

# Infra-Estrutura de Comunicação (IF678)

Módulo II

Fonte: kurose  
Adaptações : Prof. Paulo Gonçalves  
pasg@cin.ufpe.br  
CIn/UFPE

# Módulo 2: Camada Aplicação

- ❑ 2.1 Princípios das aplicações de rede
- ❑ 2.2 Web e HTTP
- ❑ 2.3 FTP
- ❑ 2.4 e-mail (Electronic Mail)
  - ❖ SMTP, POP3, IMAP
- ❑ 2.5 DNS
- ❑ 2.6 compartilhamento de arquivos (P2P)
- ❑ 2.7 Programação de Sockets com TCP
- ❑ 2.8 Programação de Socket com UDP
- ❑ 2.9 Construindo um servidor Web

# Módulo 2: Camada Aplicação

## Nossos objetivos:

- ❑ conceitos, aspectos de implementação de protocolos de aplicação de rede
  - ❖ Modelos de serviço da camada de transporte
  - ❖ Paradigma cliente-servidor
  - ❖ Paradigma peer-to-peer
- ❑ Aprendizado sobre protocolos examinando protocolos da camada aplicação
  - ❖ HTTP
  - ❖ FTP
  - ❖ SMTP / POP3 / IMAP
  - ❖ DNS
- ❑ Programação de aplicações de rede
  - ❖ API de socket

# Algumas aplicações de rede

- ❑ E-mail
- ❑ Web
- ❑ Instant messaging (IM)
- ❑ Login remoto
- ❑ Compartilhamento de arquivos (P2P)
- ❑ Jogos multi-usuários em rede
- ❑ Streaming de vídeo clips armazenados
- ❑ Telefonia Internet
- ❑ Vídeo-conferência em tempo-real
- ❑ Computação paralela massiva

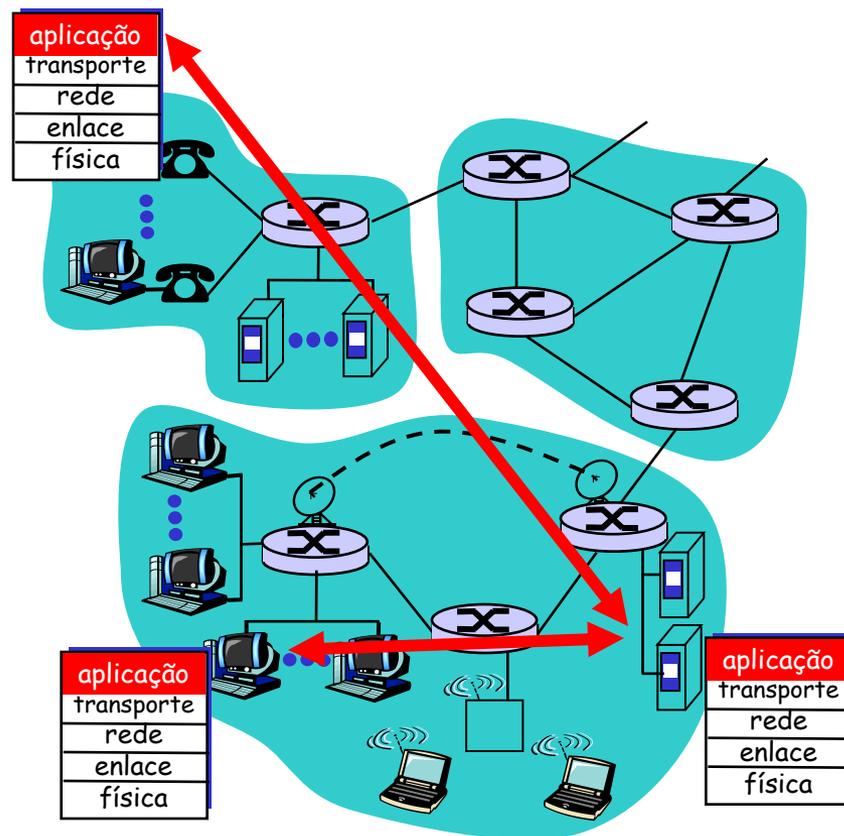
# Criando uma aplicação de rede

## Escreva programas que

- ❖ Executem em diferentes end systems e se comuniquem através de uma rede
- ❖ e.g., Web: o software do servidor Web se comunica com o software do browser

## Pouco software escrito para dispositivos dentro do núcleo da rede

- ❖ Dispositivos do núcleo da rede não executam código de aplicação do usuário
- ❖ Aplicações nos end systems permitem desenvolvimento e disseminação rápidos



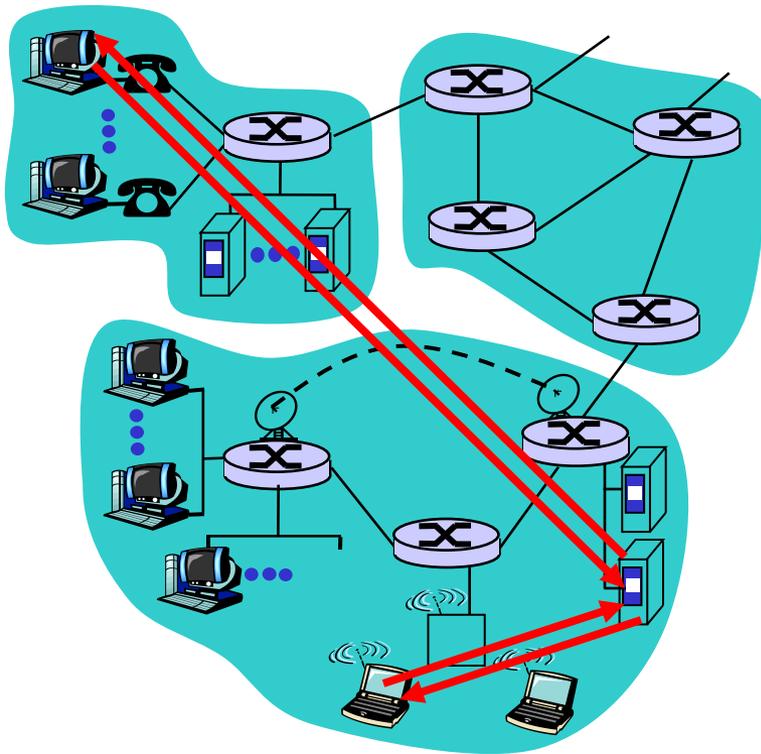
# Módulo 2: Camada Aplicação

- ❑ 2.1 Princípios das aplicações de rede
- ❑ 2.2 Web e HTTP
- ❑ 2.3 FTP
- ❑ 2.4 e-mail (Electronic Mail)
  - ❖ SMTP, POP3, IMAP
- ❑ 2.5 DNS
- ❑ 2.6 compartilhamento de arquivos (P2P)
- ❑ 2.7 Programação de Sockets com TCP
- ❑ 2.8 Programação de Socket com UDP
- ❑ 2.9 Construindo um servidor Web

# Arquiteturas de Aplicações

- ❑ Cliente-servidor
- ❑ Peer-to-peer (P2P)
- ❑ Híbrido cliente-servidor e P2P

# Arquitetura cliente-servidor



## servidor:

- ❖ Host sempre "on-line"
- ❖ Endereço IP permanente
- ❖ Múltiplos servidores para escalabilidade

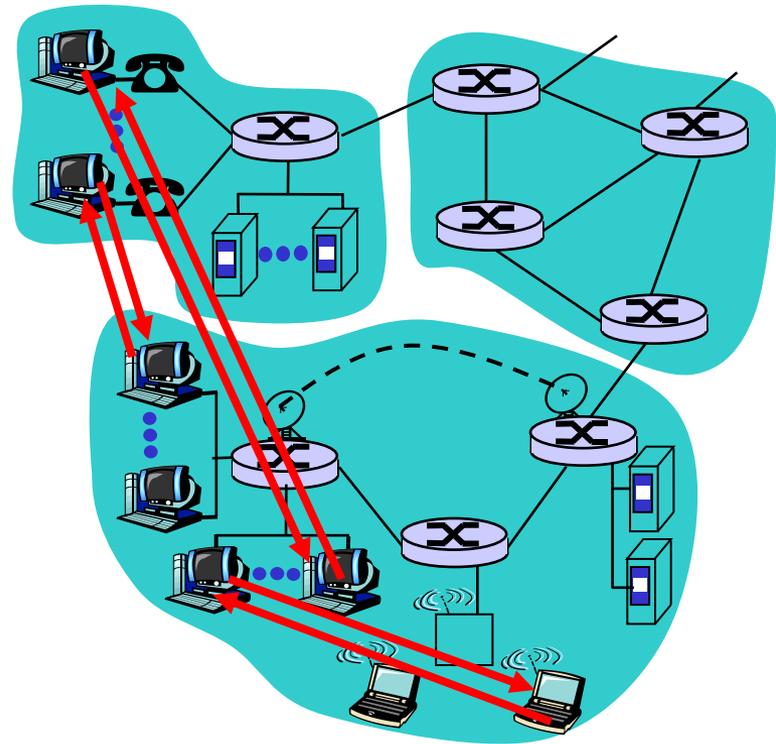
## clientes:

- ❖ Se comunicam com o servidor
- ❖ Podem ter conexão intermitente (nem sempre "on-line")
- ❖ Podem ter endereço IP dinâmico
- ❖ Não se comunicam diretamente entre eles

# Arquitetura P2P pura

- ❑ Nenhum servidor sempre "on-line"
- ❑ end systems arbitrários se comunicam diretamente
- ❑ peers são conectados de forma intermitente e mudam de endereço IP
- ❑ exemplo: Gnutella

Altamente escalável mas difícil de gerenciar



# Híbrido: cliente-servidor e P2P

## Skype

- ❖ Aplicação para telefonia via Internet
- ❖ Obtenção do endereço do computador remoto: servidor(es) centralizado(s)
- ❖ Conexão entre os Clientes (usuários) é direta (sem intermediação de servidores)

## Instant messaging

- ❖ Chatting entre dois usuários é P2P
- ❖ Detecção de presença/localização centralizada:
  - Usuário registra seu endereço IP no servidor central quando estiver on-line
  - Usuário contacta servidor central para encontrar IP dos "camaradas"

# Comunicação entre processos

- Processo:** programa executando em um host
- No mesmo host, 2 processos se comunicam através de uma **API de comunicação entre processos** (definida pelo Sistema Operacional)
  - processos em diferente hosts se comunicam através da troca de **mensagens**

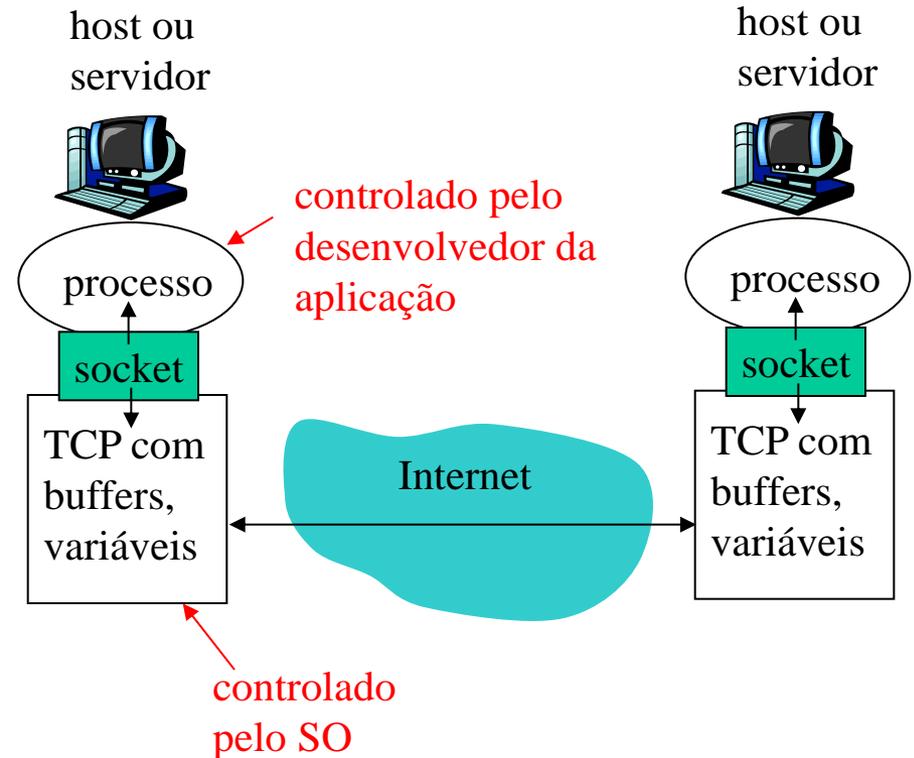
**Processo Cliente:**  
processo que inicia a comunicação

**Processo Servidor:**  
processo que aguarda ser contactado

- Nota: aplicações baseadas em arquitetura P2P possuem processos clientes e processos servidores

# Sockets

- processo **envia/recebe** mensagens **para/de** seu **socket**
- socket é análogo a uma porta
  - ❖ Processo emissor envia mensagem para fora da "porta"
  - ❖ Processo emissor conta com a infraestrutura de transporte no outro lado da porta que leva a mensagem ao socket do processo receptor



- API: (1) escolha do protocolo de transporte; (2) habilidade para escolher poucos parâmetros (**muito mais em breve**)

# Endereçamento de processos

- ❑ Para receber mensagens processos devem possuir um *identificador*
- ❑ O host possui um endereço IP de 32 bits único
- ❑ **Q:** o endereço IP de um host onde executa um processo é suficiente para identificar o processo?

# Endereçamento de processos

- ❑ Para receber mensagens processos devem possuir um *identificador*
- ❑ O host possui um endereço IP de 32 bits único
- ❑ **Q:** o endereço IP de um host onde executa um processo é suficiente para identificar o processo?
  - ❖ **Resp:** Não, muitos processos podem estar em execução no mesmo host
- ❑ *identificador* inclui ambos **endereço IP** e **número da porta** associado ao processo no host.
- ❑ Exemplo de números de porta:
  - ❖ Servidor HTTP: 80
  - ❖ Servidor de e-mail: 25
- ❑ Para enviar uma mensagem HTTP para o servidor web [www.cin.ufpe.br](http://www.cin.ufpe.br):
  - ❖ Endereço IP: 172.21.0.3
  - ❖ Porta número: 80
- ❑ Mais em breve...

# Protocolos da camada aplicação definem

- ❑ Tipos de mensagens trocadas,
  - ❖ e.g., request, response
- ❑ Sintaxe da Mensagem:
  - ❖ Quais campos compõem a mensagem & como campos estão delineados
- ❑ Semântica da Mensagem
  - ❖ Significado da informação em cada campo
- ❑ Regras de quando e como processos enviam e respondem mensagens

## Protocolos de domínio público:

- ❑ definidos em RFCs
- ❑ permite interoperabilidade
- ❑ e.g., HTTP, SMTP

## Protocolos proprietários

- ❑ e.g., KaZaA

# Que tipo de serviço de transporte uma aplicação precisa?

## Perda de Dados

- ❑ algumas aplicações (e.g., áudio) podem tolerar alguma perda
- ❑ Outras aplicações (e.g., ftp, telnet) requerem transferência de dados 100% confiável

## Tempo ("Prazo de entrega")

- ❑ Algumas aplicações (e.g., telefonia Internet, jogos interativos) requerem baixo atraso para funcionarem corretamente

## Banda passante

- ❑ Algumas aplicações (e.g., multimídia) requerem uma quantidade mínima de banda passante para funcionar de modo adequado
- ❑ Outras aplicações ("aplicações elásticas") fazem uso de qualquer quantidade de banda passante que elas conseguem

## Requisitos do serviço de transporte para aplicações comuns

<u>aplicação</u>	<u>Perdas</u>	<u>Banda passante</u>	<u>Sensibilidade ao tempo</u>
ftp	não	elástica	não
e-mail	não	elástica	não
Documentos Web	não	elástica	não
áudio/vídeo (tempo real)	tolera	áudio: 5kbps-1Mbps vídeo: 10kbps-5Mbps	sim, 100's msec
Streaming de áudio/vídeo	tolera	Mesmo de cima	sim, poucos seg
Jogos interativos	tolera	poucos kbps ou mais	sim, 100's msec
instant messaging	não	elástica	sim e não!

# Serviços dos protocolos de transporte Internet

## Serviço TCP:

- ❑ *Orientado à conexão:* setup requerido entre processos cliente e servidor
- ❑ *Transporte confiável* entre processo emissor e receptor
- ❑ *Controle de fluxo:* emissor não envia mais que o receptor pode receber
- ❑ *Controle de congestionamento:* limita o emissor quando a rede está sobrecarregada
- ❑ *Não provê:* "prazo de entrega", garantia mínima de banda passante

## Serviço UDP:

- ❑ Transferência não-confiável de dados entre o processo emissor e receptor
- ❑ Não provê: setup de conexão, confiabilidade, controle de fluxo, controle de congestionamento, "entrega no prazo", nem garantia de banda passante

Q: Se é assim por que então existe o UDP?

