

4 – IPC – Comunicação entre Processos

Prof. Marco Aurélio S Birchal
PUC Minas

Aspectos de comunicação do middleware:

- Comunicação por datagramas
- Comunicação por stream
- Representação do dado



4.1 – API para protocolos Internet

Características de IPC

- Passagem de mensagem: send e receive
 - Uso de buffer para guardar a mensagem enquanto esta é enviada
- Comunicação Síncrona ou assíncrona
- Confiabilidade
- Ordenamento

4.1 – API para protocolos Internet

a) Características de IPC

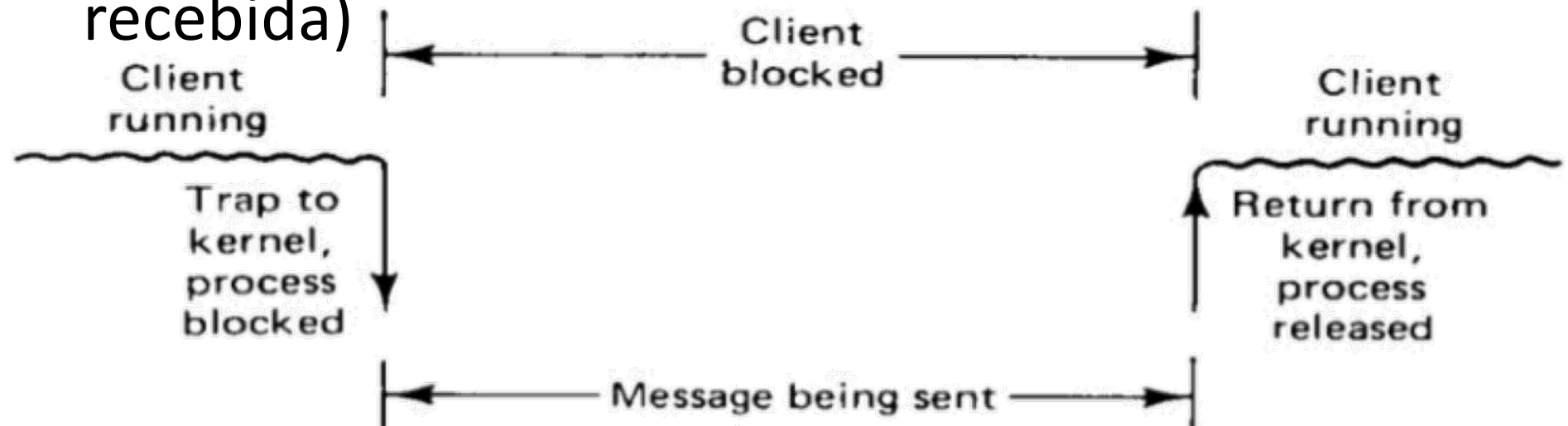
- Comunicação Síncrona: bloqueante
 - Os processos sincronizados a cada mensagem
 - Send e receive bloqueantes
- Comunicação assíncrona: não bloqueante
 - O processo origem não bloqueia (send não bloqueante)
 - O processo destino pode bloquear ou não

4.1 – API para protocolos Internet

a) Características de IPC

Comunicação Síncrona

- bloqueante
 - Processos sincronizados a cada mensagem
 - Send e receive bloqueantes (tanto o processo origem quanto o destino permanecem bloqueados até que a mensagem seja totalmente recebida)



4.1 – API para protocolos Internet

a) Características de IPC

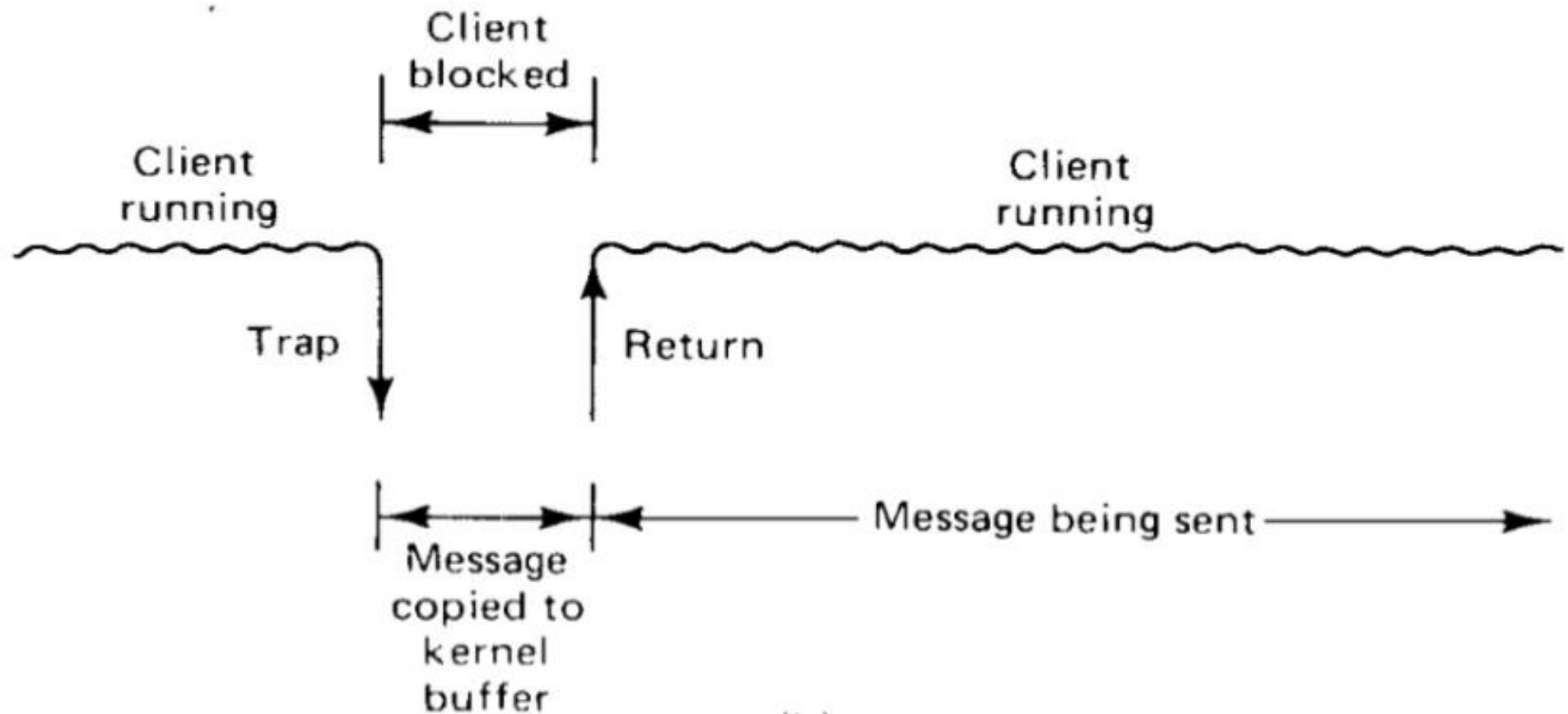
Comunicação Assíncrona

- Não bloqueante
 - Processos sincronizados a cada mensagem
 - Send não bloqueante (processo origem não bloqueia)
 - Receive bloqueante ou não (processo destino pode ter um buffer de recepção e não bloqueia)

4.1 – API para protocolos Internet

a) Características de IPC

Comunicação Assíncrona



4.1 – API para protocolos Internet

a) Características de IPC

Confiabilidade

- Dois aspectos: Validade e Integridade
- Validade: garantia de entrega (confiável: garante entrega; não confiável, não garante entrega)
- Integridade: a mensagem entregue não pode estar corrompida

