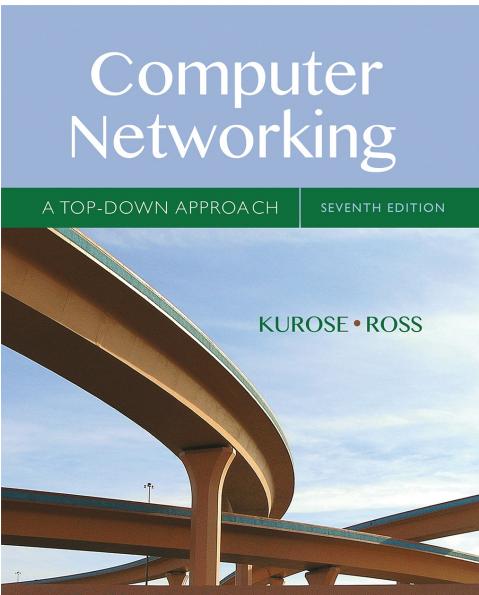


HTTP

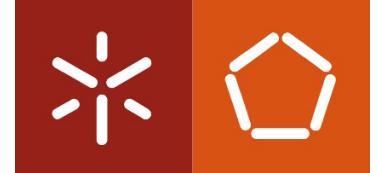


Comunicações por Computador
Licenciatura em Engenharia Informática

3º ano/1º semestre
2021/2022



***Computer Networking: A Top Down Approach,
Capítulo 2***
Jim Kurose, Keith Ross, Addison-Wesley ©2016 .



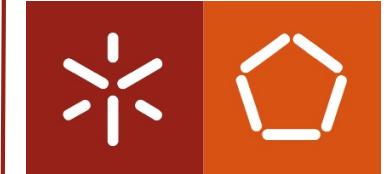
Conceitos básicos, bem conhecidos...

- **Uma página Web consiste numa coleção de objetos**
- **Um objeto pode ser um ficheiro HTML, uma imagem JPEG image, um applet Java, um ficheiro audio...**
- **Uma página Web consiste num ficheiro de base HTML que inclui várias referências a outros objetos**
- **Cada objeto é endereçado por uma URL (Uniform Resource Locator)**

URL exemplo:

http://www.di.uminho.pt/cursos/miei.html

The URL 'http://www.di.uminho.pt/cursos/miei.html' is shown. The 'http://www.di.uminho.pt' part is highlighted with a light orange oval and labeled 'host name'. The '/cursos/miei.html' part is labeled 'path name'.



HTTP

Como funciona?

HTTP: hypertext transfer protocol

- **Protocolo do nível da aplicação**
- **Modelo cliente/servidor**
 - *cliente*: browser pede, recebe e mostra objetos Web
 - *servidor*: servidor envia objetos como resposta a pedidos

HTTP 0.9: versão inicial (não oficial)

HTTP 1.0: RFC 1945 (maio 1996)

HTTP 1.1: RFC 2068 (janeiro 1997)

HTTP 2: RFC 7540 (maio 2015)

HTTP 3: Draft de 23 março 2021

PC a executar o Firefox

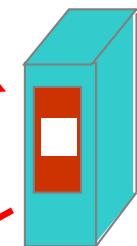


Pedido HTTP
Resposta HTTP



Mac a executar o Safari

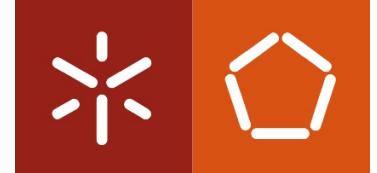
Pedido HTTP
Resposta HTTP



Servidor a executar o servidor WEB Apache

HTTP

Como funciona?



Utiliza o TCP:

- O cliente inicia uma conexão TCP (cria um *socket*) com um servidor HTTP (porta 80).
- O servidor TCP aceita o pedido de conexão do cliente
- São trocadas mensagens HTTP (mensagens de protocolo de nível de aplicação) entre o browser (cliente HTTP) e o servidor Web (servidor HTTP)
- A ligação TCP é terminada

O HTTP não tem estado

- O servidor não mantém estado acerca dos pedidos anteriores dos clientes

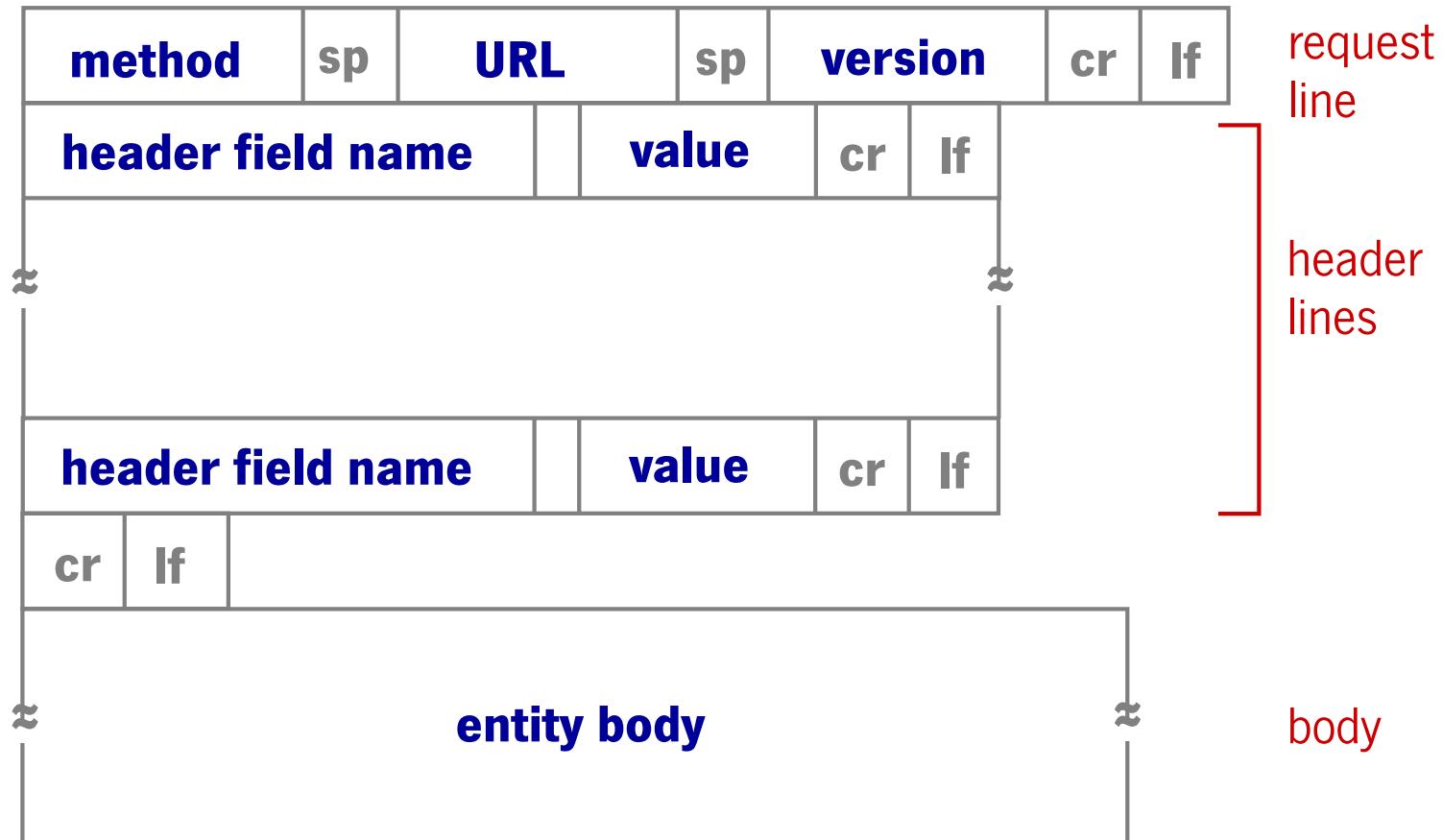
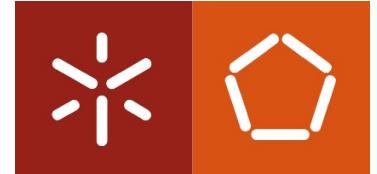
Os protocolos orientados ao estado são mais complexos!

- O passado tem que ser armazenado
- Se o servidor/cliente falham a sua visão do estado pode ficar inconsistente e terá que ser sincronizada

Aplicações de rede

Exemplo: HTTP

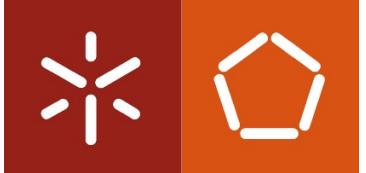
Formato dos PDU



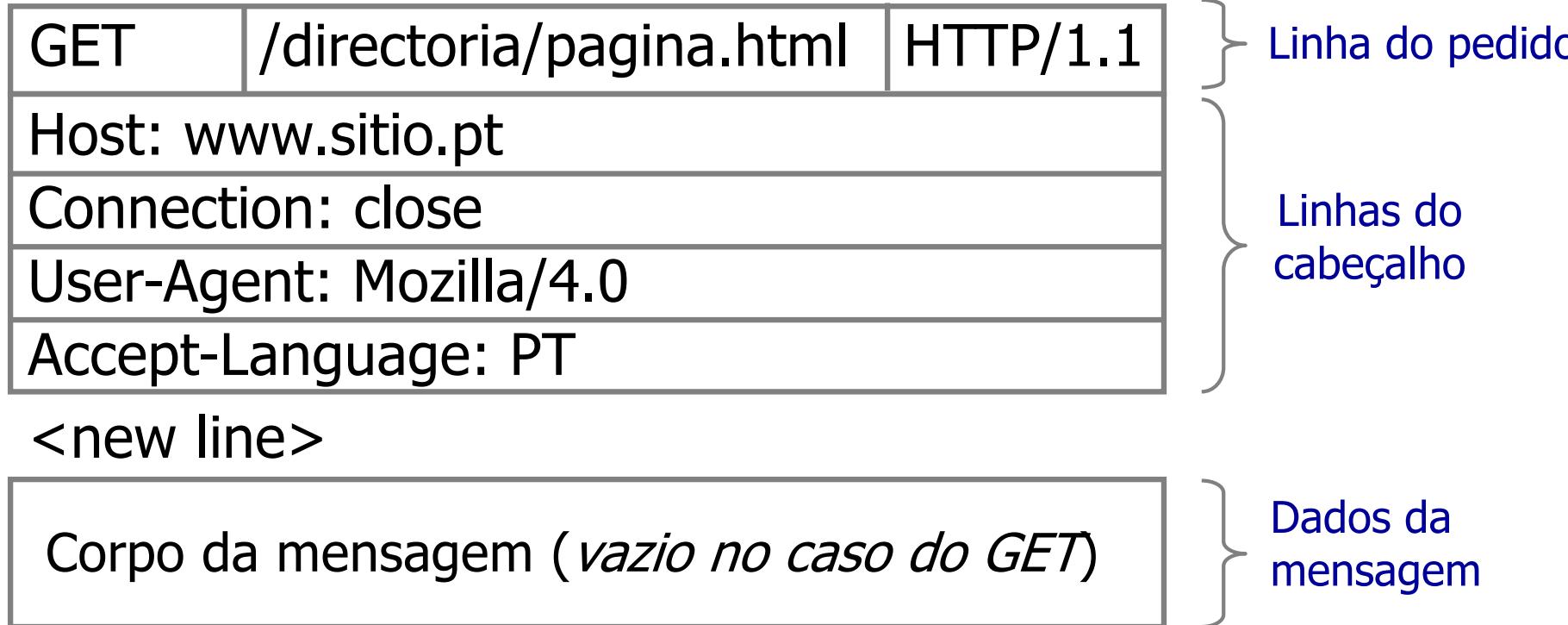
Aplicações de rede

Exemplo: HTTP

Formato dos PDU



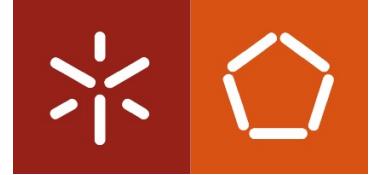
Exemplo de uma *HTTP Request Message*



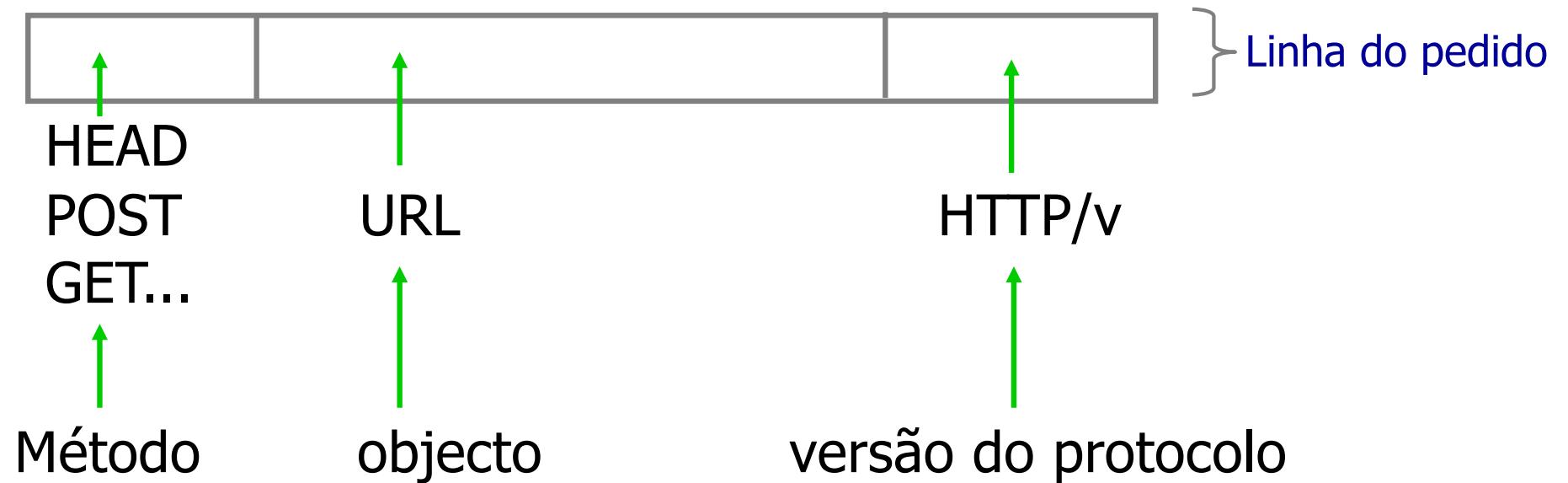
Aplicações de rede

Exemplo: HTTP

Formato dos PDU



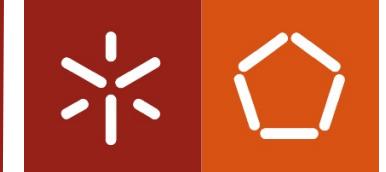
HTTP Request Message



HTTP/1.0 usa conexões TCP não-persistentes:

a conexão é terminada após o envio de cada mensagem

HTTP/1.1 usa conexões TCP persistentes, por defeito



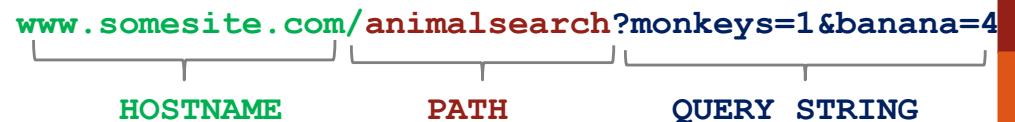
Input de dados através de formulários

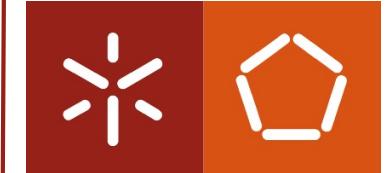
Método Post:

- É frequente as páginas Web incluírem um formulário para introdução de dados.
- Nesse caso pode utilizar-se o método POST em vez do método GET.
- O método POST é muito semelhante ao método GET, mas o objeto requerido depende do *input* introduzido pelo utilizador através de um formulário.
- O *Input* introduzido pelo utilizador é enviado para o servidor HTTP no corpo da *HTTP Request Message*, utilizando o método POST.

