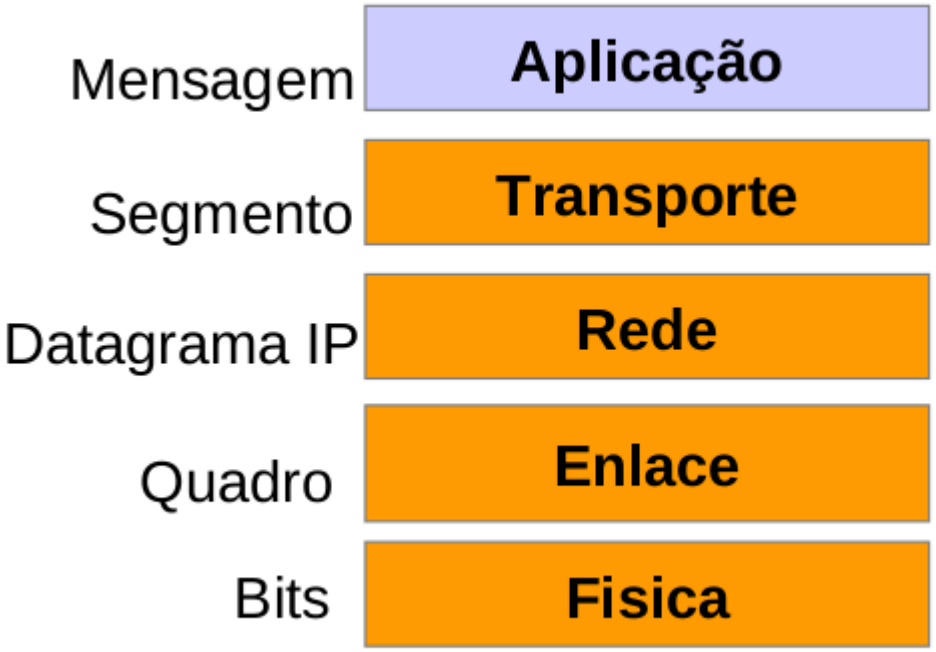
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

Sistemas Distribuídos

Ciências da Computação

Comunicação Interprocessos e Sockets em Sistemas Distribuídos

Camada de Aplicação
Arquitetura TCP/IP



Comunicação entre Processos

Aplicação distribuída em rede

- Processos trocam mensagens entre si pela rede

Processos assumem papel de cliente ou servidor

*No contexto de uma sessão de comunicação entre um par de processos, o processo que inicia a comunicação é rotulado de **cliente** e o processo que espera ser contactado para iniciar a sessão é o **servidor**.*

Mesmo em aplicações P2P devem existir processos que atuem como clientes e servidores, no sentido da comunicação entre processos.

Endereçamento de Processos

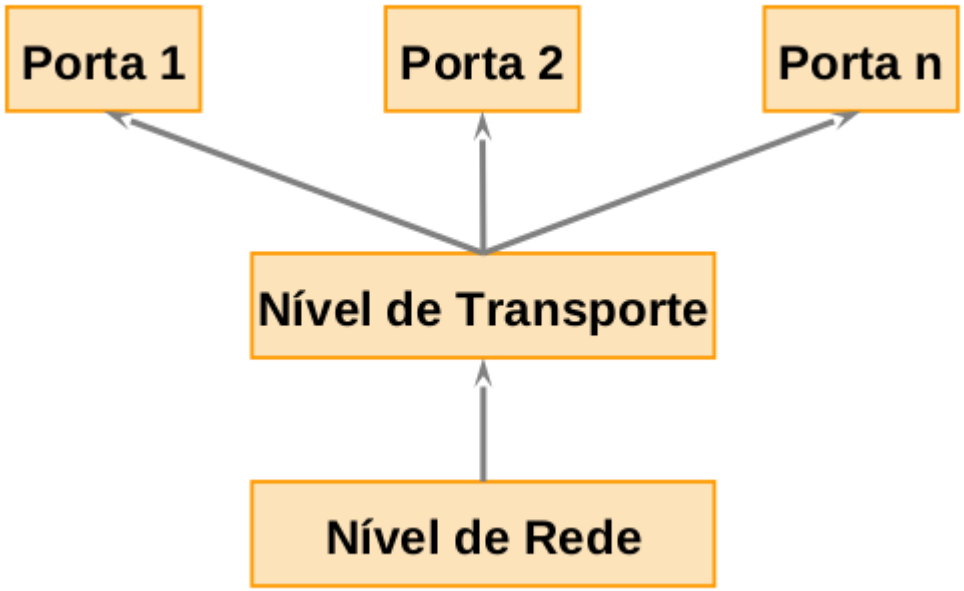
Processos são identificados e endereçados com o par (endereço IP, porta)

- Endereço IP: estação onde o processo se encontra
- Porta: identificador do processo na estação

