INSTITUTO FEDERAL
MINAS GERAIS

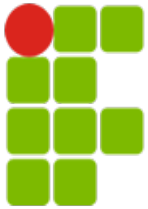
Redes de Computadores

Camada de Aplicação

Prof. Everthon Valadão

Material baseado nos slides de: Dorgival G. (UFMG), Fábio C. (UFG), Kurose & Ross (Pearson)

(última modificação: 26/08/2020)



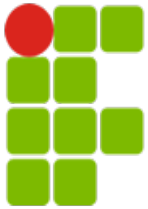
Camada de aplicação

Nosso objetivo:

- conceituar, aspectos de implementação de protocolos de aplicação
  - paradigma cliente-servidor
  - modelos de serviço
- aprender sobre protocolos examinando algumas aplicações populares

Tópicos abordados:

- requisitos de aplicações e serviços demandados
- protocolo HTTP
- protocolo DNS



Camada de aplicação

Nosso objetivo:

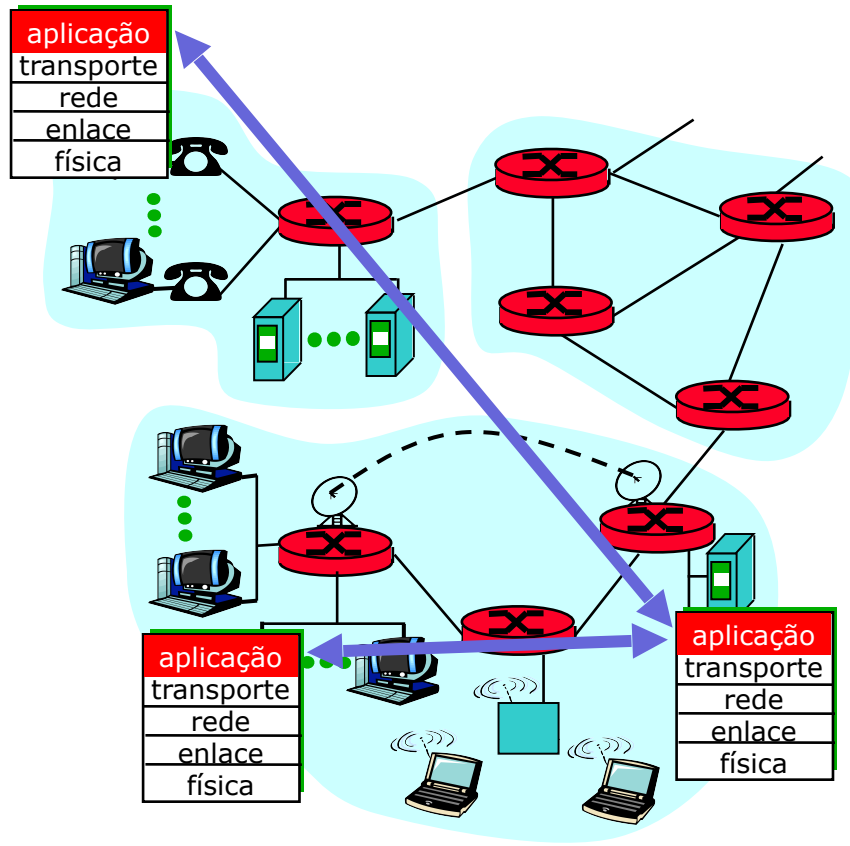
- conceituar, aspectos de implementação de protocolos de aplicação
  - paradigma cliente-servidor
  - modelos de serviço
- aprender sobre protocolos examinando algumas aplicações populares

Tópicos abordados:

- requisitos de aplicações e serviços demandados
- protocolo HTTP
- protocolo DNS

Aplicações e protocolos

- Aplicação: processos distribuídos comunicantes
  - e.x., email, FTP, Web
- Protocolos de aplicação
  - definem mensagens trocadas e as ações tomadas
  - usam serviços das camadas inferiores



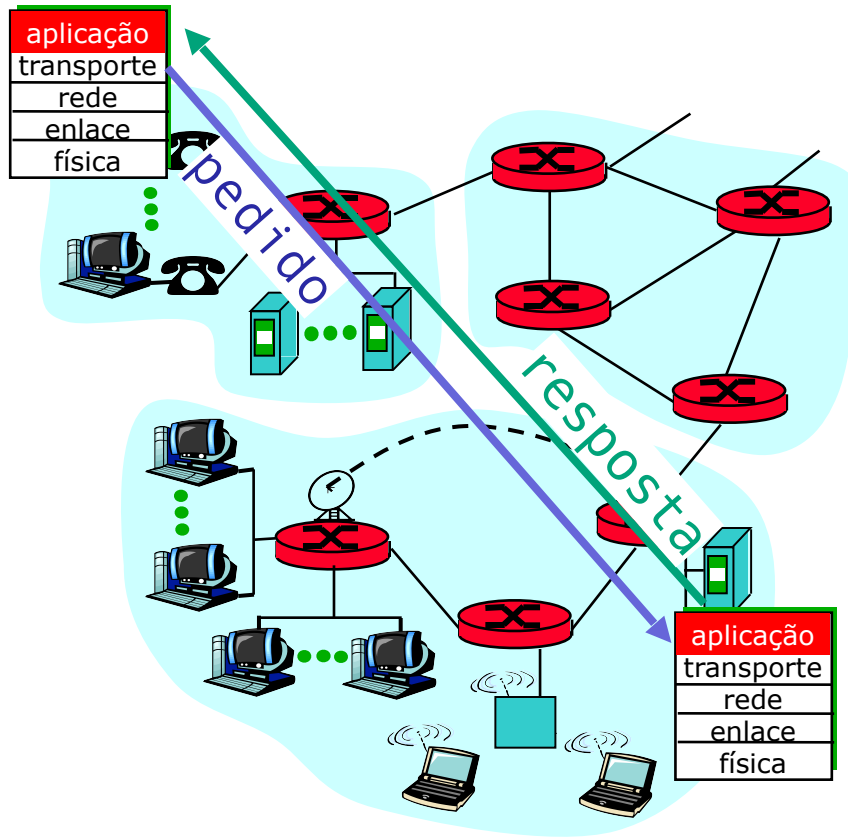
Paradigma cliente-servidor

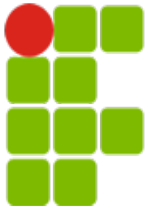
Cliente: inicia comunicação com o servidor (“fala primeiro”)

- solicita serviços do servidor,
- ex.s: cliente web implementado no *browser*; leitor de correio eletrônico (e-mail).

Servidor: fornece os serviços solicitados pelo(s) cliente(s)

- ex.s: *web server* envia a página *web* solicitada; servidor de e-mail envia as mensagens.





