

HTTP

Baseado no Capítulo 2 do livro
Computer Networking: A Top Down Approach,
Jim Kurose, Keith Ross, Addison-Wesley ©2016

Comunicações por Computador

Licenciatura em Engenharia Informática

Universidade do Minho



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/miei.html`

host name path name

HTTP

Como funciona?



- **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



Pedido HTTP

Resposta HTTP

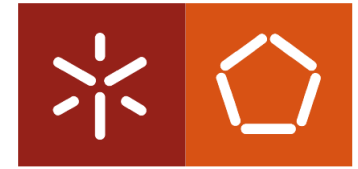
Servidor a
executar
o servidor
WEB Apache



Mac a executar o
Safari

HTTP

Como funciona?



Utiliza o TCP:

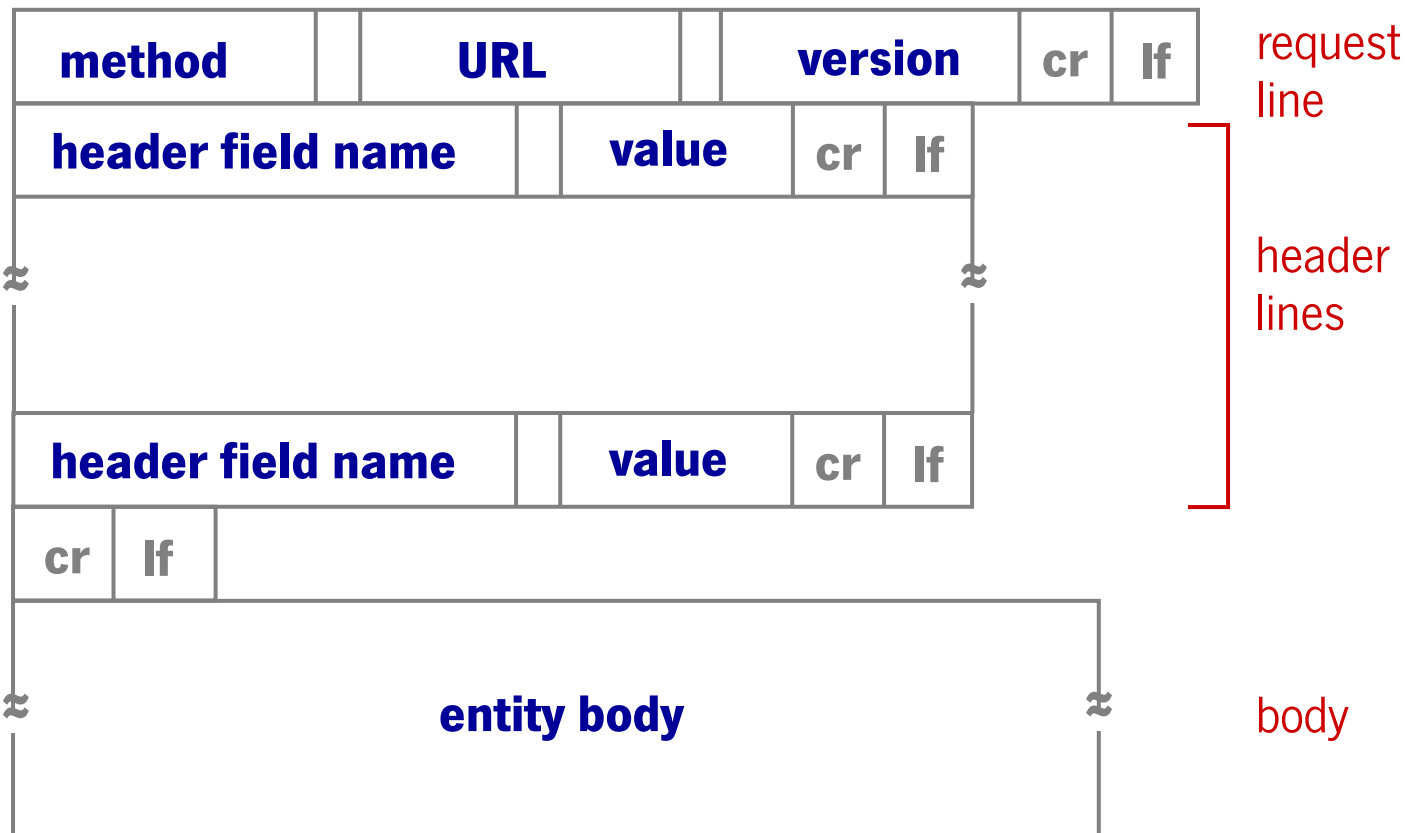
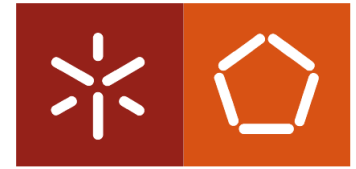
- O cliente cria um *socket* e inicia uma conexão TCP com um servidor HTTP (por defeito, à escuta na porta 80);
- O servidor TCP aceita o pedido de conexão do cliente;
- São trocadas mensagens HTTP (mensagens de protocolo de nível aplicacional) entre o *browser* (cliente HTTP) e o servidor web/HTTP;
- 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 duma mensagem...



HTTP

Exemplo da sintaxe duma mensagem...



HTTP Request Message Example:

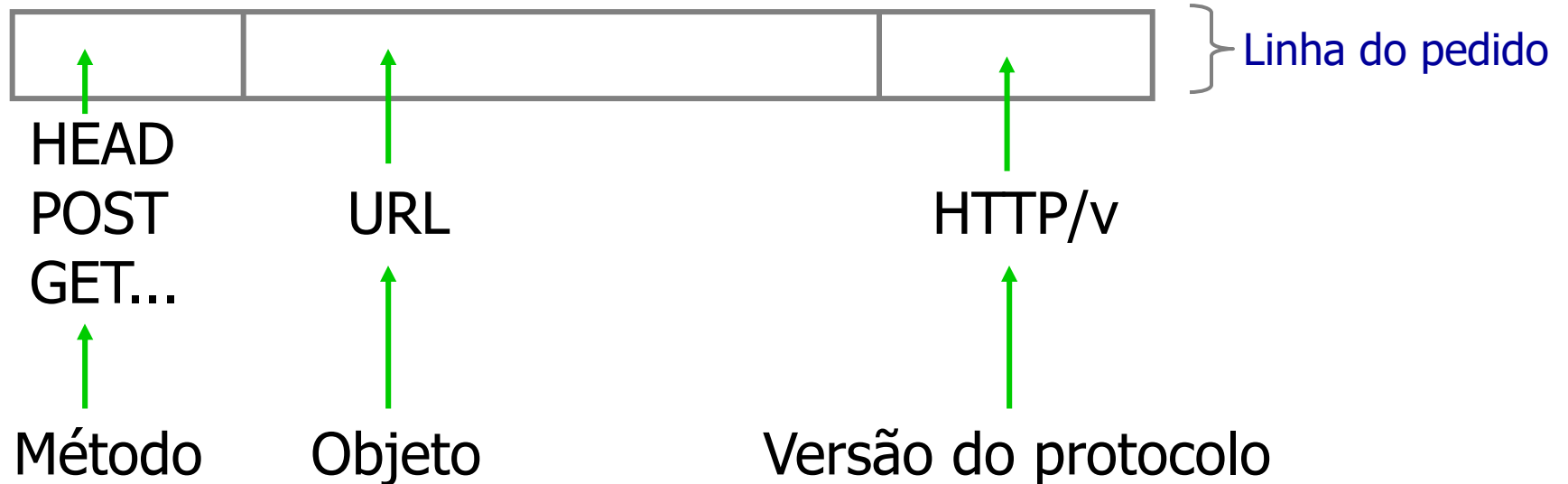
GET	/directoria/pagina.html	HTTP/1.1	}	Linha do pedido
Host: www.sitio.pt				
Connection: close			}	Linhas do cabeçalho
User-Agent: Mozilla/4.0				
Accept-Language: PT				
<new line>				
Corpo da mensagem (<i>vazio no caso do GET</i>)			}	Dados da mensagem

HTTP

Sintaxe duma mensagem...

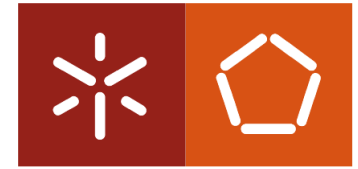


HTTP Request Message:



HTTP

Input dados num formulário...



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

Métodos



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)

HTTP

Métodos (baseados no paradigma RESP API)



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 pedido HTTP!	Lista todos os livros; Pedido: vazio; Resposta: listagem de livros.	Atualiza um conjunto de livros passados no 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

Exemplo de outra mensagem...



HTTP Response Message:

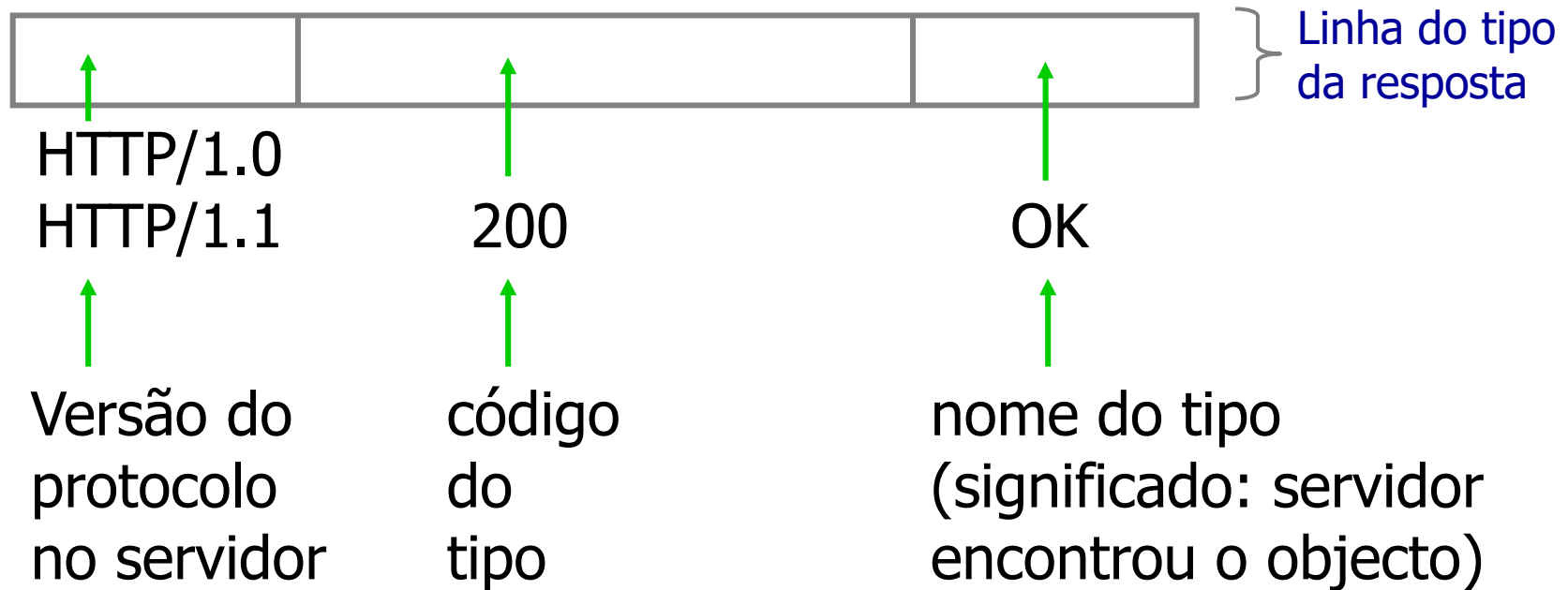
HTTP/1.1	200	OK	}	Linha do tipo da resposta
Connection: close				
Date: 07 Mai 2003 11:35:15 UTC+1			}	Linhas do cabeçalho
Server: Apache/1.3.0 (Unix)				
Last-Modified: 05 Mai 2003 09:23:45 UTC+1				
Content-Length: 6825				
Content-Type: text/html				
<new line>			}	Dados da mensagem
Corpo da mensagem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

HTTP

Sintaxe de outra mensagem...

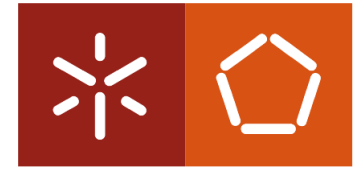


HTTP Response Message:



HTTP

Sintaxe das mensagens



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/Browser de interface simples...



