

Redes de Computadores I

Ronaldo T. Oikawa

Camada de Aplicação

Camada de Aplicação

Nossos objetivos:

- conceitual, aspectos de implementação de protocolos de aplicação para redes
 - paradigma cliente-servidor
 - modelos de serviço
- aprenda sobre protocolos examinando algumas aplicações populares

Outros objetivos do capítulo

- protocolos específicos:
 - http
 - ftp
 - smtp
 - pop
 - dns
- programação de aplicações de rede
 - socket API

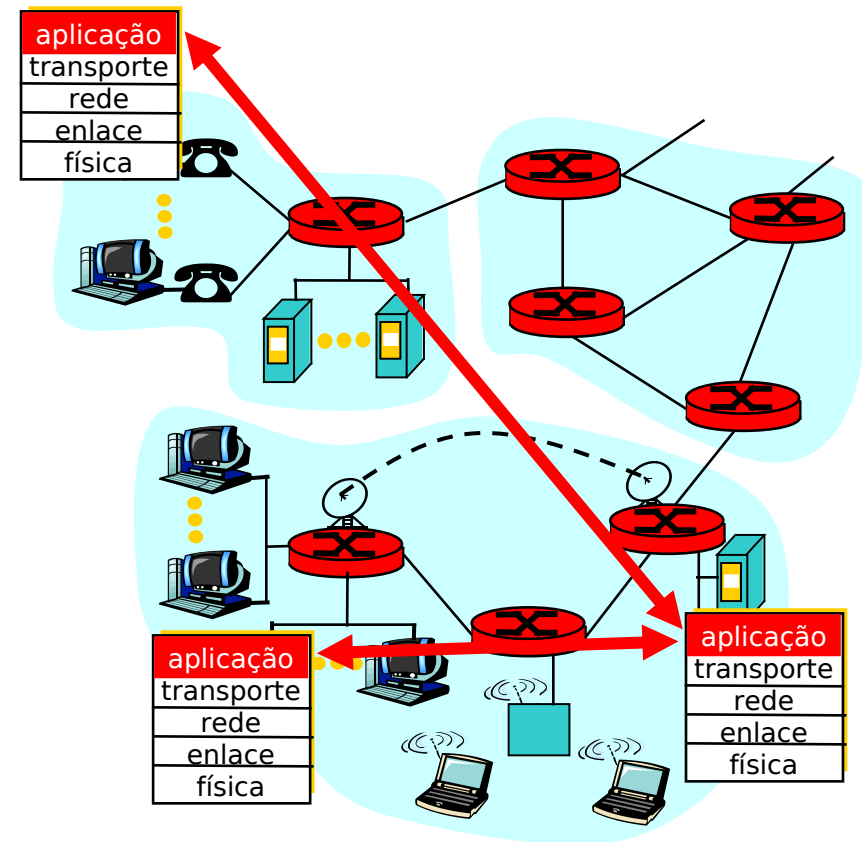
Aplicações e Protocolo de Aplicação

Aplicações de rede: processos distribuídos em comunicação

- rodam nos computadores usuários da rede como programas de usuário
- trocam mensagens para realização da aplicação
- e.x., email, ftp, Web

Protocolos de aplicação

- fazem parte das aplicações
- HTTP, SMTP, POP3, ...
- definem mensagens trocadas e as ações tomadas
- usam serviços de comunicação das camadas inferiores



Aplicações de Rede

Processo: programa executando num host.

- dentro do mesmo host: **interprocess communication** (definido pelo OS).
- processos executando em diferentes hosts se comunicam com um **protocolo da camada de aplicação**

- **agente usuário:** software que interfaceia com o usuário de um lado e com a rede de outro.
 - implementa protocolo da camada de aplicação
 - Web: browser
 - Netscape, IE,
 - E-mail: leitor de correio
 - MSOutlook, ...
 - streaming audio/video: media player

Paradigma Cliente-Servidor

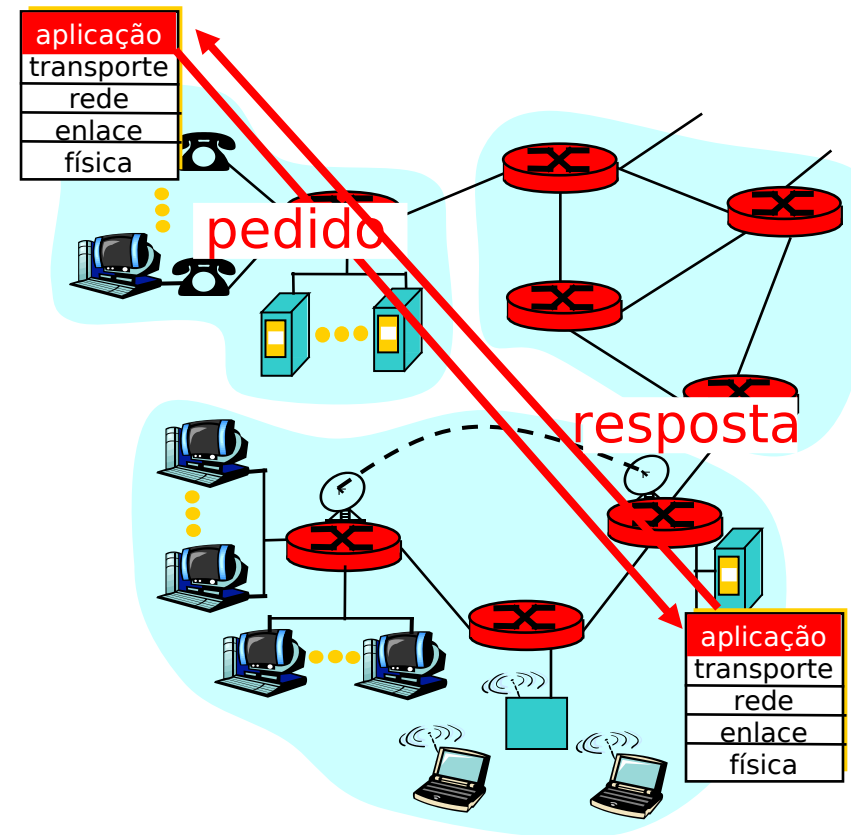
Aplicações de rede típicas têm duas partes: *cliente* and *servidor*

Cliente:

- inicia comunicação com o servidor (“fala primeiro”)
- tipicamente solicita serviços do servidor,
- Web: cliente implementado no browser; e-mail: leitor de correio

Servidor:

- fornece os serviços solicitados ao cliente
- e.x., Web server envia a página Web solicitada, servidor de e-mail envia as mensagens, etc.



Interfaces de Programação

API: application programming interface

- define a interface entre a camada de aplicação e de transporte
- socket: Internet API
 - dois processos se comunicam enviando dados para o socket e lendo dados de dentro do socket

Q: Como um processo “identifica” o outro processo com o qual ele quer se comunicar?

- **IP address** do computador no qual o processo remoto executa
- “**port number**” - permite ao computador receptor determinar o processo local para o qual a mensagem deve ser entregue.

Serviços de Transporte

Perda de dados

- algumas aplicações (e.x., áudio) podem tolerar alguma perda
- outras aplicações (e.x., transferência de arquivos, telnet) exigem transferência de dados 100% confiável

Temporização

- algumas aplicações (e.x., telefonia Internet, jogos interativos) exigem baixos atrasos para operarem
 - Aplicações interativas em tempo real: telefonia na Internet, ambientes virtuais, teleconferencia, jogos multiusuário, ...

Largura de Banda

- algumas aplicações (e.x., multimedia) exigem uma banda mínima para serem utilizáveis
 - Voz: 32Kbps
- outras aplicações (“aplicações elásticas”) melhoram quando a banda disponível aumenta
 - Coreio eletronico, transferencia de aquivos, acesso remoto e transferencias Web,

Requisitos de Transporte de Aplicações Comuns

Aplicação	Perdas	Banda	Sensível ao Atraso
file transfer	sem perdas	elástica	não
e-mail	sem perdas	elástica	não
Web documents	tolerante	elástica	não
real-time audio/video	tolerante	aúdio: 5Kb-1Mb vídeo:10Kb-5Mb	sim, 100's msec
stored audio/video	tolerante	igual à anterior	sim, segundos
jogos interativos	tolerante	Kbps	sim, 100's msec
e-business	sem perda	elástica	sim

Serviços de Transporte da Internet

serviço TCP:

- *orientado á conexão:* conexão requerida entre cliente e servidor
- *transporte confiável* dados perdidos na transmissão são recuperados
- *controle de fluxo:* compatibilização de velocidade entre o transmissor e o receptor
- *controle de congestionamento :* protege a rede do excesso de tráfego
- *não oferece:* garantias de temporização; de banda mínima; taxa de transmissão mínima;

serviço UDP:

- transferência de dados não confiável entre os processos transmissor e receptor
- *não oferece:* estabelecimento de conexão, confiabilidade, controle de fluxo e de congestionamento, garantia de temporização e de banda mínima.

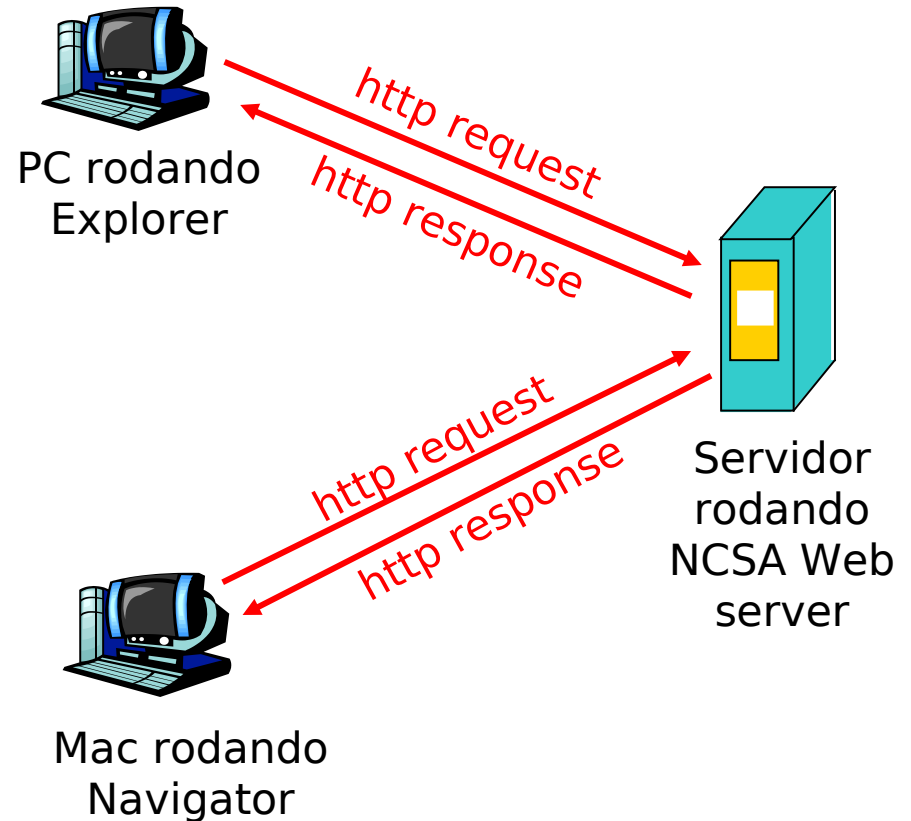
Aplicações e Protocolos de Transporte da Internet

	Aplicação	Protocolo de Aplicação	Protocolo de Transporte
	e-mail	smtp [RFC 821]	TCP
acesso de	terminais remotos	telnet [RFC 854]	TCP
	Web	http [RFC 2068]	TCP
	transferência de arquivos	ftp [RFC 959]	TCP
	streaming multimedia	RTP ou proprietário (e.g. RealNetworks)	TCP ou UDP
servidor de	arquivos remoto	NSF	TCP ou UDP
	telefonia Internet	RTP ou proprietary (e.g., Vocaltec)	tipicamente UDP

Protocolo HTTP

http: hypertext transfer protocol

- protocolo da camada de aplicação da Web
- modelo cliente/servidor
 - *cliente*: browser que solicita, recebe e apresenta objetos da Web
 - *server*: envia objetos em resposta a pedidos
- http1.0: RFC 1945
- http1.1: RFC 2068



Protocolo HTTP

http: protocolo de transporte TCP:

- cliente inicia conexão TCP (cria socket) para o servidor na porta 80
- servidor aceita uma conexão TCP do cliente
- mensagens http (mensagens do protocolo de camada de aplicação) são trocadas entre o browser (cliente http) e o servidor Web (servidor http)
- A conexão TCP é fechada

http é “stateless” Sem Estado

- o servidor não mantém informação sobre os pedidos passados pelos clientes;
- Reenvia novamente sempre que solicitado;

Protocolos que mantêm informações de estado são complexos!