Serviços da camada de transporte

Controle de perda de dados

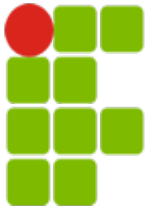
- Certas aplicações podem tolerar alguma perda
  - ex., áudio, vídeo

- Outras aplicações exigem transferência de dados 100% confiável
  - ex., FTP (transferência de arquivos), SSH (conexão remota)

Serviços da camada de transporte

Controle de temporização

- Algumas aplicações (p.ex., telefonia IP, jogos interativos) exigem baixos atrasos para operarem
- Outras exigem atrasos regulares, mesmo que maiores (ex.: serviço de monitoração constante)
- Já outras, ainda, toleram grandes variações nos atrasos (p.ex., Web)

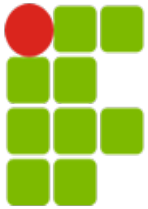


Serviços da camada de transporte

Gerência da banda passante

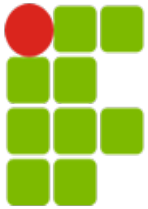
- Algumas aplicações (p.ex., multimídia) exigem uma banda mínima para serem utilizáveis

- Outras aplicações (p.ex., e-mail, web) melhoram quando a banda aumenta, mas funcionam também com menos
  - são conhecidas como “aplicações elásticas”



Requisitos de aplicações usuais

Aplicação	Perdas	Banda	Sensível ao Atraso
transferência de arquivos	sem perdas	elástica	não
e-mail	sem perdas	elástica	não
documentos Web	sem perdas	elástica	não
áudio/vídeo em tempo real	tolerante	aúdio: **5Kbps**-1Mbps	sim, 40 ms
		vídeo: 10Kbps-**5Mbps**	sim, segundos
áudio/vídeo armazenado	tolerante	igual à anterior	sim, 40 ms
jogos interativos	tolerante	alguns Kbps	sim, 20 ms
comércio eletrônico	sem perdas	elástica	sim, segundos



Camada de aplicação

Nosso objetivo:

- conceituar, aspectos de implementação de protocolos de aplicação
  - paradigma cliente-servidor
  - modelos de serviço
- aprender sobre protocolos examinando algumas aplicações populares

Tópicos abordados:

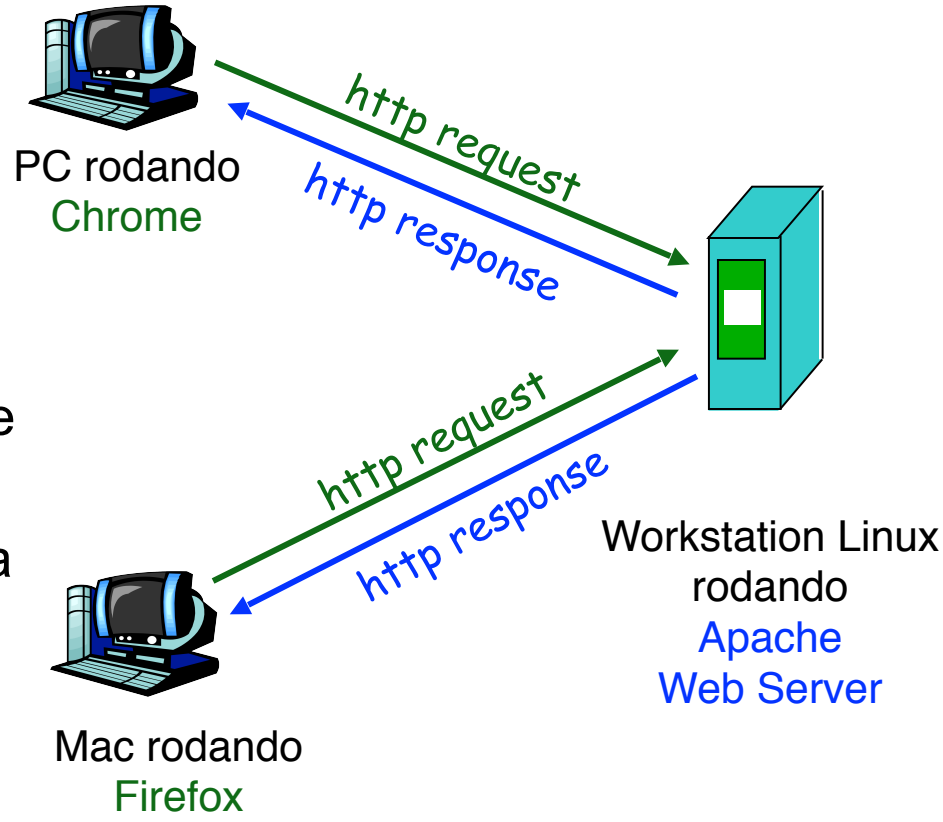
- requisitos de aplicações e serviços demandados
- protocolo HTTP
- protocolo DNS

Protocolo de Aplicação: HTTP

HTTP: *HyperText Transfer Protocol*

- protocolo da camada de aplicação da Web

- modelo cliente/servidor
  - *cliente:* browser que solicita, recebe e apresenta objetos
  - *server:* envia objetos em resposta a pedidos



Protocolo de Aplicação: HTTP

Protocolo de transporte utilizado: TCP

- cliente inicia conexão TCP (cria socket) para o servidor no porto 80
- servidor aceita uma conexão TCP do cliente
- mensagens HTTP (protocolo de camada de aplicação) são trocadas entre o navegador (cliente HTTP) e o servidor Web (servidor HTTP)
- ao final, a conexão TCP é fechada

Exemplo de operação

Usuário entra com a URL: www.formiga.ifmg.edu.br/index.php

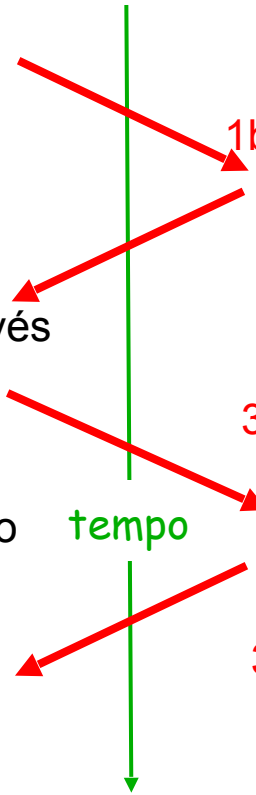
(contém referências p/ mais de 20 imagens)

1a. cliente inicia conexão TCP c/ servidor em www.formiga.ifmg.edu.br no porto 80 (a usual para um servidor http).

1b. servidor no host www.formiga.ifmg.edu.br espera pela conexão no porto 80 e "aceita" conexão, notificando cliente

2. cliente envia pedido com a URL através da conexão TCP

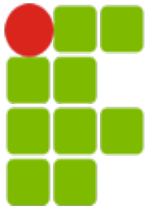
3. servidor recebe pedido, forma *resposta* com o objeto solicitado (index.php) e envia mensagem através do soquete



4. cliente recebe resposta com o arquivo HTML, exibindo o texto dele. Analisando o arquivo HTML, encontra referências para 23 imagens JPEG e PNG

3b. servidor fecha conexão TCP.

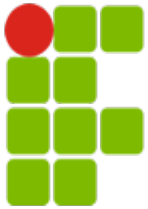
5. Após, os passos 1-4 são repetidos para cada um dos objetos (ex.: imagens)!