# Aplicações Internet: aplicação, protocolos de transporte

<b>Aplicação</b>	<b>Protocolo da camada Aplicação</b>	<b>Protocolo de transporte usado</b>
e-mail	SMTP [RFC 2821]	TCP
Acesso a terminal remoto	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
Transferência de arquivo	FTP [RFC 959]	TCP
Streaming multimídia	proprietário (e.g. RealNetworks)	TCP ou UDP
Telefonia Internet	proprietário (e.g., Skype, Google Talk)	tipicamente UDP

# Módulo 2: Camada Aplicação

- ❑ 2.1 Princípios das aplicações de rede
- ❑ 2.2 Web e HTTP
- ❑ 2.3 FTP
- ❑ 2.4 e-mail (Electronic Mail)
  - ❖ SMTP, POP3, IMAP
- ❑ 2.5 DNS
- ❑ 2.6 compartilhamento de arquivos (P2P)
- ❑ 2.7 Programação de Sockets com TCP
- ❑ 2.8 Programação de Socket com UDP
- ❑ 2.9 Construindo um servidor Web

# Web e HTTP

## Primeiramente alguns jargões

- ❑ **Página Web** consiste em **objetos**
- ❑ Objetos podem ser arquivos HTML, JPEG, applets Java, arquivos de áudio, arquivos de vídeo ...
- ❑ Página Web consiste em um **arquivo HTML de base** que inclui diversos objetos referenciados
- ❑ Cada objeto é endereçado através de uma **URL (Universal Resource Locator)**
- ❑ Exemplo de URL:

`www.someschool.edu/someDept/pic.gif`

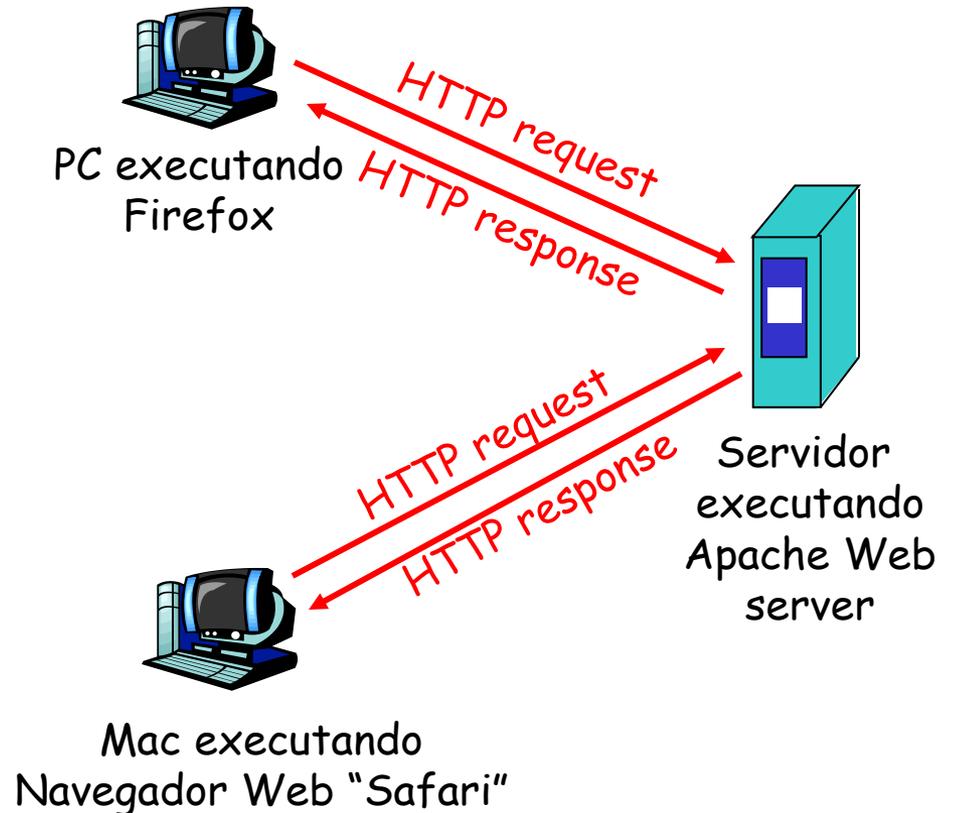
Nome do host

Nome do caminho

# Visão geral do HTTP

## HTTP: hypertext transfer protocol

- ❑ Protocolo da camada aplicação da Web
- ❑ Modelo cliente/servidor
  - ❖ *cliente*: browser que requisita, recebe, "mostra" objetos Web
  - ❖ *servidor*: servidor Web envia objetos em resposta a uma requisição
- ❑ HTTP 1.0: RFC 1945
- ❑ HTTP 1.1: RFC 2068



# Visão geral do HTTP (cont.)

## Usa TCP:

- ❑ cliente inicia uma conexão TCP (cria um socket) com servidor, porta 80
- ❑ servidor aceita a conexão TCP do cliente
- ❑ Mensagens HTTP (mensagens do protocolo da camada aplicação) trocadas entre o browser (cliente HTTP) e o servidor Web (servidor HTTP)
- ❑ Conexão TCP encerrada

## HTTP é "stateless" (sem estado)

- ❑ Servidor não mantém informações sobre requisições passadas de clientes

à parte

## Protocolos que mantêm "estado" são complexos!

- ❑ História passada (estado) tem que ser mantido
- ❑ se servidor/cliente "dá problema", a visão de "estado" do outro lado pode estar diferente (visões inconsistentes), necessidade de reconciliação de visões

# Conexões HTTP

## HTTP Não-persistente

- ❑ No máximo um objeto é enviado pela conexão TCP
- ❑ HTTP/1.0 usa HTTP não-persistente

## HTTP persistente

- ❑ Múltiplos objetos podem ser enviados em uma única conexão TCP entre cliente e servidor.
- ❑ HTTP/1.1 usa conexões persistentes como padrão

# HTTP Não-persistente

Suponha que usuário digite a seguinte URL

`www.someSchool.edu/someDepartment/home.index`

(contém texto,  
referência à 10  
imagens jpeg)

1a. Cliente HTTP inicia conexão TCP ao servidor HTTP (processo) em `www.someSchool.edu` na porta 80

1b. Servidor HTTP no host `www.someSchool.edu` aguardando conexão na porta 80. "aceita" conexão, notifica cliente

2. Cliente HTTP envia **mensagem de requisição** (contendo URL) pela conexão TCP (socket). Mensagem indica que cliente quer objeto `someDepartment/home.index`

3. Servidor HTTP recebe a mensagem de requisição, contrói uma **mensagem de resposta** contendo o objeto requisitado, envia a mensagem através de seu socket

tempo  
↓

# HTTP Não-persistente (cont.)

4. Servidor HTTP encerra a conexão TCP.



5. Cliente HTTP recebe a mensagem de resposta contendo o arquivo html, mostra o html. Tradutor html encontra 10 objetos referenciados

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

tempo



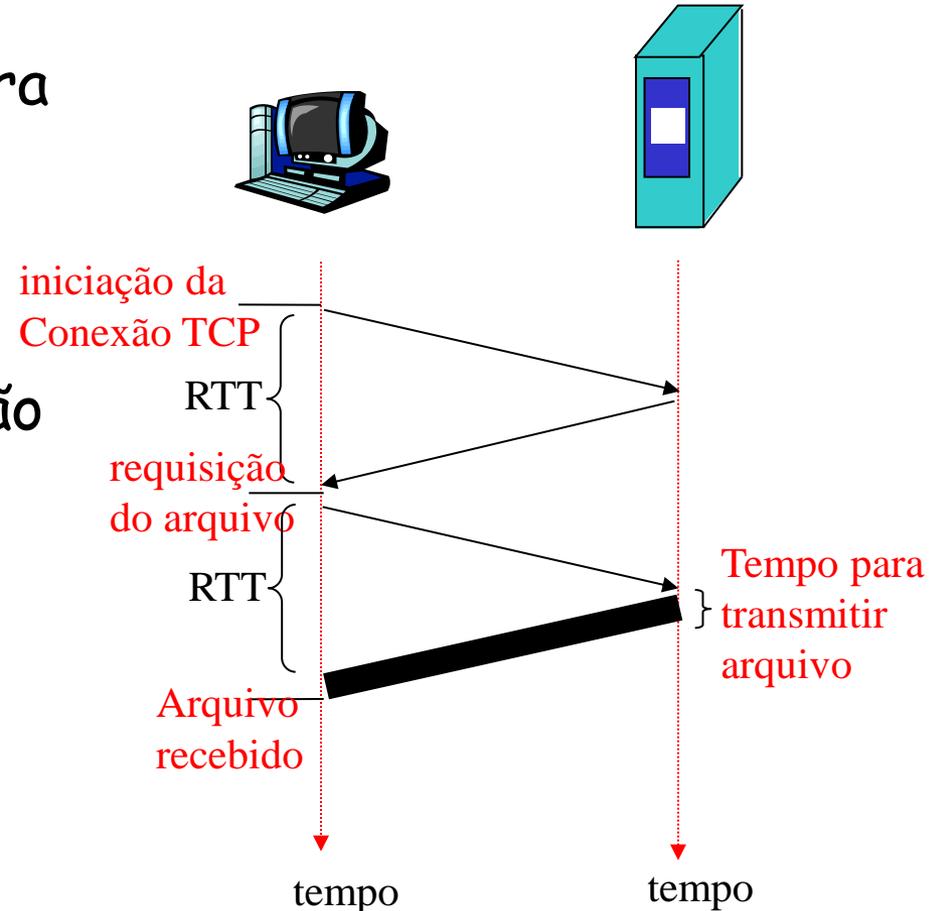
# HTTP não-persistente: Tempo de resposta

**Definição de RTT:** tempo que um pequeno pacote leva para ir do cliente ao servidor e voltar ao cliente.

## Tempo de resposta:

- ❑ 1 RTT para iniciar a conexão TCP
- ❑ 1 RTT para a requisição HTTP e primeiros poucos bytes da resposta HTTP retornar
- ❑ Tempo de transmissão do arquivo

**total = 2RTT + tempo para transmissão**



# HTTP persistente

## Problemas do HTTP não-persistente:

- ❑ requer 2 RTTs por objeto
- ❑ Overhead no SO para cada conexão TCP
- ❑ browsers frequentemente abrem conexões TCP paralelas para buscar objetos referenciados

## HTTP persistente

- ❑ Servidor mantém a conexão aberta após enviar resposta
- ❑ Mensagens HTTP subsequentes entre cliente/servidor enviadas através da conexão aberta

## Persistente sem pipelining:

- ❑ Cliente envia nova requisição somente quando a anterior tiver sido recebida
- ❑ 1 RTT para cada objeto referenciado

## Persistente com pipelining:

- ❑ Padrão no HTTP/1.1
- ❑ cliente envia requisições tão rápido quanto ele encontra objetos referenciados
- ❑ Tão baixo quanto 1 RTT para todos os objetos referenciados

# Mensagem de requisição HTTP

- ❑ 2 tipos de mensagens HTTP: *request, response*
- ❑ **Mensagem de requisição HTTP:**
  - ❖ ASCII (formato para leitura humana)

Linha de requisição  
(comandos GET,  
POST, HEAD)

linhas  
do cabeçalho

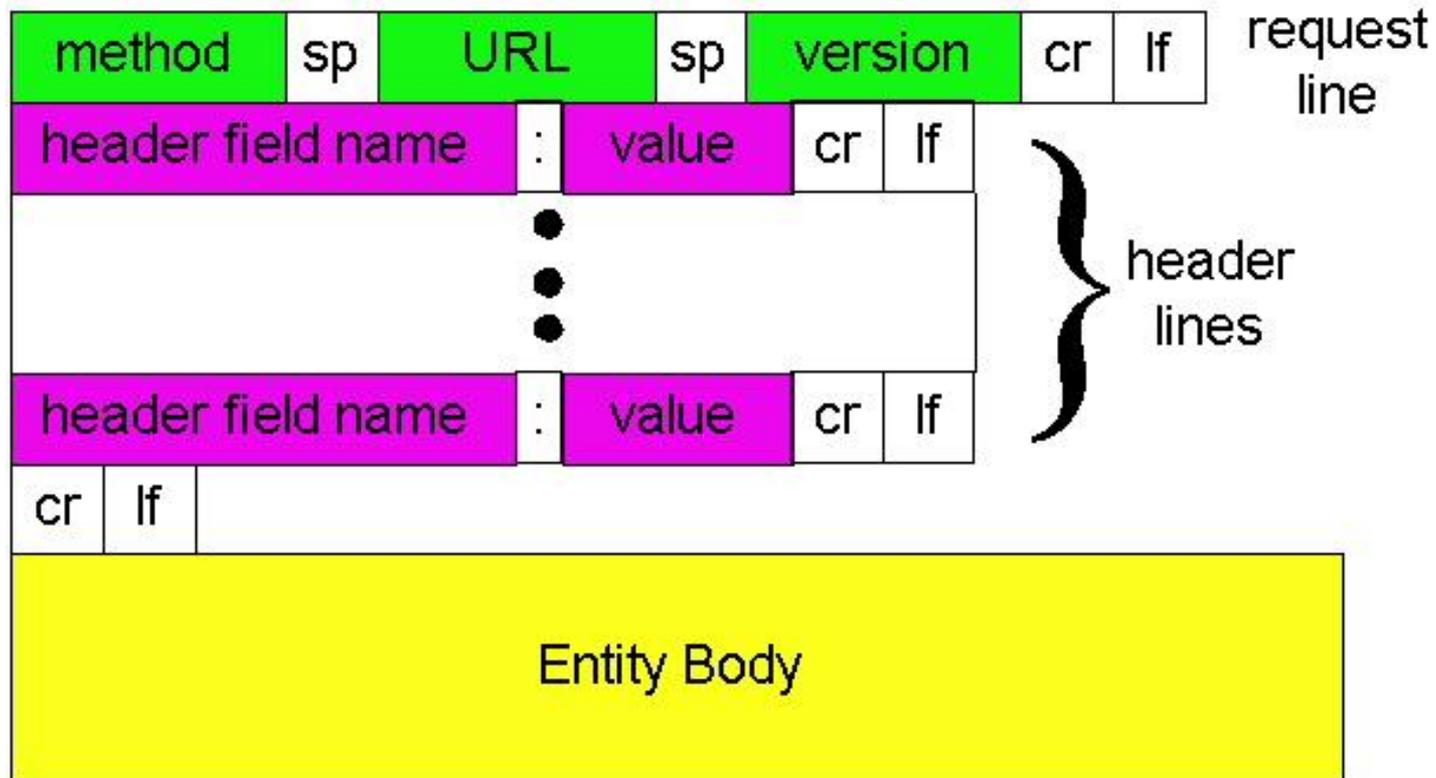
```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language: fr
```

"Carriage return,  
line feed"

indica fim de  
mensagem

(extra carriage return, line feed)

# Mensagem de requisição HTTP: formato geral



# Mais detalhes do método GET

## Propriedades:

- Seguro
  - ❖ GET não pode ser usado para produzir mudanças nos dados do servidor (e.g. atualização de BD)
- Idempotente
  - ❖ Múltiplas requisições ao mesmo recurso devem ter o mesmo resultado que teria uma requisição apenas
  - ❖ Há exceções: blogs ...

## Idempotente

Propriedade de um número que, multiplicado por ele mesmo, tem ele mesmo como resultado

$$(n \times n) = n$$

# Fazendo o upload do conteúdo de "forms"

## Método POST:

- ❑ Página Web frequentemente inclui "forms" (e.g. [www.google.com.br](http://www.google.com.br))
- ❑ Conteúdo é enviado ao servidor no campo "entity body"

## Método URL:

- ❑ Usa método GET
- ❑ Conteúdo do "form" é submetido no campo URL da linha de requisição:



`www.somesite.com/animalsearch?monkeys&banana`

# Tipos de Métodos

## HTTP/1.0

- ❑ GET
- ❑ POST
- ❑ HEAD
  - ❖ Servidor responde normalmente mas sem enviar o objeto requisitado na resposta

## HTTP/1.1

- ❑ GET, POST, HEAD
- ❑ PUT
  - ❖ Faz upload de arquivo no campo "entity body" para o caminho especificado no campo URL
- ❑ DELETE
  - ❖ Apaga arquivo especificado no campo URL

# Mensagem de resposta HTTP

Linha de status  
(código de status  
do protocolo  
Frase de status)

Linhas de cabeçalho

dados, e.g.,  
arquivo  
HTML requisitado

```
HTTP/1.1 200 OK
Connection close
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

dados dados dados dados dados ...
```

# Códigos de status de respostas

## HTTP

Na primeira linha da mensagem de resposta servidor->cliente.

Alguns exemplos:

### **200 OK**

- ❖ Requisição bem sucedida, objeto requisitado a seguir nesta mensagem

### **301 Moved Permanently**

- ❖ Objeto requisitado movido, nova localização especificada a seguir nesta mensagem (Location:)

### **400 Bad Request**

- ❖ Mensagem de requisição não compreendida pelo servidor

### **404 Not Found**

- ❖ Documento requisitado não foi encontrado neste servidor

### **505 HTTP Version Not Supported**

# Códigos de status de respostas

## HTTP

Em geral: código de status é composto por 3 dígitos, o primeiro indica a classe da mensagem

### **1xx: Informação**

- ❖ utilizada para enviar informações para o cliente de que sua requisição foi recebida e está sendo processada

### **2xx: Sucesso**

- ❖ indica que a requisição do cliente foi bem sucedida

### **3xx: Redirecionamento**

- ❖ informa a ação adicional que deve ser tomada para completar a requisição

### **4xx: Erro do Cliente**

- ❖ avisa que o cliente fez uma requisição que não pode ser atendida

### **5xx: Erro do Servidor**

- ❖ ocorreu um erro no servidor ao cumprir uma requisição válida

# Testando você mesmo o HTTP (lado cliente)

## 1. Telnet para o seu servidor Web favorito:

```
telnet cin.ufpe.br 80
```

Abre conexão TCP para a porta 80  
(porta padrão do servidor HTTP) do cin.ufpe.br  
Nada é enviado por enquanto

## 2. Digite uma requisição HTTP (GET):

```
GET /~seulogin/index.html HTTP/1.1  
Host: cin.ufpe.br
```

Digitando isto (hit carriage  
return 2 vezes), você envia  
esta requisição mínima (mas completa)  
GET ao servidor HTTP

## 3. Veja a resposta enviada pelo servidor HTTP!

# Vamos ver o HTTP em ação

- ❑ Exemplo telnet
- ❑ Exemplo Ethereal

Mais detalhes em aula prática ...