- necessidade de organizar informações passadas
- se ocorrer um crash as informações podem ser perdidas ou gerar inconsistências entre o cliente e o servidor

Exemplo de Operação

Usuário entra com a URL: `www.someSchool.edu/someDepartment/home.index` (contém referência a 10 imagens jpeg)

1a. cliente http inicia conexão TCP ao servidor http (processo) em `www.someSchool.edu`. Porta 80 é a default para o servidor http .

1b. servidor http no host `www.someSchool.edu` esperando pela conexão TCP na porta 80. “aceita” conexão, notificando o cliente

2. cliente http client envia http *request message* (contendo a URL) para o socket da conexão TCP

3. servidor http recebe mensagem de pedido, forma *response message* contendo o objeto solicitado (`someDepartment/home.index`), envia mensagem para o socket

tempo



```
graph TD; 1a[1a. cliente http inicia conexão TCP ao servidor http (processo) em www.someSchool.edu. Porta 80 é a default para o servidor http .] --> 1b[1b. servidor http no host www.someSchool.edu esperando pela conexão TCP na porta 80. "aceita" conexão, notificando o cliente]; 1b --> 2[2. cliente http client envia http request message (contendo a URL) para o socket da conexão TCP]; 2 --> 3[3. servidor http recebe mensagem de pedido, forma response message contendo o objeto solicitado (someDepartment/home.index), envia mensagem para o socket];
```

Exemplo (cont.)

4. servidor http fecha conexão TCP.

5. cliente http recebe mensagem de resposta contendo o arquivo html, apresenta o conteúdo html. Analisando o arquivo html encontra 10 objetos jpeg referenciados

6. Passos 1-5 são repetidos para cada um dos 10 objetos jpeg.

tempo

Conexões persistentes e não-persistentes

Não-persistente

- http/1.0: servidor analisa pedido, envia resposta e fecha a conexão TCP
- 2 RTTs para obter um objeto
 - Conexão TCP
 - solicitação e transferência/recebimento do objeto
- cada transferência sofre por causa do mecanismo de **slow-start** do TCP
 - Transferencia lenta no inicio da transmissão
- muitos browser abrem várias conexões paralelas
- **RTT** – round-tripe time: tempo de viagem de ida e volta

Persistente

- modo default para http/1.1
- na mesma conexão TCP são trazidos vários objetos
- o cliente envia pedido para todos os objetos referenciados tão logo ele recebe a página HTML básica .
- poucos RTTs, menos slow start.

Formato das Mensagens HTTP

- dois tipos de mensagens HTTP: *request*, *response*
- http request message:
 - ASCII (formato legível para humanos)

linha de pedido/
Requisição
(comandos GET
, POST, HEAD)

linhas de
cabeçalho

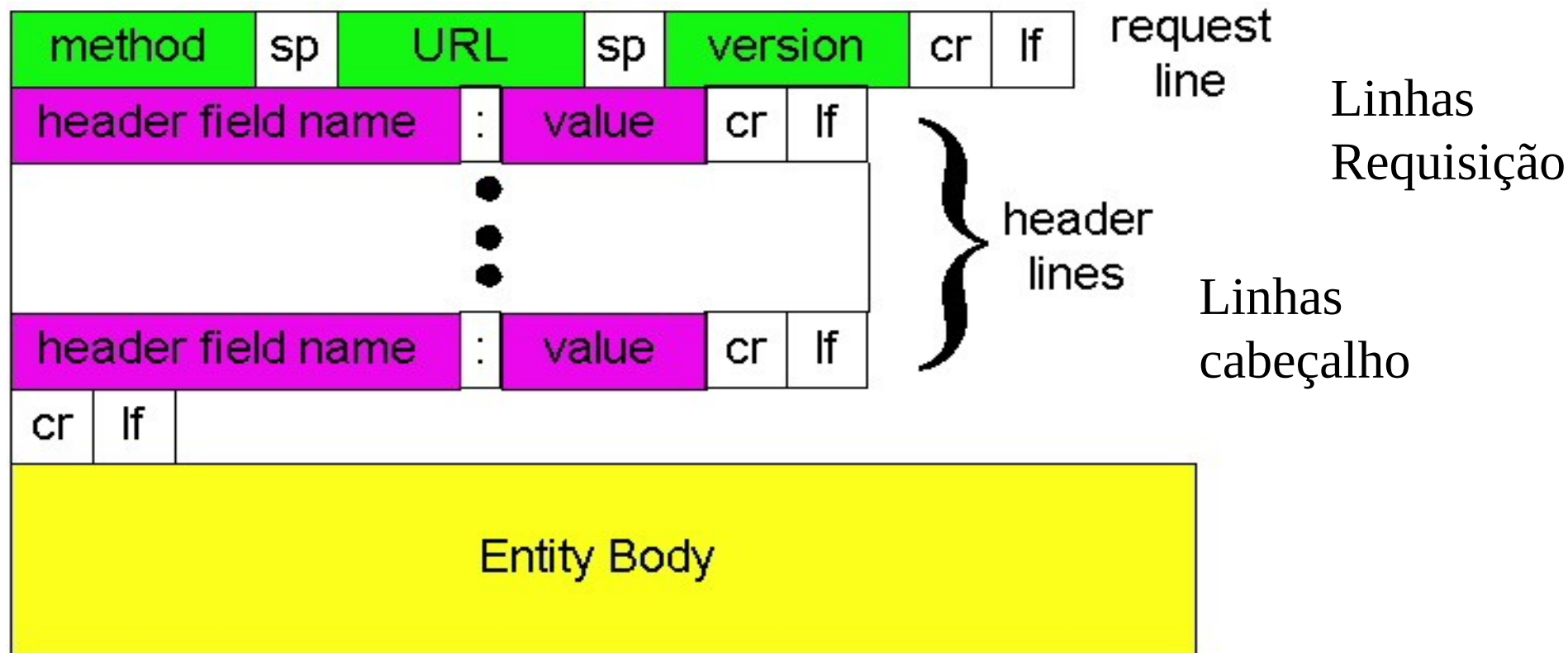
```
GET /somedir/page.html HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Accept-language: fr
```

Carriage return,
line feed

(extra carriage return, line feed)

indica fim da mensagem

HTTP request: formato geral



Linhas de requisição: Metodo: *GET, POST e HEAD

Linhas cabeçalho: connect, close, User-Agent (tipo de browser),

formatos HTTP: response

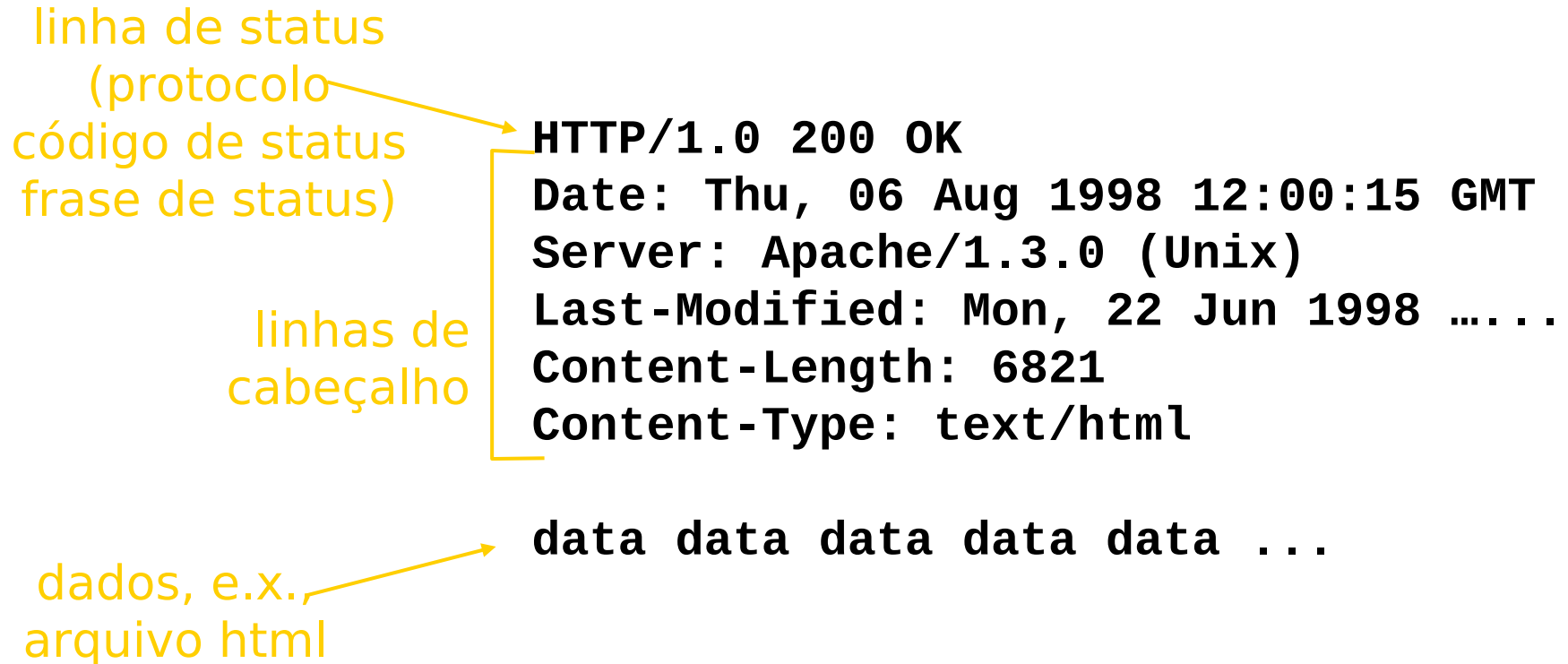
linha de status
(protocolo
código de status
frase de status)

linhas de
cabeçalho

dados, e.x.
arquivo html

```
HTTP/1.0 200 OK
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 .....
Content-Length: 6821
Content-Type: text/html

data data data data data ...
```

The diagram illustrates the structure of an HTTP response. It consists of three main parts: a status line, a header section, and a body. The status line is labeled 'linha de status (protocolo código de status frase de status)' and points to 'HTTP/1.0 200 OK'. The header section is labeled 'linhas de cabeçalho' and points to a block of five lines: 'Date: Thu, 06 Aug 1998 12:00:15 GMT', 'Server: Apache/1.3.0 (Unix)', 'Last-Modified: Mon, 22 Jun 1998', 'Content-Length: 6821', and 'Content-Type: text/html'. The body is labeled 'dados, e.x. arquivo html' and points to the line 'data data data data data ...'.