Protocolo de Aplicação: HTTP

- Protocolo sem informações de estado (“stateless”)
  - servidor não mantém informação sobre os pedidos passados
  - imagine se o servidor Web tivesse que armazenar informações dos milhões de clientes que ele atende...

- OBS.: protocolos que mantêm estado são complexos!
  - necessidade de organizar informações passadas
  - se ocorrer uma falha (*crash*) as informações podem ser perdidas ou gerar *inconsistências* entre o cliente e o servidor
  - PORÉM, isso tem um impacto no desenvolvimento de aplicações interativas
    - p.ex.: como manter o carrinho de compras do usuário?



Conexões persistentes e não-persistentes

Não-persistente

- HTTP/1.0: servidor analisa pedido, envia resposta e fecha a conexão TCP
- mais trabalho para cada objeto por conta da conexão
- navegadores abriam várias conexões paralelas para contornar isso

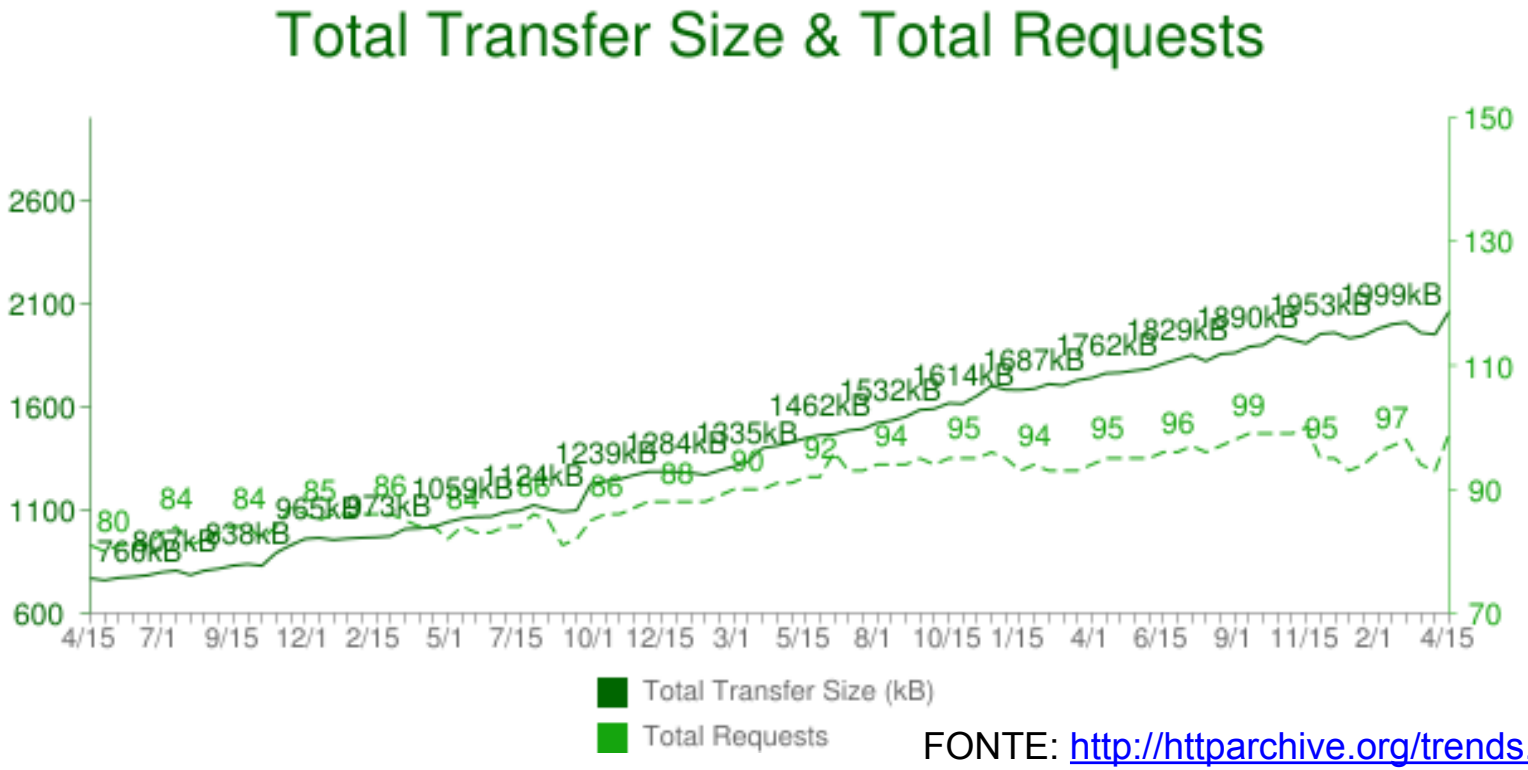
Persistente

- modo padrão para HTTP/1.1
- na mesma conexão TCP são transferidos vários objetos
- o cliente envia *pedido para todos os objetos referenciados* tão logo ele recebe a página HTML básica
- evita o trabalho e tempo de estabelecer cada conexão individualmente

Conexões persistentes e não-persistentes

- O protocolo HTTP/1.1 serviu a Web por mais de 15 anos, mas está já começando a envelhecer.
- Carregar uma página Web é cada dia mais intensivo no uso de recursos!

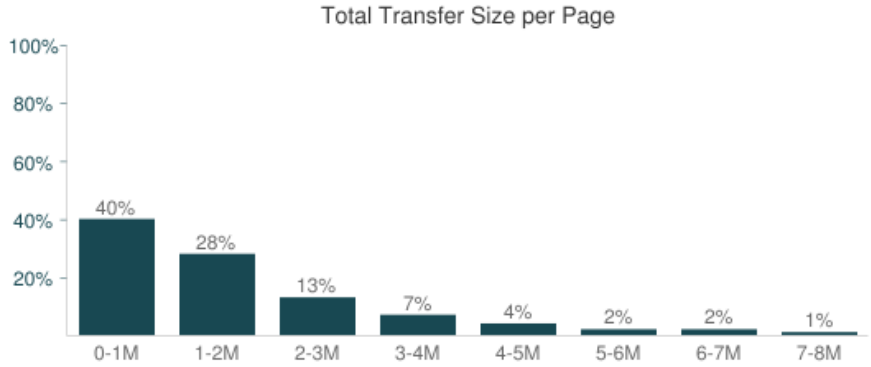
(vide abaixo a comparação entre 2011-2015)