Método URL:

- Utiliza o método GET
- O Input é enviado para o servidor HTTP utilizando o campo URL da *HTTP Request Message*, com o método GET.





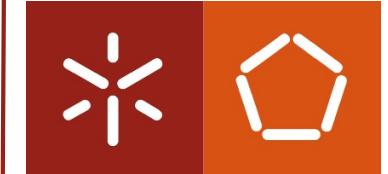
Tipos de Métodos

HTTP/1.0

- **GET**
- **POST**
- **HEAD**
 - pede ao servidor para não incluir o objeto requerido na resposta

HTTP/1.1

- **GET, POST, HEAD**
- **PUT**
 - faz o *upload* do objeto contido no corpo da mensagem na localização especificada no campo URL da mesma mensagem
- **DELETE**
 - apaga o ficheiro especificado no campo URL



Métodos HTTP: REST API Design

- Lista de operações sobre um **RECURSO** (ex: livros) é definida aproveitando a semântica dos métodos do protocolo HTTP:

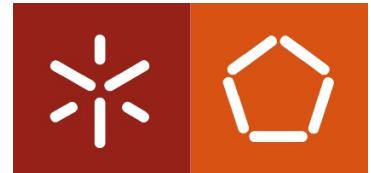
Recurso	POST (Create)	GET (Read)	PUT (Update)	DELETE (Delete)
/livros	Cria um novo livro; Pedido: objeto “livro” no corpo do HTTP Request!	Lista todos os livros; Pedido: vazio; Resposta: listagem de livros;	Atualiza um conjunto de livros passados no corpo do pedido HTTP	Apaga todos os livros; Pedido: vazio; Resposta: sucesso ou insucesso;
/livros/01	Normalmente não é usado! Erro!	Devolve o objeto que representa o livro com id 01	Se existe livro 01 então atualiza-o; Senão dá erro!	Se existe livro 01 apaga-o;

CRUD (Create / Read / Update / Delete)

Aplicações de rede

Exemplo: HTTP

Formato dos PDU



HTTP Response Message

HTTP/1.1	200	OK
Connection: close		
Date: 07 Mai 2003 11:35:15 UTC+1		
Server: Apache/1.3.0 (Unix)		
Last-Modified: 05 Mai 2003 09:23:45 UTC+1		
Content-Length: 6825		
Content-Type: text/html		

<new line>

Corpo da mensagem (objecto)

Linha do tipo
da resposta

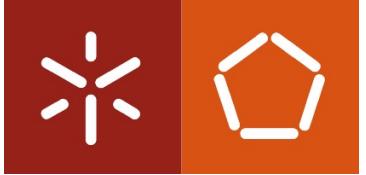
Linhas do
cabeçalho

Dados da
mensagem

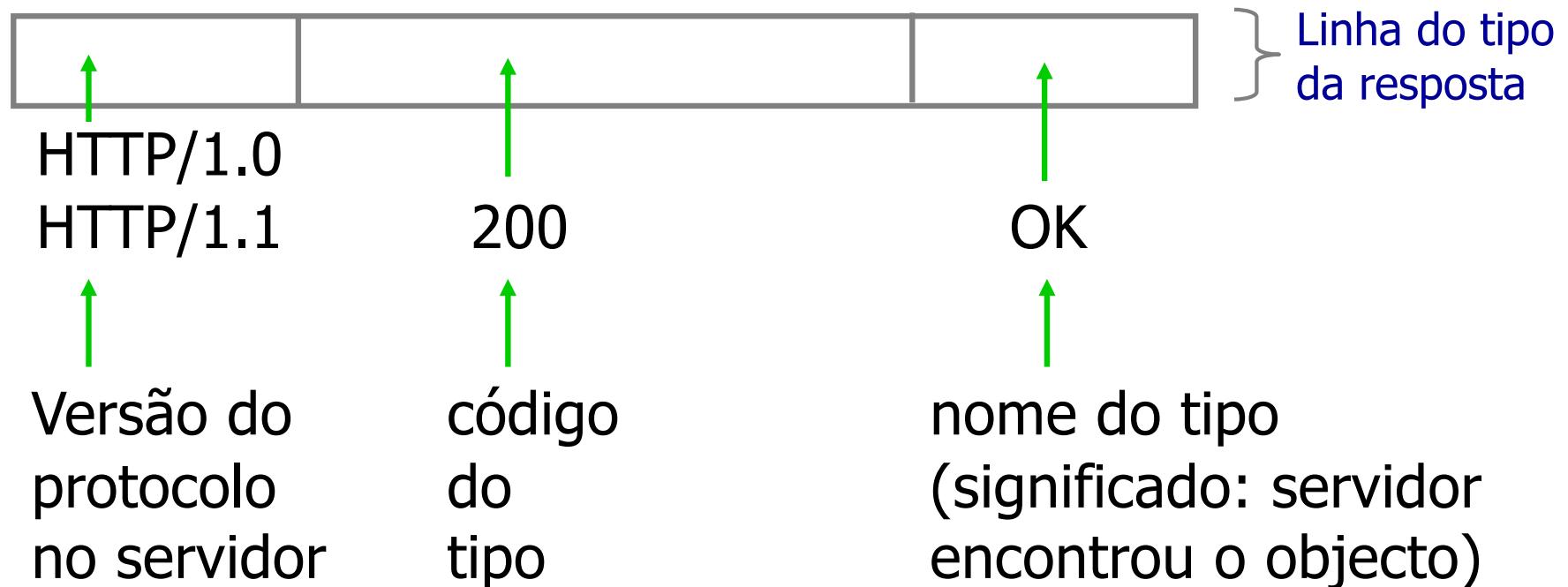
Aplicações de rede

Exemplo: HTTP

Formato dos PDU



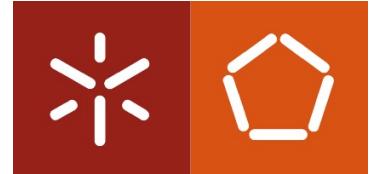
HTTP Response Message



Aplicações de rede

Exemplo: HTTP

Formato dos PDU



Alguns códigos de tipo e seu significado

200 OK

301 Moved permanently, location: xyz

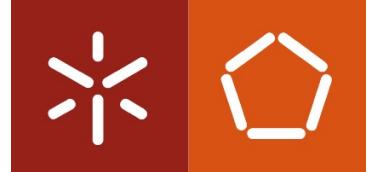
304 Not modified

400 Bad request (pedido não entendido)

401 Authorization required

404 Not found (objecto não encontrado)

505 HTTP version not supported



HTTP (command line)

```
$ http -v GET www.di.uminho.pt
```

GET / HTTP/1.1

Accept: */*

Accept-Encoding: gzip, deflate

Connection: keep-alive

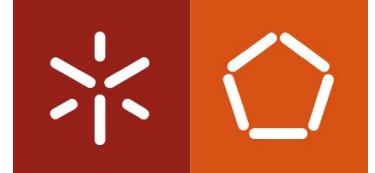
Host: www.di.uminho.pt

User-Agent: HTTPie/2.0.0

....

(ver os últimos 4 slides, para mais exemplos com API REST,

eo site do HTTPie <https://httpie.org/>)

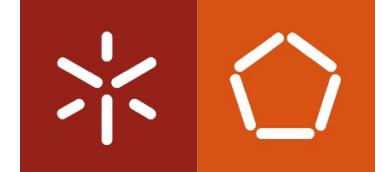


HTTP não persistente

- **Só pode ser enviado no máximo um objeto Web por cada conexão estabelecida**
- **O HTTP/1.0 utiliza HTTP não persistente**

HTTP persistente

- **Podem ser enviados múltiplos objetos Web por cada ligação estabelecida entre o cliente e o servidor.**
- **O HTTP/1.1 usa por defeito conexões persistentes**



HTTP não persistente:

- exige 2 RTTs por objecto
- O Sistema Operativo tem que reservar recursos para cada ligação TCP estabelecida
- Muitos browsers abrem ligações TCP paralelas para irem buscar os objectos referidos

HTTP persistente:

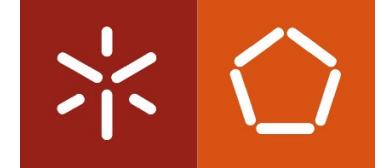
- O servidor deixa a ligação aberta depois de enviar a mensagem de resposta
- Os pedidos HTTP posteriores são enviados através da mesma ligação

Persistente sem pipelining:

- O cliente envia um novo pedido apenas quando recebe a resposta ao anterior
- Um **RTT por cada objeto referido**

Persistente com pipelining:

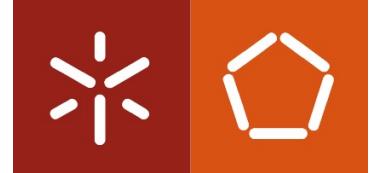
- Modo por defeito no **HTTP/1.1**
- O cliente envia os pedido assim que os encontra no objecto referenciador
- No mínimo é consumido **um RTT por todos os objetos referenciados**



Supondo que o utilizador introduziu a url `www.uminho.pt/DI/index.html`

(contém texto e referência
para imagens jpeg)

- 1a.** O cliente HTTP inicia uma conexão TCP com o servidor HTTP que está a ser executado no sistema `www.uminho.pt` e está à escuta na porta 80
 - 1b.** O servidor HTTP que está a ser executado no sistema `www.uminho.pt` e está à escuta na porta 80 aceita o pedido de conexão e avisa o cliente
 - 2.** O cliente HTTP envia uma mensagem HTTP do tipo *request message* (contendo a URL) através de um novo socket TCP. A mensagem indica que o cliente deseja o objecto Web `DI/index.html`
 - 3.** O servidor HTTP recebe a *request message* e constrói uma *response message* que contém o objeto Web requerido, enviando depois essa mensagem através do socket TCP estabelecido
- tempo

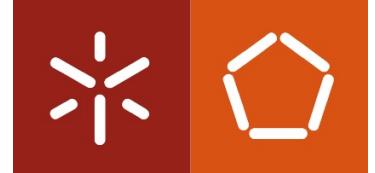


5. O cliente HTTP recebe a response message que contem o ficheiro html, mostra o ficheiro e faz o *parsing* do seu conteúdo encontrando a referênciia a vários objetos jpeg

4. O servidor HTTP pede para terminar a conexão, mas a ligação só é terminada quando o cliente receber a response message

6. Repete os passos 1-5 para cada objecto referenciado

tempo



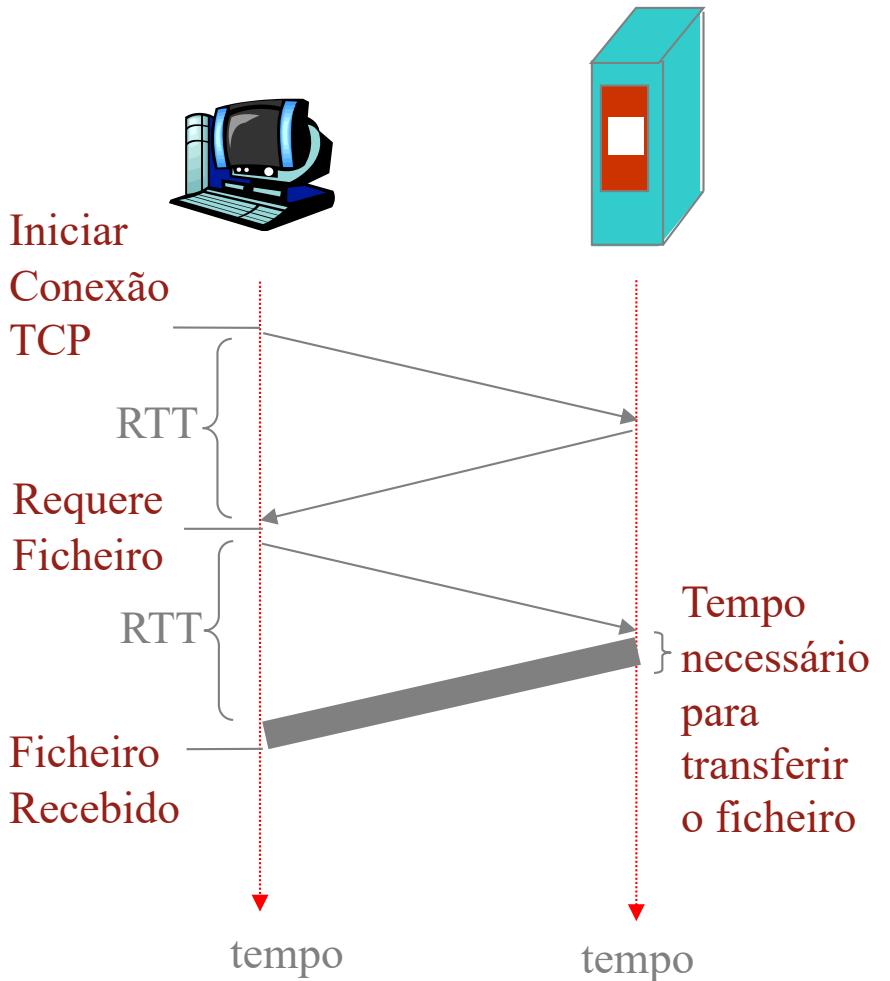
Definição de RTT: tempo que o sinal (1 bit) demora a ir do cliente para o servidor e voltar

$$(2 * \text{TempoPropagação} + N * \text{tempoEsperanasQueues} + N * \text{tempoProcessamento})$$

Tempo de Resposta

- um RTT para iniciar uma conexão TCP
- um RTT para enviar a *request message* e receber o primeiro bit da *response message*
- tempo de transmissão do ficheiro

$$\text{total} = 2\text{RTT} + \text{tempo_transmissão}$$

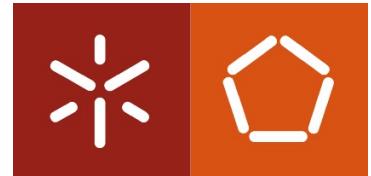


Exercício

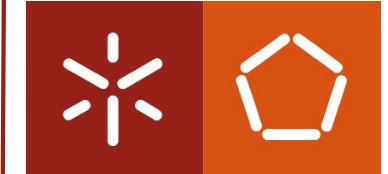


- Pretende-se estimar o tempo mínimo necessário para obter um documento da Web. O documento é constituído por 6 objectos: o objecto base HTML e cinco imagens referenciadas no objecto base. O *browser* está ligado ao servidor HTTP por uma única linha com RTT de 20 ms. O tempo mínimo de transmissão na linha do objecto base HTML é de 8 ms e o tempo mínimo de transmissão na linha de cada imagem é de 80 ms. Admita que o *browser* só pode pedir as imagens quando receber completamente o objecto base. Admita que o utilizador o utilizador sabe o endereço IP do servidor, indicando-o no *browser*. A dimensão dos pacotes de estabelecimento de ligação, de confirmação de estabelecimento de ligação e de envio dos pedidos HTTP é desprezável. Os tempos de processamento dos pacotes são também desprezáveis. Não há mais tráfego nenhum na rede.
 - Ilustrando a situação com um diagrama temporal, qual o tempo necessário para obter o documento (todos os objectos) se utilizar HTTP não persistente com um máximo de 4 ligações paralelas?
 - Ilustrando a situação com um diagrama temporal, qual o tempo necessário para obter o documento (todos os objectos) se utilizar HTTP/1.1 com *pipelining* em todos os pedidos?

