Sistemas Distribuídos

Comunicação entre Processos

Edeyson Andrade Gomes

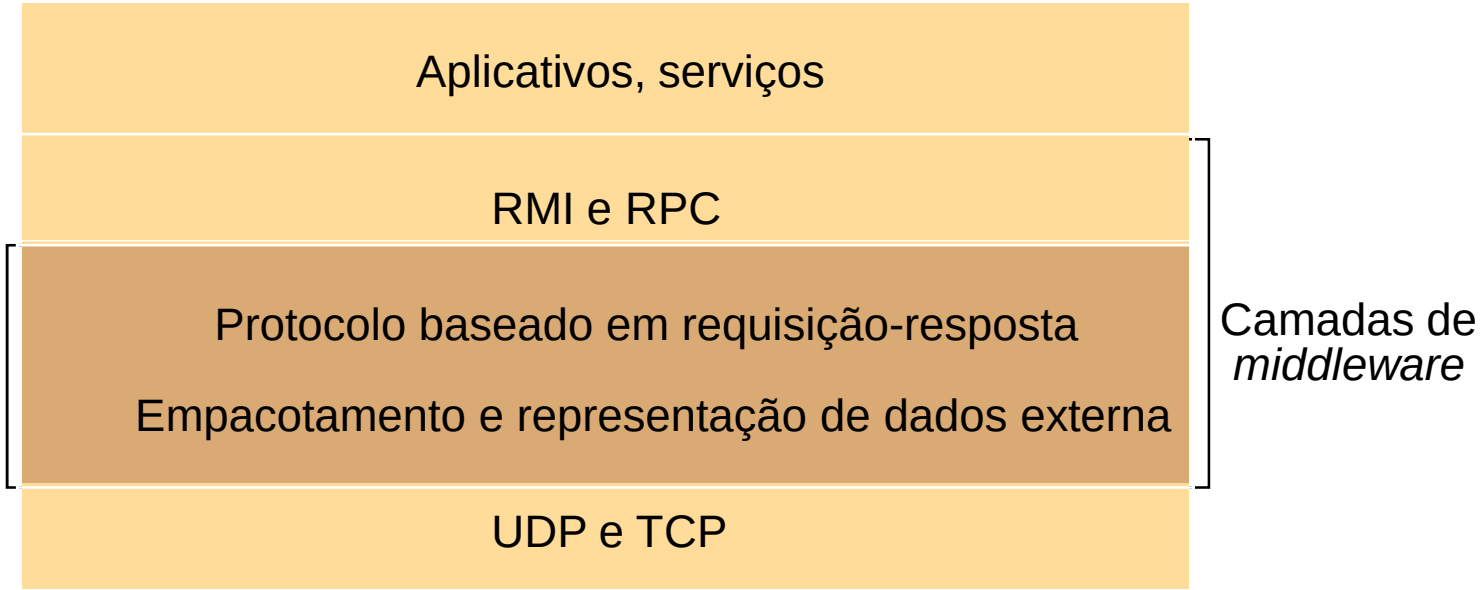
www.edeyson.com.br

SUMÁRIO

- Visão geral
- A API para protocolos Internet
- Representação externa de dados e empacotamento
- Comunicação cliente-servidor
- Comunicação em grupo
- Comunicação entre processos no UNIX

VISÃO GERAL

- Como tratar projeto de componentes?
- Como os *middlewares* e aplicativos utilizam UDP e TCP?
- Visão do programador.



VISÃO GERAL

- Interface de programa aplicativo para UDP
  - Abstração de **passagem de mensagem**
  - Datagrama
    - Pacotes independentes contendo essas mensagens.
  - Nas APIs Java e Unix o soquete é utilizado para identificar o destino.
- Interface de programa aplicativo para TCP
  - Abstração de um **fluxo (*stream*) bidirecional**.
  - Fluxo contínuos de dados, sem noção de limites da mensagem.
  - Dados são enfileirados na chegada até que o consumidor esteja pronto para recebê-los.

VISÃO GERAL

- Como transformar objetos e estruturas de dados numa forma conveniente para envio em mensagens pela rede?
  - Diferentes computadores podem utilizar diferentes representação para tipos simples de dados.
- Representação conveniente para referência a objeto.
- Protocolos convenientes para suportar comunicação cliente-servidor e em grupo.
  - Requisição/resposta → cliente/servidor na forma RMI ou RPC
  - Protocolos *multicast* → comunicação em grupo

API para protocolos Internet

- Características da comunicação entre processos
  - Comunicação síncrona e assíncrona
  - Confiabilidade e ordenamento
- Soquetes
  - API Java para endereços Internet
- Comunicação por datagrama UDP
  - Modelo de falhas
  - API Java para datagramas UDP
- Comunicação por fluxo TCP
  - Modelo de falhas
  - API Java para fluxos TCP

API2IP – Características

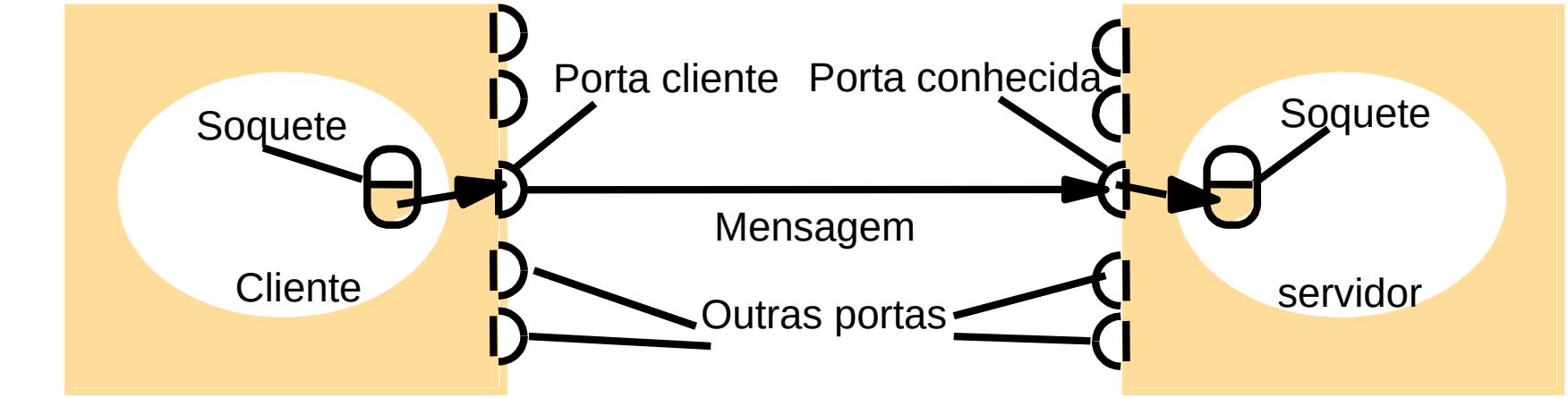
- Operações de comunicação de mensagem: *send* e *receive*.
- Definidas em termos de destino e mensagens.
- Comunicação síncrona e assíncrona
  - Uma fila é associada a cada destino de mensagem.
  - Síncrona:
    - Sincronização entre processos remetente e destino a cada mensagem
    - *Send* e *receive* são operações bloqueantes.
  - Assíncrona:
    - *Send* é não bloqueante.
    - *Receive* tem duas variantes. Sistemas atuais são bloqueantes.