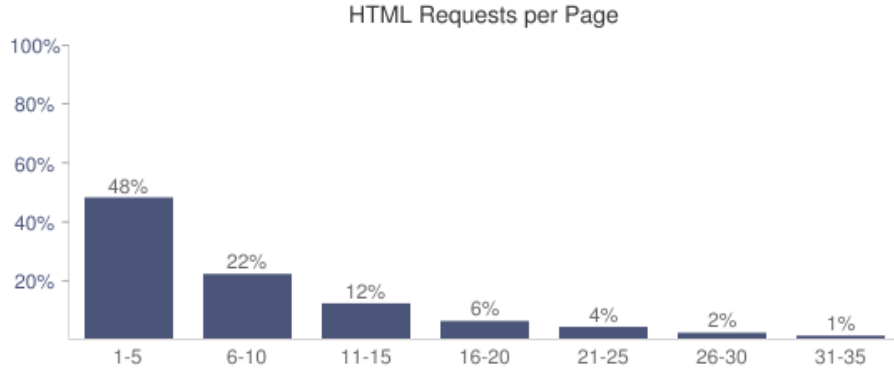


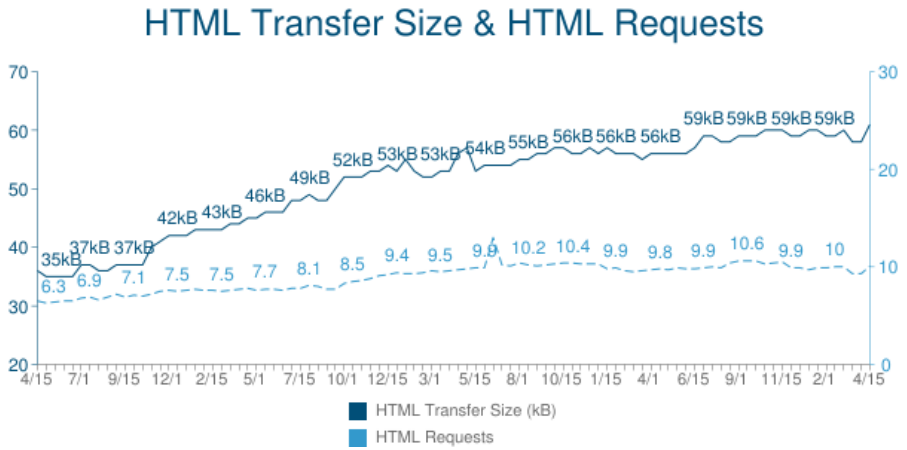
FONTE: http://httparchive.org/trends.php

Conexões persistentes e não-persistentes

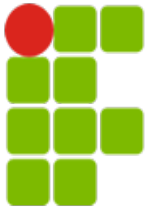
- 60% das páginas Web tem um tamanho total maior que 1 MB
- 52% das páginas Web demandam mais que 5 solicitações HTTP por página







FONTE: http://httparchive.org/trends.php



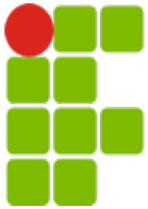
Conexões persistentes e não-persistentes

- Carregar todos estes objetos Web de maneira eficiente é difícil, pois o HTTP/1.1 praticamente permite apenas 01 requisição pendente por conexão TCP

- Para contornar esta limitação, os navegadores Web (browsers) utilizavam várias conexões TCP para solicitar os objetos em paralelo

- Porém, se múltiplas conexões TCP são utilizadas
  - é contraproducente pois o controle de congestionamento do TCP é negado, prejudicando o desempenho da rede
  - é fundamentalmente injusto pois os navegadores estão tentando obter mais do que sua "fatia" dos recursos da rede

- Assim, requisições HTTP/1.1 tem muita sobrecarga associada a elas e se muitas requisições são feitas, isso fere o desempenho

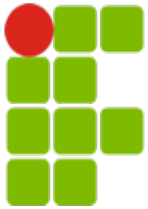


HTTP/2.0

- Padrão desenvolvido pela IETF, com contribuições de grandes nomes como Sir Tim Berners-Lee, Firefox, Chrome, Twitter, Microsoft, Akamai

- As diferenças principais em relação ao HTTP/1.x são
  - é **binário** ao invés de textual
  - é totalmente **multiplexado**, ao invés de ordenado e “bloqueante"
  - assim, pode utilizar uma conexão para **paralelismo**
  - utiliza **compressão do cabeçalho** para reduzir a sobrecarga
  - permite os servidores enviar respostas proativamente (**push**) nas caches dos clientes

FONTE: https://http2.github.io/faq/



Porque o HTTP/2.0 é **binário**

- **Protocolos binários** são mais eficientes de se analisar, mais compactos no enlace e menos propensos a erros

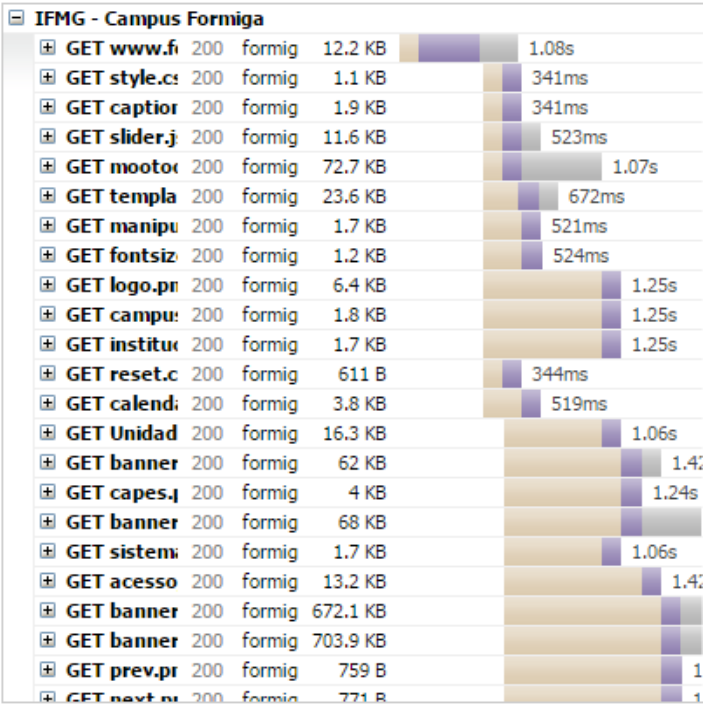
- Isso se comparado a protocolos textuais como o HTTP/1.x, que costumam ter uma série de detalhismos
  - ex.: como lidar com espaços em branco, capitalização, final de linha, etc

- Por exemplo, o HTTP/1.1 define quatro maneiras diferentes de se analisar uma mensagem, já no HTTP/2.0 há somente uma forma

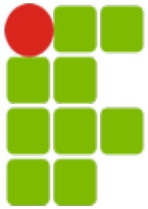
FONTE: https://http2.github.io/faq/

Porque o HTTP/2.0 é **multiplexado**

- O HTTP/1.x tem um problema chamado "*head-of-line blocking*" em que apenas uma requisição pode estar pendente em uma conexão de cada vez
  - o HTTP/1.1 tentou resolver isso com *pipelining*, mas não completamente (uma resposta grande ou demorada ainda bloca aquelas anteriores a ela)

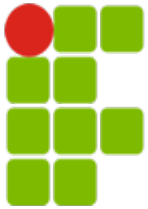


  - Isso força os clientes a usar algum artifício para determinar que requisições colocar em qual conexão para o servidor, resultando em um efeito de cascata
    - ex.: https://gtmetrix.com/reports/www.formiga.ifmg.edu.br/jt3BdBqG
  - A multiplexação ataca estes problemas permitindo múltiplas requisições e múltiplas respostas serem transmitidas ao mesmo tempo
    - isso permite ao cliente utilizar apenas uma conexão por origem para carregar a página Web



