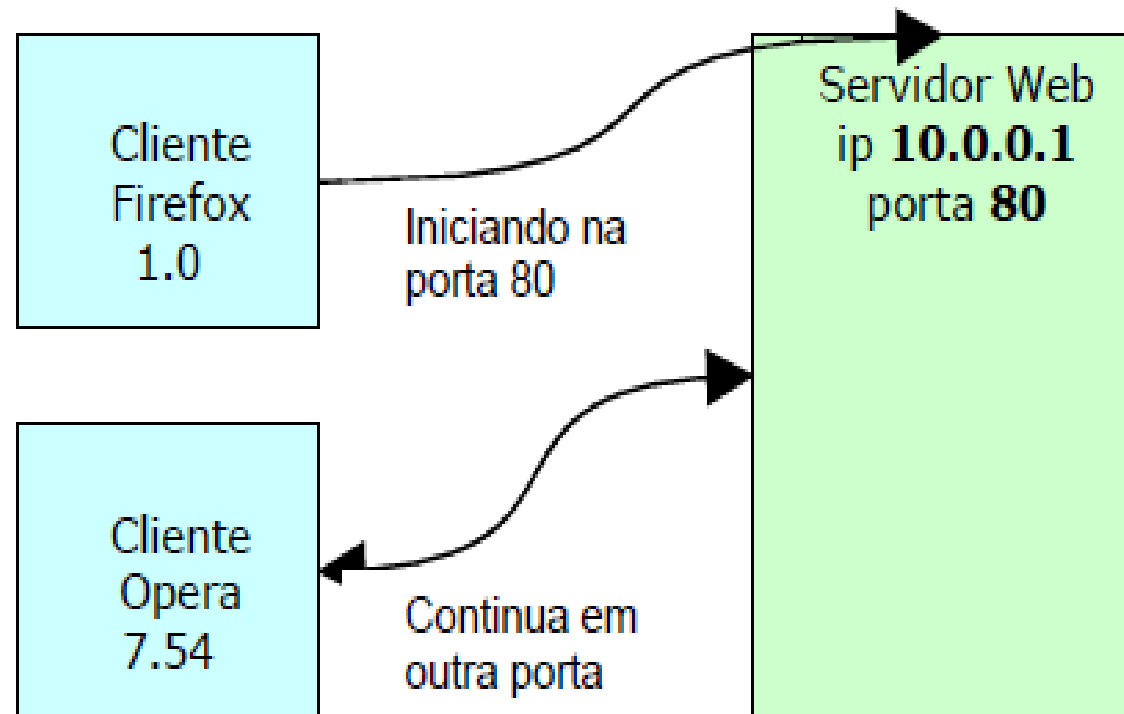
- Em Java, isso deve ser feito através de threads e o processo de aceitar a conexão deve ser rodado o mais rápido possível.



Sockets

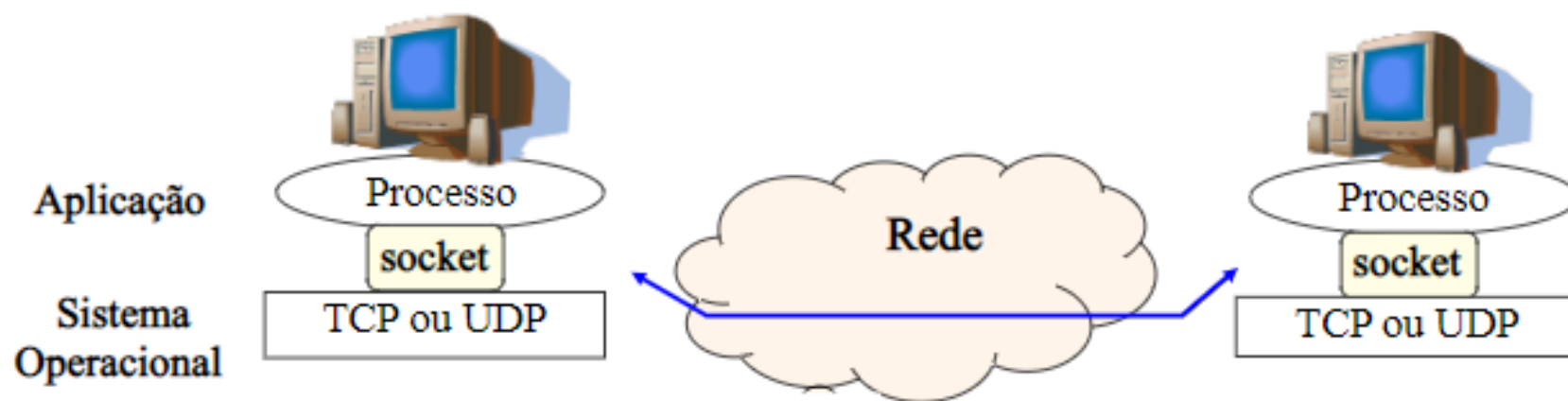
Conectando-se a máquinas remotas

Um Socket é uma combinação de um computador com o número da porta. Essa combinação identifica uma “porta” de entrada para uma aplicação.



