Sockets em Java

Luiz Antonio

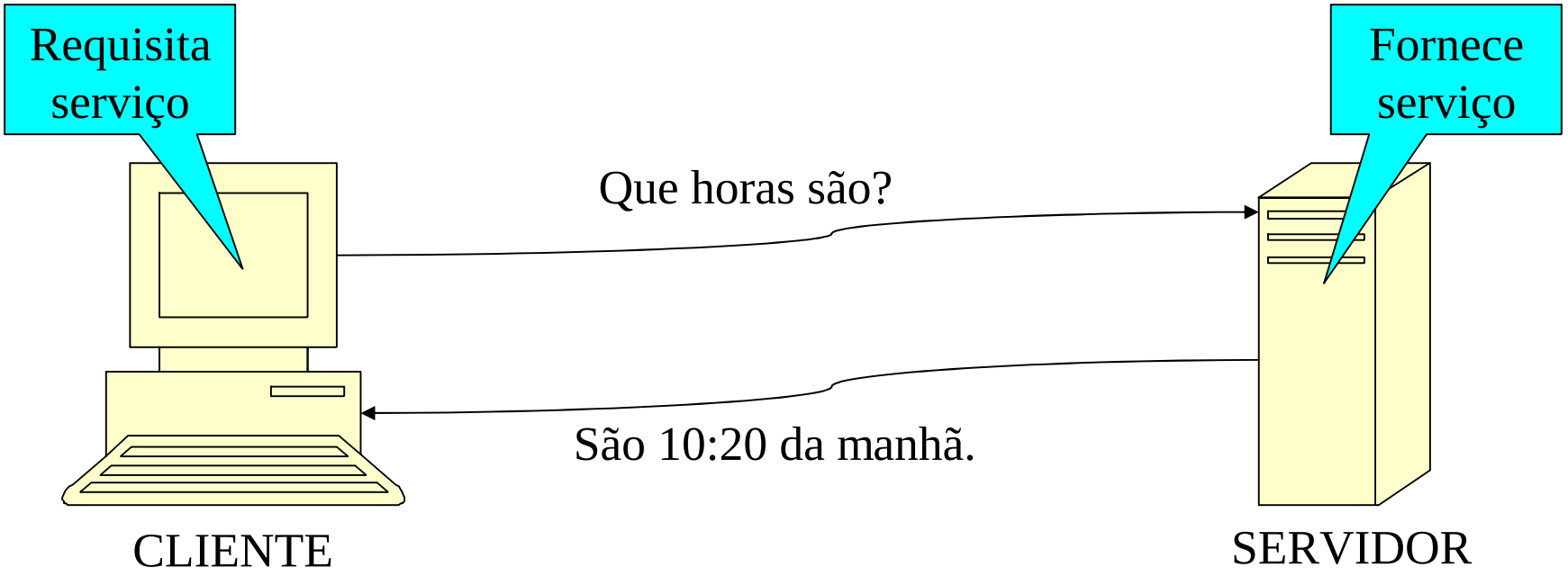
Sumário

- Introdução
- *Sockets* TCP
- *Sockets* UDP
- *Sockets Multicast*
- Referências

O que são *sockets*?

- São uma abstração para endereços de comunicação através dos quais os processos se comunicam;
- Cada endereço desse tem um identificador único composto pelo endereço da máquina e a porta usado pelo processo;
- Para que dois computadores possam manter comunicação, cada um precisa de um *socket*.
- O emprego de *sockets* está geralmente relacionado ao paradigma **cliente/servidor**.

Cliente/Servidor



Os papeis de clientes e servidores não são fixos em processos: um servidor pode ser cliente de outro serviço.

Sockets na Internet

- Endereçando serviços na Internet:
  - Na Internet, todas as máquinas têm um endereço IP;
  - Os serviços em uma máquina são identificados por uma porta;
- O *socket* é criado no momento do ***binding***, quando o processo se associa a um par endereço e porta.
- Para se comunicar com um servidor, um cliente precisa saber o endereço da máquina servidor e o número da porta do serviço em questão.
  - Ex: 200.201.81.50**:21**.