API2IP – Características

- As mensagens são enviadas para destinos identificados pelo par **endereço IP e porta**.
- Para proporcionar transparência de localização, os clientes se referem aos serviços pelo nome e usam um servidor de nomes.
- Confiabilidade
  - Validade: garantia que as mensagens foram entregues.
  - Integridade: mensagens não corrompidas e sem duplicação.
- Ordenamento
  - Algumas aplicações exigem a ordem de emissão.

API2IP – Soquetes

- A comunicação entre processos consiste em transmitir uma mensagem entre um soquete de um processo e um soquete de outro processo.



API2IP – Soquetes

- Características
  - O processo deve vincular um endereço IP e porta.
  - Mensagens só podem ser lidas pelo processo associado.
  - O mesmo soquete pode ser usado para envio e recebimento.
  - Existem 2^{16} (65.536) números de portas num computador.
  - Um processo pode utilizar várias portas, mas dois processos não podem compartilhar uma porta no mesmo computador.
    - Processos que usam *multicast* IP são um exceção.
  - Vários processos podem enviar mensagens para uma porta.
  - Cada soquete é associado a um protocolo particular, TCP ou UDP.

API2IP – Soquetes

- API Java para endereços Internet
  - A classe *InetAddress* é utilizada por UDP e TCP.
  - As instâncias de *InetAddress* são criadas através do método estático *InetAddress*, passando um nome como argumento.
    - ***InetAddress a Computer = InetAddress.getByName("www.frb.edu.br")***
  - O método usa o DNS para obter o endereço IP.
  - A classe encapsula detalhes da representação dos endereços IP
    - IPv4 utiliza 4 bytes
    - IPv6 utiliza 16 bytes

API2IP – Datagrama UDP

- Tamanho da mensagem
  - Mensagens grandes demais são truncadas
- Bloqueio
  - Normalmente, o *send* é não bloqueante e o *receive* bloqueante.
- Timeouts
  - Evita que um processo espere indefinidamente um recepção.
- Recepção anônima
  - O método *receive* não especifica o remetente.
  - É possível associar um soquete de datagrama a um remetente.

API2IP – Datagrama UDP

- Modelo de falhas
  - Falhas por omissão
    - Mensagens descartadas por *checksum* ou espaço de buffer.
  - Ordenamento
    - Rotas diferentes.
  - As falhas podem ser controladas pelas aplicações.
- API Java para datagramas UDP
  - As classes *DatagramPacket* e *DatagramSocket*
    - Métodos *DatagramPacket: getData, getPort, getAddress*
    - Métodos *DatagramSocket: send, receive, setSoTimeout* e *connect (único remetente)*
  - Instâncias de *DatagramPacket* são transmitidas com *send* e *receive*.

API2IP – Datagrama UDP

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[ ]){
                // args fornece o conteúdo da mensagem e o nome de host do servidor
                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();}
   }
}
```

API2IP – Datagrama UDP

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

API2IP – Datagrama UDP

- Emprego do protocolo UDP
  - *Voice over* IP – VoIP
  - Aplicações multimidia
  - *Domain Name Service* – DNS

API2IP – Fluxo TCP

- Características da rede ocultas pela abstração de fluxo
  - Tamanho das mensagens
    - Aplicativos escolhem o tamanho a ser transmitido em um ou mais pacotes
  - Mensagens perdidas
    - O protocolo TCP usa um esquema de confirmação com janelas deslizantes
  - Controle de fluxo
    - Combinação dos processos que lêem com os que escrevem
  - Duplicação e ordenamento
    - Identificadores são associados a cada datagrama IP
  - Destinos de mensagem
    - Estabelecimento de uma conexão antes do envio da mensagem

API2IP – Fluxo TCP

- Dois fluxos de dados: entrada e saída
- Necessidade de encerramento do soquete.
- Problemas relacionados a comunicação por fluxo:
  - Correspondência de itens de dados
    - Ex.: P1 envia(int, double) → P2 recebe(int,double)
  - Bloqueio
    - Ex.: P1 é bloqueado, ao tentar enviar, pois o buffer de P2 está cheio
  - *Threads*
    - Ao aceitar uma nova conexão é criada uma nova *thread* separada para se comunicar com o novo cliente.

API2IP – Fluxo TCP

- Modelo de falhas
  - Integridade: *checksum* contra pacotes corrompidos e número de seqüência contra duplicação.
  - Validade: utilização de timeouts e retransmissões.
- TCP não garante entrega para todas as dificuldades
  - Limites de retransmissão excedida ou rompimento da rede.
- API Java para fluxo TCP
  - As classes *ServerSocket* e *Socket*
    - Métodos *ServerSocket: accept*
    - Métodos *Socket: getInputStream* e *getOutputStream*
  - O resultado da execução *accept* é uma instância de *Socket*.

API2IP – Fluxo TCP

```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[ ]) {
        // os argumentos fornecem a mensagem e o nome de host do destino
        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 é uma codificação de string
                    String data = in.readUTF(); // Unicode Transformation Format
                    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){/*close falhou*/}}
        }
}
```

API2IP – Fluxo TCP

```
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());}
   }
}

// o código continua no próximo slide
```

API2IP – Fluxo TCP

```
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 {                                       // Servidor de eco
                    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 falhou*/}}
        }
}
```

Rep. Ext. e Empacotamento

- As mensagens são seqüências puras de bytes.
- Dados primitivos podem ser *big-endian* ou *little-endian*.
- Conjunto de códigos usado para representar caracteres:
  - Unix, IBM, Intel → *American Standard Code for Information Interchange* – ASCII (Um byte por caractere)
  - Unisys →*Extended Binary Coded Decimal Interchange Code* – EBCDIC
  - Macintosh →ASCII misto
  - Windows, Java → Unicode (dois bytes por caractere)

Rep. Ext. e Empacotamento

- Métodos para troca de valores de dados binários:
  - Conversão para um formato externo (PCs diferentes).
  - Transmissão no formato do remetente junto com a indicação.
- Padrão aceito para a representação externa de dados e valores primitivos.
- Empacotamento (*marshalling*): remetente monta o conjunto de itens de dados para transmissão em uma mensagem.
- Desempacotamento (*unmarshalling*) é executado no receptor.
- Compactação pode ser tratado no projeto.

Rep. Ext. e Empacotamento

- Representação externa de dados e empacotamento:
  - *Common Data Representation* – CDR do CORBA
    - Pode ser usada por diversas linguagens de programação.
    - Executado por uma camada de *middleware*, empacotado em uma forma binária e inclui apenas os valores dos objetos transmitidos.
  - Serialização de Objetos da linguagem Java
    - Usado apenas na linguagem Java.
    - Executado por uma camada de *middleware*, empacotado em uma forma binária e inclui informações sobre os tipos de forma serializada.
  - *Extensible Marckup Language* – XML
    - Define um formato textual (maior) para representar dados estruturados.
    - Podem se referir a conjuntos de nomes (tipos) definidos externamente.

R.E.E. – CDR do CORBA

- Representação externa de dados definida no CORBA 2.0
- Representa argumentos e valores de retorno, o tipo de um item de dados não é passado na mensagem
- Tipos primitivos:
  - *short* (16 bits), *long* (32 bits), *unsigned short*, *unsigned long*, *float* (32 bits), *double* (64 bits), *char*, *boolean* (TRUE, FALSE), *octet* (8 bits) e *any* (qualquer tipo primitivo ou construído).
  - Transmissão na ordem do remetente (*big-endian / little-endian*).
  - Cliente e servidor acordam um código para caracteres.

R.E.E. – CDR do CORBA

- Tipos construídos ou compostos:
  - *Sequence*: Comprimento (*unsigned long*) seguido de seus elementos, em ordem.
  - *String*: Comprimento (*unsigned long*) seguido pelos caracteres que o compõem (um caractere pode ocupar mais de um byte).
  - *Array*: Elementos de vetor, fornecidos em ordem (nenhum comprimento especificado, pois é fixo.
  - *Struct*: Na ordem da declaração dos componentes.
  - *Enumerated*: *unsigned long* (os valores são especificados pela ordem declarada)
  - *Union*: identificador de tipo seguido do membro selecionado

R.E.E. – CDR do CORBA

- Um valor primitivo de *n* bytes (*n* = 1, 2, 4 ou 8) é anexado a uma seqüência de acordo com seu tamanho. É criado um índice, múltiplo de *n* no fluxo de bytes e que inicia em zero.

Índice na seqüência de bytes	← 4 bytes →	*Observações sobre A representação*
0–3	5	*Comprimento do string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*Comprimento do string*
16–19	"Lond"	*'London'*
20-23	"on__"	
24–27	1934	*unsigned long*

A forma simplificada representa uma *struct* Person com valor: {'Smith', 'London', 1934}

R.E.E. – CDR do CORBA

- Empacotamento no CORBA:
  - Os tipos das estruturas de dados e os tipos dos itens de dados básicos estão descritos no *Interface Definition Language* – IDL
  - Usando o IDL para descrever a estrutura *Person*

