



UNIVERSIDADE FEDERAL DO CEARÁ - Campus Quixadá

Cursos: SI, ES, RC, CC e EC

Código: QXD0043

Disciplina: Sistemas Distribuídos

Capítulo 4 – Comunicação entre processos

Prof. Rafael Braga

Agenda

- ⌘ ***API para comunicação entre processos (seção 4.2)***
 - ⌘ ***Representação externa de dados (seção 4.3)***
 - ⌘ ***Comunicação em grupo (seção 4.4)***
-

API para Comunicação

⌘ *Interface de programação para utilizar os serviços básicos oferecidos pelo subsistema de comunicação subjacente*

⌘ *Tópicos abordados:*

- ▤ Características da comunicação entre processos
 - ▤ Resolução de nomes
 - ▤ Comunicação usando ***sockets***
-

Características de Comunicação

Aplicações e serviços

RMI e RPC

Comunicação cliente-servidor e grupal

Representação externa de dados

UDP e TCP

Características de Comunicação

⌘ ***Duas operações básicas: send e receive***

- ▤ Definidas em termos de **destinos e mensagens**
 - ▤ Implementadas através de “filas” (***buffers***) de mensagens em cada lado da comunicação
 - ▤ Podem envolver a **sincronização** entre o processo que envia mensagem e o processo que a recebe
-

Características de Comunicação

⌘ *Comunicação síncrona*

- Envio e recebimento sincronizados para cada mensagem (operações “**bloqueantes**”)
 - send* - bloqueia o processo remetente até que o *receive* correspondente tenha sido executado no lado servidor
 - receive* - bloqueia o processo de destino até a chegada de uma mensagem
-

Características de Comunicação

⌘ *Comunicação assíncrona*

- Envio e recebimento sem sincronização (operações “*não-bloqueantes*”)

- send*

- libera o processo remetente assim que a mensagem tiver sido copiada para a fila local;

- execução e transmissão ocorrem em paralelo

- receive*

- libera o processo destino antes da chegada da mensagem, a qual deve ser retirada da fila posteriormente, através de um mecanismo de *pooling* ou de notificação (*callback*)

Características de Comunicação

⌘ ***Comunicação síncrona é a mais usada na prática!***

⚡ Operações **não-bloqueantes** ainda podem exigir alguma forma de sincronização no nível da aplicação

⌘ ***Operações bloqueantes podem ser utilizadas sem interromper totalmente o processo que as invocou quando executadas em fluxos (threads) separados de execução***

Características de Comunicação

⌘ ***Destino = (endereço de rede + porta)***

- ⌘ **Endereço de rede** corresponde ao IP do computador onde é executado o processo que receberá a mensagem
- ⌘ **Porta** corresponde ao destino de uma mensagem dentro de um mesmo computador (especificado como um inteiro entre 0 e 65535: **2^{16} portas**)
 - ⌘ A cada porta é associado um **único processo destino (exceto** no caso de portas usadas para difusão seletiva de mensagens)
 - ⌘ Um mesmo processo pode ter **múltiplas portas associadas a ele**
 - ⌘ Qualquer processo que conhece o número da porta de outro processo pode enviar mensagens para ele através dessa porta
 - ⌘ Servidores costumam divulgar suas portas para os seus clientes ou usam portas padrão (ex.: HTTP = 80, FTP = 21, etc.)

⌘ ***Não há transparência de localização!***

Características de Comunicação

⌘ ***Destinos transparentes quanto a sua localização podem ser obtidos da seguinte forma:***

- /// Utilizando **nomes simbólicos** para referenciar endereços de rede e um serviço de nomes para traduzir nomes em endereços em tempo de execução
 - /// Forma mais comum de implementar transparência de localização na Internet!
 - /// Suporta **re-alocação dos serviços e distribuição de carga**
-

Características de Comunicação

⌘ **Confiabilidade**

- ▤ Definida em termos da **validade** e da **integridade** do *serviço de comunicação*
 - ▤ **Validade:** garantia de que as mensagens serão entregues mesmo diante de um número “razoável” de **falhas** de transmissão (perdas de pacotes)
 - ▤ **Integridade:** *garantia de que as mensagens serão entregues sem serem* **corrompidas** ou **duplicadas**
-

Características de Comunicação

⌘ *Ordenação*

- /// Serviço de comunicação deve garantir que as mensagens serão entregues na **ordem** em que são **enviadas** pelo processo remetente
 - /// Entrega fora da ordem de envio pode ser considerada **falha de transmissão por** aplicações que dependem da ordem das mensagens
 - /// (Exemplos?)
-

