

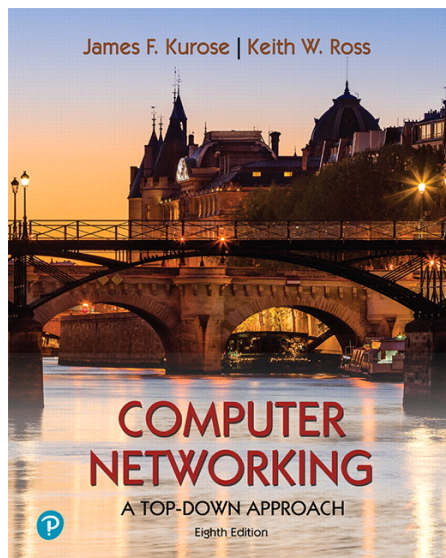
# HTTP

## Comunicações por Computador

Mestrado Integrado em Engenharia Informática

3º ano/2º semestre

2024/2025

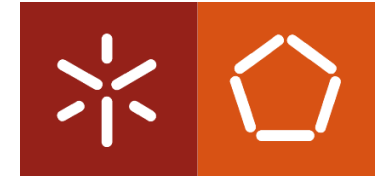


***Computer Networking: A Top Down Approach, Capítulo 2***  
**Jim Kurose, Keith Ross, Addison-Wesley ©2021 .**



# HTTP

## Hypertext Transfer Protocol



Conceitos básicos, bem conhecidos:

- Uma página *web* consiste numa coleção de objetos incluída num ficheiro base HTML que pode incluir várias referências a outros objetos/páginas *web*
- Um objeto pode ser um outro ficheiro HTML, uma imagem JPEG, um *applet* Java, um ficheiro áudio, etc.,
- Cada objeto é endereçado/referido por um *Uniform Resource Locator* (URL).

Exemplo de URL:

`http://www.di.uminho.pt/cursos/lei.html`

**host name**

**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)**

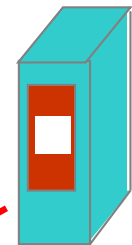
**HTTP/2: RFC 7540 (maio 2015)**

**HTTP/3: RFC 9114 (junho 2022)**

PC a executar o  
Firefox



Pedido HTTP  
Resposta HTTP



Servidor a  
executar  
o servidor  
WEB Apache

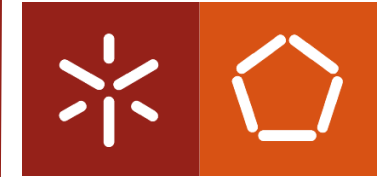


Pedido HTTP  
Resposta HTTP

Mac a executar o  
Safari

# HTTP

## *Como funciona?*



### Utiliza o TCP:

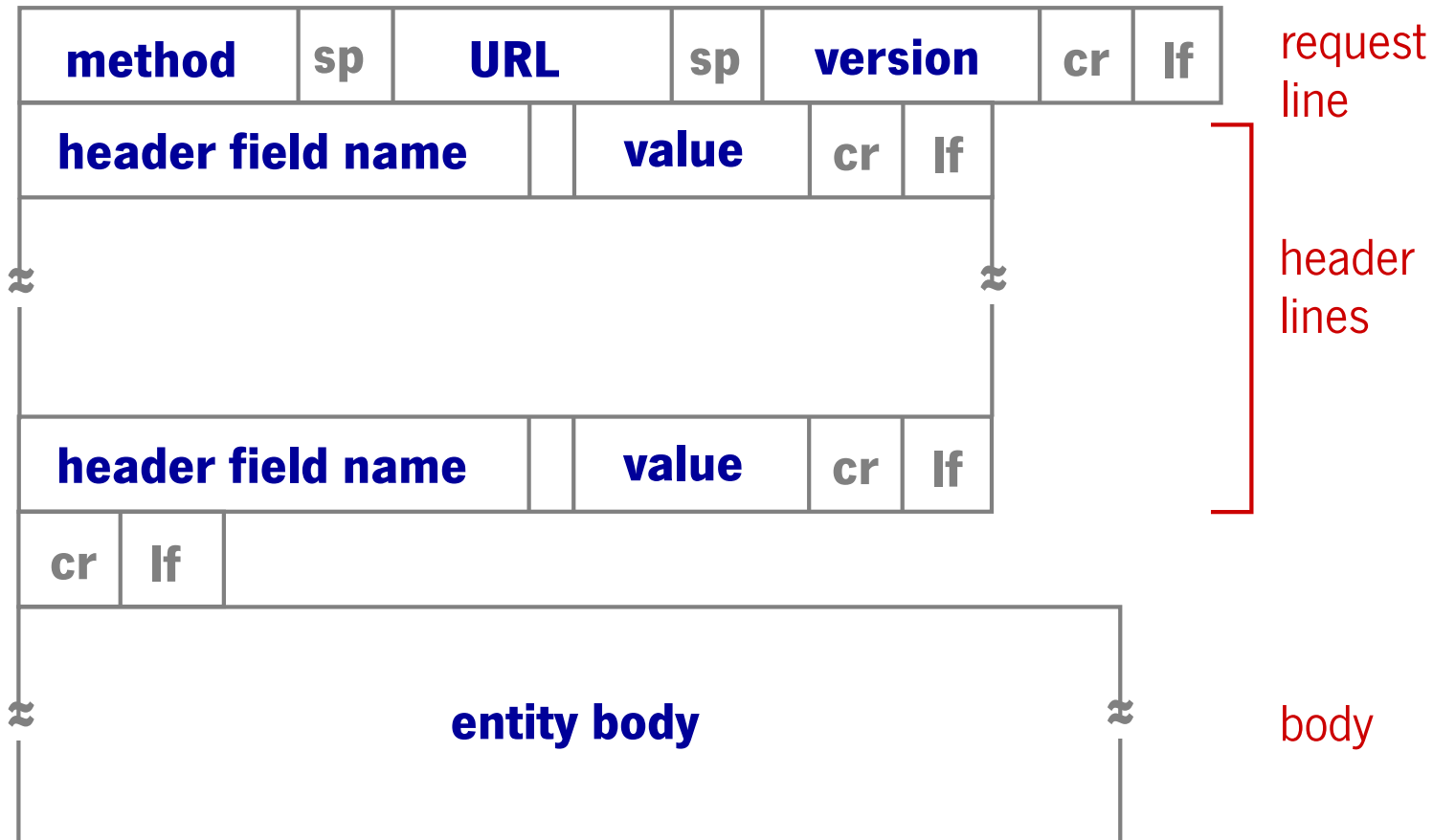
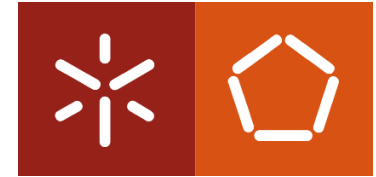
- O cliente cria um *socket* e inicia uma conexão TCP com um servidor HTTP (por defeito, à escuta na porta 80);
- O servidor TCP aceita o pedido de conexão do cliente;
- São trocadas mensagens HTTP (mensagens de protocolo de nível aplicacional) entre o *browser* (cliente HTTP) e o servidor web/HTTP;
- 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 pois os estados passados têm que ser armazenados. Se o servidor/cliente falha a sua visão do estado pode ficar inconsistente e terá que ser sincronizada.

# HTTP

## *Formato das mensagens: pedido HTTP*



# HTTP

## *Exemplo da sintaxe duma mensagem...*

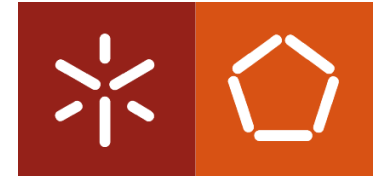


### ***HTTP Request Message***

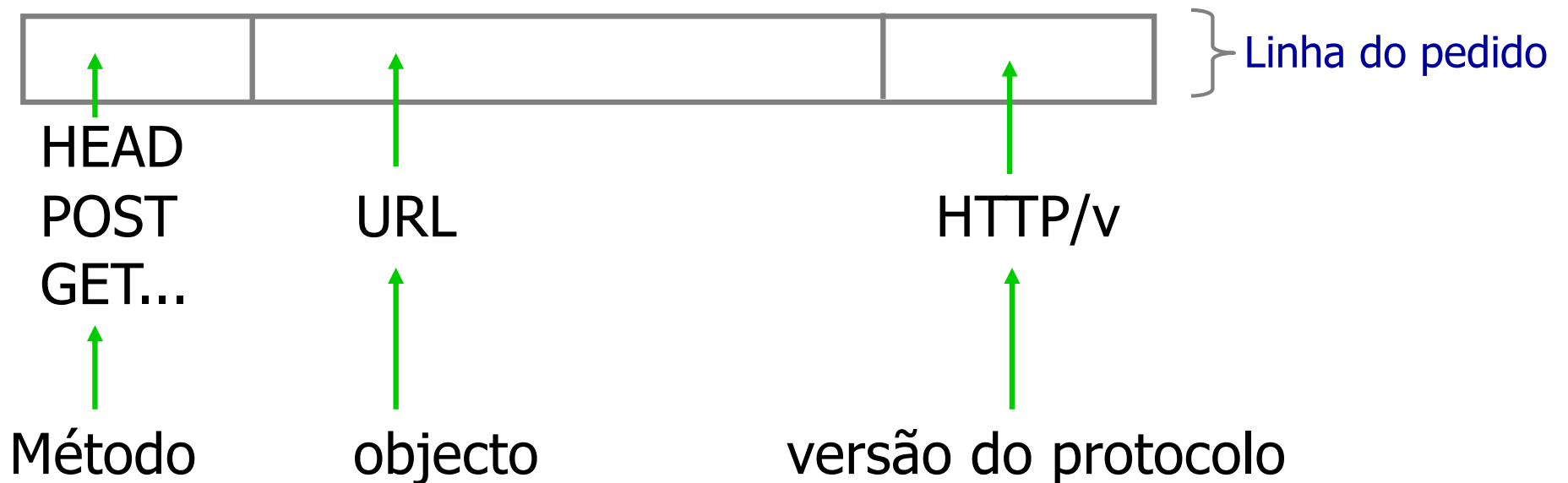
GET	/directoria/pagina.html	HTTP/1.1	}	Linha do pedido
Host: www.sitio.pt				
Connection: close			}	Linhas do cabeçalho
User-Agent: Mozilla/4.0				
Accept-Language: PT				
<new line>				
Corpo da mensagem ( <i>vazio no caso do GET</i> )			}	Dados da mensagem

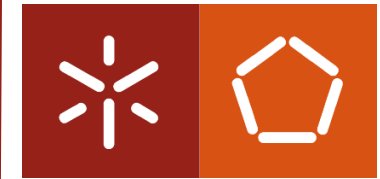
# HTTP

## *Exemplo da sintaxe duma mensagem...*



### HTTP Request Message





### 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.

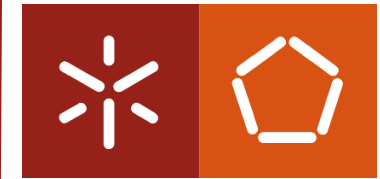
### Método URL:

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

`www.somesite.com/animalsearch?monkeys=1&banana=4`

HOSTNAME                      PATH                      QUERY\_STRING





### HTTP/1.0

- **GET**
- **POST**
- **HEAD**

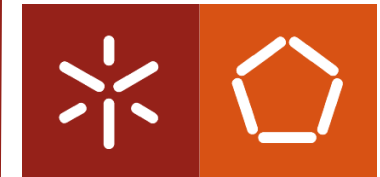
(pede ao servidor para não incluir o objeto requerido na resposta, apenas o cabeçalho do objeto)

### HTTP/1.1

- **GET, POST, HEAD**
- **PUT**

(faz o *upload* do objeto para a localização especificada no campo URL)

- **DELETE**  
(apaga o ficheiro especificado no URL)



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)**

# HTTP

## *Formato das mensagens: resposta HTTP*



### HTTP Response Message

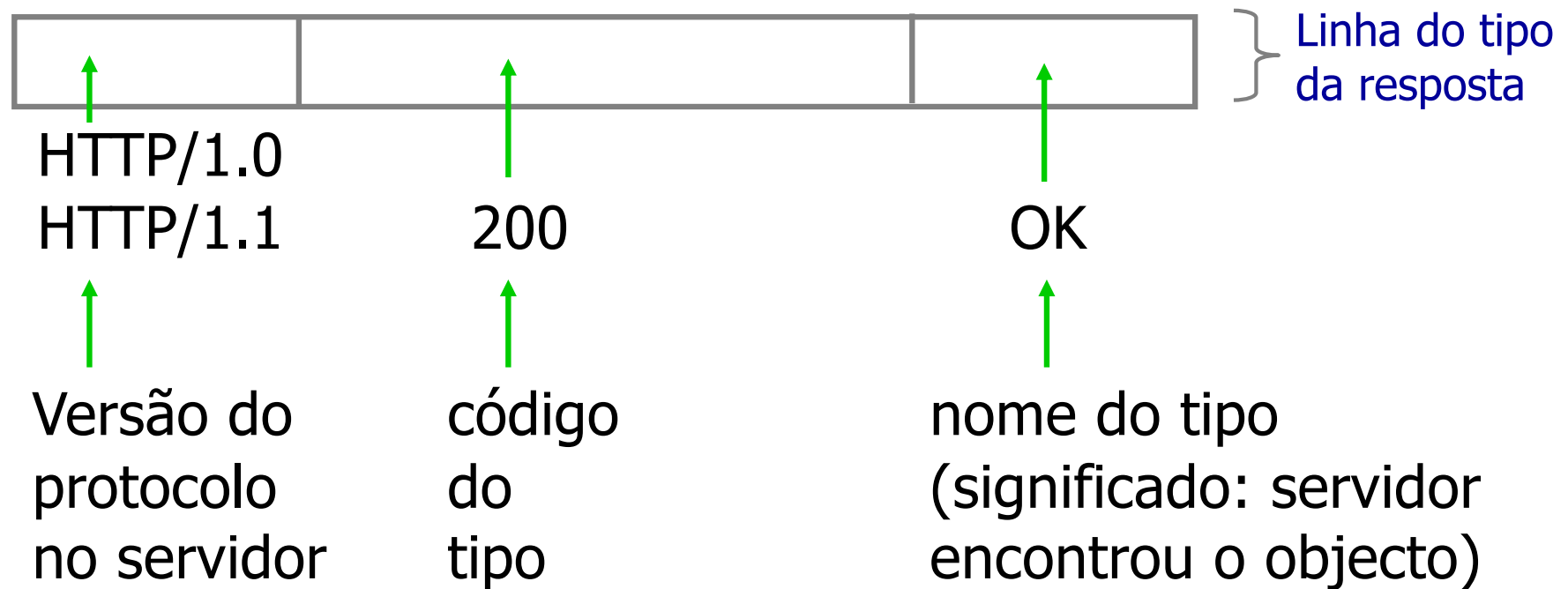
HTTP/1.1	200	OK	}	Linha do tipo da resposta
Connection: close				
Date: 07 Mai 2003 11:35:15 UTC+1			}	Linhas do cabeçalho
Server: Apache/1.3.0 (Unix)				
Last-Modified: 05 Mai 2003 09:23:45 UTC+1				
Content-Length: 6825				
Content-Type: text/html				
<new line>			}	Dados da mensagem
Corpo da mensagem (objecto)				