Sockets

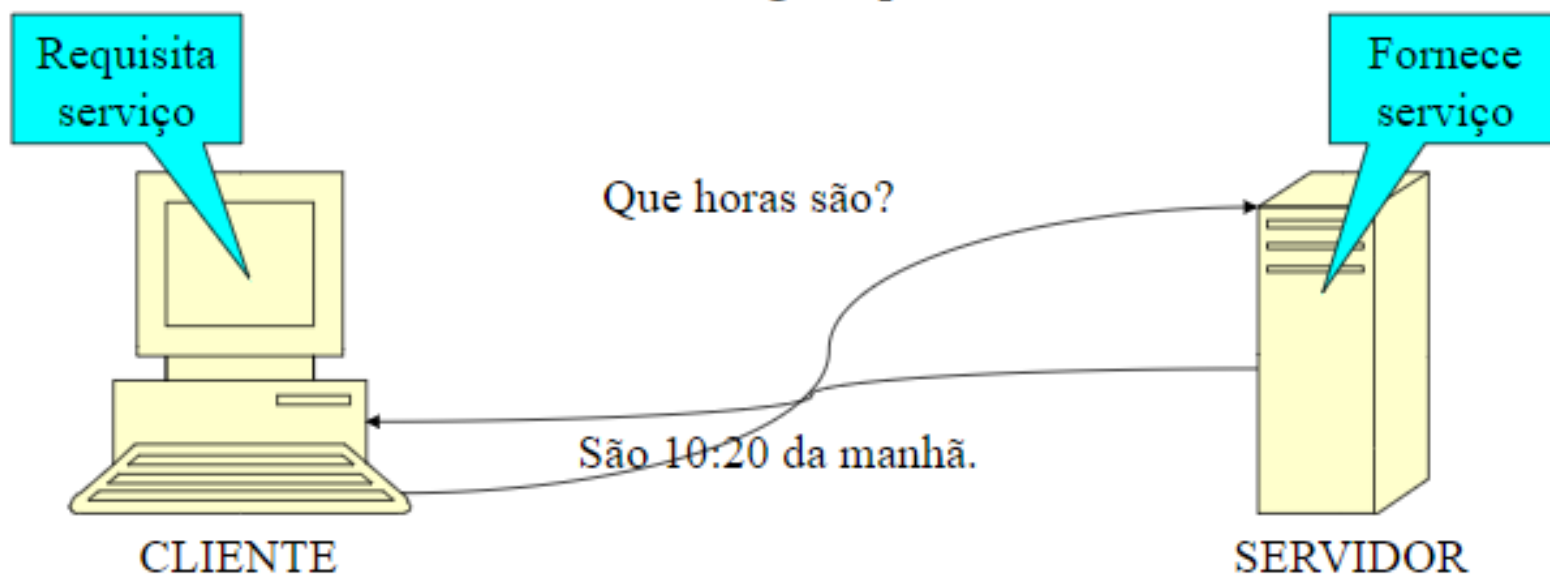
- São abstrações que representam pontos de comunicação através dos quais processos se comunicam;
- Para que dois computadores possam trocar informações
 - Cada um utiliza um *socket*.



Sockets

Cliente/Servidor

- Os *sockets* adotam o paradigma **cliente/servidor** .
 - Um computador é o **Servidor**: abre o *socket* e fica na escuta, a espera de mensagens ou pedidos de conexões dos clientes
 - O outro é o **Cliente**: envia mensagens para o *socket* servidor



Linguagem de Programação III (LP35A)

Usando socket em java

Usando socket em java

Usando *Sockets* em Java

- Pacote `java.net`;
- Principais classes:
 - TCP: `Socket` e `ServerSocket`;
 - UDP: `DatagramPacket` e `DatagramSocket`;
 - *Multicast*: `DatagramPacket` e `MulticastSocket`.
- Este pacote também contém classes que fornecem suporte a manipulação de URLs e acesso a serviços HTTP, entre outras coisas.

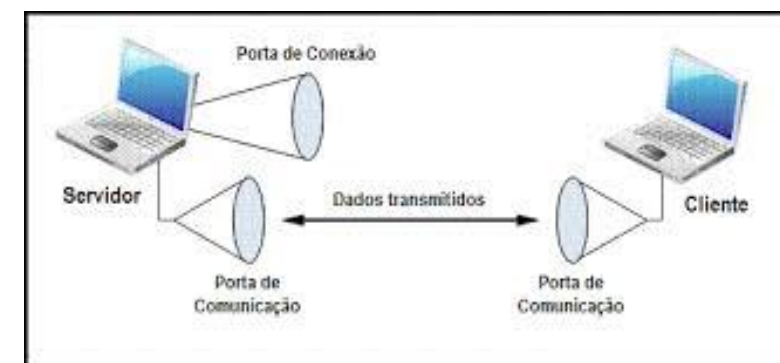
Usando socket em java

*Sockets*TCP

- Clientes e servidores são **diferentes**.
 - Servidor usa a classe `ServerSocket` para “escutar” uma porta de rede da máquina a espera de requisições de conexão.
 - Clientes utilizam a classe `Socket` para requisitar conexão a um servidor específico e então transmitir dados.

```
ServerSocket sSocket = new ServerSocket(12345,5);
```

```
Socket cSocket = new Socket("213.13.45.2",12345);
```



Usando socket em java

Servidor TCP

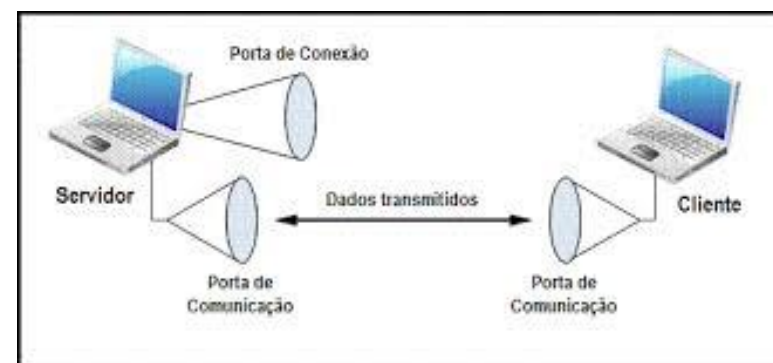
- Inicialmente, um servidor deve criar um *socket* que associe o processo a uma porta do *host*,

```
ServerSocket sSocket = new ServerSocket(porta, backlog);
```

porta : número da porta que o *socket* deve esperar requisições;

backlog : tamanho máximo da fila de pedidos de conexão que o sistema pode manter para este *socket*.

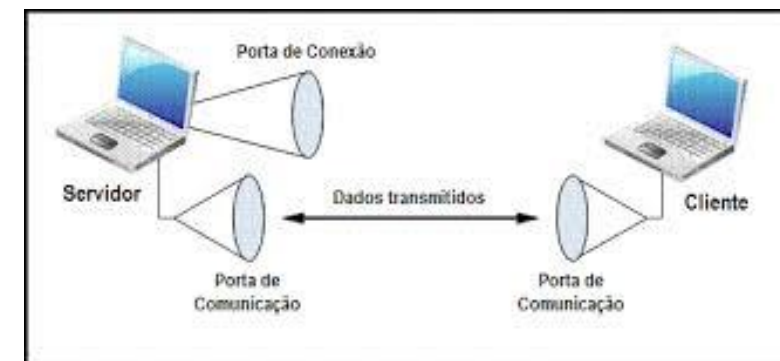
- O *backlog* permite que requisições sejam enfileiradas (em estado de espera) enquanto o servidor está ocupado executando outras tarefas.
- As requisições são processadas uma a uma
- Se uma requisição de conexão chegar ao *socket* quando esta fila estiver cheia, a conexão é negada.



Usando socket em java

Servidor TCP

- O método **accept** é usado para retirar as requisições da fila.
 - Fica bloqueado até que um cliente solicite uma requisição
- O método **accept** retorna uma conexão com o cliente



Usando socket em java

Servidor TCP

- Quando do recebimento da conexão, o servidor pode interagir com o cliente através da leitura e escrita de dados no *socket*.
- A comunicação em si é feita com o auxílio de classes tipo *streams*, que são classes do pacote `java.io`.