```
Struct Person{
    String name;
    String place;
    Unsigned long year;
};
```

  - O compilador da interface CORBA faz o *marshalling* e *unmarshalling* apropriados através da estrutura.

R.E.E. – Serialização Java

- Objetos e valores de dados primitivos podem ser enviados.
- Classe Java equivalente a *struct Person:*

```
public class Person implements Serializable{
    private String name;
    private String place;
    private int year;
    public Person (string a Name, string aPlace, int aYear){
        name = aName;
        place = aPlace;
        year = aYear;
    }// outros metodos
}
```

R.E.E. – Serialização Java

- A interface *Serializable* não tem método. (Pacote *java.io.)*
- Simplifica um objeto ou conjunto em uma forma seqüencial.
- O processo responsável pela desserialização não conhece o objeto, mas deve ser capaz de reconstruí-lo.
- As informações das classes são nome e versão.
- Um objeto é serializado junto com todos os outros que ele referencia.
- O identificador (*handle*) é uma referência a um objeto dentro da forma serializada.
- Ocorrência subseqüente de objeto não é gravada, só *handle*.

R.E.E. – Serialização Java

- Tipos primitivos são gravados em um formato binário portável, usando métodos da classe *ObjectOutputStream*.
- Strings e caracteres são gravados pelo método *writeUTF* usando o formato *Universal Transfer Format* – UTF-8.
- Para serializar deve-se criar uma instância da classe *ObjectOutputStream* e invocar seu método *writeObject*, passando o objeto a ser serializado como argumento.
- Na desserialização é criada uma instância de *ObjectInputStream* e invocado o método *readObject*.
- Serialização e desserialização são executados na camada de *middleware*.

R.E.E. – Serialização Java

- *Person p = new Person("Smith","London", 1934)*

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 de nome das variáveis de instancia*
1934	5 Smith	6 London	h1	*Valores das variáveis De instancia*

Na realidade, a forma serializada inclui marcas adicionais de tipos:
h0 e h1 São identificados.

R.E.E. – XML

- Definida pelo *World Wide Web Consortium* – W3C.
- Representa um texto e os detalhes de sua estrutura.
- *Tags* são usadas para descrever a estrutura lógica dos dados e para associar pares atributo-valor as estruturas lógicas.
- Utilização:
  - Documentos estruturados para web.
  - Arquivamento e recuperação de sistemas.
  - Especificação de interfaces com o usuário.
  - Codificação de arquivos de configuração em S.O..
  - SOAP (*Simple Object Access Protocol*) é um formato XML, com *tags* publicadas para utilização em serviços web.

R.E.E. – XML

- O uso de texto torna a XML independente de plataforma.
- Podem ser lidas por humanos.
- Tempo de transmissão e processamento maior.
  - Requisição SOAP são 14 vezes maior que o CORBA e demora 882 vezes mais que uma invocação CORBA. (Olson e Ogbuji)
- Arquivos e mensagens podem ser compactadas.
  - Ex.: HTTPv1.1
- Pode ser estendido pelos usuários que criam as suas próprias *tags*.

R.E.E. – XML

- Elementos e atributos XML
  - Elemento: conjunto (contêiner) de dados caractere entre *tags*.
    - Tag vazia: <european/>
  - Atributo: usado para rotular dados

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

R.E.E. – XML

- Análise (*parsing*) e documentos bem formados
  - CDATA
    - Denotação de caracteres especiais.
    - Ex.: colocação de um apóstofro
      - *<place> King&apos Cross</place>*
      - *<place> <!CDATA [King's Cross]]></place>*
  - Prólogo XML
    - Primeira linha do documento XML
    - Deve especificar, pelo menos, a versão do XML usada.
    - Ex.:
      - *<?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>*

R.E.E. – XML

- Espaços de nomes na XML
  - Fornecem uma maneira para dar escopo aos nomes.
  - Conjunto de nomes para uma coleção de tipos e atributos de elemento, que é referenciado por um URL.
  - Permite utilizar vários conjuntos de definições externas em diferentes espaços de nomes, sem risco de conflito de nomes.

```
<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>
```

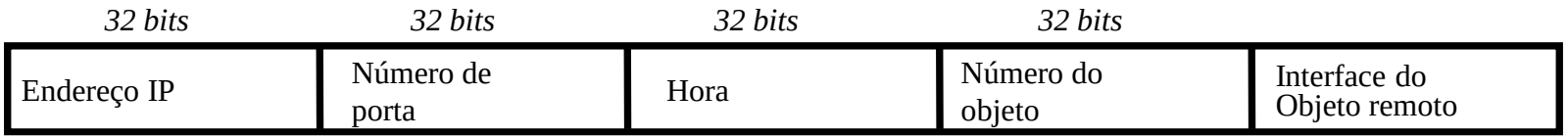
R.E.E. – XML

- Esquemas XML
  - Define os elementos e atributos que podem aparecer em um documento. A ordem, quantidade, se está vazio ou se pode conter texto e o modo como os elementos são aninhados.