Códigos de status das respostas

200 OK

- request succeeded, requested object later in this message

301 Moved Permanently

- requested object moved, new location specified later in this message (Location:)

400 Bad Request

- request message not understood by server

404 Not Found

- requested document not found on this server

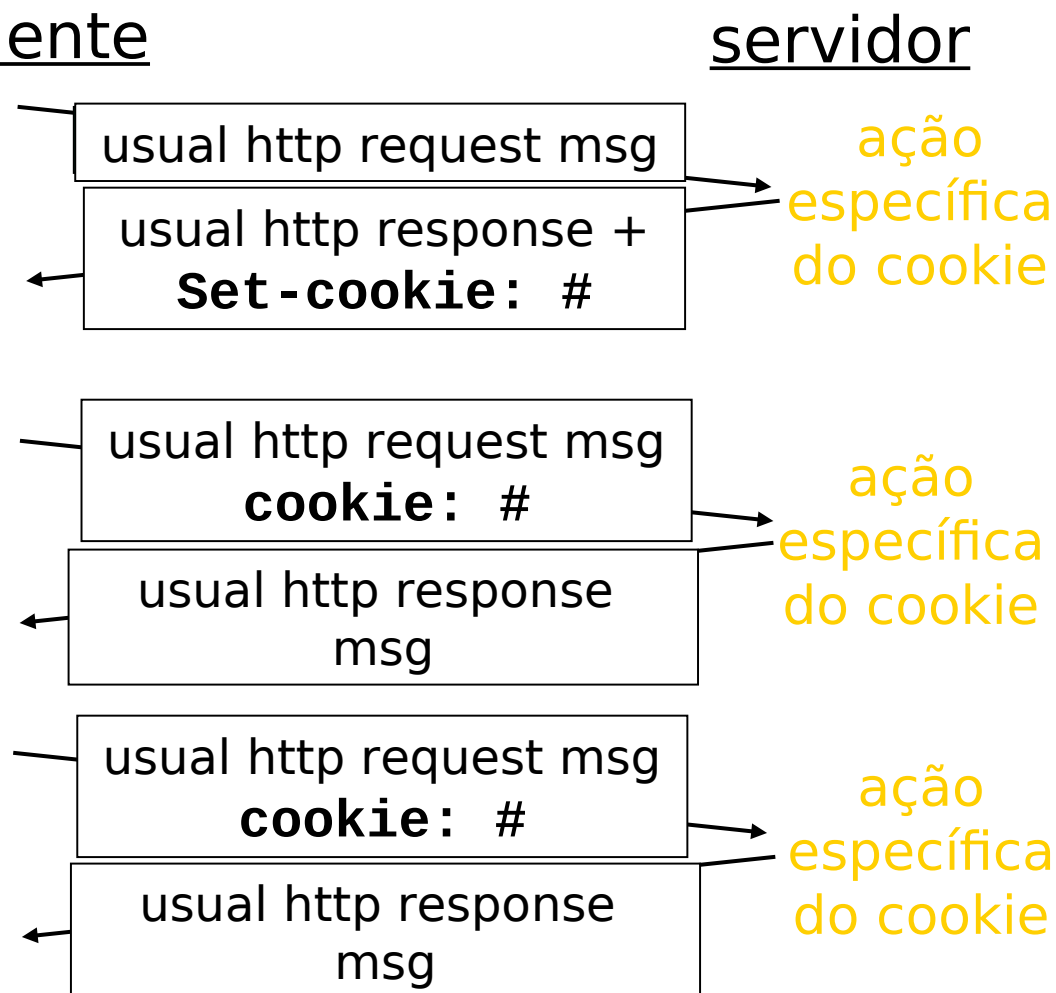
505 HTTP Version Not Supported

Cookies - RFC 2109

- **Razão:** obter informações do usuário
- gerados e lembrados pelo servidor, usados mais tarde para:
 - autenticação
 - lembrar preferências dos usuários ou prévias escolhas
- servidor envia “cookie” ao cliente na resposta HTTP

Set-cookie: 1678453
- cliente apresenta o cookie em pedidos posteriores

cookie: 1678453

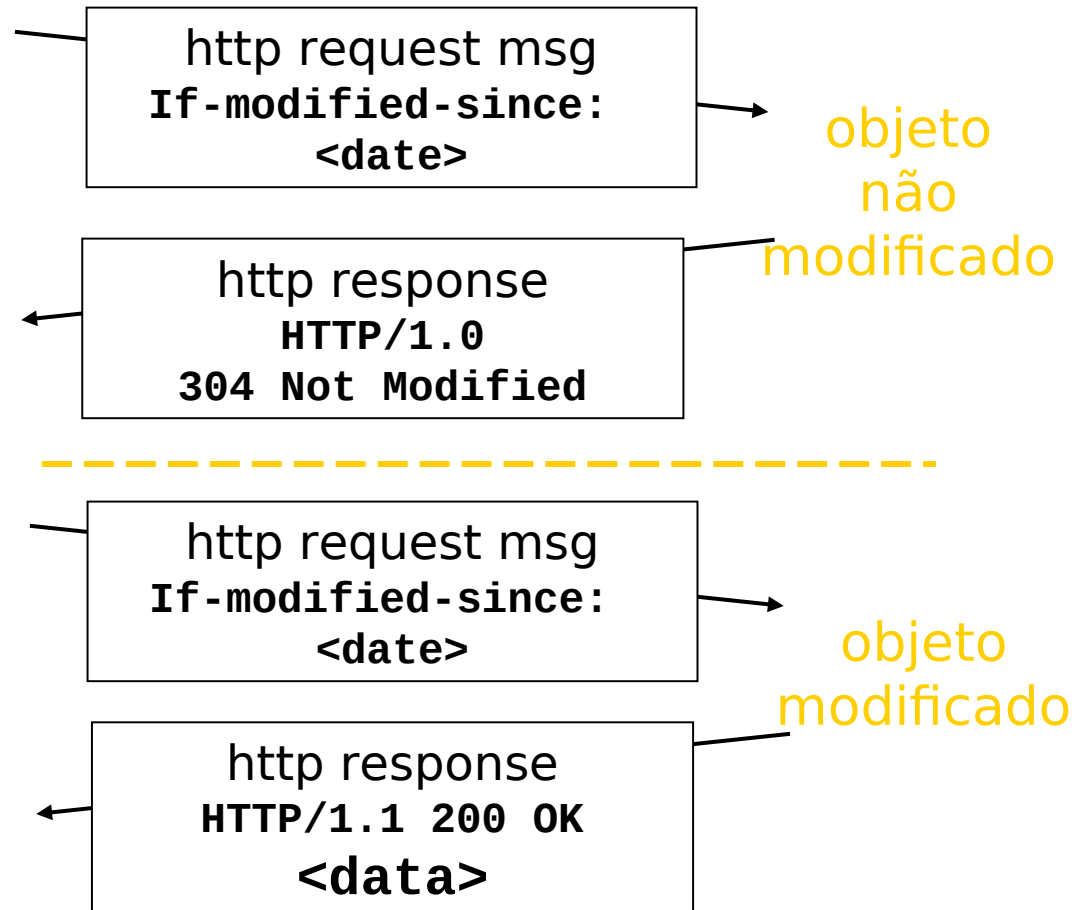


Conditional GET: armazenando no cliente

- **Razão:** não enviar objeto se a versão que o cliente já possui está atualizada.
- cliente: especifica data da versão armazenada no pedido HTTP
`If-modified-since: <date>`
- servidor: resposta não contém objeto se a cópia é atualizada:
`HTTP/1.0 304 Not Modified`

cliente

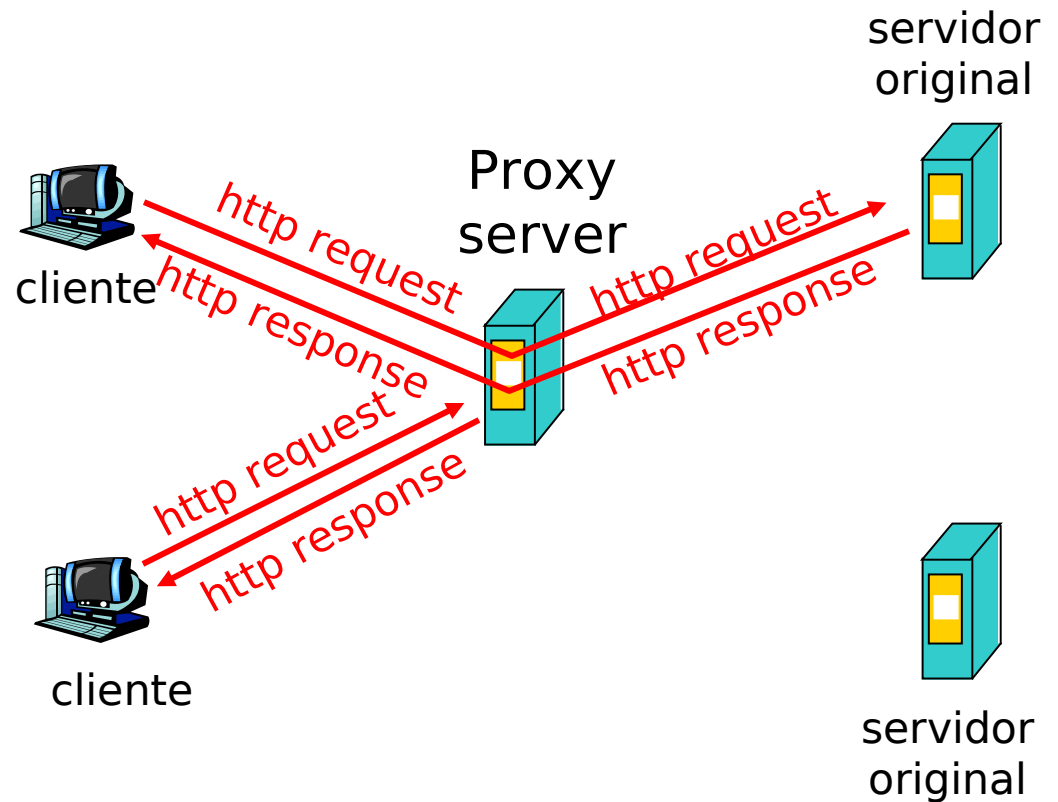
servidor



Web Caches (proxy server)

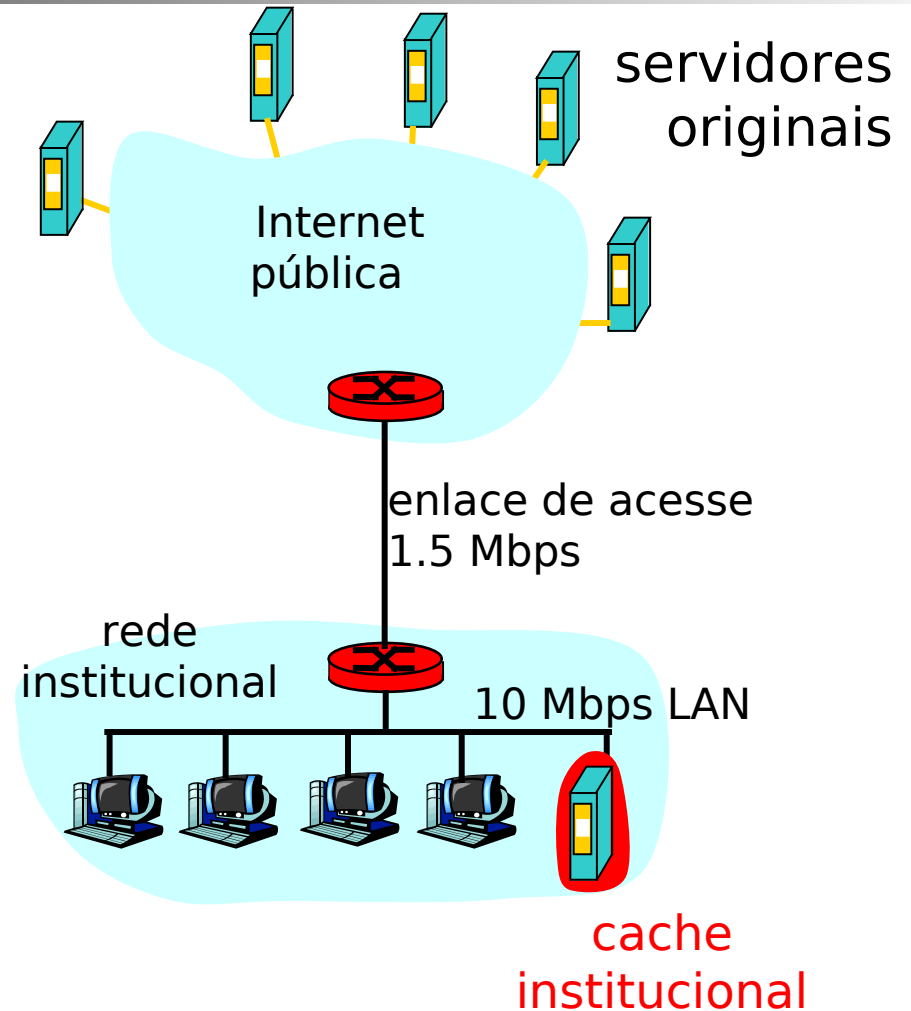
Objetivo: atender o cliente sem envolver o servidor Web originador da informação

- usuário configura o browser: acesso Web é feito através de um proxy
- cliente envia todos os pedidos http para o web cache
 - se o objecto existe no web cache: web cache retorna o objecto
 - ou o web cache solicita objecto do servidor original, então envia o objeto ao cliente.