Porque apenas uma conexão TCP

- Com o HTTP/1, navegadores abriam entre 4 e 8 conexões por origem
  - como muitos sites tem múltiplas origens, isso poderia significar que uma única página abriria mais de 30 conexões!
- Uma aplicação abrindo tantas conexões simultâneas quebra uma série de considerações sobre as quais o TCP foi construído
  - como cada conexão receberá um fluxo de dados na resposta, há o risco real de que os buffers da rede subjacente sobrecarreguem, causando congestionamentos e consequentes retransmissões
- Utilizar tantas conexões é injusto pois monopoliza recursos da rede, "roubando-os" de outras aplicações mais "bem comportadas" (ex.: VoIP)

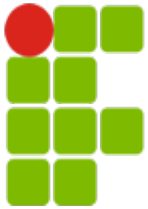


O benefício do *Server Push*

- Quando um navegador requisita uma página, o servidor envia o HTML em resposta, daí espera o navegador analisar o HTML

- Só depois de analisar o HTML é que o navegador (cliente) solicita ao servidor todos demais os objetos referenciados pela página Web (ex.: JavaScript, CSS, imagens)

- *Server Push* permite ao servidor evitar este atraso na espera ao enviar proativamente as respostas que julgar que o cliente irá precisar em sua cache



Porque precisamos de compressão do cabeçalho?

- Visualize:
  - se considerarmos que uma página Web tenha em torno de 80 objetos (estimativa conservativa)
  - e que cada requisição tenha 1400 bytes de cabeçalho (considerando *cookies*, *referer*, etc)
  - isso levaria ao menos 7-8 viagens de ida-e-volta (RTT) para obter os cabeçalhos pelo enlace
  - isso sem contar o tempo de resposta, só pra enviá-las do cliente

- Assim, até mesmo uma mínima compressão do cabeçalho permite que essas requisições viagem pelo enlace em 1 RTT, talvez até mesmo em 1 só pacote

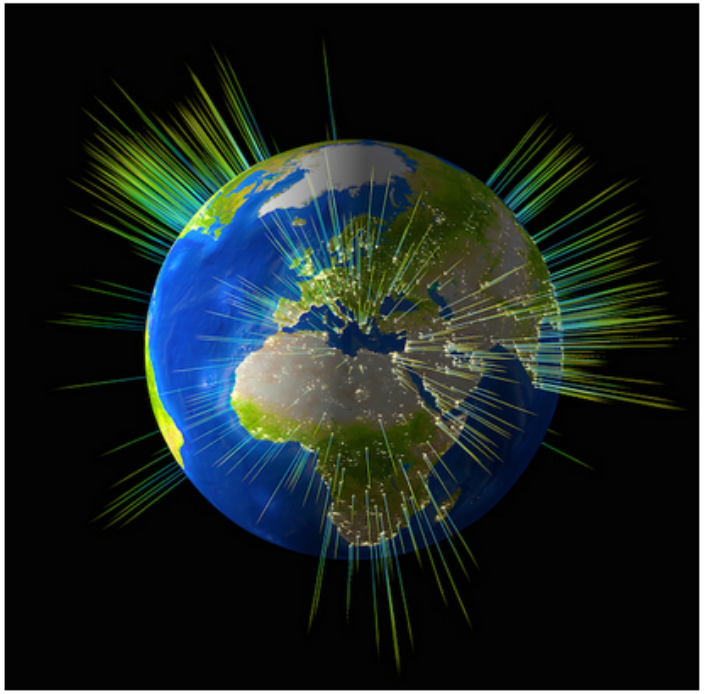
- Esta sobrecarga é considerável, especialmente se considerarmos o impacto em clientes móveis, com RTT de centenas de milisegs.

Demonstração do desempenho do HTTP/2.0

- https://http2.akamai.com/demo (utilize aba anônima no **Wi-Fi** ou rede **3G**)

HTTP/1.1

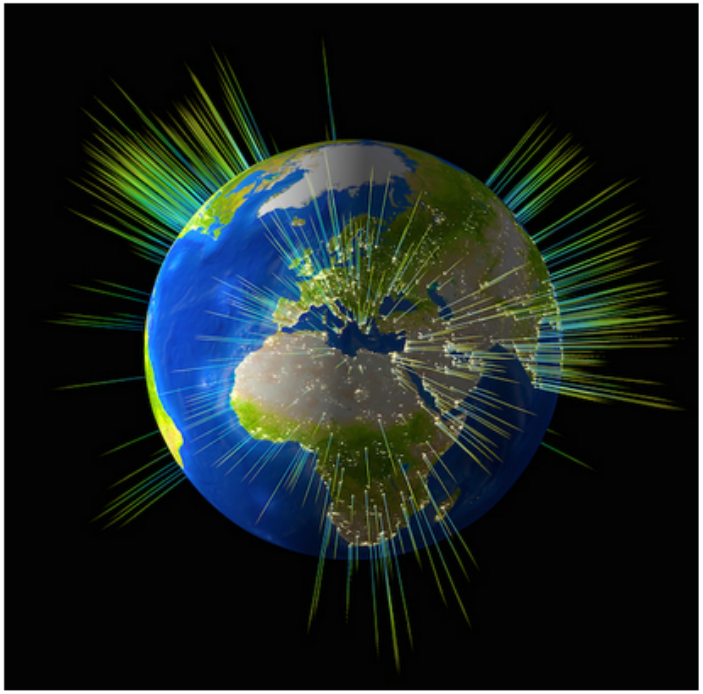
Latency: 70ms Image load time: 3.63s



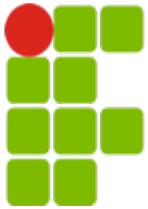
Demo concept inspired by Golang's Gophertiles

HTTP/2

Latency: 71ms Image load time: 0.80s



Return to Akamai's HTTP/2 page



Cookies

- Forma de se manter estado entre pedidos HTTP

- Não fazem parte do protocolo em si, são extensões "entendidas" pelo cliente e pelo servidor:

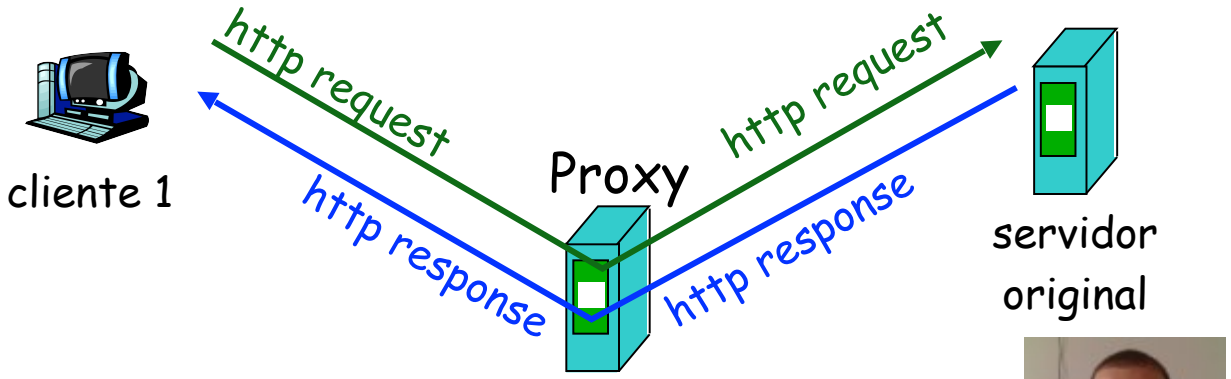
  - Servidor **entrega** um *cookie* (pequeno arquivo de texto) ao cliente