- *Stream* de Leitura:

```
DataInputStream in = new  
    DataInputStream(cliente.getInputStream());
```

Este *stream* tem vários métodos *read* (ver pacote `java.io`).

- *Stream* de Escrita:

```
DataOutputStream out = new  
    DataOutputStream(socket.getOutputStream());
```

Este *stream* tem vários métodos *write* (ver pacote `java.io`).

Usando socket em java

Servidor TCP

- Encerramento da conexão.

- Com um cliente em específico:

```
socket.close();
```

- Do *socket* servidor (terminando a associação com a porta do servidor):

```
sSocket.close();
```

```
ServerSocket serverSocket = new ServerSocket(port, backlog);  
  
do{  
    //aguarda conexão  
    Socket socket = serverSocket.accept();  
    //obtem o stream de entrada e o encapsula DataInputStream  
    dataInput =  
        new DataInputStream(socket.getInputStream());  
    //obtem o stream de saída e o encapsula  
    DataOutputStream dataOutput =  
        new DataOutputStream(socket.getOutputStream());  
    //executa alguma coisa... no caso, um eco.  
    String data = dataInput.readUTF();  
    dataOutput.writeUTF(data);  
    //fecha o socket  
    socket.close();  
}while(notExit());  
  
serverSocket.close();
```

Usando socket em java

Cliente TCP

- **Inicialmente** , o cliente deve criar um *socket* especificando o endereço e a porta do serviço a ser acessado:

```
Socket socket = new Socket(host, porta);
```

host é endereço ou nome do servidor

porta é o número da porta em que o servidor está escutando

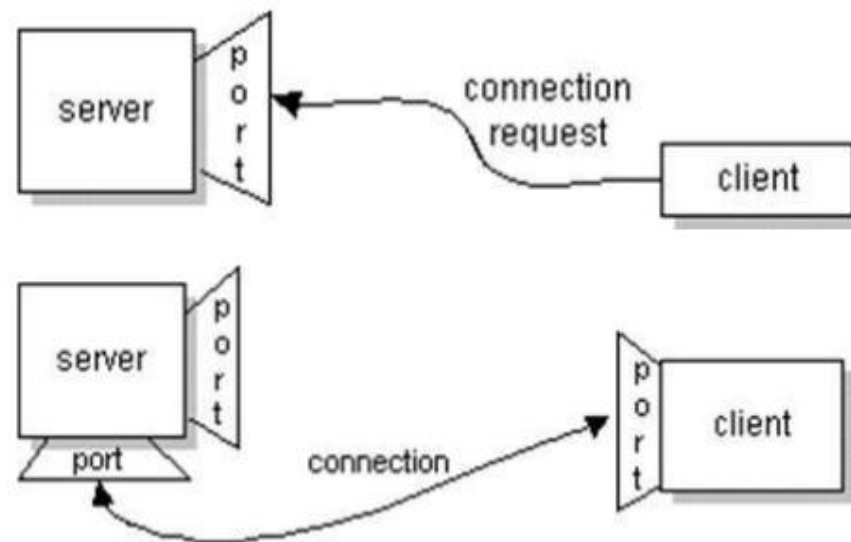
Esta chamada representa a requisição de uma conexão com o servidor. Se o construtor for executado sem problemas, a conexão está estabelecida.

- **A partir daí** , da mesma forma que no servidor, o cliente pode obter os *streams* de entrada e saída.

```
DataInputStream in = new  
DataInputStream(cliente.getInputStream());
```

```
DataOutputStream out = new  
DataOutputStream(socket.getOutputStream());
```

- **Ao final**, o *socket* é fechado da mesma forma que no servidor.



Usando socket em java

Cliente TCP

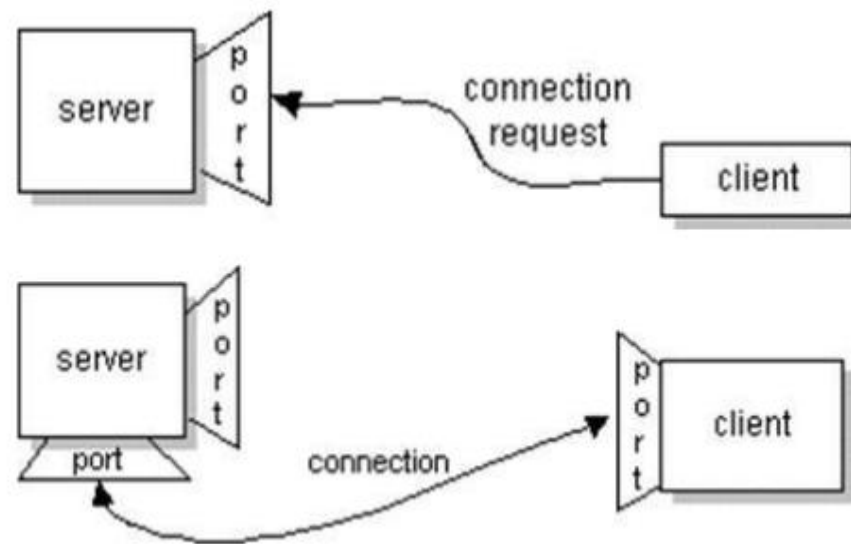
```
InetAddress address = InetAddress.getByName(name);

Socket serverSocket = new Socket(address,port);

//obtem o stream de saída e o encapsula
DataOutputStream dataOutput =
new DataOutputStream(socket.getOutputStream());
//obtem o stream de entrada e o encapsula
DataInputStream dataInput =
new DataInputStream(socket.getInputStream());

//executa alguma coisa... no caso, envia uma mensagem
//e espera resposta.
dataOutput.writeUTF(request);
String response = dataInput.readUTF();

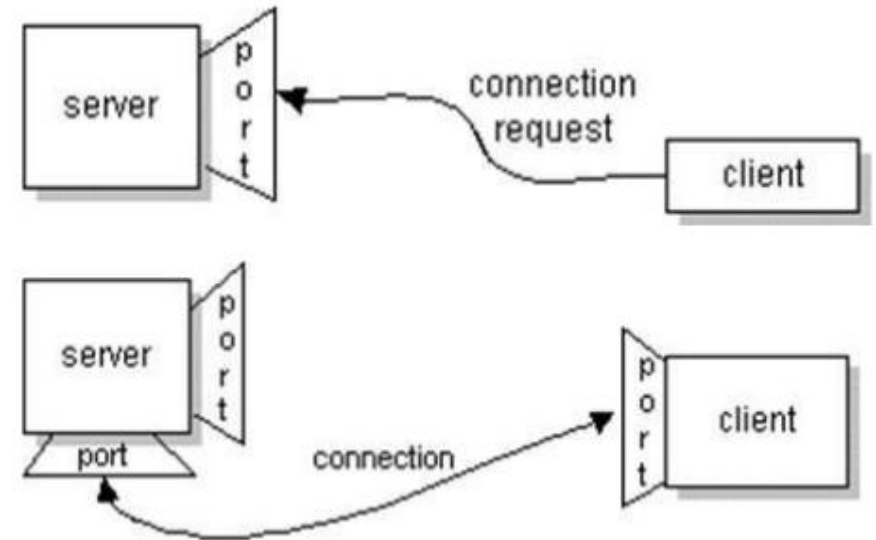
//fecha o socket
socket.close();
```