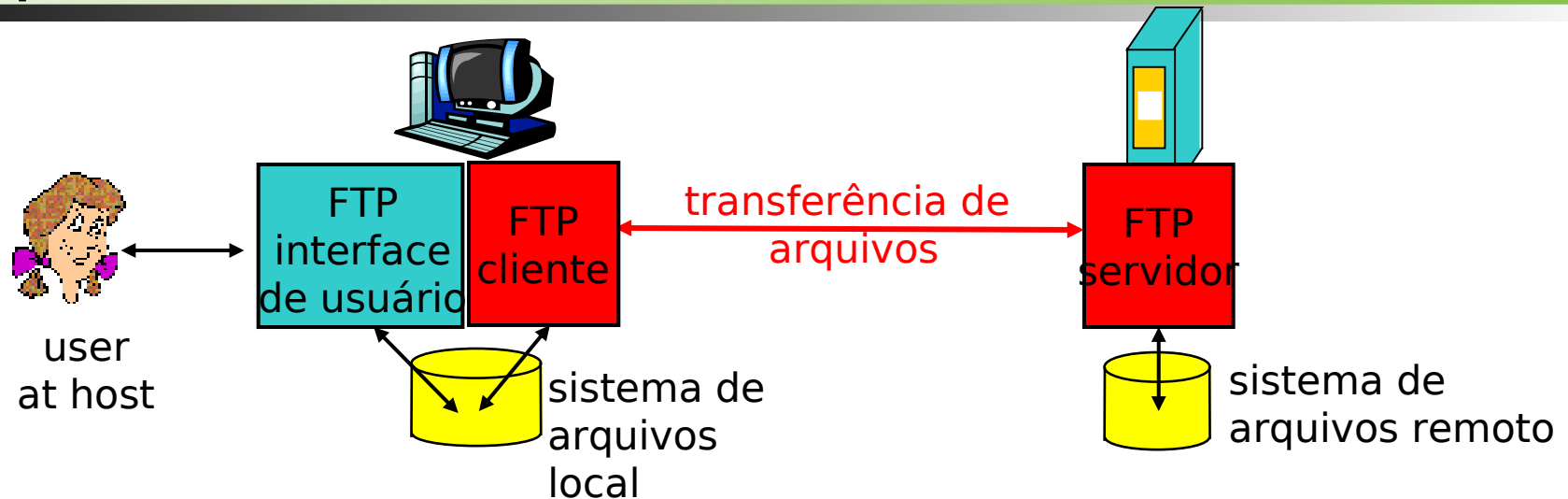


Porque Web Caching?

- armazenamento está “perto” do cliente (ex., na mesma rede)
- menor tempo de resposta
- reduz o tráfego para servidor distante
 - links externos podem ser caros e facilmente congestionáveis



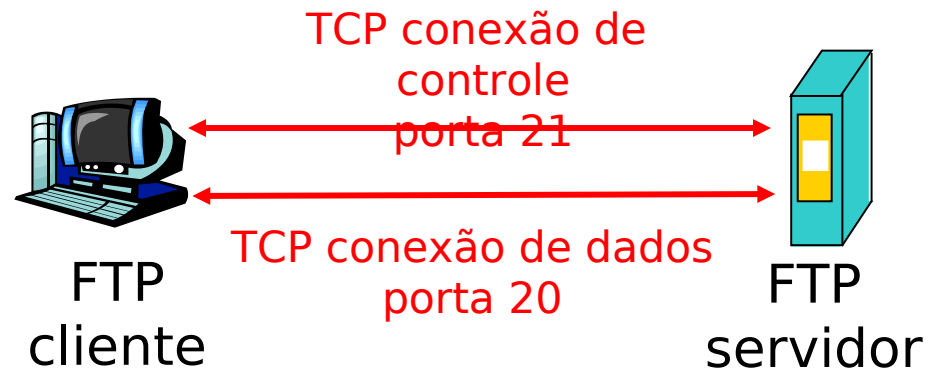
ftp: o protocolo de transferência de arquivos



- transferência de arquivos de e para o computador remoto
- modelo cliente servidor
 - *cliente*: lado que inicia a transferência (seja de ou para o lado remoto)
 - *servidor*: host remoto
- ftp: RFC 959
- ftp servidor: porta 21

ftp: controle separado, conexões de dados

- cliente ftp contata o servidor ftp na porta 21, especificando TCP como protocolo de transporte
- duas conexões TCP paralelas são abertas:
 - **controle**: troca de comandos e respostas entre cliente e servidor.
“controle out of band”
 - **dados**: dados do arquivo trocados com o servidor
- servidor ftp mantém o “estado”: diretório corrente, autenticação anterior



ftp comandos, respostas

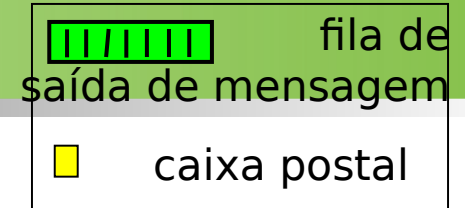
Exemplos de comandos:

- envie um texto ASCII sobre canal de controle
- **USER *username***
- **PASS *password***
- **LIST** retorna listagem do arquivo no diretório atual
- **RETR *filename*** recupera (obtém) o arquivo
- **STOR *filename*** armazena o arquivo no host remoto

Exemplos de códigos de retorno

- código de status e frase (como no http)
- **331 Username OK, password required**
- **125 data connection already open; transfer starting**
- **425 Can't open data connection**
- **452 Error writing file**

Correio Eletrônico

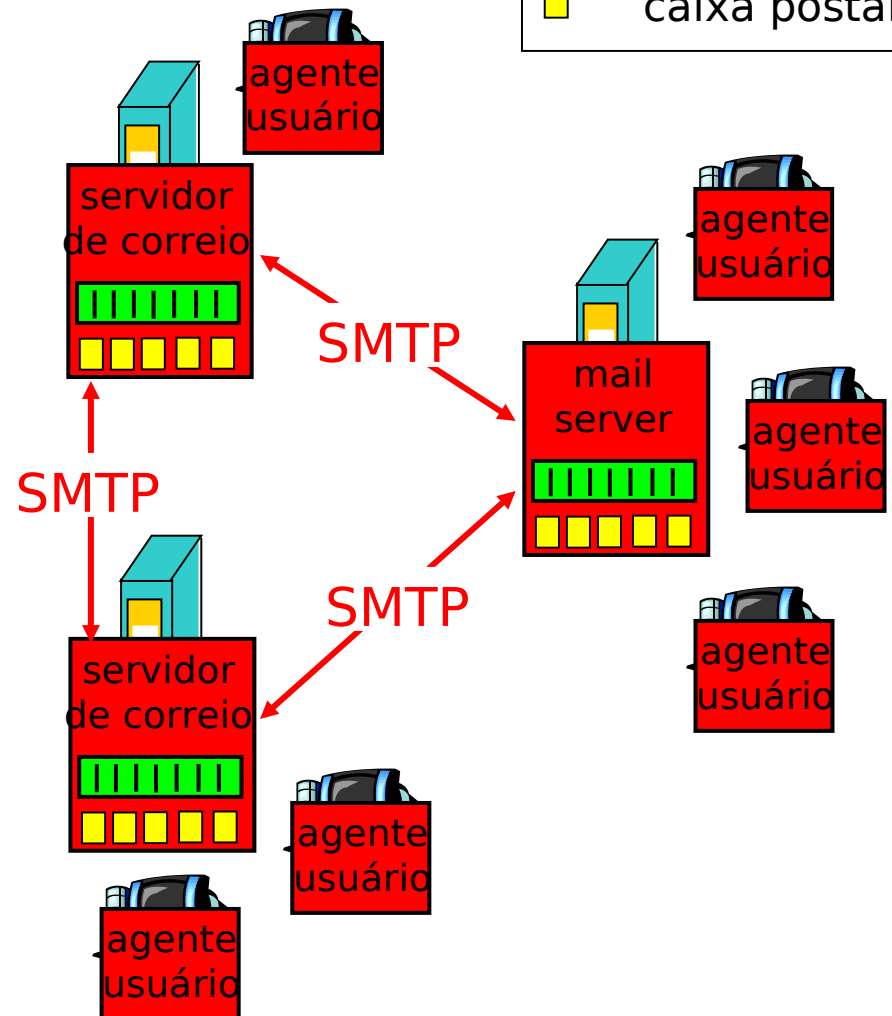


Três componentes principais:

- agentes de usuário
- servidores de correio
- simple mail transfer protocol: smtp

Agente de usuário

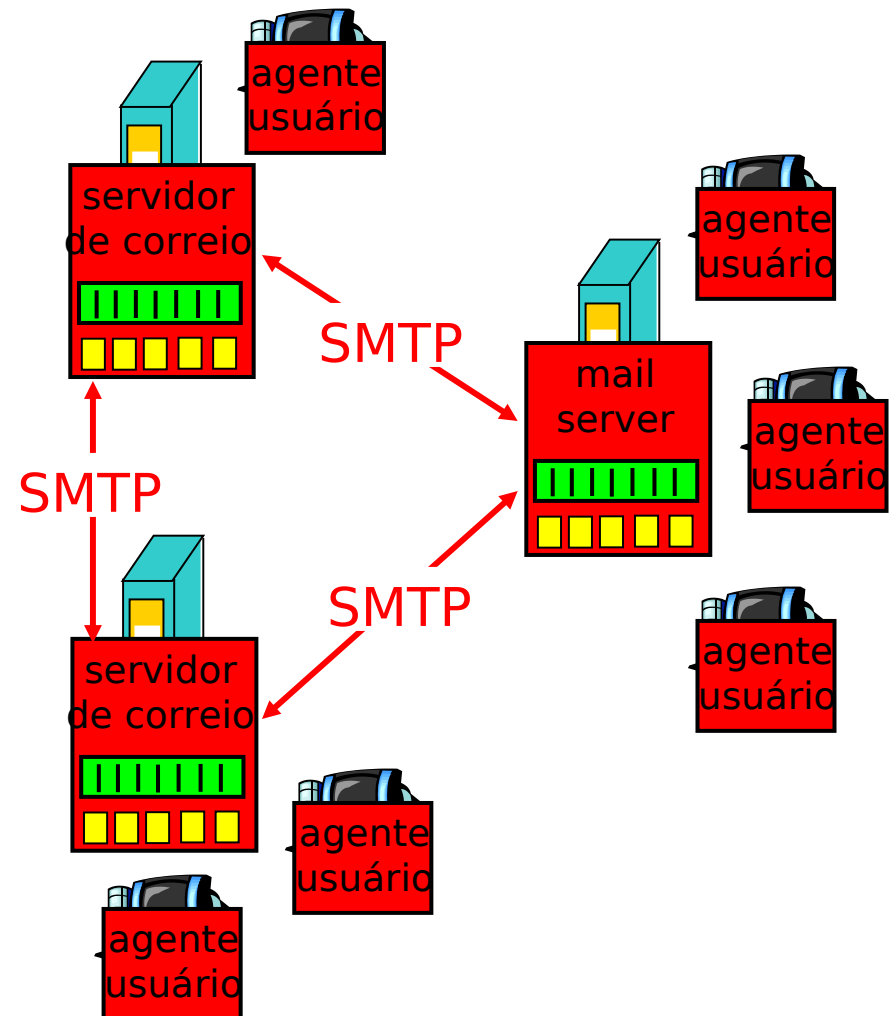
- “leitor de correio”
- composição, edição, leitura de mensagens de correio
- ex., Eudora, Outlook, elm, Netscape Messenger
- mensagens de entrada e de saída são armazenadas no servidor



Correio eletrônico: servidores de correio

Servidores de Correio

- **caixa postal** contém mensagens que chegaram (ainda não lidas) para o usuário
- **fila de mensagens** contém as mensagens de correio a serem enviadas
- **protocolo smtp** permite aos servidores de correio trocarem mensagens entre eles
 - cliente: servidor de correio que envia
 - “servidor”: servidor de correio que recebe



Correio Eletrônico: smtp [RFC 821]

- usa TCP para transferência confiável de mensagens de correio do cliente ao servidor, porta 25
- transferência direta: servidor que envia para o servidor que recebe
- três fases de transferência
 - handshaking (apresentação)
 - transferência de mensagens
 - fechamento
- interação comando/resposta
 - **comandos**: texto ASCII
 - **resposta**: código de status e frase
- mensagens devem ser formatadas em código ASCII de 7 bits

Exemplo de interação SMTP

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C:   How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

SMTP: palavras finais

- SMTP usa conexões persistentes
- SMTP exige que as mensagens (cabeçalho e corpo) estejam em ASCII de 7 bits
- algumas seqüências de caracteres não são permitidas nas mensagens (ex., CRLF.CRLF). Assim mensagens genéricas têm que ser codificadas (usualmente em “base-64” ou “quoted printable”)
- Servidor SMTP usa CRLF.CRLF para indicar o final da mensagem