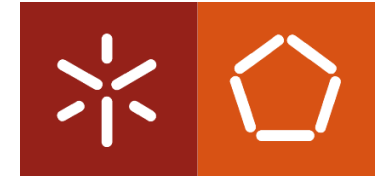
# HTTP

## *Formato das mensagens: resposta HTTP*



### HTTP Response Message





### **Alguns códigos de tipo e seu significado**

200 OK

301 Moved permanently, location: xyz

304 Not modified

400 Bad request (pedido não entendido)

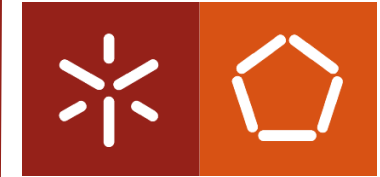
401 Authorization required

404 Not found (objecto não encontrado)

505 HTTP version not supported

# HTTP

## *Aplicação cliente de linha (browser)*



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

```
GET / HTTP/1.1
```

```
Accept: */*
```

```
Accept-Encoding: gzip, deflate
```

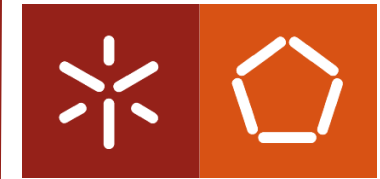
```
Connection: keep-alive
```

```
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/> )

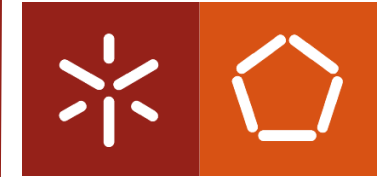


### **HTTP não persistente:**

- Só pode ser enviado no máximo um objeto *web* por cada conexão estabelecida.
- O HTTP/1.0 utiliza HTTP não persistente.
- Mínimo de 2 RTT/objeto.
- Exige mais recursos do S.O.
- Alguns browsers abrem várias conexões TCP em paralelo para pedirem vários objetos referidos no mesmo objeto.

### **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.
- Servidor mantém conexão TCP aberta.
- Com ou sem estratégia de *pipelining*.



### **Sem pipelining:**

- O cliente envia um novo pedido apenas quando recebe a resposta ao anterior.
- No cenário mais otimista consome-se um RTT por cada objeto referido.

### **Com pipelining:**

- Modo por defeito no HTTP/1.1.
- O cliente envia os pedidos assim que os encontra no objeto referenciador.
- No cenário mais otimista é consumido um RTT para o conjunto de todos os objetos referenciados.



# HTTP

## *Exemplo não persistente...*



URL: **www.uminho.pt/DI/index.html**

(contém texto e referências para imagens)

1a. O cliente HTTP inicia uma conexão TCP com o servidor que está a ser executado no sistema **www.uminho.pt** e está à escuta na porta 80

1b. O servidor HTTP aceita o pedido de conexão e avisa o cliente.

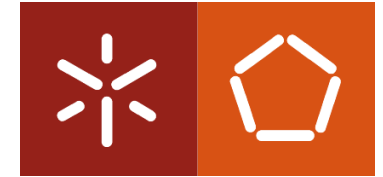
2. O cliente HTTP envia uma mensagem do tipo *request message* (contendo a URL) através de um novo *socket* TCP. A mensagem indica que o cliente deseja o objeto web **DI/index.html**.

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**

# HTTP

## *Exemplo não persistente...*



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

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 referência a vários objetos que são imagens. Fecha a conexão TCP.

[...] Repete os passos 1-5 para cada objeto referenciado.

**tempo**

# HTTP

## *Tempo de Resposta*



$$RTT = 2 * TP + N * TEQ + N * PR$$

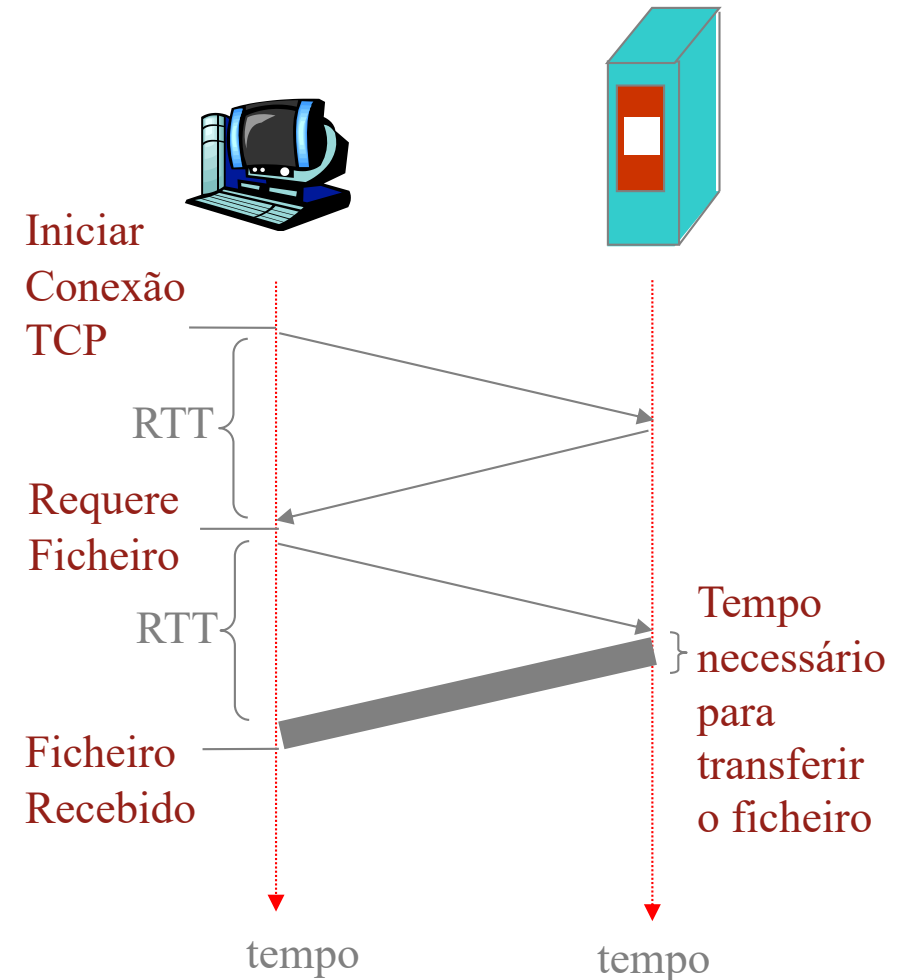
*TP*: Tempo de Propagação

*TEQ*: Tempo de Espera nas filas de todos os sistemas (origem, destino e intermédios)

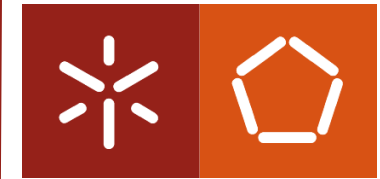
*PR*: Tempo de Processamento em todos os sistemas (origem, destino e intermédios)

$$\text{Tempo de Resposta} = 2 * RTT + TT$$

- um *RTT* para iniciar uma conexão TCP
- um *RTT* para enviar a *request message* e começar a receber o primeiro bit do ficheiro na *response message*
- e o *TT*, que é o tempo de transmissão do ficheiro

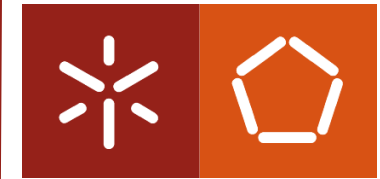


# Exercício

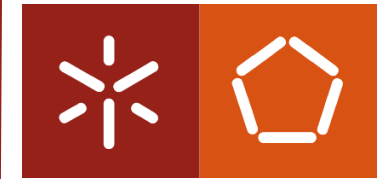