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

```
GET / HTTP/1.1
```

```
Accept: */*
```

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

```
Connection: keep-alive
```

```
Host: www.di.uminho.pt
```

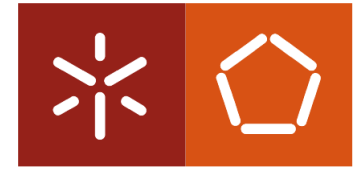
```
User-Agent: HTTPie/2.0.0
```

```
....
```

(ver os últimos slides para mais exemplos com API REST,
e o site do HTTPie <https://httpie.org/>)

HTTP

Tipo de conexões TCP



HTTP não persistente:

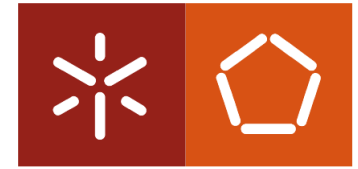
- Só pode ser enviado no máximo um objeto *web* por cada conexão estabelecida.
- O HTTP/1.0 utiliza HTTP não persistente.
- Mínimo de 2 RTT/objeto.
- Exige mais recursos do S.O.
- Alguns browsers abrem várias conexões TCP em paralelo para pedirem vários objetos referidos no mesmo objeto.

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*.

HTTP

Persistente



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 de estratégia 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 de estratégia 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

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

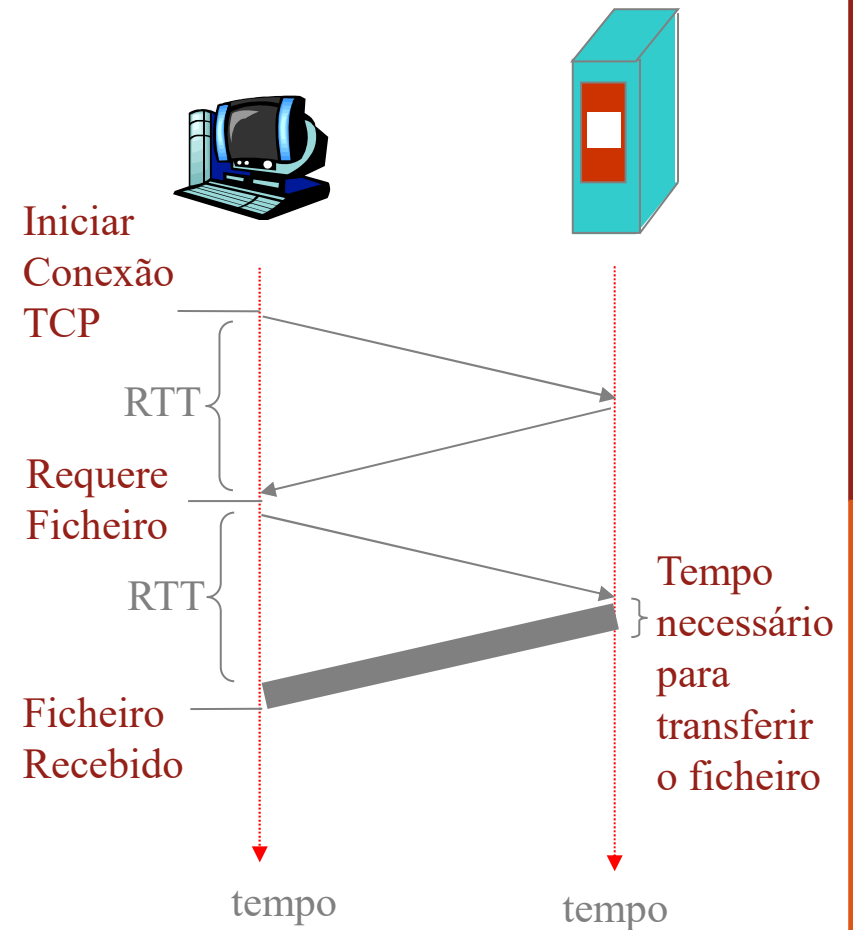
TP: Tempo de Propagação

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



HTTP

Exercícios



1. Pretende-se estimar o tempo mínimo necessário para obter um documento da web. O documento é constituído por 6 objetos: o objeto base HTML e cinco imagens referenciadas no objeto 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 objeto 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 objeto base e que o utilizador sabe o endereço IP do servidor, indicando-o no *browser*. A dimensão e tempo de processamento dos pacotes de estabelecimento de ligação, de confirmação de estabelecimento de ligação e de envio dos pedidos HTTP é desprezável e que 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 objetos) 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 objetos) se utilizar HTTP/1.1 com *pipelining* em todos os pedidos?

Dados: $RTT = 20 \text{ ms}$

6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



Dados: $RTT = 20 \text{ ms}$

6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo

Dados: $RTT = 20 \text{ ms}$

6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

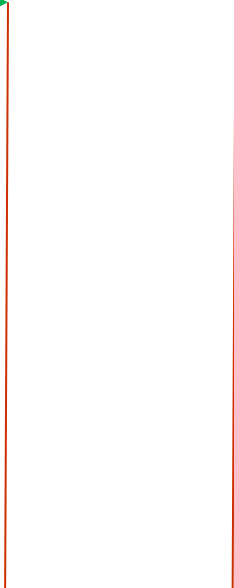
$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo

$t=0$ ----->



Dados: $RTT = 20 \text{ ms}$

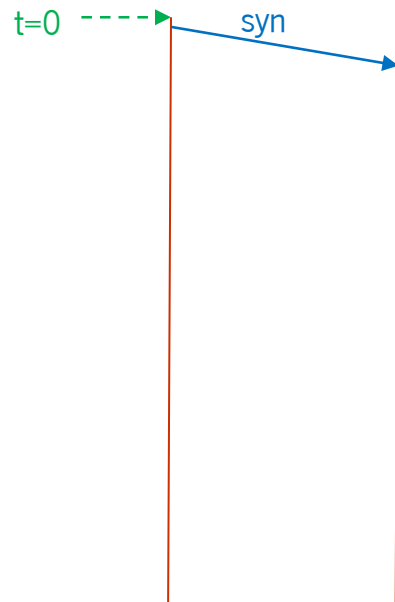
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