```
<xsd:schema  xmlns:xsd = URL das definições de esquema XML >
      <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>
```

R.E.E. – Ref. Objs. Remotos

- Uma referência de objeto remoto é o identificador de um objeto remoto, e é válido em todo o SD.
- A referência de objeto remoto é passada na mensagem de invocação para especificar qual objeto deve ser ativado.
- Exclusividade de uma referência:
  - Concatenar IP, porta, hora de criação e nº de objeto.
  - Em Java RMI simples, msgs enviadas para IP, processo e porta.

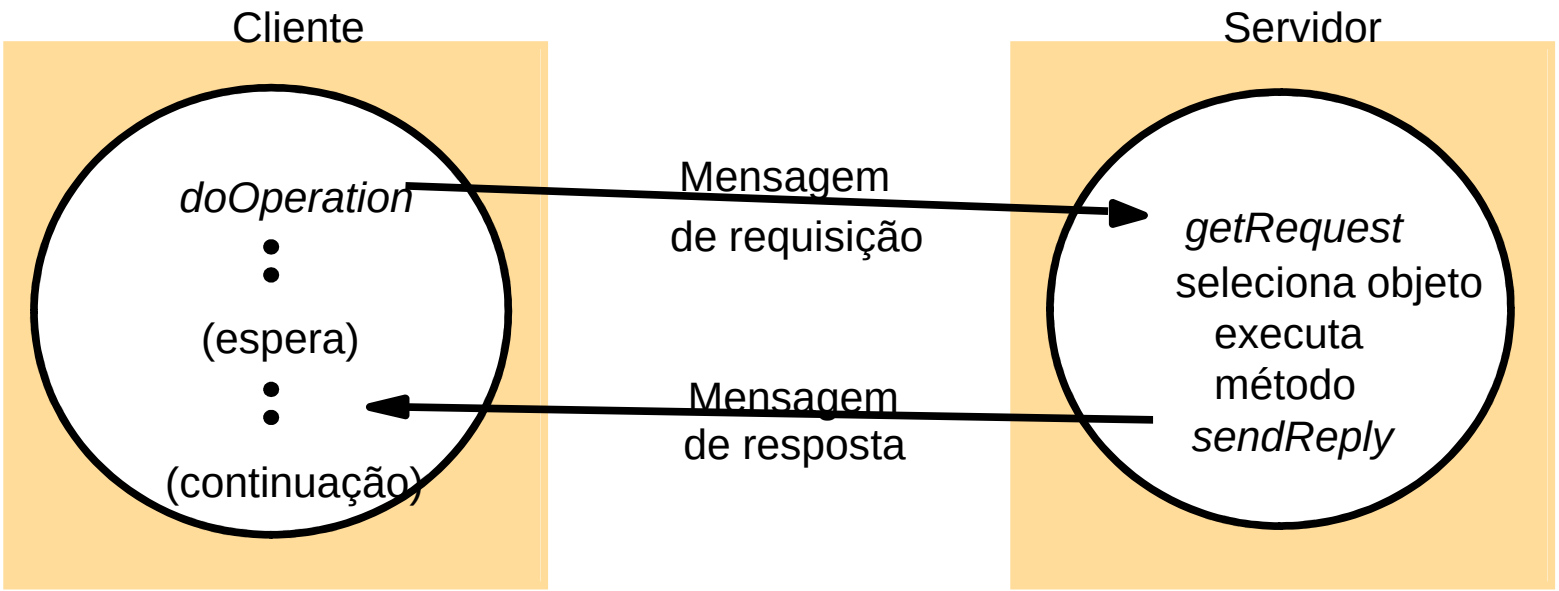


Comunicação client/server

- Comumente, a comunicação requisição e resposta é:
  - Síncrona: o processo cliente bloqueia.
  - Confiável: a resposta do servidor serve de confirmação.
- Alternativamente pode ser assíncrona por conta do cliente.
  - Utilização de UDP no lugar de TCP
    - Evita confirmações redundantes.
    - Estabelecimento de conexão cria mensagens extra.
    - Controle de fluxo não é necessário para pequenas invocações e respostas.

Comunicação client/server

- O protocolo requisição-resposta
  - Baseado em 3 primitivas de comunicação
    - *doOperation, getRequest* e *sendReply*.



Comunicação client/server

- O protocolo requisição-resposta

public byte[ ] doOperation (RemoteObjectRef o, int methodId, byte[ ] args);

Envia uma mensagem de requisição para um objeto remoto e retorna uma resposta RMI. Os argumentos especificam o objeto remoto (REE), o método a ser invocado e os argumentos desse método. Utiliza *receive.*

public byte[ ] getRequest ();

Lê uma requisição de cliente por meio da porta de servidor.

public void sendReply (byte[ ] reply, InetAddress clientHost, int clientPort);

Envia a mensagem de reposta para o cliente em seu endereço IP e porta, respectivos.

Comunicação client/server

- O protocolo requisição-resposta
  - Informações transmitidas em uma mensagem

Tipo da mensagem	*Int (0=Request, 1= Reply)*
Identificador Requisição (RequestId)	*int*
Referência do Objeto	*RemoteObjectRef*