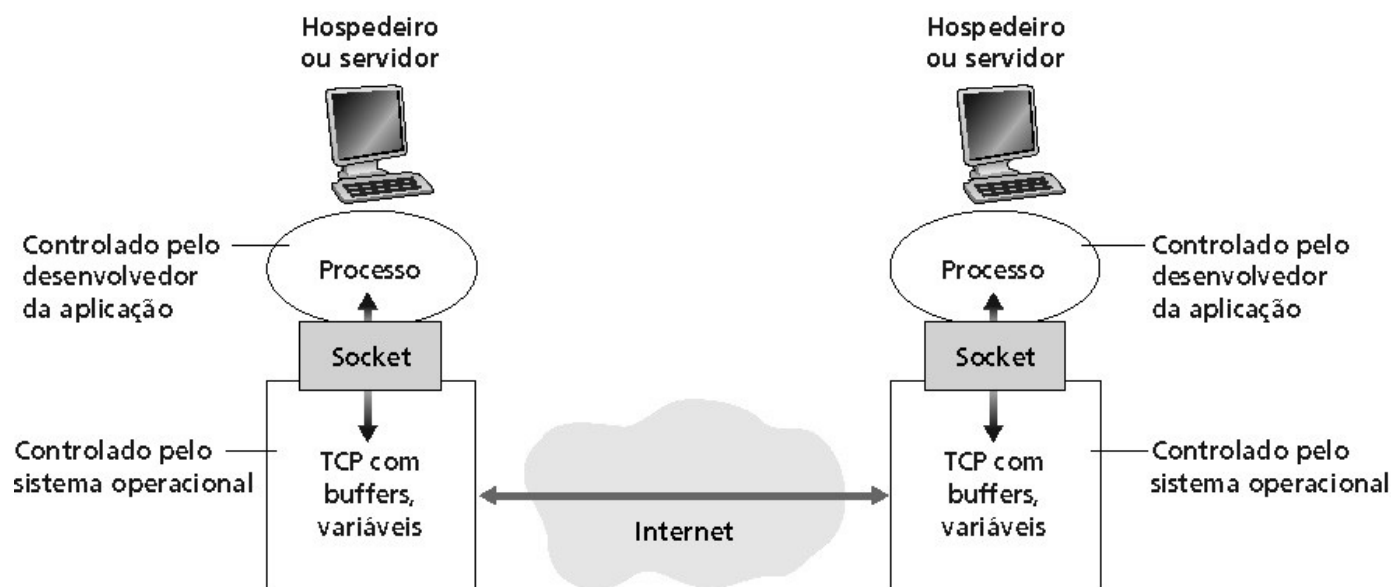
Características de Comunicação

- ***API de Java para resolução de nomes de domínio***
- ***Implementada através da classe `InetAddress`***
 - Métodos encapsulam **localização e consulta** a servidor DNS
 - **Independente** do formato da representação interna (IPv4, IPv6, ...)

```
import java.net.*;
...
try {
    InetAddress ip =
        InetAddress.getByName("www.ufc.br");
} catch (UnknownHostException e){
    System.out.println("Endereço desconhecido!");
}
```

Características de Comunicação

- **Socket:** uma porta entre o processo de aplicação e o protocolo de transporte fim-a-fim (UCP ou TCP)
- **Serviço TCP:** transferência confiável de **bytes** de um processo para outro



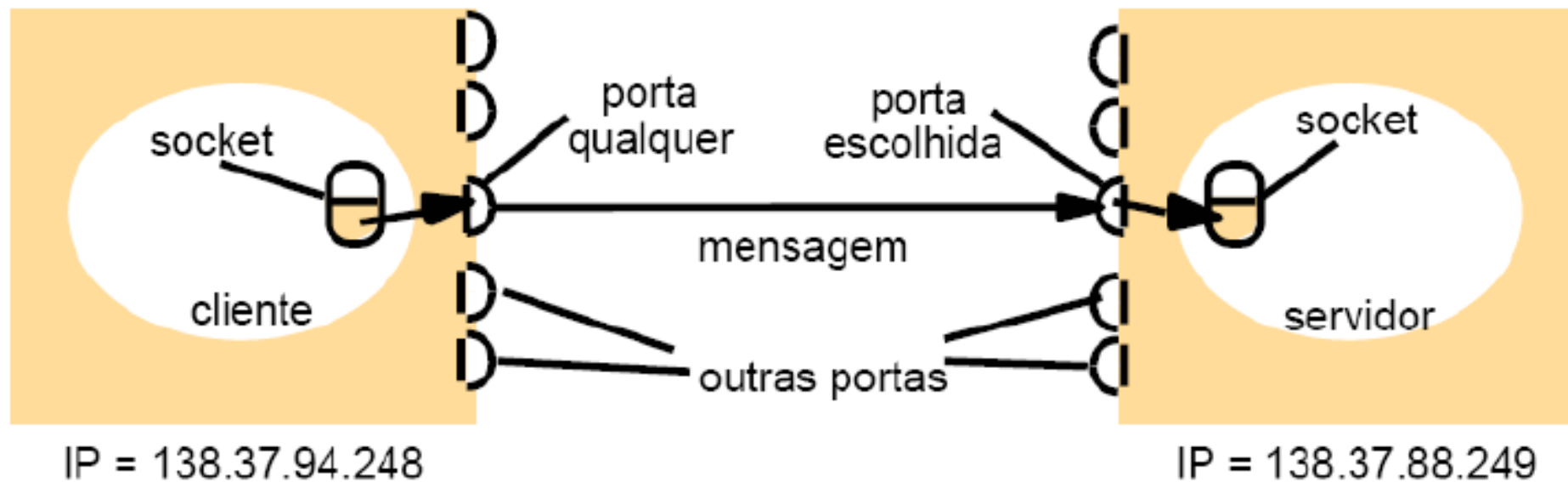
Comunicação usando *sockets* (I)