4.1 – API para protocolos Internet

a) Características de IPC

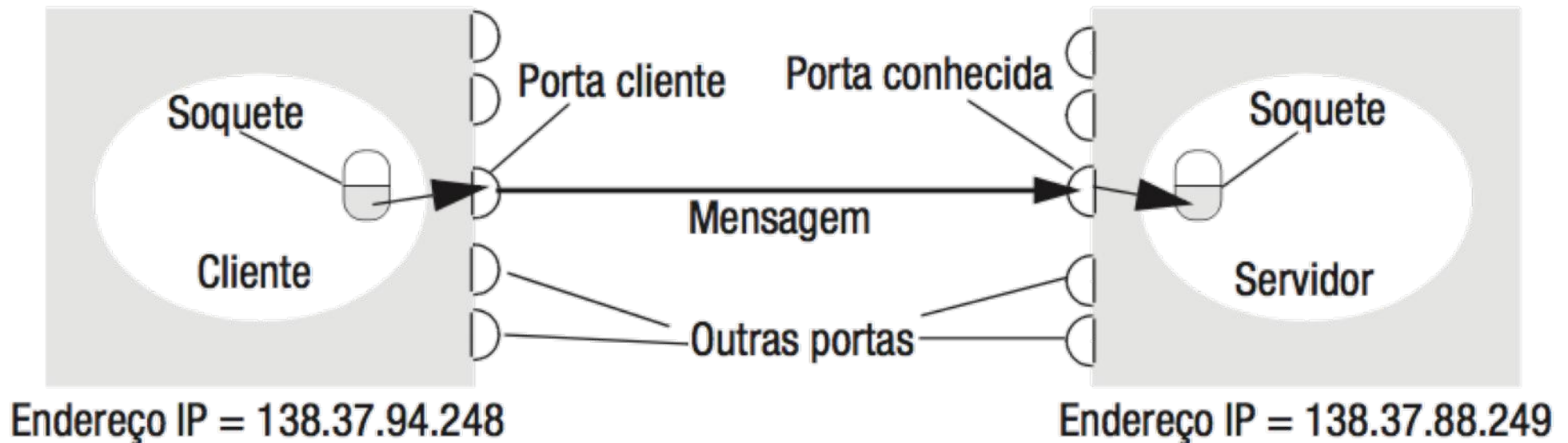
Ordenamento

- Garante a ordem de entrega: orientada à conexão
- Não garante a ordem de entrega: não orientada à conexão

4.1 – API para protocolos Internet

b) Sockets

- TCP e UDP usam a API sockets para comunicação
- Cada processo abre uma porta para comunicação
- Par IP:Porta define cada lado do canal
- Servidores: portas bem conhecidas (RFC 1700)



4.1 – API para protocolos Internet

b) Sockets

- API Java para endereços Internet:
 - Classe `InetAddress`, representa IP, para a utilização dos protocolos TCP e UDP.

`InetAddress aComputer = InetAddress.getByName(“bruno.dcs.qmul.ac.uk”)`

4.1 – API para protocolos Internet

c) Comunicação por datagrama UDP

- Envio sem confirmação ou novas tentativas.
- Se houver falha, a mensagem não chega.
- Tamanho máximo do datagrama: 64 kbytes.
- Normalmente: send não bloqueante e receive bloqueante (com buffer de datagramas no receptor).
- Pode-se configurar timeout na recepção.
- Recepção anônima: o receive recebe de qualquer origem (no entanto, pode-se configurar o IP origem)

4.1 – API para protocolos Internet

c) Comunicação por datagrama UDP

MODELO DE FALHAS

- **Falhas por omissão no canal de comunicação:**
mensagens podem ser descartadas devido a erros de soma de verificação ou porque não há espaço disponível no buffer, na origem ou no destino.
- **Ordenamento:** Às vezes, as mensagens podem ser entregues em uma ordem diferente da que foram emitidas.

4.1 – API para protocolos Internet

c) Comunicação por datagrama UDP

- duas classes: DatagramPacket e DatagramSocket.
- DatagramPacket: fornece um construtor para uma instância composta por um vetor de bytes (mensagem), o comprimento da mensagem, o endereço IP e o número da porta local do soquete de destino.

4.1 – API para protocolos Internet

c) Comunicação por datagrama UDP

- duas classes: DatagramPacket e DatagramSocket.
- DatagramSocket: oferece mecanismos para criação de sockets para envio e recepção de datagramas.

Métodos:

- Send
- Receive
- SetSoTimeout

4.1 – API para protocolos Internet

c) datagrama UDP: Cliente

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        // args give message contents and destination hostname
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request =
                new DatagramPacket(m, args[0].length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        }catch (IOException e){System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
    }
}
```



4.1 – API para protocolos Internet

c) datagrama UDP: Servidor

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            // create socket at agreed port
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(), request.getLength(),
                    request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
    }finally {if(aSocket != null) aSocket.close();}
}
```

4.1 – API para protocolos Internet

c) datagrama UDP



The image shows two overlapping terminal windows from a macOS environment. The top window, titled 'cap 4 — java UDPServer — 80x24', shows the command `java UDPServer` being executed, with a cursor on the next line. The bottom window, titled 'cap 4 — -bash — 80x21', shows the command `java UDPClient "teste abc" 127.0.0.1` being executed, followed by the output 'Reply: teste abc' and a new prompt line.

```
cap 4 — java UDPServer — 80x24
[MacBook-Air-de-Marco:cap 4 birchal$ java UDPServer
]

cap 4 — -bash — 80x21
[MacBook-Air-de-Marco:cap 4 birchal$ java UDPClient "teste abc" 127.0.0.1
Reply: teste abc
MacBook-Air-de-Marco:cap 4 birchal$
```

4.1 – API para protocolos Internet

d) Comunicação por stream TCP

- Tamanho de mensagens: arbitrário.
- Orientação à conexão: garantia de sequencia.
- Confiável: garantia de entrega e integridade.
- Presença de cliente (connect) e servidor (listen).
- Servidor pode ser multi client.
- Canal full duplex criado por dois fluxos de dados em sentidos opostos.

4.1 – API para protocolos Internet

d) Comunicação por stream TCP

MODELO DE FALHAS

- **integridade:** uso de CRC e de números de sequencia
- **Validade:** uso de timeout e retransmissão

4.1 – API para protocolos Internet

d) Comunicação por stream TCP

- duas classes: Server Socket e Socket.
- ServerSocket: usada por um servidor para criar um soquete em uma porta de serviço para receber requisições de connect dos clientes. Seu método accept recupera um pedido connect da fila ou, se a fila estiver vazia, bloqueia até que chegue um. O resultado da execução de accept é uma instância de Socket – um soquete para dar acesso aos fluxos para comunicação com o cliente.

4.1 – API para protocolos Internet

d) Comunicação por stream TCP

- duas classes: ServerSocket e Socket.
- Socket: usada pelos dois processos de uma conexão. O cliente usa um construtor para criar um soquete, especificando o nome DNS do host e a porta do servidor. Esse construtor não apenas cria um soquete associado a uma porta local, mas também o conecta com o computador remoto e com o número de porta especificado.

4.1 – API para protocolos Internet

d) Comunicação por stream TCP

- duas classes: `ServerSocket` e `Socket`.
- Socket: métodos `getInputStream` e `getOutputStream` para acessar os dois fluxos associados a um soquete.
- Os tipos de retorno desses métodos são `InputStream` e `OutputStream`, respectivamente – classes abstratas que definem métodos para ler e escrever os bytes.
- Os valores de retorno podem ser usados como argumentos de construtores para fluxos de entrada e saída.

4.1 – API para protocolos Internet

d) stream TCP: Cliente

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]);           // UTF is a string encoding see Sn. 4.4
            String data = in.readUTF();      // read a line of data from the stream
            System.out.println("Received: "+ data) ;
        }catch (UnknownHostException e){System.out.println("Socket:"+e.getMessage());}
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());}
        }catch (IOException e){System.out.println("readline:"+e.getMessage());}
        }finally {if(s!=null) try {s.close();}
            catch (IOException e){System.out.println("close:"+e.getMessage());}}
    }
}
```


4.1 – API para protocolos Internet

d)

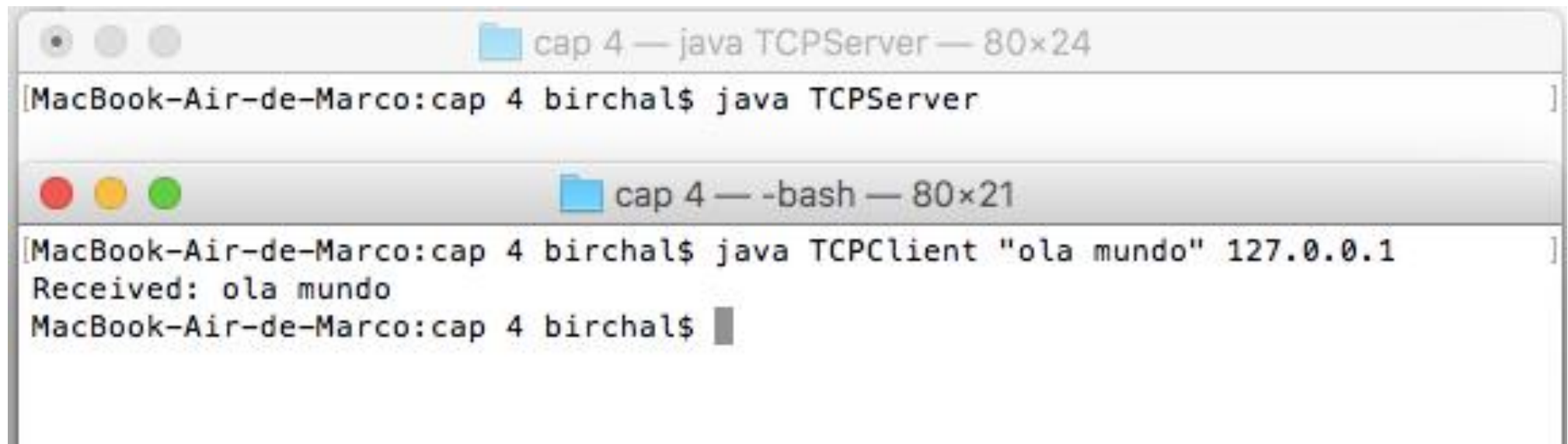
```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896; // the server port
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e) {System.out.println("Listen socket:"+e.getMessage());}
    }
}