# Estado Usuário-Servidor: cookies

## (RFC 2109)

Muitos dos grandes Web sites usam cookies

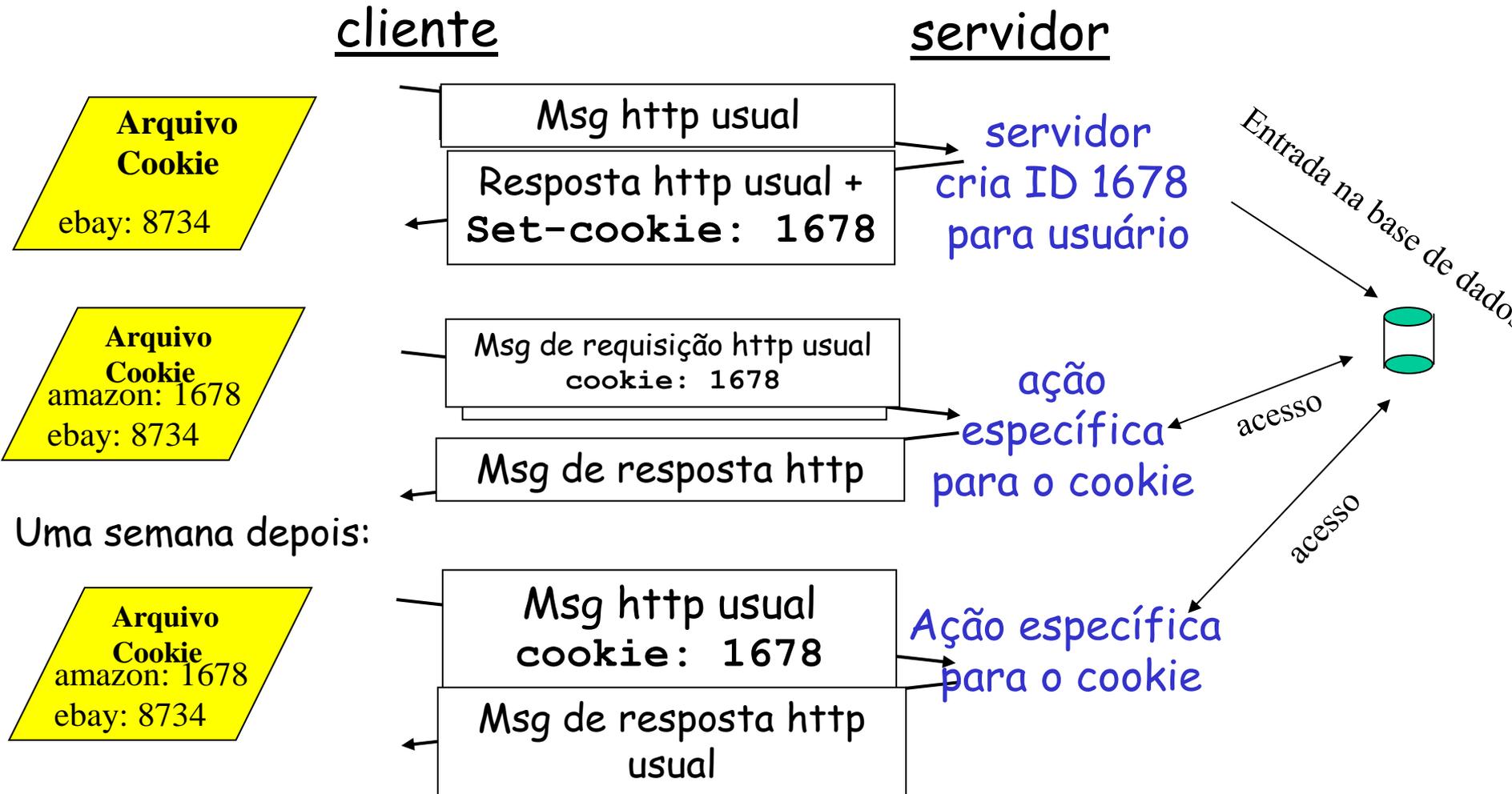
### 4 componentes:

- 1) Cabeçalho relacionado ao cookie na mensagem de resposta HTTP
- 2) Cabeçalho relacionado ao cookie na mensagem de requisição HTTP
- 3) Arquivo cookie mantido no computador do usuário e gerenciado pelo browser no computador do usuário
- 4) Base de dados no site Web

### Exemplo:

- ❖ Susan acessa a Internet sempre do mesmo PC
- ❖ Ela visita um site específico de e-commerce pela primeira vez
- ❖ Quando a requisição HTTP inicial chega ao site, o mesmo cria um ID único e cria uma entrada em sua base de dados

# Cookies: mantendo "estado" (cont.)



# Cookies (cont.)

## Usos do cookie:

- ❑ autorização
- ❑ Carrinho de compras
- ❑ Recomendações/preferências
- ❑ Estado da sessão do usuário (Webmail)

à parte

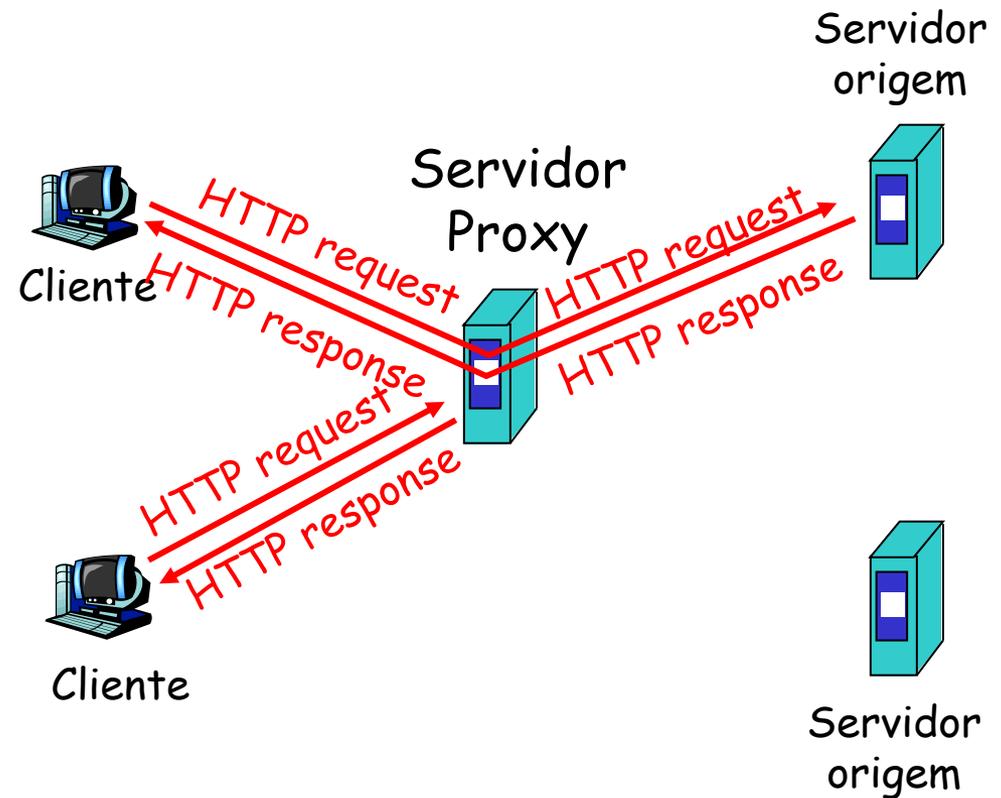
## Cookies e privacidade:

- ❑ cookies permitem que sites aprendam bastante sobre você
- ❑ Você pode fornecer nome e e-mail para os sites
- ❑ Ferramentas de busca usam redirecionamento e cookies para aprenderem ainda mais
- ❑ Empresas de "publicidade" obtêm informações através de sites

# Web caches (servidor proxy)

**Objetivo:** satisfazer a requisição do cliente sem envolver servidor origem

- usuário seta no browser: acesso Web via proxy
- browser envia todas as requisições HTTP para o proxy
  - ❖ Objeto no cache: proxy retorna objeto
  - ❖ senão proxy requisita objeto do servidor origem, e em seguida retorna objeto ao cliente



# Mais sobre Web caching

- ❑ Proxy atua tanto como cliente como servidor
- ❑ Tipicamente o proxy é instalado pelo ISP (universidade, empresa, ISP residencial)

## Por quê Web caching?

- ❑ Reduzir tempo de resposta para o cliente.
- ❑ Reduzir tráfego no enlace de acesso das instituições.
- ❑ Internet densa com proxies permite servidores "pobres" entregar conteúdo efetivamente (mas faz compartilhamento de arquivo P2P)

# Exemplo de Caching:

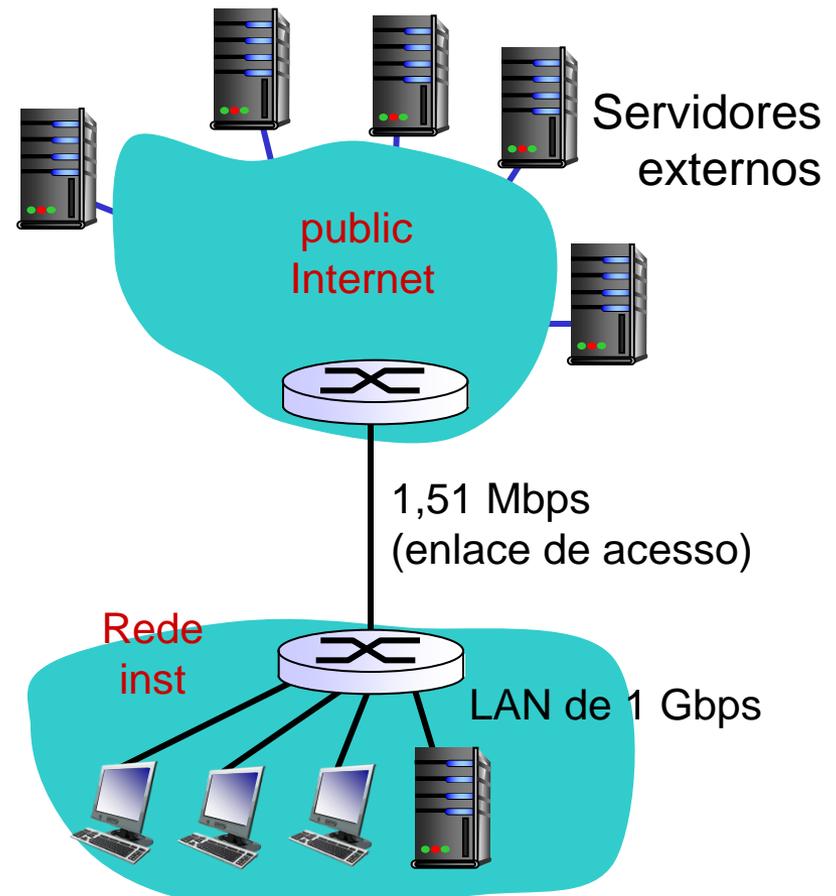
## Assume-se:

- ❖ Tam. Médio objeto: 100K bits
- ❖ Taxa média de requisição dos browsers para os servidores externos: 15/segundo
- ❖ Taxa média de dados para os browsers: 1,50 Mbps
- ❖ RTT do roteador de saída para Internet para qualquer servidor externo: 2 segundos
- ❖ BW do enlace de acesso: 1,51 Mbps

## consequências:

- ❖ Utilização da LAN: 0,15%
- ❖ Utilização do enlace de acesso = **99%**
- ❖ Atraso total = atraso Internet + atraso acesso + atraso LAN  
= 2 segundos + minutos + microsegundos

*problema!*



# Exemplo de Caching: 1 solução?

## Assume-se:

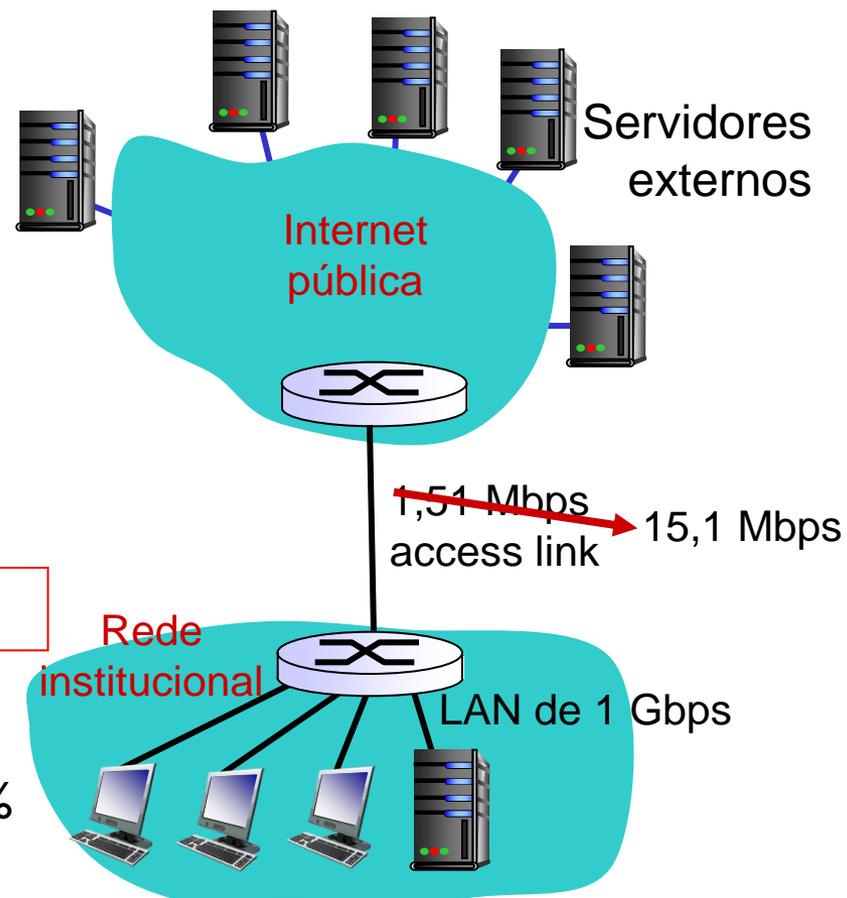
- ❖ Tam. Médio objeto: 100K bits
- ❖ Taxa média de requisição dos browsers para os servidores externos: 15/segundo
- ❖ Taxa média de dados para os browsers: 1,50 Mbps
- ❖ RTT do roteador de saída para Internet para qualquer servidor externo: 2 segundos
- ❖ BW do enlace de acesso: 1,51 Mbps

## conseqüências:

- ❖ Utilização da LAN: 0,15%
- ❖ Utilização do enlace de acesso = 99%
- ❖ Atraso total = atraso Internet + atraso acesso + atraso LAN

= 2 segundos + minutos + microsegundos

msecs



**Custo:** aumentar BW do enlace de acesso (não é barato!) Camada Aplicação

# Exemplo de Caching: instalação de cache local

## Assume-se:

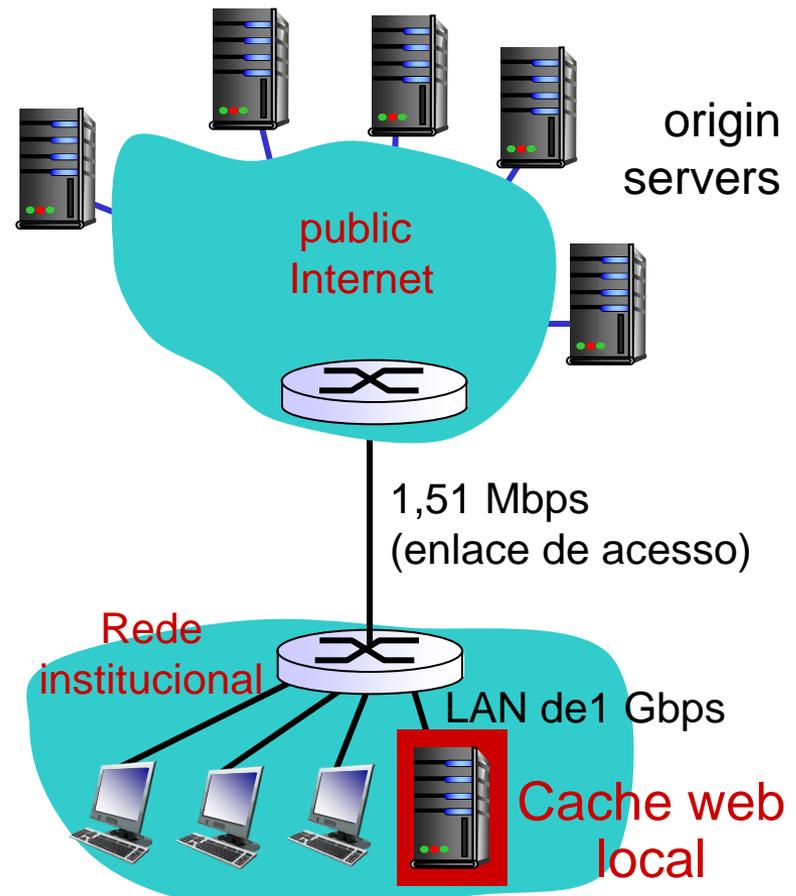
- ❖ Tam. Médio objeto: 100K bits
- ❖ Taxa média de requisição dos browsers para os servidores externos: 15/segundo
- ❖ Taxa média de dados para os browsers: 1,50 Mbps
- ❖ RTT do roteador de saída para Internet para qualquer servidor externo: 2 segundos
- ❖ BW do enlace de acesso: 1,51 Mbps

## consequências:

- ❖ Utilização da LAN: ?
- ❖ Utilização do enlace de acesso: ?