Dados: RTT = 20 ms

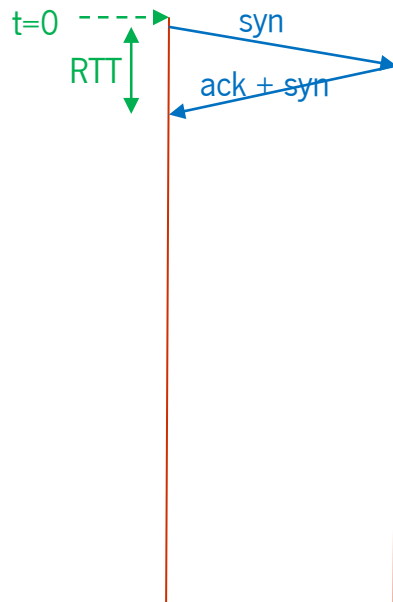
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



Dados: RTT = 20 ms

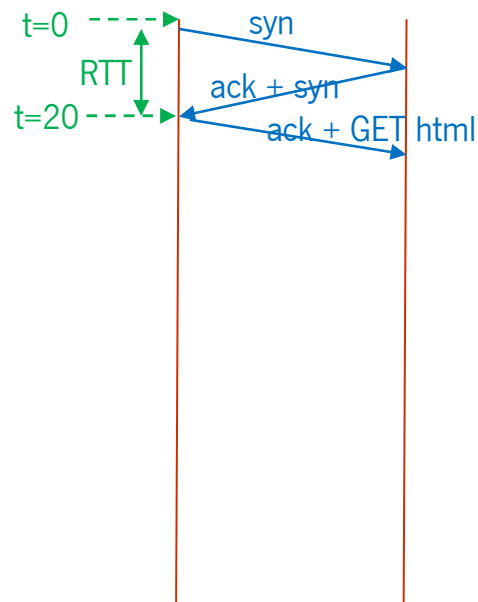
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



Dados: $RTT = 20 \text{ ms}$

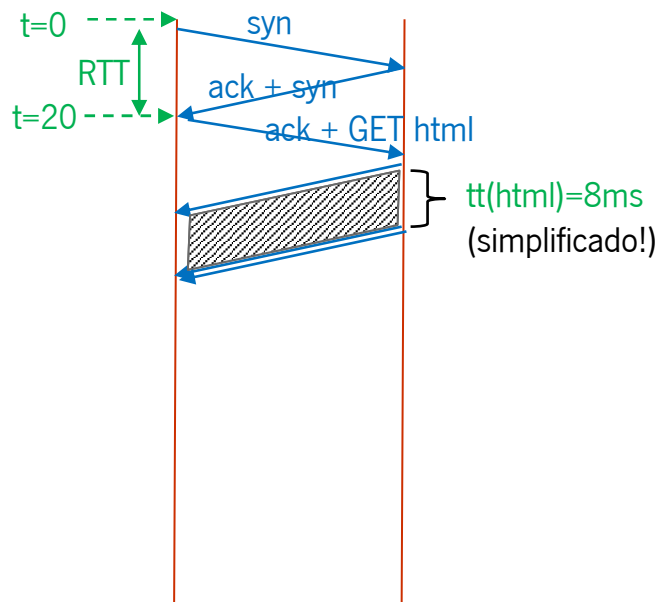
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo

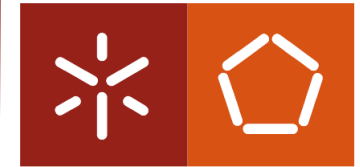


Dados: $RTT = 20 \text{ ms}$

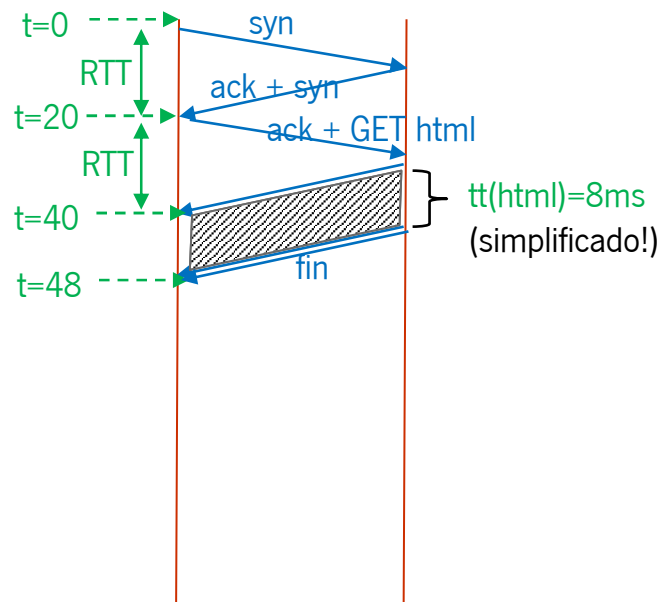
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



Dados: RTT = 20 ms

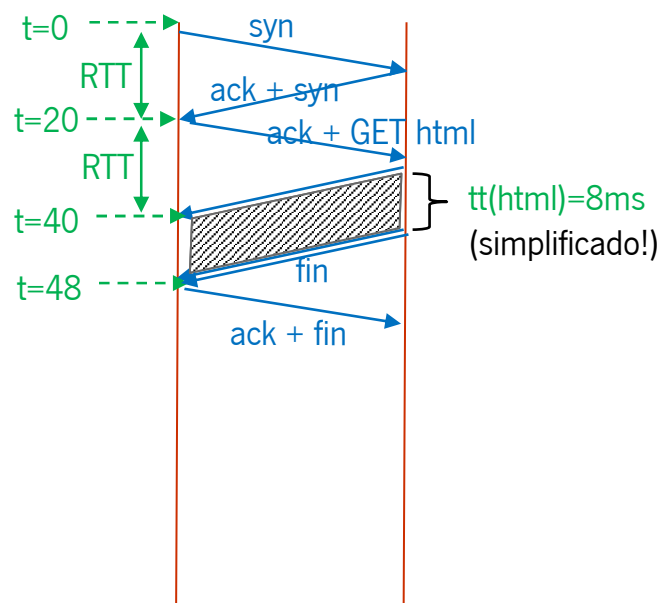
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt(html) = 8 ms