class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream( clientSocket.getInputStream());
            out = new DataOutputStream( clientSocket.getOutputStream());
            this.start();
        } catch(IOException e) {System.out.println("Connection:"+e.getMessage());}
    }
    public void run(){
        try {
            // an echo server

            String data = in.readUTF(); // read a line of data from the stream
            out.writeUTF(data);
        } catch (EOFException e){System.out.println("EOF:"+e.getMessage());}
        } catch(IOException e) {System.out.println("readline:"+e.getMessage());}
        } finally{ try {clientSocket.close();} catch (IOException e){/*close failed*/}}
    }
}
```

4.1 – API para protocolos Internet

d) stream TCP



```
cap 4 — java TCPServer — 80x24
[MacBook-Air-de-Marco:cap 4 birchal$ java TCPServer

cap 4 — -bash — 80x21
[MacBook-Air-de-Marco:cap 4 birchal$ java TCPClient "ola mundo" 127.0.0.1
Received: ola mundo
MacBook-Air-de-Marco:cap 4 birchal$
```

4.2 – Representação externa de dados e empacotamento

- A. Aspectos gerais
- B. CORBA CDR (*Common Data Representation*)
- C. Serialização de objetos Java
- D. XML (Extensible Markup Language)
- E. Referência a objetos remotos

4.2 – Representação externa de dados e empacotamento – Aspectos Gerais

- Informações armazenadas em programas são representadas como uma estruturas de dados
- Na transmissão, os dados são representados numa sequência de bytes
- Necessidade de conversão: estrutura de dados \leftrightarrow sequência de bytes
- Problema
 - Diferentes sistemas possuem diferentes estruturas de dados
 - Ex: Código ASCII x UNICODE, *big endian* x *little endian*, formatos de ponto flutuante, etc.
- Solução
 - Conversão para um formato de dados de comum acordo (*eXternal Data Representation* – XDR)
 - Enviar no formato do emissor e incluir informação sobre o formato empregado

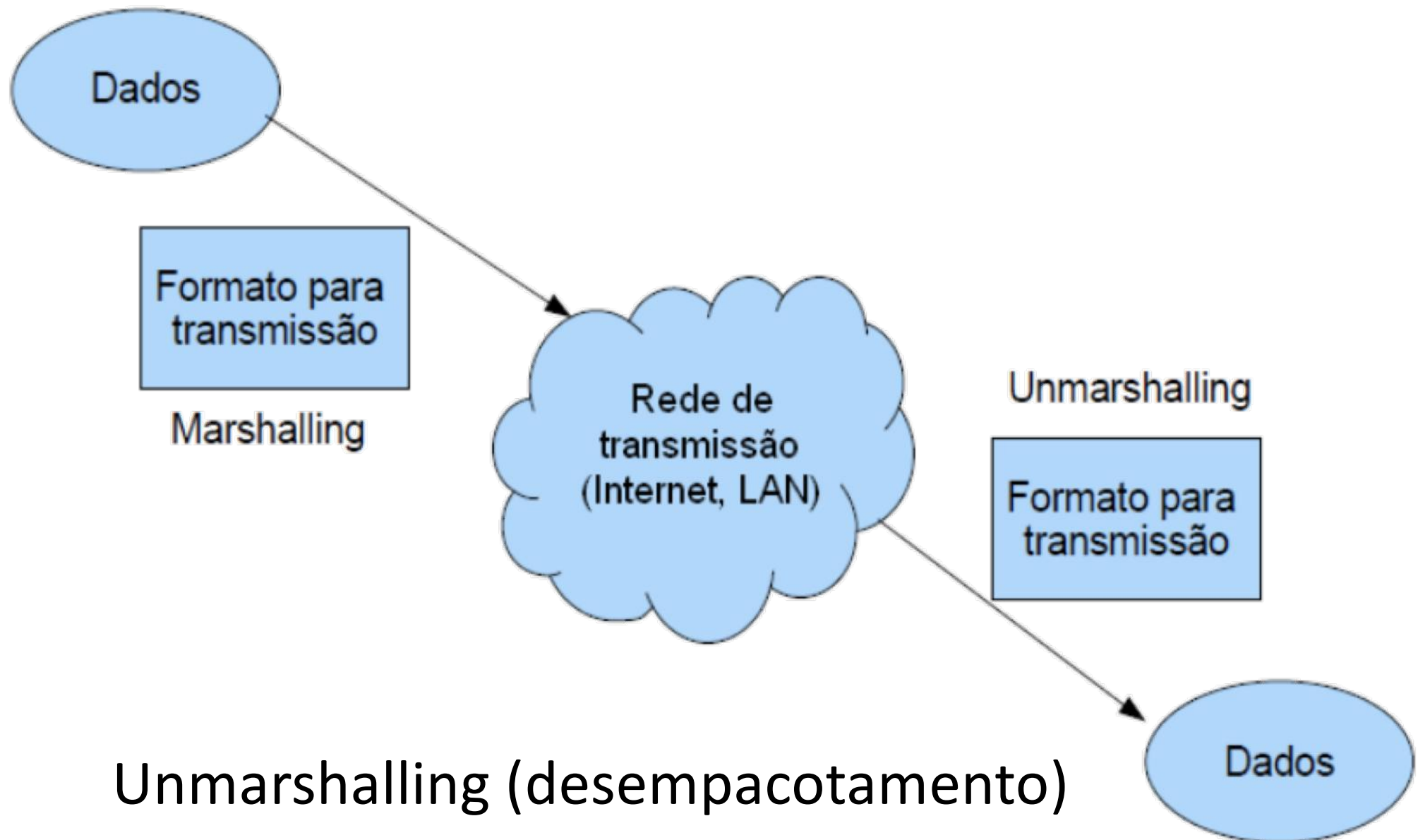
4.2 – Representação externa de dados e empacotamento – Aspectos Gerais

- Formato independente de linguagem, SO, etc
- Utilizada para comunicação dos dados de requisições e respostas entre clientes e servidores
- Formato serializado
- *Marshalling* (empacotamento): conversão entre os formatos primitivos e externos, para transmissão
 - CORBA Common Data Representation*
 - Serialização de objetos em Java*
 - XML (*Extensible Markup Language*)**

* Feitos pelo próprio middleware

** Feito por programação

4.2 – Representação externa de dados e empacotamento – Aspectos Gerais



4.2 – Representação externa de dados e empacotamento – Aspectos Gerais

Marshalling (Empacotamento)

- Processo de converter uma coleção de dados e organizá-los em um formato próprio para transmissão
 - Unmarshalling é o processo oposto
- Ideal não haver o envolvimento explícito da aplicação (transparência)
 - Responsabilidade do middleware
- Duas técnicas básicas
 - Conversão dos dados para um formato binário
 - Conversão dos dados para um formato texto (ASCII)
 - Serialização Java e XML

4.2 – Representação externa de dados

B. CORBA - Representação comum de dados (CDR)

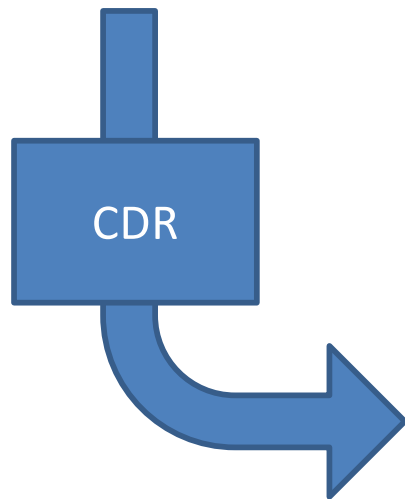
O CDR pode representar todos os tipos de dados que são usados como argumentos e valores de retorno em invocações a métodos remotos no CORBA.

- Representação para tipos primitivos:
 - inteiros, ponto flutuante, octeto
 - *big-endian* e *little-endian*
 - posicionamento de cada item de dados na msg.
- Representação para tipos compostos (ou construídos):
 - sequência, string, array, struct, enumeração, união
 - construída a partir das sequências de bytes correspondentes aos tipos primitivos constituintes
- Informação de tipo: Os tipos das estruturas de dados e os tipos dos itens de dados básicos estão descritos no IDL (*Interface Definition Language*) do CORBA

4.2 – Representação externa de dados

B. CORBA - Representação comum de dados (CDR)