- ⌘ ***Socket: abstração para cada um dos extremos (end-points) da comunicação entre dois processos***
 - ▤ Originário do *BSD UNIX* e posteriormente incorporado em virtualmente **todos os outros sistemas operacionais**
 - ▤ Usado tanto para comunicação **assíncrona (protocolo UDP)** quanto para comunicação **síncrona (protocolo TCP)**
-

Comunicação usando *sockets* (II)

- ⌘ ***Comunicação exige a criação de um socket pelo processo remetente e de um outro pelo processo destinatário***
 - ▤ ***socket*** do destinatário (**servidor**) deve estar exclusivamente acoplado a uma porta do computador local
 - ▤ Um mesmo ***socket*** pode ser usado tanto para **enviar** quanto para **receber** mensagens (comunicação bidirecional)
 - ▤ Um processo pode usar **múltiplas portas, mas não pode** compartilhar portas com **outros processos do mesmo** computador (exceto no caso de *sockets* associados a endereços de difusão seletiva)
-

Comunicação usando *sockets* (III)



Comunicação usando *sockets* - *UDP* (I)

⌘ *Transmissão não confiável de “pacotes” (datagrams) entre um processo cliente e um processo servidor*

- ⚡ Não garante **entrega** nem **ordem** de recebimento das mensagens
- ⚡ Exige que clientes e servidores criem um **socket** *acoplado ao* IP e a uma porta da máquina local
 - ⚡ Servidor escolhe uma **porta específica** (conhecida pelos clientes)
 - ⚡ Cliente utiliza uma **porta qualquer** dentre as portas disponíveis

⌘ *Comunicação sem conexão prévia*

- ⚡ No cliente: endereço e porta do servidor passados como parâmetros de entrada para cada operação **send**
 - ⚡ No servidor: endereço e porta do cliente recebidos como parâmetros de saída de cada operação **receive**
-

Comunicação usando *sockets* - UDP (II)

⌘ *Operações bloqueantes e não-bloqueantes*

- ⚡ *send* retorna logo após repassar a mensagem para os protocolos subjacentes (UDP/IP)
 - ⚡ Mensagem é descartada na chegada se não há um **socket** acoplado ao endereço e à porta de destino
- ⚡ *receive* bloqueia processo se não houver mensagens na fila
 - ⚡ Variação **não-bloqueante** disponível em algumas implementações
 - ⚡ Bloqueio pode ser controlado através de temporizadores (**timeouts**)
 - ⚡ Pode receber mensagens de diferentes endereços de origem

⌘ *Modelo de falha*

- ⚡ **Checksums** para detectar e rejeitar mensagens corrompidas
 - ⚡ Garantia de **entrega e ordem** de chegada a cargo da aplicação
-

Comunicação usando *sockets* - *UDP* (III)

⌘ *Uso de UDP*

- /// Em algumas aplicações pode ser aceitável utilizar um serviço sujeito a falhas de omissão ocasionais (ex.: **DNS, VoIP**)
 - /// Uso de UDP **evita custos** normalmente associados com a garantia de entrega das mensagens:
 - /// Necessidade de armazenar informações de estado na **origem e no destino**
 - /// Transmissão de **mensagens extras** (além dos dados da aplicação)
 - /// **Latência** no envio
-

Exemplos de uso em Java e C/Unix

⌘ **Em Java:**

- API implementada em algumas **classes do pacote *java.net***

 - DatagramPacket*

 - DatagramSocket*

 - Socket*

 - ServerSocket*

⌘ **Em C/Unix:**

- API padrão (biblioteca de funções) para manipulação de *sockets*

- Definições básicas no arquivo de cabeçalho *socket.h*

API de Java com *sockets sobre UDP*

⌘ *Classes principais:*

▤ *DatagramPacket*

- ▤ Usada pelo cliente para construir um ***datagram*** a partir de um array de bytes e do **endereço+porta** de um servidor
- ▤ Usada pelo servidor para armazenar o ***datagram*** recebido juntamente com o **endereço+porta** do socket cliente

▤ *DatagramSocket*

- ▤ Usada para **criar sockets** para envio e recebimento de datagramas
 - ▤ Construtores **com ou sem a opção de especificar a porta desejada** (se o programador não especificar a porta, o sistema escolhe a primeira porta disponível)
 - ▤ Lança exceção ***SocketException*** se a porta especificada já estiver em uso
-