Identificador de método (methodId)	*int or Method*
Argumentos	*array of bytes*

Comunicação client/server

- Modelos de falhas do protocolo requisição-resposta
  - Implementação com UDP: Falhas por omissão, entrega e ordenação não garantida.
    - *Timeouts*
      - Retorno imediato de *doOperation*.
      - Envio de requisições repetidamente até receber uma resposta.
    - Descarte de mensagens de requisição duplicadas
      - Filtro baseado no identificador de requisição
    - Perda de mensagens de resposta
      - Repetição da operação.
      - Operações idempotente.

Comunicação client/server

- Modelos de falhas do protocolo requisição-resposta
  - Implementação com UDP: Falhas por omissão, entrega e ordenação não garantida.
    - Histórico
      - Estrutura que contém um identificador de requisição.
      - Consumo de memória.
        - Descarte por confirmação da resposta.
        - Descarte por tempo.

Comunicação client/server

- Modelos de falhas do protocolo requisição-resposta
  - Protocolos RPC
    - Protocolo *request* (R) → implementado sobre datagramas UDP.
    - Protocolo *request-reply* (RR) → baseado no protocolo requisição-resposta.
    - Protocolo *request-reply-acknowledge reply* (RRA) → ack do *requestID*.

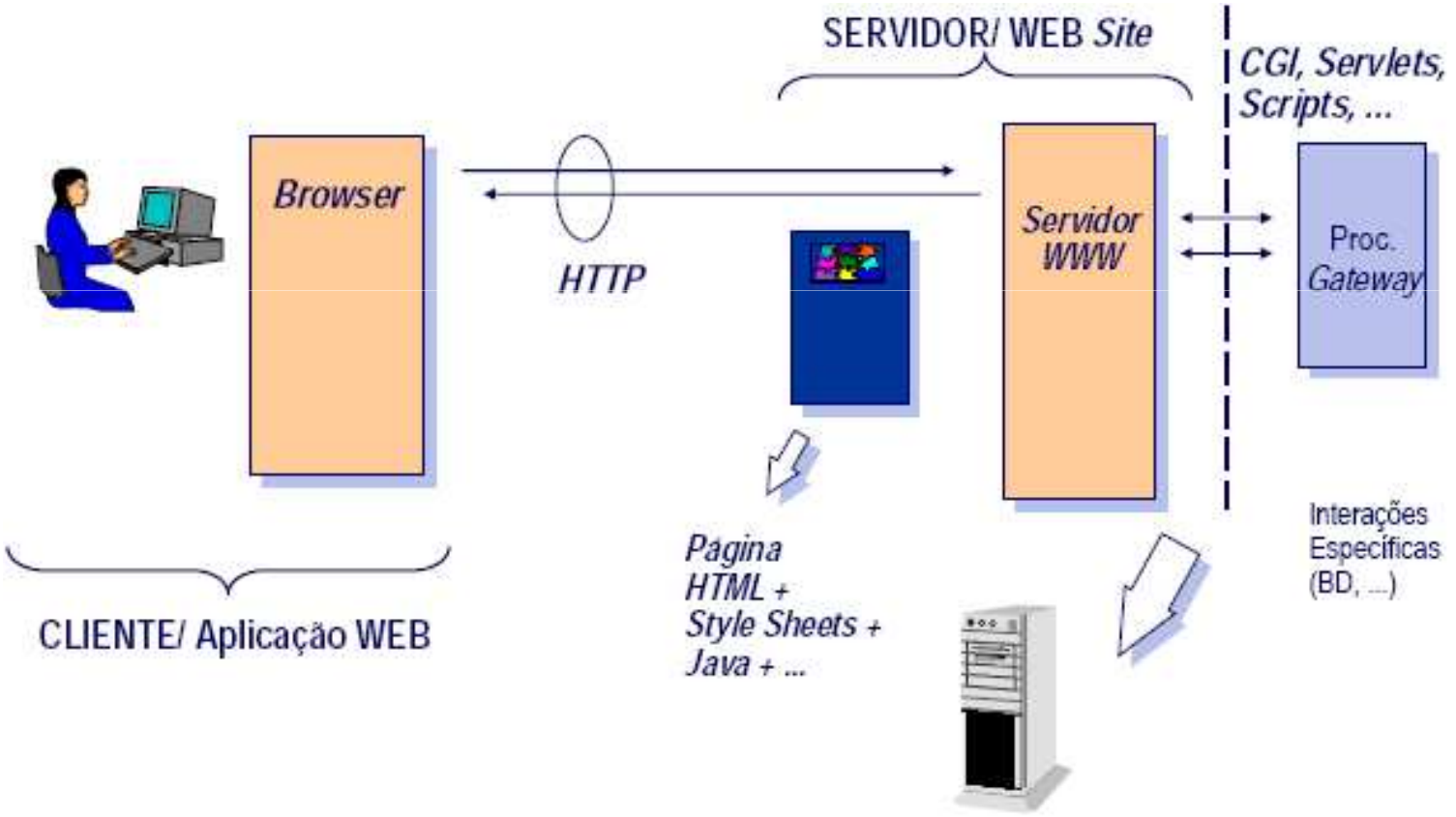
Nome	*Mensagens enviadas para*		
	Cliente	*Servidor*	*Cliente*
R	*Request*		
R R	*Request*	*Reply*	
R R A	*Request*	*Reply*	*Acknowledge reply*

Comunicação client/server

- Protocolo requisição-resposta implementado com TCP
  - Limite de 8KB não adequado em sistemas transparentes RMI.
  - Simplificação na construção do protocolo requisição-resposta.
  - Vantagens
    - Evita a implementar protocolo que quebra a mensagem em *p* pacotes.
    - Evita tratar da retransmissão de mensagens. (confiabilidade)
    - Evita implementar filtros de duplicatas ou de tratamento de históricos.
    - Evita tratar de mecanismos de controle de fluxo.
  - Desvantagens
    - Sobrecarga de conexão (requisição-resposta não estejam no mesmo fluxo)