*Como calcular utilização do enlace de acesso e atraso?*

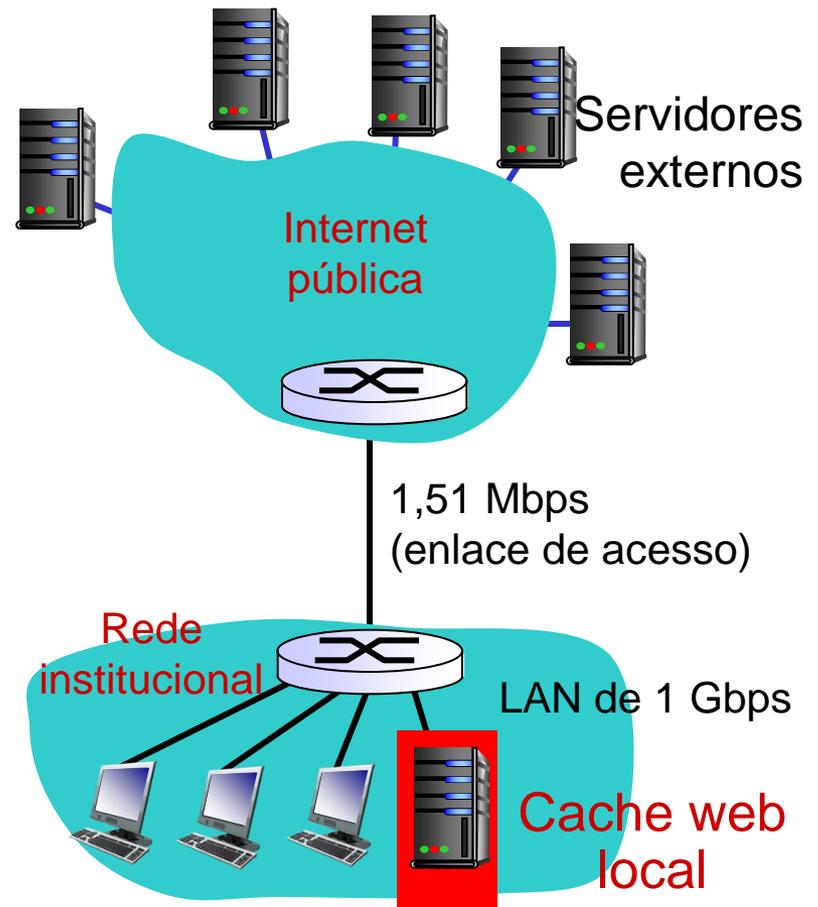
*Custo: cache web (barato!)*



# Exemplo de Caching: instalação de cache local

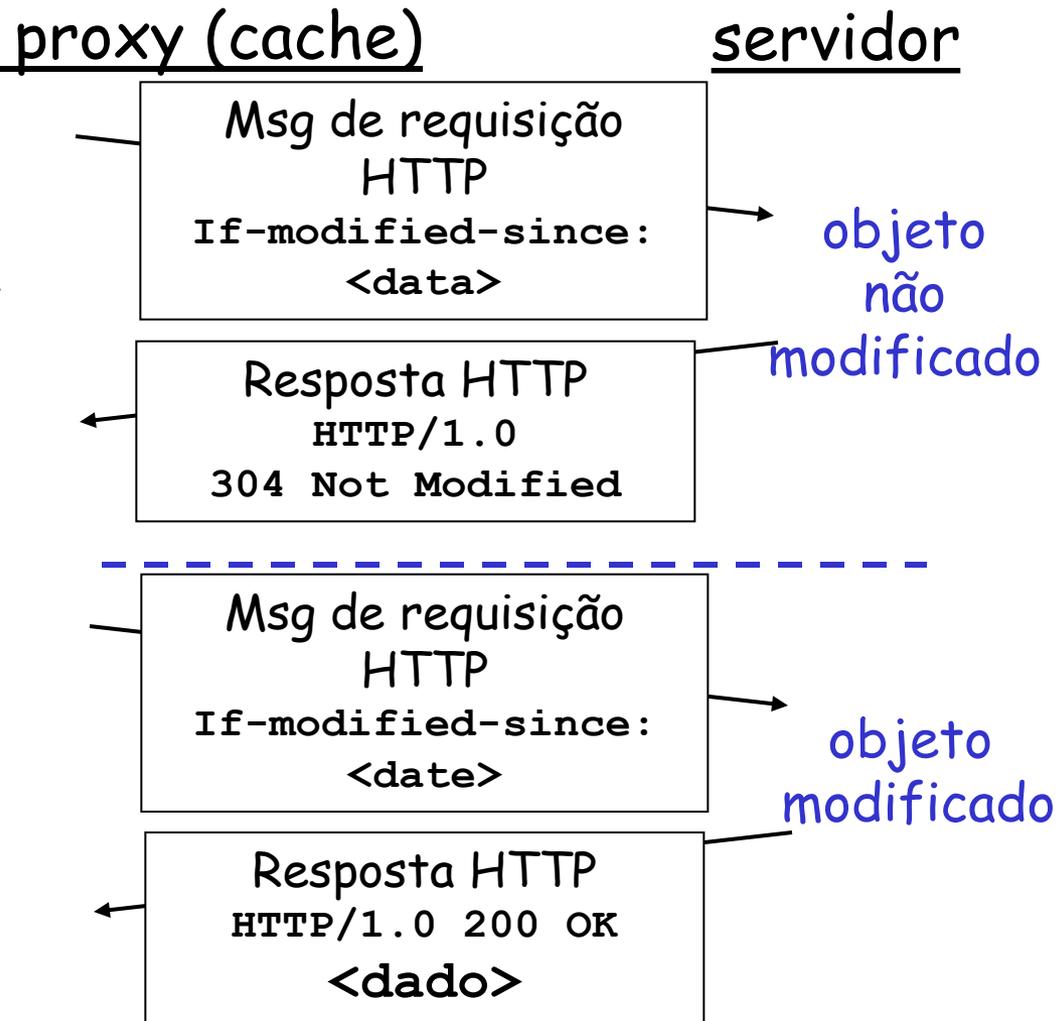
## Calculando ....

- Suponha taxa de acerto de 0.4
  - ❖ 40% das requisições satisfeitas pela cache, 60% das requisições satisfeitas pelos servidores externos
- ❖ Utilização do enlace de acesso:
  - 60% das requisições usam o enlace de acesso
- ❖ Taxa de dados para os browsers (dados pelo enlace de acesso) =  $0,6 * 1,50 \text{ Mbps} = 0,9 \text{ Mbps}$ 
  - utilization =  $0,9 / 1,51 = 0,59$
- ❖ Atraso total
  - =  $0,6 * (\text{atraso servidores ext.} \rightarrow \text{rede inst.}) + 0,4 * (\text{atraso quando req. satisfeitas pela cache})$
  - =  $0,6 (2,01) + 0,4 (\sim \text{msecs})$
  - =  $\sim 1,2 \text{ segundos}$
  - Menos tempo do que quando tínhamos um enlace de acesso de 151 Mbps (e mais barato!)



# GET Condicional

- ❑ **Objetivo:** não envie objeto se cache possui uma versão atualizada
- ❑ Proxy(cache): especifica data da cópia na requisição HTTP  
If-modified-since:  
<data>
- ❑ servidor: resposta contém nenhum objeto se a cópia no proxy (cache) está em dia:  
HTTP/1.0 304 Not Modified



# Parênteses: Já temos HTTP 2.0

(não coberto)

- ❑ Principais diferenças para HTTP 1.x (grande mudança de paradigma)
  - ❖ Binário em vez de texto (adeus telnet)
  - ❖ Paralelismo e multiplexação - tudo é enviado multiplexado em uma única conexão
  - ❖ Servidor envia respostas proativamente para cliente guardar na cache
  - ❖ Compressão de cabeçalho para reduzir overhead

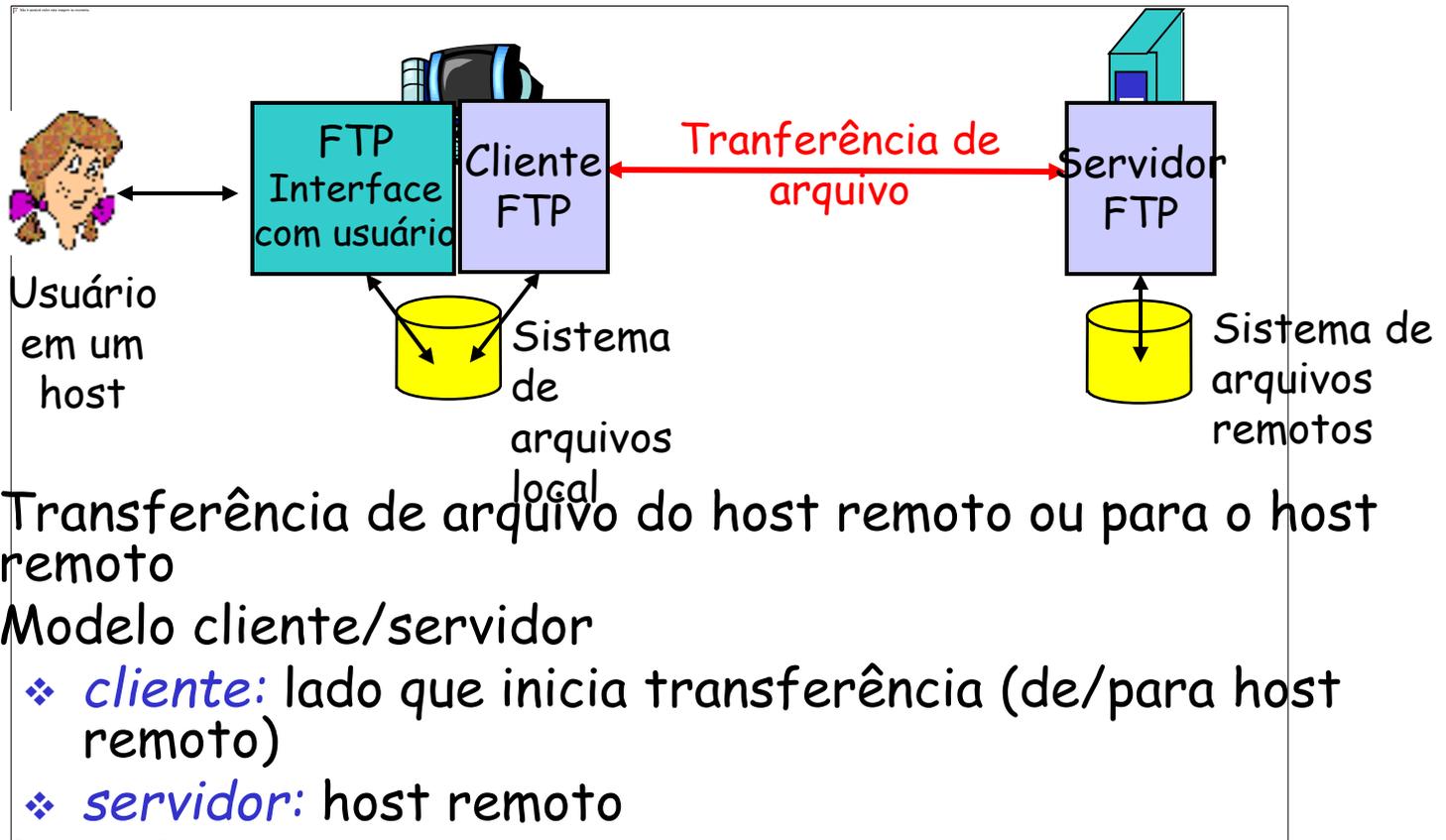
## ❑ Demo

- ❖ Veja o que acontece na primeira vez que acessa o link abaixo:
- ❖ <https://http2.akamai.com/demo>
- ❖ Na segunda vez que acessar, o seu browser terá feito caching e o download parecerá mais rápido

# Módulo 2: Camada Aplicação

- ❑ 2.1 Princípios das aplicações de rede
- ❑ 2.2 Web e HTTP
- ❑ 2.3 FTP
- ❑ 2.4 e-mail (Electronic Mail)
  - ❖ SMTP, POP3, IMAP
- ❑ 2.5 DNS
- ❑ 2.6 compartilhamento de arquivos (P2P)
- ❑ 2.7 Programação de Sockets com TCP
- ❑ 2.8 Programação de Socket com UDP
- ❑ 2.9 Construindo um servidor Web

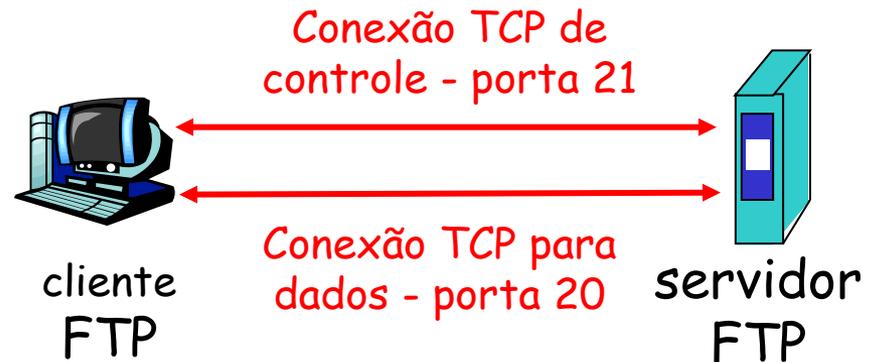
# FTP: Protocolo de Transferência de Arquivos



- ❑ Transferência de arquivo do host remoto ou para o host remoto
- ❑ Modelo cliente/servidor
  - ❖ *cliente*: lado que inicia transferência (de/para host remoto)
  - ❖ *servidor*: host remoto
- ❑ ftp: RFC 959
- ❑ Servidor ftp: porta 21

# FTP: controle e dados separados por conexões distintas

- ❑ Cliente FTP contacta servidor na porta 21, especificando o TCP como protocolo de transporte
- ❑ Cliente obtém autorização sobre conexão de controle
- ❑ Cliente navega pelo diretório remoto através do envio de comandos pela conexão de controle.
- ❑ Quando servidor recebe um comando para uma transferência de arquivo, o servidor abre uma conexão TCP de dados para o cliente
- ❑ Após transferir um arquivo, servidor fecha a conexão.



- ❑ Servidor abre uma segunda conexão TCP de dados para transferir outro arquivo.
- ❑ Conexão de controle: "for a de banda - out of band"
- ❑ Servidor FTP server mantém "estado": diretório atual, autenticação anterior

# FTP: comandos, respostas

## Exemplo de comandos:

- ❑ Enviados como texto ASCII através da canal de controle
- ❑ `USER username`
- ❑ `PASS password`
- ❑ `LIST` retorna a lista de arquivos no diretório atual
- ❑ `RETR filename "pega"` (get) arquivo
- ❑ `STOR filename` armazena arquivo no host remoto (put)

## Exemplo 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

# Módulo 2: Camada Aplicação

- ❑ 2.1 Princípios das aplicações de rede
- ❑ 2.2 Web e HTTP
- ❑ 2.3 FTP
- ❑ 2.4 e-mail (Electronic Mail)
  - ❖ SMTP, POP3, IMAP
- ❑ 2.5 DNS
- ❑ 2.6 compartilhamento de arquivos (P2P)
- ❑ 2.7 Programação de Sockets com TCP
- ❑ 2.8 Programação de Socket com UDP
- ❑ 2.9 Construindo um servidor Web

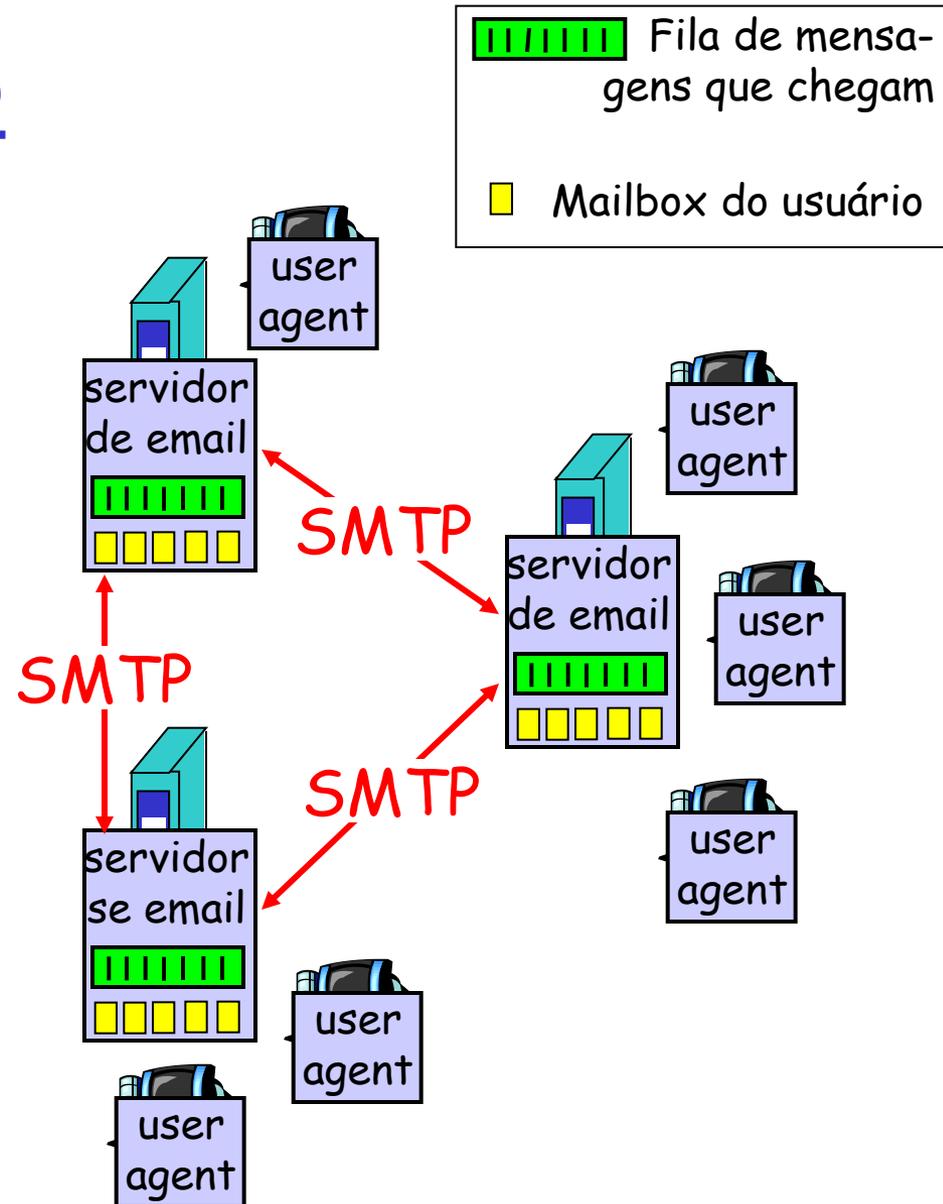
# Correio Eletrônico

## 3 componentes principais:

- ❑ Agentes de usuários (**user agents**)
- ❑ Servidor de email
- ❑ simple mail transfer protocol: SMTP