- Pretende-se estimar o atraso na receção de um documento Web usando o protocolo HTTP. Sabemos que o atraso de ida-e-volta entre cliente e servidor é **4 ms**, que o débito do caminho que une o cliente ao servidor é **1024 Kbps** e que cada segmento TCP contém no máximo **128 bytes** de dados. Desprezam-se 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 objeto base com 2048 *bytes*, a memória de receção 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 receção 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 receção 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.

# 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 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.
  - a) 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?
  - b) 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?



### 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*

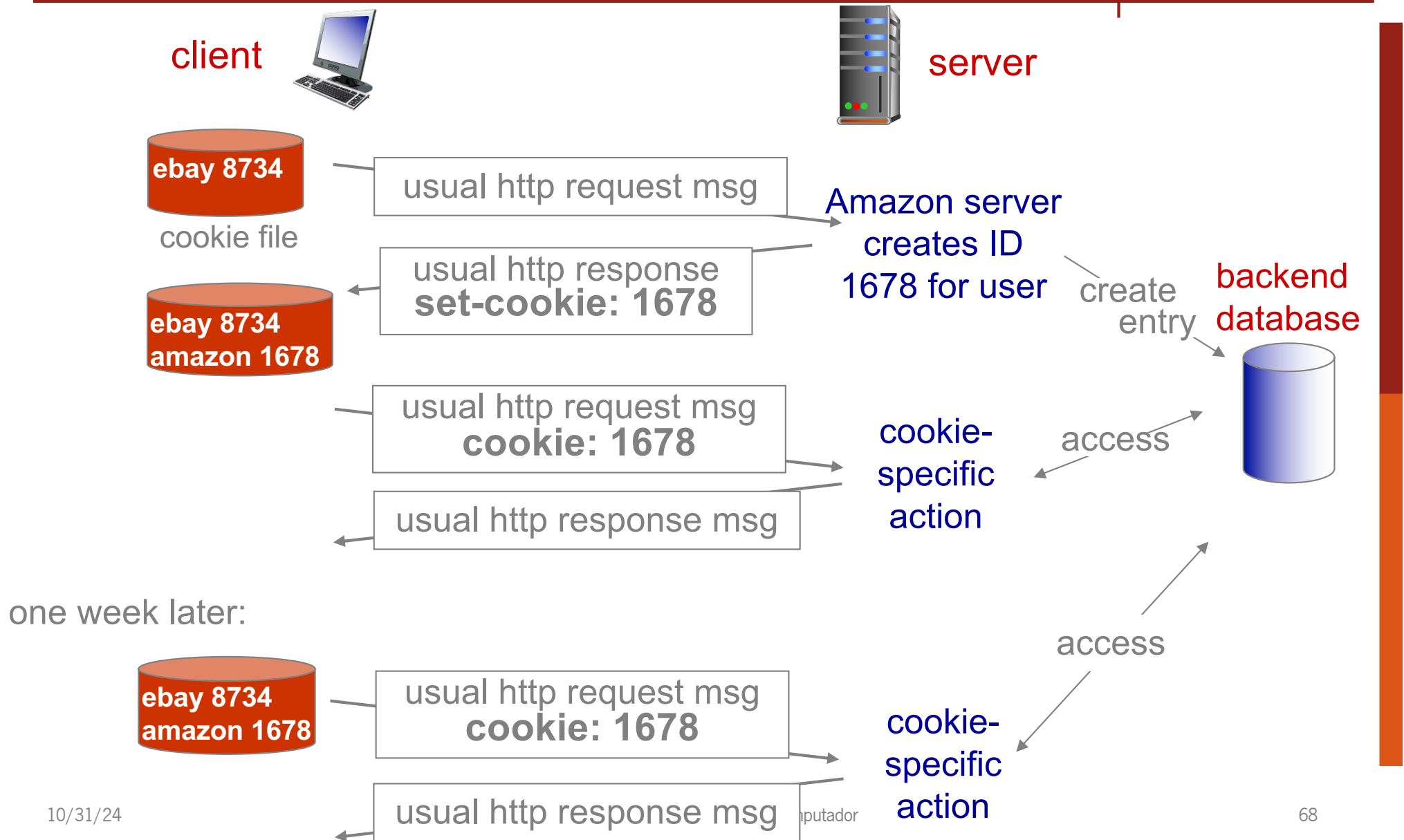
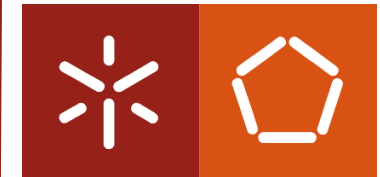
### Exemplo:

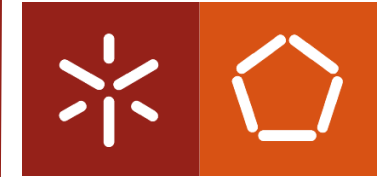
Um utilizador acede sempre à Internet a partir do seu PC e visita um site de comércio eletrónico pela primeira vez. Quando o primeiro pedido chega ao servidor *Web*, este gera:

- Um Identificador único, e
- Uma entrada na base de dados de suporte para esse Identificador.

# HTTP

## Informação de estado – *Cookies*





### efeitos colaterais

#### O que os cookies permitem:

- autorizar
- implementar cabaz de compras
- fazer sugestões ao utilizador
- manter informação da sessão por cada utilizador (ex: *Web e-mail*)
- etc...

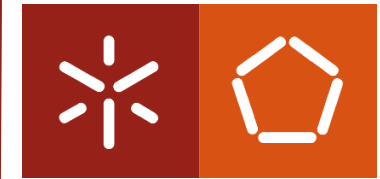
#### Os Cookies e a privacidade:

- os cookies ensinam muito aos servidores a respeito dos utilizadores e seus hábitos
- o utilizador pode estar a fornecer dados ao servidor sem saber...

#### Como manter informação do “estado”:

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



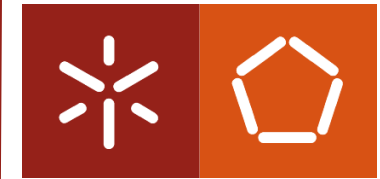


### Porquê?

- 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).
- a Internet está povoada de *caches* e que permitem que fornecedores de conteúdos mais “pobres” disponibilizem efetivamente os seus conteúdos (um pouco como as redes de partilha de ficheiros P2P...)

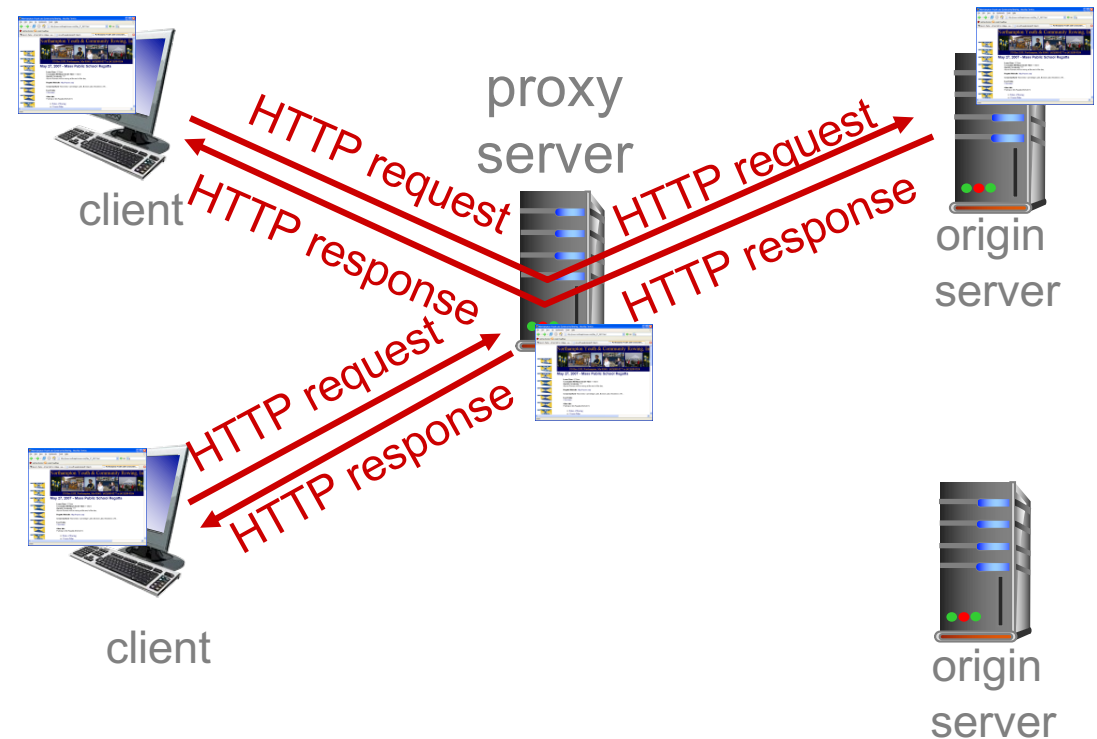
### Como?

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



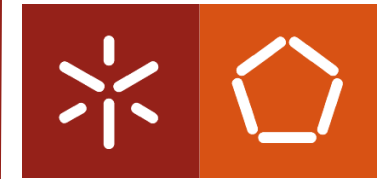
**Objetivo:** satisfazer o pedido do cliente sem envolver o servidor HTTP alvo (que está longe)

- 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 uma cópia do objeto requerido está na *cache* do proxy o servidor proxy retorna essa cópia;
  - Senão, o servidor proxy contacta o servidor HTTP alvo, envia-lhe a *HTTP request message*, aguarda a resposta que guarda em cache e retorna ao cliente.



# HTTP

## Exemplo *Web Proxy - Caching*

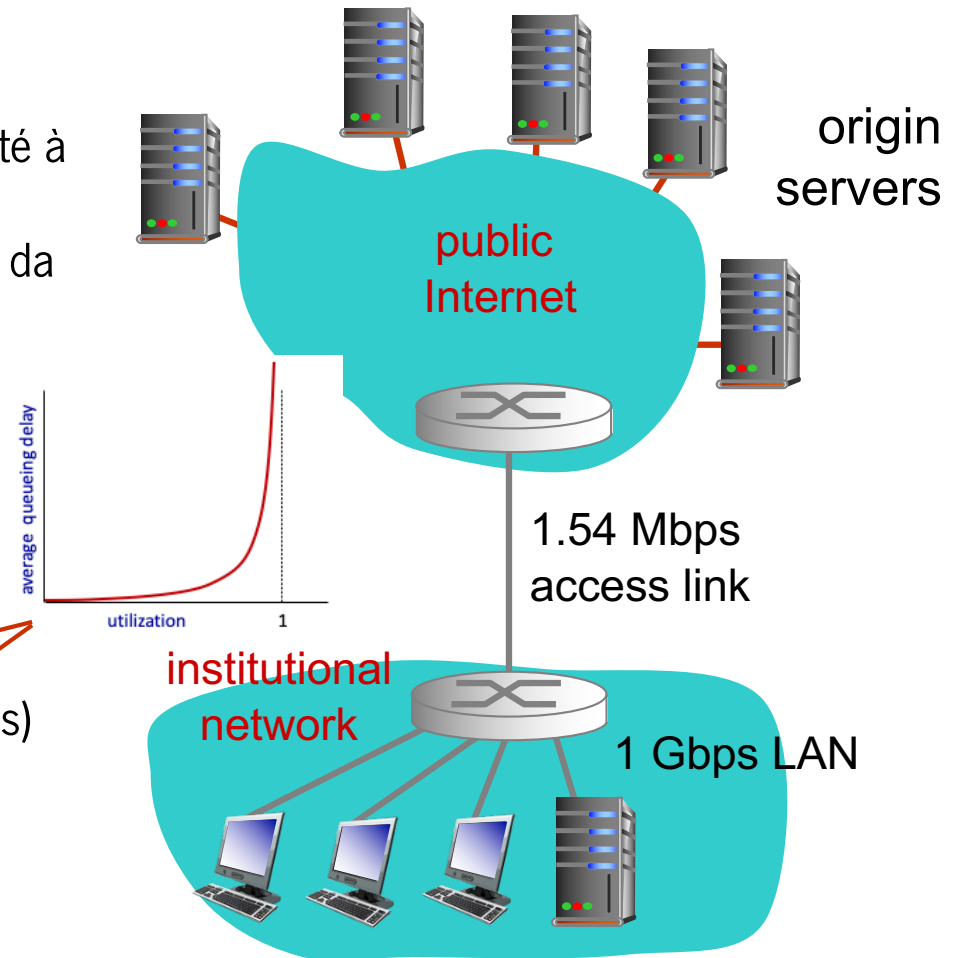


### Pressupostos:

- Tamanho médio dos objetos = 100Kbits
- Tempo médio de atraso desde o pedido HTTP até à chegada da resposta = 2 seg
- Taxa média de pedidos efetuados pelos clientes da instituição para servidores HTTP = 15/seg
  - Taxa de transmissão média = 1.50 Mbps

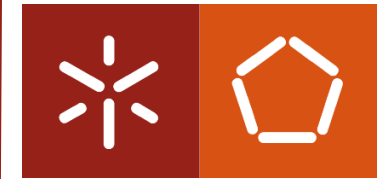
### Consequências:

- Utilização da LAN = **15%**
- Utilização do *Link* de acesso = **97%**
- Total do atraso (*delay*) =  
= atraso Internet + atraso acesso + atraso LAN  
= 2 segundos + **minutos** + milissegundos



# HTTP

## Exemplo *Web Proxy - Caching*

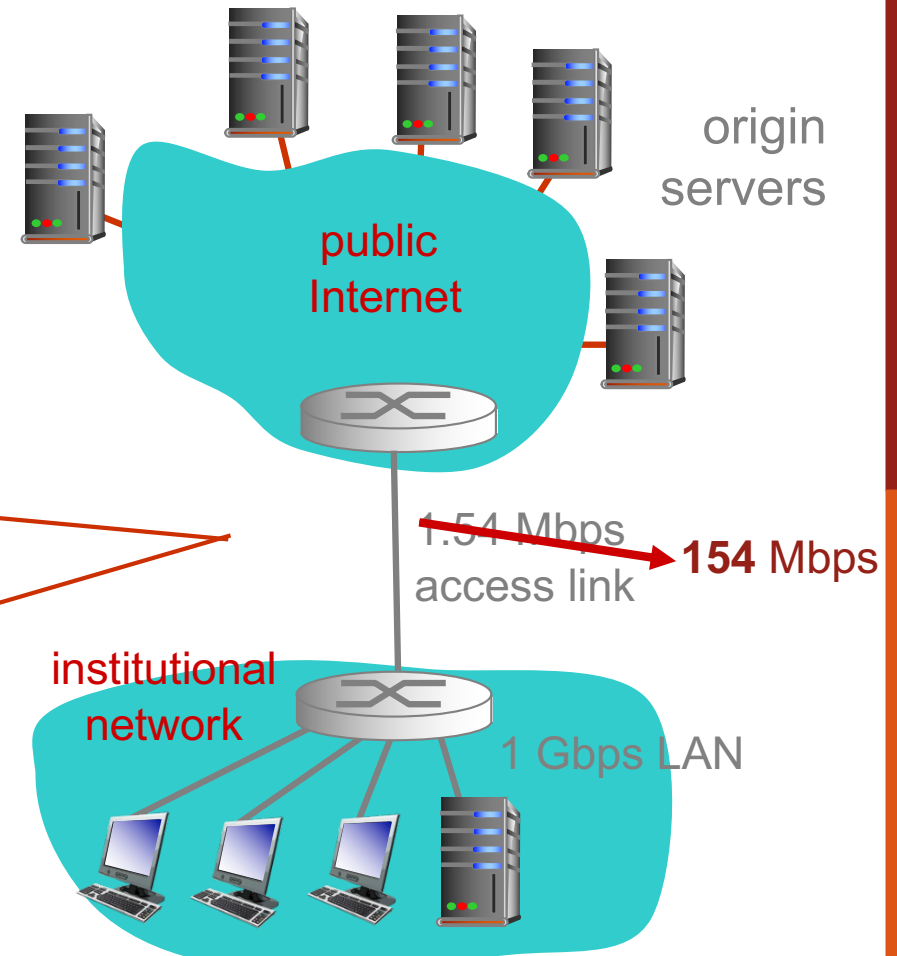