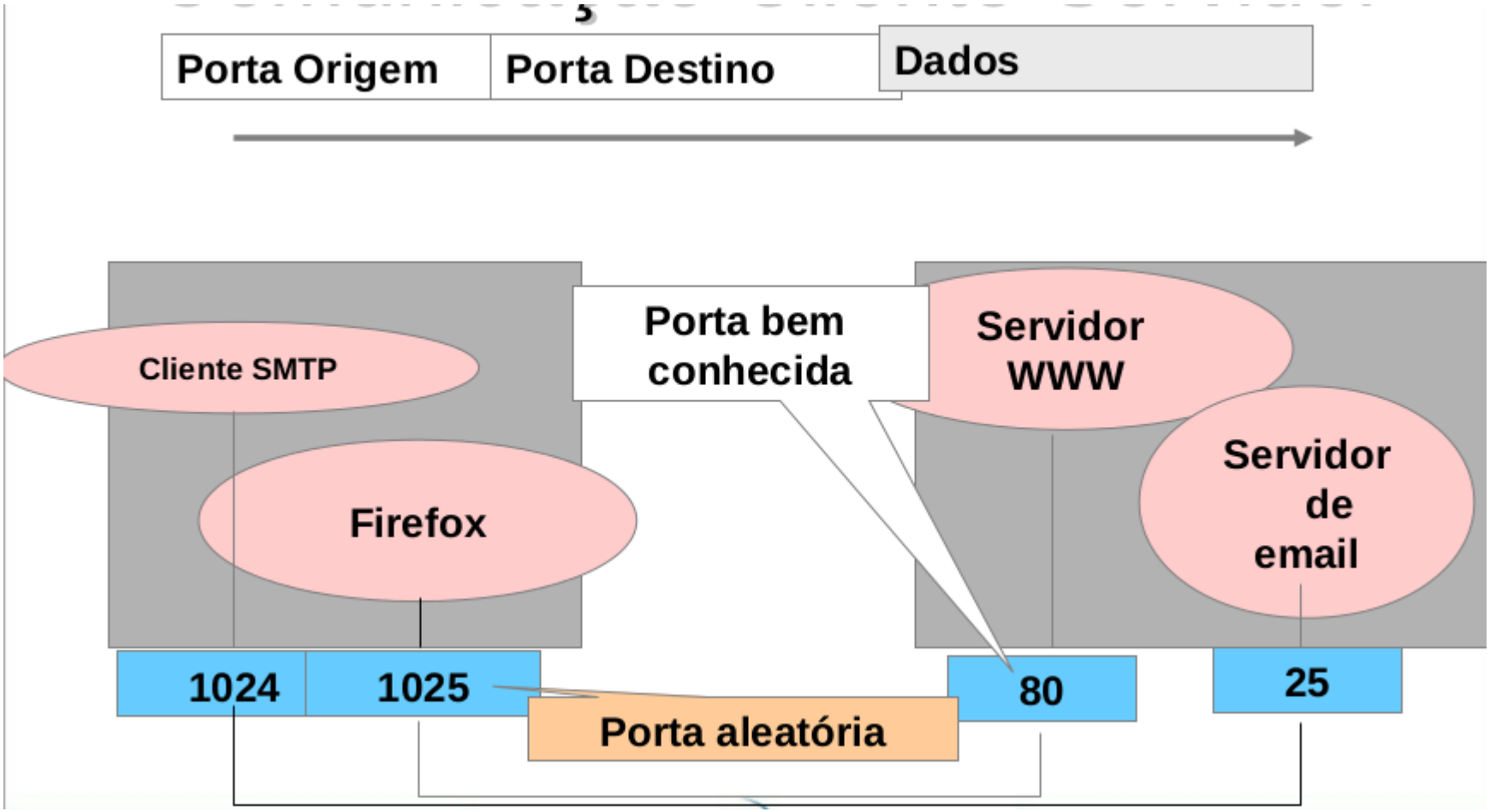
Portas

- SAP no nível de transporte
- Identificam os processos origem e destino
- Viabilizam a comunicação fim-a-fim
- Permite envio e recepção de datagramas de forma independente

Portas



Comunicação cliente-servidor



Comunicação Cliente-Servidor

Modelo geral: requisição-e-resposta

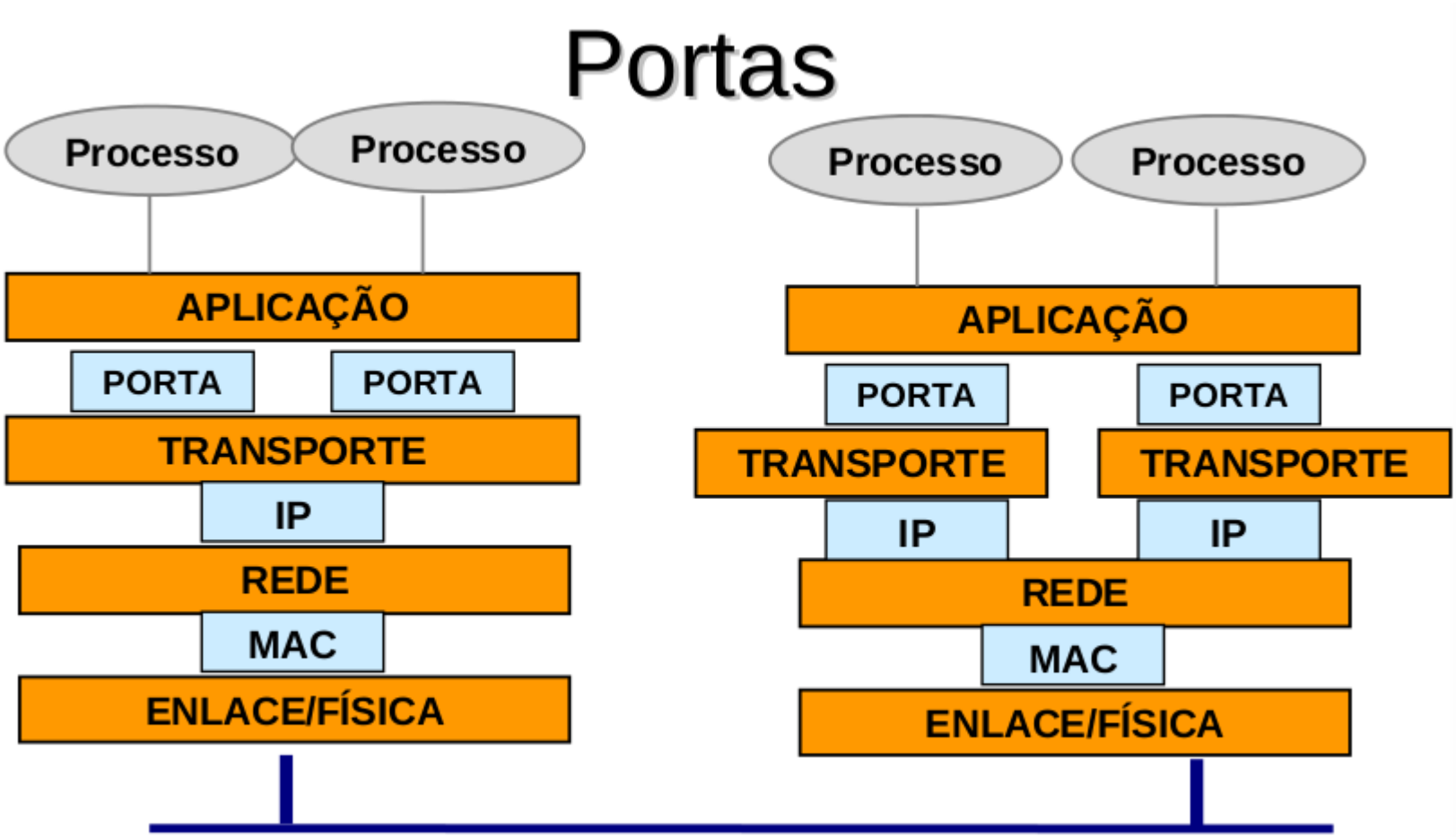
Variantes:

- síncrona: cliente bloqueia até receber a resposta
- assíncrona: cliente recupera (explicitamente) a resposta em um instante posterior
  - não-bloqueante