Usando socket em java

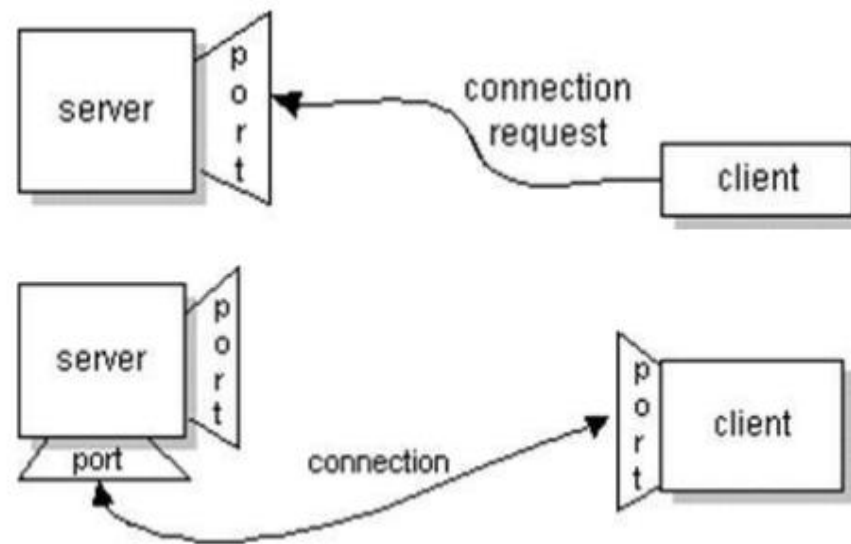
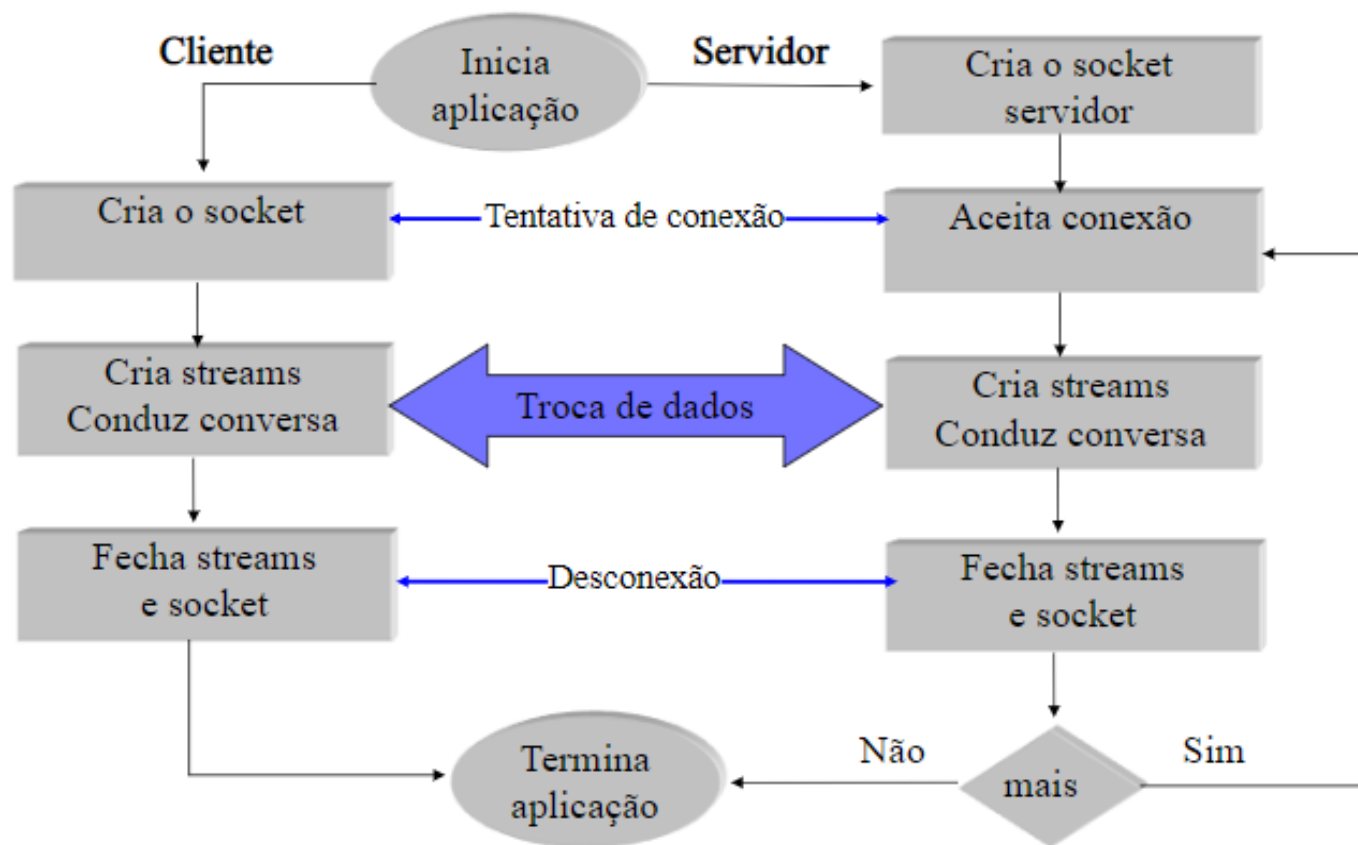
Resumo – *Sockets*TCP