Exercício



- Pretende-se estimar o atraso na recepção de um documento Web usando o protocolo HTTP. Sabemos que o atraso de ida-e-volta entre cliente e servidor é 4 ms, que o débito do caminho que une o cliente ao servidor é 1024 Kbps e que cada segmento TCP contém no máximo 128 bytes de dados. Desprezam-se os tempos de transmissão dos cabeçalhos; em particular, despreza-se o tempo de transmissão dos segmentos que não contêm dados pertencentes ao documento Web. As respostas às alíneas seguintes devem ser ilustradas com diagramas espaço-tempo
 - Se o documento consistir num único objecto base com 2048 bytes, a memória de recepção TCP for ilimitada e o TCP utilizar o mecanismo de arranque lento ("slow-start"), mudando para a fase de "congestion avoidance" quando a janela atinge os 4 segmentos, determine o atraso na recepção do documento, desde o instante em que o cliente estabelece contacto com o servidor até que o documento é recebido na totalidade.
 - Assuma, agora, que o documento Web contém 4 imagens que são referenciadas no objecto base. Cada imagem contém 1024 bytes e a versão de HTTP usada é não-persistente (1.0) suportando um máximo de 2 sessões paralelas. Determine o atraso até à recepção do documento, considerando que a largura de banda disponível é repartida equitativamente entre sessões paralelas.
 - Considere agora que usa a versão 1.1 do protocolo HTTP primeiro sem possibilidade de pedidos em sequência ("pipelining") e depois com pipelining.



Cookies: informação de estado

A maioria dos sites Web usa cookies

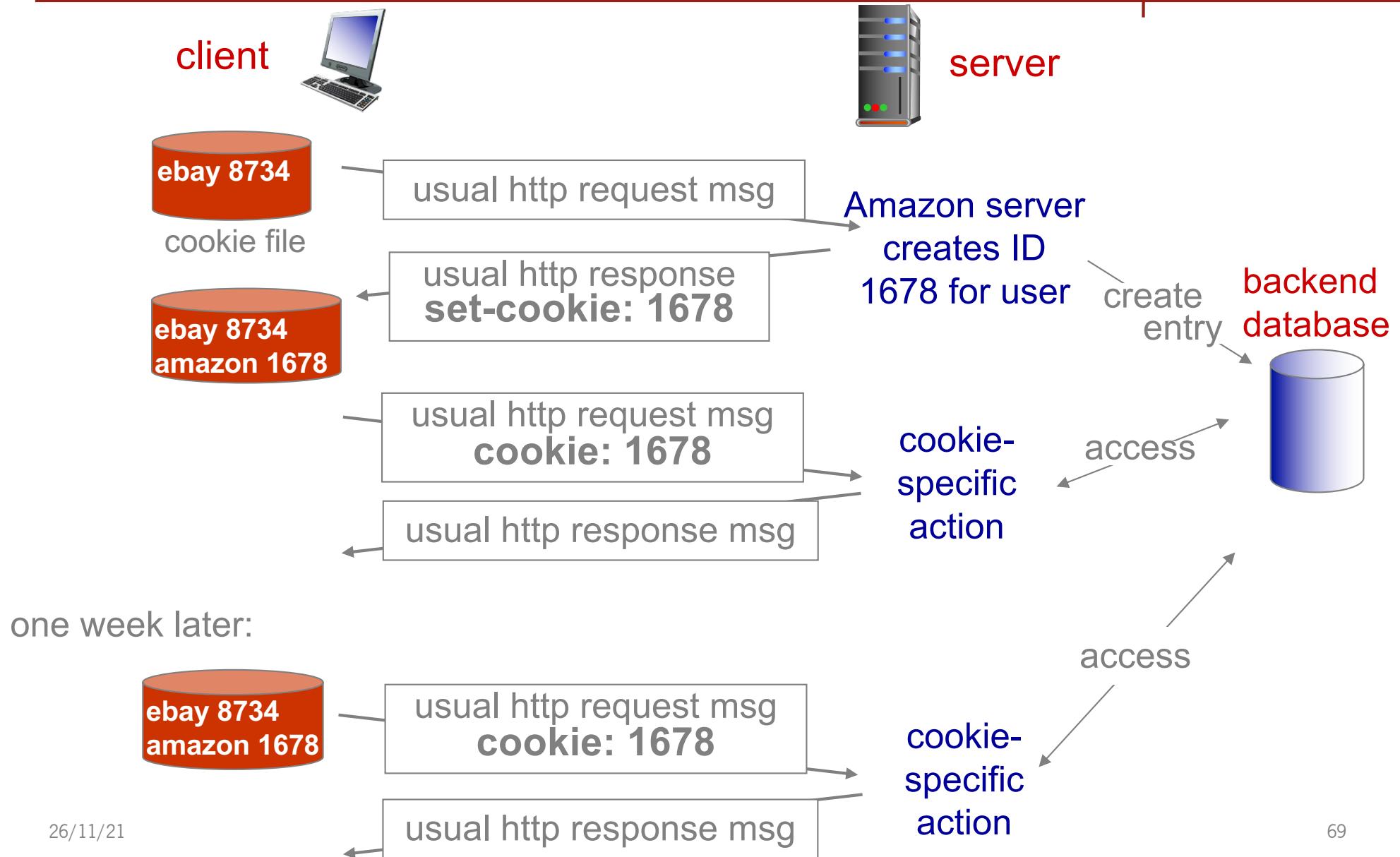
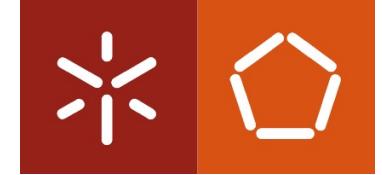
Quatro componentes:

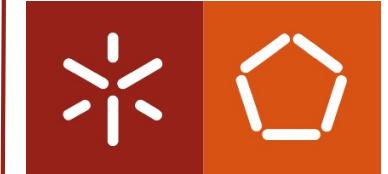
- 1) Linha com *cookie* no cabeçalho da mensagem *HTTP response*
- 2) Linha com *cookie* no cabeçalho da mensagem *HTTP request*
- 3) Ficheiro com *cookies* mantido na máquina do utilizador, gerido pelo seu browser
- 4) Uma base de dados de suporte do lado servidor Web

Exemplo:

- **Susana acede sempre à Internet a partir do seu PC**
- **visita um site de comércio electrónico pela primeira vez**
- **quando o primeiro pedido chega ao servidor Web, o servidor gera:**
 - Um Identificador (ID) único
 - Uma entrada na base de dados de suporte para esse ID

Cookies: informação de estado





Cookies: informação de estado

O que os cookies permitem:

- autorização
- cabaz de compras
- sugestões ao utilizador
- informação de sessão por utilizador (ex: Web e-mail)

efeitos colaterais

Os Cookies e a privacidade:

- os **cookies** ensinam muito aos servidores a respeito dos utilizadores e seus hábitos
- o utilizador pode fornecer nome e e-mail ao servidor

Como manter “estado”:

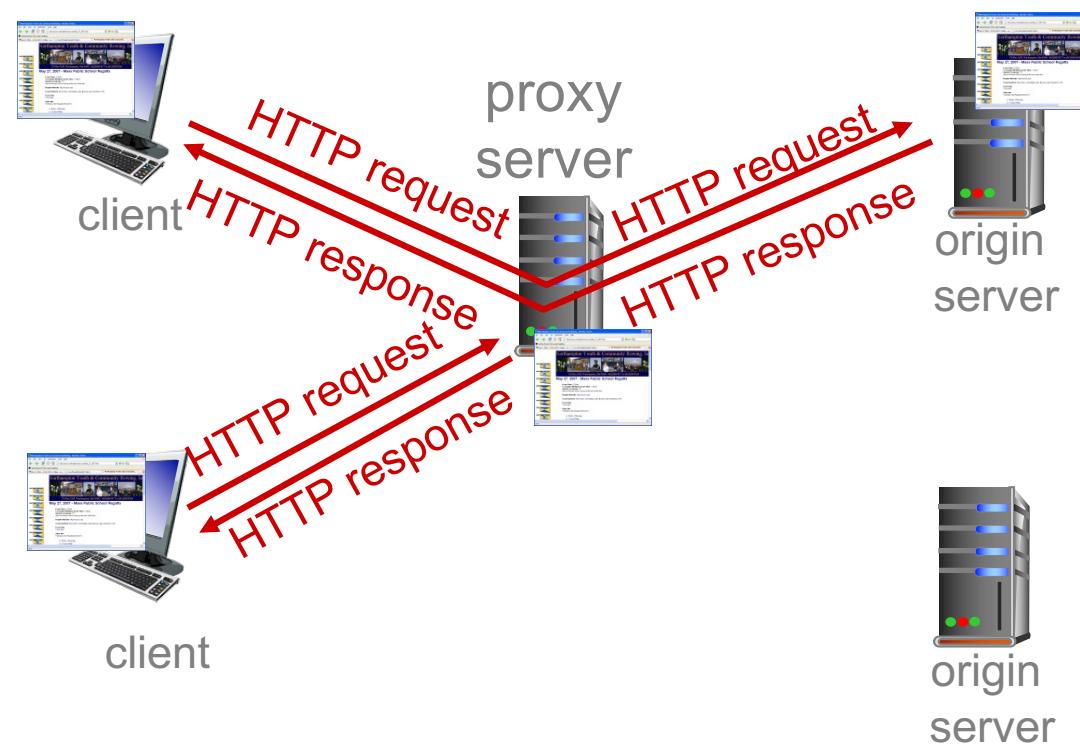
- **entidades protocolares:** guardam estado por emissor/recetor entre transações distintas
- **cookies:** forma como as mensagens http transportam a informação de estado

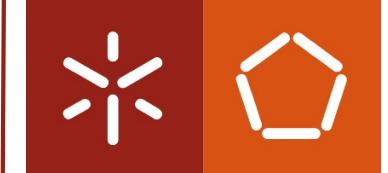


Web caches (servidor proxy)

Objectivo: satisfazer o pedido do cliente sem envolver o servidor HTTP alvo

- O utilizador **configura o cliente HTTP (browser) para aceder à Web através de um servidor proxy**
- O browser enviar todas as ***HTTP request messages*** para o **servidor proxy**
 - Se o objeto requerido está na cache do proxy o servidor proxy retorna o objeto
 - Senão o servidor proxy contacta o servidor HTTP alvo, reenvia-lhe a *HTTP request message*, aguarda a resposta que retorna ao browser



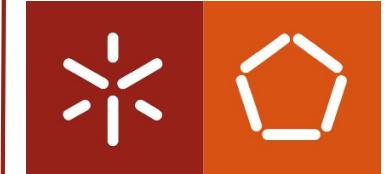


Web caching

- o **servidor proxy/cache** tem de atuar simultaneamente como cliente e como servidor
- são tipicamente instalados pelos ISP ou pelas próprias instituições (universidades, empresas, ISP residenciais, etc)

Porquê *Web caching*?

- **reduz o tempo de resposta** para os pedidos dos clientes
- **reduz o tráfego** nos links de acesso ao exterior (os mais problemáticos para a instituição).
- Internet está povoada de *caches*: permitem que fornecedores de conteúdos mais “pobres” disponibilizem efetivamente os seus conteúdos (mas isso também as redes de partilha de ficheiros P2P)



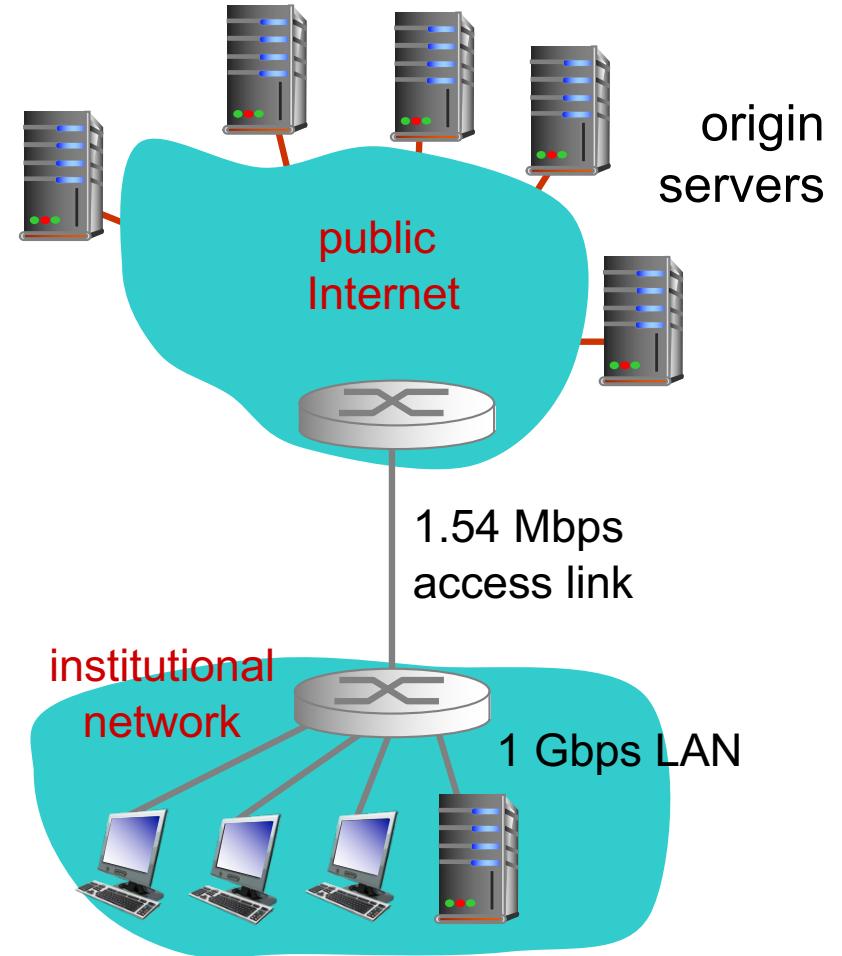
Exemplo de Caching

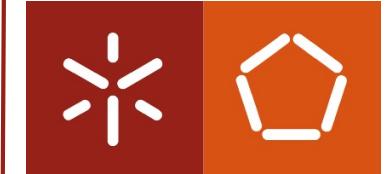
Pressupostos

- **Tamanho médio dos objetos = 100,000 bits**
- Taxa média de pedidos efectuados pelos *browsers* da instituição para servidores HTTP = 15/sec
- Tempo médio de atraso desde o pedido HTTP até à chegada da resposta = 2 sec

Consequências

- **Utilização da LAN = 15%**
 $(15 \text{ pedidos/sec}).(100\text{Kbits/pedido})/(10\text{Mbps})$
- **Utilização do Link de acesso = 99%**
 $(15 \text{ pedidos/sec}).(100\text{Kbits/pedido})/(1.54\text{Mbps})$
- **Total delay =**
= Internet delay + access delay + LAN delay
= 2 sec + minutes + milliseconds