Exemplo de uso no lado do cliente

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){

        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();}
    }
}
```

Exemplo de uso no lado do servidor

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket (6789);
            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();}
    }
}
```

Comunicação usando *sockets* - *TCP (I)*

⌘ *Transmissão confiável de mensagens entre um processo cliente e um processo servidor*

- /// Cliente solicita a conexão de um **socket** *acoplado a* uma porta local qualquer para o endereço e a porta do *socket no servidor*
 - /// Servidor aceita conexões entre seu **socket local** e os *sockets dos clientes*
 - /// Conexão feita através de um par de “fluxos” (*streams*)
 - /// Um **fluxo de saída** no cliente é conectado a um fluxo de **entrada no servidor**, e vice-versa
 - /// Permite a comunicação **bi-lateral** e concorrente entre clientes e servidores
-

Comunicação usando *sockets* - TCP (II)

⌘ *Noção de fluxos de dados abstrai das seguintes características da rede:*

- ⚡ A aplicação pode escolher a **quantidade de dados** que vai ler ou escrever em um fluxo; cabe à implementação do TCP o quanto desses dados serão transmitidos em cada pacote IP
 - ⚡ Um esquema de **confirmações e temporizadores** permite a detecção e retransmissão de mensagens extraviadas
 - ⚡ Um esquema de **controle de fluxo** permite sincronizar a velocidade de envio na origem com a velocidade de recebimento no destino
 - ⚡ Se o **remetente** estiver enviando dados muito rapidamente, ele é **bloqueado** até que o destinatário tenha recebido uma quantidade de **dados suficiente**
 - ⚡ Identificadores de mensagens são associados a cada pacote IP, o que permite ao destinatário **detectar e rejeitar mensagens duplicadas, ou reordenar** eventuais mensagens recebidas fora de ordem
-

Comunicação usando *sockets* - *TCP* (III)

⌘ *Protocolos de interação*

- ⚡ **Clientes e servidores** devem concordar sobre uma seqüência para a troca de mensagens e sobre como interpretar seu conteúdo
- ⚡ Falta de entendimento pode causar **erros de interpretação** (ex.: receber um inteiro esperando uma string) e **bloqueios indevidos** (ex.: tentativa de receber mais dados do que os que foram de fato enviados)

⌘ *Operações bloqueantes*

- ⚡ ***send*** - bloqueia o processo até que haja espaço disponível na fila correspondente ao *stream de entrada* do *socket de destino*
 - ⚡ ***receive*** - bloqueia o processo até que haja uma mensagem disponível na fila correspondente ao *stream* de entrada do *socket local*
-

Comunicação usando *sockets* - *TCP* (IV)

⌘ *Modelo de falha*

- ⚡ **Checksums** para detectar e rejeitar mensagens corrompidas
- ⚡ **Números seqüenciais** para detectar e rejeitar mensagens duplicadas
- ⚡ **Temporizadores e retransmissões** para lidar com perdas de pacotes

⌘ *Uso de TCP*

- ⚡ Muitas aplicações da Internet rodam sobre conexões TCP, em portas reservadas, incluindo:
 - ⚡ HTTP (porta 80)
 - ⚡ FTP (porta 20/21)
 - ⚡ Telnet (porta 23)
 - ⚡ SMTP (porta 25)
-

API de Java com *sockets sobre TCP*

⌘ **Socket**

- ⚡ Usada pelo cliente para **criar um *socket* (acoplado a uma porta local qualquer)** para ser conectado ao ***socket de um servidor remoto***
- ⚡ Lança exceção ***UnkownHostException se houver*** problemas com o endereço do servidor, ou ***IOException, se*** houver erros de E/S
- ⚡ Métodos ***getInputStream e getOutputStream*** permitem acesso ao ***stream de entrada e ao stream de saída***, respectivamente, associados ao **socket**

⌘ **ServerSocket**

- ⚡ Usada pelo servidor para **criar um *socket* (acoplado a uma determinada porta local)** para aceitar **conexões dos clientes**
 - ⚡ Método ***accept*** devolve um objeto da classe ***Socket***, já conectado ao socket de um cliente que tenha solicitado conexão, ou **bloqueia** o servidor até que um novo pedido de conexão seja recebido
-

Exemplo de uso no lado do cliente

```
package sockets.TCP;

import java.net.*;

public class TCPClient {
    public static void main(String args[]) {
        // arguments supply message and hostname
        Socket s = null;
        try {
            int serverPort = 7896;
            s = new Socket("localhost", serverPort);

            DataOutputStream out = new DataOutputStream(s.getOutputStream());
            out.writeUTF("turma_sd_2021.1"); // UTF is a string encoding see Sn. 4.4

            DataInputStream in = new DataInputStream(s.getInputStream());
            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());
                }
        }
    }
}
```

Exemplo no lado do servidor

```
package sockets.TCP;

import java.net.*;

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

// this figure continues on the next slide

Exemplo no lado do servidor (cont.)