- Servidor
 - Cria o *socket* servidor e aguarda conexão
 - Usa método `accept()` para pegar novas conexões
 - Cria *streams* entrada/saída para o *socket* da conexão
 - Faz a utilização dos *streams* conforme o protocolo
 - Fecha os *streams*
 - Fecha *socket* da conexão
 - Repete várias vezes
 - Fecha o *socket* servidor
- Cliente
 - Cria o socket com conexão cliente
 - Associa *streams* de leitura e escrita com o socket
 - Utiliza os *streams* conforme o protocolo do servidor
 - Fecha os *streams*
 - Fecha o *socket*



Usando socket em java

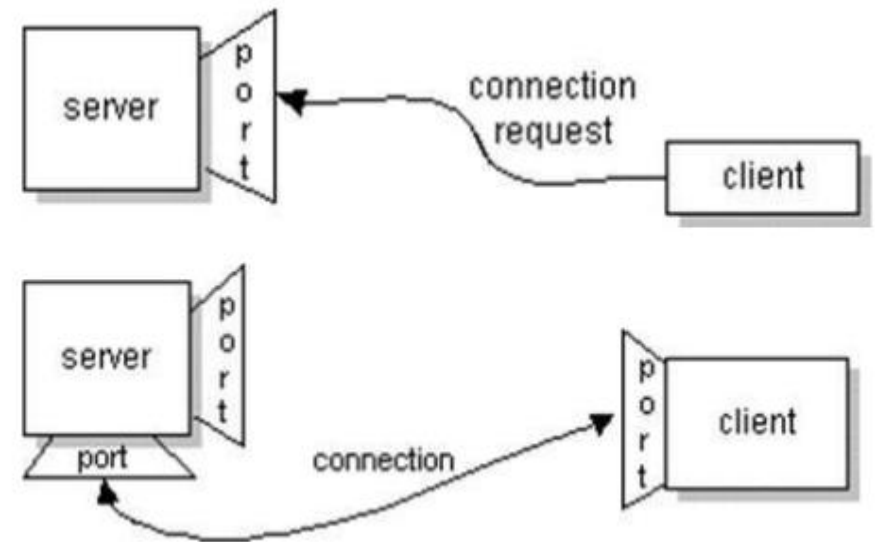
Resumo – Sockets TCP



Usando socket em java

Resumo – *Sockets* TCP

- Servidor apresentado processa uma conexão de cada vez
- Servidor concorrente
 - Cria uma nova thread para cada conexão aberta
 - Consegue processar várias conexões simultaneamente



Usando socket em java

*Sockets*UDP

- A comunicação UDP é feita através de duas classes:

- A **mesma** classe é usada pelo cliente e pelo servidor

- Classe *DatagramSocket*:

- *Socket*servidor:

```
DatagramSocket socket = new DatagramSocket(porta);
```

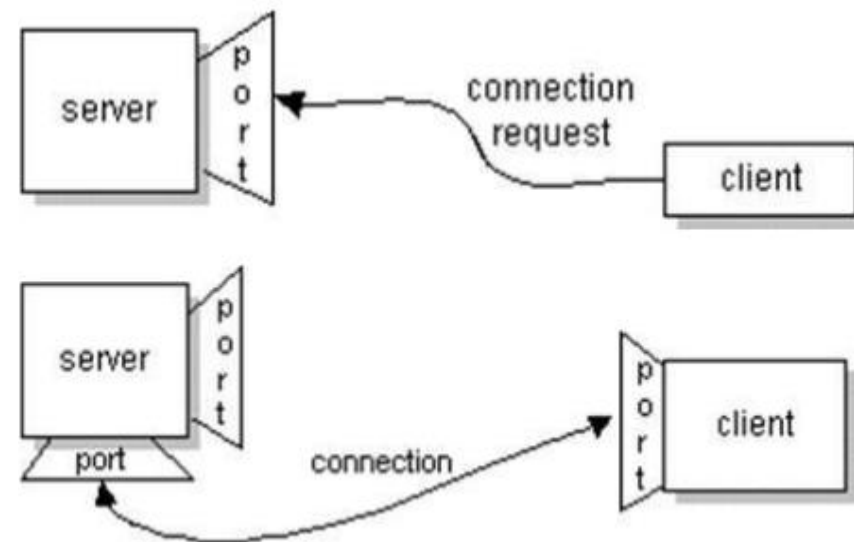
O **servidor** deve especificar sua porta. **Omissão** → próxima porta livre.

- *Socket*cliente:

```
DatagramSocket socket = new DatagramSocket();
```

- Classe *DatagramPacket*

- As comunicações ocorrem através da troca de **datagramas**
- Os datagramas contêm:
 - os dados a serem enviados e recebidos
 - endereço de destino/origem do datagrama.



Usando socket em java

Servidor UDP

- Inicialmente o servidor deve criar um *socket* que o associe a uma porta da máquina.
 - `DatagramSocket socket = new DatagramSocket(porta);`
 - `porta`: número da porta que o socket deve esperar requisições;
- Depois do *socket* criado, o servidor fica bloqueado até o recebimento de um datagrama (método *receive*)

```
byte[] buffer = new byte[n];    \{array de bytes}  
DatagramPacket dg = new DatagramPacket(buffer,n);  
socket.receive(dg);
```

- Os dados recebidos devem caber no buffer do datagrama. Desta forma, protocolos mais complexos baseados em datagramas devem definir cabeçalhos e mensagens de controle.
- Fechamento do *socket* `socket.close();`

Usando socket em java

Servidor UDP

- Inicialmente o servidor deve criar um *socket* que o associe a uma porta da máquina.
 - `DatagramSocket socket = new DatagramSocket(porta);`
 - `porta`: número da porta que o socket deve esperar requisições;
- Depois do *socket* criado, o servidor fica bloqueado até o recebimento de um datagrama (método *receive*)

```
byte[] buffer = new byte[n];    \{array de bytes}  
DatagramPacket dg = new DatagramPacket(buffer,n);  
socket.receive(dg);
```

- Os dados recebidos devem caber no buffer do datagrama. Desta forma, protocolos mais complexos baseados em datagramas devem definir cabeçalhos e mensagens de controle.
- Fechamento do *socket* `socket.close();`

Usando socket em java

Servidor UDP

- Para enviar dados para um *socket* datagrama
 - Necessário um endereço, **classe InetAddress**
 - `InetAddress addr = InetAddress.getByName("alambique.das.ufsc.br");`
- O envio de datagramas é realizado de forma bastante simples:

```
String toSend = "Este eh o dado a ser enviado!";
byte data[] = toSend.getBytes();
DatagramPacket sendPacket =
    new DatagramPacket(data, data.length, host, porta);
socket.send(sendPacket);
```
- **A String** deve ser convertida para **array de bytes**.
- Endereço destino (*host + porta*) é incluído no **DatagramPacket**.
- Como saber quem enviou o pacote ?
 - `pacoteRecebido.getAddress();`
 - `pacoteRecebido.getPort();`

```
DatagramSocket socket = new DatagramSocket(port);

do{
    //recebimento dos dados em um buffer de 1024 bytes
    DatagramPacket recPacket = new DatagramPacket(
        new byte[1024], 1024);
    socket.receive(recPacket); //recepção

    //envio de dados para o emissor do datagrama recebido
    DatagramPacket sendPacket = new DatagramPacket(
        recPacket.getData(), recPacket.getData().length,
        recPacket.getAddress(), recPacket.getPort());
    socket.send(sendPacket); //envio
}while(notExit());

socket.close();
```