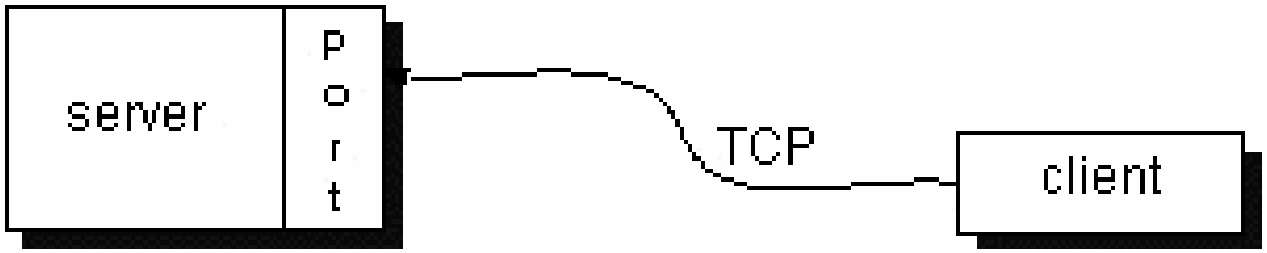
Sockets na Internet

- Comunicação ponto a ponto:
  - Orientado a conexão: TCP (*Transport Control Protocol*);
  - Sem conexão: UDP (*User Datagram protocol*).
- Comunicação multiponto:
  - Sem conexão: UDP sobre *Multicast* IP.

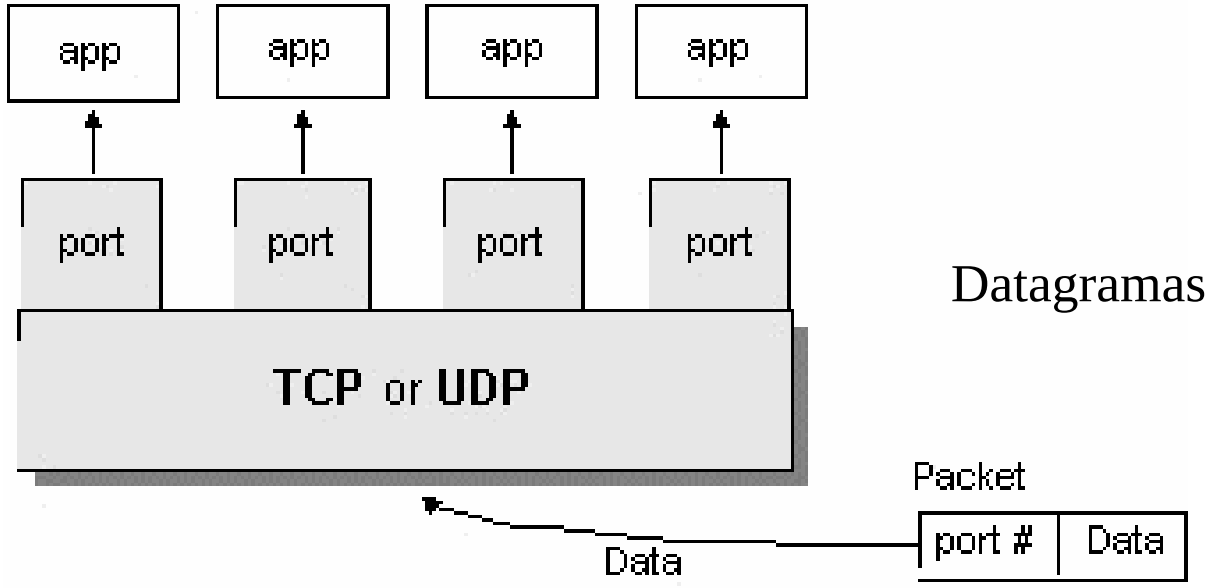
Sockets TCP vs UDP

- TCP - Orientado a conexão:
  - A conexão deve ser estabelecida antes da transmissão dos dados;
  - A conexão deve ser encerrada após a transmissão dos dados;
  - Em termos de qualidade de serviço da comunicação: **confiável** e respeita **ordem FIFO**.
- UDP - Sem conexão:
  - O endereço destino é especificado em cada datagrama.
  - Em termos de qualidade de serviço: não garante confiabilidade e nem ordenação;
  - Menos *overhead* na comunicação.

Sockets TCP vs UDP



Orientado a Conexão (TCP)



Datagramas (UDP)

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.

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.

```
ServerSocket server = new ServerSocket(2040);
```

  - Clientes utilizam a classe `Socket` para requisitar conexão a um servidor específico e então transmitir dados.

```
Socket sock = new Socket(“200.201.81.50”,2040);
```

Servidor TCP

- Tendo este *socket* sido criado, o servidor espera um pedido de conexão.

```
Socket socket = server.accept();
```

O método *accept* fica bloqueado até que um cliente requeira uma requisição para este *socket* (endereço: *host* e porta).

Servidor TCP

- Quando do recebimento da conexão, o servidor pode interagir com o cliente através da leitura e escrita de dados no *socket*.
  - *Stream* de Leitura:

    ```
    InputStream in = socket.getInputStream();
    ```

    Este *stream* tem vários métodos *read* e pode ser encapsulado por outros *streams* de entrada ou leitores (ver pacote `java.io`).
  - *Stream* de Escrita:

    ```
    OutputStream out = socket.getOutputStream();
    ```

    Este *stream* tem vários métodos *write* e pode ser encapsulado por outros *streams* de saída ou escritores (ver pacote `java.io`).