Comparação com http:

- http: pull
- email: push
- ambos usam comandos e respostas em ASCII, interação comando / resposta e códigos de status
- http: cada objeto encapsulado na sua própria mensagem de resposta
- smtp: múltiplos objetos são enviados numa mensagem multiparte

Formato das Mensagens

smtp: protocolo para trocar mensagens de e-mail

RFC 822: padrão para mensagens do tipo texto:

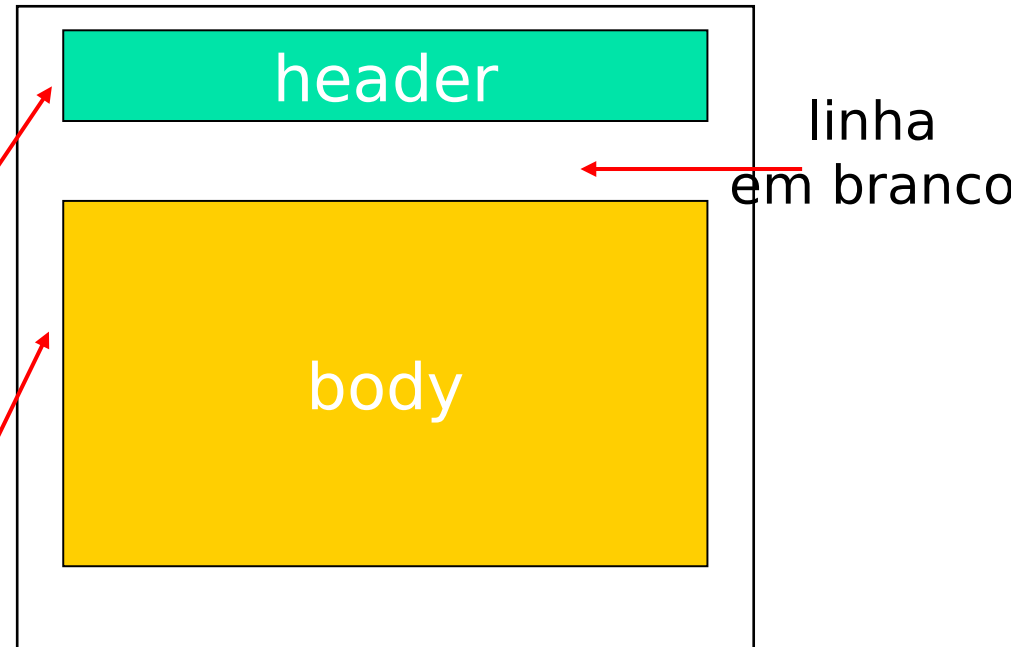
- linhas de cabeçalho, e.g.,

- To:
- From:
- Subject:

diferente dos comandos SMTP!

- corpo

- a “mensagem”, ASCII somente com caracteres



Formato das Mensagens: extensões multimedia

- MIME: multimedia mail extension, RFC 2045, 2056
- linhas adicionais no cabeçalho declaram o tipo de conteúdo MIME

MIME versão

método usado
para codificar dados

multimedia data
tipo, subtipo,

declaração de parâmetro

dados codificados

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data .....
.....
.....base64 encoded data
```

Tipos MIME

Content-Type: type/subtype; parâmetros

Text

- exemplo de subtipos: **plain**, **html**

Image

- exemplo de subtipos: **jpeg**, **gif**

Audio

- exemplo de subtipos: **basic** (codificado 8-bit μ -law), **32kadpcm** (codificação 32 kbps)

Video

- exemplo de subtipos: **mpeg**, **quicktime**

Application

- outros dados que devem ser processados pelo leitor antes de serem apresentados “visualmente”
- exemplo de subtipos: **msword**, **octet-stream**

Tipo Multiparte

From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=98766789

--98766789

Content-Transfer-Encoding: quoted-printable
Content-Type: text/plain

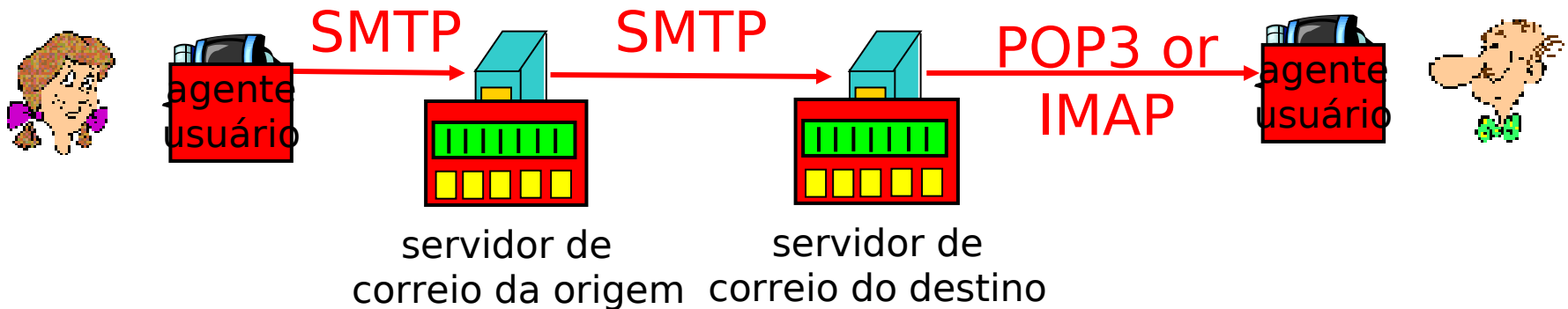
Dear Bob,
Please find a picture of a crepe.

--98766789

Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data
.....
.....base64 encoded data
--98766789--

Protocolos de acesso ao correio



- SMTP: entrega e armazena no servidor do destino
- Protocolo de acesso: recupera mensagens do servidor
 - POP: Post Office Protocol [RFC 1939]
 - autorização (agente <-->servidor) e download
 - IMAP: Internet Mail Access Protocol [RFC 1730]
 - maiores recursos (mais complexo)
 - manipulação de mensagens armazenadas no servidor
 - HTTP: Hotmail , Yahoo! Mail, etc.

protocolo POP3

fase de autorização

- comandos do cliente:
 - **user**: declara nome do usuário
 - **pass**: password
- respostas do servidor
 - **+OK**
 - **-ERR**

fase de transação, cliente:

- **list**: lista mensagens e tamanhos
- **retr**: recupera mensagem pelo número
- **dele**: apaga
- **quit**

```
S: +OK POP3 server ready
C: user alice
S: +OK
C: pass hungry
S: +OK user successfully logged on

C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

DNS: Domain Name System

Pessoas: muitos identificadores:

- RG, nome, passaporte

Internet hosts, roteadores:

- endereços IP (32 bit) - usados para endereçar datagramas
- “nome”, ex.,
gaia.cs.umass.edu - usados por humanos

Q: relacionar nomes com endereços IP?

Domain Name System:

- *base de dados distribuída* implementada numa hierarquia de muitos *servidores de nomes*
- *protocolo de camada de aplicação* host, roteadores se comunicam com servidores de nomes para *resolver* nomes (translação nome/endereço)
 - nota: função interna da Internet, implementada como protocolo da camada de aplicação
 - complexidade na “borda” da rede

Servidores de Nomes DNS

Porque não centralizar o DNS?

- ponto único de falha
- volume de tráfego
- base de dados distante
- manutenção

Não cresce junto com a rede!

- nenhum servidor tem todos os mapeamentos de nomes para endereços IP

servidores de nomes locais:

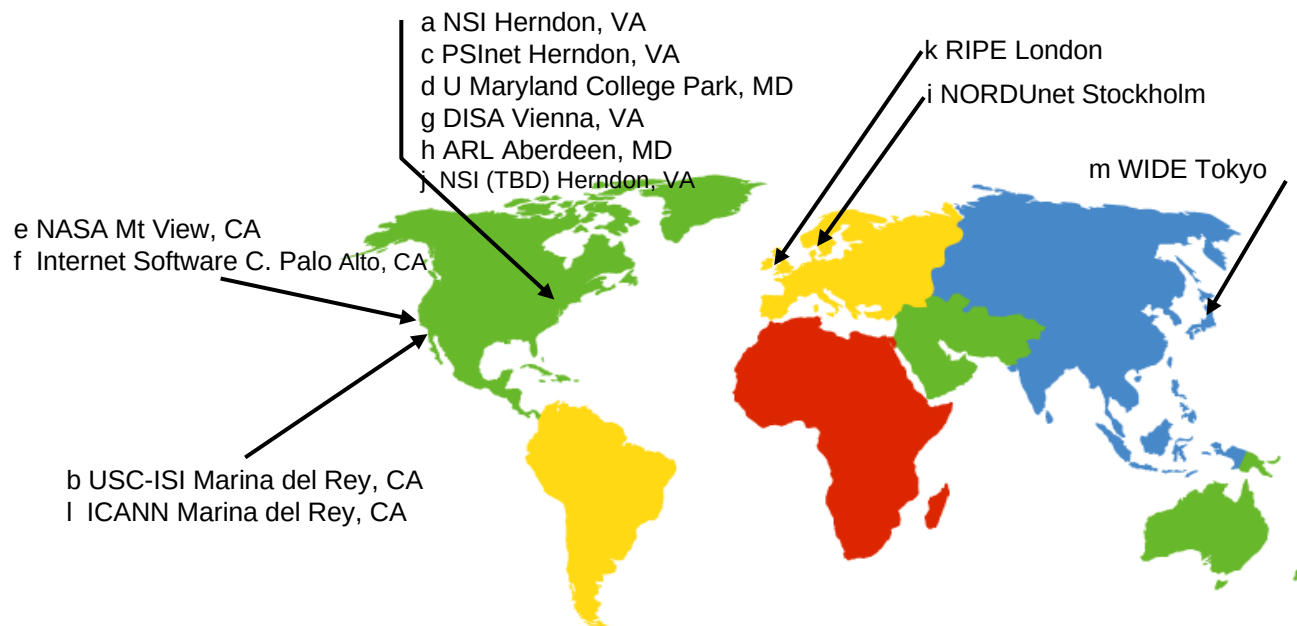
- cada ISP ou empresa tem um *servidor de nomes local (default)*
- Consultas dos computadores locais ao DNS vão primeiro para o servidor de nomes local

servidor de nomes autoritativo:

- para um computador: armazena o nome e o endereço IP daquele computador
- pode realizar mapeamentos de nomes para endereços para aquele nome de computador

DNS: Servidores de Nomes Raiz

- são contatados pelos servidores de nomes locais que não podem resolver um nome
- servidores de nomes raiz::
 - buscam servidores de nomes autoritativos se o mapeamento do nome não for conhecido
 - conseguem o mapeamento
 - retornam o mapeamento para o servidor de nomes local