Implementação sobre protocolo baseado em datagramas é mais eficiente

- evita: confirmações (acks) redundantes, mensagens de estabelecimento de conexão, controle de fluxo

Comunicação cliente-servidor



Programação de sockets

Objetivo: aprender a construir aplicações cliente-servidor que se comunicam usando sockets

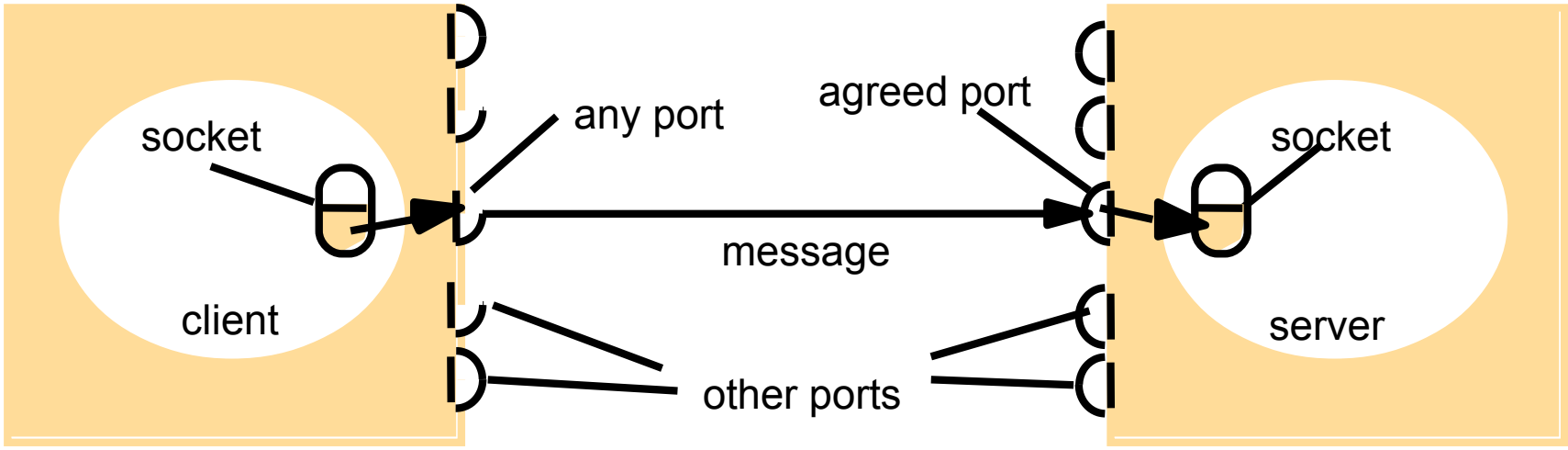
Socket API

- Introduzida no BSD4.1 UNIX, 1981
- Explicitamente criados, usados e liberados pelas aplicações
- Paradigma cliente-servidor
- Dois tipos de serviço de transporte via socket API:
- Datagrama não confiável
- Confiável, orientado a cadeias de bytes

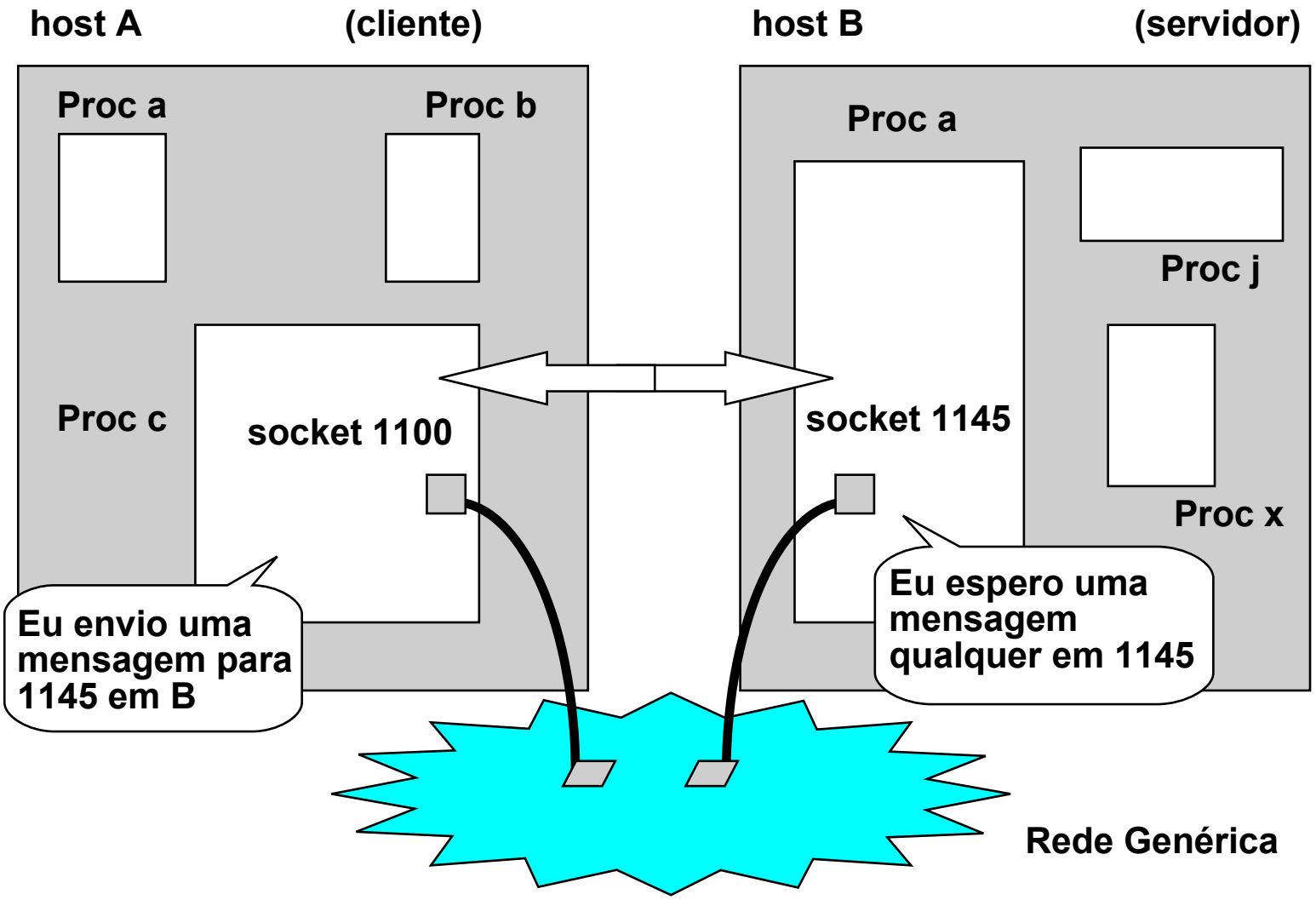
Socket

Uma interface local, criada por aplicações, controlada pelo OS (uma "porta") na qual os processos de aplicação podem tanto enviar quanto receber mensagens de e para outro processo de aplicação (local ou remoto)