Usando socket em java

Cliente UDP

- Inicialmente o cliente deve criar um *socket*.

```
DatagramSocket socket = new DatagramSocket();
```

- **Opcional:** o cliente pode conectar o *socket* a um servidor específico, de tal forma que todos os seus datagramas enviados terão como destino esse servidor.

```
socket.connect(host,porta);
```

Parâmetros: **host** é endereço ou nome do servidor e **porta** é o número da porta em que o servidor espera respostas.

Executando o *connect*, o emissor não necessita mais definir endereço e porta destino para cada datagrama a ser enviado.

- A recepção e o envio de datagramas, bem como o fechamento do *socket*, ocorrem da mesma forma que no servidor.

```
InetAddress address = InetAddress.getByName(name);

DatagramSocket socket = new DatagramSocket();
//socket.connect(address,port);

byte[] req = ...
//envio de dados para o emissor do datagrama recebido
DatagramPacket sendPacket = new
DatagramPacket(req, req.length, address, port);
//DatagramPacket dgl = new DatagramPacket(req, req.length);
socket.send(sendPacket); //envio

//recebimento dos dados em um buffer de 1024 bytes
DatagramPacket recPacket = new DatagramPacket(
new byte[1024], 1024);
socket.receive(recPacket); //recepção
byte[] resp = recPacket.getData();

socket.close();
```


Usando socket em java

Resumo – *Sockets*UDP

- **Cliente**

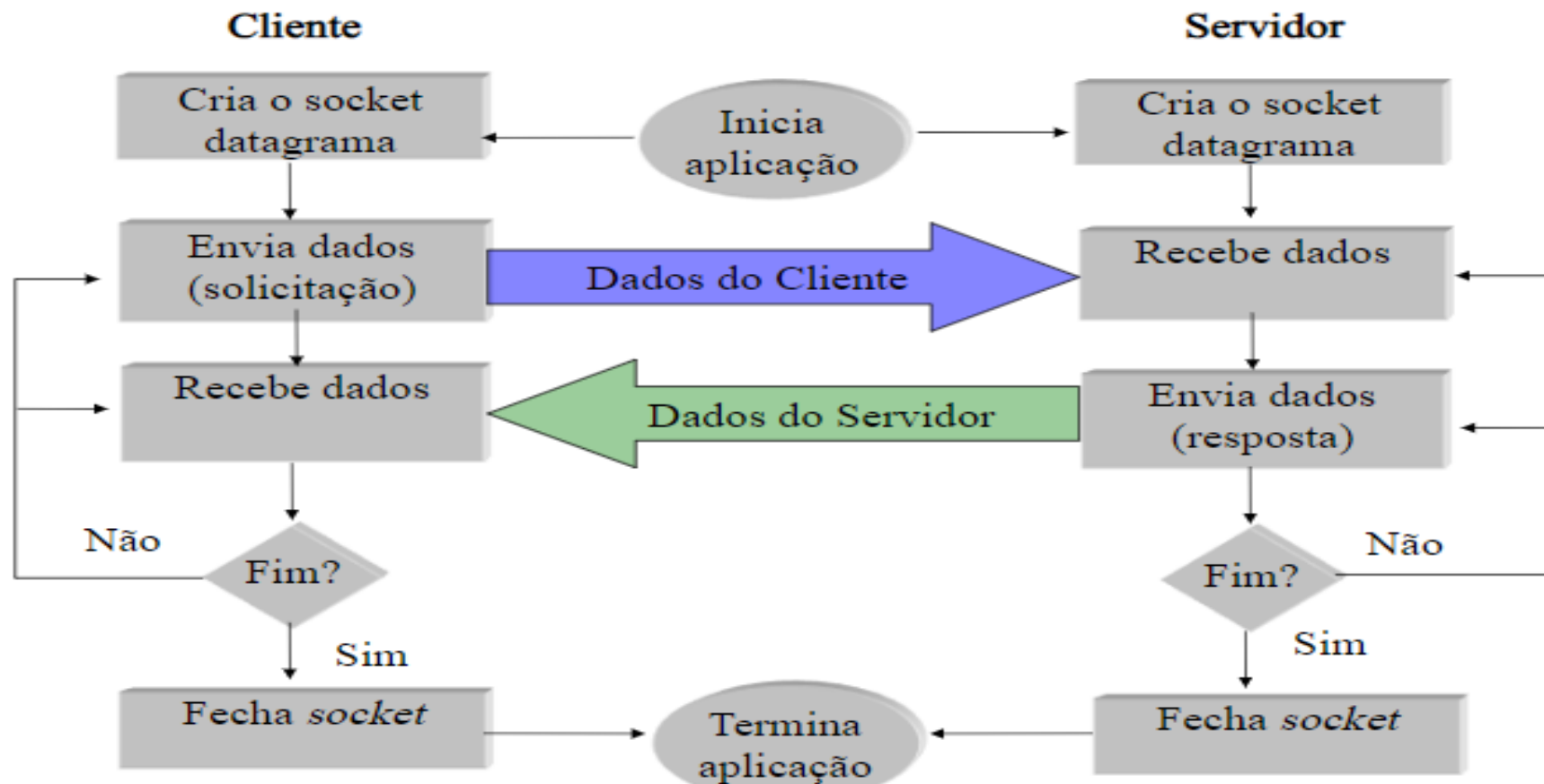
- Cria um *socket* datagrama em qualquer porta livre
- Cria um endereço destinatário
- Envia os dados conforme o protocolo em questão
- Espera por dados com a resposta
- Repete várias vezes
- Fecha o socket datagrama

- **Servidor**

- Cria um *socket* datagrama em uma porta específica
- Chama `receive()` para esperar por pacotes
- Responde ao pacote recebido conforme protocolo em questão
- Repete várias vezes
- Fecha o socket