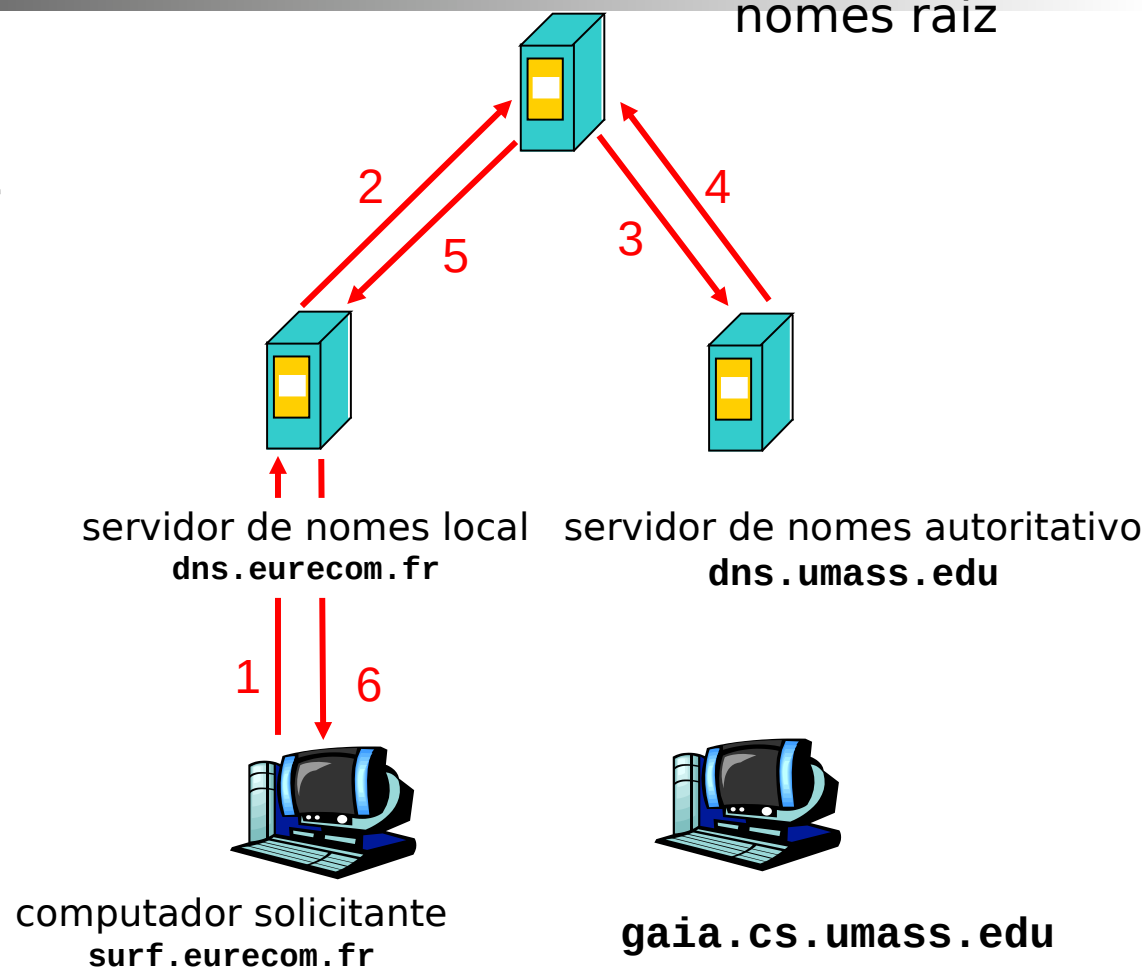
existem 13 servidores
de nomes raiz no
mundo

DNS: exemplo simples

servidor de
nomes raiz

host **surf.eurecom.fr** quer
o endereço IP de
gaia.cs.umass.edu

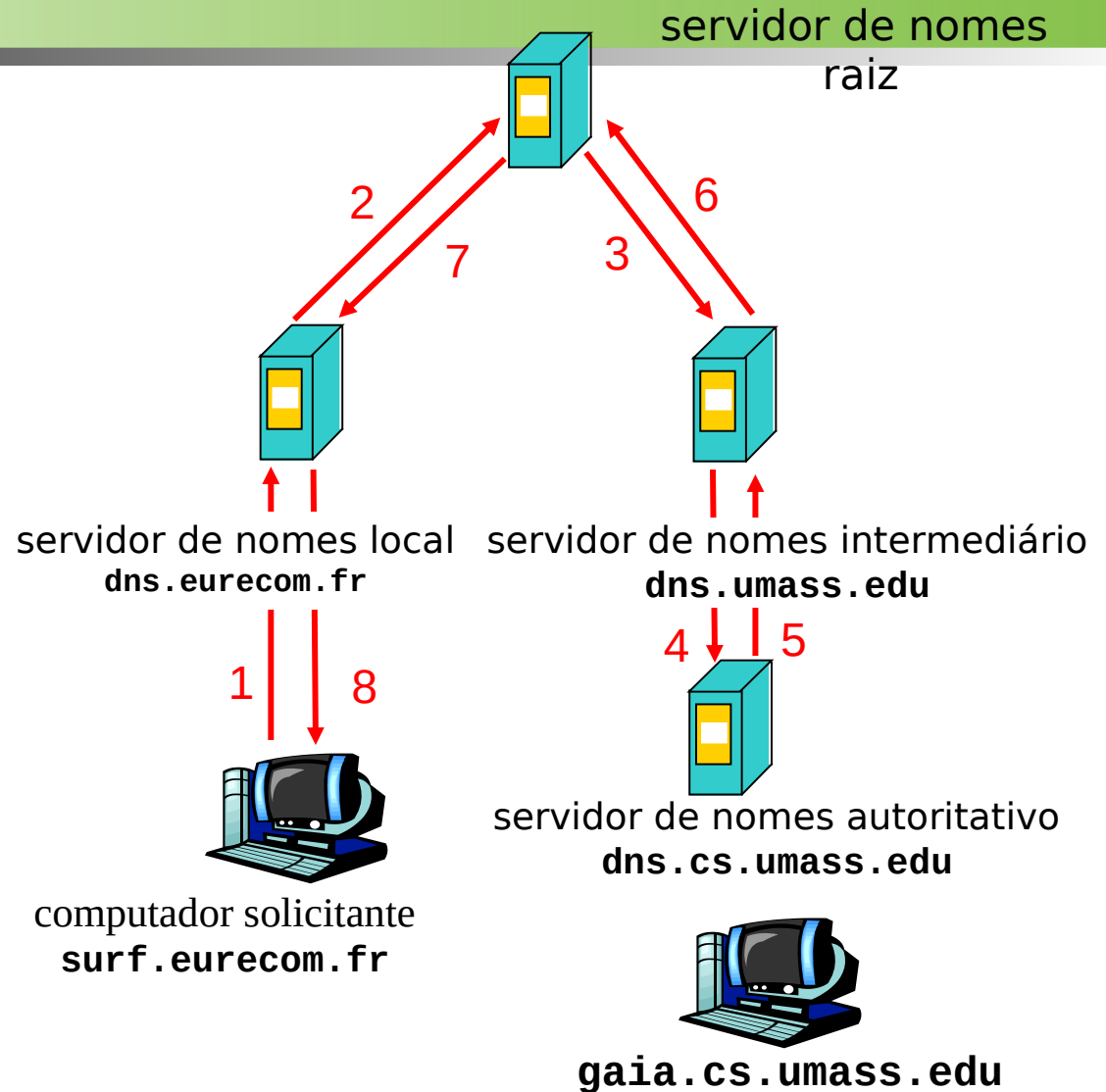
1. contata seu servidor DNS local, **dns.eurecom.fr**
2. **dns.eurecom.fr** contata o servidor de nomes raiz se necessário
3. o servidor de nomes raiz contata o servidor de nomes autoritativo, **dns.umass.edu**, se necessário



DNS: exemplo

Servidor de nomes raiz:

- pode não conhecer o servidor de nomes autoritativo para um certo nome
- pode conhecer: *servidor de nomes intermediário*: aquele que deve ser contactado para encontrar o servidor de nomes autoritativo



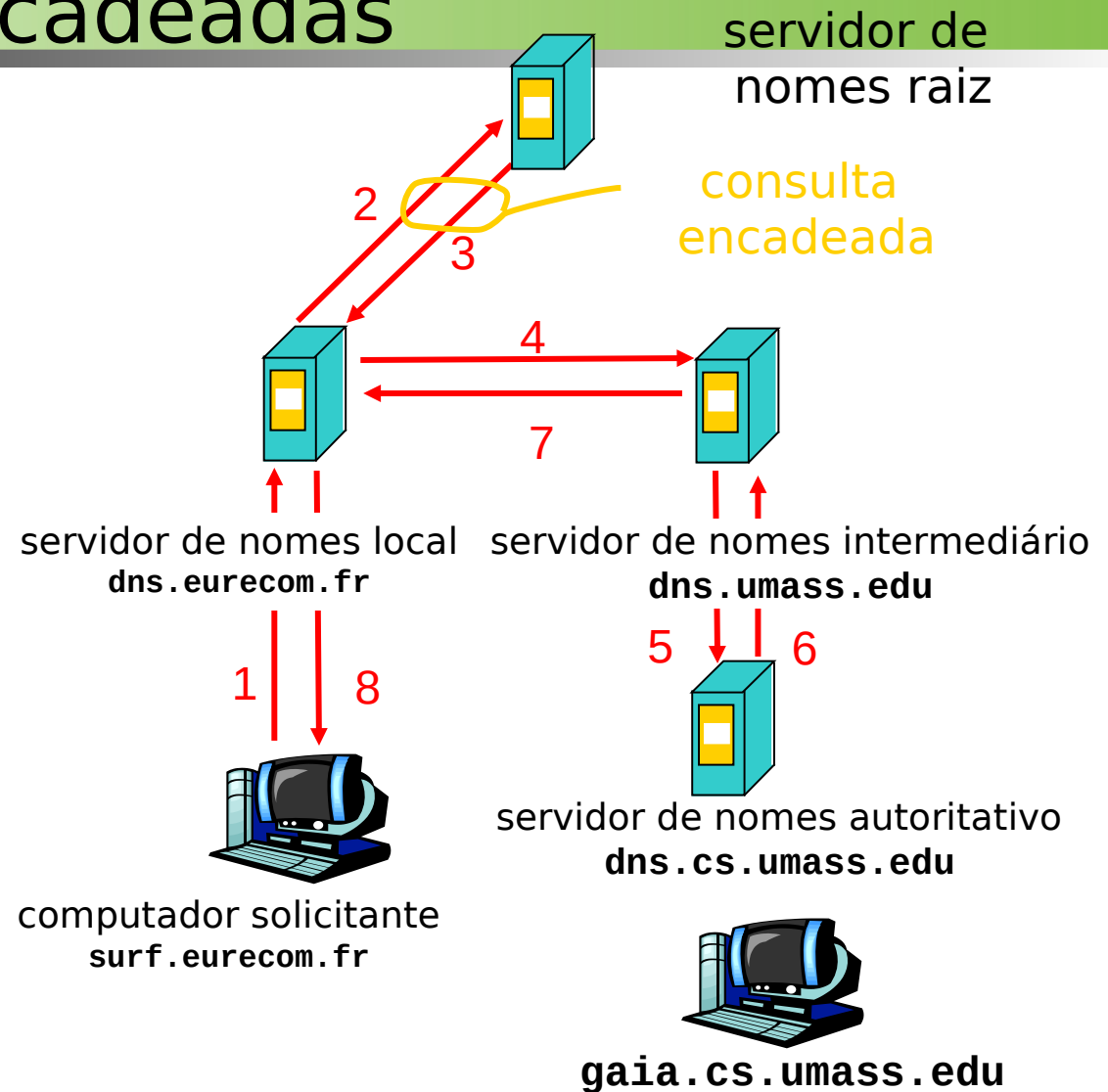
DNS: consultas encadeadas

consulta recursiva:

- transfere a tarefa de resolução do nome para o servidor de nomes consultado
- carga pesada?

consulta encadeada:

- servidor contactado responde com o nome de outro servidor de nomes para contato
- “Eu não sei isto ,mas pergunte a este servidor”



DNS: armazenando e atualizando registros

- uma vez que um servidor de nomes apreende um mapeamento, ele armazena o mapeamento num registro do tipo *cache*
 - registro do cache tornam-se obsoletos (desaparecem) depois de um certo tempo
- mecanismos de atualização e notificação estão sendo projetados pelo IETF
 - RFC 2136
 - <http://www.ietf.org/html.charters/dnsind-charter.html>

Registros do DNS

DNS: base de dados distribuída que armazena registros de recursos (RR)

formato dos RR: (name, value,
type, ttl)

- Type=A
 - **name** é o nome do computador
 - **value** é o endereço IP
- Type=NS
 - **name** é um domínio (ex. foo.com)
 - **value** é o endereço IP do servidor de nomes autoritativo para este domínio
- Type=CNAME
 - **name** é um “apelido” para algum nome “canônico” (o nome real)
www.ibm.com é realmente servereast.backup2.ibm.com
 - **value** é o nome canônico
- Type=MX
 - **value** é o nome do servidor de correio associado com **name**

DNS: protocolo e mensagens

protocolo DNS: mensagem de *consulta* e *resposta*, ambas com o mesmo *formato de mensagem*

cabeçalho da msg

- **identificação**: número de 16 bit para consulta, resposta usa o mesmo número
- **flags**:
 - consulta ou resposta
 - recursão desejada
 - recursão disponível
 - resposta é autoritativa

identification	flags
number of questions	number of answer RRs
number of authority RRs	number of additional RRs
questions (variable number of questions)	
answers (variable number of resource records)	
authority (variable number of resource records)	
additional information (variable number of resource records)	