  - Cliente **devolve** esse *cookie* em cada requisição HTTP posterior (ao mesmo servidor)
    - ex.: *cookie* poderia conter identificador do cliente (nome do usuário, código da conta, etc)

Web caches (servidor *proxy*)

Objetivo: atender o cliente sem envolver o servidor original

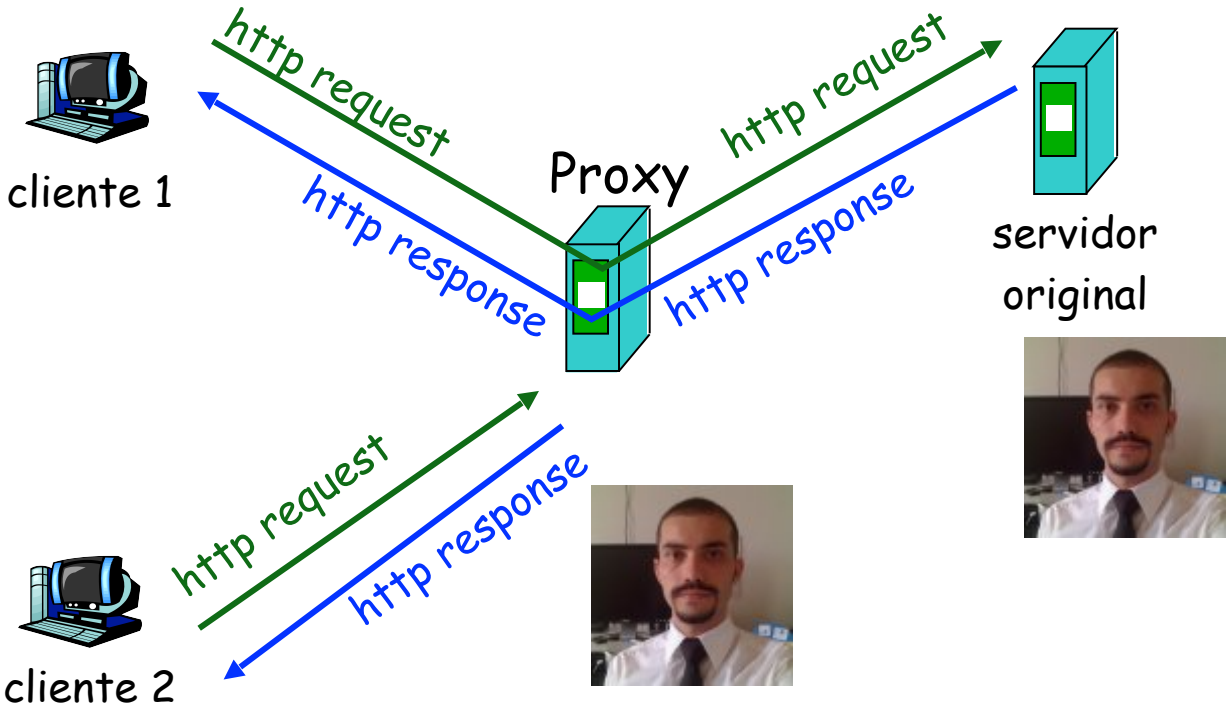
- o 1º cliente solicita um objeto, sendo obtido no servidor original;
- após, caso o proxy já possua o objeto, encaminha-o aos próximos clientes (caso contrário, solicitaria o novo objeto ao servidor original);



Web caches (servidor *proxy*)

Objetivo: atender o cliente sem envolver o servidor original

- o 1º cliente solicita um objeto, sendo obtido no servidor original;
- após, caso o proxy já possua o objeto, encaminha-o aos próximos clientes (caso contrário, solicitaria o novo objeto ao servidor original);



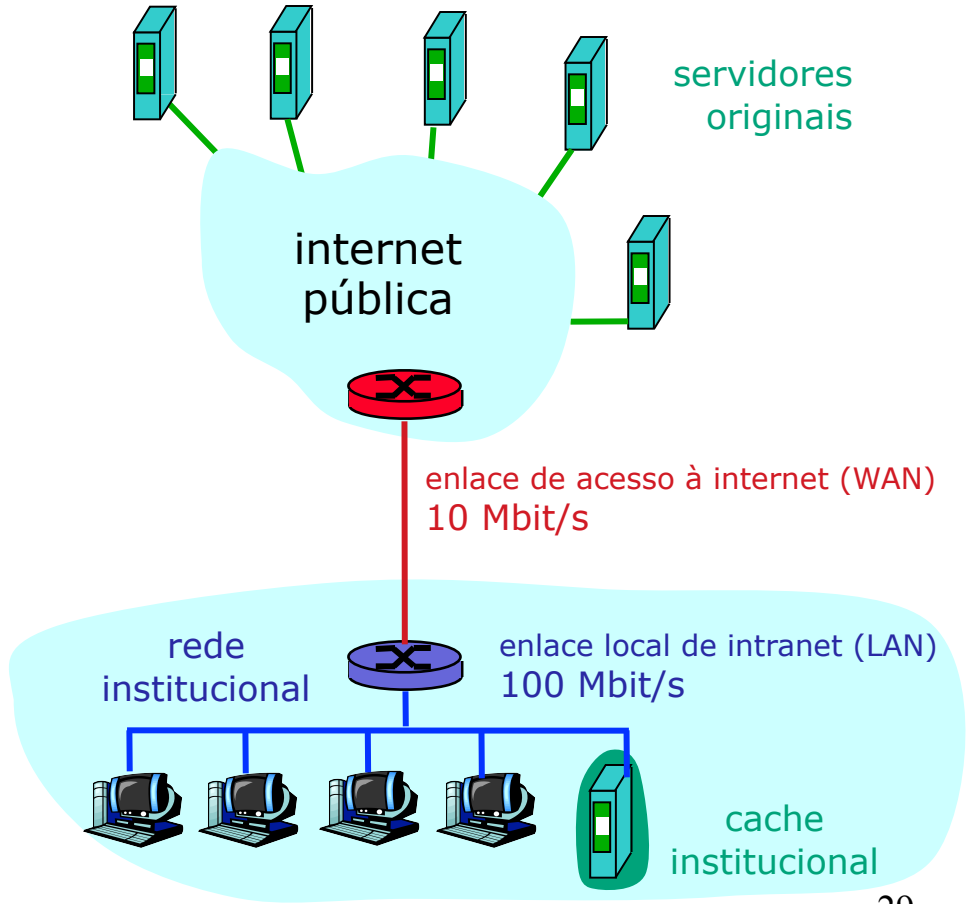
Por que usar caches?