```
struct Person{  
    string name;  
    string place;  
    unsigned long year;  
};
```



*Índice na
sequência de bytes*

← 4 bytes →

*Observações sobre
a representação*

0–3	5	<i>Comprimento do string</i>
4–7	"Smit"	<i>'Smith'</i>
8–11	"h____"	
12–15	6	<i>Comprimento do string</i>
16–19	"Lond"	<i>'London'</i>
20–23	"on____"	
24–27	1984	<i>unsigned long</i>

Mensagem no CDR do CORBA.

4.2 – Representação externa de dados

B. CORBA - Representação comum de dados (CDR)

- No caso de CORBA são incluídos apenas os valores
 - Considera-se que o cliente já saiba os tipos dos dados e a ordem de chegada destes dados
- Entretanto, XML e Java incluem tanto os tipos quanto os próprios dados

4.2 – Representação externa de dados

C. Serialização de Objetos JAVA

- Em Java, o termo *serialização* se refere à atividade de simplificar um objeto, ou um conjunto de objetos conectados, em uma forma sequencial conveniente para ser armazenada em disco ou transmitida em uma mensagem.
- A *desserialização* consiste em restaurar o estado de um objeto ou conjunto de objetos a partir de sua forma serializada.
- Pressupõe-se que o processo que faz a desserialização não tenha nenhum conhecimento anterior dos tipos dos objetos na forma serializada.

4.2 – Representação externa de dados

C. Serialização de Objetos JAVA

- Transformação de um objeto Java para uma sequência de bytes
- Informações de tipo e classe dos objetos são incluídas na forma serializada
 - Permitem a reconstrução dos objetos sem conhecimento prévio de suas classes
- Processo recursivo: serializa todos os objetos referenciados pelo objeto em questão
- Classes precisam implementar a *interface Serializable*

4.2 – Representação externa de dados

C. Serialização de Objetos JAVA

- Forma de disponibilização ao usuário
 - Classes ***ObjectOutputSream*** e ***ObjectInputStream*** e seus respectivos métodos ***writeObject*** e ***readObject***
 - Classe do objeto implementa a *interface Serializable*
- Conteúdo do formato serializado
 - Nome da classe
 - Versão da classe (quando são feitas alterações na classe)
 - Referências a outros objetos (*handles*)
 - Valores das instâncias das variáveis

4.2 – Representação externa de dados

C. Serialização de Objetos JAVA

- Objetos (instância de uma classe Java) e valores de dados primitivos podem ser passados como argumentos e resultados de invocações de método

Ex.: *struct Person*

```
public class Person implements Serializable {  
    private String name;  
    private String place;  
    private int year;  
    public Person(String aName, String aPlace, int aYear) {  
        name = aName;  
        place = aPlace;  
        year = aYear;  
    } // seguido dos métodos para acessar as variáveis de instância  
}
```

4.2 – Representação externa de dados

Serialização de Objetos JAVA

Serializando a instância de objeto *Person*:

Person p = new Person(“Smith”, “London”, 1984);

Valores serializados				Explicação
Person	Número da versão de 8 bytes		h0	Nome da classe, número da versão
3	int year	java.lang.String name	java.lang.String place	Número, tipo e nome das variáveis de instância
1984	5 Smith	6 London	h1	Valores das variáveis de instância

Na realidade, a forma serializada inclui marcas adicionais de tipos; h0 e h1 são identificadores.

4.2 – Representação externa de dados

C. Serialização de Objetos JAVA

```
import java.io.*;

public class Serializadora {
    public static void main(String[] args) {
        Serializacao s = new Serializacao();
        s.setNome("qualquerNome");
        try {
            ObjectOutputStream o = new ObjectOutputStream(new FileOutputStream("./objeto.teste"));
            o.writeObject(s);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```


4.2 – Representação externa de dados

C. Serialização de Objetos JAVA

```
import java.io.Serializable;

public class Serializacao implements Serializable {
    private static final long serialVersionUID = -475160693380890471L;
    private String nome;
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

4.2 – Representação externa de dados

C. Serialização de Objetos JAVA

- Serialização e desserialização geralmente são operações feitas automaticamente pelo *middleware*
- Java oferece meios para o programador personalizar a serialização
- Exemplo é o uso da palavra-chave ***transient*** que serve para informar que uma variável **não** deve ser serializada
 - Por exemplo, variáveis que referenciam recursos locais como arquivos ou sockets

4.2 – Representação externa de dados

D. XML (Extensible Markup Language)

- Linguagem de marcação assim como HTML
- Definir documentos estruturados para a Web
- Os itens de dados são rotulados com *strings* de marcação
- Diferente do HTML, permite que os usuários definam suas próprias tags (Extensível)

4.2 – Representação externa de dados

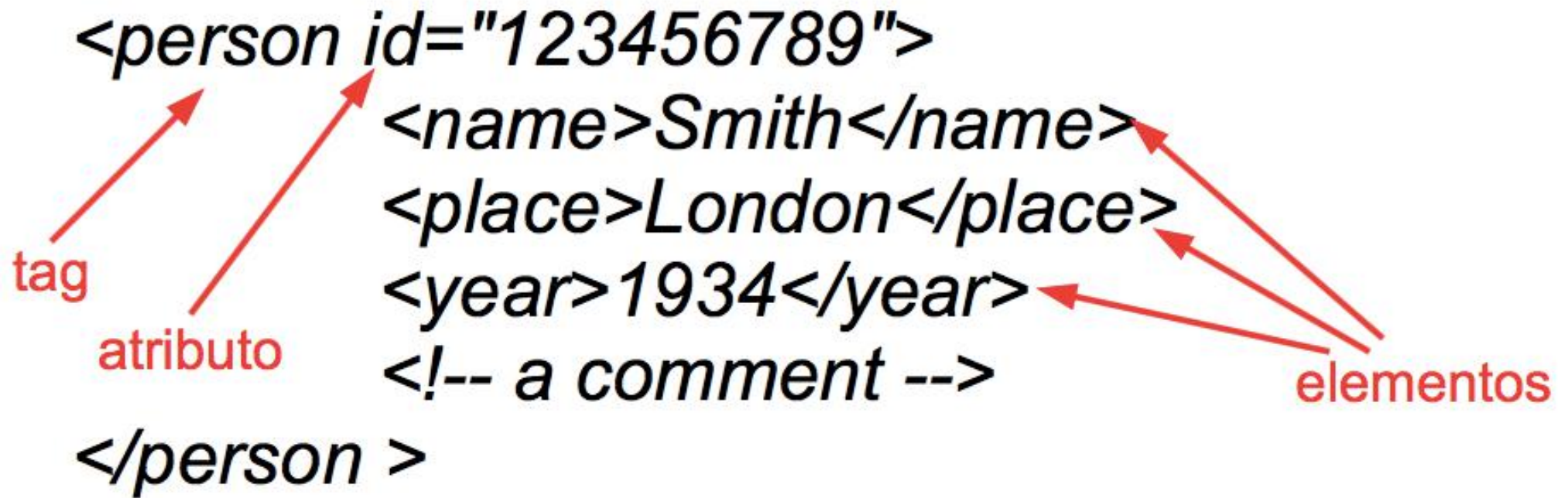
D. XML (Extensible Markup Language)

- Usos de interesse aqui:
 - definir a interface de serviços Web
 - prover a representação externa de dados na comunicação entre clientes e serviços
- Representação textual: independente de plataforma Representação hierárquica
- Textual e Auto-descritiva: *tags*, esquemas e *namespaces*
- Desempenho inferior comparado ao CORBA CDR

4.2 – Representação externa de dados

D. XML (Extensible Markup Language)

Definição da Estrutura do XML



- **Atributos**: rotular dados
- **Elementos**: *container* para dados

4.2 – Representação externa de dados

D. XML (Extensible Markup Language) – NAMESPACES XML

- Permitem definir contextos para os marcadores (*tags*) utilizados em um documento
- Referenciados através de URLs
- Evitam choques de nomes de *tags* em contextos diferentes
- As tags *name*, *place* e *year* podem ser usadas em diferentes documentos

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">  
  <pers:name> Smith </pers:name>  
  <pers:place> London </pers:place >  
  <pers:year> 1934 </pers:year>  
</person>
```