tt(img) = 80 ms



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



Dados: $RTT = 20\text{ ms}$

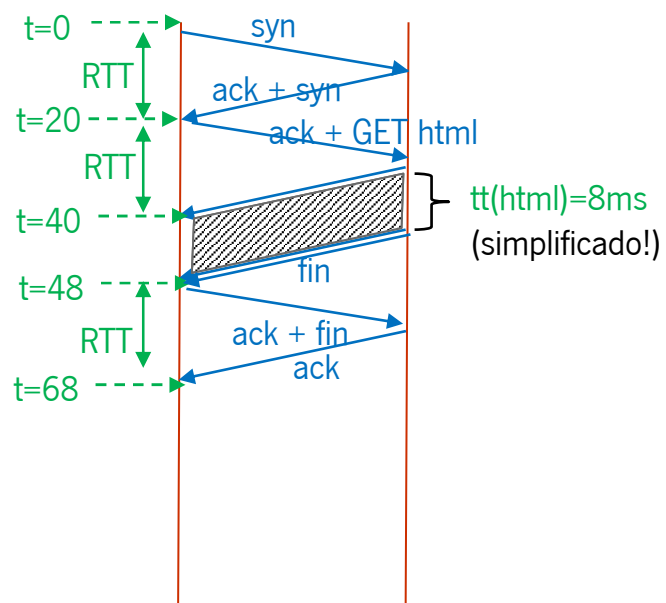
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8\text{ ms}$

$tt(\text{img}) = 80\text{ ms}$



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



Dados: $RTT = 20\text{ ms}$

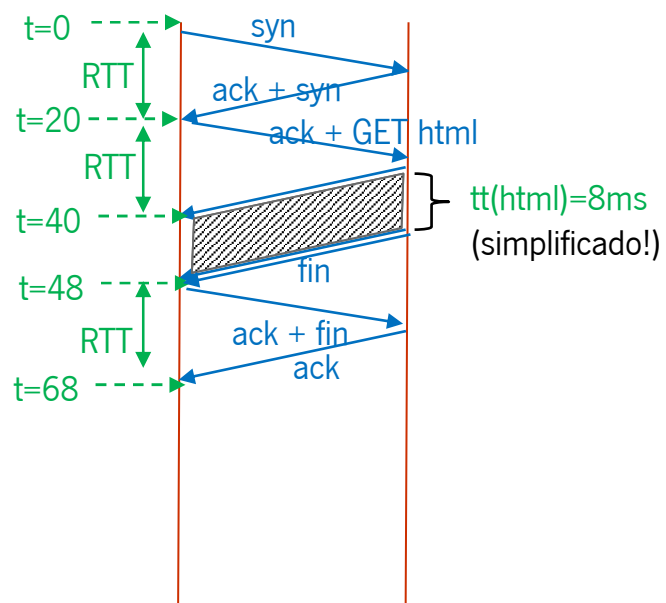
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8\text{ ms}$

$tt(\text{img}) = 80\text{ ms}$



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



Dados: RTT = 20 ms

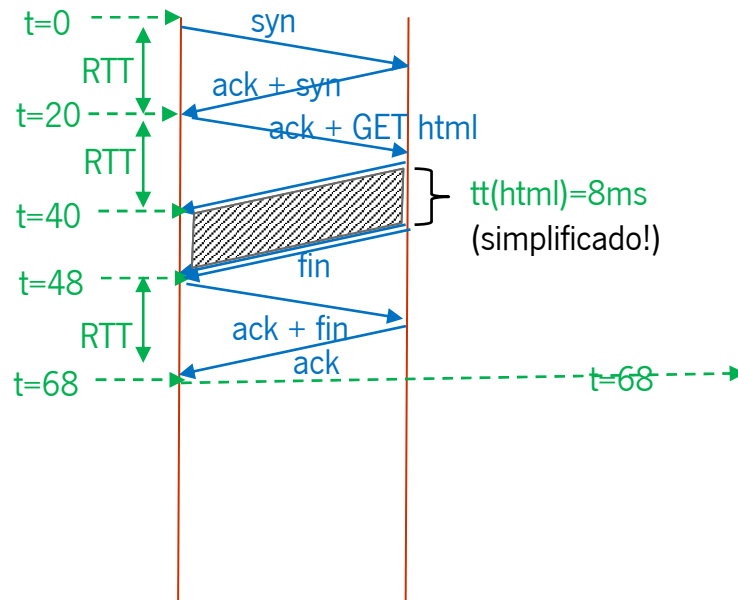
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt(html) = 8 ms

tt(img) = 80 ms



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



4 conexões em paralelo (4 threads)

Dados: RTT = 20 ms

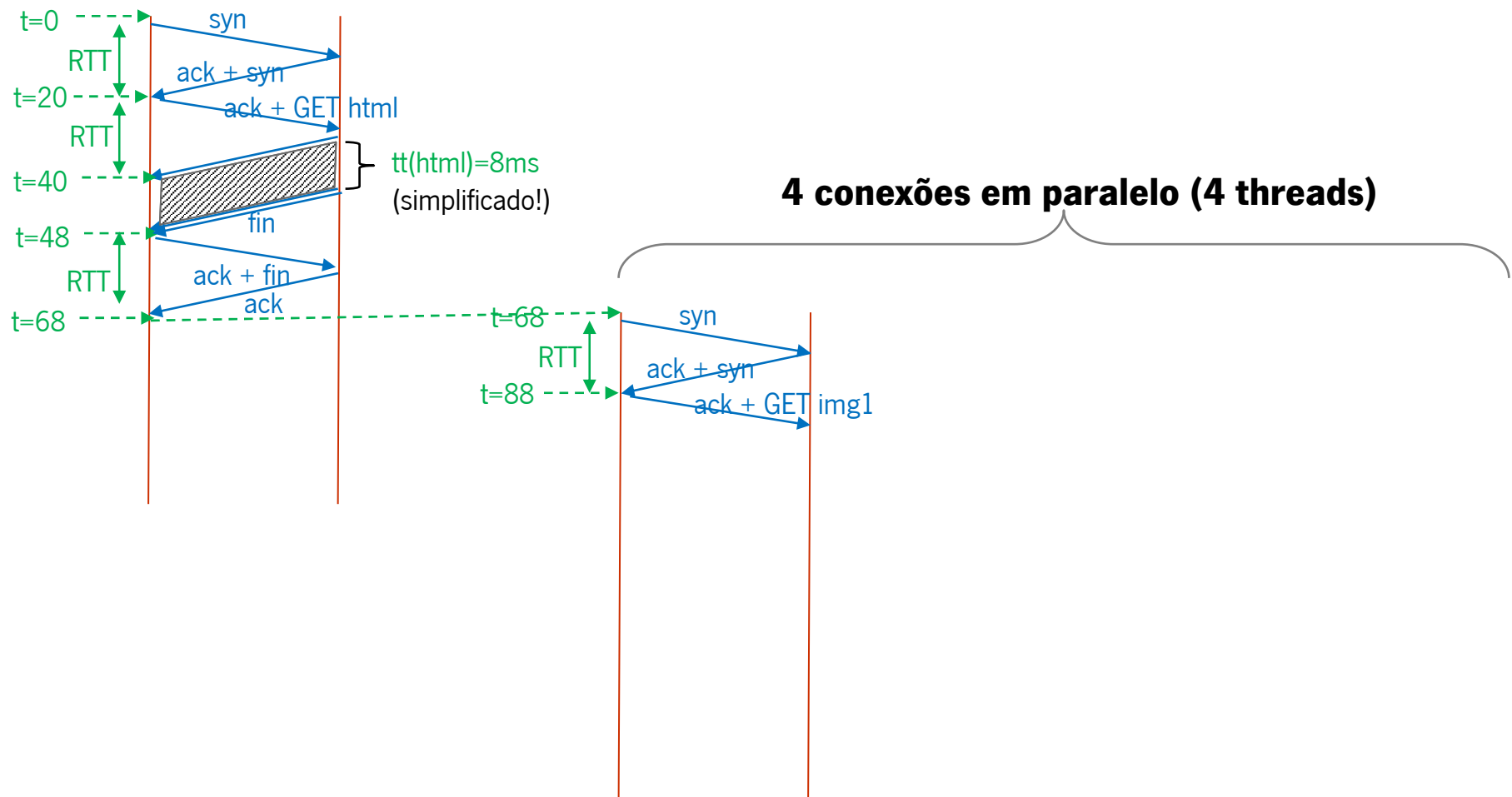
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