### Solução possível

- Aumentar a largura de banda do link de acesso para **154 Mbps**

### Consequência

- Utilização da LAN = 15%
- Utilização do Link de Acesso = **9,7%**
- Total delay = Internet delay + access delay + LAN delay  
= 2 segundos + **msegundos** + msegundos
- É habitualmente muito dispendioso fazer o upgrade do link de acesso de uma instituição



# HTTP

## Exemplo *Web Proxy - Caching*

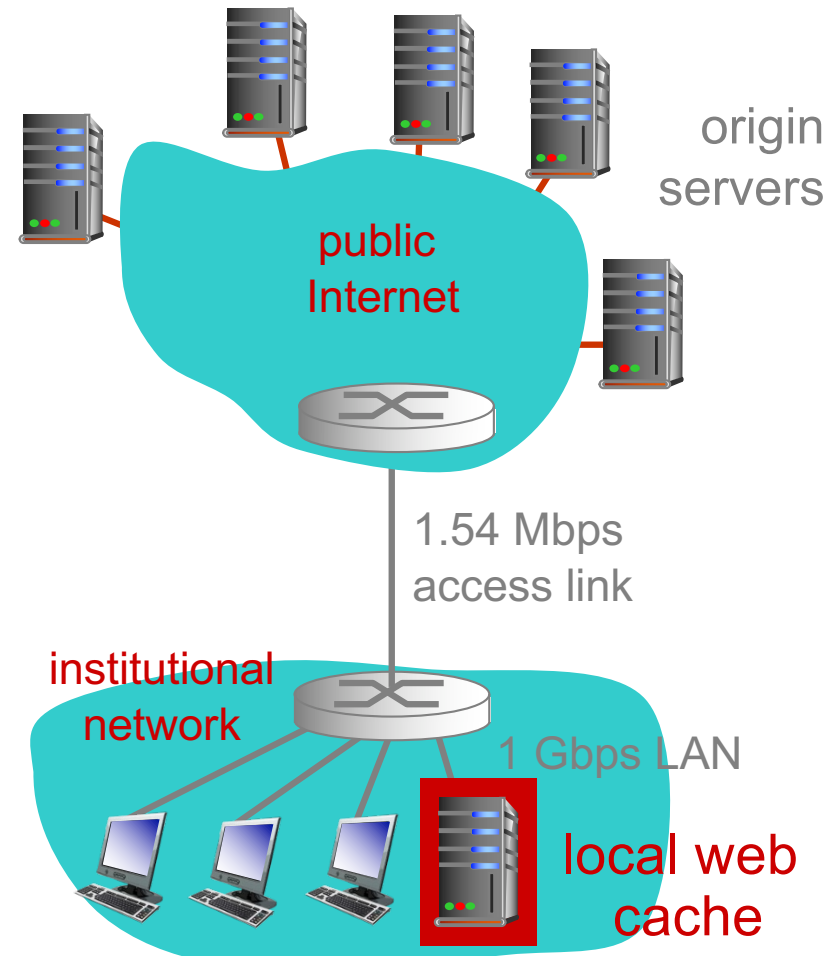


### Outra solução é instalar o Web Proxy

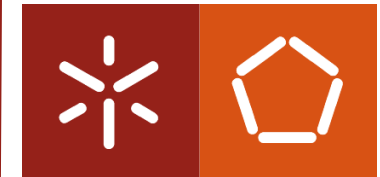
- Se a taxa de acerto for de 40% ...

### Consequências:

- 40% dos pedidos satisfeitos localmente na *cache*
- 60% dos pedidos terão que ser redirecionados para o servidor HTTP respetivo
  - Taxa de transmissão no link de acesso =  
 $= 0.6 * 1.50 \text{ Mbps} = 0.9 \text{ Mbps}$
  - Utilização do link de acesso =  
 $= 0.9 / 1.54 = 0.58$  (**58%**)
  - Resulta em atrasos negligenciáveis (milisegundos)
- **total avg delay** =  
= Internet delay + access delay + LAN delay  
=  $.6 * (2.01) \text{ seg} + 0.4 * 10 \text{ mseg} < 1.2 \text{ segundos}$



# GET Condicional



- **Objetivo:** não enviar o objeto se a cópia mantida em cache está atualizada
- Cabeçalho do pedido HTTP inclui data da cópia guardada na cache:  
**If-modified-since: <date>**
- A resposta do servidor não contém nenhum objeto se a cópia mantida em cache estiver atualizada  
**HTTP/1.0 304 Not Modified**

client



server



HTTP request msg  
**If-modified-since: <date>**

object  
not  
modified  
before  
<date>

HTTP response  
**HTTP/1.0  
304 Not Modified**

-----

HTTP request msg  
**If-modified-since: <date>**

object  
modified  
after  
<date>

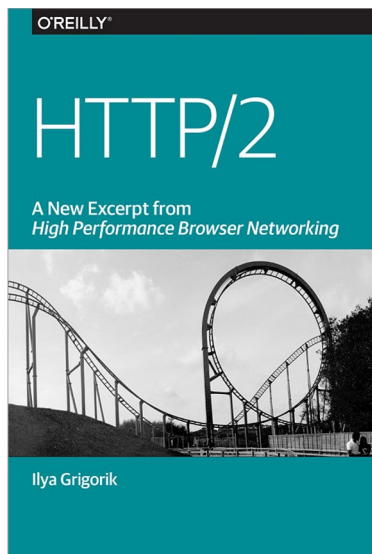
HTTP response  
**HTTP/1.0 200 OK  
<data>**

# HTTP2, HTTP3 +QUIC

**Comunicações por Computador**

Mestrado Integrado em Engenharia Informática

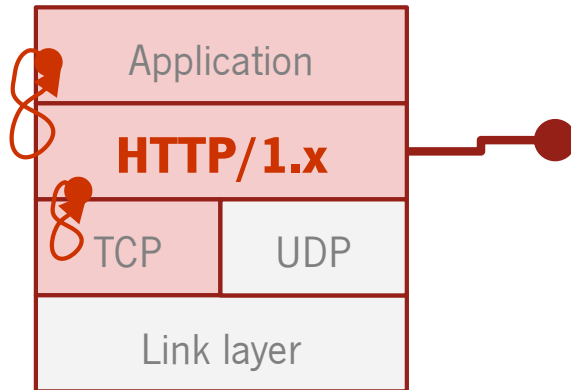
3º ano/2º semestre



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



# 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

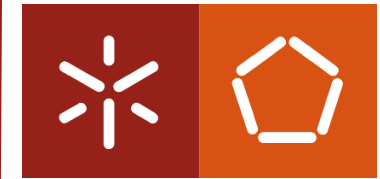
- Bloqueio do cabeça de fila, acumula pedidos em queue e atrasa a solicitação por parte do cliente
- Servidor obrigado a responder pela ordem (ordem restrita)

## Overhead protocolar é elevado

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



# Desempenho HTTP/1.\*















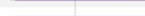

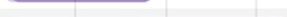












- **Como melhorar o desempenho do HTTP/1.\*?**
- **Quais as melhores práticas, simples e eficazes, que têm sido usadas com regularidade?**
  - Reduzir o número de consultas ao DNS (*DNS Lookups*)
  - Reutilizar conexões TCP
  - Utilizar CDNs (*Content Delivery Network*)
  - Minimizar o número de redireccionamentos 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.\*



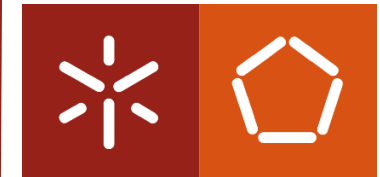
- Paralelismo é limitado pelo número de conexões...

×	Elements	Resources	Network	Sources	Timeline	Profiles	Audits	Console	PageSpeed		
Name	Method	Status	Type	...	...	Time	Start Time	302 ms	453 ms	604 ms	755 m
 localhost	GET	200	text/html	...	...	17 ms					
 01.jpeg	GET	202	image/jpeg	...	...	242 ms					
 02.jpeg	GET	202	image/jpeg	...	...	243 ms					
 03.jpeg	GET	202	image/jpeg	...	...	242 ms					
 04.jpeg	GET	202	image/jpeg	...	...	241 ms					