↑
12 bytes
↓

DNS: protocolo e mensagens

Campos de nome e tipo
para uma consulta

RRs de resposta
a uma consulta

registros para
servidores autoritativos

informação adicional
que pode ser útil

identification	flags
number of questions	number of answer RRs
number of authority RRs	number of additional RRs
questions (variable number of questions)	
answers (variable number of resource records)	
authority (variable number of resource records)	
additional information (variable number of resource records)	

↑
12 bytes
↓

Programação de Sockets

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

Socket API

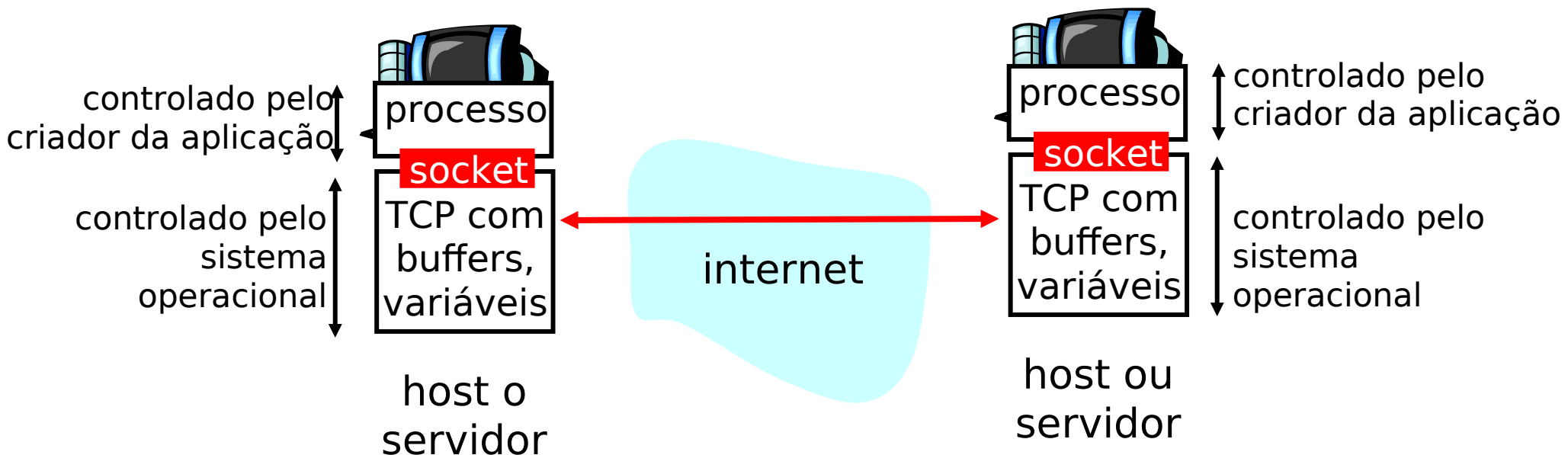
- introduzida no BSD4.1 UNIX, 1981
- explicitamente criados, usados e liberados pelas aplicações
- paradigma cliente/servidor
- dois tipos de serviço de transporte via socket API:
 - datagrama não confiável
 - confiável, orientado a cadeias de bytes

socket interface *local*, criada e possuída pelas aplicações, *controlada pelo OS* (uma “porta”) na qual os processo de aplicação podem *tanto enviar quanto receber* mensagens de e para outro processo de aplicação (local ou remoto)

Programação de Sockets com TCP

Socket: uma porta 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 contactar o servidor


- processo servidor já deve estar executando antes de ser contactado
- servidor deve ter criado socket (porta) que aceita o contato do cliente

Cliente contacta o servidor através de:

- 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 contactado pelo cliente, **o TCP do servidor cria um novo socket** para o processo servidor comunicar-se com o cliente
 - permite o servidor conversar com múltiplos clientes

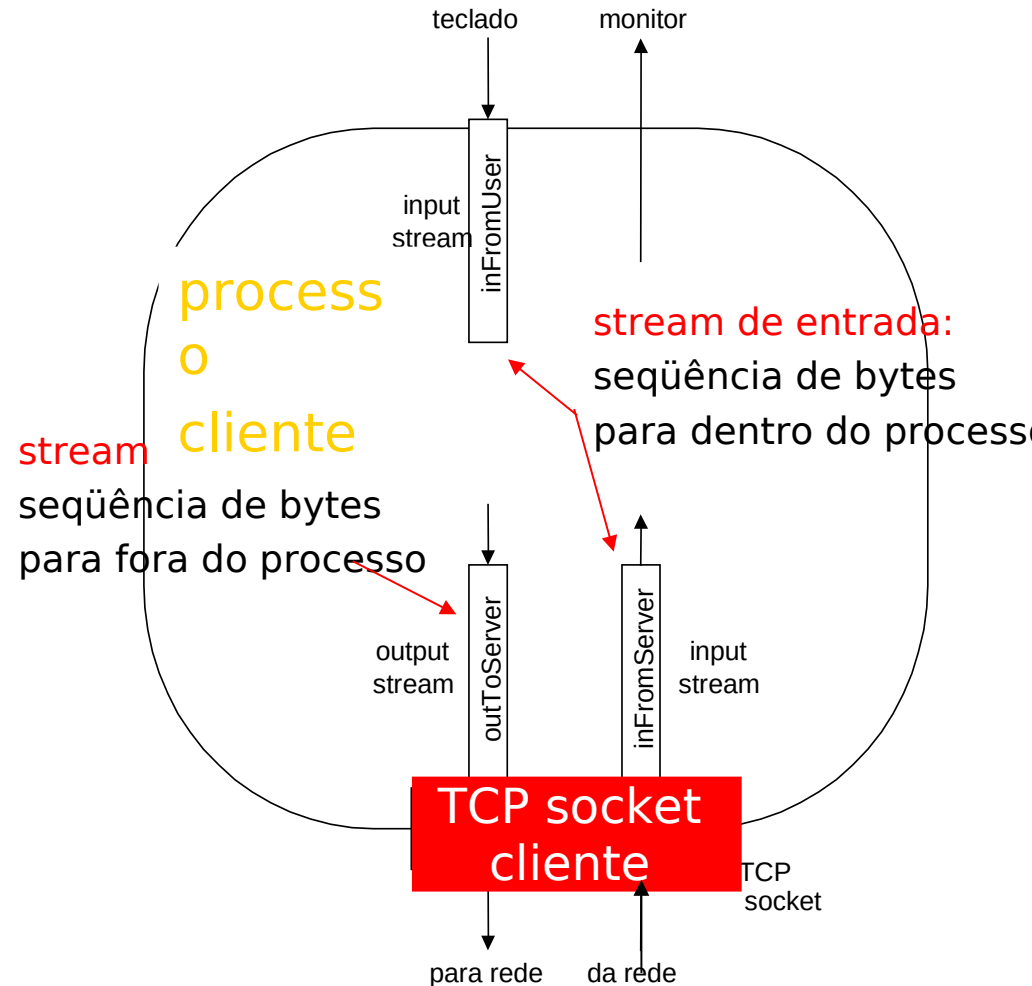
ponto de vista da aplicação

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

Programação de Sockets *com* *TCP*

Exemplo de aplicação cliente-servidor:

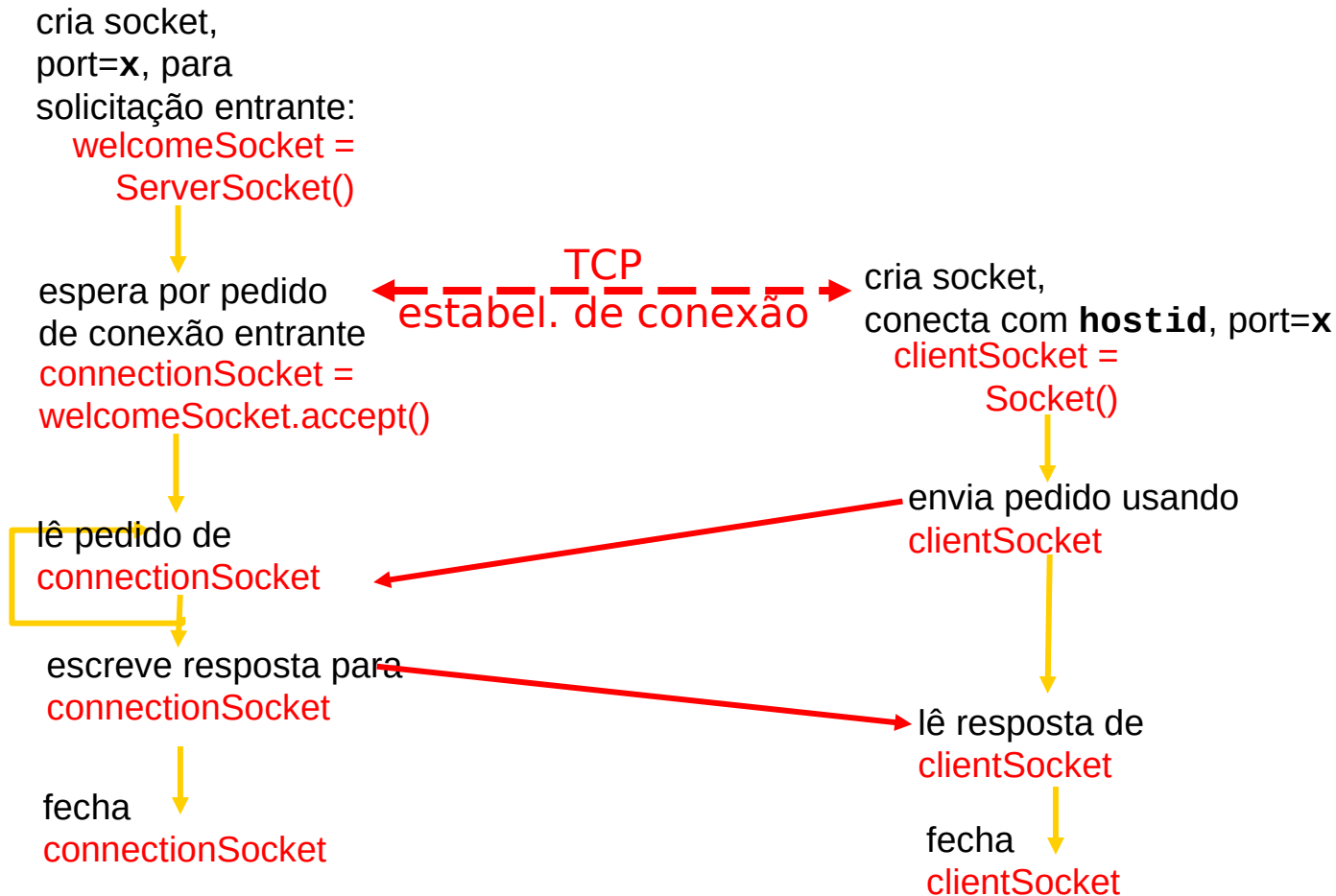
- cliente lê linha da entrada padrão do sistema (**inFromUser** stream) , envia para o servidor via socket (**outToServer** stream)
- servidor lê linha do socket
- servidor converte linha para letras maiúsculas e envia de volta ao cliente
- cliente lê a linha modificada através do (**inFromServer** stream)



Interação Cliente/servidor: TCP

Servidor (executando em `hostid`)

Cliente



Exemplo: cliente Java (TCP)

```
import java.io.*;  
import java.net.*;  
class TCPClient {
```

```
    public static void main(String argv[]) throws Exception  
    {
```

```
        String sentence;  
        String modifiedSentence;
```

Cria
stream de entrada → `BufferedReader inFromUser =
 new BufferedReader(new InputStreamReader(System.in));`

Cria
socket cliente, conecta ao servidor → `Socket clientSocket = new Socket("hostname", 6789);`

Cria
stream de saída ligado ao socket → `DataOutputStream outToServer =
 new DataOutputStream(clientSocket.getOutputStream());`

Exemplo: cliente Java (TCP), cont.