Dados: RTT = 20 ms

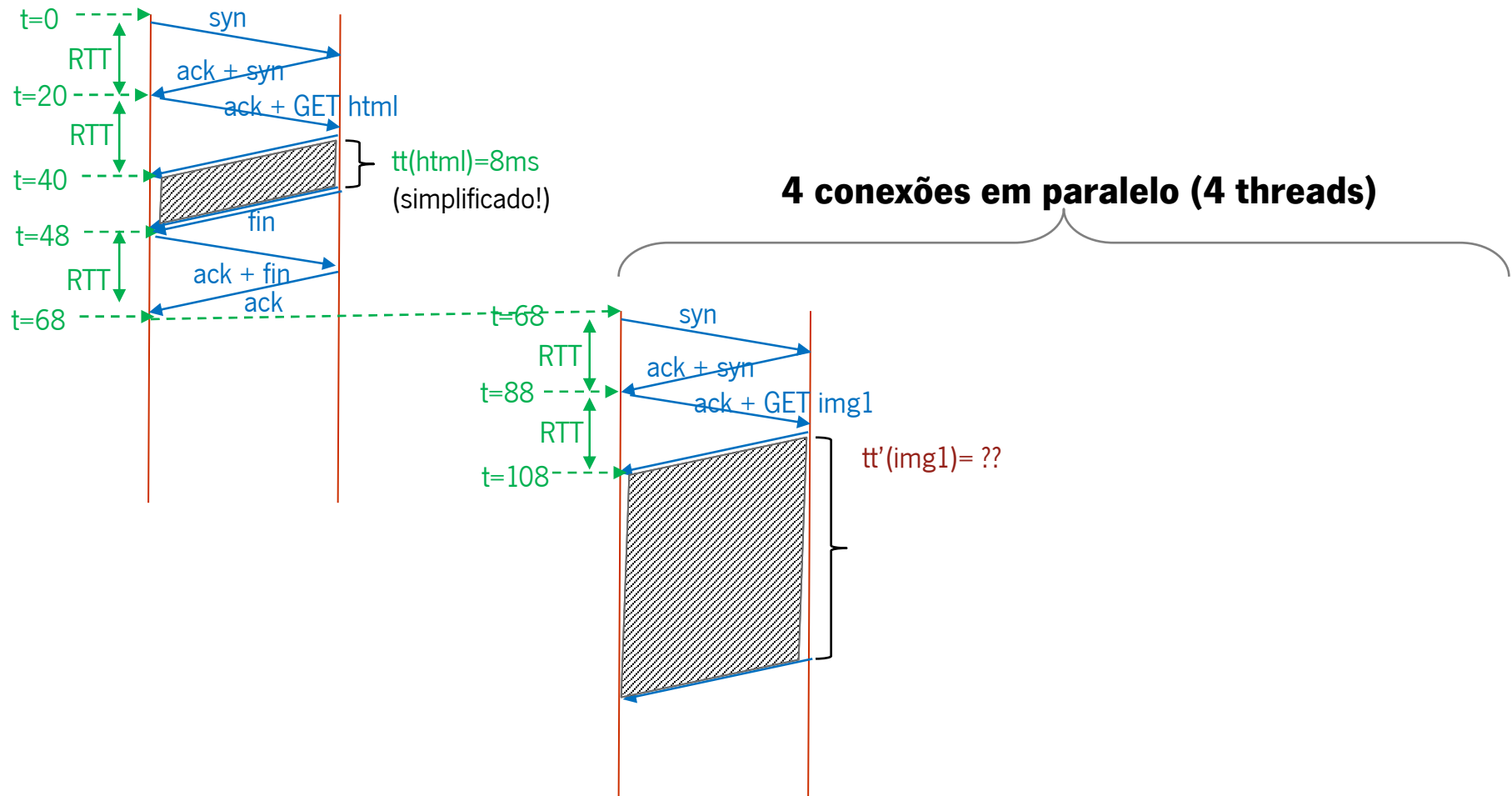
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