API para a camada de transporte: *Sockets* e portas



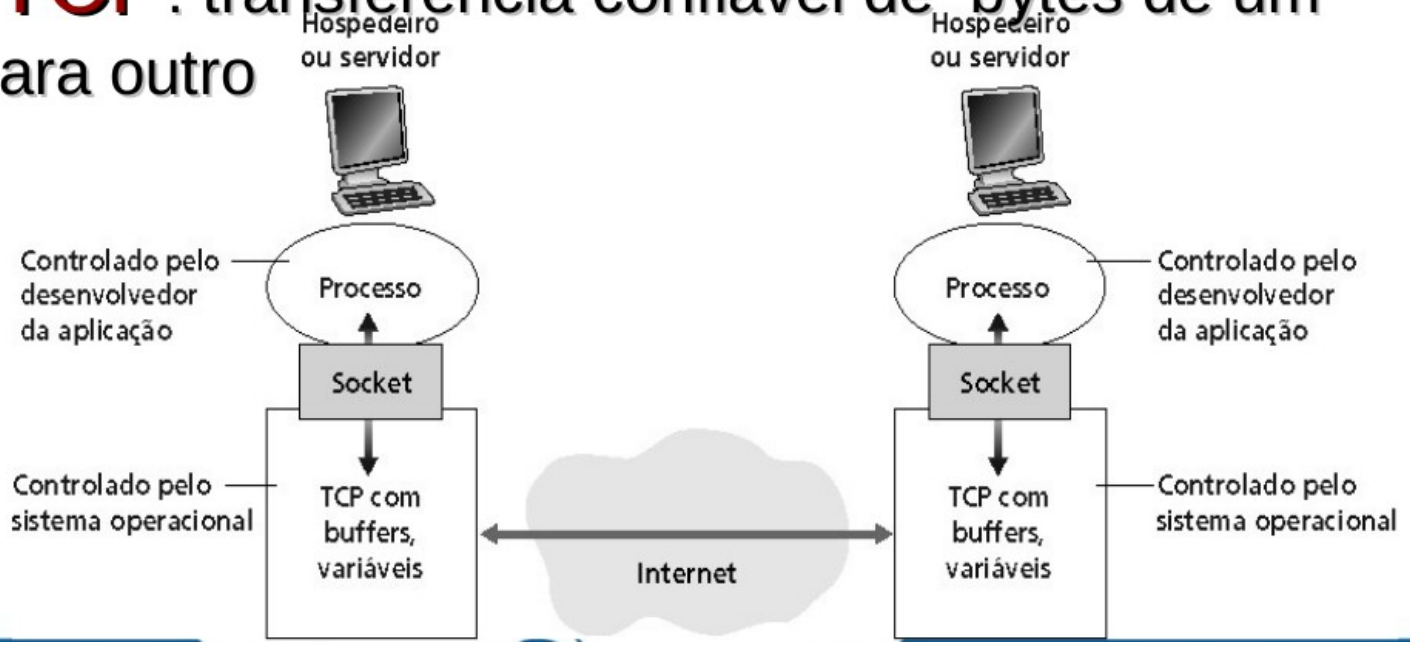
Sockets


host A
(cliente)
host B
(servidor)
Proc a
Proc b
Proc a
Proc j
Proc c
socket 1100
socket 1145
Proc x
Eu envio uma mensagem para 1145 em B
Eu espero uma mensagem qualquer em 1145
Rede Genérica

Programação de sockets com TCP

Socket: uma interface entre o processo de aplicação e o protocolo de transporte fim-a-fim (UDP or TCP)

Serviço TCP: transferência confiável de bytes de um processo para outro



Programação de sockets com TCP

Cliente deve contatar o servidor

- Processo servidor já deve estar em execução
- Servidor deve ter criado socket (porta) que aceita o contato do cliente

Cliente contata o servidor

- Criando um socket TCP local
- Especificando endereço IP e número da porta do processo servidor
- Quando o cliente cria o socket: cliente TCP estabelece conexão com o TCP do servidor

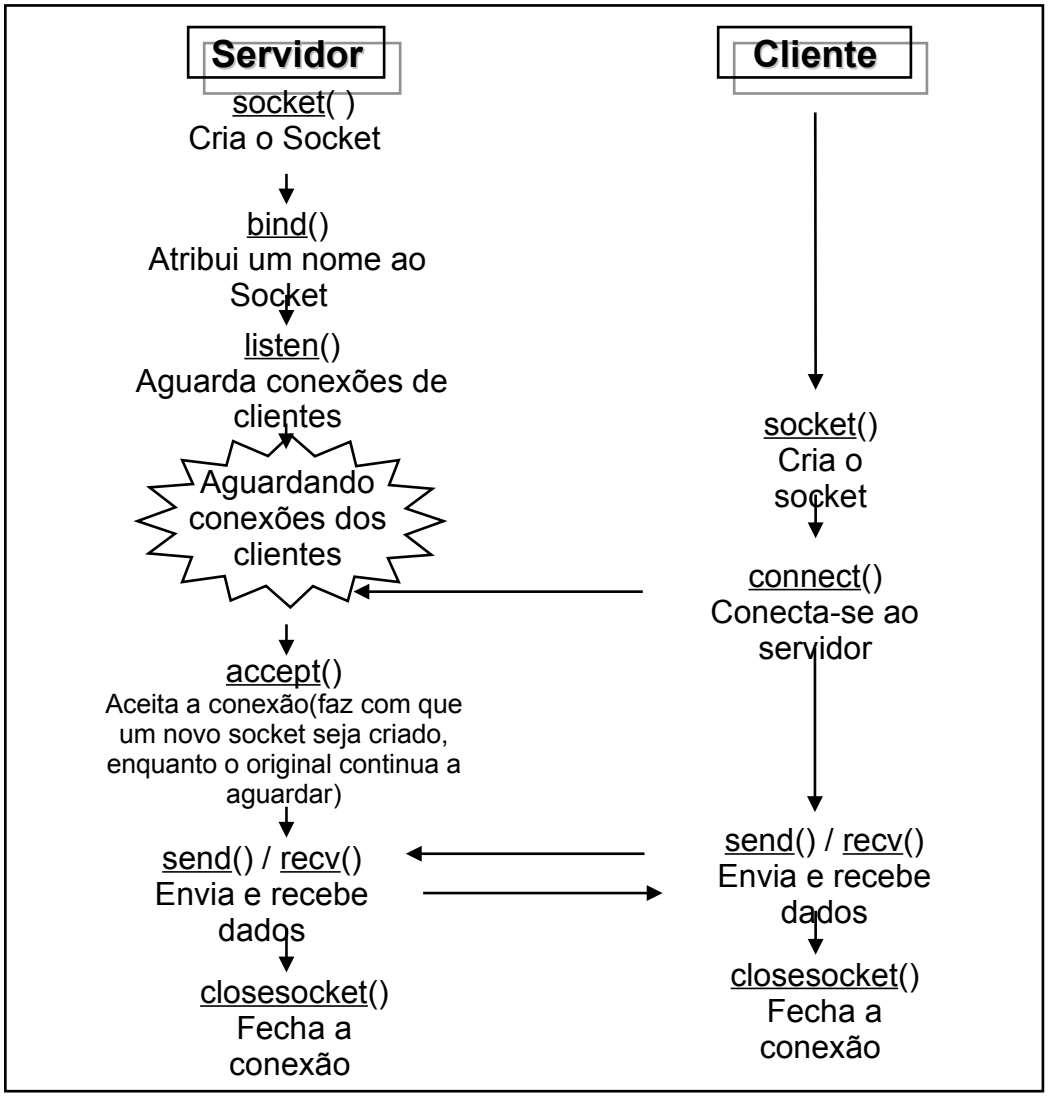
Quando contatado pelo cliente, o TCP do servidor cria um novo socket para o processo servidor comunicar-se com o cliente