4.2 – Representação externa de dados

D. XML (Extensible Markup Language) – ESQUEMAS XML: XSD

- Definem:
 - os elementos e atributos que podem aparecer em documentos XML (i.e., um vocabulário)
 - como os elementos são aninhados
 - a ordem, o número e o tipo dos elementos
- Usados para validar documentos XML
- Um mesmo esquema pode ser compartilhado por vários documentos

4.2 – Representação externa de dados

D. XML (Extensible Markup Language) – ESQUEMAS XML: XSD

```
<xsd:schema xmlns:xsd = URL of XML schema definitions >  
  <xsd:element name= "person" type = "personType" />  
    <xsd:complexType name="personType">  
      <xsd:sequence>  
        <xsd:element name = "name" type="xs:string"/>  
        <xsd:element name = "place" type="xs:string"/>  
        <xsd:element name = "year" type="xs:positiveInteger"/>  
      </xsd:sequence>  
      <xsd:attribute name= "id" type = "xs:positiveInteger"/>  
    </xsd:complexType>  
</xsd:schema>
```


4.2 – Representação externa de dados

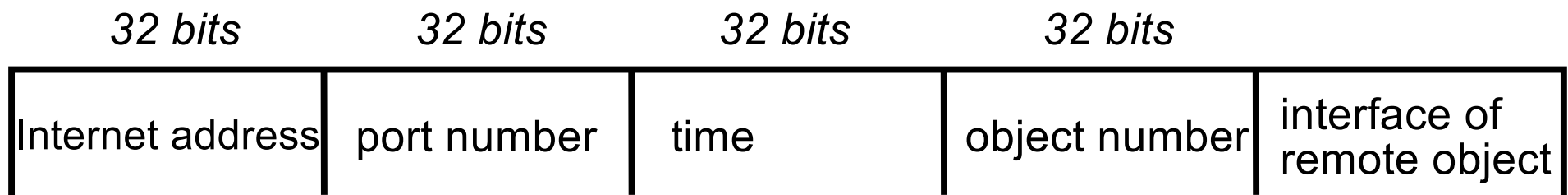
E. Referências a Objetos Remotos

- Num sistema distribuído, um processo servidor pode conter vários objetos remotos (de vários clientes)
- Necessário uma maneira de identificar um objeto remoto **de forma única** dentro de um sistema distribuído
- Necessária quando objetos remotos são passados como parâmetro

4.2 – Representação externa de dados

E. Referências a Objetos Remotos

- A referência é serializada (não o objeto)
- Devem conter toda informação necessária para identificar (e endereçar) um objeto **unicamente** no sistema distribuído. Ex.: hora da criação, nº da porta.
- Devem ser descartadas quando o objeto remoto deixar de existir



4.3 – Comunicação por Multicast

Aspectos gerais

Mensagens multicast:

- Tolerância a falhas baseada em serviços replicados
- Localização de servidores de descoberta em redes espontâneas
- Melhoria de desempenho através de dados replicados
- Propagação de notificação de eventos

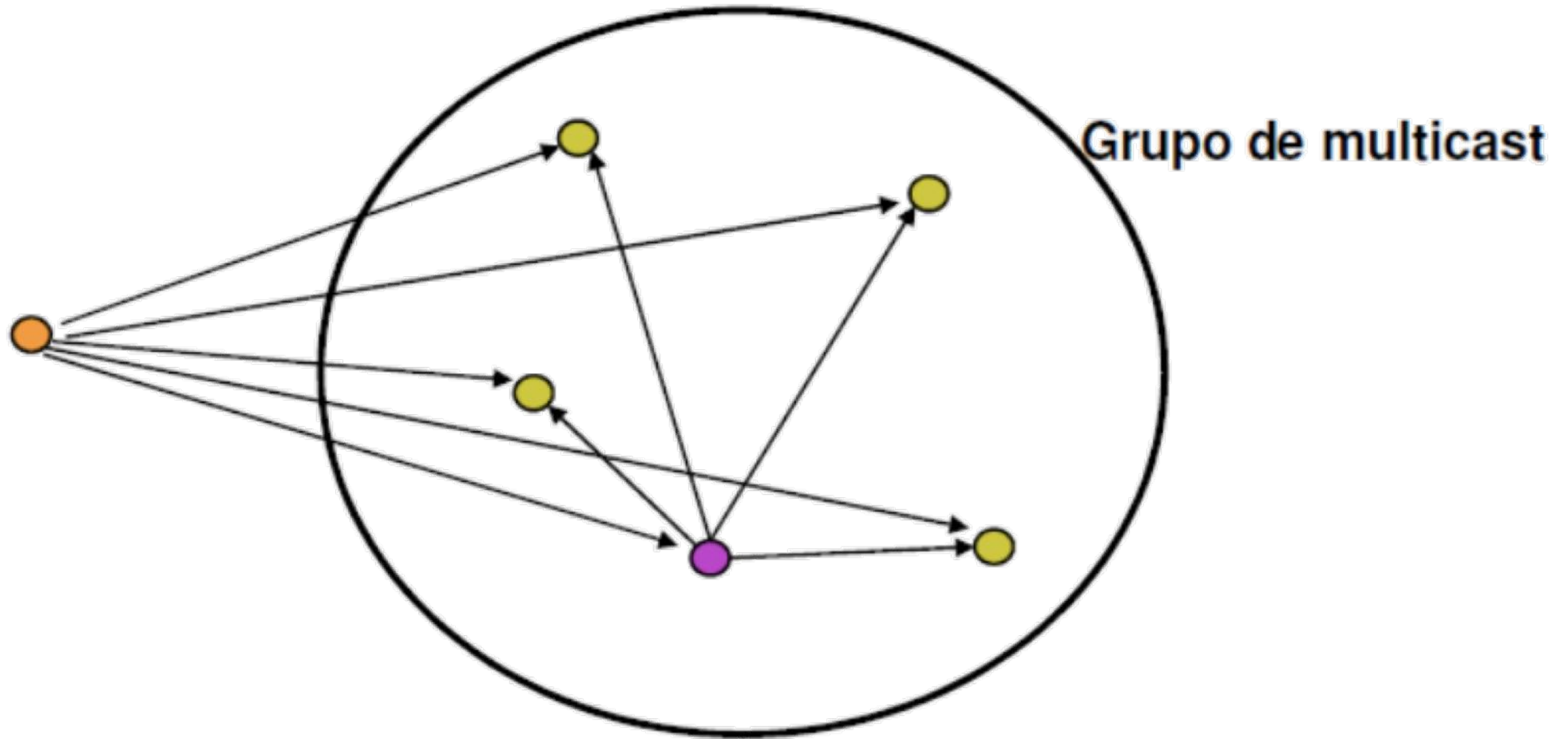
4.3 – Comunicação por Multicast

Aspectos gerais

- Processo envia a mensagem para o grupo (não para seus membros diretamente)
- Entrega de uma mesma mensagem, enviada por um processo, para cada um dos processos que são membros de um determinado grupo
- O conjunto de membros do grupo é transparente para o processo que envia a mensagem
- Mensagens comunicadas através de operações de *multicast*