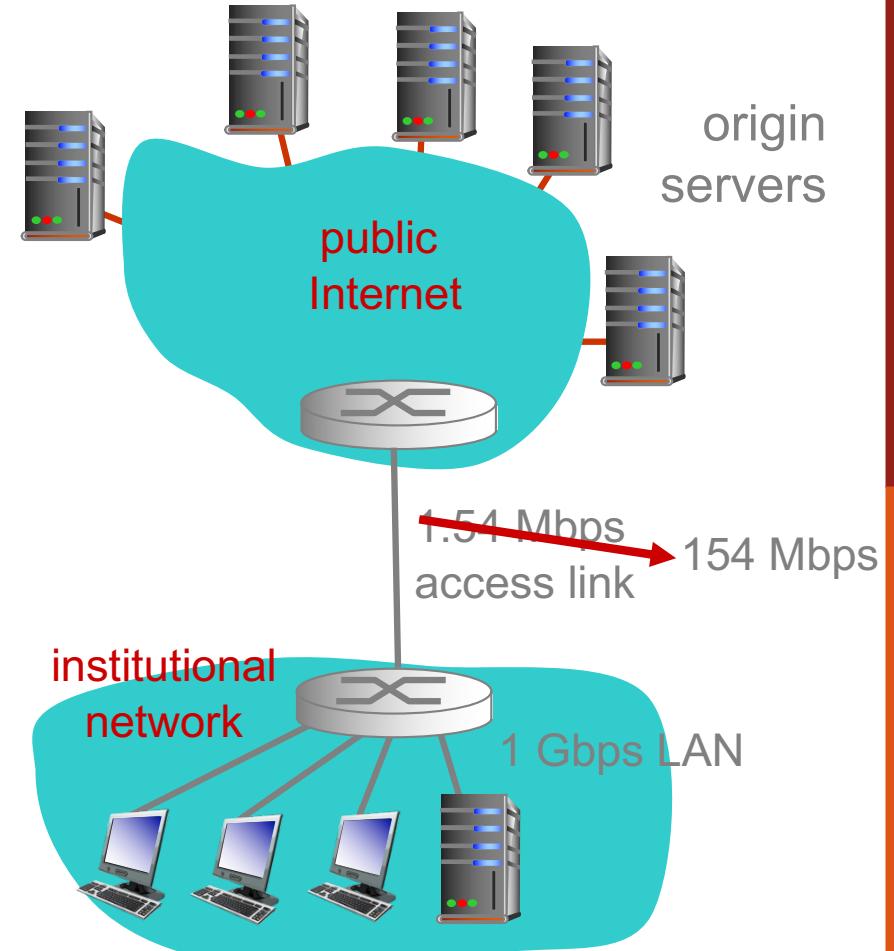
Exemplo de Caching (cont)

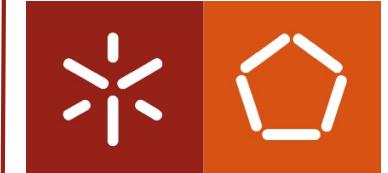
Solução possível

- Aumentar a largura de banda do link de acesso para 10 Mbps

Consequência

- Utilização da LAN = 15%
- Utilização do Link de Acesso = 15%
- Total delay = Internet delay + access delay + LAN delay
 $= 2 \text{ sec} + \text{msecs} + \text{msecs}$
- **É habitualmente muito dispendioso fazer o upgrade do link de acesso de uma instituição**





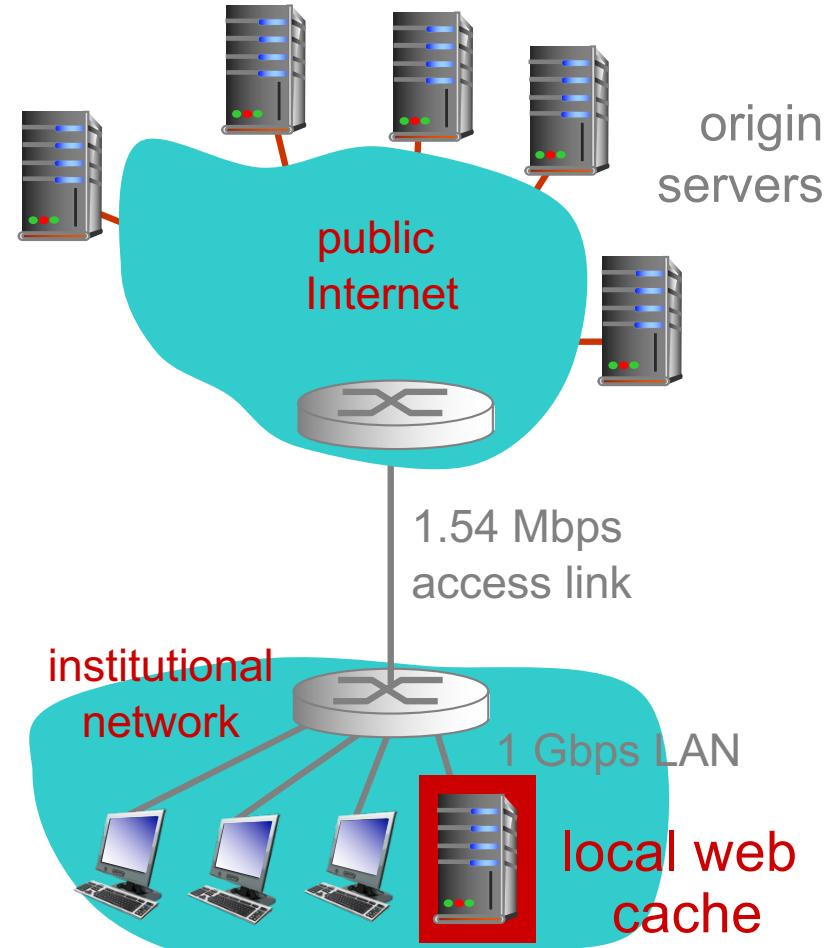
Exemplo de Caching (cont)

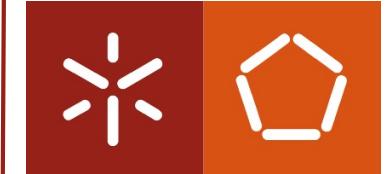
Solução possível: instalar o Web Proxy

- Se a taxa de acerto for de 0.4

Consequências

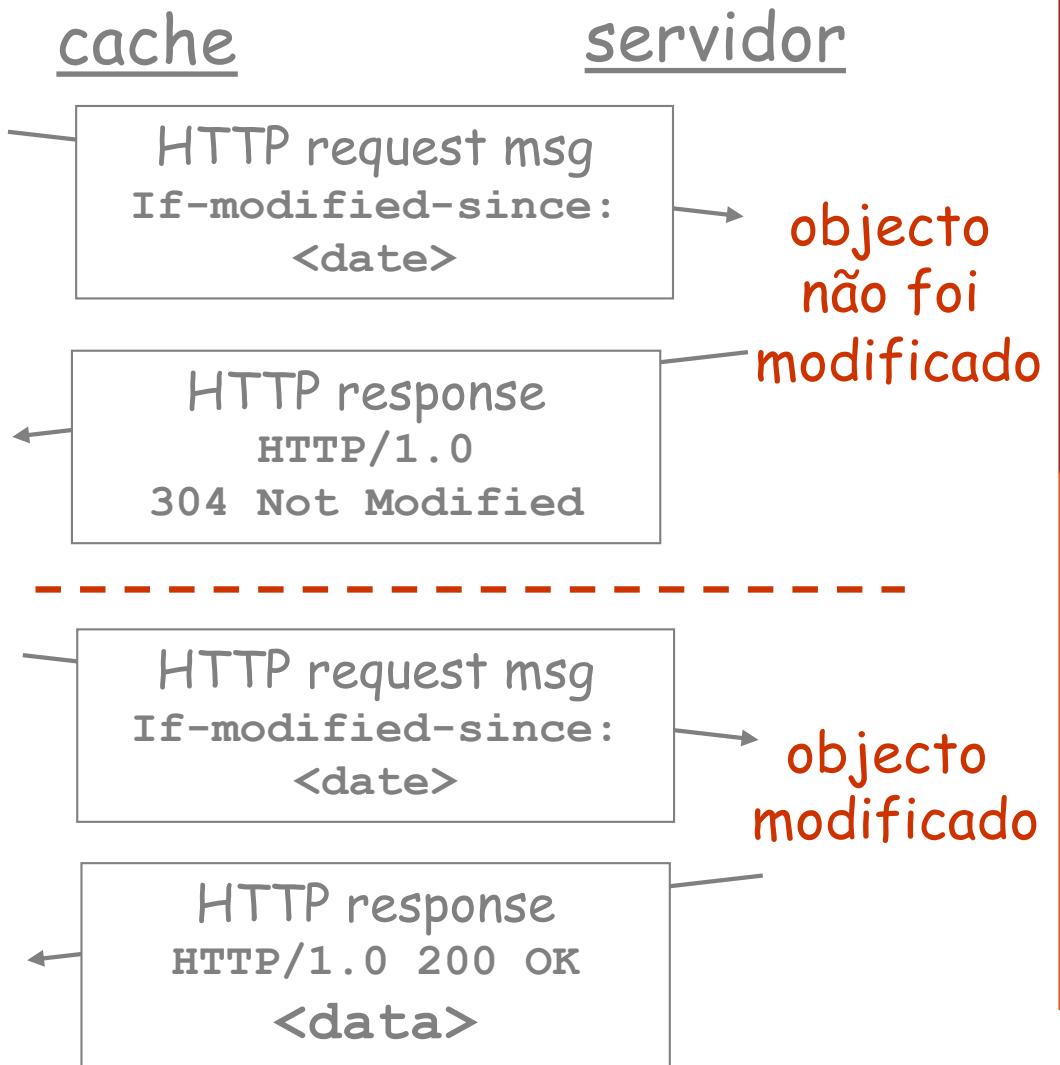
- 40% dos pedidos serão satisfeitos imediatamente
- 60% dos pedidos terão que ser redirecionados para o servidor HTTP respetivo
- A utilização do link de acesso será reduzida para 60% resultando em atrasos negligenciáveis (10 msec)
- **total avg delay =
= Internet delay + access delay + LAN delay
= .6*(2.01) secs + .4*10msec < 1.4 secs**



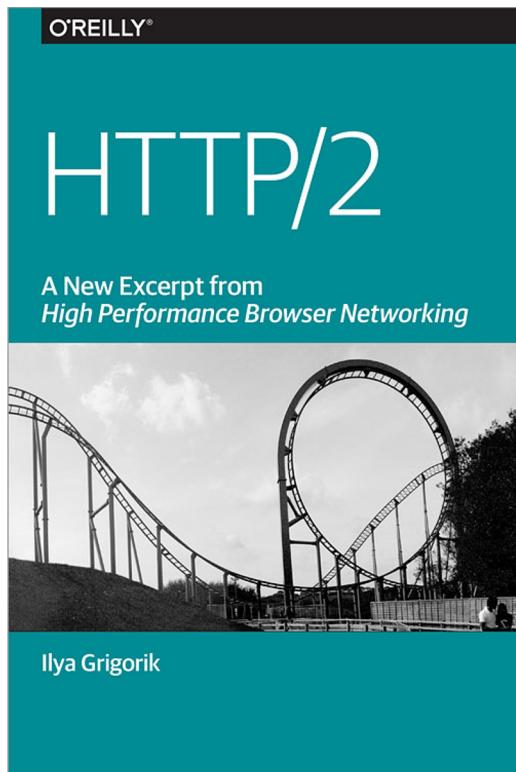


GET Condicional

- **Objectivo:** não enviar o objecto se a cópia mantida em cache está actualizada
- **cache:** inclui no cabeçalho do pedido HTTP, a data da cópia guardada na cache
`If-modified-since: <date>`
- **servidor:** resposta não contém nenhum objecto se a cópia mantida em cache estiver actualizada:
`HTTP/1.0 304 Not Modified`



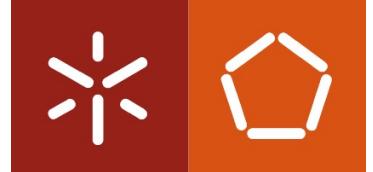
HTTP2



Comunicações por Computador
Mestrado Integrado em Engenharia Informática
3º ano/2º semestre
2017/2018



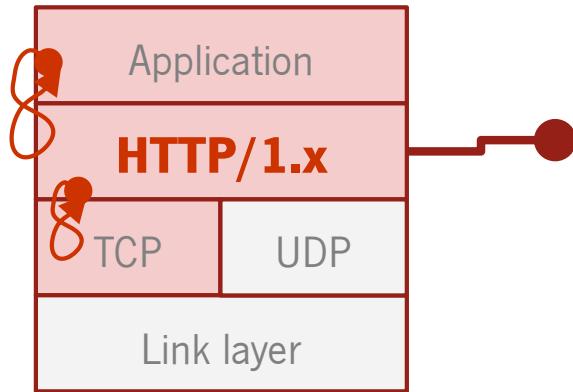
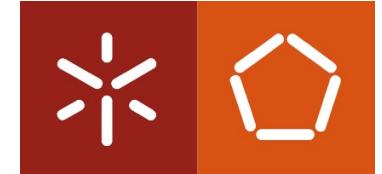
Disponível online (grátis):hpbn.co/http2
Slides: bit.ly/http2-opt



Desempenho HTTP/1.*

- **Como melhorar o desempenho do HTTP/1.*?**
- **Quais as melhores práticas, simples e eficazes, que têm sido usadas com regularidade?**
 - Reduzir o número de consultas ao DNS (*DNS Lookups*)
 - Reutilizar conexões TCP
 - Utilizar CDNs (*Content Delivery Network*)
 - Minimizar o número de redireccionamentos HTTP (*HTTP Redirects*)
 - Eliminar bytes desnecessários nos pedidos HTTP (cabeçalhos)
 - Comprimir os artefactos na transmissão (compressão corpo)
 - Cache dos recursos do lado do cliente
 - Eliminar o envio de recursos desnecessários

Problemas de desempenho do HTTP/1.*



Paralelismo limitado

- O paralelismo está limitado ao número de conexões
- Na prática, mais ou menos 6 conexões por origem

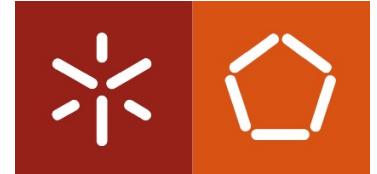
Head-of-line blocking

- Bloqueio do cabeça de fila, acumula pedidos em queue e atrasa a solicitação por parte do cliente
- Servidor obrigado a responder pela ordem (ordem restrita)

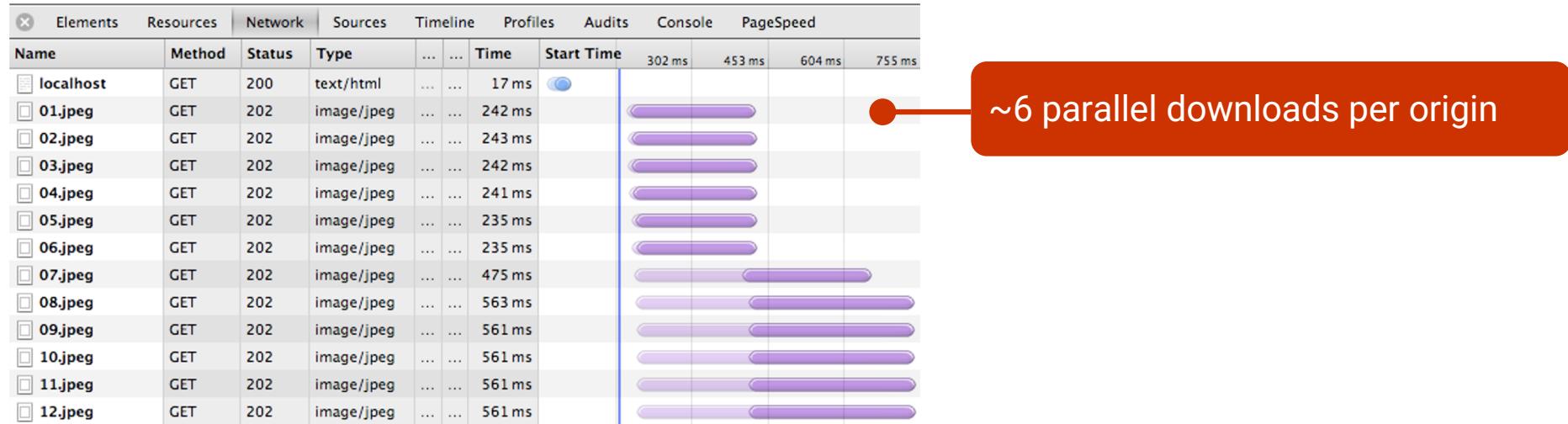
Overhead protocolar é elevado

- Metadados do cabeçalho não são compactados
- Aproximadamente 800 bytes de metadados por pedido, mais os cookies

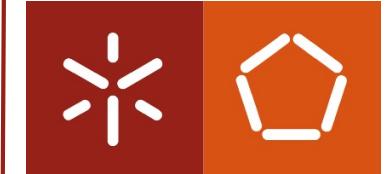
Problemas de desempenho do HTTP/1.*



- Paralelismo é limitado pelo número de conexões...