- Permite ao servidor conversar com múltiplos clientes
- Números da porta de origem são usados para distinguir o cliente

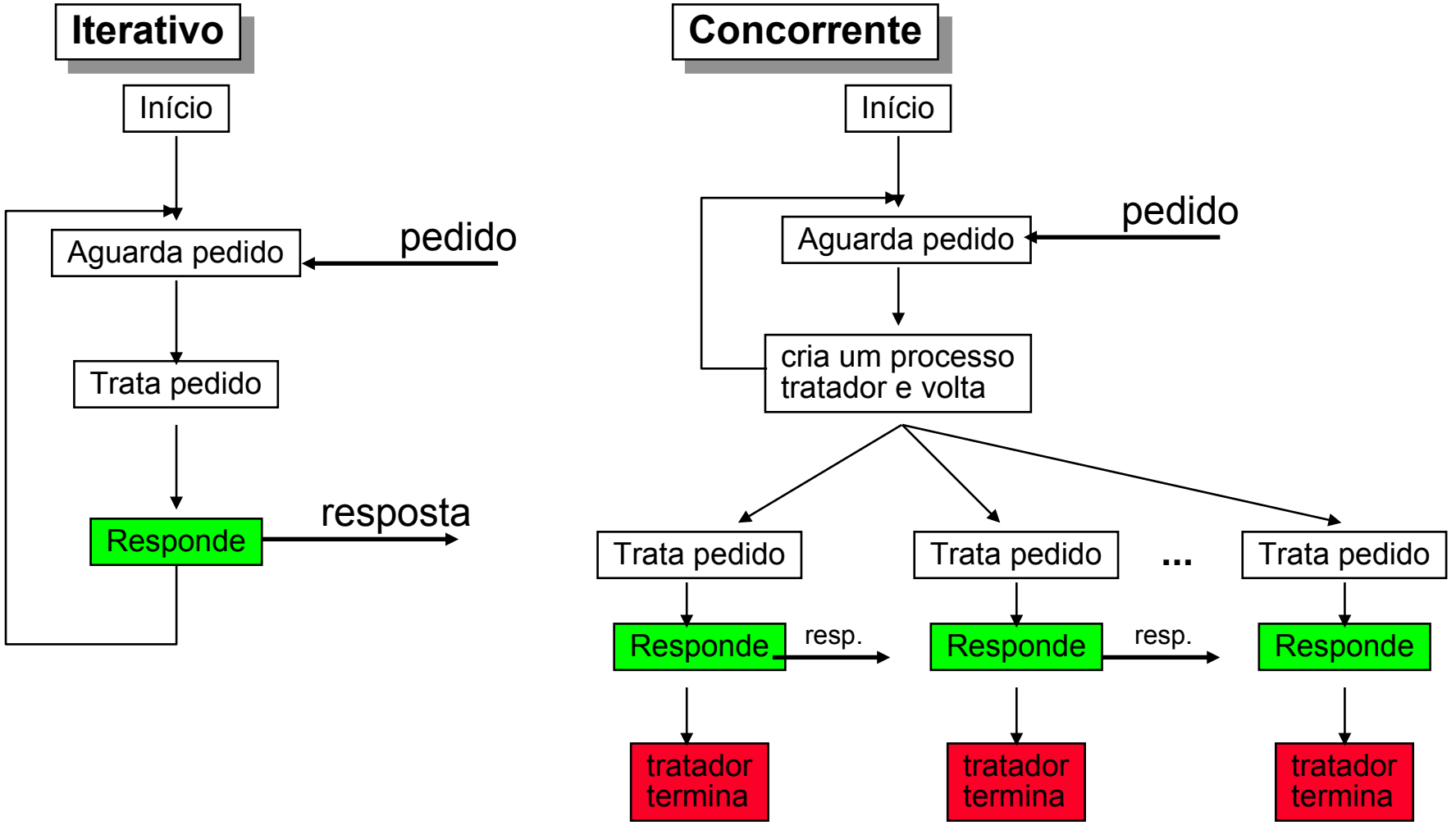
Ponto de vista da aplicação

TCP fornece a transferência confiável, em ordem de bytes (“pipe”) entre o cliente e o servidor