4.3 – Comunicação por Multicast

Aspectos gerais



4.3 – Comunicação por Multicast

Multicast IP – difusão seletiva

- É o protocolo básico para comunicação de grupo
- Assim como o IP (*unicast*): não-confiável
 - mensagens podem ser perdidas (falha de omissão)
 - i.e., não entregues para alguns membros do grupo
 - mensagens podem ser entregues fora de ordem
- Acessível às aplicações através de UDP
- Grupos são identificados por: endereço IP + porta, utiliza endereços IP que iniciam por 1110 (IPv4)
- Processos se tornam membros de grupos, mas não conhecem os demais membros

4.3 – Comunicação por Multicast

Multicast IP – difusão seletiva

- Um computador é membro de um grupo se ele possui um ou mais processos com sockets que se juntaram ao grupo: *multicast sockets*
- Camada de rede:
 - recebe mensagens endereçadas a um grupo, se computador for membro
 - entrega as mensagens para cada um dos *sockets* locais que participa do grupo
 - processos membros são identificados pelo número de porta associado ao grupo
 - vários processos compartilham o mesmo número de porta

4.3 – Comunicação por Multicast

Multicast IP – difusão seletiva

- *Alocação de endereços multicast*: endereços da Classe D (intervalo 224.0.0.0 a 239.255.255.255)
 - Bloco de controle de rede local (224.0.0.0 a 224.0.0.225), para tráfego *mul- ticast* dentro de determinada rede local.
 - Bloco de controle de Internet (224.0.1.0 a 224.0.1.225).
 - Bloco de controle *ad hoc* (224.0.2.0 a 224.0.255.0) para tráfego que não se encaixa em nenhum outro bloco.
 - Bloco de escopo administrativo (239.0.0.0 a 239.255.255.255), que é usado para implementar mecanismo de escopo para tráfego *multicast* (para restrin- gir a propagação).

4.3 – Comunicação por Multicast

Multicast IP – difusão seletiva - JAVA

- Classe MulticastSocket
 - Derivada de DatagramSocket
 - Principais métodos:
 - joinGroup: método para entrar em um grupo
 - leaveGroup: método para sair de um grupo
 - setTimeToLive: serve para configurar o TTL
- TTL (*Time to live*): serve para limitar a distância de propagação de um datagrama multicast pela rede
 - Pequenas distâncias: redes locais
 - Grandes distâncias: redes externas, Internet (uso de roteadores externos)

4.3 – Comunicação por Multicast

Multicast IP – difusão seletiva - JAVA

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
        // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s = null;
        try {
            InetAddress group = InetAddress.getByName(args[1]);
            s = new MulticastSocket(6789);
            s.joinGroup(group);
            byte [] m = args[0].getBytes();
            DatagramPacket messageOut =
                new DatagramPacket(m, m.length, group, 6789);
            s.send(messageOut);

            // this figure continued on the next slide
        }
    }
}
```

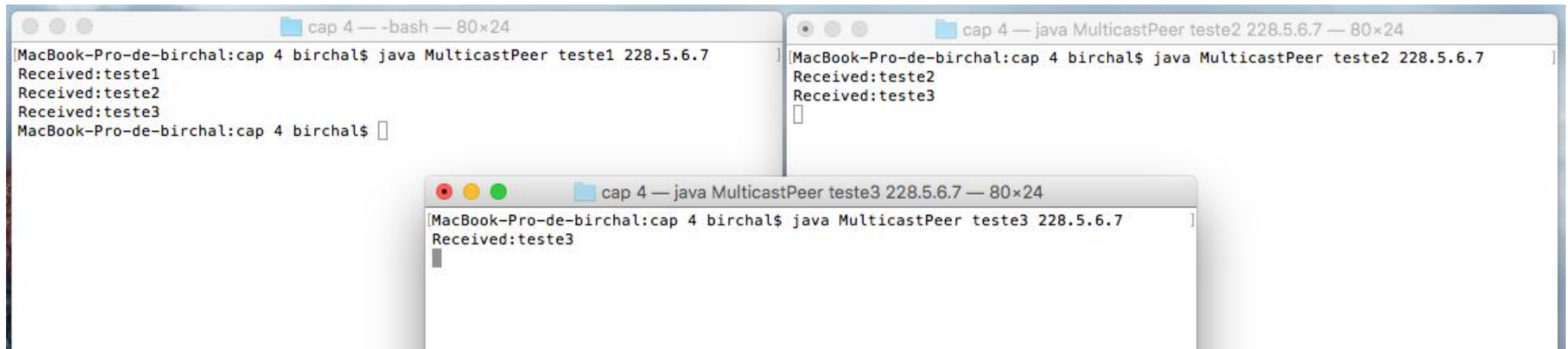
4.3 – Comunicação por Multicast

Multicast IP – difusão seletiva - JAVA

```
// get messages from others in group
    byte[] buffer = new byte[1000];
    for(int i=0; i< 3; i++) {
        DatagramPacket messageIn =
            new DatagramPacket(buffer, buffer.length);
        s.receive(messageIn);
        System.out.println("Received:" + new String(messageIn.getData()));
    }
    s.leaveGroup(group);
} catch (SocketException e){System.out.println("Socket: " + e.getMessage());
} catch (IOException e){System.out.println("IO: " + e.getMessage());}
}finally {if(s != null) s.close();}
}
}
```

4.3 – Comunicação por Multicast

Multicast IP – difusão seletiva - JAVA



The image displays three overlapping terminal windows on a macOS system, each running a Java application named `MulticastPeer`. The windows are titled `cap 4 — -bash — 80x24`, `cap 4 — java MulticastPeer teste2 228.5.6.7 — 80x24`, and `cap 4 — java MulticastPeer teste3 228.5.6.7 — 80x24`. The first window shows the execution of `java MulticastPeer teste1 228.5.6.7`, which outputs `Received:teste1`, `Received:teste2`, and `Received:teste3`. The second window shows the execution of `java MulticastPeer teste2 228.5.6.7`, which outputs `Received:teste2` and `Received:teste3`. The third window shows the execution of `java MulticastPeer teste3 228.5.6.7`, which outputs `Received:teste3`. All windows show a prompt `MacBook-Pro-de-birchal:cap 4 birchal$` before the command and a cursor after the output.

```
MacBook-Pro-de-birchal:cap 4 birchal$ java MulticastPeer teste1 228.5.6.7
Received:teste1
Received:teste2
Received:teste3
MacBook-Pro-de-birchal:cap 4 birchal$

MacBook-Pro-de-birchal:cap 4 birchal$ java MulticastPeer teste2 228.5.6.7
Received:teste2
Received:teste3