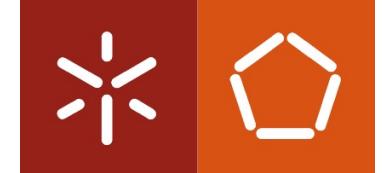
- Cada conexão implica overhead de handshake inicial
- Se for HTTPS, ainda tem mais um overhead do handshake TLS
- Cada conexão gasta recursos do lado do servidor
- As conexões competem umas com as outras



HTTP/1.* - Truques do lado do servidor

- Porque não subdividir em N sub-domínios, em vez de um único domínio por servidor? (***domain sharding***)
 - Aumenta o paralelismo — passamos a ter 6 conexões por subdomínio
 - Aumenta as consultas ao DNS...
 - Mais servidores, competição nas conexões, complexidade nas aplicações
- Reduzir pedidos → concatenar objetos (***concatenated assets***)
 - Vários CSS ou vários JS num único objeto! Resulta...
 - Atrasa o processamento no cliente, pode dificultar o uso da cache
- Incluir recursos em linha no HTML (***inline objects***)
 - Os mesmos objetivos do anterior: reduzir pedidos, antecipar conteúdos...
 - Os mesmos problemas: atrasa processamento no cliente, dificulta o uso da cache

HTTP2



- Em meados de 2009, a Google inicia o seu projeto **SPDY!**
 - Objetivo n°1: reduzir em 50% o tempo de carregamento de página (**PLT Page Load Time**)
 - Outros objetivos:
 - Evitar que os autores Web tenham de mexer nos conteúdos
 - Minimizar o tempo de implantação e as alterações na infraestrutura
 - Desenvolver em parceria com a comunidade Open Source
 - Teste com dados reais que validem ou invalidem o protocolo
- Cientes: Firefox, Opera e Chrome aderiram rapidamente...
- Servidores: Twitter, Facebook, e Google, claro!...
- E o **IETF** (Internet Engineering Task Force)?
 - Teve de ir atrás, a reboque, e formar um grupo de trabalho **HTTP/2**

HTTP2



- Normalizado em menos de 3 anos!! Muita pressão...



Mid 2009: SPDY introduced as an experiment by google

Mar, 2012: Firefox 11 had support, turned on by default in version 13

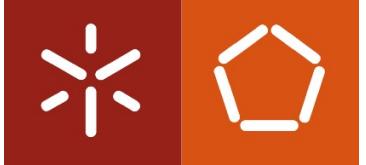
Mar, 2012: Call for proposals for HTTP/2 – resulted in 3 proposals but SPDY was chosen as the basis for H/2

Nov, 2012: First draft of HTTP/2 (based on SPDY)

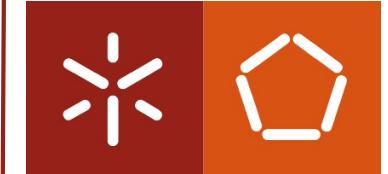
Aug, 2014: HTTP/2 draft-17 and HPACK draft-12 are published

Aug, 2014: Working Group last call for HTTP/2

Feb, 2015: (IESG) Internet Engineering Steering Group approved HTTP/2



- **HTTP2 é uma extensão e não uma substituição do HTTP/1.1**
 - Não se mexe nos métodos, URLs, headers, códigos de resposta, etc.
 - **Semântica** – para a aplicação – deve é a mesma!
 - Não há alterações na **API aplicacional...**
- **Alvo → as limitações de desempenho das versões anteriores**
 - Primeiras versões do HTTP foram desenhadas para serem de fácil implementação!
 - Clientes HTTP/1.* obrigados a lançar várias conexões em paralelo para baixar a latência.
 - Não há compressão nem prioridades
 - **Mau uso** da conexão TCP de suporte!...



HTTP2 – Tudo num único slide!

1. Uma única conexão TCP!

2. Request → Stream

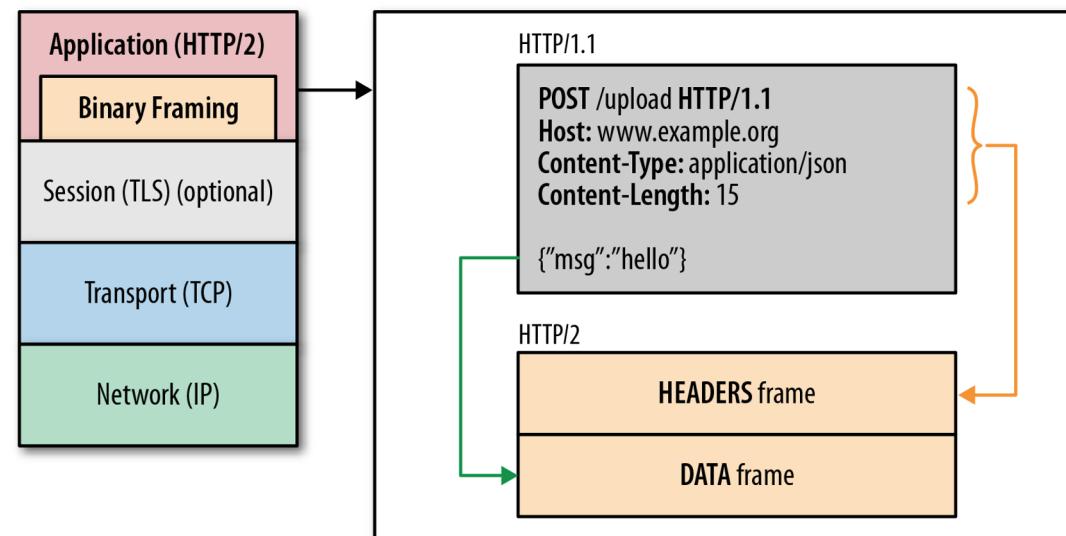
- Streams são multiplexadas!
- Streams são priorizadas!

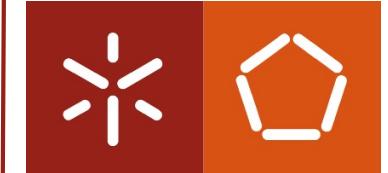
3. Camada de “framing”

binário

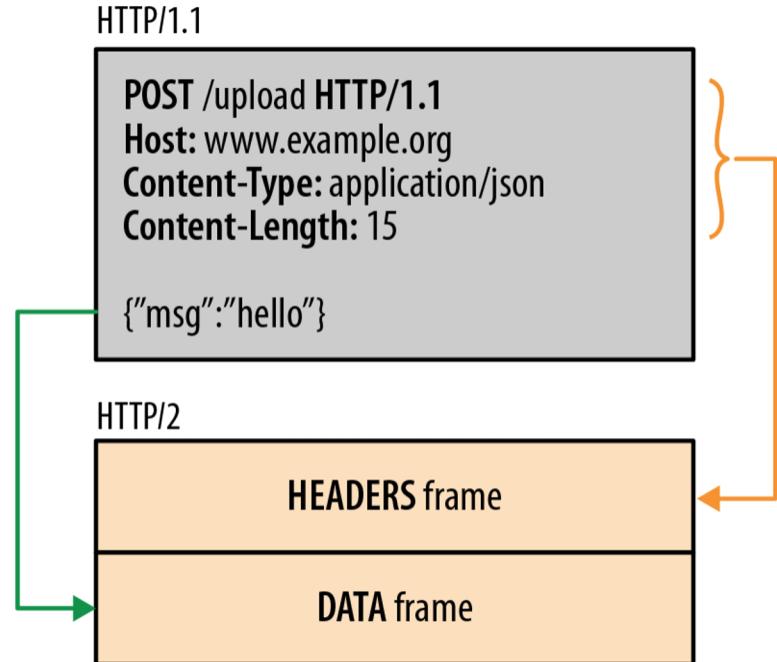
- Priorização
- Controlo de Fluxo
- Server push

4. Compressão do cabeçalho (HPACK)





HTTP2 – “Framing” binário



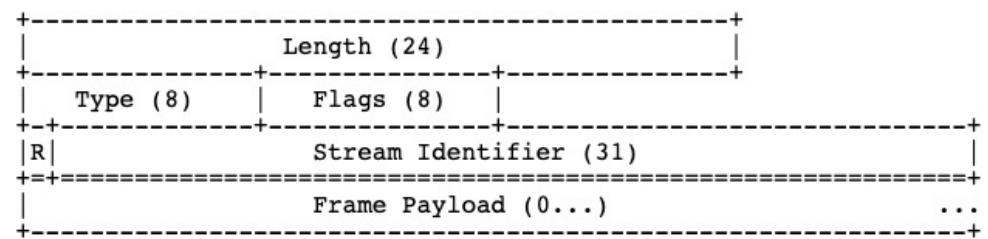
- **Mensagens HTTP** são divididas em

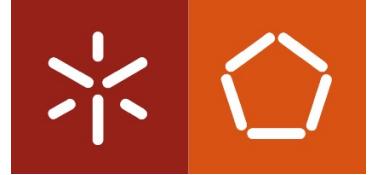
- **Mensagens HTTP** são divididas em
uma ou mais **frames**

- HEADERS para metadados
- DATA para dados (payload)
- RST_STREAM para cancelar
- ...

- Cada **frame** tem um cabeçalho comum

- 9-byte, com tamanho à cabeça
- De parsing fácil e eficiente

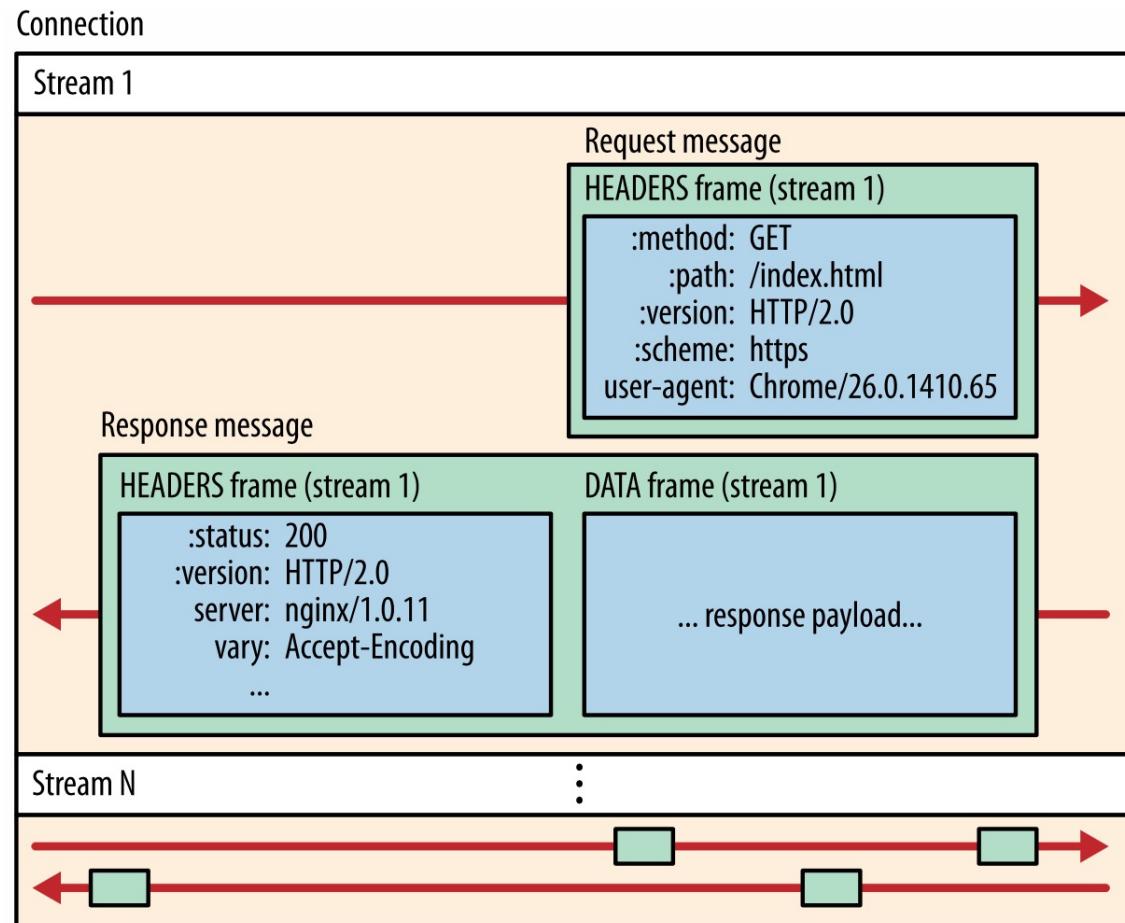


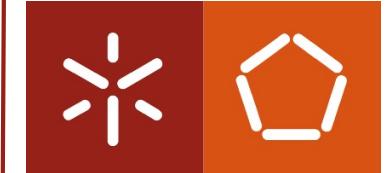


HTTP2 – “Framing” binário

- **Terminologia HTTP2**

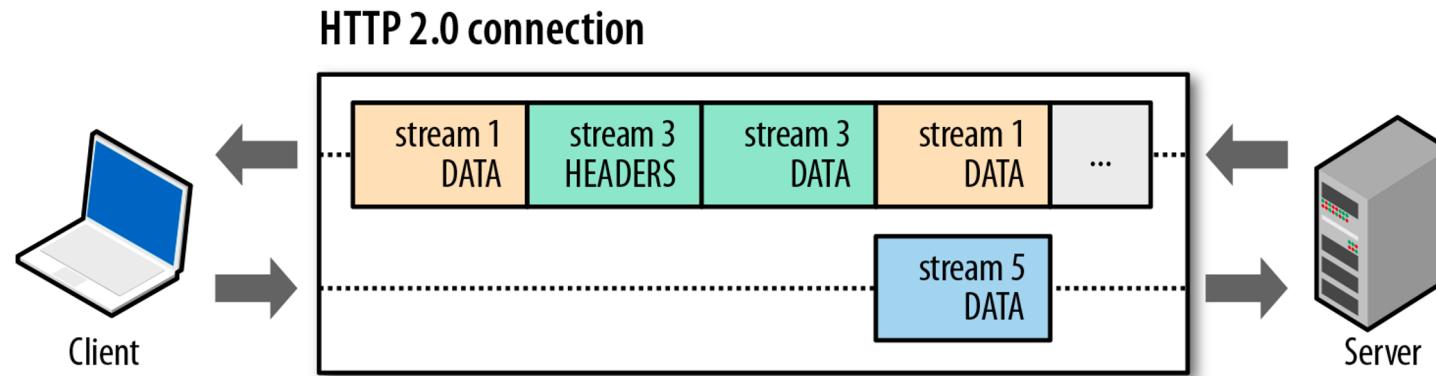
- **Stream** – um fluxo bidirecional de dados, dentro de uma conexão, que pode carregar uma ou mais mensagens
- **Mensagem** - Uma sequência completa de *frames* que mapeiam num pedido ou numa resposta HTTP
- **Frame** – A unidade de comunicação mais pequena no HTTP2, contendo um cabeçalho que no mínimo identifica a *Stream* a que pertence





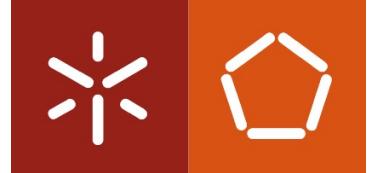
HTTP2 – fluxo de dados

- Fluxo de dados numa conexão HTTP2



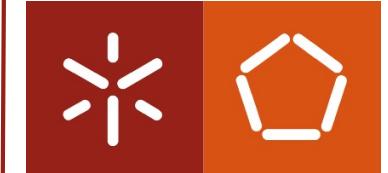
As streams são multiplexadas porque as **frames** pode ser intercaladas umas com as outras!