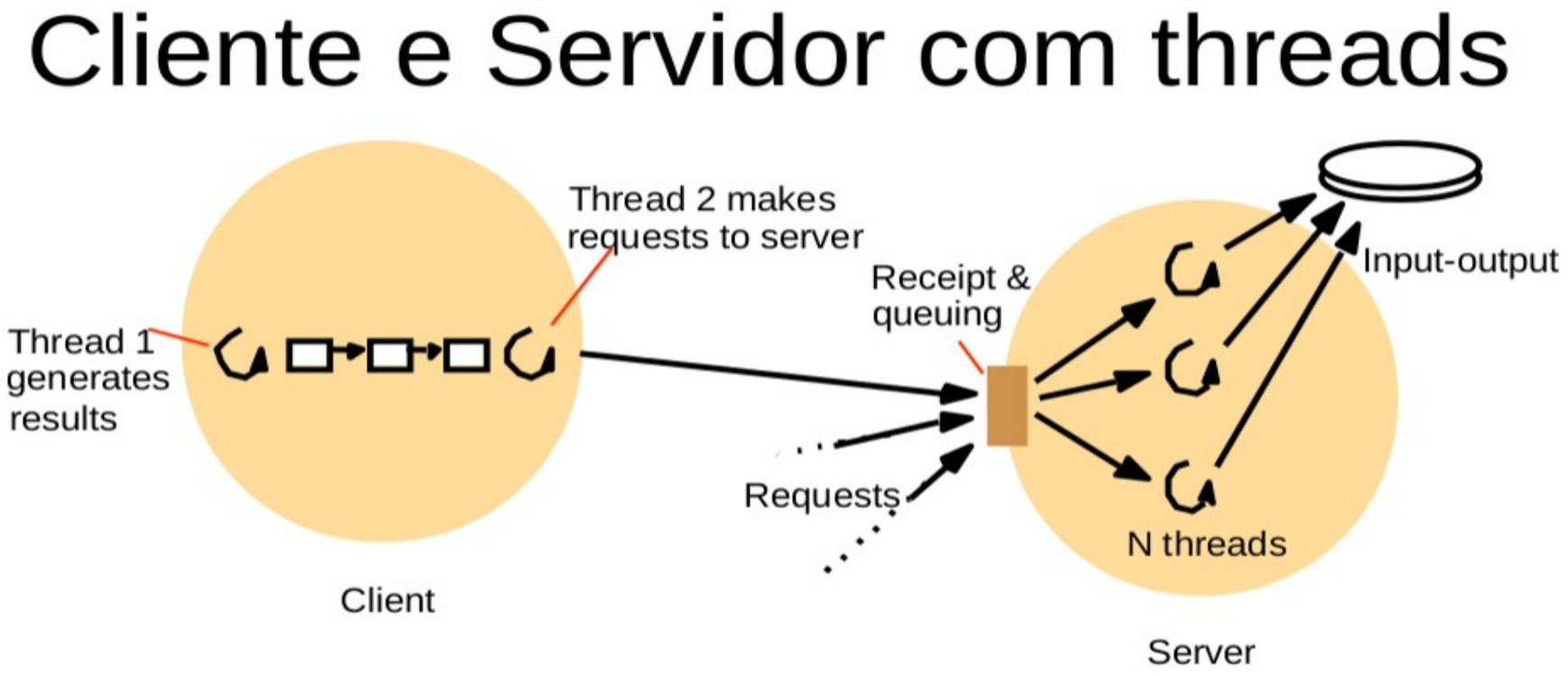
Socket TCP


Servidor
socket( )
Cria o Socket
bind()
Atribui um nome ao Socket
listen()
Aguarda conexões de clientes
Aguardando conexões dos clientes
accept()
Aceita a conexão(faz com que um novo socket seja criado, enquanto o original continua a aguardar)
send() / recv()
Envia e recebe dados
closesocket()
Fecha a conexão
Cliente
socket()
Cria o socket
connect()
Conecta-se ao servidor
send() / recv()
Envia e recebe dados
closesocket()
Fecha a conexão

Servidores Iterativos e Concorrentes

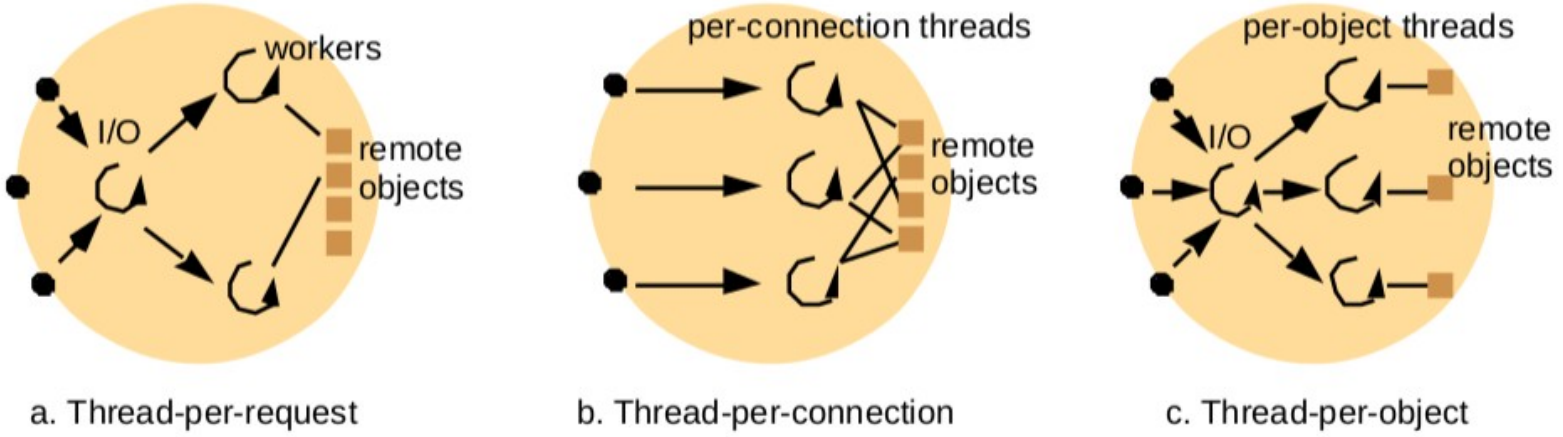


Servidores Iterativos e Concorrentes



Servidores Iterativos e Concorrentes

Arquiteturas de Servidores Multithread



a. Thread-per-request

b. Thread-per-connection

c. Thread-per-object

Cliente TCP: estabelece conexão com servidor, envia requisição e recebe resposta

```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[]) {
        // arguments supply message and hostname of destination
        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 4.3
                    String data = in.readUTF();
                    System.out.println("Received: "+ data) ;
               }catch (UnknownHostException e){
                            System.out.println("Sock:"+e.getMessage());
               }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
               }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }finally {if(s!=null) try {s.close();}catch (IOException e)
{System.out.println("close:"+e.getMessage());}}
        }
}
```

Servidor TCP: aceita conexões dos clientes e ecoa seu conteúdo de volta ao cliente

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

// this figure continues on the next slide
```

Servidor TCP: continuação

```
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();
                    out.writeUTF(data);
            } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
            } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
            } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
        }
}
```

```
/**
 * Java Client Server Socket Communication
 *
 * @author Greg Gagne, Peter Galvin, Avi Silberschatz
 * @version 1.0 - July 15, 1999
 * Copyright 2000 by Greg Gagne, Peter Galvin, Avi Silberschatz
 * Applied Operating Systems Concepts - John Wiley and Sons,
Inc.
 */
```

Funcionamento:

O cliente cria um socket (S) e envia através dele uma mensagem para o servidor pedindo a hora. O servidor cria um socket para receber as mensagens dos clientes e cria uma thread (chamada Connection) para tratar e responder individualmente cada pedido.