MacBook-Pro-de-birchal:cap 4 birchal$ java MulticastPeer teste3 228.5.6.7
Received:teste3
```

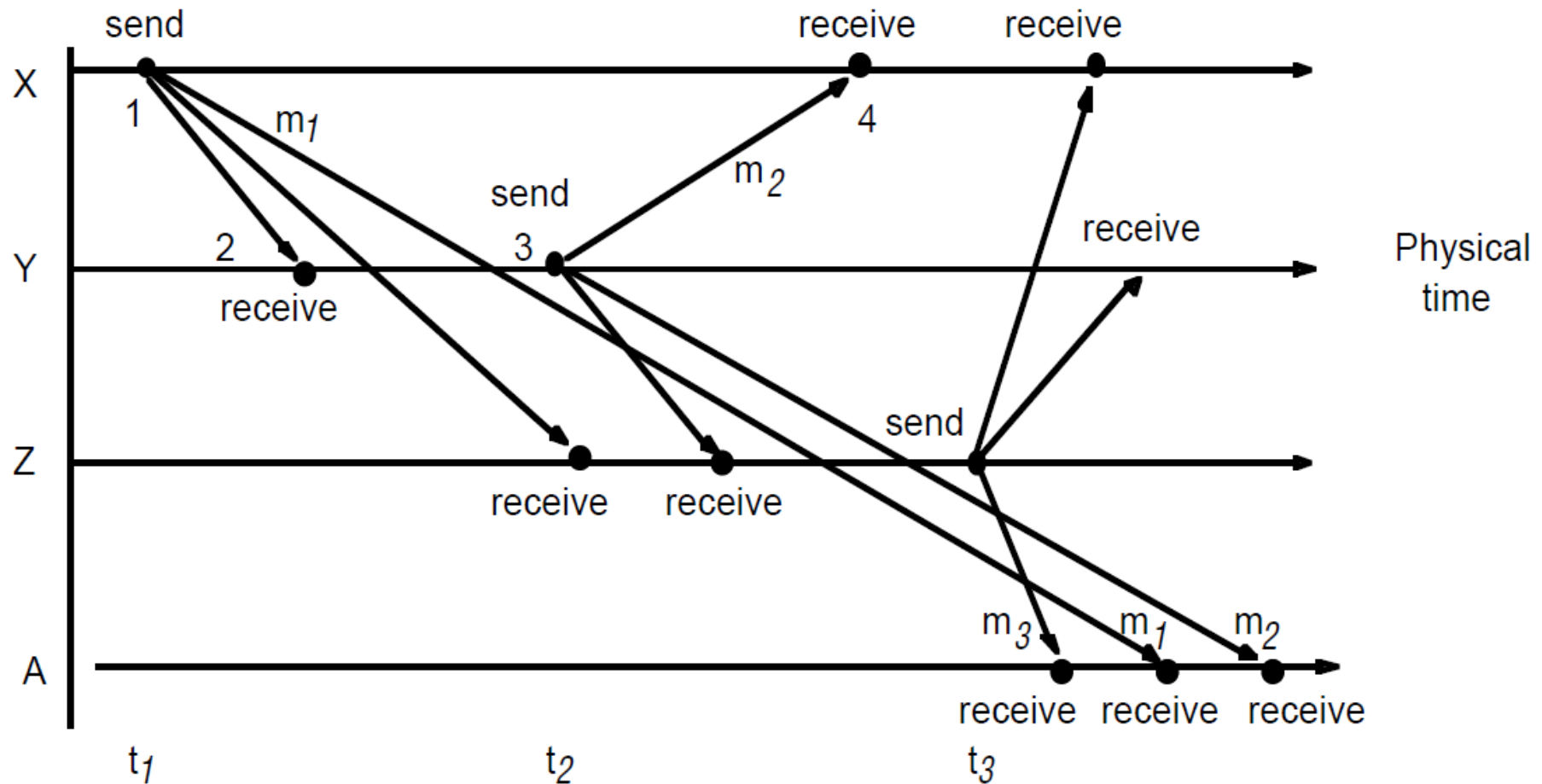
4.3 – Comunicação por Multicast

Multicast IP – difusão seletiva - Falhas

- Fontes de falhas
 - membros de um grupo podem perder mensagens devido a congestionamento (fila de chegada cheia)
 - falhas em roteadores de *multicast*
 - roteador não propaga a mensagem para os membros que estão após ele na rede
- Falhas de ordenação
 - Mensagens enviadas por um processo podem ser recebidas por outros processos em ordens diferentes
 - Mensagens enviadas por diferentes processos podem não chegar na mesma ordem em todos os demais processos

4.3 – Comunicação por Multicast

Multicast IP – difusão seletiva – Falha de ordenação



4.3 – Comunicação por Multicast

Multicast IP – difusão seletiva - conclusão

- Protocolo de *multicast* não-confiável
- Uso sobre redes locais: utiliza *multicast* físico
 - ex.: em redes Ethernet
- Uso na Internet: utiliza roteadores de *multicast*
 - configurados através de algum protocolo de roteamento multicast *time-to-live* para limitar a propagação de mensagens

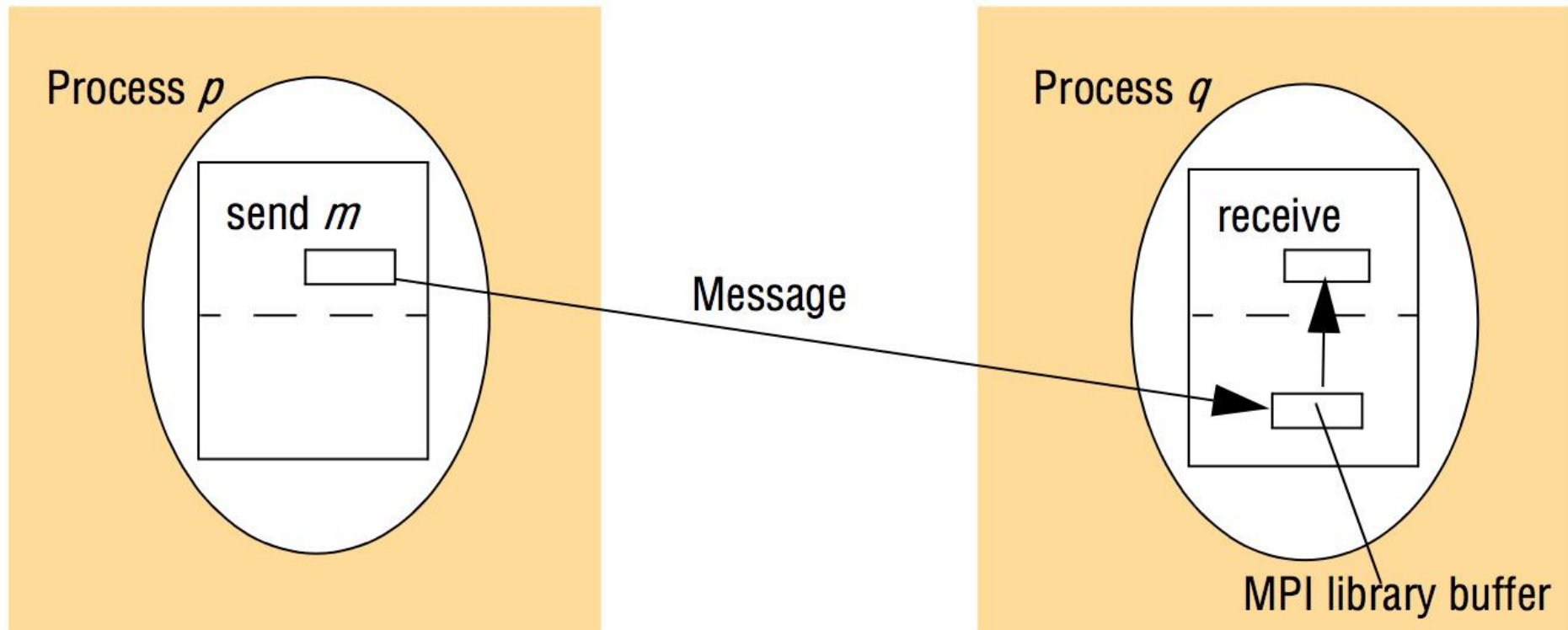
4.4 – Estudo de Caso - MPI

Message Passing Interface

- Paradigma de programação distribuída, leve, eficiente e mínima.
- Uso: computação de alto desempenho (onde o desempenho é fundamental).
- Computação em *grid*.
- Independência de SO e de linguagem.
- Buffers no envio e no recebimento permitem ampla implementação de comunicação síncrona e assíncrona, com bloqueio ou sem.

4.4 – Estudo de Caso – MPI

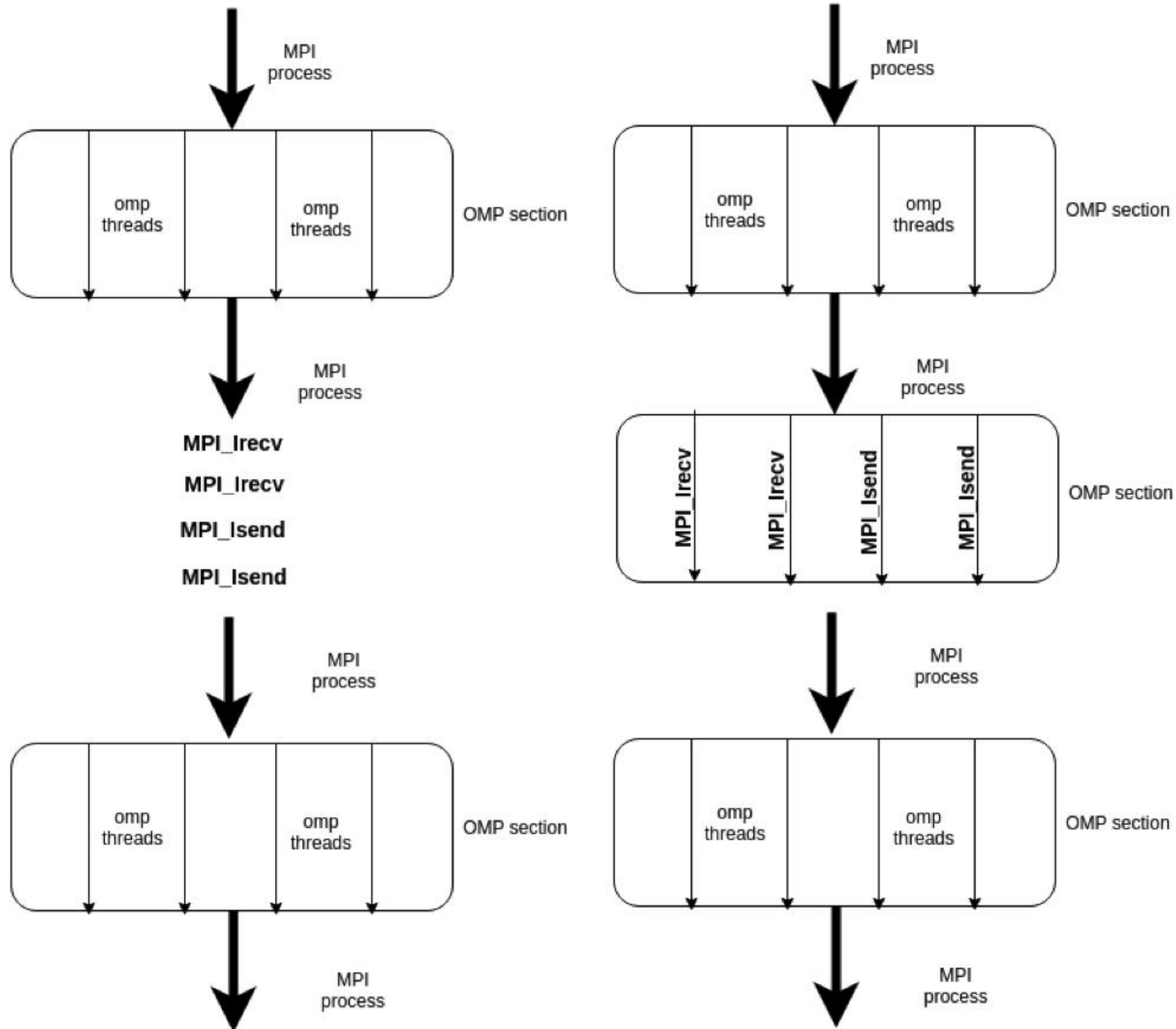
Visão geral da comunicação ponto a ponto MPI



Modelo send/receive acrescido da capacidade de buffer tanto no remetente quanto no destinatário

<i>Operações send</i>	<i>Com bloqueio</i>	<i>Sem bloqueio</i>