- Todas as frames (ex: **HEADERS**, **DATA**, etc.) são enviadas numa única conexão TCP
- As frames são entregues por prioridades, tendo em conta os pesos das streams e as dependências entre elas!
- As frames **DATA** estão sujeitas a um controlo de fluxo por stream e por conexão



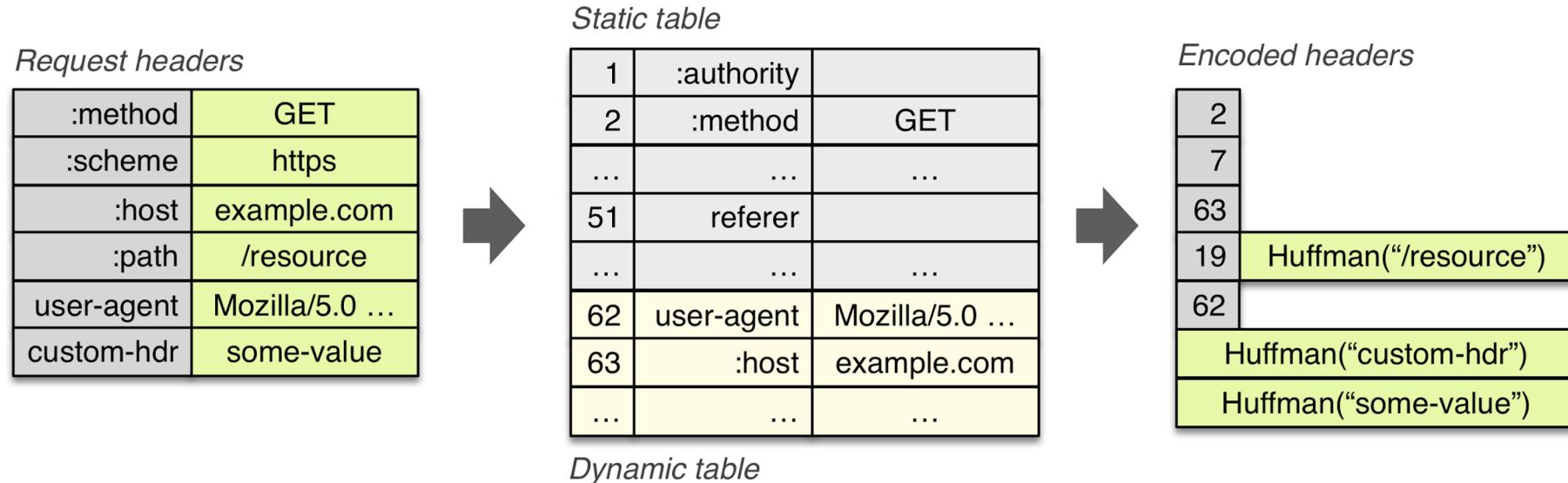
HTTP2 – Tipos de frames

- As frames definidas no RFC7540 são:
 - HEADERS – headers de um pedido ou de uma resposta
 - DATA – corpo dos objetos (dados)
 - PRIORITY – define a prioridade da stream para o originador
 - RST_STREAM – permite o término imediato da stream
 - SETTINGS – para definir parâmetros de configuração
 - SETTINGS_HEADER_TABLE_SIZE, SETTINGS_ENABLE_PUSH,
SETTINGS_MAX_CONCURRENT_STREAMS, SETTINGS_INITIAL_WINDOW_SIZE,
SETTINGS_MAX_FRAME_SIZE, SETTINGS_MAX_HEADER_LIST
 - PUSH_PROMISE – permite o push de conteúdos
 - WINDOW_UPDATE – permite reajuste da janela de fluxo da stream
 - CONTINUATION – para prolongar frames como HEADERS ou outros
 - PING, GOAWAY...

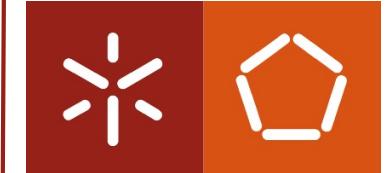


HTTP2 – Compressão do cabeçalho

- **HPACK**



- Valores literais (texto) são codificados com **código de Huffman** estático
- Tabela **indexação estática** → por ex: “2” corresponde a “method: GET”
- Tabela **indexação dinâmica** → Valores enviados anteriormente pode ser indexados!



HTTP2 – Server “push”



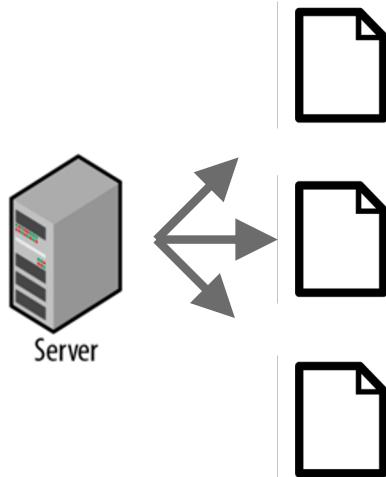
Server: “You asked for `/product/123`, but you’ll need `app.js`, `product-photo-1.jpg`, as well... I promise to deliver these to you. That is, unless you decline or cancel.”

- Maior granularidade no envio de recursos
 - Evita o inlining e permite caching eficiente dos recursos
 - Permite multiplexar e definir prioridades no envio dos recursos
 - Precisa de controlo de fluxo, para o cliente dizer basta/quero mais



HTTP2 – Server “push”

- Há espaço para estratégias de “Server push” inteligente
- Ex: implementação Jetty

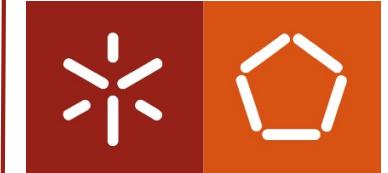


1. Servidor observa o tráfego de entrada

- Constrói um modelo de dependências baseado no campo **Referer** do cabeçalho (ou outras):
 - e.g. index.html → {style.css, app.js}

2. Servidor inicia um push inteligente de acordo com as dependências que aprendeu

- client → GET index.html
- server → push style.css, app.js, index.html



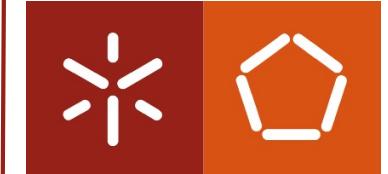
HTTP2 – Controlo de fluxo



I want image geometry and preview, and I'll fetch the rest later...

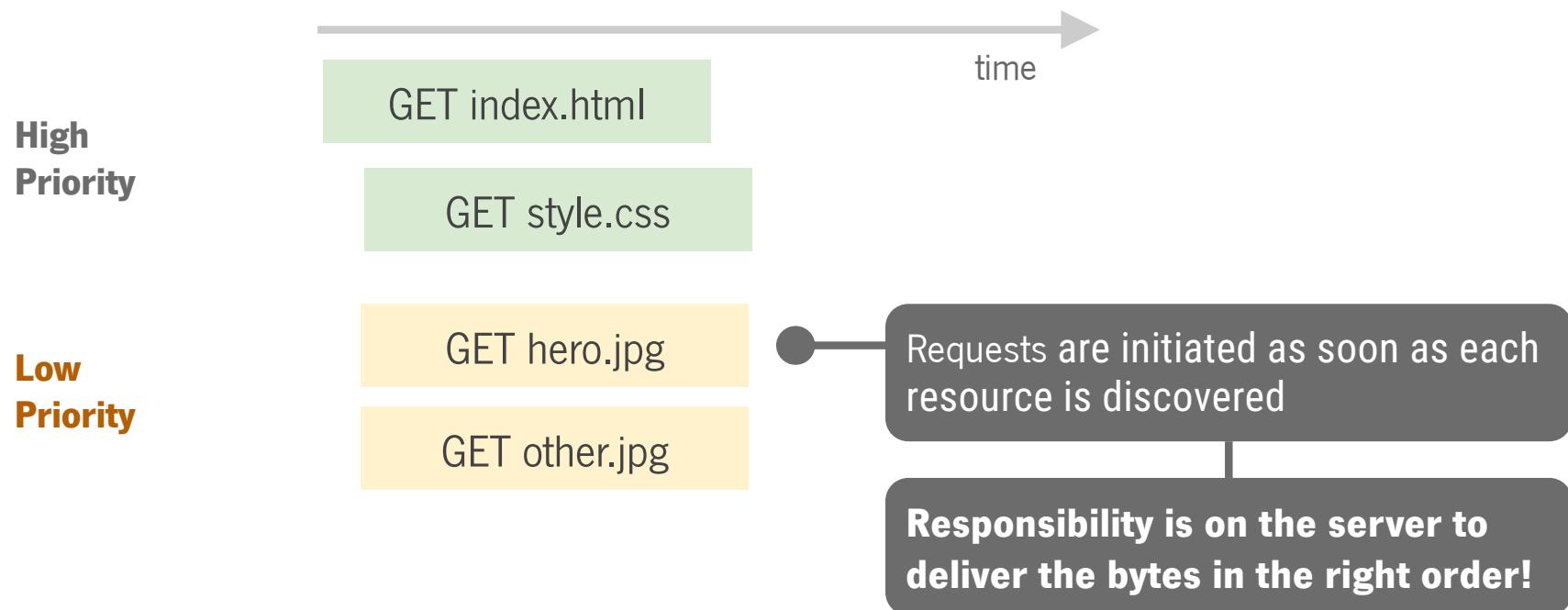
- **Client:** "I want first 20KB of photo.jpg"
- **Server:** "Ok, 20KB... pausing stream until you tell me to send more."
- **Client:** "Send me the rest now."

- Permite ao cliente fazer uma pausa na *stream* e retomar o envio mais tarde
- Controlo de fluxo baseado num sistema de créditos (janela):
 - Cada frame do tipo **DATA** decrementa o valor
 - Cada frame do tipo **WINDOW_UPDATE** atualiza o valor

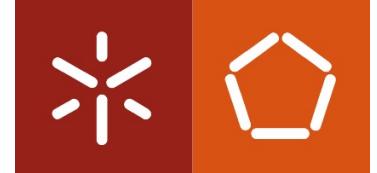


HTTP2 – priorização

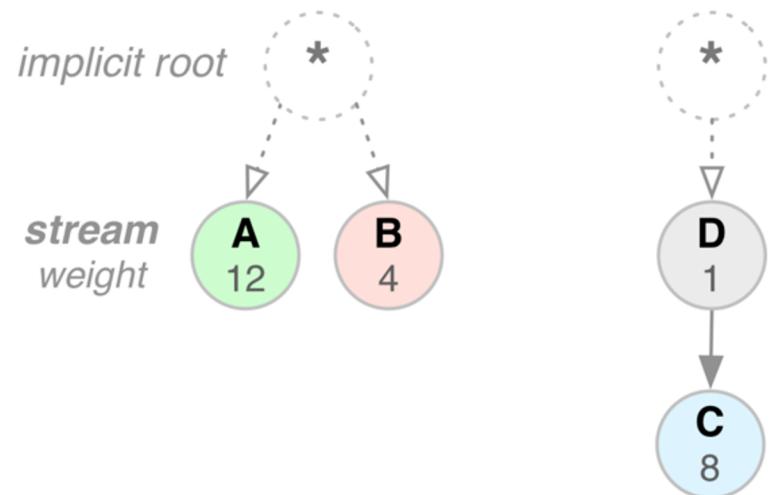
- Priorização é fundamental para um *rendering* eficiente!
- Com HTTP2, o cliente define as prioridade e faz logo os pedidos; cabe ao servidor entregar os conteúdos com a prioridade certa



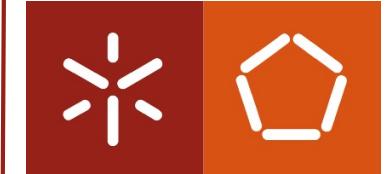
HTTP2 – pesos e dependências



- Exemplo: stream A deve ter 12/16 e a B 4/16 dos recursos totais
- Exemplo: stream D deve ser entregue antes da stream C



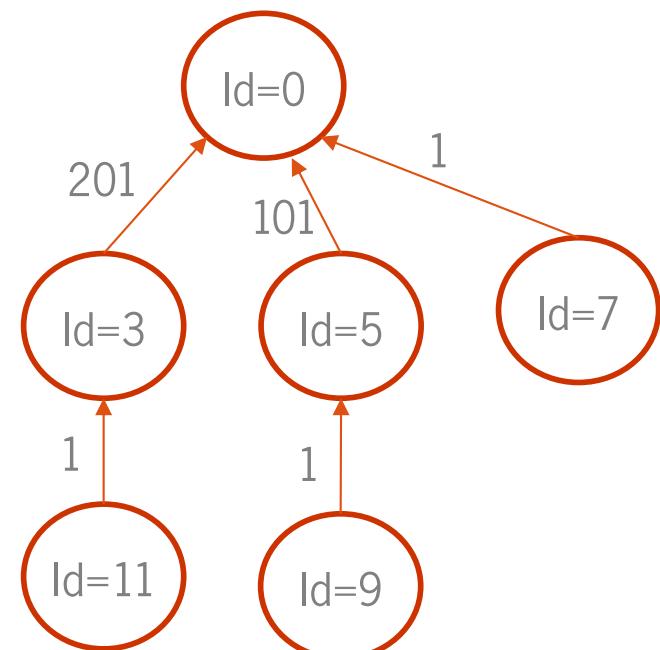
- **Cada stream pode ter um peso**
 - [1-256] integer value
- **Cada stream pode ter uma dependência**
 - ... uma outra stream ID

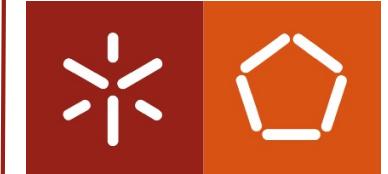


HTTP2 – pesos e dependências

- **Ex: Pesos e dependências definidos na biblioteca “nghttp2”**

- 5 PRIORITY *frames* para criar
5 *streams* adormecidas 3, 5, 7, 9 e 11
e respetivas dependências
- A *stream* 0 não existe (apenas raiz)
- O HTML base → *stream* 11
- CSS, JS referenciados no
<head> → *stream* 3, peso 2
- CSS,JS referenciados no
<body> → *stream* 5, peso 2
- *Images* → *stream* 11, peso 12
- Outros → *stream* 11, peso 2





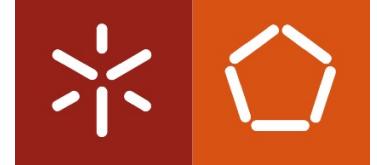
HTTP2 – Negociação protocolar

- Três formas que o cliente tem para usar HTTP2 (não podendo assumir que todos os servidores são HTTP2):

1. Começando em HTTP/1.* e pedido "upgrade" da conexão
 - Semelhante ao mecanismo usado para os WebSockets
2. Usando HTTPS e negociando o protocolo HTTP2 durante o *handshake* TLS inicial
3. Sabendo que o servidor é HTTP2 – envia sequencia inicial HTTP2

NOTA: A Google e outros defendiam que destes três mecanismos só se deveria usar sempre o 2 (HTTPS). Foi o IETF que impôs os restantes...