- Armazenamento está “perto” do cliente:

→ menor tempo de resposta

(µs na LAN X ms na WAN)

→ reduz o tráfego na Internet



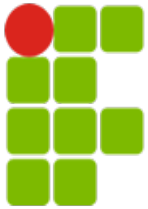
Camada de aplicação

Nosso objetivo:

- conceituar, aspectos de implementação de protocolos de aplicação
  - paradigma cliente-servidor
  - modelos de serviço
- aprender sobre protocolos examinando algumas aplicações populares

Tópicos abordados:

- requisitos de aplicações e serviços demandados
- protocolo HTTP
- protocolo DNS



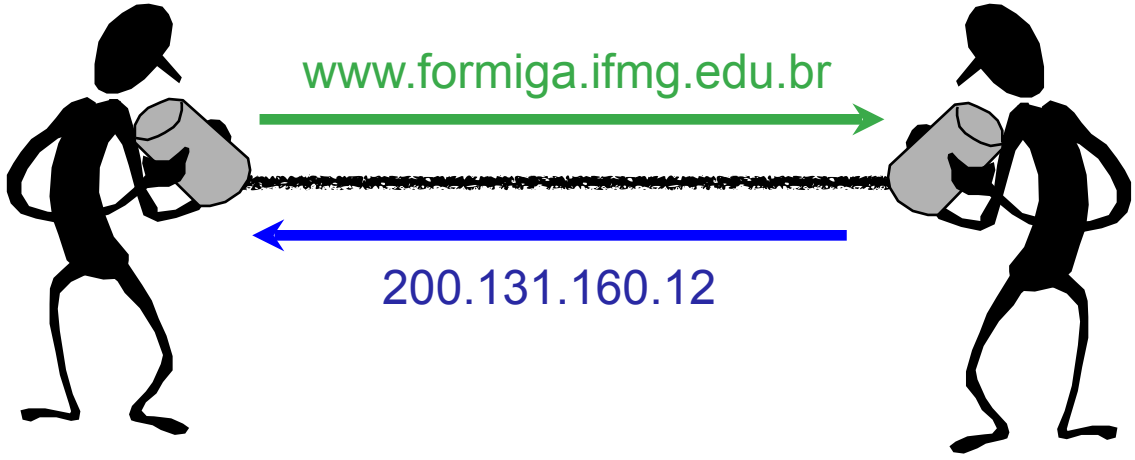
Aplicação: nomes e endereços

- Normalmente, aplicações e usuários fazem referência a um computador através de um nome
  … mas na Internet, servidores possuem endereços numéricos (IPs)!
- É necessário um mecanismo de mapeamento de nome para endereço e vice-versa
- Quando havia a Arpanet, um arquivo **`hosts.txt`** tinha todos os computadores e endereços IP da rede
  - Inviável com o crescimento da Internet, hoje tem fins restritos
- Nova solução:
  - DNS, especificado nas RFCs 1034 e 1035

Protocolo DNS

Domain Name System

- Sistema hierárquico distribuído para tradução de nomes de hosts para seus respectivos endereços IP



Consulta DNS via comando dig

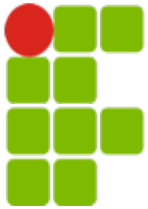
```
-bash
[~] $ dig www.formiga.ifmg.edu.br

; <<>> DiG 9.10.6 <<>> www.formiga.ifmg.edu.br
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 384
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.formiga.ifmg.edu.br.       IN      A

;; ANSWER SECTION:
www.formiga.ifmg.edu.br. 3587   IN      A       200.131.160.12

;; Query time: 80 msec
;; SERVER: 192.168.2.64#53(192.168.2.64)
;; WHEN: Thu Aug 27 05:41:36 -03 2020
;; MSG SIZE  rcvd: 68
```

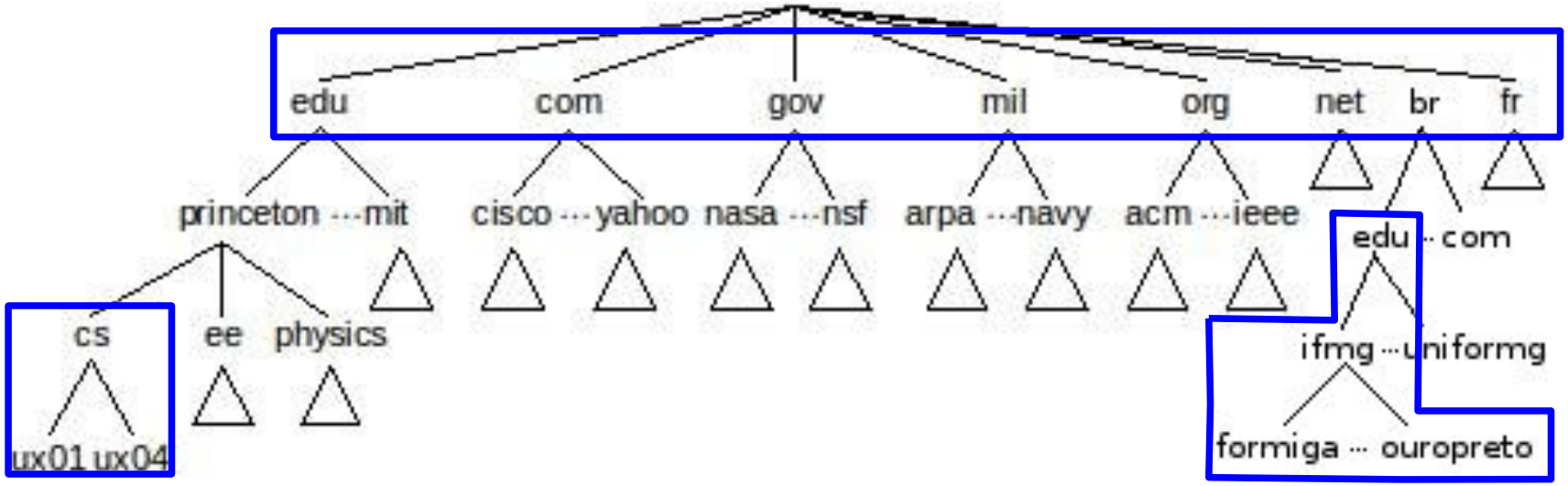


DNS: Domain Name System

Domain Name System:

- *base de dados distribuída* implementada numa hierarquia de muitos *servidores de nomes*
- hosts e roteadores se comunicam com servidores de nomes para *resolver* nomes (tradução nome/endereço)
- nenhum servidor tem todos os mapeamentos
  - servidores de nomes locais: cada ISP/organização tem uma base com seus dados locais (autoridade para aqueles dados)
  - servidores raiz: mantêm um ponto de origem para a hiearquia
  - cada servidor conhece a raiz e os servidores dos seus sub-domínios