### Agente de Usuário

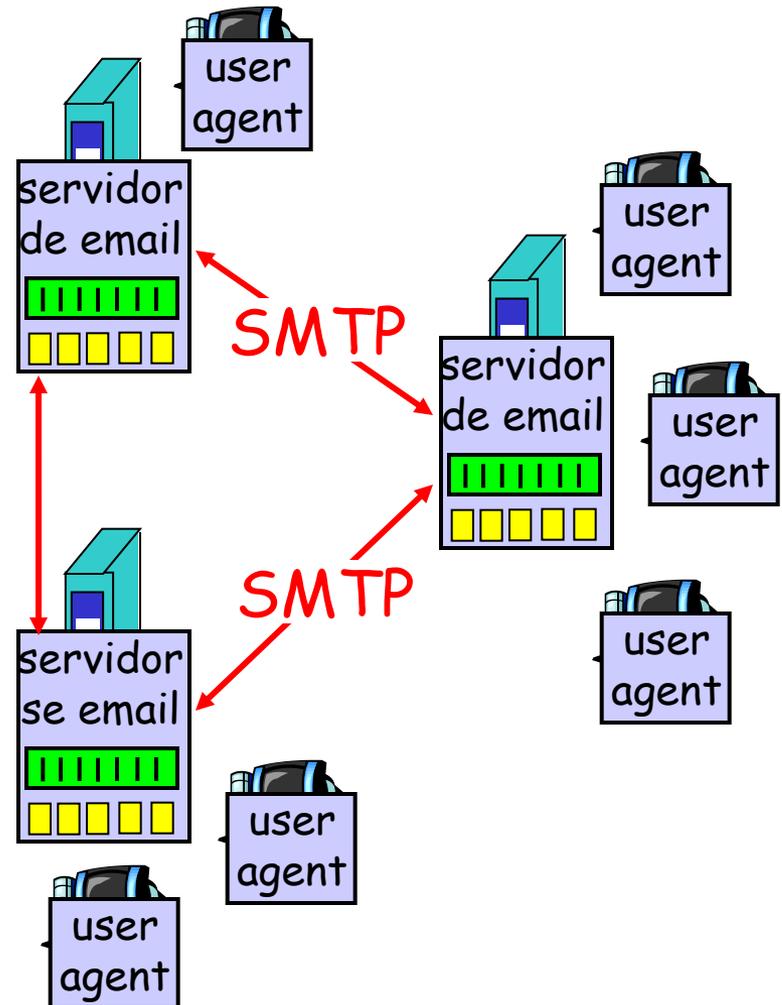
- ❑ = "programa de email"
- ❑ compor, editar, ler mensagens de email
- ❑ e.g., Eudora, Outlook, elm, Netscape Messenger
- ❑ Mensagens que "chegam" ou "devem sair" armazenadas no servidor



# Correio Eletrônico: servidores de email

## Servidores de Email

- ❑ **mailbox** contém mensagens que chegam para o usuário
- ❑ **Fila de mensagens** para mensagens a serem enviadas
- ❑ **protocolo SMTP** entre servidores de email para envio das mensagens
  - ❖ cliente: servidor de email "emissor"
  - ❖ "servidor": servidor de email "receptor"

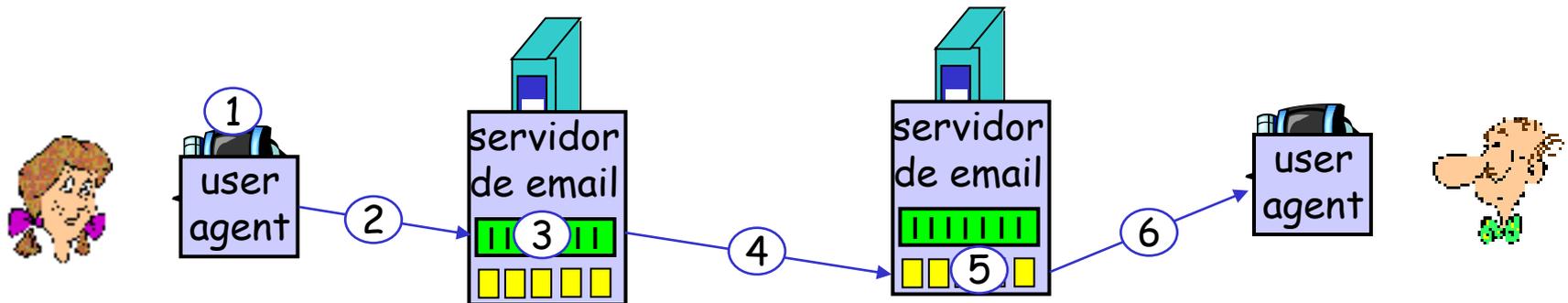


# Correio Eletrônico: SMTP [RFC 2821]

- ❑ usa TCP para transferência confiável de mensagem de email do cliente para o servidor, porta 25
- ❑ Transferência direta: servidor "emissor" para servidor "receptor"
- ❑ 3 fases de transferência
  - ❖ handshaking (cumprimento)
  - ❖ Transferência de mensagens
  - ❖ fechamento
- ❑ Interação comando/resposta
  - ❖ **comandos:** ASCII text
  - ❖ **resposta:** código de status e frase
- ❑ mensagens devem ser em ASCII 7-bits

# cenário: Alice envia mensagem à Bob

- 1) Alice usa programa de email para escrever mensagem e no "para" insere bob@someschool.edu
- 2) UA da Alice envia mensagem para o seu servidor de email; mensagem colocada na fila de mensagens
- 3) Lado cliente do SMTP abre conexão TCP com servidor de email do Bob
- 4) Cliente SMTP envia mensagem da Alice através da conexão TCP
- 5) Servidor de email do Bob coloca a mensagem no mailbox do Bob
- 6) Bob abre o seu programa de email para ler mensagem



# 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: Vamos pra balada?
C: Por volta das 23hs?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

# Tente você mesmo uma interação SMTP:

- ❑ `telnet servername 25`
- ❑ Veja resposta 220 do servidor
- ❑ Entre com os comandos HELO, MAIL FROM, RCPT TO, DATA, QUIT

Permite envio de email sem usar um programa de email! Mais na aula prática ...

# SMTP: últimas palavras

- ❑ SMTP usa conexões persistentes
- ❑ SMTP requer mensagem (cabeçalho & corpo) em ASCII 7-bits
- ❑ Servidor SMTP usa CRLF.CRLF para determinar o final da mensagem

## Comparação com HTTP:

- ❑ HTTP: "pull - puxa"
- ❑ SMTP: "push - empurra"
- ❑ Ambos possuem comandos/respostas de interação ASCII, código de status
- ❑ HTTP: cada objeto é encapsulado em sua própria mensagem de resposta
- ❑ SMTP: múltiplos objetos enviados em mensagem "multi-parte"

# Formato da mensagem de email

SMTP: protocolo para troca de mensagens

RFC 822: padrão para mensagem no formato texto:

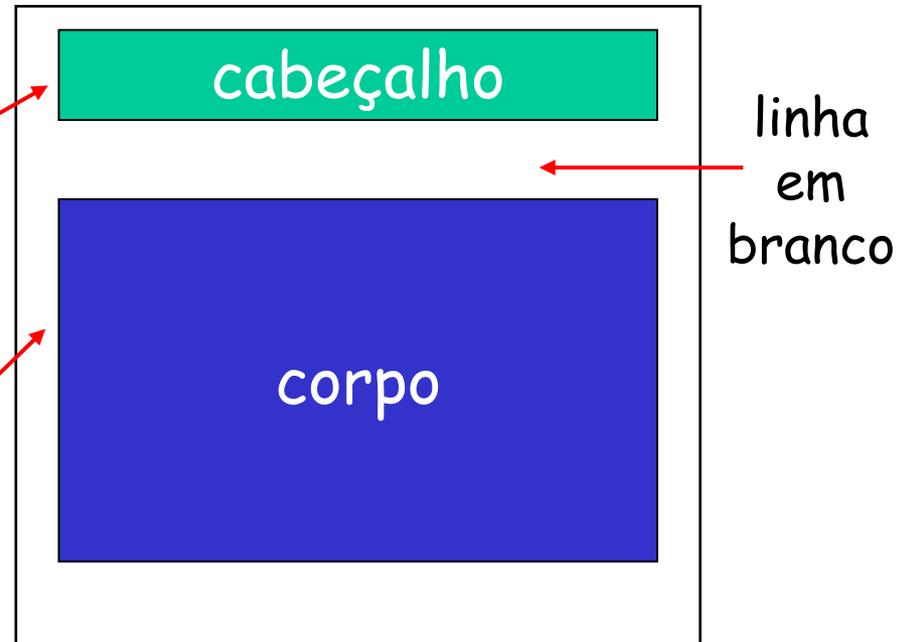
□ Linhas de cabeçalho, e.g.,

- ❖ To:
- ❖ From:
- ❖ Subject:

*diferente de comandos SMTP!*

□ corpo

- ❖ a "mensagem", caracteres ASCII somente



# Formato da mensagem: extensões multimídia

- ❑ MIME: multimedia mail extension, RFC 2045, 2056
- ❑ Msgs adicionais no cabeçalho declaram conteúdo do tipo MIME

Versão do MIME

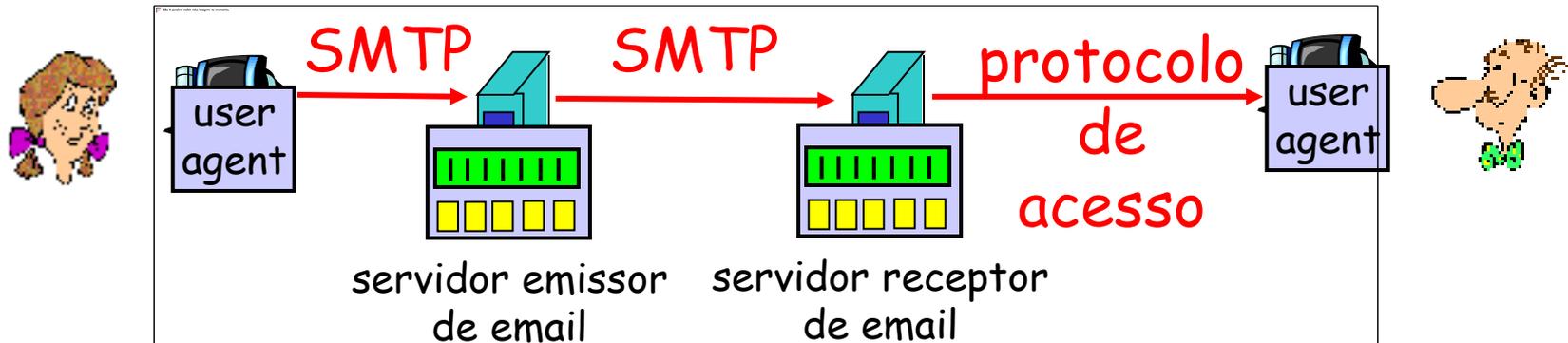
Método usado para codificar dados

Tipo/subtipo de dado multimedia,

declaração de parâmetros (este último opcional)  
dado codificado

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

# Protocolos de Acesso ao Correio Eletrônico



- ❑ SMTP: entrega/armazenamento (servidor receptor)
- ❑ Protocolo de acesso ao correio eletrônico: recuperação de msg do servidor
  - ❖ POP: Post Office Protocol [RFC 1939]
    - autorização (agente <-->servidor) e download
  - ❖ IMAP: Internet Mail Access Protocol [RFC 1730]
    - mais possibilidades (mais complexo)
    - manipulação de msgs armazenadas no servidor
  - ❖ HTTP (Webmail): Hotmail , Yahoo! Mail, etc.

# Protocolo POP3 (RFC 1939)

## fase de autorização

- ❑ comandos do cliente:
  - ❖ user: username
  - ❖ pass: password
- ❑ respostas do servidor
  - ❖ +OK
  - ❖ -ERR

## fase de transação, cliente:

- ❑ list: lista números de msgs
- ❑ retr: recupera msg pelo número
- ❑ dele: deleta
- ❑ quit

```
S: +OK POP3 server ready
C: user bob
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
```

# POP3 (more) and IMAP

## Mais sobre POP3

- ❑ Exemplo anterior usa modo "download e delete".
- ❑ Bob não pode ler email se ele muda de cliente ou computador
- ❑ "Download e mantenha": copias das msgs em diferentes clientes
- ❑ POP3 não informa estado através das sessões
  - ❖ POP3 é "stateless"
  - ❖ Mas servidor mantém estado para saber quais msgs apagar!

## IMAP (Internet Mail Access Protocol - RFC 2060)

- ❑ Mantém todas as msgs em um lugar: no servidor
- ❑ Permite usuários organizarem msgs em pastas
- ❑ IMAP mantém "estado" do usuário através de sessões:
  - ❖ Nome de pastas e mapeamento entre Ids de msgs e nome de pasta

# Módulo 2: Camada Aplicação

- ❑ 2.1 Princípios das aplicações de rede
- ❑ 2.2 Web e HTTP
- ❑ 2.3 FTP
- ❑ 2.4 e-mail (Electronic Mail)
  - ❖ SMTP, POP3, IMAP
- ❑ 2.5 DNS
- ❑ 2.6 compartilhamento de arquivos (P2P)
- ❑ 2.7 Programação de Sockets com TCP
- ❑ 2.8 Programação de Socket com UDP
- ❑ 2.9 Construindo um servidor Web

# DNS: Domain Name System

**Pessoas:** muitos identificadores:

- ❖ RG, CPF, nome, passaporte #

**Hosts e roteadores Internet:**

- ❖ Endereço IP (32 bits) - usado para endereçar datagramas
- ❖ "nome", e.g., www.yahoo.com - usado por humanos

**Q:** como é o mapeamento entre endereços IP e nomes ?

**Domain Name System:**

- *base de dados distribuída* implementada de forma hierárquica com muitos servidores de nome
- *Protocolo da camada aplicação* que permite hosts e servidores de nome se comunicarem para *resolução* de endereços (tradução endereço/nome)
  - ❖ nota: função do núcleo da Internet implementada como protocolo da camada aplicação
  - ❖ complexidade nas "bordas" da rede

# DNS (RFC 1034, RFC 1035 e outras)

## Serviços do DNS

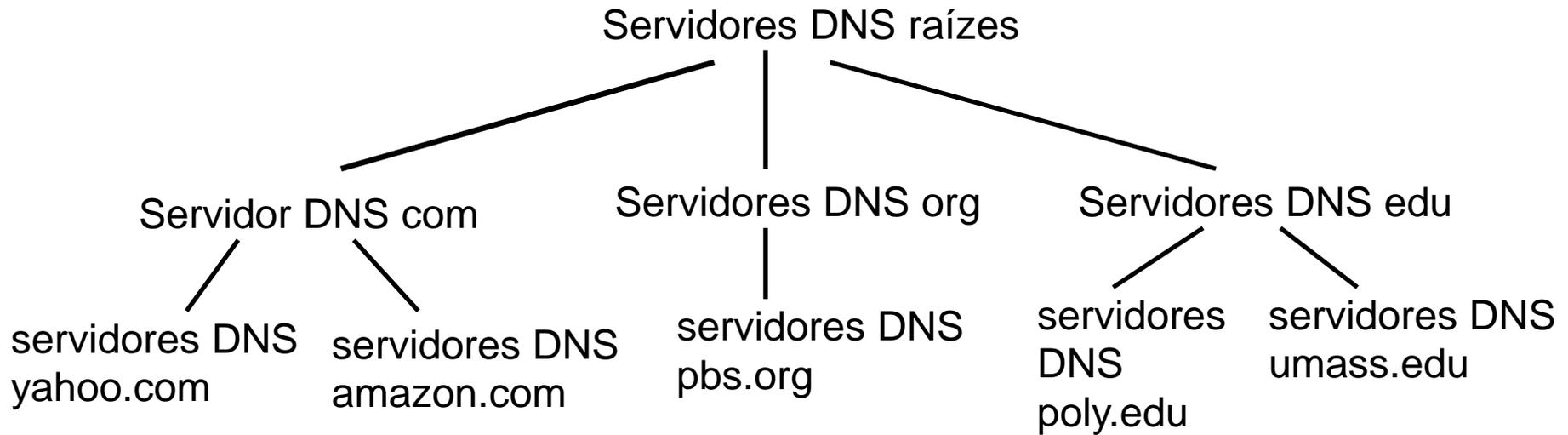
- ❑ Tradução do nome do host para endereço IP
- ❑ Host aliasing
  - ❖ Nome canônico e alternativo
- ❑ Mail server aliasing
- ❑ Distribuição de carga
  - ❖ Replicação de servidores: conjunto de endereços IP para um nome canônico

## Por que não centralizar o DNS?

- ❑ ponto único de falha
- ❑ volume de tráfego
- ❑ Base de dados centralizada distante
- ❑ manutenção

não escala!