```
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.toUpperCase());
        } 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 */
            }
        }
    }
}
```


Agenda

- ⌘ ***API para comunicação entre processos (seção 4.2)***
 - ⌘ ***Representação externa de dados (seção 4.3)***
 - ⌘ ***Comunicação em grupo (seção 4.4)***
-

Representação externa de dados

⌘ *Motivação:*

- Programas e rede **manipulam dados em diferentes níveis de abstração (abstrações de dados X seqüência de bytes)**
 - Abstrações de dados devem ser “**achatadas**” (ou “**serializadas**”) pelo remetente, antes da transmissão, e então **reconstruídas** pelo destinatário, após o seu recebimento
- Nem todos os programas utilizam a **mesma forma de representação** para os mesmos tipos de dados
 - Diferentes **conjuntos de tipos primitivos**
 - Diferentes **tamanhos** (em bytes) para os mesmos tipos
 - Diferentes **ordem de bits** para tipos do mesmo tamanho
 - Diferentes **arquiteturas para números** de ponto flutuante
 - Diferentes **padrões de caracteres** (UTF-8, ASCII, Unicode, ...)

⌘ *Como fazer para que programas em diferentes computadores troquem dados de forma consistente?*

Representação externa de dados

⌘ *Duas abordagens:*

- 1. Remetente envia os dados no seu **formato original (local)**, junto com uma **indicação do formato utilizado**, e destinatário os converte novamente para o seu formato local, se necessário
 - /// A favor: evita **conversões desnecessárias**
 - /// Contra: destinatário precisa conhecer **todos os outros formatos**
- 2. Remetente converte os dados para um **formato externo** (acordado previamente) antes da transmissão, e destinatário os converte novamente do formato externo para o seu formato local
 - /// A favor: remetente e destinatário só precisam **conhecer um único formato externo**
 - /// Contra: **conversões desnecessárias quando os computadores** envolvidos utilizam o mesmo formato (como otimizar?)

⌘ *Exemplos práticos: (CORBA, RMI, SOAP(XML), ProtocolBuffers)*

Empacotamento e Desempacotamento (I)

⌘ ***Empacotamento (marshalling):***

- ⌘ Agrupamento de um conjunto de itens de dados num formato adequado (representação externa) para transmissão via mensagens

⌘ ***Desempacotamento (unmarshalling):***

- ⌘ Re-agrupamento dos dados recebidos no formato externo para produzir um conjunto equivalente de itens de dados no formato local do destinatário

⌘ ***Realizados automaticamente pelas camadas do middleware, sem intervenção do usuário ou programador***

- ⌘ Necessitam da **especificação detalhada dos tipos de dados** a serem transmitidos em uma linguagem apropriada (IDL, WSDL, etc)
 - ⌘ Evitam **erros de programação**
 - ⌘ Podem ser usados para **outros fins (ex.: persistência)**
-

Empacotamento/Desempacotamento (II)

⌘ *Algumas abordagens:*

▤ CDR (CORBA)

- ▤ Representação de dados externos em **formato binário**, utilizada na invocação de objetos remotos com CORBA
- ▤ Suporte para **múltiplas linguagens** de programação

▤ Serialização de objetos (Java RMI)

- ▤ Representação de dados externos em **formato binário**, utilizada, entre inúmeros outros usos, na invocação de objetos remotos com Java RMI
- ▤ Apenas para uso com Java

▤ XML (SOAP)

- ▤ Representação de dados externos em **formato textual** (documentos XML), utilizada na invocação de serviços web
- ▤ Suporte pra **múltiplas linguagens de programação**

⌘ *Representações textuais geralmente são bem maiores do que representações binárias equivalentes (Vantagens? Desvantagens?)*

Representação de dados - CORBA

- ⌘ ***Representação comum de dados (Common Data Representation – CDR) introduzida na versão CORBA 2.0***
 - ⌘ ***Definição de formato para todos os tipos de dados que podem ser usados como argumentos e valores de retorno em invocações remotas para objetos CORBA***
 - ⌘ ***15 tipos primitivos: short (16 bits), long (32 bits), double (64 bits), boolean, char(16 bits), float(32bits), dentre outros.***
 - ⌘ ***6 tipos compostos: sequence, string, array, struct, enumerated, union***
 - ⌘ ***Ordem dos bits (little-endian ou big-endian) segue o estilo local do remetente e é indicada em cada mensagem (conversão a critério do destinatário)***
 - ⌘ ***Valores primitivos dos tipos compostos adicionados em seqüência, seguindo uma ordem específica para cada tipo***
-

Representação de dados - CORBA

<i>Tipo</i>	<i>Representação</i>
<i>sequence</i>	Tamanho (<i>unsigned long</i>) seguido pelos elementos em ordem
<i>string</i>	Tamanho (<i>unsigned long</i>) seguido pelos caracteres em ordem
<i>array</i>	Elementos em ordem (tamanho fixo)
<i>struct</i>	Na ordem da declaração dos componentes
<i>enumerated</i>	Valores (<i>unsigned long</i>) na ordem especificada
<i>union</i>	Marcador de tipo seguido pelo membro selecionado

Representação de dados - CORBA

■ **Definição do dado Pessoa na IDL (Interface Definition Language) do CORBA:**

■ *struct Pessoa { string nome; string lugar; unsigned long ano};*

■ **Representação em CORBA CDR de um item de dado do tipo**

■ *Pessoa com os atributos {'Smith', 'London', 1934}:*

Índice (seq. de bytes) ← 4 bytes →		Comentário
0-3	5	tamanho da string
4-7	"Smith"	'Smith'
8-11	"h "	
12-15	6	tamanho da string
16-19	"Lond"	'London'
20-23	"on "	
24-27	1934	unsigned long

Serialização de objetos em Java

- ⌘ ***Serialização: atividade de “achatar” o estado de um objeto (ou de uma hierarquia de objetos relacionados) para uma forma seqüencial mais adequada para armazenamento ou transmissão***
 - ▤ Valores dos atributos primitivos escritos num formato **binário portátil**
 - ▤ Atributos não primitivos serializados recursivamente
 - ▤ Não pressupõe conhecimento sobre o tipo dos objetos para o processo de reconstrução (“**desachatamento**”)
 - ▤ Informações sobre a classe dos objetos, como nome e número de versão, são incluídas na forma serial
 - ▤ Implementada através dos métodos *writeObject* e *readObject* das classes ***ObjectOutputStream*** e ***ObjectInputStream***, respectivamente
 - ▤ Transforma tipos primitivos em formato binário
 - ▤ String e caracteres são gravados pelo método *writeUTF*
 - ⌘ ***Executada de forma transparente pelas camadas da middleware em Java (uso intensivo de Reflexão!)***
-

Exemplo de Serialização em Java

- Definição de um tipo Pessoa em Java:

```
public class Pessoa implements Serializable {  
    private String nome; private String lugar; private int ano;  
    public Pessoa(String nome, String lugar, int ano) {  
        this.nome = nome; this.lugar = lugar; this.ano = ano;  
    } // continua com a definição dos métodos...  
}
```

- Exemplo (simplificado) de serialização de um objeto do tipo

Pessoa: Pessoa p = new Pessoa("Smith", "London", 1934);

Valores serializados				Comentário
Pessoa	No. de versão (8 bytes)		h0	nome da classe, número de versão
3	int ano	java.lang.String nome:	java.lang.String lugar:	número, tipo e nome das variáveis de instância
1934	5 Smith	6 London	h1	valores das variáveis de instância

Reflexão

- ⌘ De posse de um objeto do tipo Class podemos chamar os seguintes métodos para obter mais informações sobre a classe (a lista a seguir é apenas um pequeno subconjunto dos métodos da classe Class):
 - ⌘ String getName() - retorna o nome da classe
 - ⌘ Object newInstance() - retorna um novo objeto que é instância da classe
 - ⌘ Method[] getMethods() - retorna uma lista de objetos da classe Method, representando os métodos da classe
 - ⌘ Field[] getFields() - retorna uma lista de objetos da classe Field, representando os campos da classe
 - ⌘ Method getMethod (nome, tipos dos parâmetros) - retorna o método que tem o nome e parâmetros dados, se existir.
-

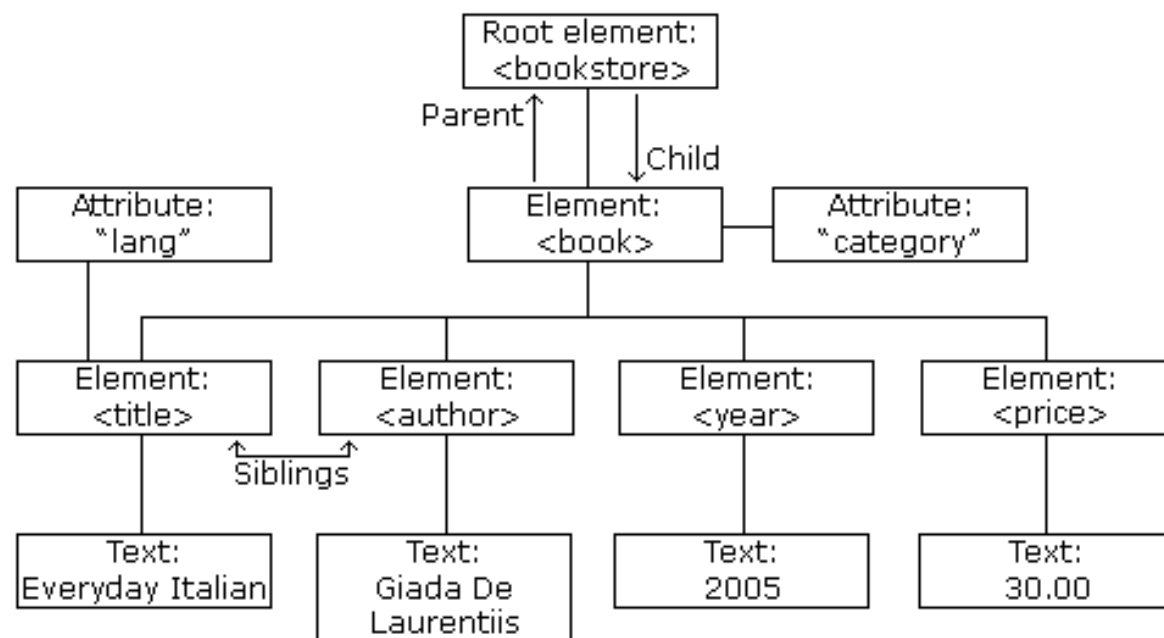
Reflexão