<i>Genéricas</i>	<i>MPI_Send</i> : o remetente bloqueia até que seja seguro retornar; isto é, até que a mensagem esteja em trânsito ou tenha sido entregue e que, portanto, o <i>buffer</i> de aplicativo do remetente possa ser reutilizado.	<i>MPI_Isend</i> : a chamada retorna imediatamente e o programador recebe um identificador de pedido de comunicação, o qual, então, pode ser usado para verificar o andamento da chamada via <i>MPI_Wait</i> ou <i>MPI_Test</i> .
<i>Síncronas</i>	<i>MPI_Ssend</i> : o remetente e o destinatário são sincronizados e a chamada só retornará quando a mensagem tiver sido entregue no endereço do destinatário.	<i>MPI_Issend</i> : semelhante a <i>MPI_Isend</i> , mas com <i>MPI_Wait</i> e <i>MPI_Test</i> indicando se a mensagem foi entregue no endereço do destinatário.
<i>Com buffer</i>	<i>MPI_Bsend</i> : o remetente aloca explicitamente um <i>buffer da</i> biblioteca MPI (usando uma chamada de <i>MPI_Buffer_attach</i> separada), e a chamada retorna quando os dados são copiados com sucesso nesse <i>buffer</i> .	<i>MPI_Ibsend</i> : semelhante a <i>MPI_Isend</i> , mas com <i>MPI_Wait</i> e <i>MPI_Test</i> indicando se a mensagem foi copiada no <i>buffer</i> MPI do remetente e, portanto, está em trânsito.
<i>Prontas</i>	<i>MPI_Rsend</i> : a chamada retorna quando o <i>buffer</i> de aplicativo do remetente pode ser reutilizado (como com <i>MPI_Send</i>), mas o programador também está indicando para a biblioteca que o destinatário está pronto para receber a mensagem, resultando em uma possível otimização da implementação subjacente.	<i>MPI_Irsend</i> : o efeito é igual ao de <i>MPI_Isend</i> , mas com <i>MPI_Rsend</i> o programador está indicando para a implementação subjacente que o destinatário está realmente pronto para receber (resultando nas mesmas otimizações).

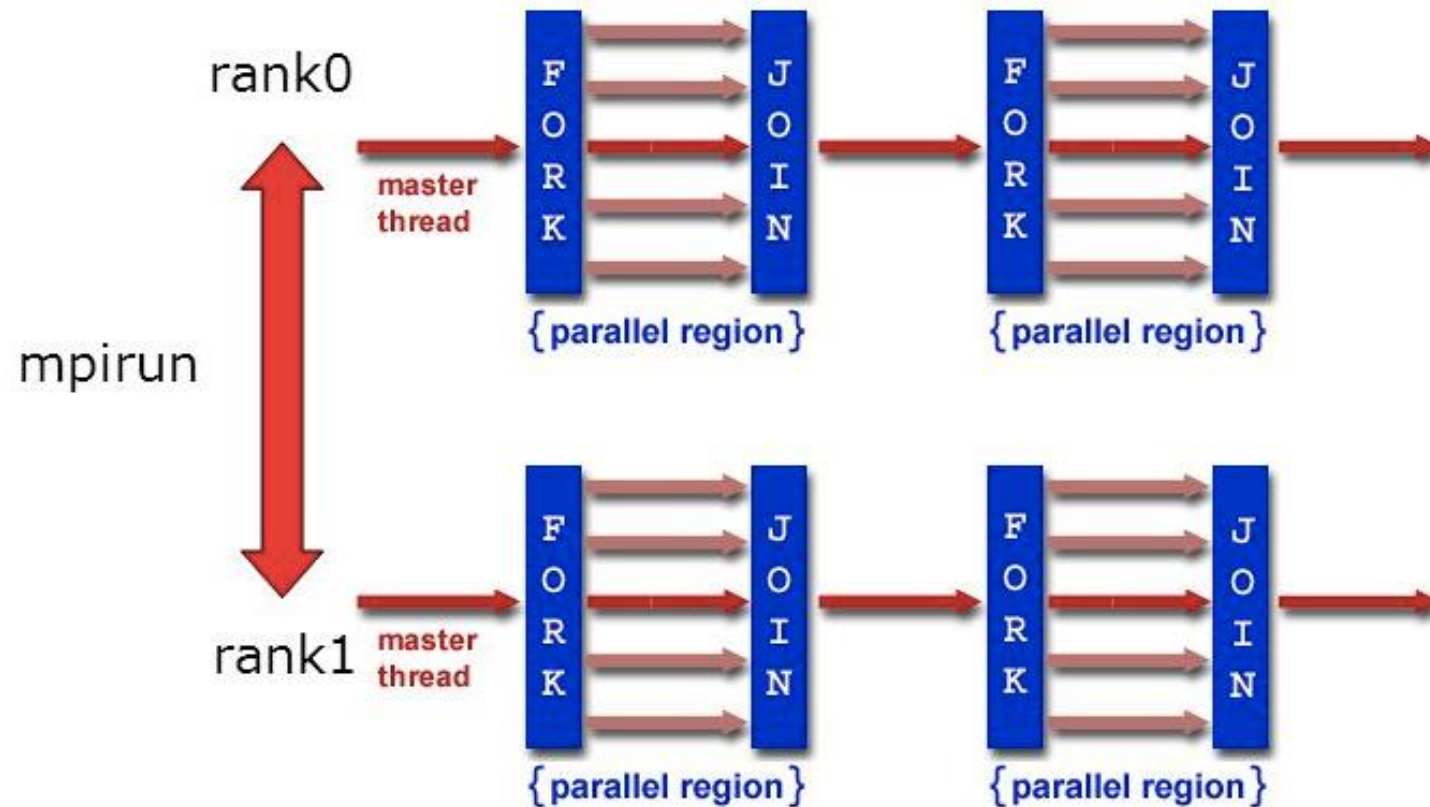
4.4 – Estudo de Caso - MPI



4.4 – Estudo de Caso - MPI

Hybrid MPI + OpenMP programming

- Each MPI process spawns multiple OpenMP threads



4.4 – Estudo de Caso - MPI