# Base de Dados Distribuída e Hierárquica

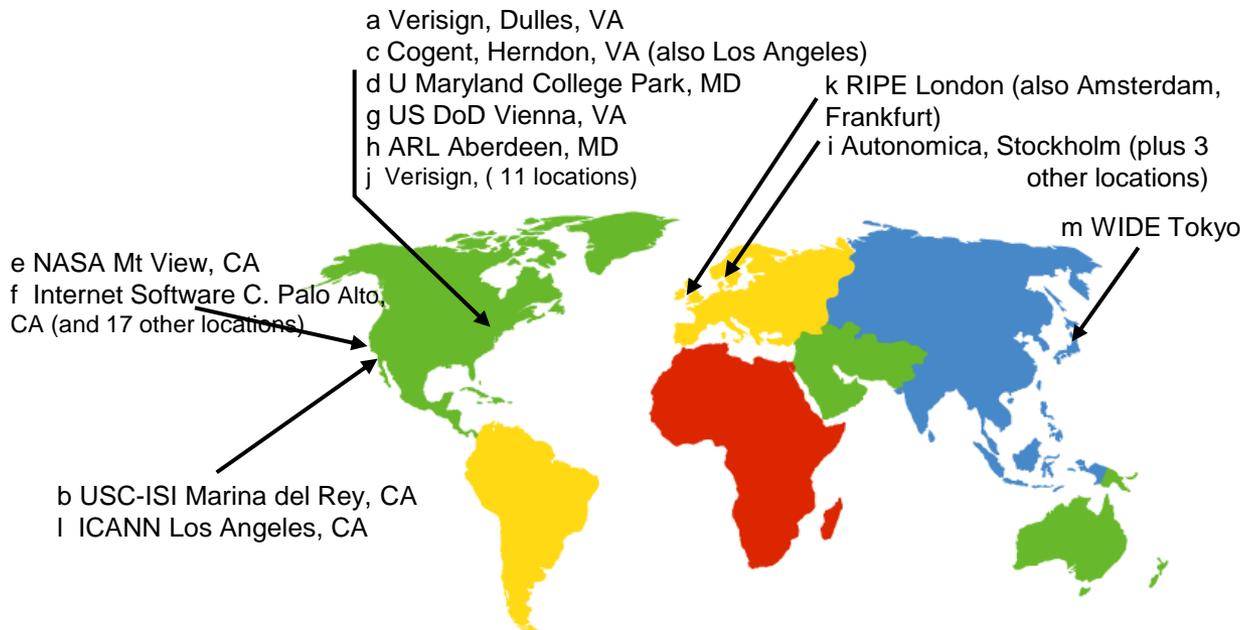


## Cliente quer IP de [www.amazon.com](http://www.amazon.com); 1ª abordagem:

- ❑ Cliente indaga um servidor raiz para encontrar servidor DNS "com"
- ❑ Cliente indaga Servidor DNS "com" para "obter" servidor DNS "amazon.com"
- ❑ Cliente indaga servidor DNS "amazon.com" para "obter" endereço IP de [www.amazon.com](http://www.amazon.com)

# DNS: servidores de nome raiz

- ❑ Contactado pelo servidor local de nome que não pode resolver nome
- ❑ Servidor de nome raiz:
  - ❖ contacta servidor de nome "autorizado" se não conhece mapeamento de nome
  - ❖ Recebe mapeamento
  - ❖ retorna mapeamento para o servidor local de nomes



13 servidores de nome raízes no mundo

# Servidores TLD e Autorizados

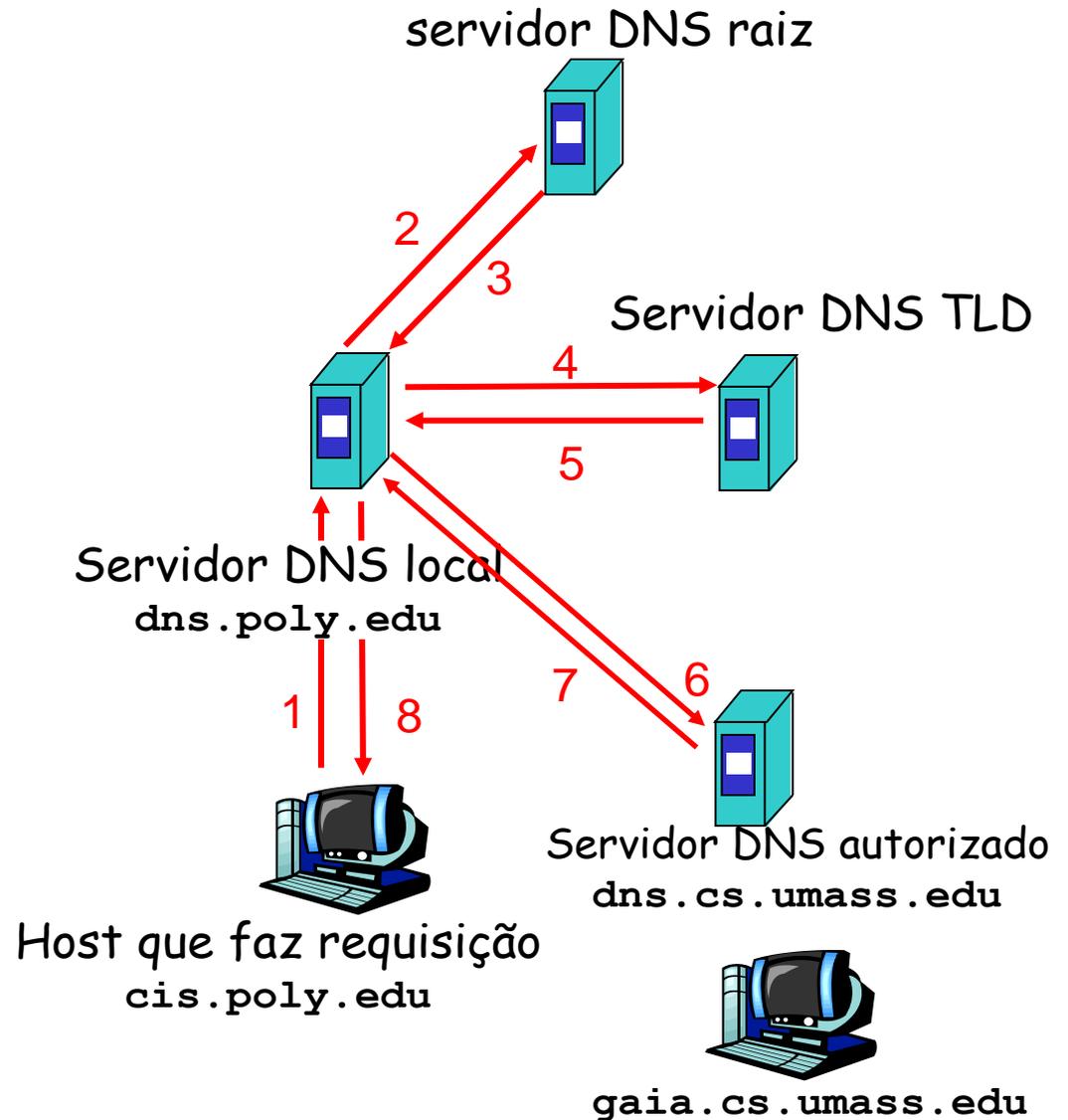
- ❑ **Servidores Top-level domain (TLD):** responsáveis por com, org, net, edu, etc, e todos os domínios de topo dos países br, uk, fr, ca, jp ...
  - ❖ Network solutions mantém servidores "com TLD"
  - ❖ Educause mantém servidores "edu TLD"
- ❑ **Servidores DNS autorizados:** servidores DNS das organizações, provendo hostnames autorizados para mapeamentos de IP para servidores de organizações (e.g. Web e email).
  - ❖ Pode ser mantido pela organização ou provedor (ISP)

# Servidor Local de Nome

- ❑ Não pertence necessariamente a uma hierarquia
- ❑ Cada ISP (ISP residencial, empresa, universidade) possui um.
  - ❖ Também chamado de "default name server"
- ❑ Quando um host faz um pedido DNS, o pedido é enviado ao seu servidor DNS local
  - ❖ Atua como um proxy, encaminha pedido na hierarquia.

# Exemplo

- Host em cis.poly.edu deseja endereço IP de gaia.cs.umass.edu



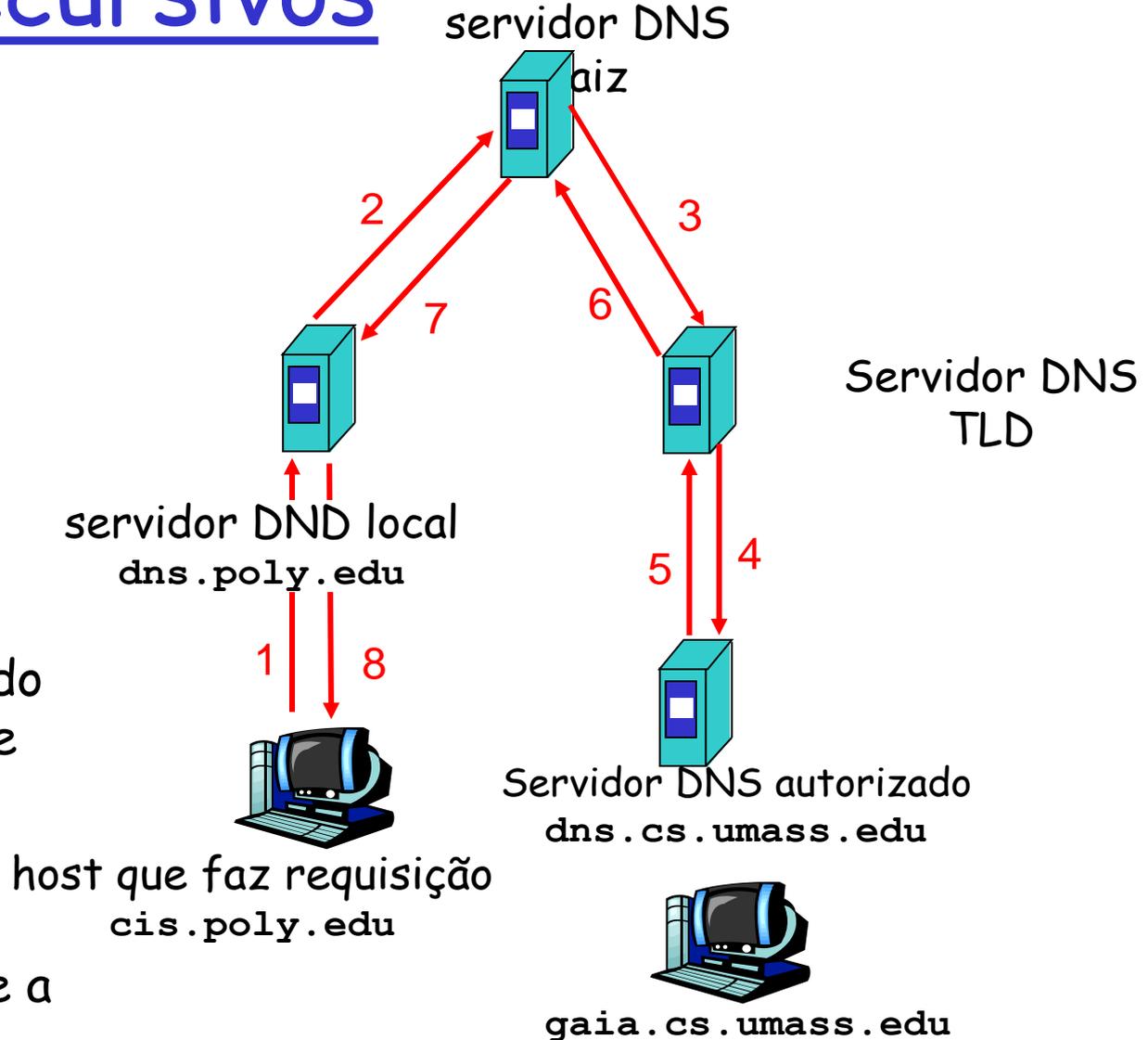
# Pedidos recursivos

## Pedido recursivo:

- ❑ Onera servidor contactado para a resolução de nome
- ❑ sobrecarga?

## Pedido iterado:

- ❑ Servidor contactado responde com nome do servidor a ser contactado
- ❑ "Não conheço este nome mas pergunte a este servidor"



# DNS: caching e atualização de registros (records)

- Assim que servidor de nome "aprende" mapeamento, ele o guarda no **cache**
  - ❖ Entradas no cache expiram (desaparecem) após algum tempo
  - ❖ Servidores TLD tipicamente "armazenados" nos servidores locais de nome
    - Deste modo servidores de nome raízes não são freqüentemente visitados
- Mecanismos de atualização/modificação em desenvolvimento pelo IETF
  - ❖ RFC 2136
  - ❖ <http://www.ietf.org/html.charters/dnsind-charter.html>

# Registros DNS

DNS: bd distribuída que armazena registros de recurso (RR)

RR formato: (nome, valor, tipo, ttl)

## □ Tipo=A

- ❖ nome é hostname
- ❖ valor é endereço IP

## □ Tipo=NS

- ❖ nome é domínio (e.g. foo.com)
- ❖ valor é hostname do servidor de nome autorizado para este domínio

## □ Tipo=CNAME

- ❖ name é "nome alternativo" para algum nome canônico (o real)

www.ibm.com é na realidade  
servereast.backup2.ibm.com

- ❖ valor é o nome canônico

## □ Tipo=MX

- ❖ valor é nome do servidor de email associado com nome

# Protocolo DNS, mensagens

Protocolo DNS: mensagens *query* e *reply*, ambas com o mesmo *formato*

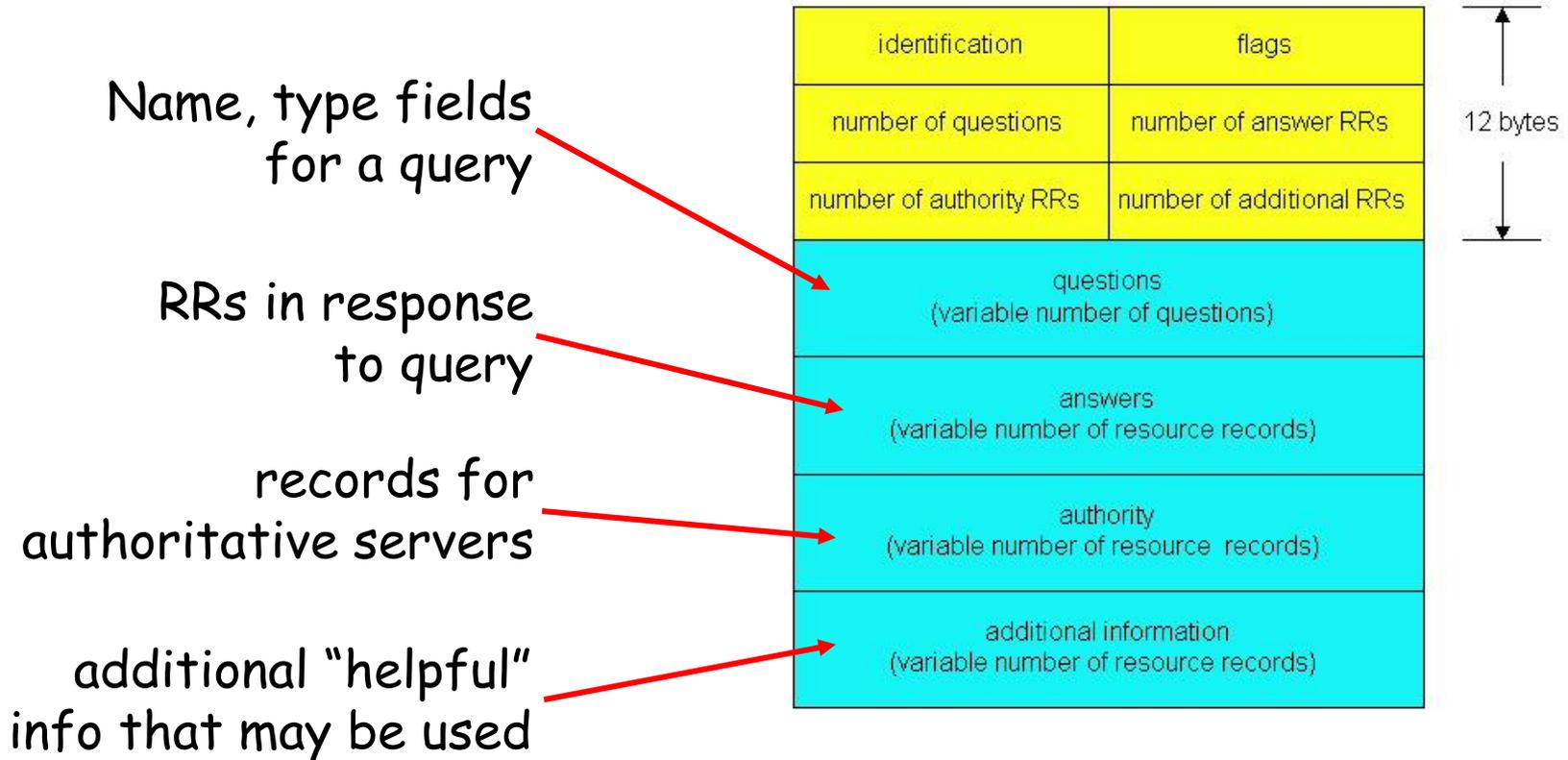
## Cabeçalho da msg

- ❑ **identificação**: # de 16 bit para query, reply ao query usa o mesmo #
- ❑ **flags**:
  - ❖ query ou reply
  - ❖ recursão desejada
  - ❖ recursão disponível
  - ❖ reply é de servidor "autorizado"

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)	



# Protocolo DNS, mensagens



# Registrando dados no DNS

- ❑ Exemplo: empresa recém criada "Network Utopia"
- ❑ Registrar nome networkutopia.com em um site de **registro** (e.g., Network Solutions, Registro.br)
  - ❖ Necessidade de prover nomes e endereços IP do seu servidor de nomes autorizado (primário e secundário)
  - ❖ Dois registros (RRs) são inseridos no servidor TLD .com:

(networkutopia.com, dns1.networkutopia.com, NS)  
(dns1.networkutopia.com, 212.212.212.1, A)

- ❑ Por no servidor autorizado o RR Tipo A para [www.networkutopia.com](http://www.networkutopia.com) e Tipo MX para networkutopia.com
- ❑ **Como as pessoas obtêm o endereço IP do seu Web site?**

# Module 2: Application layer

- ❑ 2.1 Principles of network applications
  - ❖ app architectures
  - ❖ app requirements
- ❑ 2.2 Web and HTTP
- ❑ 2.4 Electronic Mail
  - ❖ SMTP, POP3, IMAP
- ❑ 2.5 DNS
- ❑ 2.6 P2P file sharing
- ❑ 2.7 Socket programming with TCP
- ❑ 2.8 Socket programming with UDP
- ❑ 2.9 Building a Web server

# P2P file sharing

## Example

- ❑ Alice runs P2P client application on her notebook computer
- ❑ Intermittently connects to Internet; gets new IP address for each connection
- ❑ Asks for "Hey Jude"
- ❑ Application displays other peers that have copy of Hey Jude.
- ❑ Alice chooses one of the peers, Bob.
- ❑ File is copied from Bob's PC to Alice's notebook: HTTP
- ❑ While Alice downloads, other users uploading from Alice.
- ❑ Alice's peer is both a Web client and a transient Web server.