 05.jpeg	GET	202	image/jpeg	...	...	235 ms					
 06.jpeg	GET	202	image/jpeg	...	...	235 ms					
 07.jpeg	GET	202	image/jpeg	...	...	475 ms					
 08.jpeg	GET	202	image/jpeg	...	...	563 ms					
 09.jpeg	GET	202	image/jpeg	...	...	561 ms					
 10.jpeg	GET	202	image/jpeg	...	...	561 ms					
 11.jpeg	GET	202	image/jpeg	...	...	561 ms					
 12.jpeg	GET	202	image/jpeg	...	...	561 ms					

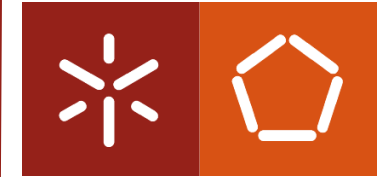
~6 parallel downloads per origin

- 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 paralelismos — 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



- Em meados de **2009**, a **Google** inicia o seu projeto **SPDY!**
  - Objetivo nº1: 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
- Clientes: **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**

# HTTP/2



- 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

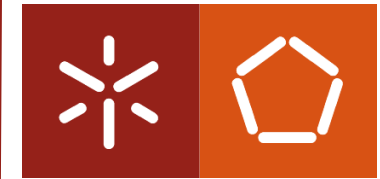
Mar, 2012: Call for proposals for HTTP/2 – resulted in 3 proposals but SPDY was chosen as the basis for H/2

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

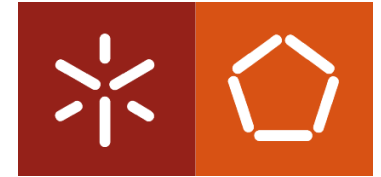
Aug, 2014: HTTP/2 draft-17 and HPACK draft-12 are published

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

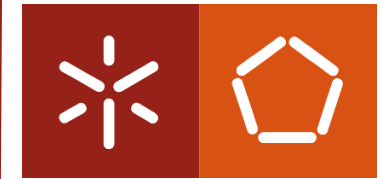
10/Feb/2015: (IESG) Internet Engineering Steering Group approved HTTP/2



- **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 ser 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!...



- **Objetivo: diminuir o atraso em pedidos HTTP de múltiplos objectos!**
- HTTP1.1: permite vários GETs em pipeline numa única ligação TCP
  - o servidor responde aos pedidos GET por ordem FCFS (*First-come, First-Served*) agendamento por ordem de chegada
  - Os objetos pequenos podem ter de esperar pela transmissão atrás dos objetos grandes! **Head-Of-line Lock (HOL)**
  - Recuperação de perda (retransmissão de segmentos TCP perdidos) paralisa a transmissão do objeto! (conexão única)



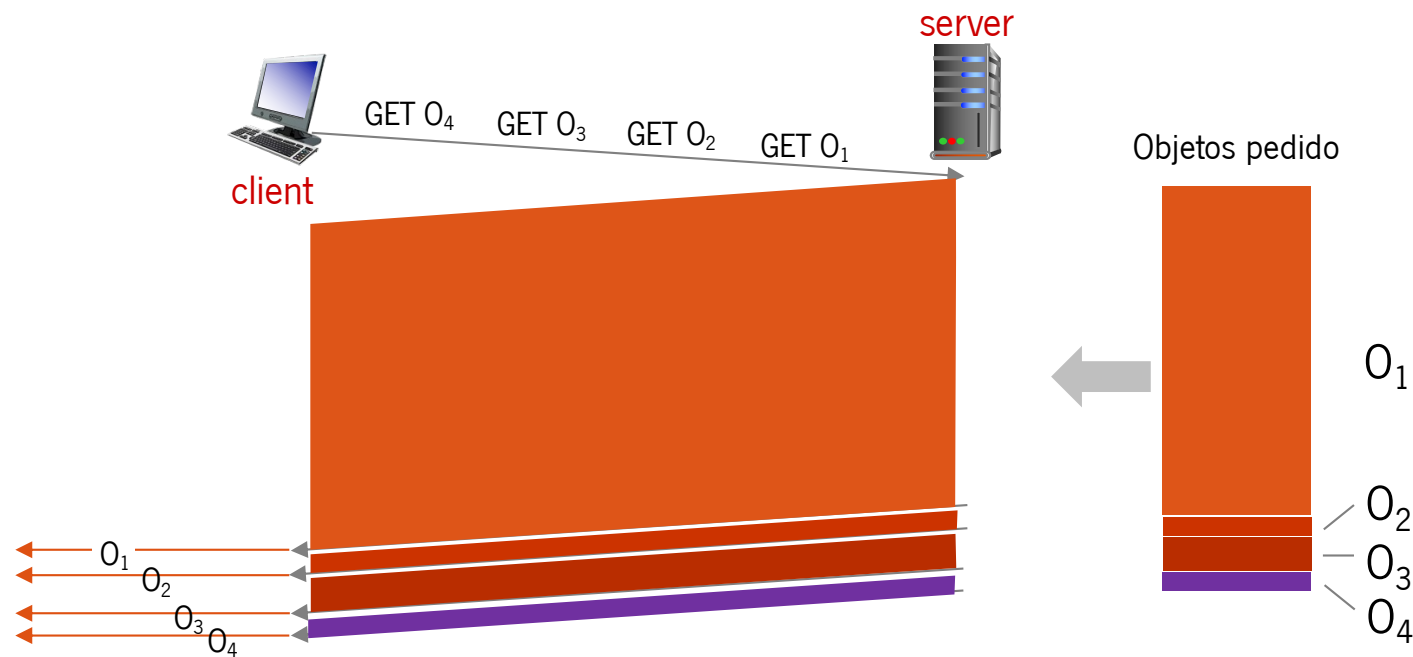
- **Objetivo: diminuir o atraso em pedidos HTTP de múltiplos objectos!**
- HTTP/2: melhor flexibilidade na forma como o servidor envia os objetos aos clientes
  - **Prioridades**: Ordem de transmissão dos objetos solicitados com base na **prioridade** do objeto especificada pelo cliente (não necessariamente FCFS)
  - **Push**: enviar objetos não solicitados ao cliente!
  - **Framing**: Dividir os objetos em **frames**, e agendar as frames de modo a mitigar o bloqueio de HOL



# HTTP/2: atenuar HOL blocking



HTTP/1.1: o cliente solicita 1 objeto grande (por exemplo, ficheiro de vídeo) e 3 objetos mais pequenos

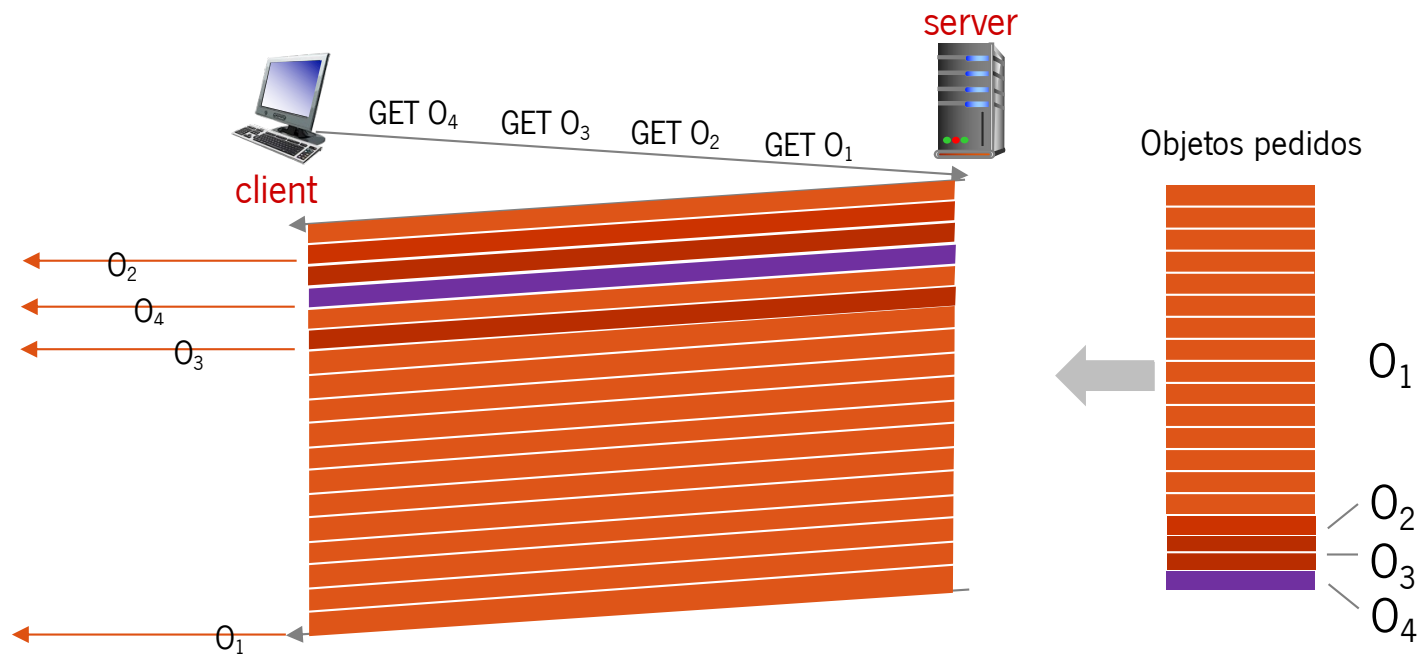


*objetos entregues pela ordem solicitada: O<sub>2</sub>, O<sub>3</sub>, O<sub>4</sub> esperam atrás de O<sub>1</sub>*

# HTTP/2: atenuar HOL blocking

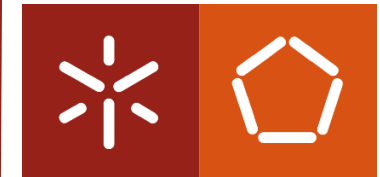


HTTP/2: objetos divididos em frames e transmissão intercalada de frames



*O<sub>2</sub>, O<sub>3</sub>, O<sub>4</sub> recebidos rapidamente, O<sub>1</sub> receção ligeiramente atrasada*

# HTTP2 – Tudo num único slide!



## 1. Uma única conexão TCP!

## 2. Request → Stream

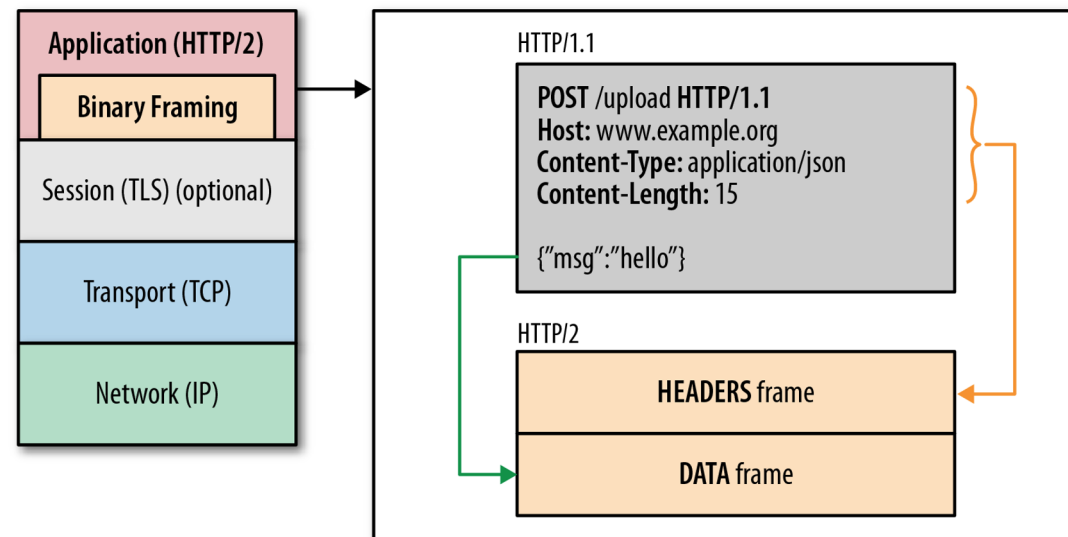
- Streams são multiplexadas!
- Streams são priorizadas!

## 3. Camada de “framing” binário

- Priorização
- Controlo de Fluxo
- Server push