```
void imprimeNomeClasse(Object obj) {  
    System.out.println("O nome da classe é " +  
        obj.getClass().getName());  
}
```

```
void imprimeNomeCamposDeclarados(Object obj) {  
    try{  
  
        Field[] fl = obj.getClass().getDeclaredFields();  
        for (int i=0; i<fl.length;i++){  
            // exibindo o nome da variável  
            System.out.println(" Nome:" + fl[i].getName());  
            // exibindo o tipo de definição da variável  
            System.out.println(" Tipo: " +  
                fl[i].getType().getName());  
        }  
    }  
    catch(ClassNotFoundException e)  
        {System.out.println("Classe não encontrada ");}
```

Representação de dados externos XML

```
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```



Representação de dados externos XML

⌘ ***XML é uma linguagem de marcação de propósito geral para uso na Web, definida e mantida pelo W3C***

⇒ Projetada para **facilitar a codificação textual de documentos** estruturados

⌘ ***Itens de dados em XML são “marcados” com strings especiais chamadas marcadores (tags), que representam a estrutura lógica dos dados***

⇒ Diferença para HTML?

⌘ ***XML é extensível!***

⇒ Usuários podem **definir** seus próprios marcadores

⇒ Extensões precisam ser **acordadas no caso de documentos** compartilhados por múltiplas aplicações

Representação de dados externos XML

- ⌘ ***XML é a base para a representação de dados externos (padrão SOAP) e interfaces (padrão WSDL) nas middlewares baseadas na tecnologia de serviços web (web services)***
 - ⌘ ***Vantagens:***
 - ⌘ Permite “compreensão” e facilita monitoramento por humanos (útil em atividades de teste e depuração)
 - ⌘ Independente de plataforma
 - ⌘ ***Desvantagens:***
 - ⌘ Mensagens bem maiores que suas equivalentes em formato binário
 - ⌘ Maior tempo de processamento para empacotamento/ desempacotamento
 - ⌘ ***Alternativas:***
 - ⌘ **Compactação de mensagens**
 - ⌘ Reduz tamanho mas piora processamento
 - ⌘ XML em **formato binário**
 - ⌘ Reduz tamanho e processamento, mas impede compreensão
-

Representação de dados externos XML

- Representação em XML de um elemento do tipo Pessoa:

```
<person id="123456789">  
    <name>Smith</name>  
    <place>London</place>  
    <year>1934</year>  
    <!-- a comment -->
```

```
</person >
```

- Espaço de nomes da estrutura Pessoa

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk4.net/person">  
    <pers:name> Smith </pers:name>  
    <pers:place> London </pers:place >  
    <pers:year> 1934 </pers:year>
```

```
</person>
```

Representação de dados externos XML

⌘ ***Um esquema XML define:***

- ▤ Os **elementos e atributos** (incluindo seus tipos e valores *default*) *que podem aparecer em um documento*
- ▤ Como os elementos são **aninhados** e em qual **ordem** e **quantidade**
- ▤ Se um elemento pode ser vazio
- ▤ Se um elemento pode conter texto

⌘ ***Um mesmo esquema pode ser compartilhado por múltiplos documentos XML***

- ▤ Permite a validação de documentos de acordo com o esquema
 - ▤ Ex.: uma mensagem SOAP pode ser validada pelo processo destinatário com base no esquema XML previamente definido para esse padrão
-

Representação de dados externos XML

■ XML Schema


- Exemplo de um esquema para o tipo **Pessoa** definido em **XML Schema**:

```
<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>
```

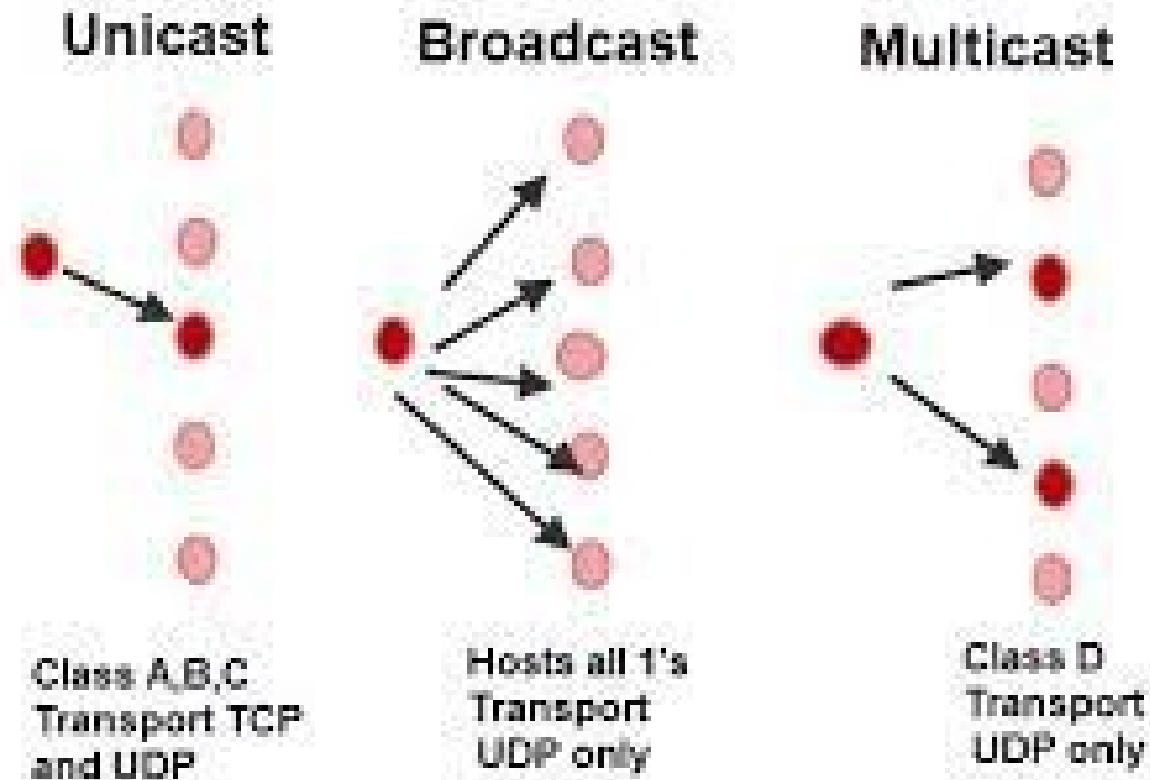
Agenda

- ⌘ *API para comunicação entre processos (seção 4.2)*
 - ⌘ *Representação externa de dados (seção 4.3)*
 - ⌘ *Comunicação em Grupo (seção 4.4)*
-

Comunicação de Grupo

- ***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***
 -  processo envia a mensagem para o grupo (não para seus membros diretamente)
 - ***Mensagens comunicadas através de operações de multicast***
-

Comunicação de Grupo



Unicast, Broadcast and Multicast IP Addressing

Aplicações da comunicação de grupo

- ***Tolerância a falhas baseada em serviços replicados***
 - ***Busca por serviços de descoberta em redes espontâneas***
 - ***Melhoria de desempenho através de dados replicados***
 - ***Propagação de eventos***
 - ***Trabalho cooperativo***
 - ***etc***
-

IP Multicast

- ***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: end. IP + porta***
 - 📁 utiliza endereços IP que iniciam por 1110 (IPv4-Classe D)
 - ***Processos se tornam membros de grupos, mas não conhecem os demais membros nem a quantidade deles.***
-

IP Multicast (cont.)

- ***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 é 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 núm. de porta
-

API Java para IP Multicast

- ◌ ***Classe MulticastSocket***
 - 📁 Derivada de DatagramSocket
 - 📁 Principais métodos:
 - ◌ joinGroup
 - ◌ leaveGroup
 - ◌ setTimeToLive
 - ◌ ***Veja exemplo no próximo slide***
-

Processo entra em um grupo de multicast e envia e recebe datagramas

```
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
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

...continuação

```
// 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();}
}
}
```

Confiabilidade e ordenação de *multicast*

◦ **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
-

Efeitos de falhas nas principais aplicações de comunicação de grupo

- ***Serviços replicados***

- 📁 causa inconsistência das réplicas: nem todas as réplicas terão processado todas as requisições

- ***Busca por serviços de descoberta***

- 📁 imune a falhas: basta que alguém responda

- ***Dados replicados***

- 📁 depende do modelo de replicação

- ***Propagação de eventos***

- 📁 dependente de aplicação

Conclusão sobre IP Multicast

- ***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 msgs
 - ***Se confiabilidade é importante: protocolos de multicast confiável (capítulo 12)***
 - 📁 ordenação total, ordenação causal
-