HTTP2 – Negociação protocolar



- **http:// pode ser servido tanto em HTTP/1.* como em HTTP2**
- **Mecanismo de Upgrade de uma conexão HTTP/1.*:**

```
GET /page HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c ①
HTTP2-Settings: (SETTINGS payload) ②
```

```
HTTP/1.1 200 OK ③
Content-length: 243
Content-type: text/html

(... HTTP/1.1 response ...)
```

(or)

```
HTTP/1.1 101 Switching Protocols ④
Connection: Upgrade
Upgrade: h2c
```

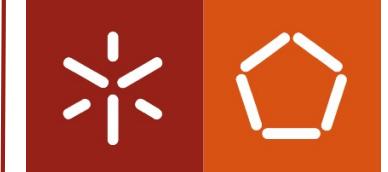
```
(... HTTP/2 response ...)
```

1. Cliente começa em HTTP1.1 e pede upgrade para HTTP2

2. Settings codificados em BASE64

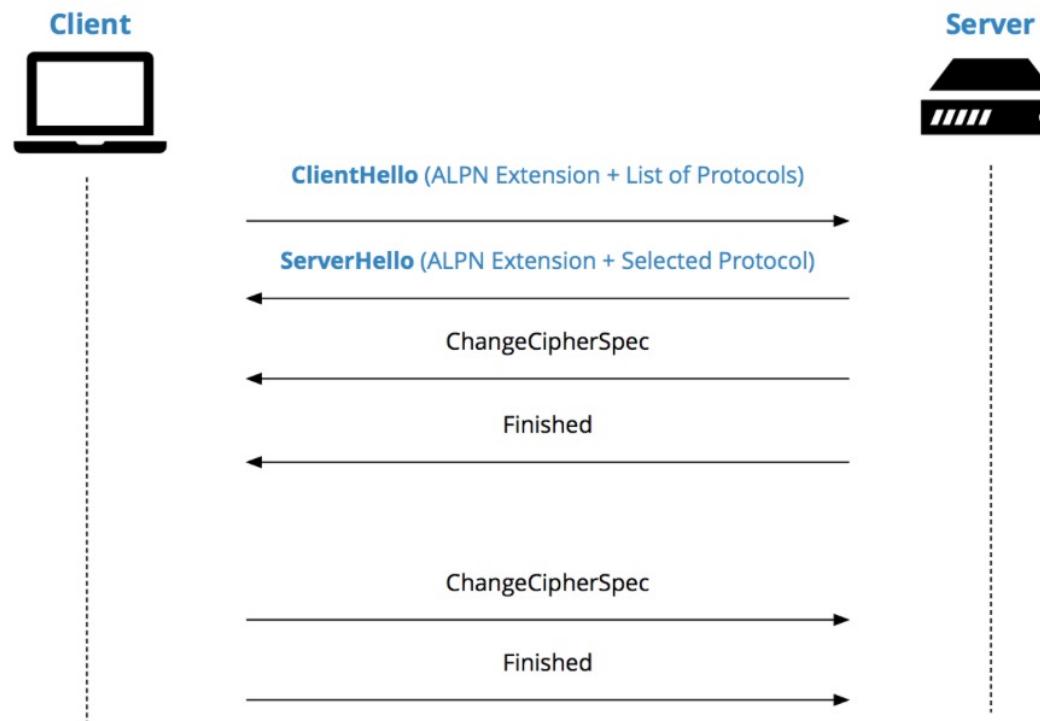
3. Servidor declina pedido, respondendo em HTTP/1.1

4. Servidor aceita pedido para HTTP2 e começa Framing binário



HTTP2 – Negociação protocolar

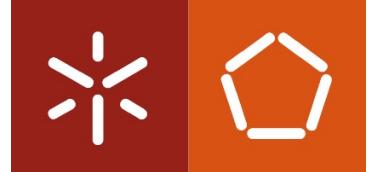
- **https://** pode ser servido quer em HTTP/1.* ou HTTP2
- Com HTTPS, negoceia-se o protocolo na fase de *Handshake* do TLS, ao mesmo tempo que se migra para conexão segura:





HTTP2 – Negociação protocolar

- É possível começar logo em HTTP2 se e só se o cliente souber que o servidor fala HTTP2:
 - Enviar a sequência de 24 octetos:
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
 - Que corresponde a """PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n""", logo seguido de uma frame de SETTINGS para definir os parâmetros da conexão HTTP2

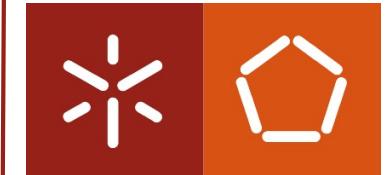


HTTP2 – Testes

- Fazer demo!
- Experimentar URLs:
 - <https://http2.akamai.com/demo>
 - <http://www.http2demo.io/>
 - <https://http2.golang.org/serverpush>
- Usar o magnífico `nghttp2` (<http://nghttp2.org/>)

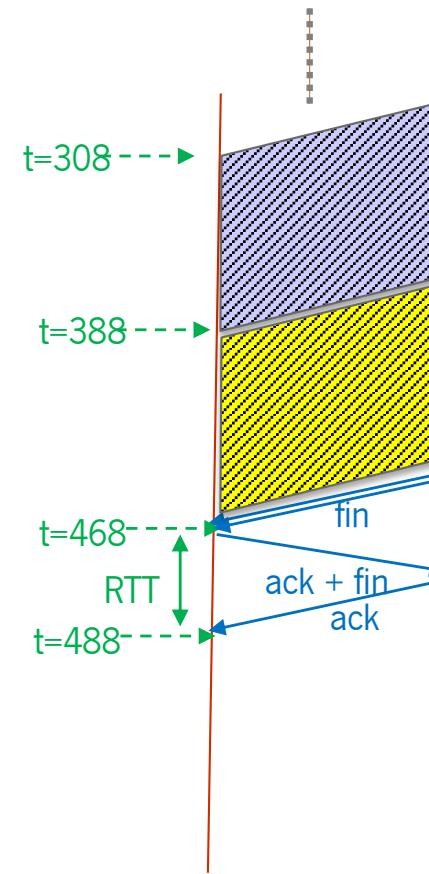
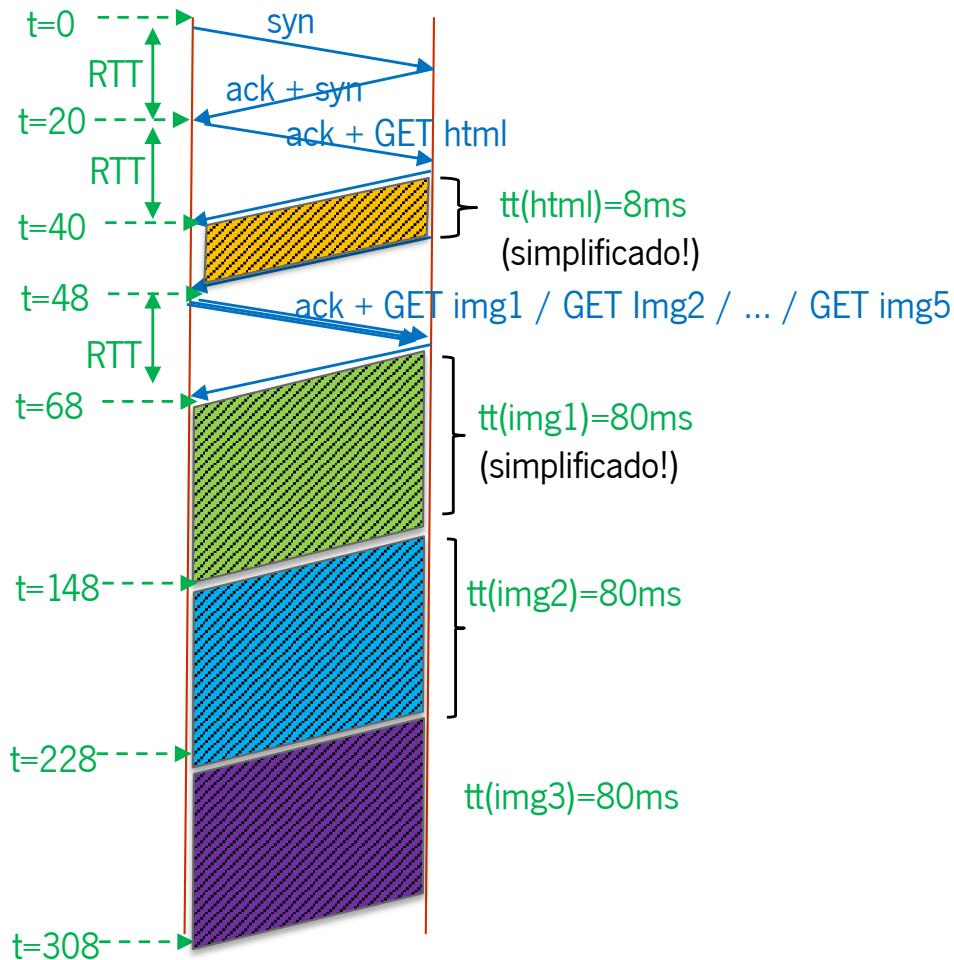
Exemplo:

```
$ nghttp -vv -a -n -y -s https://http2.golang.org/serverpush
```

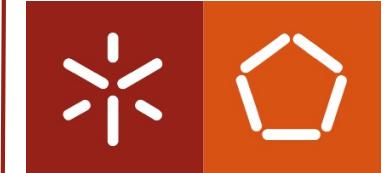


Relembrar o exercício HTTP

c) HTTP/1.1 persistente, com pipeline, sem conexões em paralelo

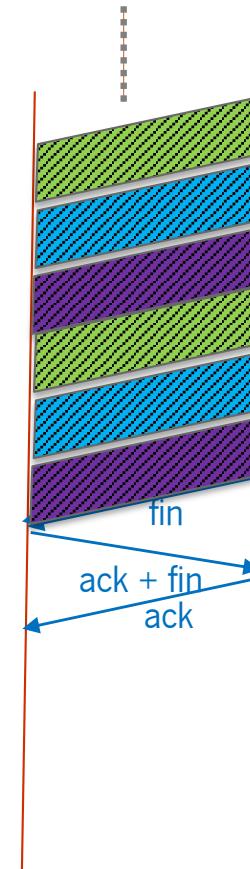
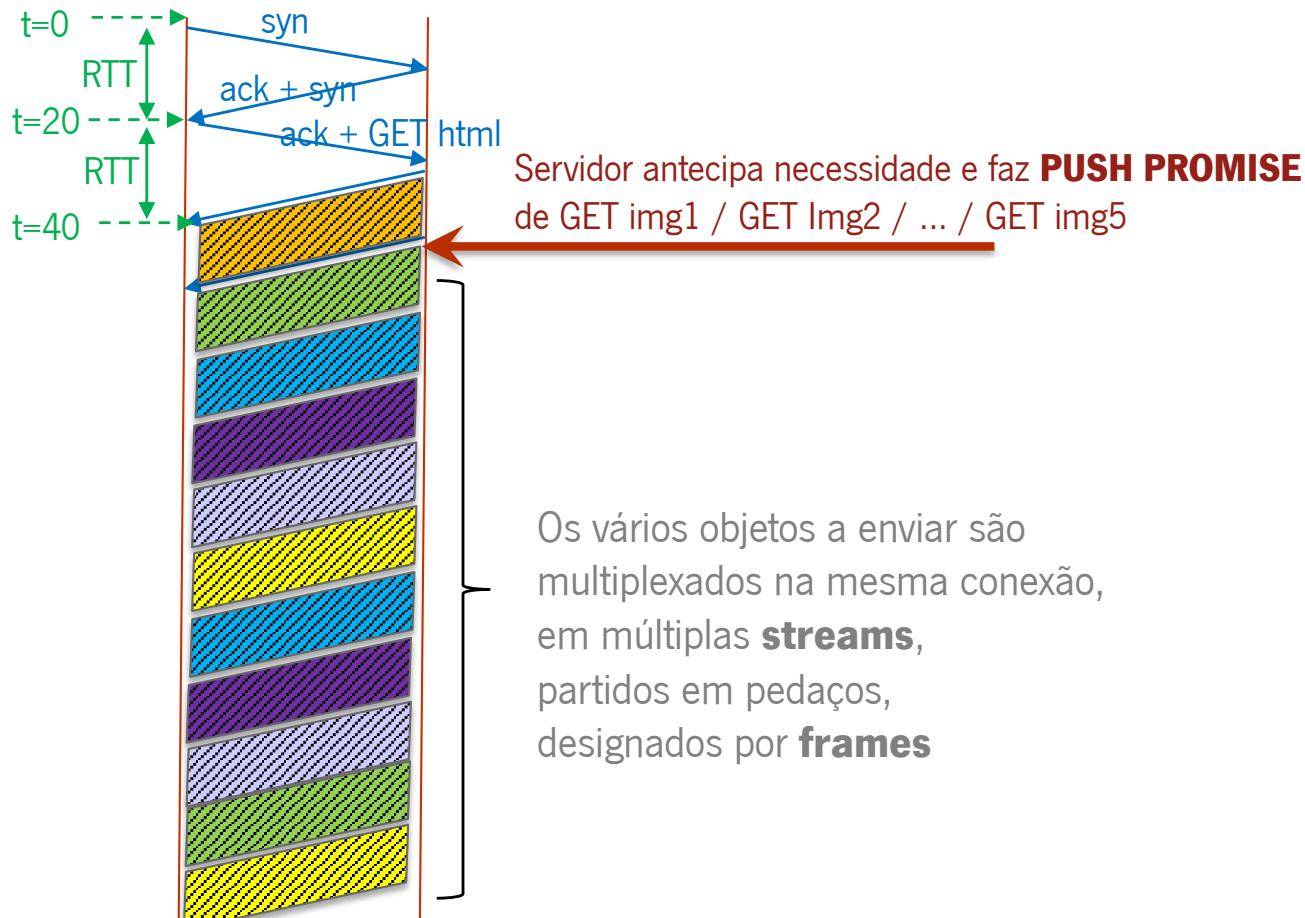


T.c.total = 468 ms (ou 488 ms contando com fecho conexão)

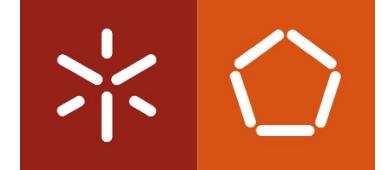


Relembrar o exercício HTTP

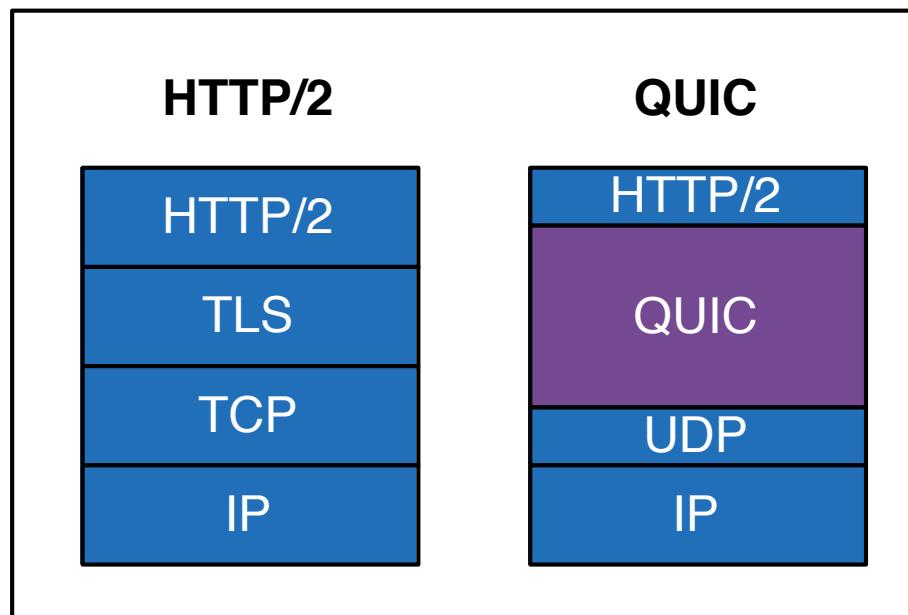
d) HTTP2 múltiplas streams numa mesma conexão com Server Push



QUIC



- Controlo de congestão, cifragem e parte do HTTP2 muda-se para o QUIC que vai correr sobre UDP...
- Nasce o HTTP/3