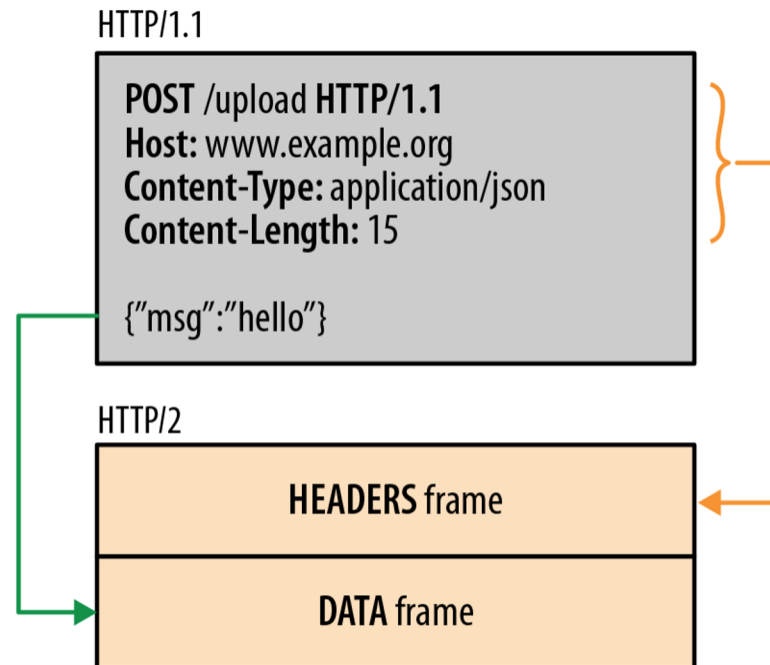
## 4. Compressão do cabeçalho (HPACK)

- No HTTP/3: compressão QPACK





# HTTP2 – “Framing” binário

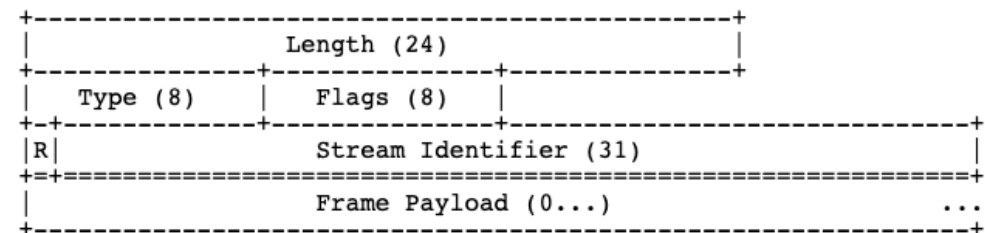


- **Mensagens HTTP** são divididas em uma ou mais **frames**

- 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

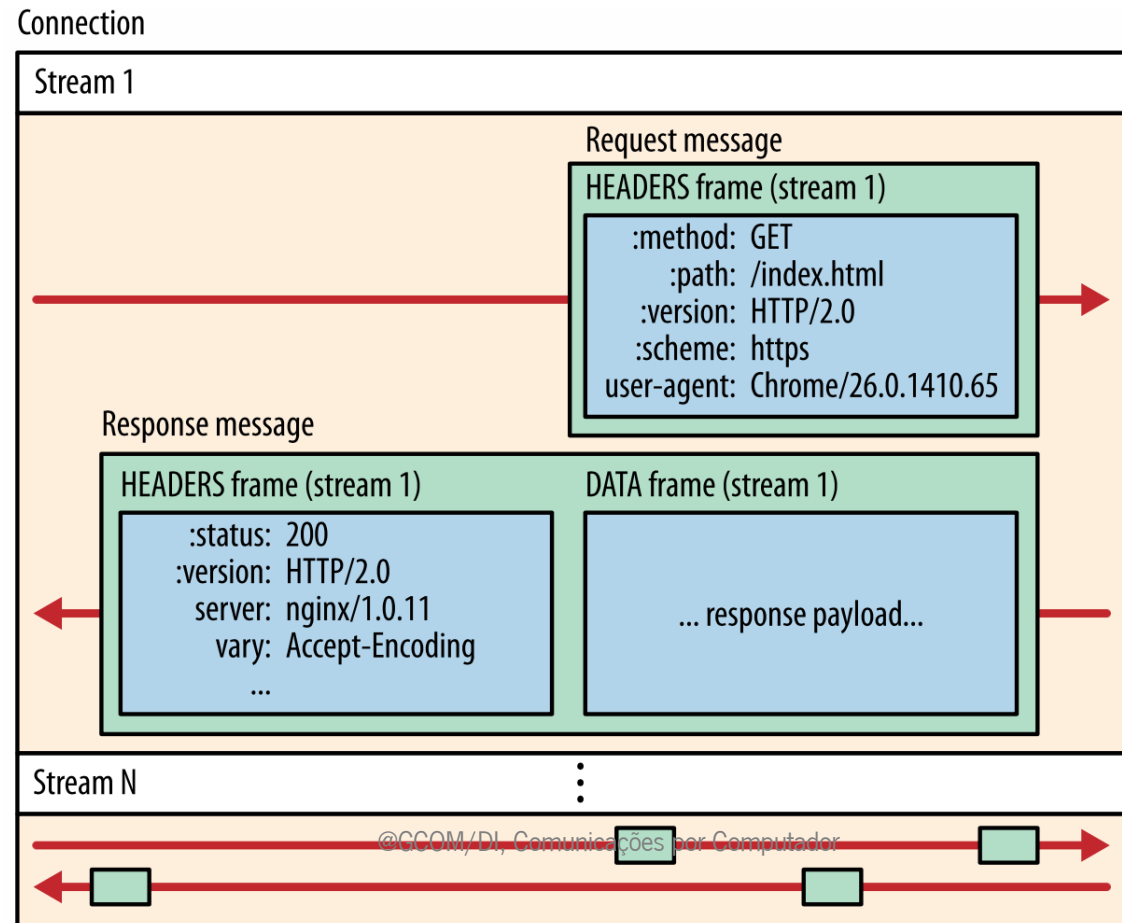


# HTTP2 – “Framing” binário

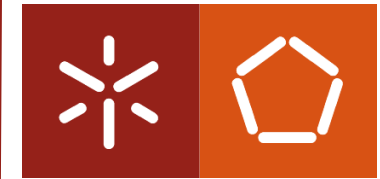


- **Terminologia HTTP2**

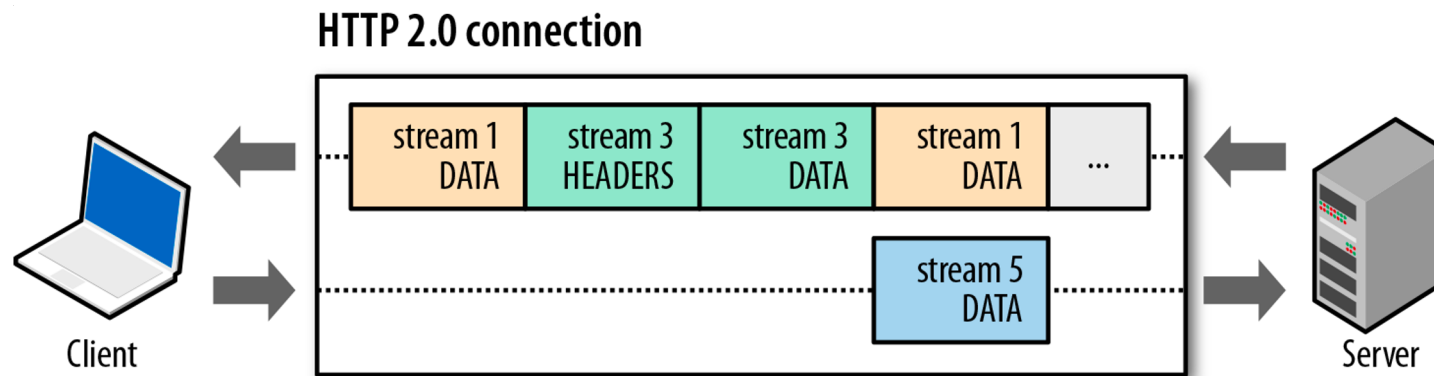
- **Stream** – um fluxo bidirecional de dados, dentro de uma conexão, que pode carregar uma ou mais mensagens
- **Mensagem** - Uma sequência completa de *frames* que mapeiam num pedido ou numa resposta HTTP
- **Frame** – A unidade de comunicação mais pequena no HTTP2, contendo um cabeçalho que no mínimo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



# HTTP2 – fluxo de dados



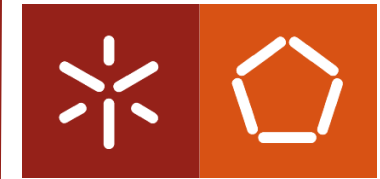
- Fluxo de dados numa conexão HTTP2



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
- A 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

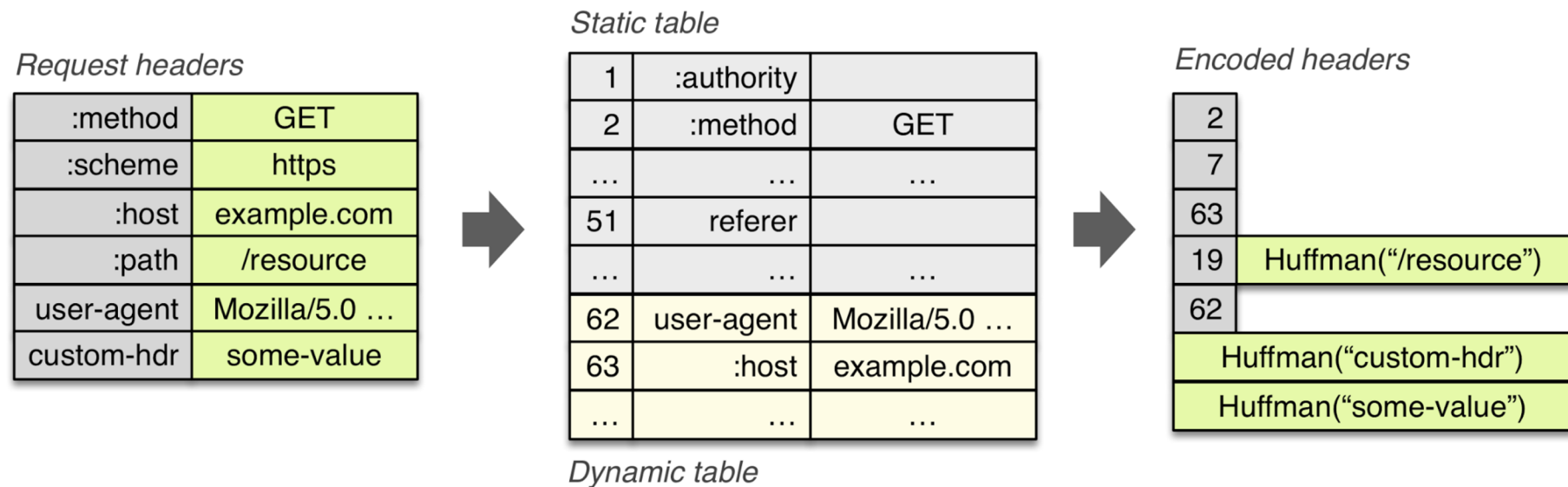


- **As *frames* definidas no RFC7540 são:**
  - HEADERS – *headers* de um pedido ou de uma resposta
  - DATA – corpo dos objetos (dados)
  - PRIORITY – define a prioridade da *stream* para o originador
  - RST\_STREAM – permite o término imediato da *stream*
  - SETTINGS – para definir parâmetros de 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



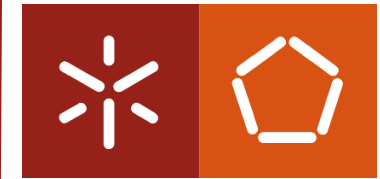
## ● HPACK



- 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!



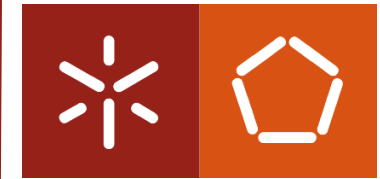
# HTTP2 – Server “push”



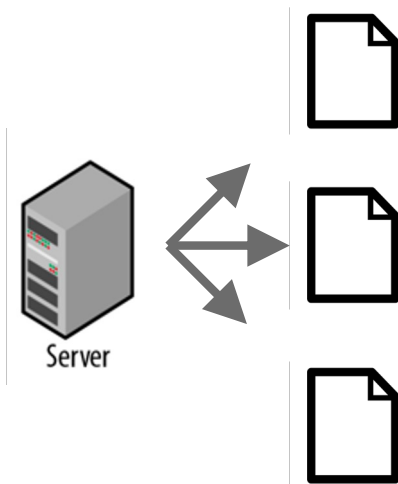
**Server:** “You asked for `/product/123`, but you’ll need `app.js`, `product-photo-1.jpg`, as well... I promise to deliver these to you. That is, unless you decline or cancel.”

- 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 ou quero mais

# HTTP2 – Server “push”



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



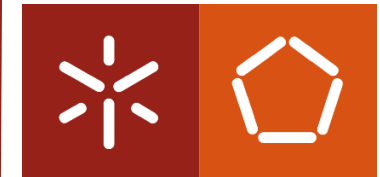
## 1. Servidor observa o tráfego de entrada

- 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

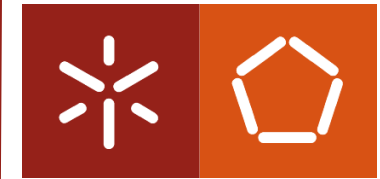


I want image geometry and preview, and I'll fetch the rest later...

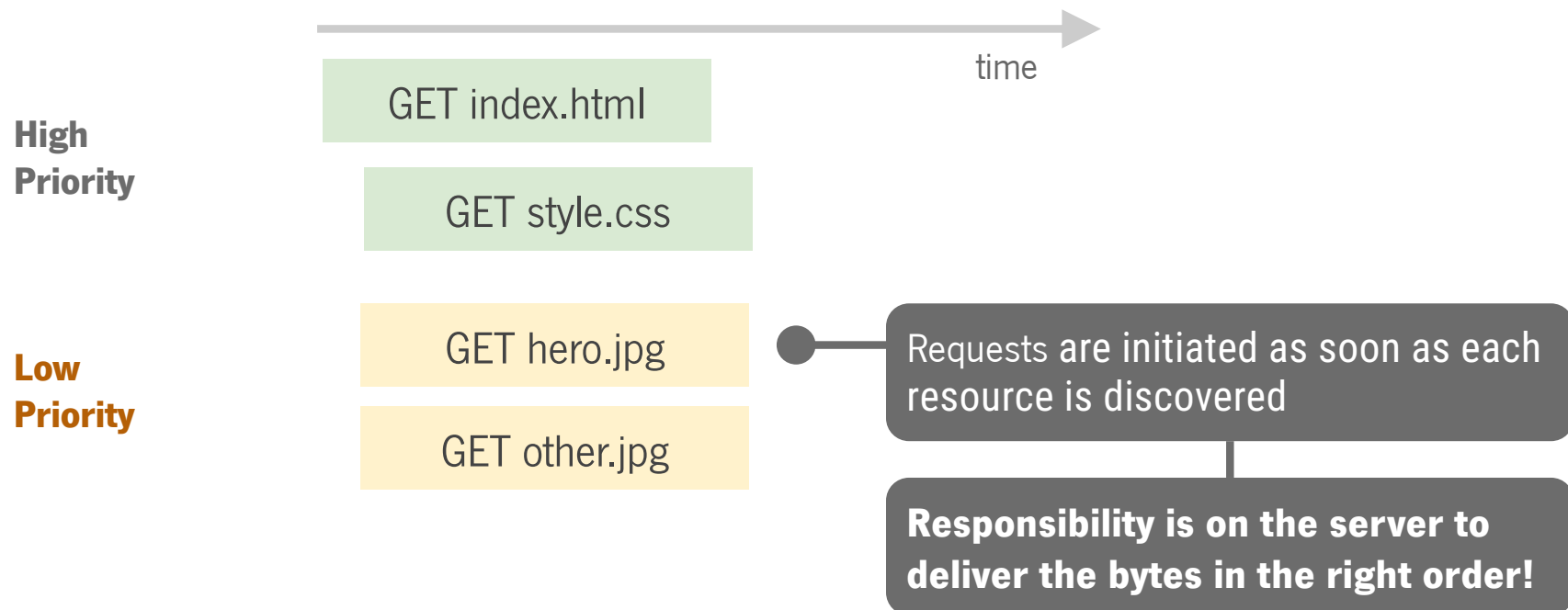
- **Client:** "I want first 20KB of photo.jpg"
- **Server:** "Ok, 20KB... pausing stream until you tell me to send more."
- **Client:** "Send me the rest now."

- 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



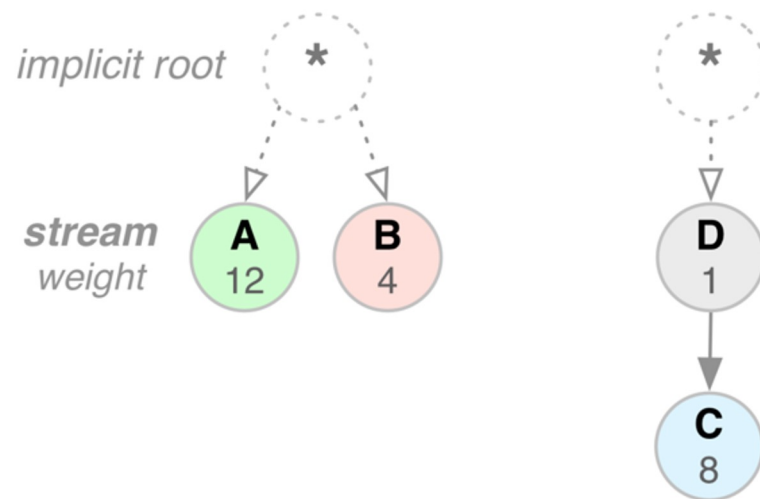
- Priorização é fundamental para um *rendering* eficiente!
- Com HTTP2, o cliente define as prioridades e faz logo os pedidos; cabe ao servidor entregar os conteúdos com a prioridade certa



# HTTP2 – pesos e dependências

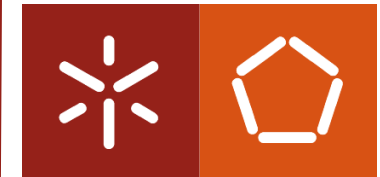


- Exemplo: stream A deve ter 12/16 e a B 4/16 dos recursos totais
- Exemplo: stream D deve ser entregue antes da stream C



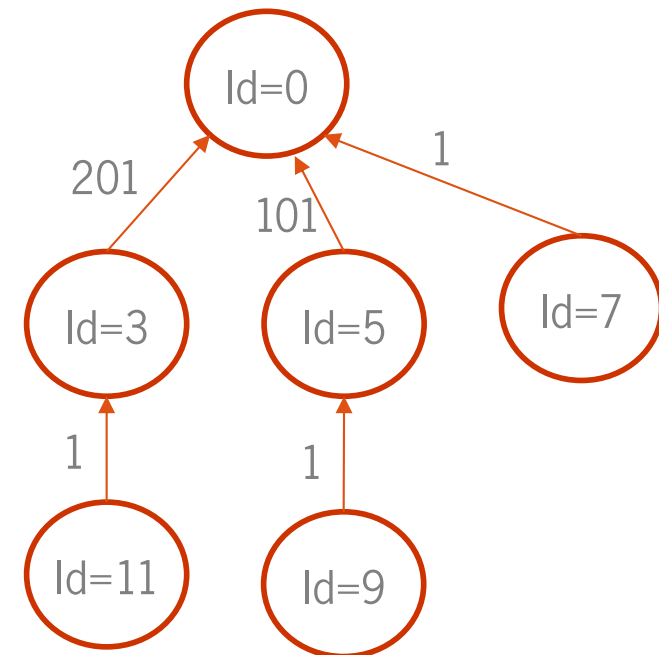
- **Cada stream pode ter um peso**
  - [1-256] integer value
- **Cada stream pode ter uma dependência**
  - ... uma outra stream ID

# 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 respectivas dependências
- A *stream* 0 não existe (apenas raiz)
- 0 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



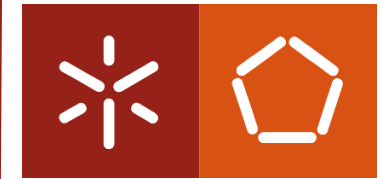