    - Aplicativo não exige todos os recursos do TCP.

Comunicação client/server

- HTTP – *Hypertext Transfer Protocol*

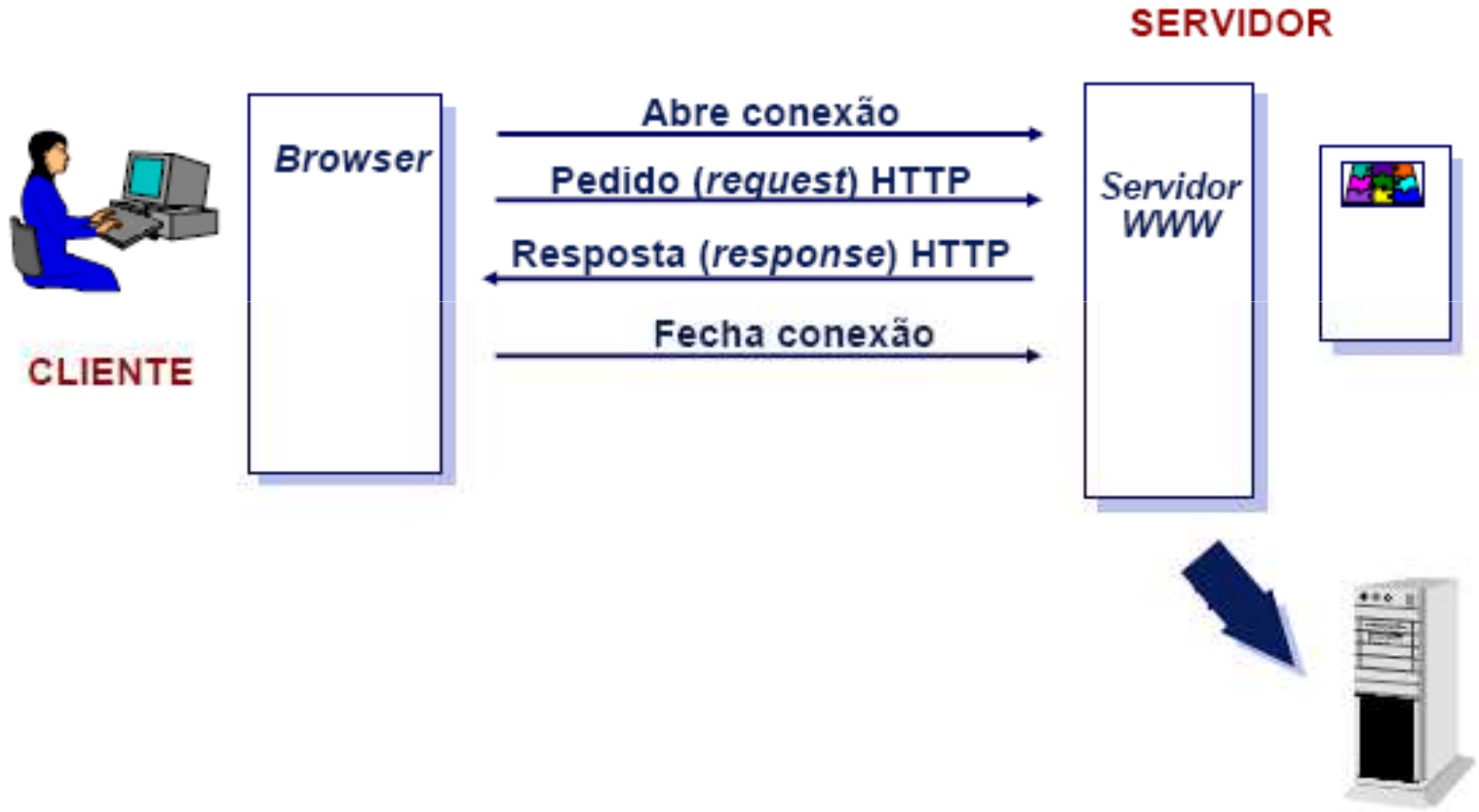


Comunicação client/server

- HTTP – *Hypertext Transfer Protocol* (v1 RFC 2068)
  - Requisição especifica URL (*host* DNS, porta e ID de recurso)
  - O protocolo especifica as mensagens, métodos, resultados e regras para representá-los (empacotamento).
  - Alguns métodos: GET, PUT, POST, DELETE, etc.
  - Permite negociação de:
    - Conteúdo: representação de dados, mídia, linguagem, etc.
    - Autenticação: Credenciais e desafios.
  - Etapas:
    - Solicitação de conexão
    - Mensagem de requisição e mensagem de resposta.
    - Encerramento da conexão

Comunicação client/server

- HTTP – *Hypertext Transfer Protocol*



Comunicação client/server

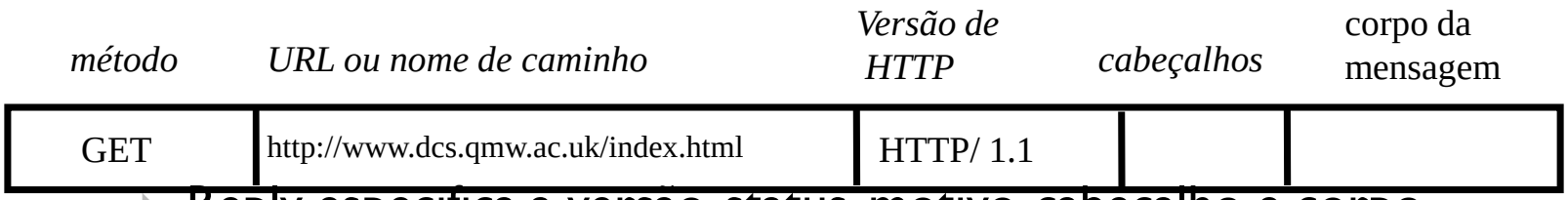
- HTTP – *Hypertext Transfer Protocol*
  - Estabelecer e fechar a conexão ocasiona sobrecarga.
  - HTTP v1.1 usa conexões persistentes.
  - Servidor encerra conexão ociosa, podendo avisar ao cliente.
  - Cliente faz novas requisições para operações idempotentes.
  - Requisições e respostas são empacotados em *strings* ASCII, e recursos podem ser em *byte* compactados.
  - Argumentos e resultados utilizam estruturas do tipo *Multipurpose Internet Mail Extensions* – MIME.
    - *text/plain, text/html, image/gif, image/jpeg,* etc.

Comunicação client/server

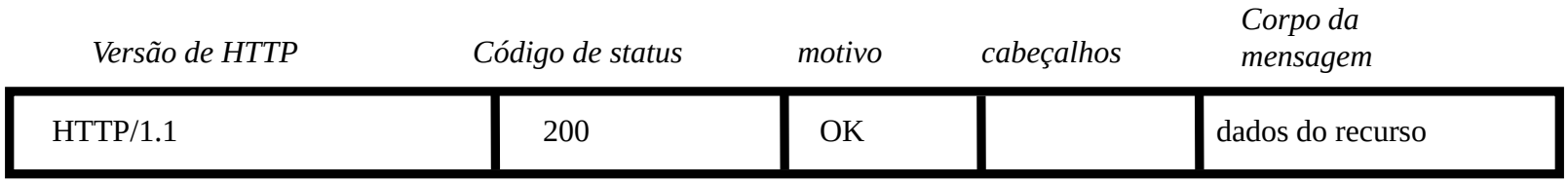
- HTTP – *Hypertext Transfer Protocol*
  - Métodos:
    - Cada requisição especifica método e um URL, e a resposta o *status*. Podem conter também dados.
    - GET: obtém dados diretamente, dados depois de executados, partes de dados, envia argumentos a formulários, etc, na URL.
    - HEAD: idêntica ao GET, porém não retorna dados.
    - POST: especifica no URL um recurso que pode tratar dos dados enviados.
    - PUT: armazena dados de um URL, como modificação ou novo recurso.
    - DELETE: exclui o recurso identificado pelo URL dado.