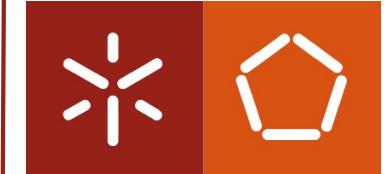


HTTP

Aplicações Web: Suporte HTTP para Web Services

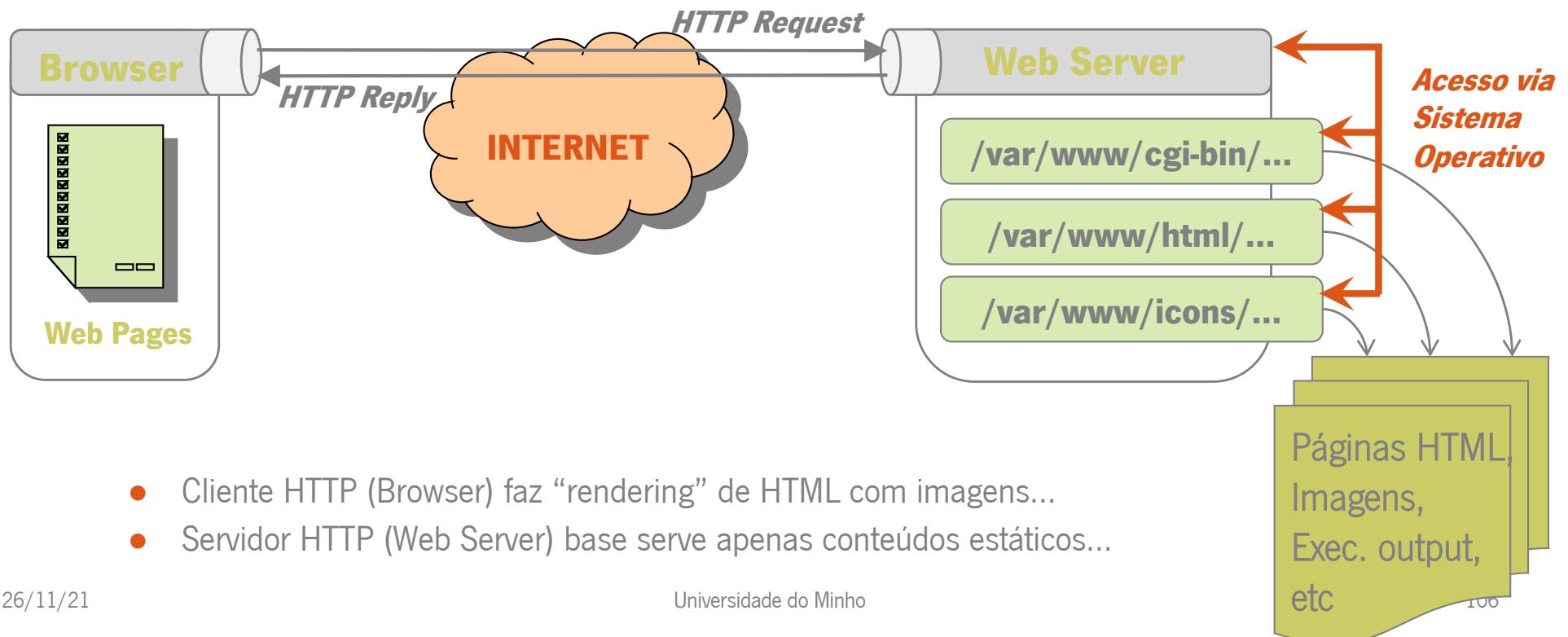
Conceitos práticos

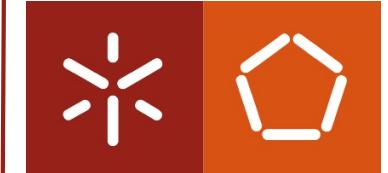




Aplicações Web: conceitos

- Protocolo HTTP - *Hypertext Transfer Protocol*
 - É *stateless* por concepção!... Só com *Cookies* se ultrapassa isso!...
 - Passa por *firewalls* ou por servidores intermediários *Proxy/Cache*...



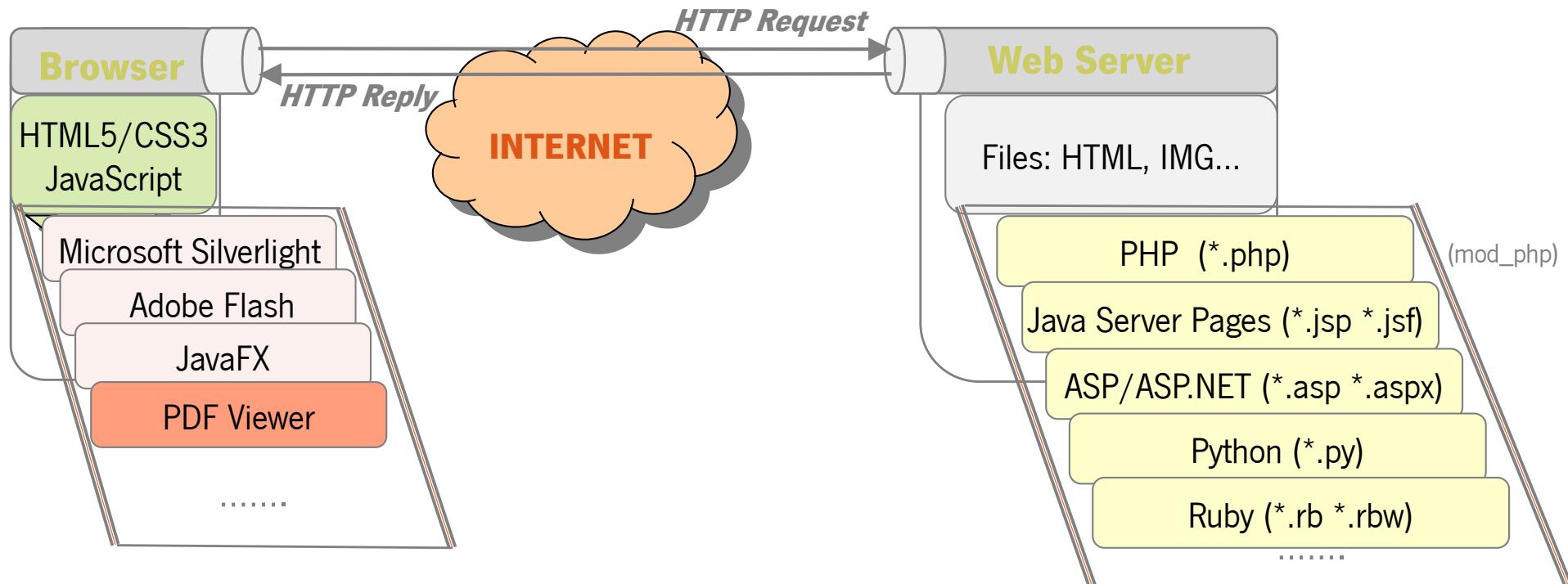


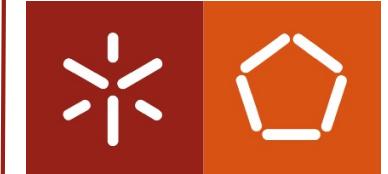
Aplicações Web: conceitos

• Web Applications

Como construir aplicações mais interactivas (tipo Desktop) sobre este modelo?

- Extensões do lado do cliente: External Viewers, Plug-Ins, Client Side Scripting
- Extensões do lado do servidor: Server Side Includes, Server Side Scripting



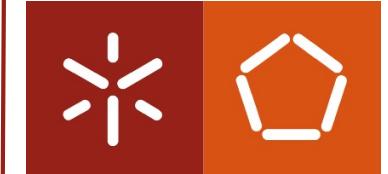


Aplicações Web: REST API Design

- Lista de operações sobre um **RECURSO** (ex: livros) é definida aproveitando a semântica dos métodos do protocolo HTTP:

Recurso	POST (Create)	GET (Read)	PUT (Update)	DELETE (Delete)
/livros	Cria um novo livro; Pedido: objeto “livro” no corpo do HTTP Request!	Lista todos os livros; Pedido: vazio; Resposta: listagem de livros;	Atualiza um conjunto de livros passados no corpo do pedido HTTP	Apaga todos os livros; Pedido: vazio; Resposta: sucesso ou insucesso;
/livros/01	Normalmente não é usado! Erro!	Devolve o objeto que representa o livro com id 01	Se existe livro 01 então atualiza-o; Senão dá erro!	Se existe livro 01 apaga-o;

CRUD (Create / Read / Update / Delete)



Ferramentas úteis

\$ curl ...

(command line)



\$ http ...

(command line)

POSTMAN

Google Chrome Plugin



WireShark

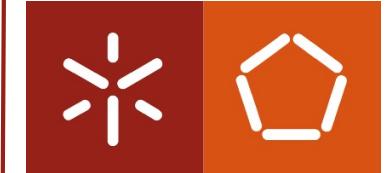
Packet Sniffer (just in case)

<https://curl.haxx.se>

<https://httpie.org>

<https://www.getpostman.com/docs/introduction>

<https://www.wireshark.org>



Teste

- Teste com “Developer Tools” do browser

The screenshot shows a web browser window displaying a page titled "Comunicações por Computador (MIEI)" with the subtitle "Ano Lectivo 2016 / 2017". Below the title, it says "Grupo de Comunicações - Dep. de Informática - Escola de Engenharia - Universidade do Minho". The browser's address bar shows the URL "marco.uminho.pt/disciplinas/CC-MIEI/". The developer tools Network tab is selected, showing a timeline of network requests. A large orange arrow points to the "Network" tab in the top toolbar of the developer tools interface.

Marco UMinho - Disciplinas - CC-MIEI - Universidade do Minho

marco.uminho.pt/disciplinas/CC-MIEI/

Comunicações por Computador (MIEI)

Ano Lectivo 2016 / 2017

Grupo de Comunicações - Dep. de Informática - Escola de Engenharia - Universidade do Minho

Network Timeline Profiles Application Security Audits AngularJS

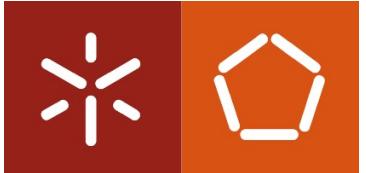
Preserve log Disable cache Offline No throttling

Filter Regex Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other

10000ms 20000ms 30000ms 40000ms 50000ms 60000ms 70000ms 80000ms 90000ms 100000ms 110000ms 120000ms 130000ms

Name	Headers	Preview	Response	Timing
CC-MIEI/ /disciplinas	General	Request URL: http://marco.uminho.pt/disciplinas/CC-MIEI/ Request Method: GET Status Code: 200 OK (from disk cache) Remote Address: 193.136.9.240:80		
costa.css /~costa	Response Headers	Accept-Ranges: bytes Content-Length: 8018 Content-Type: text/html		
valid-html401 www.w3.org/Icons				
created-with-vim.png /disciplinas/CC-MIEI				

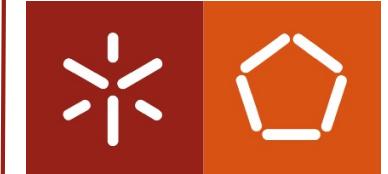
Teste



- Teste com o Plug In **POSTMAN** do Google Chrome!

Resultado!

```
1 {  
2   "args": {  
3     "test": "123"  
4   },  
5   "headers": {  
6     "host": "echo.getpostman.com",  
7     "accept": "application/json",  
8     "accept-encoding": "gzip, deflate, sdch, br",  
9     "accept-language": "en-US,en;q=0.8,pt;q=0.6",  
10    "cache-control": "no-cache",  
11    "postman-token": "350a46cf-34f4-21ea-7b9b-5a4ca88e39ea",  
12    "user-agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87  
Safari/537.36",  
13    "x-forwarded-port": "443",  
14    "x-forwarded-proto": "https"  
15  },  
16  "url": "https://echo.getpostman.com/get?test=123"  
17 }
```

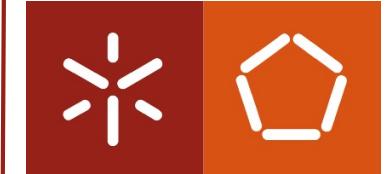


REST API: GET (read all)

```
$ http GET http://ec2-54-175-51-193.compute-1.amazonaws.com/PHPWebServices/biblioteca.php/livros -v
GET /PHPWebServices/biblioteca.php/livros HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: ec2-54-175-51-193.compute-1.amazonaws.com
User-Agent: HTTPie/0.9.4
```

```
HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Length: 53
Content-Type: application/json
Date: Mon, 28 Nov 2016 13:34:09 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.23 (Amazon) OpenSSL/1.0.1k-fips PHP/5.6.28
X-Powered-By: PHP/5.6.28
```

```
[
  {
    "autor": "Camoës",
    "id": "01",
    "titulo": "Os Lusiadas"
  }
]
```

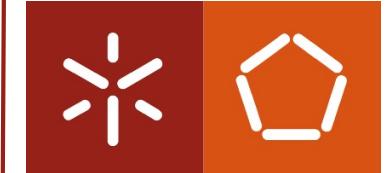


API REST: GET (read one)

```
$ http GET http://ec2-54-175-51-193.compute-1.amazonaws.com/PHPWebServices/biblioteca.php/livros/01 -v  
GET /PHPWebServices/biblioteca.php/livros/01 HTTP/1.1  
Accept: */*  
Accept-Encoding: gzip, deflate  
Connection: keep-alive  
Host: ec2-54-175-51-193.compute-1.amazonaws.com  
User-Agent: HTTPie/0.9.4
```

```
HTTP/1.1 200 OK  
Connection: Keep-Alive  
Content-Length: 51  
Content-Type: application/json  
Date: Mon, 28 Nov 2016 13:39:20 GMT  
Keep-Alive: timeout=5, max=100  
Server: Apache/2.4.23 (Amazon) OpenSSL/1.0.2k-fips PHP/5.6.28  
X-Powered-By: PHP/5.6.28
```

```
{  
    "autor": "Camoës",  
    "id": "01",  
    "titulo": "Os Lusiadas"  
}
```

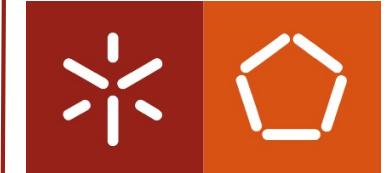


API REST: POST (create new)

```
$ http POST http://ec2-54-175-51-193.compute-1.amazonaws.com/PHPWebServices/biblioteca.php/livros id=02
autor="Eça de Queirós" titulo="Os Maias" -v
POST /PHPWebServices/biblioteca.php/livros HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 71
Content-Type: application/json
Host: ec2-54-175-51-193.compute-1.amazonaws.com
User-Agent: HTTPie/0.9.4

{
    "autor": "Eça de Queirós",
    "id": "02",
    "titulo": "Os Maias"
}
```

```
HTTP/1.1 201 Created
Connection: Keep-Alive
Content-Length: 0
Content-Type: text/html; charset=UTF-8
Date: Mon, 28 Nov 2016 13:42:23 GMT
Keep-Alive: timeout=5, max=100
Location:
Server: Apache/2.4.23 (Amazon) OpenSSL/1.0.1k-fips PHP/5.6.28
X-Powered-By: PHP/5.6.28
```



API REST: PUT (modify one)

```
$ http PUT http://ec2-54-175-51-193.compute-1.amazonaws.com/PHPWebServices/biblioteca.php/livros/02  
id=02 autor="Eca de Queiros" titulo="Os Maias" -v  
PUT /PHPWebServices/biblioteca.php/livros/02 HTTP/1.1  
Accept: application/json  
Accept-Encoding: gzip, deflate  
Connection: keep-alive  
Content-Length: 61  
Content-Type: application/json  
Host: ec2-54-175-51-193.compute-1.amazonaws.com  
User-Agent: HTTPie/0.9.4  
  
{  
    "autor": "Eca de Queiros",  
    "id": "02",  
    "titulo": "Os Maias"  
}
```

HTTP/1.0 500 Internal Server Error

Connection: close
Content-Length: 0
Content-Type: text/html; charset=UTF-8
Date: Mon, 28 Nov 2016 13:50:27 GMT
Server: Apache/2.4.23 (Amazon) OpenSSL/1.0.1k-fips PHP/5.6.28
X-Powered-By: PHP/5.6.28