# HTTP2 – Negociação protocolar



- **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

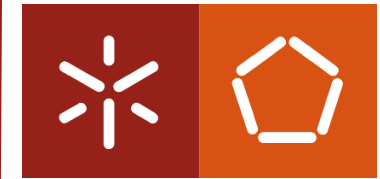
2. Settings codificados em BASE64

3. Servidor declina pedido, respondendo em HTTP/1.1

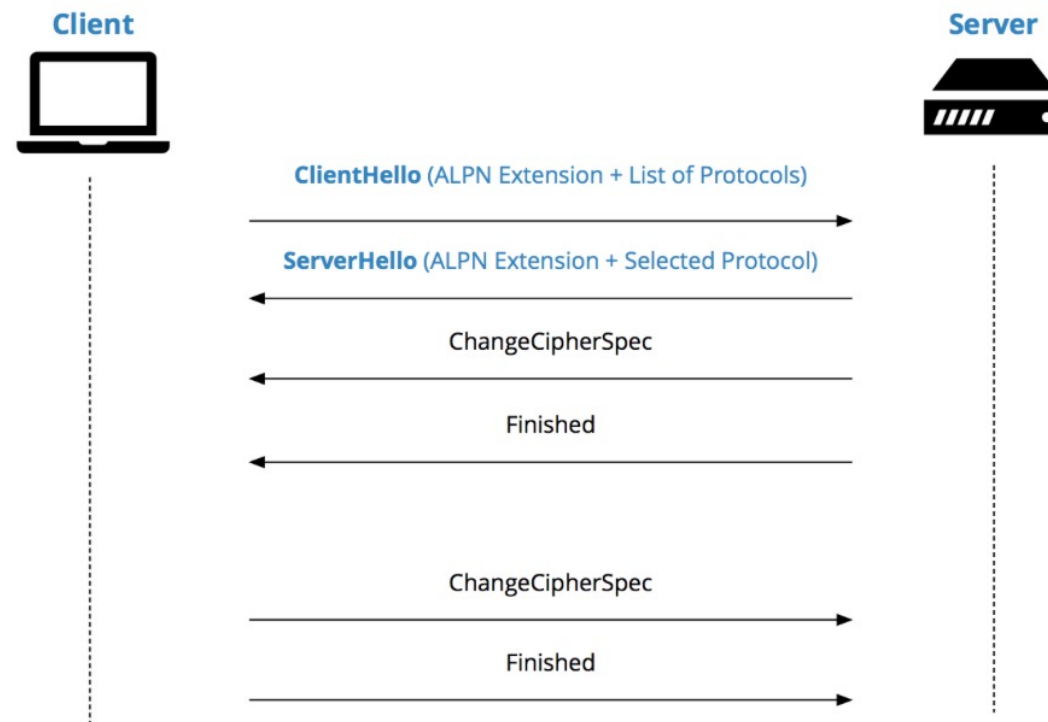
4. Servidor aceita pedido para HTTP2 e começa Framing binário



# HTTP2 – Negociação protocolar



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

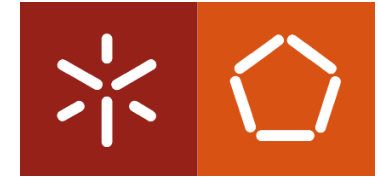


# HTTP2 – Negociação protocolar



- **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/>)**

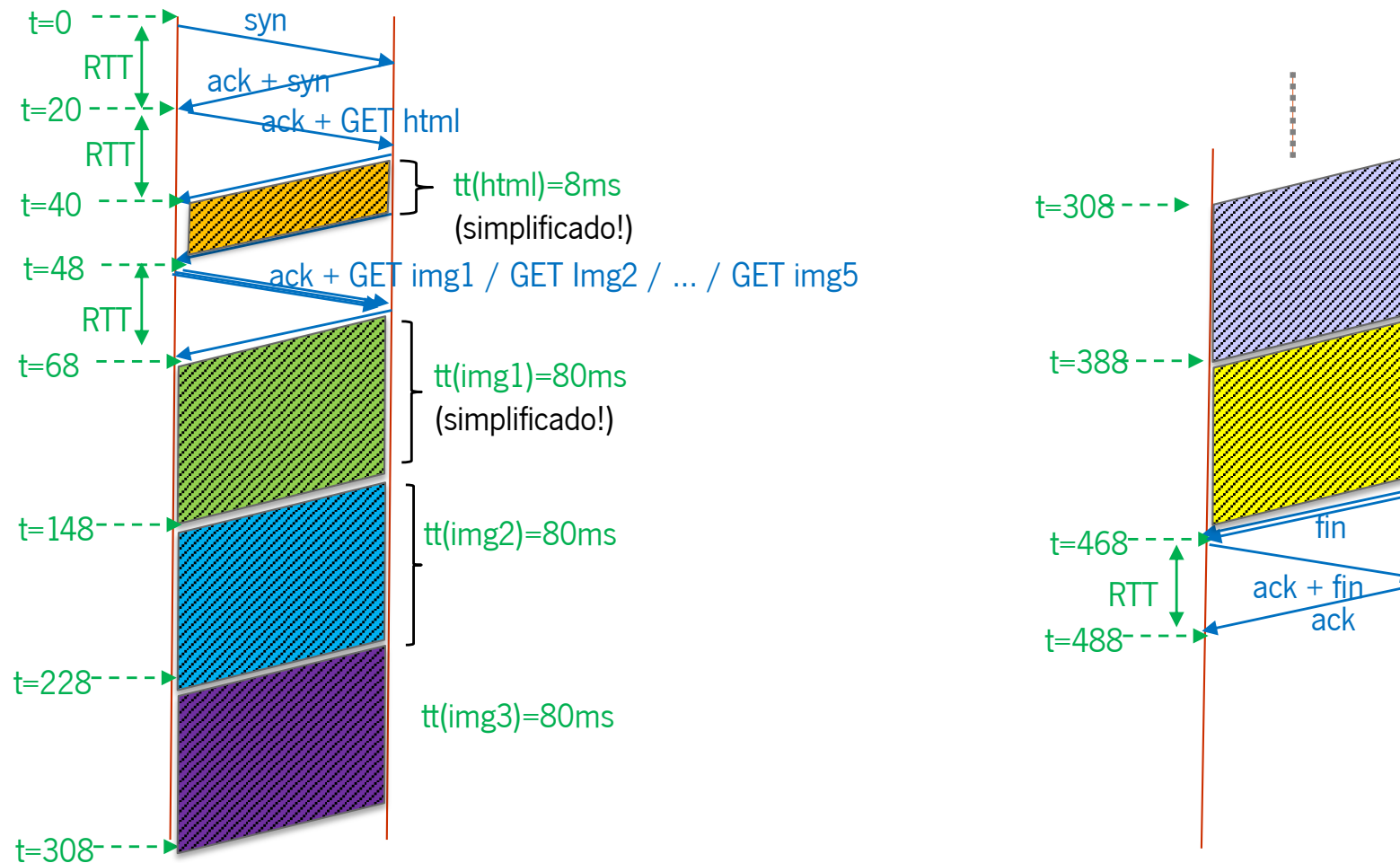
Exemplo:

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

# Relembrar o exercício HTTP



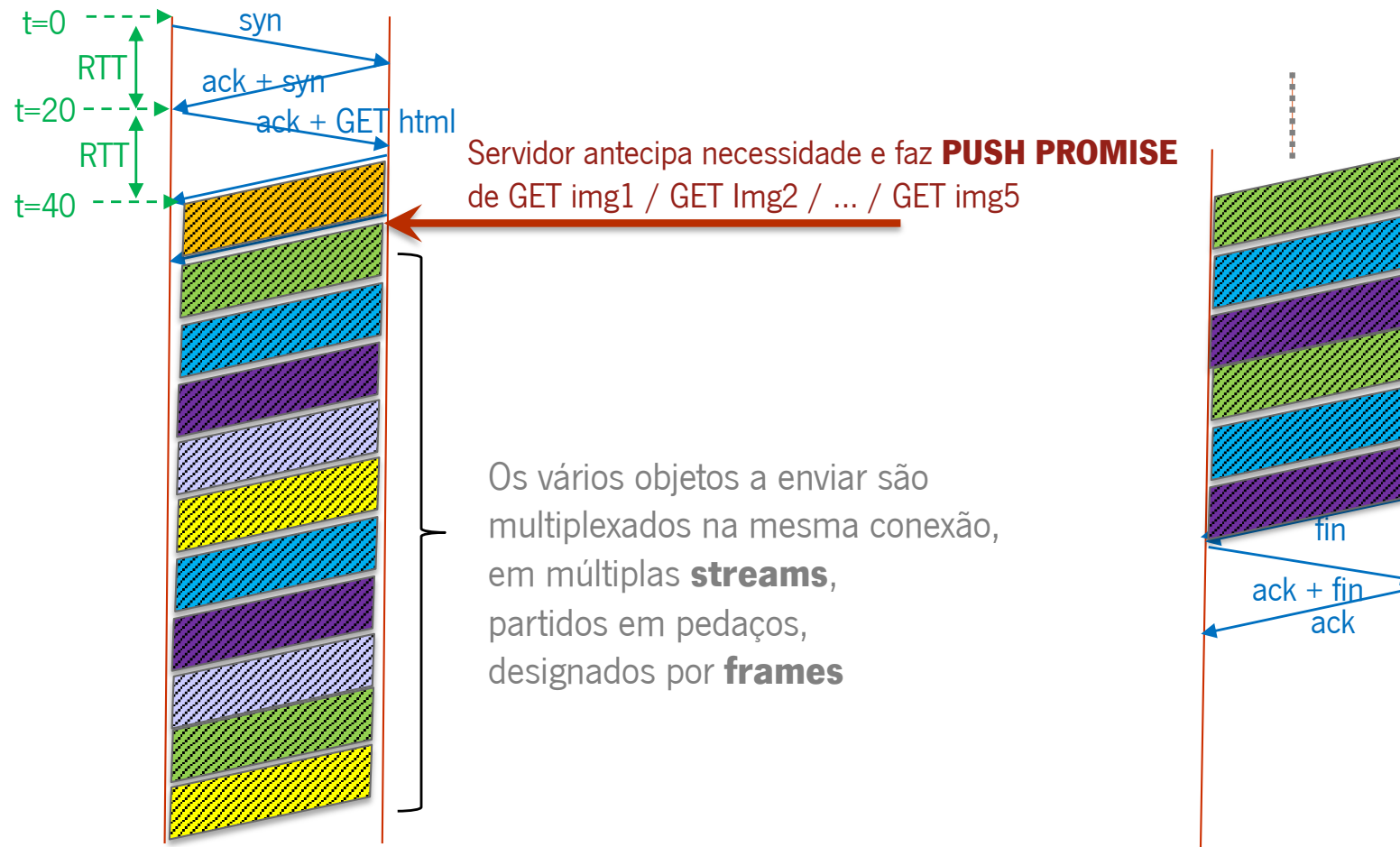
## c) HTTP/1.1 persistente, com pipeline, sem conexões em paralelo



# Relembrar o exercício HTTP

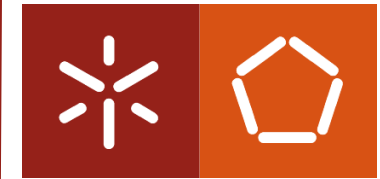


## d) HTTP2 múltiplas streams numa mesma conexão com Server Push



**T.c.total = 448 ms (ou 468 ms contando com fecho conexão)**

# HTTP 3 + QUIC



- **Razões para a evolução de HTTP/2 para HTTP/3**
  - a recuperação da perda de pacotes ainda paralisa todas as transmissões de objetos
    - tal como no HTTP 1.1, os browsers têm incentivo para abrir múltiplas ligações TCP paralelas para reduzir a paragem e aumentar o rendimento global
  - sem segurança na ligação TCP base (TLS opcional)
  - HTTP/3: adiciona segurança, controlo de erros e congestionamento por objeto (mais pipeline) sobre UDP

# HTTP3 + QUIC

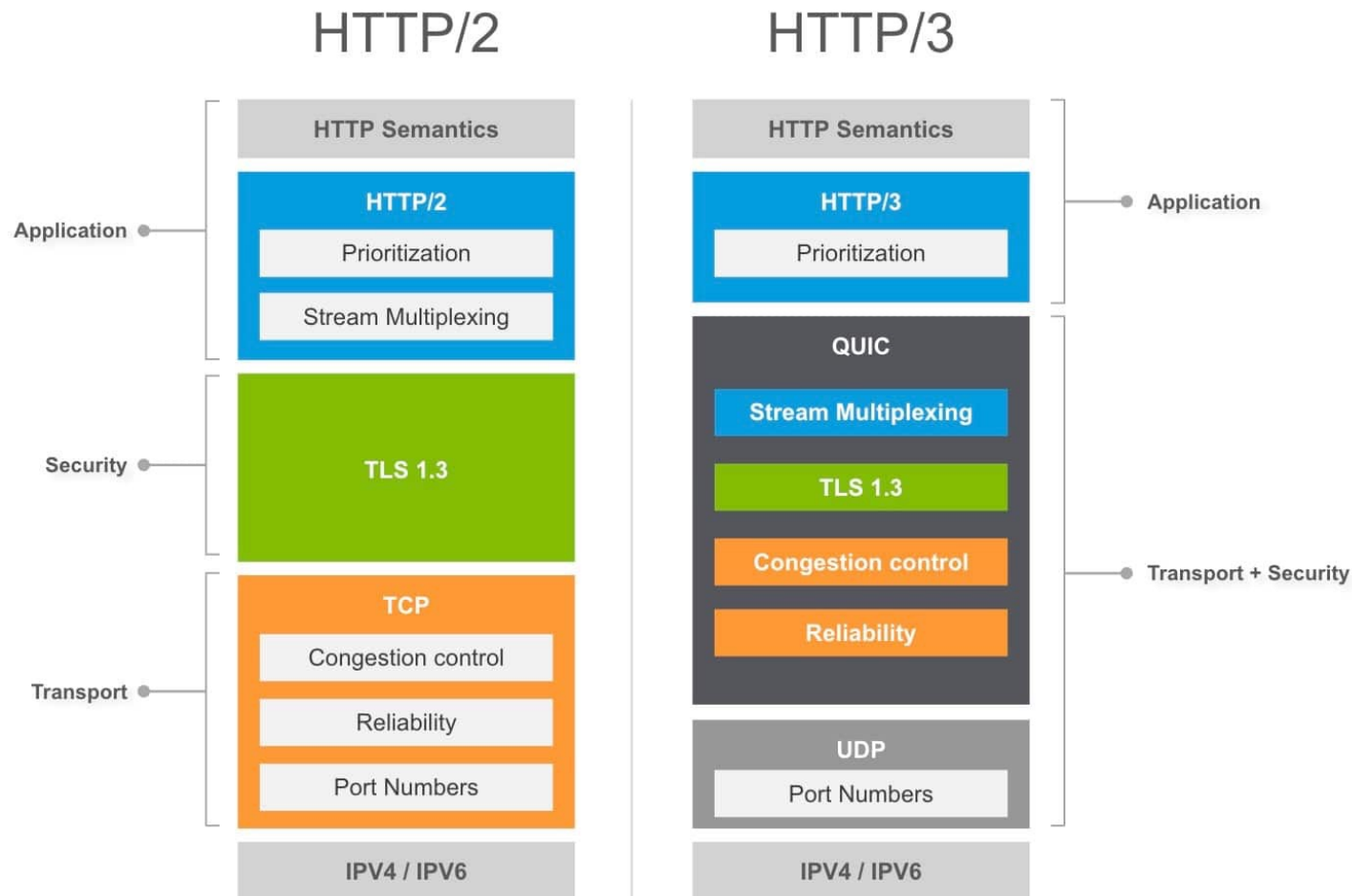
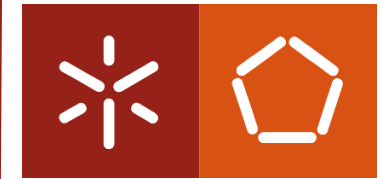


Imagem <https://www.akamai.com/blog/performance/deliver-fast-reliable-secure-web-experiences-http3>

# HTTP3 + QUIC



- **O que muda:**

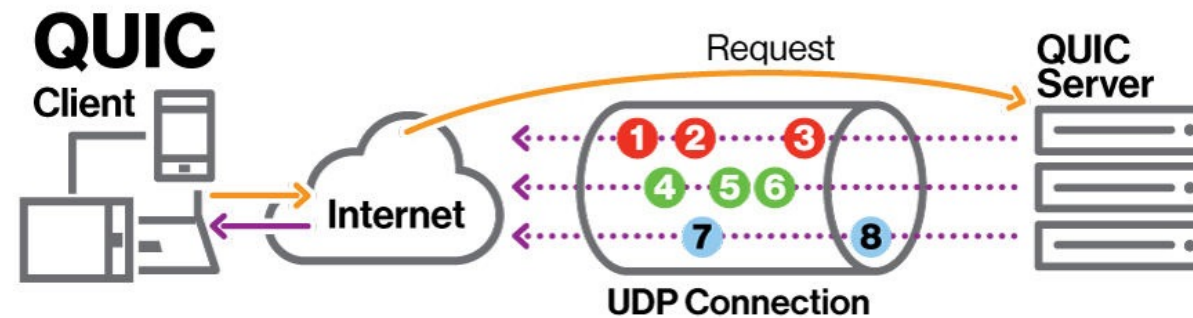
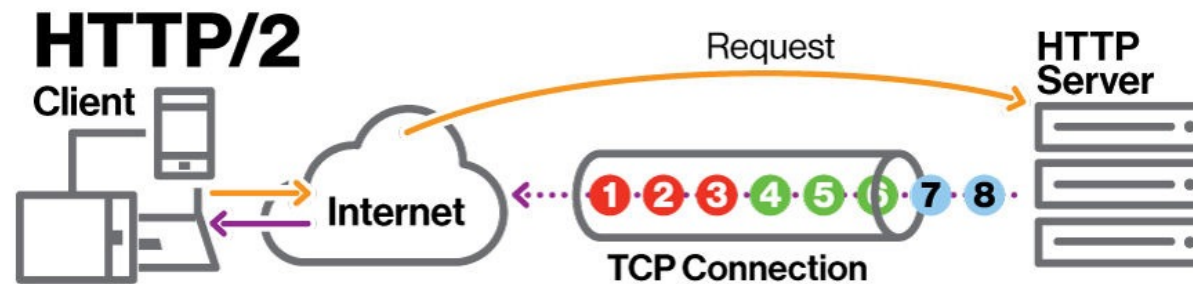
- Protocolo HTTP/3 lida apenas com as prioridades
- Multiplexagem de frames passa para um protocolo próprio: QUIC
- Camada TLS de segurança (que abordaremos mais tarde) integrada no QUIC
- QUIC corre sobre protocolo não fiável UDP
  - Incluir controlo de erros e controlo de congestão no QUIC

- **QUIC**

- Em que camada está realmente?
- Orientado à conexão, garante entrega ordenada dentro de uma *stream* (não entre *streams*), recuperação de erros com retransmissão, com cigrafem, usa de Stream ID para multiplexagem de *streams*

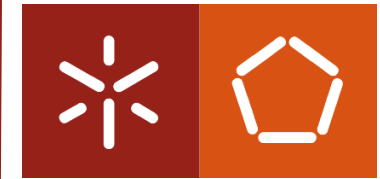


# HTTP3 + QUIC



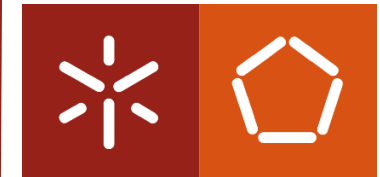
*Imagem Devopedia. 2021. "QUIC." Version 5, March 8 (CC BY-SA 4.0)*

# HTTP3 + QUIC



- **HTTP/3 aproveita as alterações já feitas no HTTP/2**
- **Usa QUIC sobre UDP em vez de TCP na camada de transporte**
- **Fundamentalmente, mapeia a semântica no QUIC**
- **Questões:**
  - Como é que os clientes sabem se os servidores suportam HTTP3?
  - Qual a estrutura das mensagens em HTTP3?
  - Como se mapeiam as mensagens em *streams* QUIC?

# HTTP3 + QUIC



- **Conexões HTTP/3**

- Não se pode assumir que o servidor suporta HTTP/3 pois isso pode causar muitos problemas de desempenho aos servidor HTTP/2 e HTTP/1.1
- QUIC usa TLS, por isso, os URLs começam sempre com https://
- O mecanismo de “*upgrade*” não faz sentido, pois o QUIC funciona sobre UDP

- **Método 1**: Alternate services

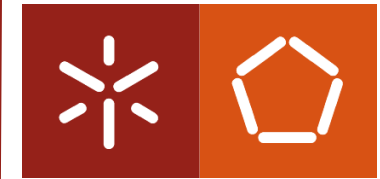
- Novo *header* a anunciar serviço alternativo que o cliente pode usar