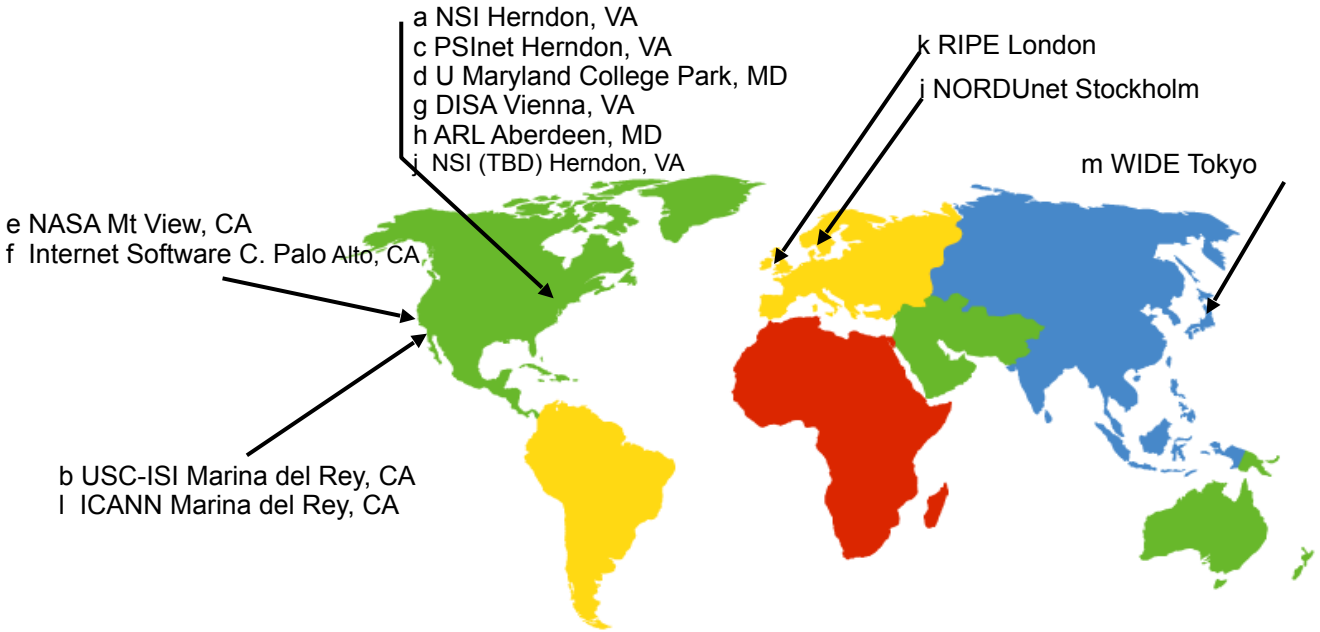
DNS



- Cada nível define um domínio independente e autônomo
- Cada domínio (ou região contígua) implementa um servidor
- Informação é distribuída pelos vários servidores da rede

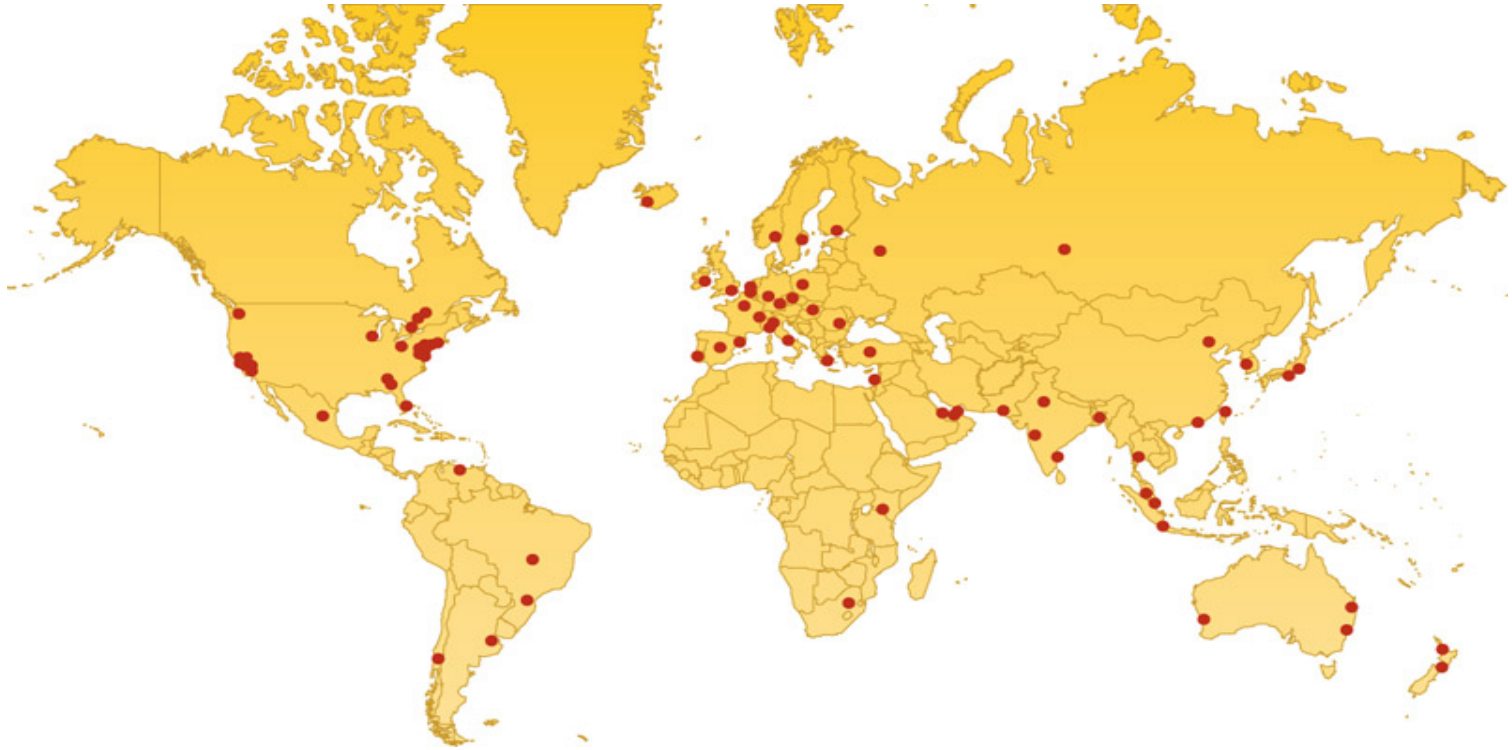
DNS: servidores de nomes raiz
(até 2006)

- buscam servidor(es) de nomes autoritativos p/ pelo menos parte do nome
- retornam o mapeamento para o servidor de nomes local
- existem 13 servidores de nomes raiz no mundo



DNS: servidores de nomes raiz
(após 2006)

- Alguns servidores replicados pela rede afora
  - Mas continuam apenas 13 endereços diferentes (*anycast*)



DNS: exemplo simples

O host **meupc.provedor.com.br** quer o endereço IP de **formiga.ifmg.edu.br**

1. contata seu servidor DNS local, **dns.provedor.com.br**

2. **dns.provedor.com.br** contata o servidor de nomes raiz, se necessário

3. o servidor de nomes raiz contata o servidor de nomes autoritativo **dns.edu.br**