Cria
stream de entrada
ligado ao socket

```
BufferedReader inFromServer =  
    new BufferedReader(new  
        InputStreamReader(clientSocket.getInputStream()));  
  
sentence = inFromUser.readLine();  
  
Envia linha  
para o servidor
```

```
outToServer.writeBytes(sentence + '\n');
```

Lê linha
do servidor

```
modifiedSentence = inFromServer.readLine();  
System.out.println("FROM SERVER: " + modifiedSentence);  
  
clientSocket.close();  
  
}  
}
```

Exemplo: servidor Java (TCP)

```
import java.io.*;  
import java.net.*;
```

```
class TCPServer {
```

```
    public static void main(String argv[]) throws Exception  
    {
```

```
        String clientSentence;  
        String capitalizedSentence;
```

Cria
socket de aceitação
na porta 6789

```
        ServerSocket welcomeSocket = new ServerSocket(6789);
```

Espera, no socket
de aceitação por
contato do cliente

```
        while(true) {
```

```
            Socket connectionSocket = welcomeSocket.accept();
```

Cria stream de
entrada, ligado
ao socket

```
            BufferedReader inFromClient =  
                new BufferedReader(new  
                    InputStreamReader(connectionSocket.getInputStream()));
```

Exemplo: servidor Java (cont)

Cria stream de
saída, ligado ao
socket

```
DataOutputStream outToClient =  
    new DataOutputStream(connectionSocket.getOutputStream());
```

Lê linha do
socket

```
clientSentence = inFromClient.readLine();
```

```
capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

Escreve linha
para o socket

```
outToClient.writeBytes(capitalizedSentence);
```

```
}  
}  
}
```

Fim do while loop,
retorne e espere por
outra conexão do cliente

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

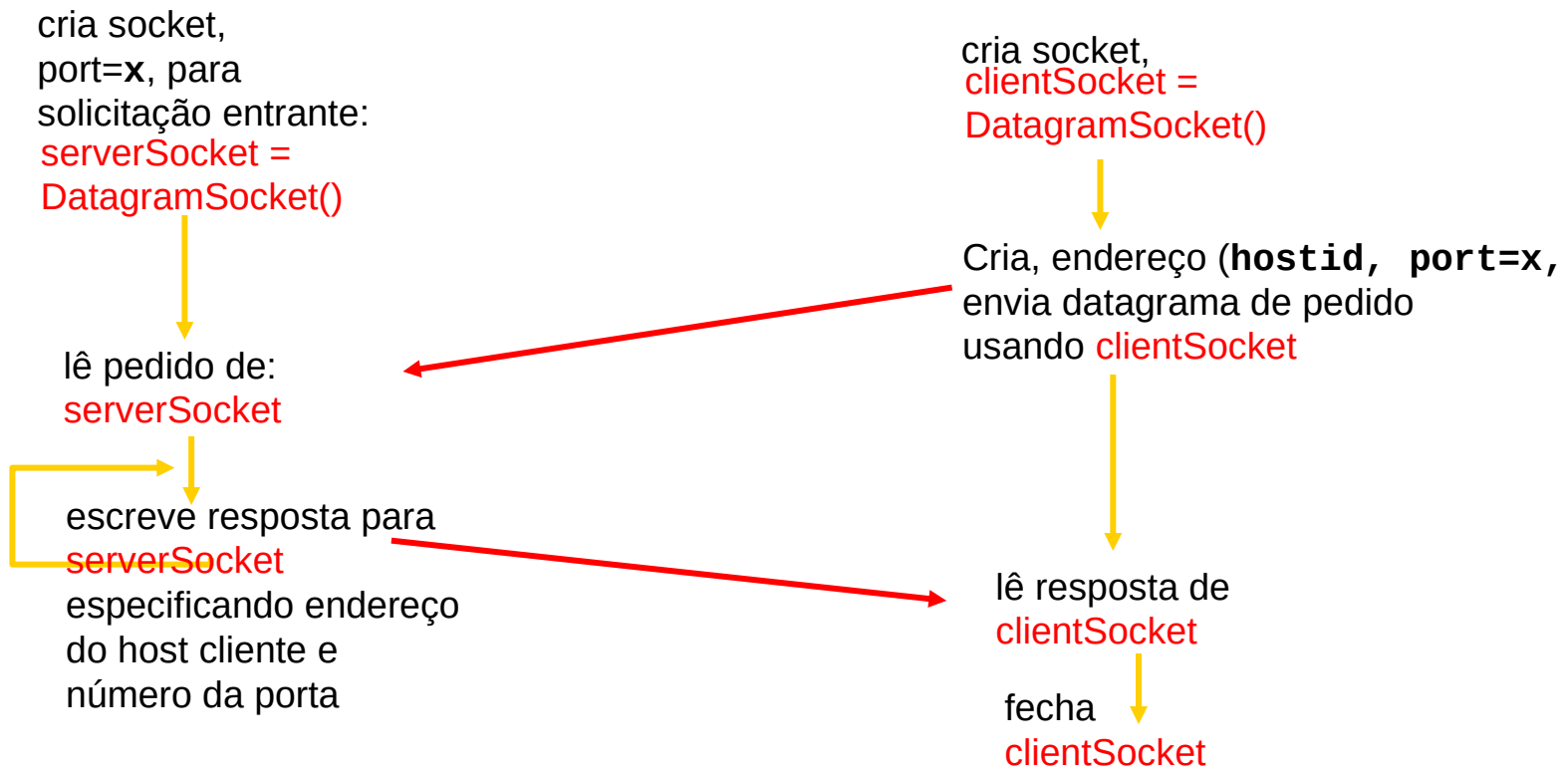
UDP fornece a transferência

não confiável de grupos de bytes ("datagramas") entre o cliente e o servidor

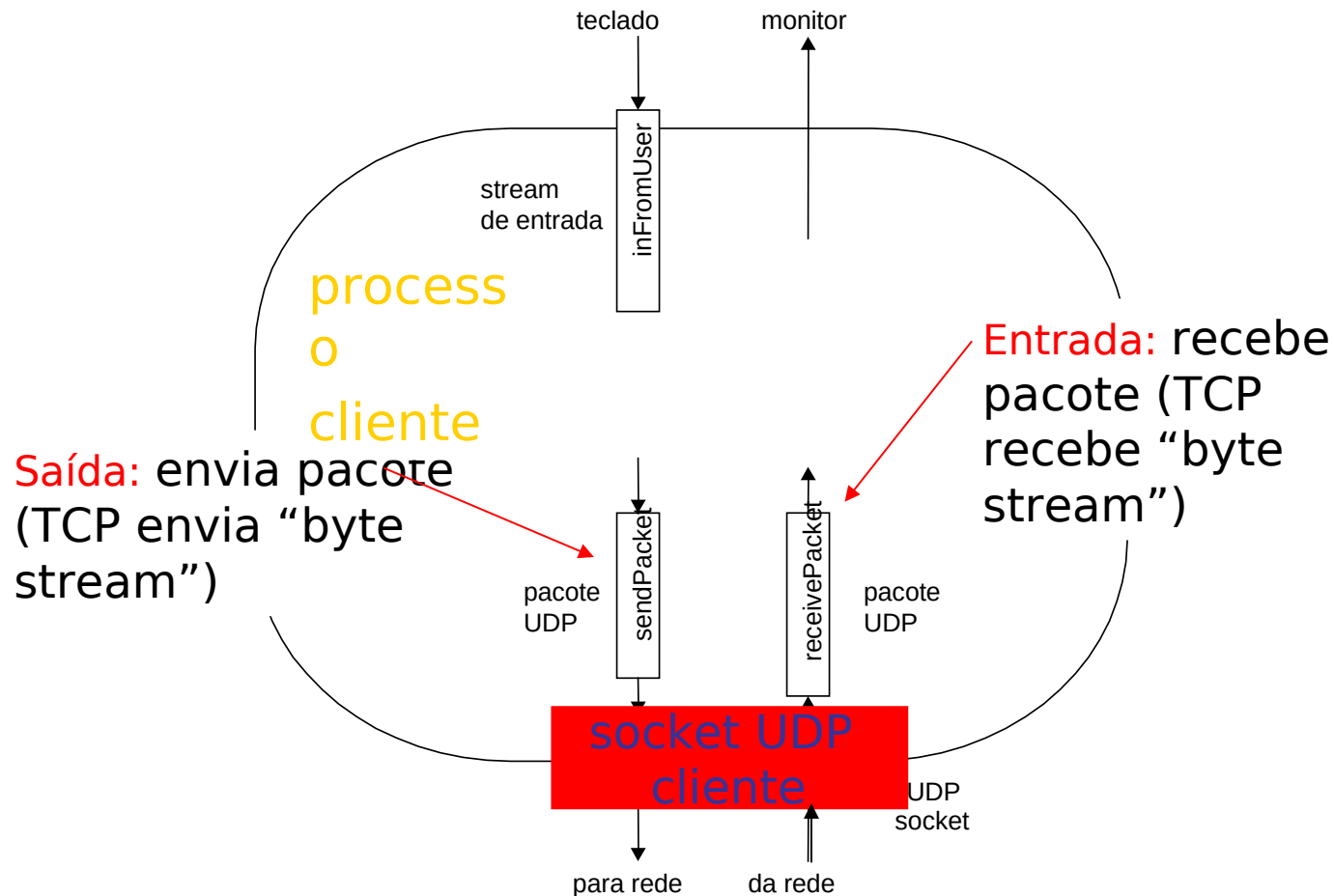
Interação Cliente/servidor: UDP

Servidor (executando `hostid`)

Cliente



Exemplo: cliente Java (UDP)



Exemplo: cliente Java (UDP)

```
import java.io.*;  
import java.net.*;
```

```
class UDPClient {  
    public static void main(String args[]) throws Exception  
    {
```

Cria
stream de entrada

```
        BufferedReader inFromUser =  
            new BufferedReader(new InputStreamReader(System.in));
```

Cria
socket cliente

```
        DatagramSocket clientSocket = new DatagramSocket();
```

Translada
nome do host para
endereço IP
usando DNS

```
        InetAddress IPAddress = InetAddress.getByName("hostname");
```

```
        byte[] sendData = new byte[1024];  
        byte[] receiveData = new byte[1024];
```

```
        String sentence = inFromUser.readLine();  
        sendData = sentence.getBytes();
```

Exemplo: cliente Java (UDP), cont.

Cria datagrama com
dados a enviar,
tamanho, endereço IP
porta

```
DatagramPacket sendPacket =  
    new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

Envia datagrama
para servidor

```
clientSocket.send(sendPacket);
```

Lê datagrama
do servidor

```
DatagramPacket receivePacket =  
    new DatagramPacket(receiveData, receiveData.length);  
  
clientSocket.receive(receivePacket);  
  
String modifiedSentence =  
    new String(receivePacket.getData());  
  
System.out.println("FROM SERVER:" + modifiedSentence);  
clientSocket.close();  
}  
}
```

Exemplo: servidor Java (UDP)

```
import java.io.*;  
import java.net.*;
```

```
class UDPServer {  
    public static void main(String args[]) throws Exception  
    {
```

Cria
socket datagrama
na porta 9876

```
        DatagramSocket serverSocket = new DatagramSocket(9876);
```

```
        byte[] receiveData = new byte[1024];  
        byte[] sendData = new byte[1024];
```

```
        while(true)  
        {
```

Cria espaço para
datagramas recebidos

```
            DatagramPacket receivePacket =  
                new DatagramPacket(receiveData, receiveData.length);
```

Recebe
datagrama

```
            serverSocket.receive(receivePacket);
```

Exemplo: servidor Java, (cont.)

```
String sentence = new String(receivePacket.getData());
```

Obtém endereço IP
e número da porta
do transmissor

```
InetAddress IPAddress = receivePacket.getAddress();
```

```
int port = receivePacket.getPort();
```

```
String capitalizedSentence = sentence.toUpperCase();
```

```
sendData = capitalizedSentence.getBytes();
```

Cria datagrama
para enviar ao cliente

```
DatagramPacket sendPacket =  
    new DatagramPacket(sendData, sendData.length, IPAddress,  
                        port);
```

Escreve o
datagrama para
dentro do socket

```
serverSocket.send(sendPacket);
```

```
}  
}  
}
```

Termina o while loop,
retorna e espera por
outro datagrama

Programação de Sockets: referências

tutorial sobre C-language tutorial (audio/slides):

- “Unix Network Programming” (J. Kurose),
<http://manic.cs.umass.edu>.

Tutoriais sobre Java:

- “Socket Programming in Java: a tutorial,”
<http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html>

Bibliografia

- James F. Kurose e Keith W. Ross
- Redes de Computadores e a Internet:
- Andrew S. Tanenbaum
- Redes de Computadores
- SOUSA, Lindeberg Barros de
- Redes de computadores: dados, voz e imagem
- Comer, Douglas E.,
- Interligação de Redes Com Tcp/ip

Atividades de Aprendizagem

- Próximas Aulas
 - Resoluções de Exercícios sobre a Camada de Aplicação
 - Avaliação Parcial 2.2