```
Alt-Svc: h3=www3.uminho.pt:8003;ma=3600 , h2=:8002;ma=3600
```

- **Método 2**: Anunciar o suporte no serviço de nomes DNS

- Método é semelhante ao anterior, mas recorre ao serviço de nomes DNS e a registos próprios como o SVCB e HTTPS para anunciar disponibilidade prévia de HTTP/3
- Inicia-se depois uma conexão QUIC, com uma *stream* de controlo onde se trocam *frames* com os SETTINGS

# HTTP3 + QUIC



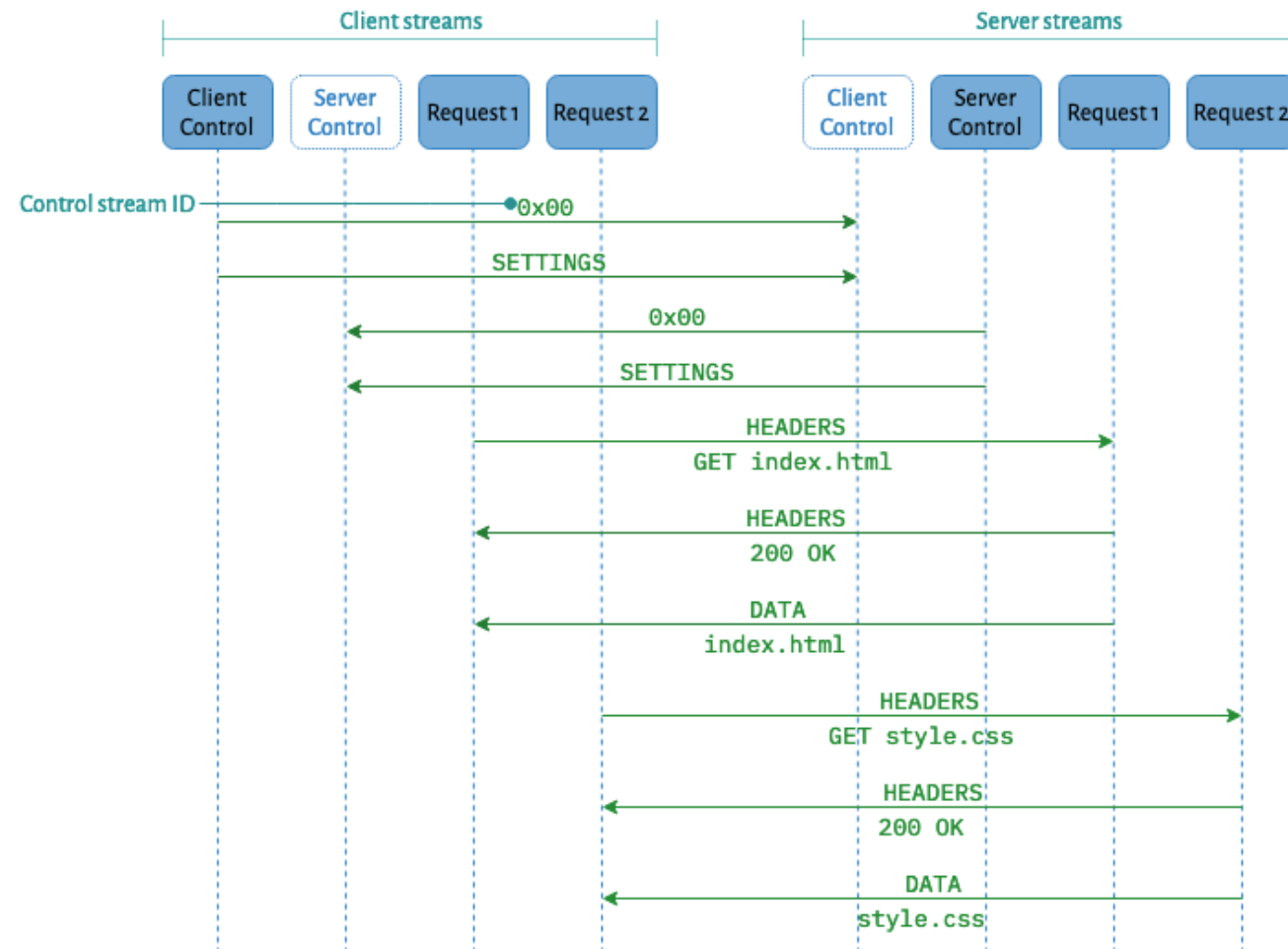
- **Formato das mensagens HTTP/3**

- Depois da conexão QUIC iniciada criam-se múltiplas *streams*
- **Streams bidirecionais** para pedidos e respostas
  - Uma mensagem HTTP é formada por uma *frame* HEADERS e opcionalmente uma ou mais *frames* DATA
- **Streams unidirecionais**, em cada sentido:
  - 4 streams iniciais:
    - *0x00 Control* – para mensagens de controlo de toda a conexão, onde são transferidas por exemplo as frames de SETTINGS e GOAWAY
    - *0x01 Push* – criadas pelo servidor para permitir “*Server push*”
    - *0x02 Encoder* – para uso do controlo de compressão QPACK
    - *0x03 Decoder* – para uso do protocolo de compressão QPACK

# HTTP3 + QUIC

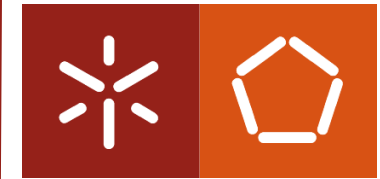


- Exemplo



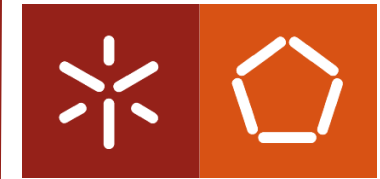
<https://www.andy-pearce.com/blog/posts/2023/Apr/http3-in-practice-http3/>

# QPACK Header compression



- Objetivo:
  - adaptar a compressão dos cabeçalhos HTTP ao protocolo QUIC
- Diferente porquê?
  - No HTTP/2 todas as frames vão na mesma conexão TCP que garante a entrega ordenada
  - Não há preocupação com a tabela de compressão dinâmica: estará sempre no mesmo estado do lado do cliente e do lado do servidor
  - Isso não acontece no QUIC que apenas garante a entrega ordenada numa stream, e não nas várias streams, pois corre sobre UDP!
- Solução:
  - QPACK requiere uma *stream* unidireccional para sincronização do estado das tabelas dinâmicas de compressão
  - Emissor inicia uma **Encoder Stream** unidireccional para envio
  - Recetor inicia uma **Decoder Stream** unidireccional para receção
  - RFC define *encoder instructions* e *decoder instructions* para modificar o estado

# QPACK Header compression



- **Não há grandes diferenças na forma como as tabelas dinâmicas são mantidas**
  - Continua a ser uma lista FIFO com capacidade máxima (definida zero, mas pode-se anunciar uma capacidade máxima)
  - Campos referidos nas frames HEADER são marcados e não podem ser removidos da tabela, as restantes podem ser removidas se faltar espaço
- ***Encoder instructions:***
  - *Set Dynamic Table capacity*
  - *Insert with Name Reference, Insert with Literal Name, Duplicate*
- ***Decoder instructions:***
  - *Section Acknowledgement*
  - *Insert Count Increment*
  - *Stream cancelation*

# QPACK Header compression



- **Encoder instructions:**

- *Set Dynamic Table capacity* – pede ao *decoder* para alterar a capacidade da tabela
- *Insert with Name Reference, Insert with Literal Name, Duplicate* – enviadas depois do emissor alterar a sua tabela, para levar o recetor a alterar a sua tabela
  - Escolhe-se a instrução de acordo com o estado anterior da tabela
  - Se existe o par `name : value`, usa-se *Duplicate*
  - Se existe apenas o `name`, usa-se *Insert with Name reference*
  - Se não existe nada, usa-se *Insert with Literal Name*

- **Decoder instructions:**

- Instruções informam o encoder sobre o processamento das cabeçalhos e atualizações da tabela dinâmica, garantindo a coerência das tabelas entre emissor e recetor
  - *Section Acknowledgement*: enviada assim que o decoder descodifica os campos do header
  - *Insert Count Increment*: confirmação semelhante à anterior, confirma que novas entradas foram recebidas e inseridas corretamente na tabela dinâmica
  - *Stream Cancellation*: receptor decide cancelar a stream por razões que não um encerramento normal (ex: sequência errada de frames) e apagar a tabela dinâmica



# QUIC – Visão geral



- **Datagramas, pacotes e frames**

- Cada datagrama UDP pode conter ou mais pacotes
- Cada pacote pode conter uma ou mais frames, conforme o tipo
- A inclusão de múltiplos pacotes num datagrama ocorre normalmente no *handshake* inicial

- **Numeração e confirmação dos pacotes**

- A fiabilidade aplica-se ao nível do pacote e não da frame
- A numeração dos pacotes começa sempre em 0 (zero)
- As confirmações seguem em *frames* próprias ACK
- A *frame* de ACK é mais complexa porque não é apenas um ACK cumulativo, mas o pacote com número mais alto recebido e uma lista de buracos para indicar os que faltam!

- **Controlo de fluxo**

- **Controlo de congestão**