Dados: RTT = 20 ms

6 Objectos \rightarrow { html, img1, img2, img3, img4, img5 }

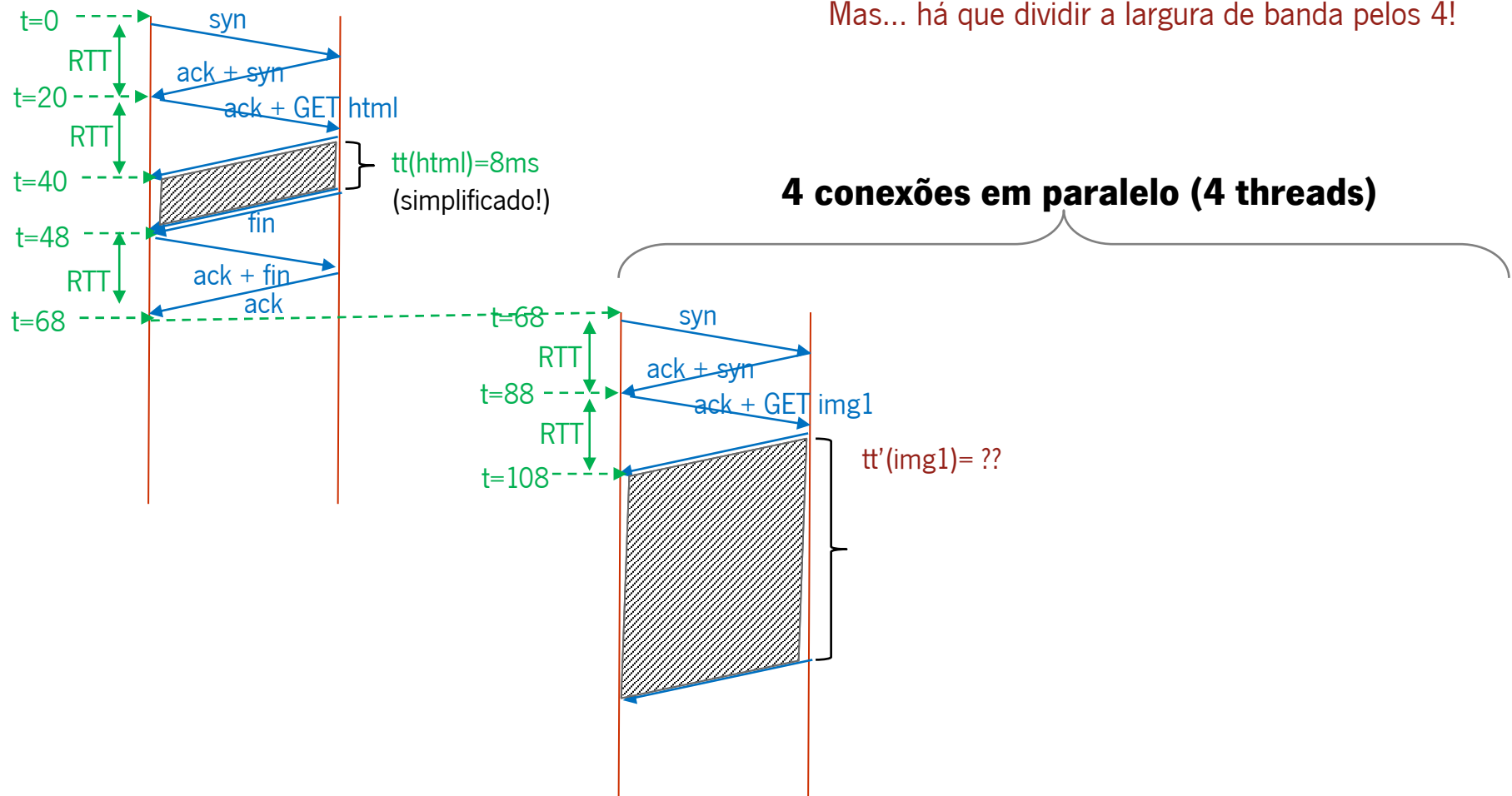
tt (html) = 8 ms

tt(img) = 80 ms



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo

Mas... há que dividir a largura de banda pelos 4!



Dados: $RTT = 20\text{ ms}$

6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

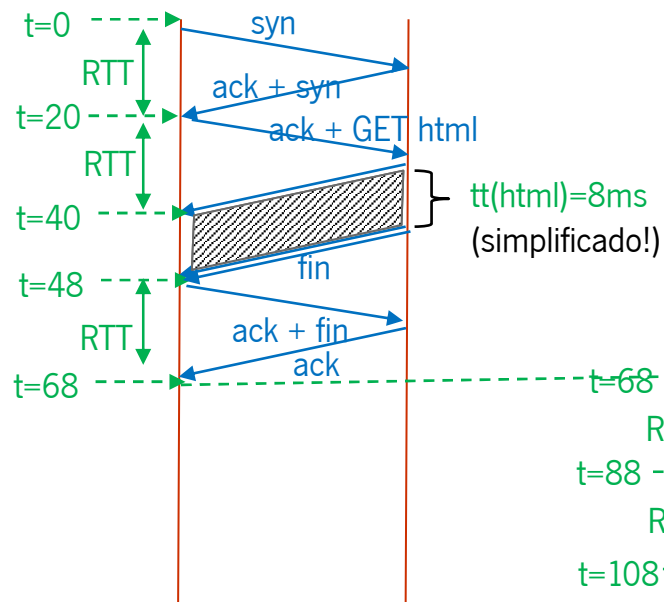
$tt(\text{html}) = 8\text{ ms}$

$tt(\text{img}) = 80\text{ ms}$

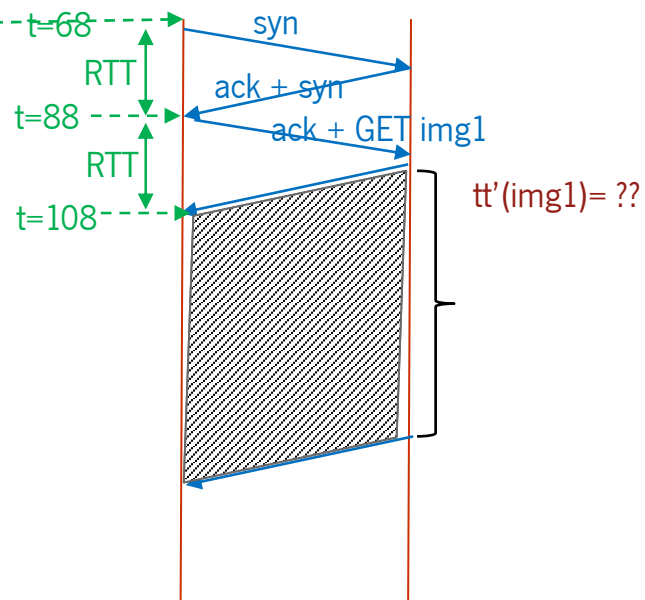


a) HTTP/1.0 Não persistente, com 4 conexões em paralelo

Mas... há que dividir a largura de banda pelos 4!
como $tt = L/R$ e agora temos $R' = R/4$