```
/**
 * Client.java - this client accesses the time of day from a server.
*/

public class Client
{
        public Client() {
                try {
                        Socket s = new Socket("127.0.0.1",5155);

                        InputStream  in = s.getInputStream();

                        BufferedReader  bin = new BufferedReader(new InputStreamReader(in));

                        System.out.println(bin.readLine());

                        s.close();
                }
                catch (java.io.IOException e) {
                        System.out.println(e);
                        System.exit(1);
                }
        }

        public static void main(String args[]) {
                Client client = new Client();
        }
}
```

```
/**
 * Connection.java - This is the separate thread that services each request
*/

public class Connection extends Thread
{
        public Connection(Socket s) {
                outputLine = s;
        }

        public void run() {
                // getOutputStream returns an OutputStream object, allowing ordinary file IO over the socket

                try {
                        // create a new PrintWriter with automatic flushing
                        PrintWriter pout = new PrintWriter(outputLine.getOutputStream(), true);

                        // now send a message to the client
                        pout.println("The Date and Time is " + new java.util.Date().toString());

                        // now close the socket
                        outputLine.close();
                }
                catch (java.io.IOException e) {
                        System.out.println(e);
                }
        }

        private Socket          outputLine;
}
```

```
/**
 * Server.java - This is a time of day server that listens on port 5155.
*/
public class Server
{
        public Server() {
                // create the socket the server will listen to
                try {
                        s = new ServerSocket(5155);
                }
                catch (java.io.IOException e) {
                        System.out.println(e);
                        System.exit(1);
                }

                // OK, now listen for connections
                System.out.println("Server is listening ....");
                try {
                        while (true) {
                                client = s.accept();
                                // create a separate thread  to service the request
                                c = new Connection(client);
                                c.start();
                        }
                }
                catch (java.io.IOException e) {
                        System.out.println(e);
                }
        }
```

```
/**
 * Server.java - This is a time of day server that listens on port 5155
*
* Continuação ...
*/



                    public static void main(String args[]) {
                            Server timeOfDayServer = new Server();
                    }

                    private ServerSocket s;
                    private Socket client;
                    private Connection c;
            }
```

Programação de sockets com UDP

UDP: não há conexão entre o cliente e o servidor

- Não existe apresentação
- Transmissor envia explicitamente endereço IP e porta de destino em cada mensagem
- Servidor deve extrair o endereço IP e porta do transmissor de cada datagrama recebido

UDP: dados transmitidos podem ser recebidos fora de ordem ou perdidos

Ponto de vista da aplicação

UDP fornece a transferência não confiável de grupos de bytes (datagramas) entre o cliente e o servidor

Cliente UDP: envia uma mensagem a um servidor e obtém uma resposta

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
            // args give message contents and server 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();}
   }
}
```

Servidor correspondente: recebe requisição e envia seu conteúdo de volta para o cliente

```
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());
                       System.out.println("IP: " + request.getAddress() + "Porta: " +
                        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();}
    }
}
```

Projeto de Protocolos

- Escolha do protocolo de transporte
- Serialização de dados e formatos de mensagens
- Estado do protocolo

Protocolo da Camada de Aplicação

Define

- Tipos de mensagens trocadas, como de requisição e resposta
- Sintaxe dos vários tipos de mensagens, tais como campos da mensagem e como os campos são delineados
- Semântica dos campos
- Regras para identificar como e quando um processo envia mensagens e responde a mensagens

Escolha do Serviço de Transporte

Perda de dados

- Algumas aplicações (ex.: áudio) podem tolerar alguma perda
- Outras aplicações (ex.: transferência de arquivos, telnet) exigem transferência de dados 100% confiável

Temporização

- Algumas aplicações (ex.: telefonia Internet, jogos interativos) exigem baixos atrasos para serem “efetivos”

Banda passante

- Algumas aplicações (ex.: multimídia) exigem uma banda mínima para serem “efetivas”
- Outras aplicações (“aplicações elásticas”) melhoram quando a banda disponível “aumenta”

Representação externa de dados

- Formato independente de linguagem, SO etc
- Utilizada para comunicação dos dados de requisições e respostas entre clientes e servidores
- Formato serializado
- *Marshalling*: conversão entre a representação interna e externa

Representação externa de dados

- CORBA Common Data Representation
- Serialização de objetos em Java
- XML (Extensible Markup Language)

CORBA CDR

- Representação para tipos primitivos
  - inteiros, ponto flutuante, octeto
  - big-endian e little-endian
  - posicionamento de cada item de dados na msg.
- Representação para tipos construídos
  - sequência, string, array, struct, enumeração, união
  - construída a partir das sequências de bytes correspondentes aos tipos primitivos constituintes
- Informação de tipo: subentendida através da definição das operações em IDL

Representação de tipos construídos em CORBA CDR

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

Mensagem em CORBA CDR

index in sequence of bytes ←	4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on__"	
24–27	1934	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

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

Marshalling em CORBA

- O código gerado automaticamente a partir das definições em IDL das operações
- Converte os parâmetros e valores de retorno das operações
- Seguindo as regras de representação CORBA CDR
  - *Stub*
    - *marshalling* de parâmetros e *unmarshalling* do valor de retorno
  - *Skeleton*: vice-versa

Serialização em Java

- Objetos e itens de dados primitivos
- 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 *Serializable*
- Objetos remotos: a referência é serializada
  - Referência para outros objetos é serializada com *Handles*

Formato de serialização de Java

Exemplo: Serialização do seguinte objeto:
Person p = new Person("Smith", "London", 1934);

Serialized values

Person	8-byte version number		h0
3	int year	java.lang.String name:	java.lang.String place:
1934	5 Smith	6 London	h1