All peers are servers = highly scalable!

# P2P: centralized directory

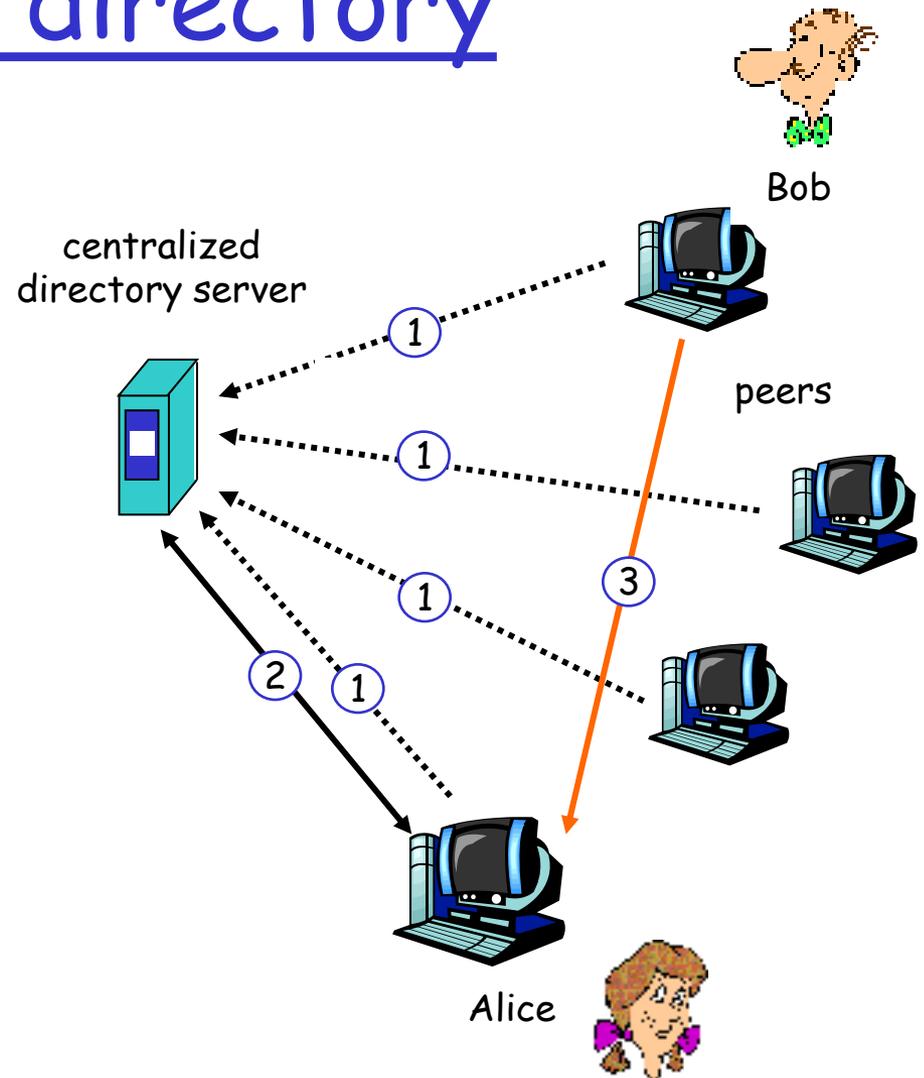
original "Napster" design

1) when peer connects, it informs central server:

- ❖ IP address
- ❖ content

2) Alice queries for "Hey Jude"

3) Alice requests file from Bob



# P2P: problems with centralized directory

- ❑ Single point of failure
- ❑ Performance bottleneck
- ❑ Copyright infringement

file transfer is decentralized, but locating content is highly centralized

# Query flooding: Gnutella

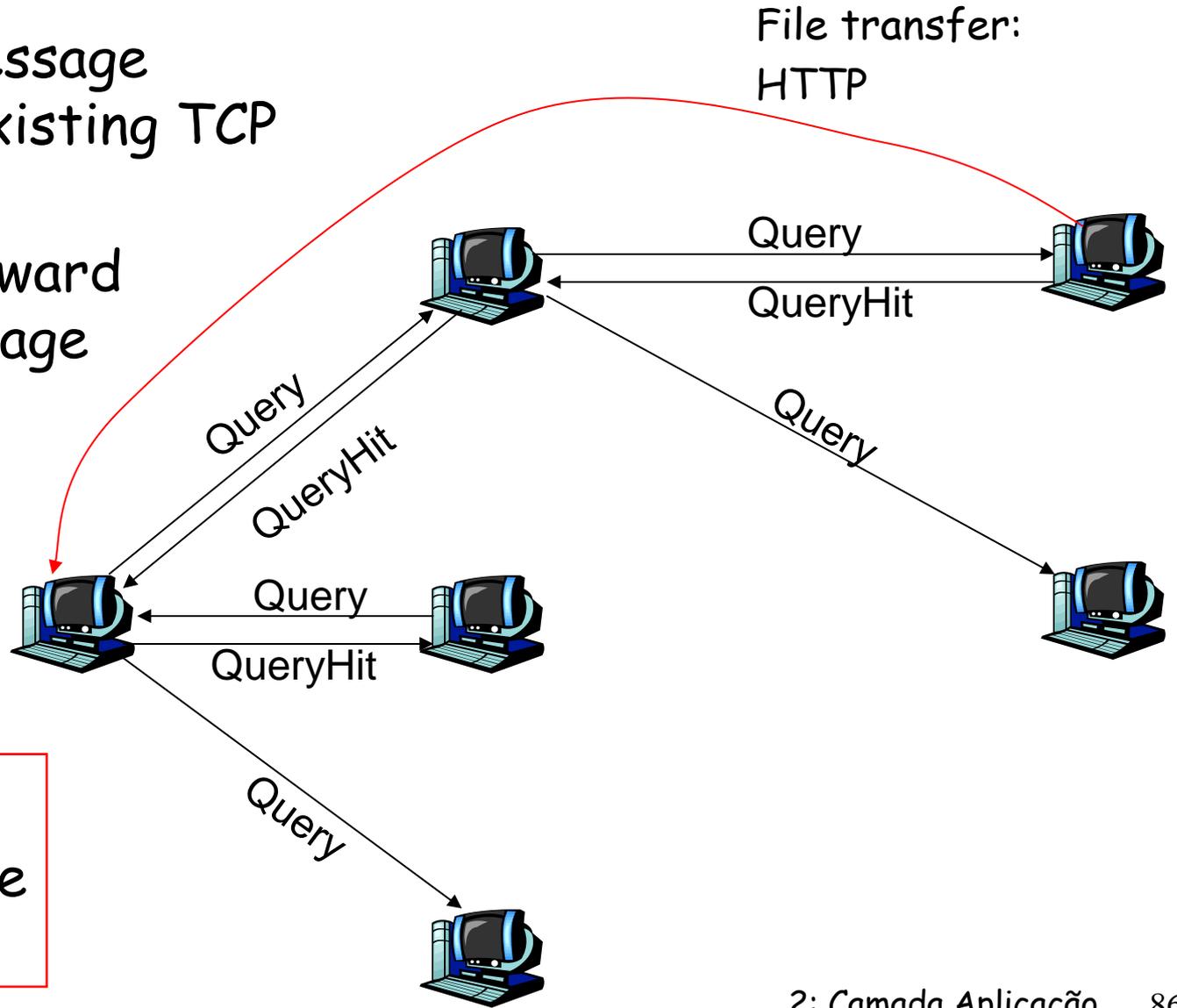
- ❑ fully distributed
  - ❖ no central server
- ❑ public domain protocol
- ❑ many Gnutella clients implementing protocol

## overlay network: graph

- ❑ edge between peer X and Y if there's a TCP connection
- ❑ all active peers and edges is overlay net
- ❑ Edge is not a physical link
- ❑ Given peer will typically be connected with  $< 10$  overlay neighbors

# Gnutella: protocol

- ❑ Query message sent over existing TCP connections
- ❑ peers forward Query message
- ❑ QueryHit sent over reverse path



Scalability:  
limited scope  
flooding

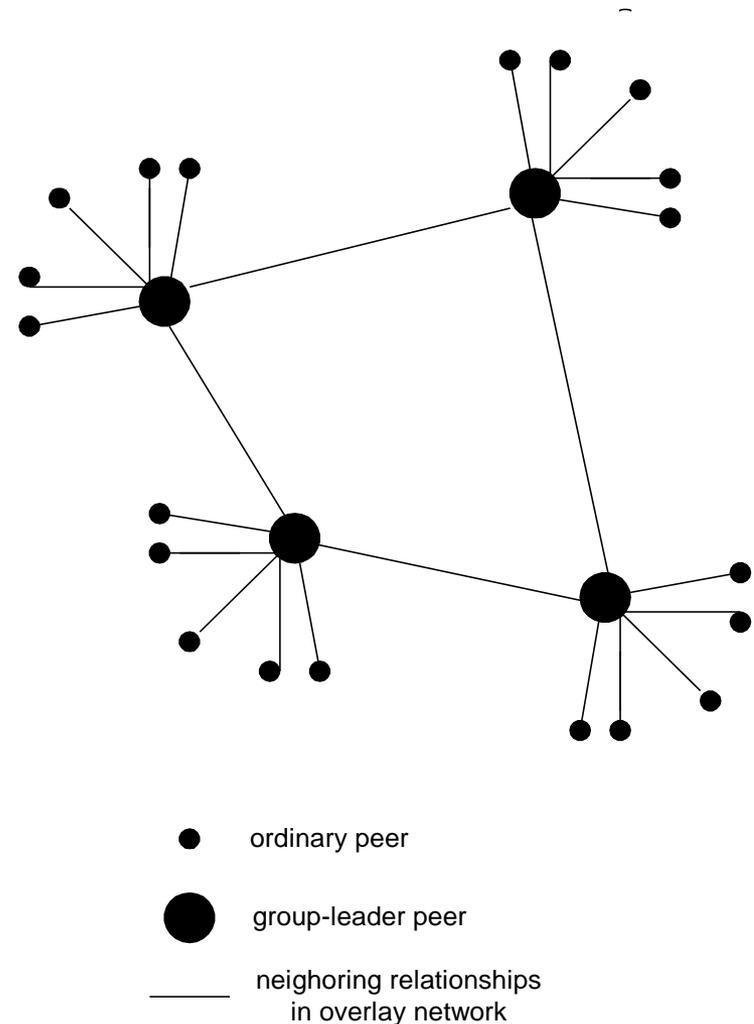
# Gnutella: Peer joining

1. Joining peer X must find some other peer in Gnutella network: use list of candidate peers
2. X sequentially attempts to make TCP with peers on list until connection setup with Y
3. X sends Ping message to Y; Y forwards Ping message.
4. All peers receiving Ping message respond with Pong message
5. X receives many Pong messages. It can then setup additional TCP connections

Peer leaving: see homework problem!

# Exploiting heterogeneity: KaZaA

- Each peer is either a group leader or assigned to a group leader.
  - ❖ TCP connection between peer and its group leader.
  - ❖ TCP connections between some pairs of group leaders.
- Group leader tracks the content in all its children.



# KaZaA: Querying

- ❑ Each file has a hash and a descriptor
- ❑ Client sends keyword query to its group leader
- ❑ Group leader responds with matches:
  - ❖ For each match: metadata, hash, IP address
- ❑ If group leader forwards query to other group leaders, they respond with matches
- ❑ Client then selects files for downloading
  - ❖ HTTP requests using hash as identifier sent to peers holding desired file

# KaZaA tricks

- ❑ Limitations on simultaneous uploads
- ❑ Request queuing
- ❑ Incentive priorities
- ❑ Parallel downloading

For more info:

- ❑ J. Liang, R. Kumar, K. Ross, "Understanding KaZaA,"  
(available via [cis.poly.edu/~ross](http://cis.poly.edu/~ross))

# Module 2: Application layer

- ❑ 2.1 Principles of network applications
- ❑ 2.2 Web and HTTP
- ❑ 2.3 FTP
- ❑ 2.4 Electronic Mail
  - ❖ SMTP, POP3, IMAP
- ❑ 2.5 DNS
- ❑ 2.6 P2P file sharing
- ❑ 2.7 Socket programming with TCP
- ❑ 2.8 Socket programming with UDP
- ❑ 2.9 Building a Web server

# Socket programming

Goal: learn how to build client/server application that communicate using sockets

## Socket API

- ❑ introduced in BSD4.1 UNIX, 1981
- ❑ explicitly created, used, released by apps
- ❑ client/server paradigm
- ❑ two types of transport service via socket API:
  - ❖ unreliable datagram
  - ❖ reliable, byte stream-oriented

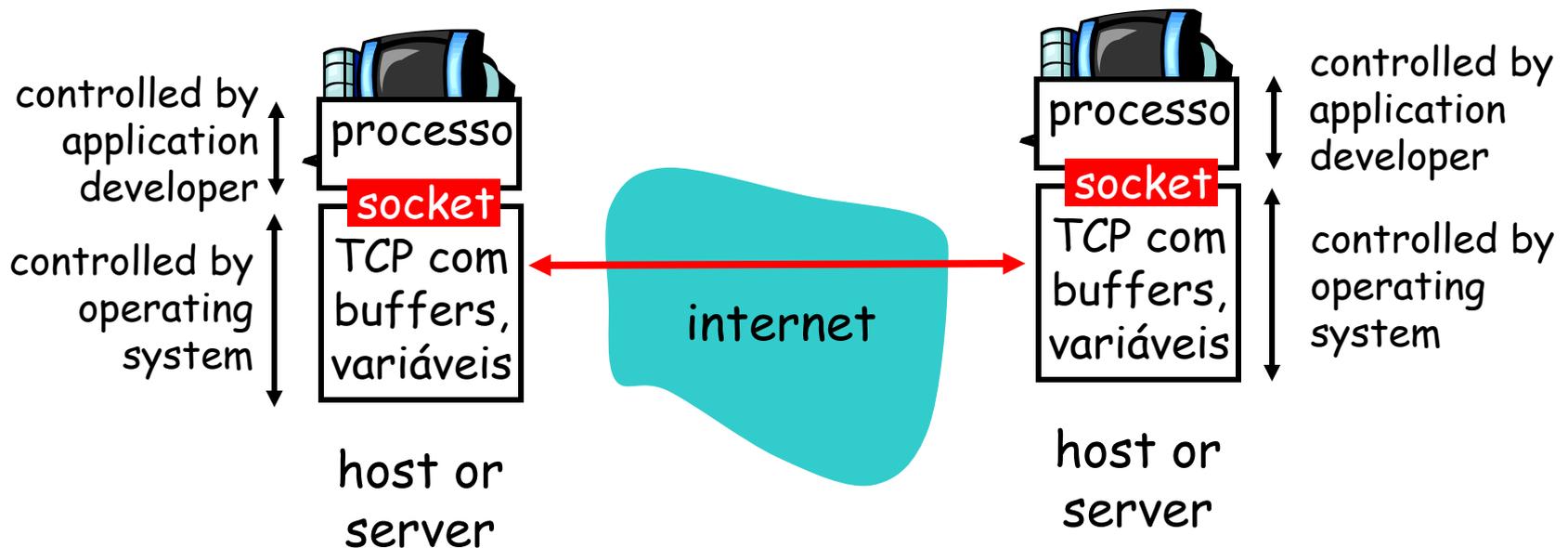
## socket

a *host-local, application-created, OS-controlled* interface (a "door") into which application process can **both send and receive** messages to/from another application process

# Socket-programming using TCP

Socket: a door between application process and end-end-transport protocol (UCP or TCP)

TCP service: reliable transfer of **bytes** from one process to another



# Socket programming *with TCP*

## Client must contact server

- ❑ server process must first be running
- ❑ server must have created socket (door) that welcomes client's contact

## Client contacts server by:

- ❑ creating client-local TCP socket
- ❑ specifying IP address, port number of server process
- ❑ When **client creates socket**: client TCP establishes connection to server TCP

- ❑ When contacted by client, **server TCP creates new socket** for server process to communicate with client
  - ❖ allows server to talk with multiple clients
  - ❖ source port numbers used to distinguish clients (more in Chap 3)

## application viewpoint

*TCP provides reliable, in-order transfer of bytes ("pipe") between client and server*

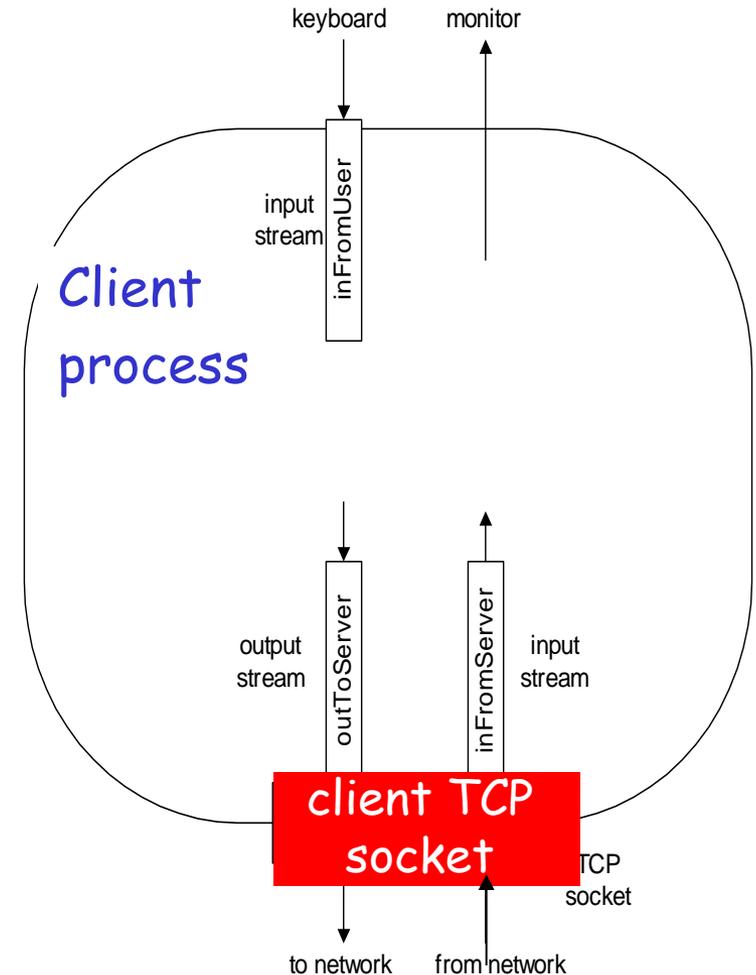
# Stream jargon

- ❑ A **stream** is a sequence of characters that flow into or out of a process.
- ❑ An **input stream** is attached to some input source for the process, e.g., keyboard or socket.
- ❑ An **output stream** is attached to an output source, e.g., monitor or socket.