Usando socket em java

Resumo – *Sockets*UDP



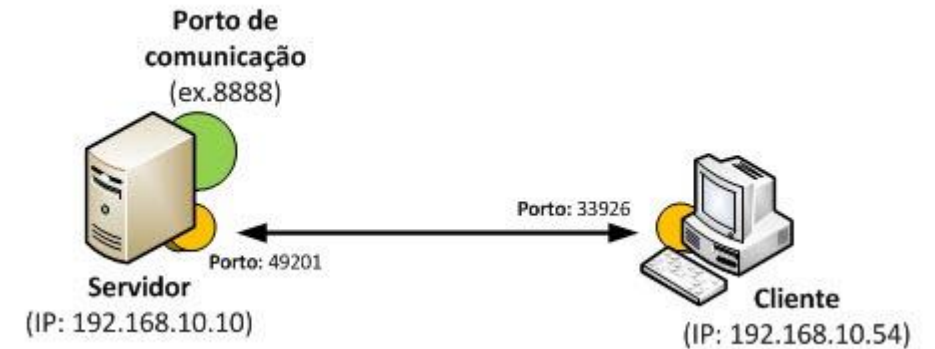
Linguagem de Programação III (LP35A)

Threads em aplicações de rede

Threads em aplicações de redes

Conectando-se a máquinas remotas

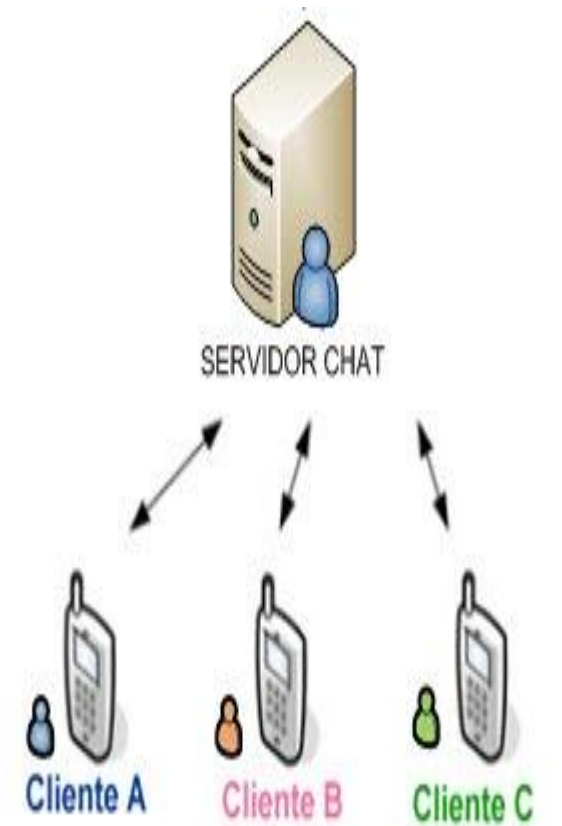
- A comunicação em rede requer a abertura de sockets bidirecionais entre um cliente e um servidor.
- Tipicamente, o servidor inicia uma aplicação que executa continuamente e bloqueia um thread onde espera a conexão de um cliente.
- Quando o cliente responde, o thread é liberado e o servidor processa os dados recebidos.



Threads em aplicações de redes

Conectando-se a máquinas remotas

- Para que um servidor possa **esperar outros clientes enquanto processa os dados do cliente conectado**, ele cria um novo thread para cada cliente ativo, e mantém o thread principal esperando.
- O cliente, por sua vez, **também emprega threads separados para a conexão de rede**, para que possa realizar outras tarefas enquanto espera dados chegarem do servidor.




Threads em aplicações de redes

- O código ao lado mostra **um servidor TCP/IP simples** que espera clientes na porta 9999.
 1. O método `accept()` de `ServerSocket` põe o thread em estado `WAITING`.
 2. A chegada de um cliente causa a notificação do thread que acorda e cria um thread novo (`RunnableWorker`) para processar o cliente.

Na sequência, o thread principal bloqueia novamente esperando o próximo cliente.

```
public class MultithreadedServerSocketExample {  
    public static void main(String[] args) throws IOException {  
        try (ServerSocket server = new ServerSocket(9999)) {  
            while (true) {  
                System.out.println("Server waiting for client.");  
                1 Socket client = server.accept(); // blocks  
                System.out.println("Client from " + client.getLocalAddress() + " connected.");  
                2 new Thread(new RunnableWorker(client, server)).start();  
            }  
        }  
    }  
}
```



O `RunnableWorker` é uma implementação de `Runnable` que contém uma referência para o `Socket` obtido pelo cliente. **Código descrito a seguir:**

Linguagem de Programação III (LP35A)

Iniciando um servidor e um cliente (outro exemplo – passo a passo)

Servidor/Cliente

SERVIDOR

Iniciando agora um modelo de servidor de chat, **o serviço do computador que funciona como base deve primeiro abrir uma porta** e ficar ouvindo até alguém tentar se conectar.

```
1. import java.net.*;
2.
3. public class Servidor {
4.
5.     public static void main(String args[]) {
6.
7.         try {
8.             ServerSocket servidor = new ServerSocket(18981);
9.             System.out.println("Porta 18981 aberta!");
10.            // a continuação do servidor deve ser escrita aqui
11.        } catch(IOException e) {
12.            System.out.println("Ocorreu um erro na conexão");
13.            e.printStackTrace();
14.
15.        }
16.
17.    }
18. }
```

Servidor/Cliente

SERVIDOR

- Se o objeto for realmente criado significa que a porta 18981 estava fechada e foi aberta.
- Se outro programa possui o controle desta porta neste instante, é normal que o nosso exemplo não funcione pois ele não consegue utilizar uma porta que já está em uso.

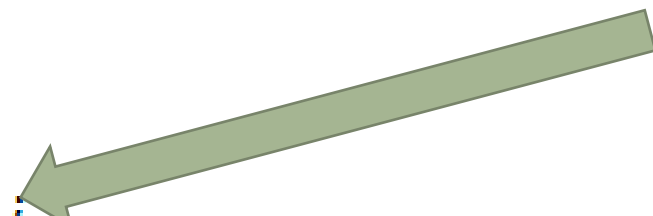
```
1. import java.net.*;
2.
3. public class Servidor {
4.
5.     public static void main(String args[]) {
6.
7.         try {
8.             ServerSocket servidor = new ServerSocket(18981);
9.             System.out.println("Porta 18981 aberta!");
10.            // a continuação do servidor deve ser escrita aqui
11.        } catch(IOException e) {
12.            System.out.println("Ocorreu um erro na conexão");
13.            e.printStackTrace();
14.
15.        }
16.
17.    }
18. }
```


Servidor/Cliente

SERVIDOR

- Após abrir a porta, precisamos esperar por um cliente através do método accept da ServerSocket.

```
Socket cliente = servidor.accept();  
System.out.println("Nova conexão com o cliente " +  
    cliente.getInetAddress().getHostAddress()  
);
```



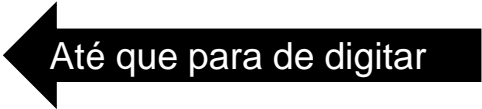
- Assim que um cliente se conectar o programa irá continuar.