Explanation

class name, version number

number, type and name of instance variables

values of instance variables

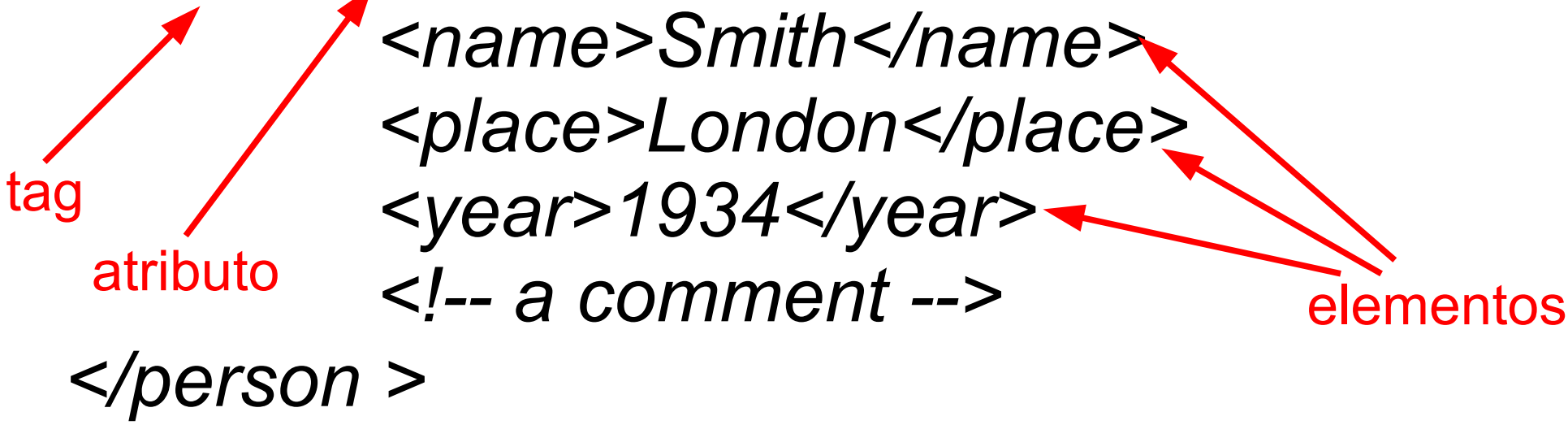
The true serialized form contains additional type markers; h0 and h1 are handles

XML (Extensible Markup Language)

- Define a estrutura lógica de documentos
- 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
- Extensível: novos tags podem ser definidos
- Autodescritiva: tags, esquemas e namespaces
- Representação textual ao invés de binário aumenta significativamente o tamanho da mensagem

Definição da estrutura *Pessoa* em XML

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



Namespaces

- Permitem definir contextos para os marcadores (*tags)* utilizados em um documento
- Referenciados através de URLs
- Evitam choques de nomes de *tags* em contextos diferentes

```
<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>
```

Esquemas XML

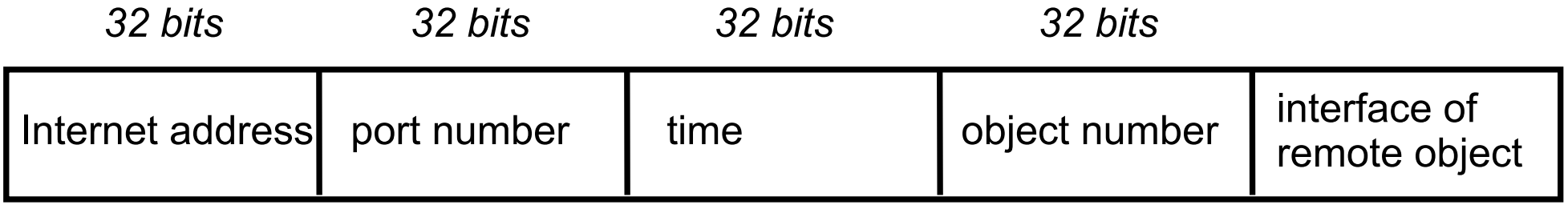
- Definem:
  - os elementos e atributos que podem aparecer em documentos XML (i.e., um vocabulário)
  - como os elementos são aninhados
  - a ordem, o número e o tipo dos elementos
- Usados para validar documentos XML
- Um mesmo esquema pode ser compartilhado por vários documentos

Um esquema XML para a estrutura *Pessoa*

```
<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>
```

Representação de Referências de Objetos

- Necessária quando objetos remotos são passados como parâmetro
- A referência é serializada (não o objeto)
- Contém toda informação necessária para identificar (e endereçar) um objeto unicamente no sistema distribuído



Exercício prático

- *Marshalling* de objetos Java em XML
- Utilizar funcionalidades padrão da biblioteca de Java para serializar objetos em formato XML
- Descrever o resultado obtido
- Utilizar uma classe representativa como exemplo

Multicast

- 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*

Aplicações da comunicação de grupo

- Tolerância a falhas baseada em serviços replicados
- Serviços de descoberta em redes espontâneas
- Melhoria de desempenho através de dados replicados
- Propagação de notificação de eventos
- Trabalho cooperativo

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)
- Processos se tornam membros de grupos, mas não conhecem os demais membros

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

Broadcasting e Multicasting

Broadcasting

- Envio de mensagens para todas as máquinas em uma subrede
- Endereço 255.255.255.255 (broadcast local) ou porção do endereço de máquina igual com bits iguais a 1.

Multicasting

- Envio de mensagens para grupo definido de máquinas

API Java para IP Multicast

- Classe MulticastSocket
  - Derivada de DatagramSocket
  - Principais métodos:
    - joinGroup
    - leaveGroup
    - setTimeToLive

Peer entra no grupo e envia/recebe mensagens

```
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);
```

// continua no próximo slide

Peer entra no grupo e envia/recebe mensagens (cont.)

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

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
```

Exercício prático de programação (sockets unicast)

- **Aplicação Cliente:**
  - registrar-se no Servidor informando: identificador do usuário (e.g., e-mail), IP e Porta utilizados para receber mensagens de outros clientes;
  - Procurar o identificador de um outro usuário no servidor quando desejar estabelecer uma comunicação;
  - Usar TCP para se comunicar com o servidor e UDP com outros clientes.
- **Aplicação Servidor:**
  - Autenticar usuário requisitante
  - Armazenar informações de registro dos clientes em uma tabela
  - Atender requisições dos clientes que desejam obter informações de outros clientes
  - Usar TCP para se comunicar com os clientes

Problemas de Implementação

- Produtor-Consumidor com Buffer Limitado
  - com monitores
  - com semáforos
- Broadcasting e multicasting
- Servidor de *echo* multitarefa
  - cliente envia mensagem
  - servidor envia mensagem de volta
  - deve haver uma mensagem de término da comunicação

Créditos

Prof. Sérgio T. Carvalho

sergio@inf.ufg.br