Servidor TCP

- Encerramento da conexão.
  - Com um cliente em específico:

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

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

    ```
    server.close();
    ```

- Outras observações:
  - Classe `InetAddress`: representa um endereço IP;
  - Para saber com quem esta conectado:

    `socket.getInetAddress()` e `socket.getPort()`.

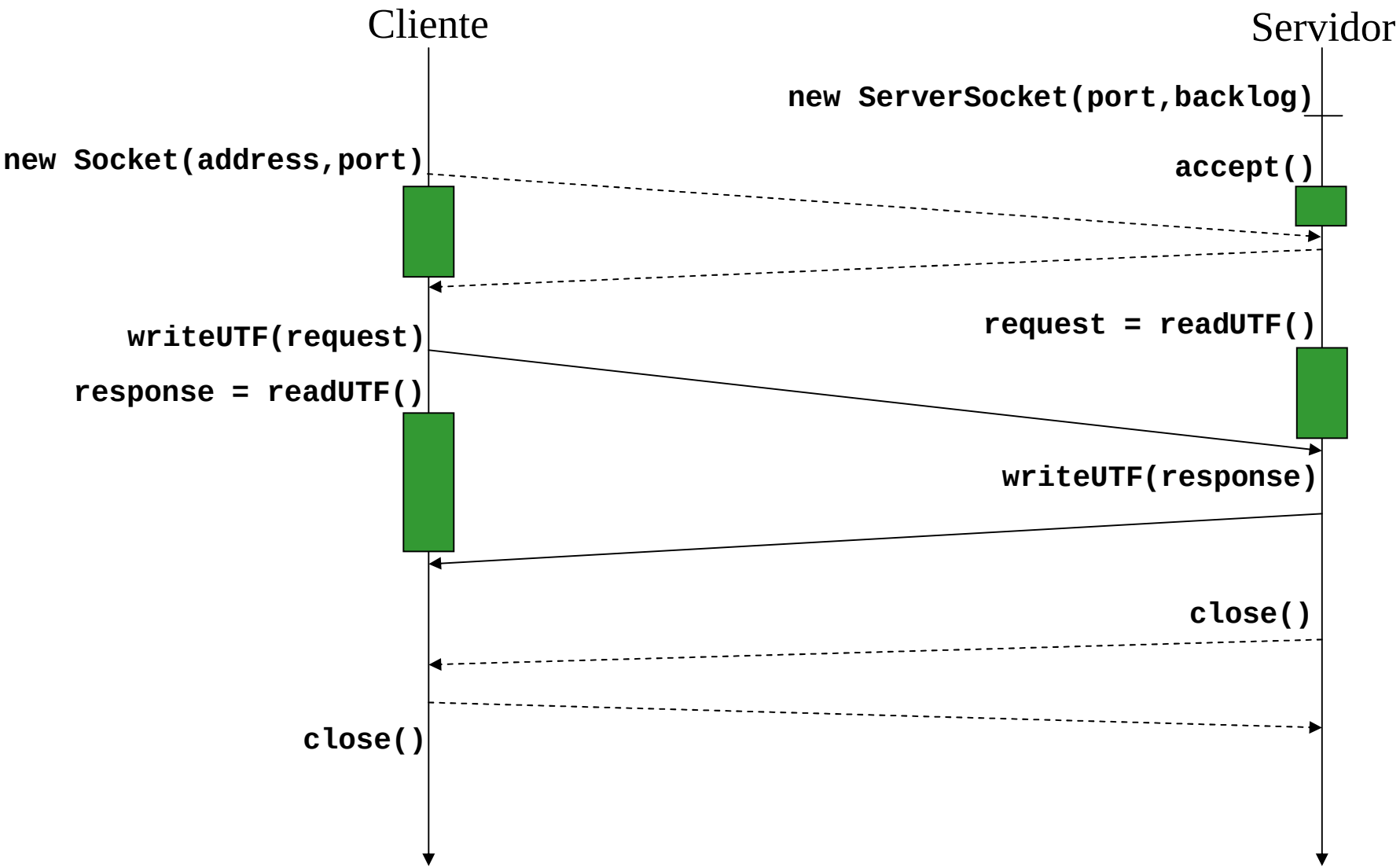
Servidor TCP

```
ServerSocket serverSocket = new ServerSocket(port,backlog);

do{
        Socket socket = serverSocket.accept();
        //obtém o stream de entrada e o encapsula
        DataInputStream dataInput =
                new DataInputStream(socket.getInputStream());
        //obtém 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();
```

Cliente TCP

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

Socket serverSocket = new Socket(address,port);