    - OPTIONS: lista os métodos do servidor e seus requisitos.
    - TRACE: o servidor envia de volta a mensagem de requisição.

Comunicação client/server

- HTTP – *Hypertext Transfer Protocol*
  - Conteúdo da mensagem
    - *Request* especifica método, URL, versão, área de cabeçalho e corpo.

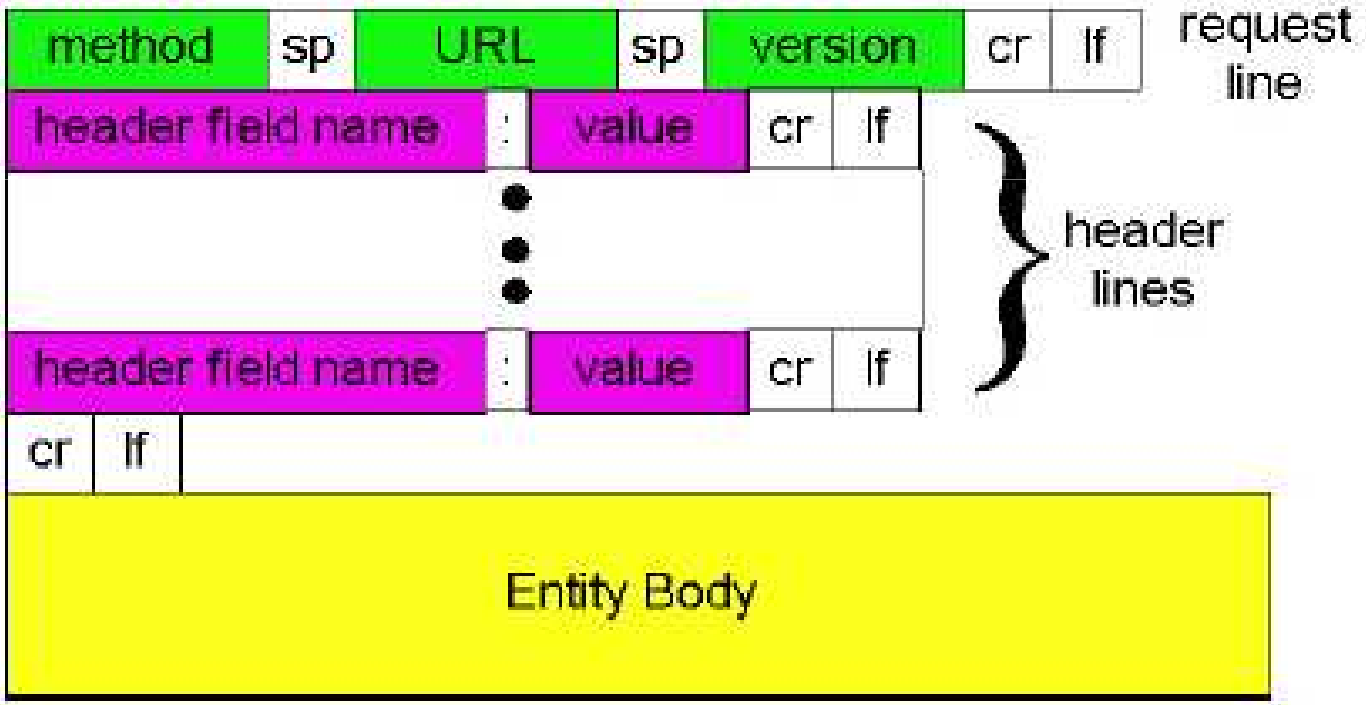


    - Reply especifica a versão, status, motivo, cabeçalho e corpo.



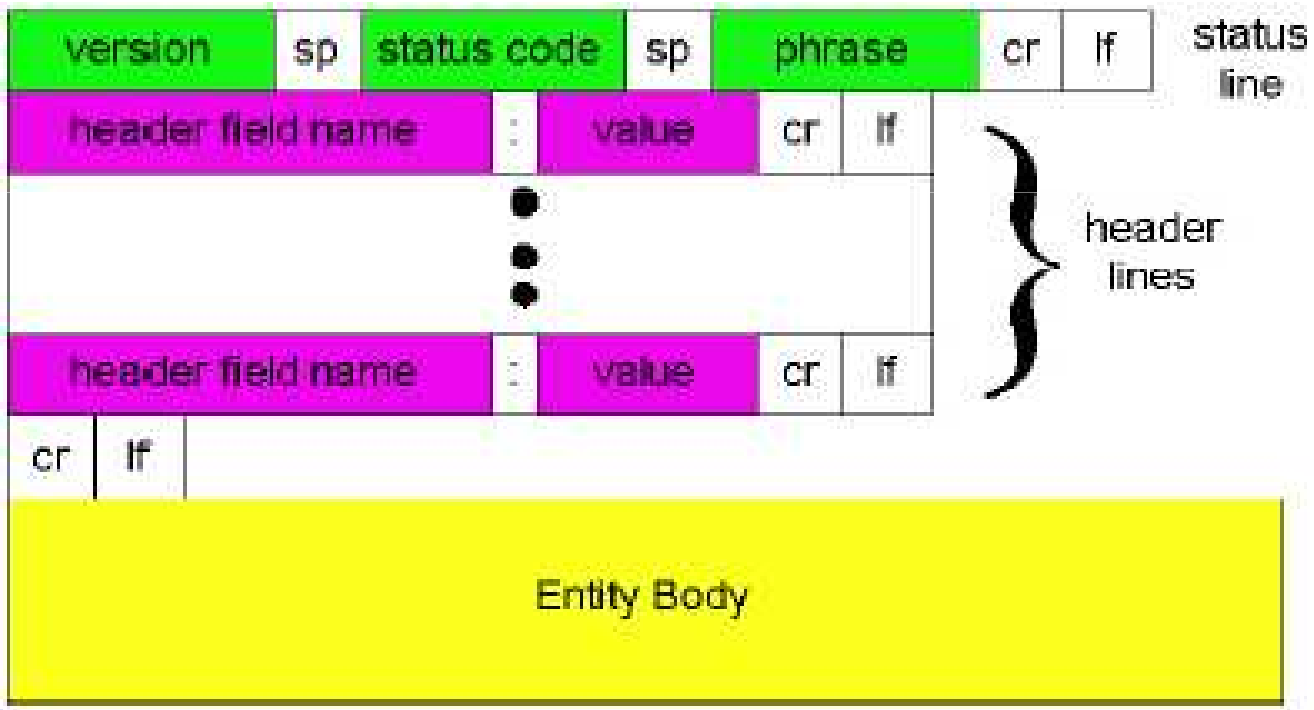
Comunicação client/server

- HTTP – *Hypertext Transfer Protocol*
  - Formato de mensagem do *Request* HTTP



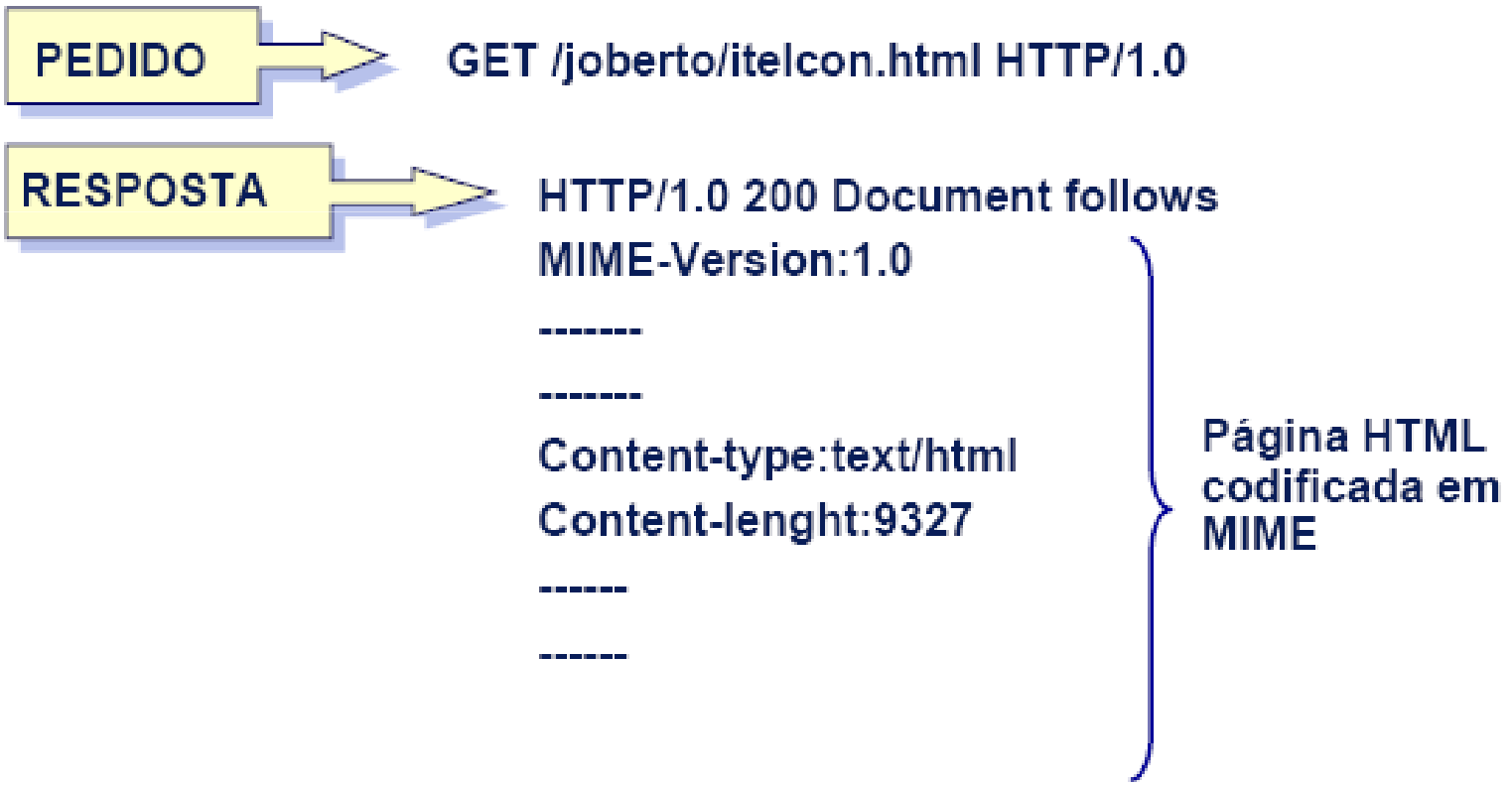
Comunicação client/server

- HTTP – *Hypertext Transfer Protocol*
  - Formato de mensagem do *Reply* HTTP



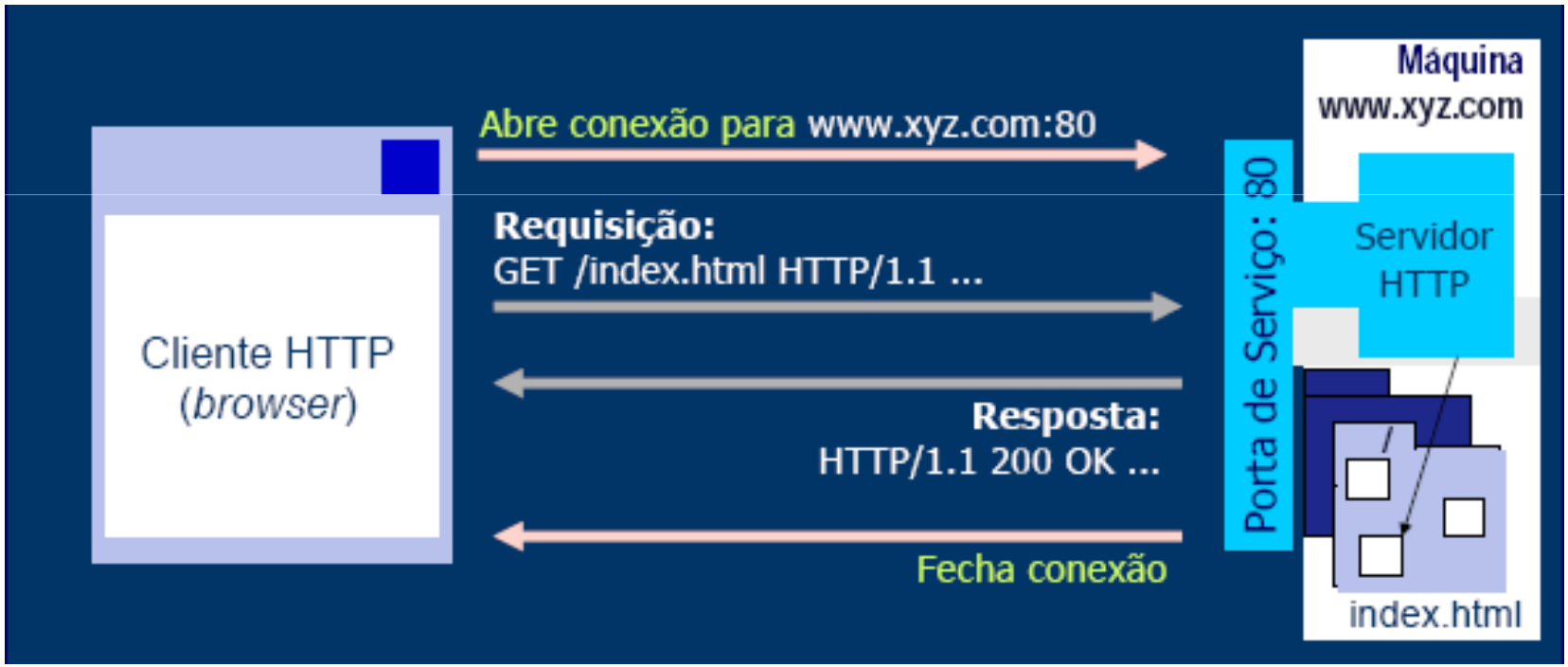
Comunicação client/server

- HTTP – *Hypertext Transfer Protocol*
  - Exemplo



Comunicação client/server

- HTTP – *Hypertext Transfer Protocol*
  - Exemplo



Comunicação client/server

- HTTP – *Hypertext Transfer Protocol*
  - Exemplo

HTTP Request
GET /index.html HTTP/1.1
Host: www.exemplo.com

HTTP Reply
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
Dados dados dados dados

Comunicação em grupo

- Comunicação de um processo com um grupo de outros processos em computadores diferentes.
- Emprego de *multicast* (difusão seletiva)
- Características das mensagens *multicast*:
  - Tolerância à falha baseada em serviços replicados.
  - Localização dos servidores de descoberta em redes espontâneas.
  - Melhor desempenho devido a replicação de dados.
  - Propagação de notificações de evento.

C. Grupo – *Multicast* IP

- Único datagrama IP para um grupo *multicast*.
- Um grupo é especificado pela classe D (1110.XXXX)
- A participação como membro do grupo é dinâmica
- É possível enviar mensagem para um grupo sem ser membro.
- *Multicast* IP apenas disponível por datagramas UDP.
- União do soquete da aplicação com o grupo (Aplicação)
- Um ou mais processadores tem soquete no grupo (Rede)
- Cópias são enviadas a todos os processos locais.