Servidor/Cliente

SERVIDOR

- Por fim, basta ler todas as informações que o cliente nos enviar:

```
BufferedReader in = new BufferedReader(  
    new InputStreamReader(cliente.getInputStream())  
);  
  
while (true) {  
  
    String linha = in.readLine();  
    if (linha == null) {  
        break;  
    }  
  
    System.out.println(linha);  
  
}
```



Até que para de digitar

Servidor/Cliente

SERVIDOR

- Agora fechamos as conexões, começando pelo fluxo:

```
in.close();
cliente.close();
servidor.close();
```
- Ao fechar a primeira conexão, o servidor escrito acima encerra as operações e termina.
- Para receber uma nova conexão após o término da primeira basta criar um loop que começa ao receber uma nova conexão e termina quando a mesma é fechada.
- Este loop não irá implementar o recurso de tratamento de diversos clientes ao mesmo tempo!

Servidor/Cliente

CLIENTE

- Agora a nossa tarefa é **criar um programa cliente que envie mensagens para o servidor...**
- o cliente é ainda mais simples que o servidor.
- O código a seguir é a parte principal e tenta se conectar a um servidor no ip 127.0.0.1

Servidor/Cliente

CLIENTE

- O primeiro passo é abrir a porta e preparar para ler os dados do cliente
- Caso ocorra algum erro no momento da conexão, envio de mensagem ao final do código.

```
1. import java.net.*;
2.
3. public class Cliente {
4.
5.     public static void main(String args[]) {
6.
7.         try {
8.             // conecta ao servidor
9.
10.            1 Socket cliente = new Socket("127.0.0.1",18981);
11.            System.out.println("O cliente se conectou ao servidor!");
12.
13.            2 // prepara para a leitura da linha de comando
14.            BufferedReader in = new BufferedReader(
15.                new InputStreamReader(System.in)
16.            );
17.
18.            /* inserir o resto do programa aqui */
19.
20.            3 // fecha tudo
21.            cliente.close();
22.
23.        } catch (Exception e) {
24.
25.            // em caso de erro
26.            System.out.println("Ocorreu um erro na conexão");
27.            e.printStackTrace();
28.
29.        }
30.    }
```

Cliente de Chat

- 1 Conecta
- 2 Lê e escreve
- 3 Fecha a conexão

Servidor/Cliente

CLIENTE

- O primeiro passo é abrir a porta e preparar para ler os dados do cliente
- Caso ocorra algum erro no momento da conexão, envio de mensagem ao final do código.

```
1. import java.net.*;
2.
3. public class Cliente {
4.
5.     public static void main(String args[]) {
6.
7.         try {
8.             // conecta ao servidor
9.
10.            1 Socket cliente = new Socket("127.0.0.1",18981);
11.            System.out.println("O cliente se conectou ao servidor!");
12.
13.            2 // prepara para a leitura da linha de comando
14.            BufferedReader in = new BufferedReader(
15.                new InputStreamReader(System.in)
16.            );
17.
18.            /* inserir o resto do programa aqui */
19.
20.            3 // fecha tudo
21.            cliente.close();
22.
23.        } catch (Exception e) {
24.
25.            // em caso de erro
26.            System.out.println("Ocorreu um erro na conexão");
27.            e.printStackTrace();
28.
29.        }
30.    }
```

Cliente de Chat

- 1 Conecta
- 2 Lê e escreve
- 3 Fecha a conexão

Servidor/Cliente

CLIENTE

- Agora basta ler as linhas que o usuário digitar através do buffer de entrada (in) e jogá-las no buffer de saída:

```
PrintWriter out = new PrintWriter(cliente.getOutputStream, true);  
while (true) {  
    String linha = in.readLine();  
    out.println(linha);  
}  
out.close();
```

- **Para testar o sistema, precisamos rodar primeiro o servidor e logo depois o cliente.**
- Tudo o que for digitado no cliente será enviado para o servidor.

Servidor/Cliente

Multithreading

- Para que o servidor **seja capaz de trabalhar com dois clientes ao mesmo tempo** é necessário criar **uma thread** logo após executar o método accept.
- A thread criada **será responsável pelo tratamento dessa conexão**, enquanto o loop do servidor irá disponibilizar a porta para uma nova conexão:

```
while (true) {  
    Socket cliente = servidor.accept();  
  
    // cria um objeto que irá tratar a conexão  
    TratamentoClass tratamento = new TratamentoClass(cliente);  
  
    // cria a thread em cima deste objeto  
    Thread t = new Thread(tratamento);  
  
    // inicia a thread  
    t.start();  
  
}
```

Linguagem de Programação III (LP35A)

Exercícios

Exercícios



Linguagem de Programação III (LP35A)

Thread safety

Orientações Gerais

Atividades aula 03

- 1) Monte todos os códigos-exemplos explorados em aula e execute para fixar os conceitos abordados;
- 2) Faça os exercícios propostos e poste sua solução no GITHUB. Coloque seu nome e matricula nos comentários dos códigos desenvolvidos;
- 3) Os códigos devem ser postados em formato doc ou txt para facilitar o teste em qualquer IDE;
- 4) Você poderá desenvolver os exercícios propostos na IDE que julgar mais confortável;
- 5) Você precisa ter testado anteriormente os exemplos para poder aproveitar o código para os exercícios.

▪ **Bom trabalho!!!**

Exemplos de codigos java

Edit

[Manage topics](#)

🕒 7 commits

🌿 1 branch

📦 0 packages

🏷 0 releases

👤 1 contributor

Branch: master ▾

New pull request

Create new file

Upload files

Find file

Clone or download ▾



elianasantos Add files via upload ...

Latest commit 1752fe3 now

📄 Aula_01.pdf	Add files via upload	now
📄 Aula_02.pdf	Add files via upload	8 minutes ago
📄 Exemplo Interrupt.docx	Add files via upload	12 hours ago
📄 Exemplo Join.docx	Add files via upload	1 hour ago
📄 Exemplo Sleep.docx	Add files via upload	2 hours ago
📄 Exercicios aula 02.pdf	Add files via upload	14 minutes ago

Obrigada!