//obtém o stream de saída e o encapsula
DataOutputStream dataOutput =
       new DataOutputStream(socket.getOutputStream());
//obtém 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();
```

Resumo TCP


Cliente
Servidor
new ServerSocket(port,backlog)
new Socket(address,port)
accept()
request = readUTF()
writeUTF(request)
response = readUTF()
writeUTF(response)
close()
close()

Sockets UDP

- A comunicação UDP é feita através de duas classes:
  - *DatagramSocket*: diferentemente do TCP, a mesma classe é utilizada na representação de *sockets* UDP tanto nos clientes quanto nos servidores.
    - *Socket* cliente:

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

    - *Socket* servidor:

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

  - *DatagramPacket*: as comunicações ocorrem através da troca de datagramas.
    - Esta classe contém os dados a serem enviados/sendo recebidos bem como o endereço de destino/origem do datagrama.

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;

- Tendo o *socket*, o servidor pode esperar pelo recebimento de um datagrama (chamada bloqueante).

```
byte[] buffer = new byte[n];
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.

Servidor UDP

- O envio de datagramas é realizado também de forma bastante simples:

```
byte[] data = ...
DatagramPacket dg = new
  DatagramPacket(data,data.length,0,host,porta);
socket.send(dg);
```

data: *array* de bytes a ser enviado completamente (`data.length` é a quantidade de bytes a ser enviada com `offset = 0`).

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

- Fechamento do *socket*: `socket.close();`

Servidor UDP

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

do{
        //recebimento dos dados em um buffer de 1024 bytes
        DatagramPacket dg1 = new DatagramPacket(
                new byte[1024],1024);
        socket.receive(dg1); //recepção

        //envio de dados para o emissor do datagrama recebido
        DatagramPacket dg2 = new DatagramPacket(
                dg1.getData(),dg1.getData().length,
                dg1.getAddress(),dg1.getPort());
        socket.send(dg2); //envio
}while(notExit());

socket.close();
```

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.

Cliente UDP

```
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 dg1 = new DatagramPacket(req,req.length,0,
       address,port);
//DatagramPacket dg1 = new DatagramPacket(req,req.length,0);
socket.send(dg1); //envio

//recebimento dos dados em um buffer de 1024 bytes
DatagramPacket dg2 = new DatagramPacket(
       new byte[1024],1024);
socket.receive(dg2); //recepção
byte[] resp = dg2.getData();

socket.close();
```

Sockets Multicast

- Por ser um modelo baseado em datagramas a programação baseada em UDP/*Multicast* IP não difere muito da programação UDP já vista.
- As duas principais diferenças, em termos de modelo de programação, são:
  - Uso da classe `MulticastSocket` (ao invés de `DatagramSocket`). A classe `MulticastSocket` estende a classe `DatagramSocket` (os métodos da segunda estão disponíveis na primeira);
  - Servidores interessados em receber mensagens de um grupo devem **entrar** nesse grupo.

Revisão: *Multicast* IP

- Extensões ao protocolo IP para comunicação multiponto;
- Assim como o IP *"unicast"*, não oferece garantias a respeito da entrega ou ordenação de pacotes;
- Cada grupo é identificado por um endereço IP classe D (`224.0.0.0` a `239.255.255.255`);
- Define grupos abertos e não provê informações sobre quem são os membros (*membership*);
- O gerenciamento de grupos é feito de maneira dinâmica através de operações implementadas pelo IGMP (*Internet Group Management Protocol*).

Sockets Multicast

- Operações para gerenciamento de grupos:
  - *CreateGroup*: Cria um grupo cujo único membro é o *host* criador;

    Executa quando um primeiro servidor se junta a um grupo.
  - *JoinGroup*: Requisita a adição deste *host* a um grupo existente;

    ```
    MulticastSocket socket = new MulticastSocket(porta);
    socket.joinGroup(groupAddress);
    ```
  - *LeaveGroup*: Pede a remoção do *host* invocador ao grupo especificado.

    ```
    socket.leaveGroup(groupAddress);
    ```
- Operações para envio e recepção de grupos: `socket.send(dg)` ou `socket.receive(dg)`.

Servidor *Multicast* IP

```
InetAddress groupAddress = InetAddress.getByName("226.1.1.1");

MulticastSocket socket = new MulticastSocket(port);
socket.joinGroup(groupAddress);

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

        //imprime a mensagem recebida
        System.out.println("received "+
                new String(dg.getData).trim());
}while(notExit());

socket.leaveGroup(groupAddress);
socket.close();
```

Cliente *Multicast* IP

```
String message = ...
InetAddress groupAddress = InetAddress.getByName("226.1.1.1");

MulticastSocket socket = new MulticastSocket();

byte[] data = message.getBytes();

//recebimento dos dados em um buffer de 1024 bytes
DatagramPacket dg = new DatagramPacket(
       data,data.length,0,groupAddres,port);
socket.send(dg); //envio

socket.close();
```