# Socket programming with TCP

## Example client-server app:

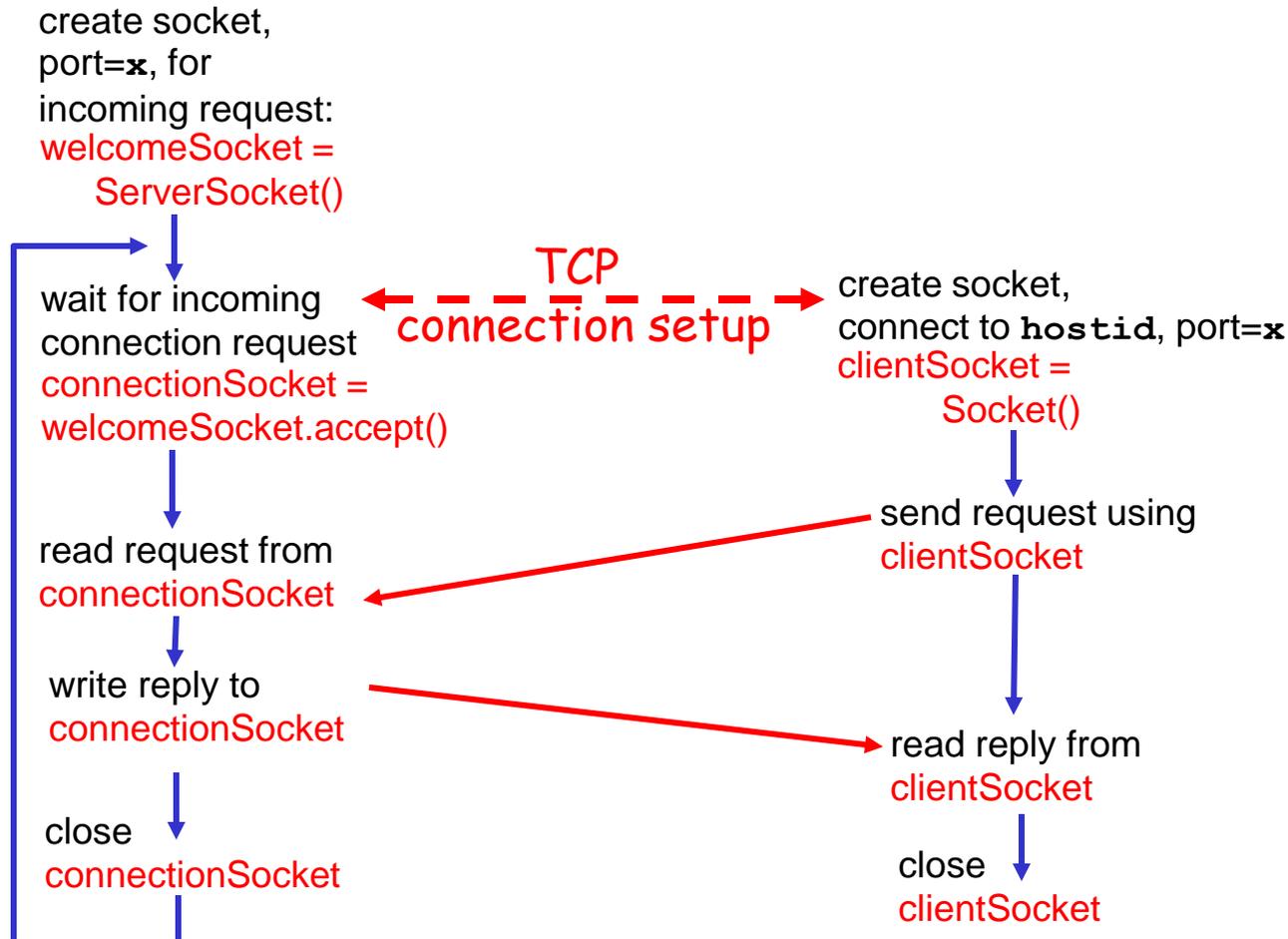
- 1) client reads line from standard input (`inFromUser` stream), sends to server via socket (`outToServer` stream)
- 2) server reads line from socket
- 3) server converts line to uppercase, sends back to client
- 4) client reads, prints modified line from socket (`inFromServer` stream)



# Client/server socket interaction: TCP

Server (running on `hostid`)

Client



# Example: Java client (TCP)

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

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

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

Create  
input stream



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

Create  
client socket,  
connect to server



```
        Socket clientSocket = new Socket("hostname", 6789);
```

Create  
output stream  
attached to socket



```
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
```

# Example: Java client (TCP), cont.

Create  
input stream  
attached to socket

```
BufferedReader inFromServer =  
    new BufferedReader(new  
        InputStreamReader(clientSocket.getInputStream()));  
  
sentence = inFromUser.readLine();  
  
Send line  
to server
```

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

Read line  
from server

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

# Example: Java server (TCP)

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

```
class TCPServer {
```

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

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

Create  
welcoming socket  
at port 6789

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

Wait, on welcoming  
socket for contact  
by client

```
        while(true) {
```

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

Create input  
stream, attached  
to socket

```
            BufferedReader inFromClient =
```

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

# Example: Java server (TCP), cont

Create output stream, attached to socket

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

Read in line from socket

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

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

Write out line to socket

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

```
}  
}  
}
```

End of while loop,  
loop back and wait for  
another client connection

# Module 2: Application layer

- ❑ 2.1 Principles of network applications
- ❑ 2.2 Web and HTTP
- ❑ 2.3 FTP
- ❑ 2.4 Electronic Mail
  - ❖ SMTP, POP3, IMAP
- ❑ 2.5 DNS
- ❑ 2.6 P2P file sharing
- ❑ 2.7 Socket programming with TCP
- ❑ 2.8 Socket programming with UDP
- ❑ 2.9 Building a Web server

# Socket programming *with UDP*

UDP: no "connection" between client and server

- ❑ no handshaking
- ❑ sender explicitly attaches IP address and port of destination to each packet
- ❑ server must extract IP address, port of sender from received packet

UDP: transmitted data may be received out of order, or lost

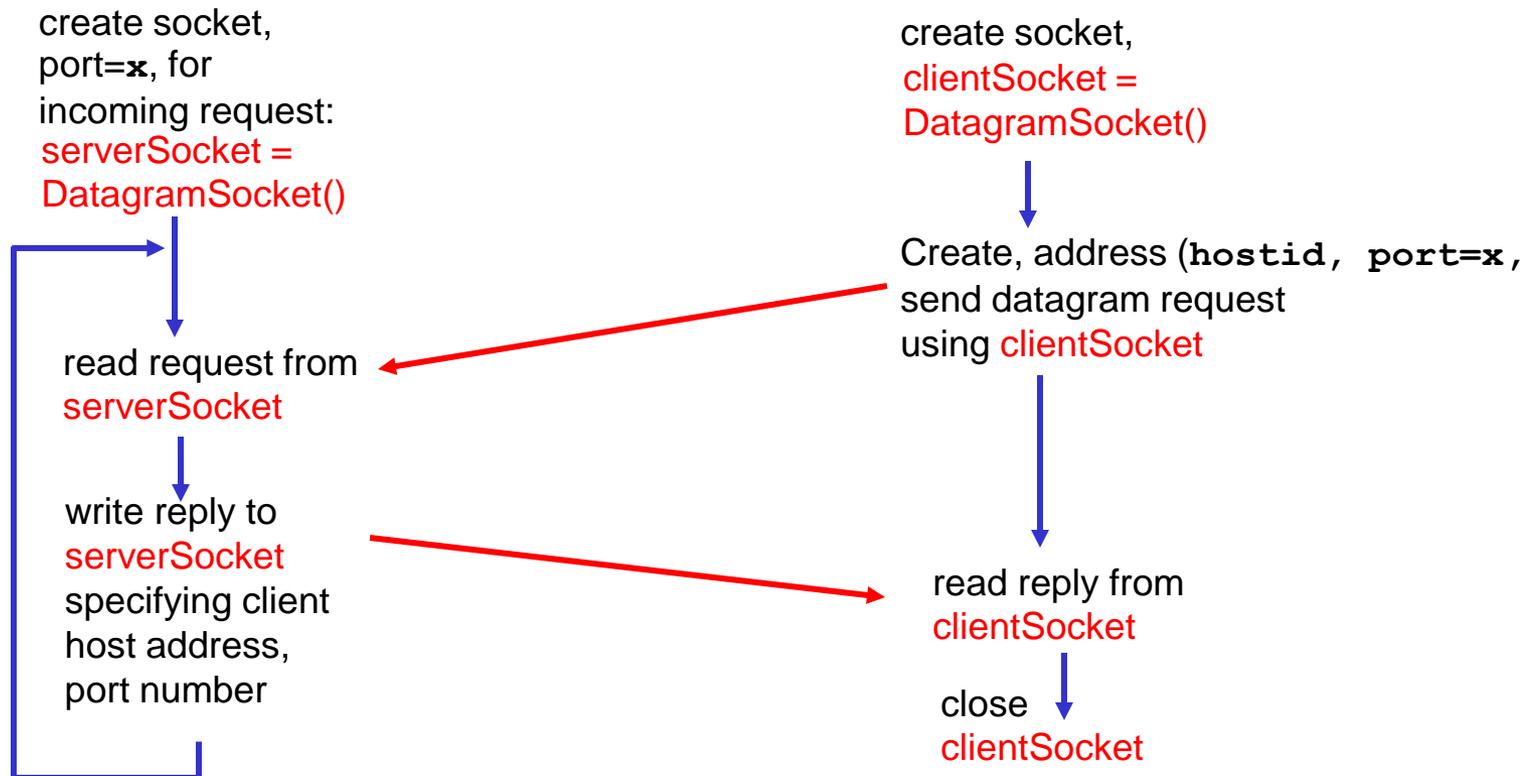
application viewpoint

*UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server*

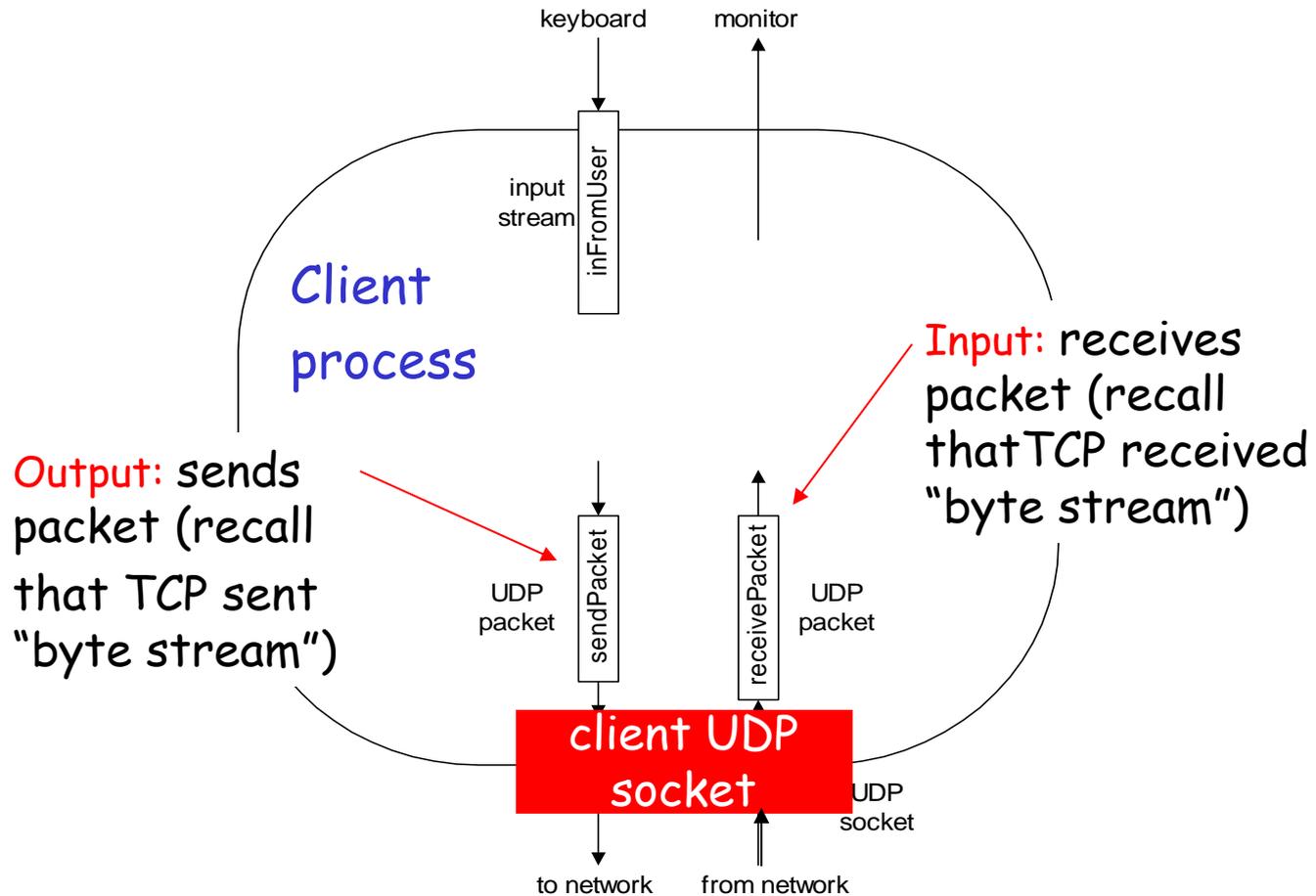
# Client/server socket interaction: UDP

Server (running on `hostid`)

Client



# Example: Java client (UDP)



# Example: Java client (UDP)

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

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

Create  
input stream

```
        BufferedReader inFromUser =
```

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

Create  
client socket

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

Translate  
hostname to IP  
address using DNS

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

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

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

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

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

# Example: Java client (UDP), cont.

```
    Create datagram  
    with data-to-send,  
    length, IP addr, port } DatagramPacket sendPacket =  
                           } new DatagramPacket(sendData, sendData.length, IPAddress, 9876);  
  
    Send datagram  
    to server } clientSocket.send(sendPacket);  
  
              DatagramPacket receivePacket =  
              new DatagramPacket(receiveData, receiveData.length);  
  
    Read datagram  
    from server } clientSocket.receive(receivePacket);  
  
                String modifiedSentence =  
                new String(receivePacket.getData());  
  
                System.out.println("FROM SERVER:" + modifiedSentence);  
                clientSocket.close();  
            }  
        }
```

# Example: Java server (UDP)

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

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

Create  
datagram socket  
at port 9876



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

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

```
while(true)  
{
```

Create space for  
received datagram



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

Receive  
datagram



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

# Example: Java server (UDP), cont

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

Get IP addr  
port #, of  
sender

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

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

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

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

Create datagram  
to send to client

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

Write out  
datagram  
to socket

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

```
}  
}  
}
```

End of while loop,  
loop back and wait for  
another datagram

# Module 2: Application layer

- ❑ 2.1 Principles of network applications
  - ❖ app architectures
  - ❖ app requirements
- ❑ 2.2 Web and HTTP
- ❑ 2.4 Electronic Mail
  - ❖ SMTP, POP3, IMAP
- ❑ 2.5 DNS
- ❑ 2.6 P2P file sharing
- ❑ 2.7 Socket programming with TCP
- ❑ 2.8 Socket programming with UDP
- ❑ 2.9 Building a Web server

# Building a simple Web server

- ❑ handles one HTTP request
  - ❑ accepts the request
  - ❑ parses header
  - ❑ obtains requested file from server's file system
  - ❑ creates HTTP response message:
    - ❖ header lines + file
  - ❑ sends response to client
- ❑ after creating server, you can request file using a browser (e.g., IE explorer)
  - ❑ see text for details

# Module 2: Summary

Our study of network apps now complete!

- Application architectures
  - ❖ client-server
  - ❖ P2P
  - ❖ hybrid
- application service requirements:
  - ❖ reliability, bandwidth, delay
- Internet transport service model
  - ❖ connection-oriented, reliable: TCP
  - ❖ unreliable, datagrams: UDP
- specific protocols:
  - ❖ HTTP
  - ❖ FTP
  - ❖ SMTP, POP, IMAP
  - ❖ DNS
- socket programming

# Module 2: Summary

## Most importantly: learned about protocols

- typical request/reply message exchange:
  - ❖ client requests info or service
  - ❖ server responds with data, status code
- message formats:
  - ❖ headers: fields giving info about data
  - ❖ data: info being communicated
- control vs. data msgs
  - ❖ in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- "complexity at network edge"