C. Grupo – *Multicast* IP

- Detalhamento:
  - Roteadores *multicast*
    - Uso de TTL limita a distância de propagação do datagrama.
  - Alocação de endereço multicast
    - 224.0.0.0 – 239.0.0.0 (1110.0000 – 1110.1111)
    - Podem ser permanentes ou temporários
      - Permanentes: 224.0.0.1 a 224.0.0.255
      - 244.0.0.1 se refere ao grupo de todos os hosts com suporte *multicast*.
  - O programa *session directory* (sd) inicia ou une uma sessão.

C. Grupo – *Multicast* IP

- Modelos de falhas para datagramas *multicast*
  - Mesmas característica dos datagramas UDP.
  - *Multicast* não confiável.
- API Java para *multicast* IP
  - Classe *MulticastSocket*, subclasse de *DatagramSocket*.
    - Dois construtores (com e sem porta).
  - Métodos de *MulticastSocket*
    - *joinGroup*() → Une a um grupo *multicast*.
    - *leaveGroup*() → Sai de um grupo *multicast*.
    - *setTimeToLive*() → Configura um TTL para um soquete. (padrão=1)

C. Grupo – *Multicast* IP

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[ ]){
         // args fornece o conteúdo da mensagem e o grupo multicast de destino
        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
```

C. Grupo – *Multicast* IP

```
// obtém mensagens de outros participantes do grupo
            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
C. Grupo – e Ordenamento

- Falhas por omissão,falha no roteador *multicast*, etc.
- Efeitos da confiabilidade e do ordenamento:
  - Tolerância a falhas baseadas em serviços replicados:
    - Serviços que exigem que todos os membros recebam as mensagens de requisição na mesma ordem dos outros.
  - Localização dos servidores de descoberta em redes espontâneas.
    - Necessárias requisições periódicas desde o início do sistema.
  - Melhor desempenho devido a replicação de dados.
    - Dados enviados podem estar fora de ordem. Ex. newsgroups é tolerânte.
  - Propagação de notificações de evento.
    - O aplicativo define a qualidade exigida de *multicast*.