4 conexões em paralelo (4 threads)



Dados: $RTT = 20\text{ ms}$

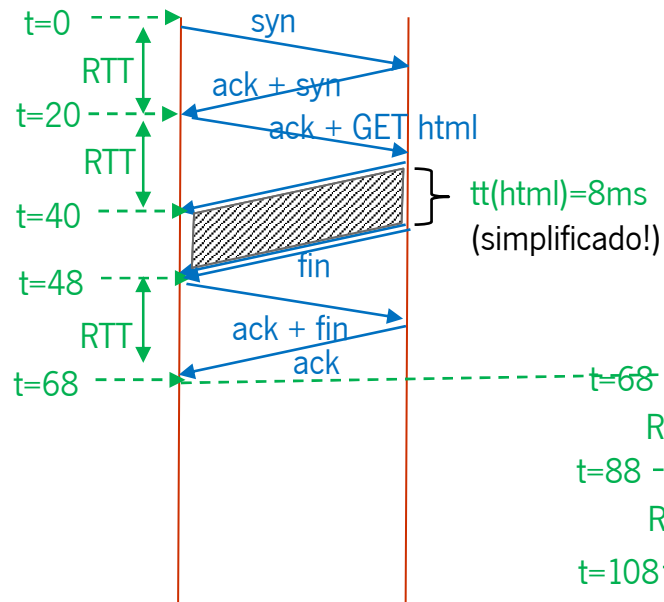
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8\text{ ms}$

$tt(\text{img}) = 80\text{ ms}$

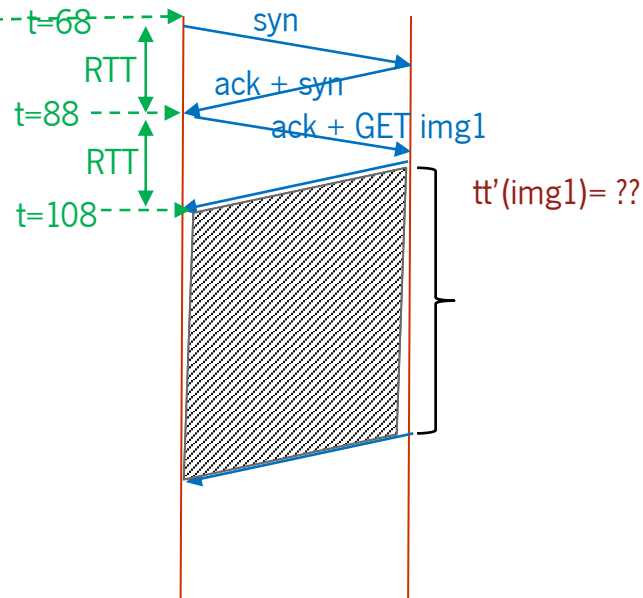


a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



Mas... há que dividir a largura de banda pelos 4!
como $tt = L/R$ e agora temos $R' = R/4$
então $tt' = L/R' = L/(R/4) = 4 * L/R \Rightarrow tt' = 4 tt$

4 conexões em paralelo (4 threads)



Dados: RTT = 20 ms

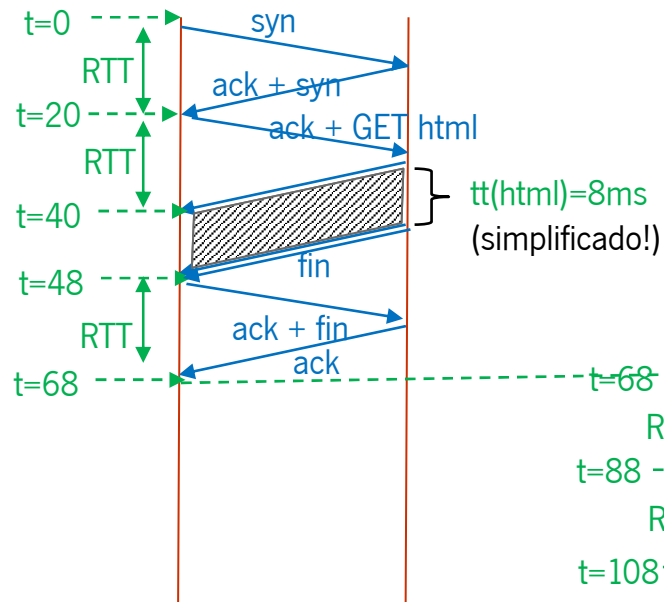
6 Objectos \rightarrow { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo

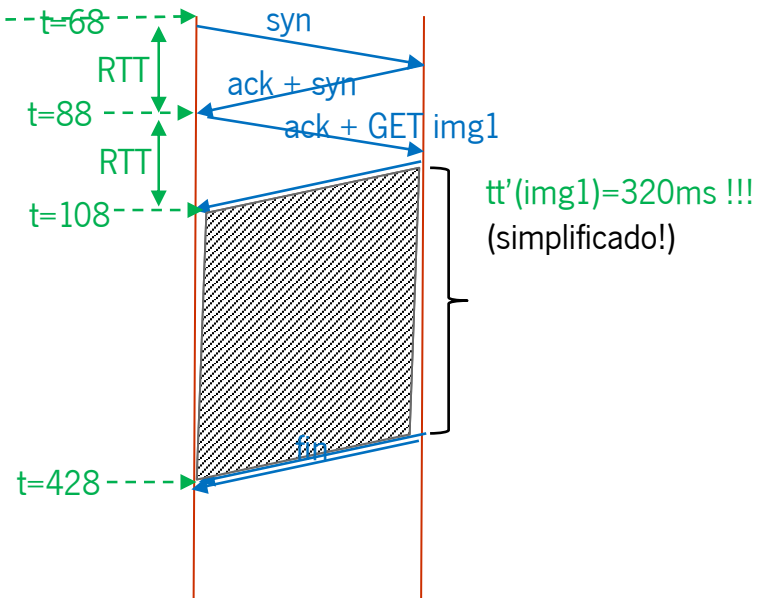


Mas... há que dividir a largura de banda pelos 4!

como $tt = L/R$ e agora temos $R' = R/4$

então $tt' = L/R' = L/(R/4) = 4 * L/R \Rightarrow tt' = 4 tt$

4 conexões em paralelo (4 threads)



Dados: $RTT = 20\text{ ms}$

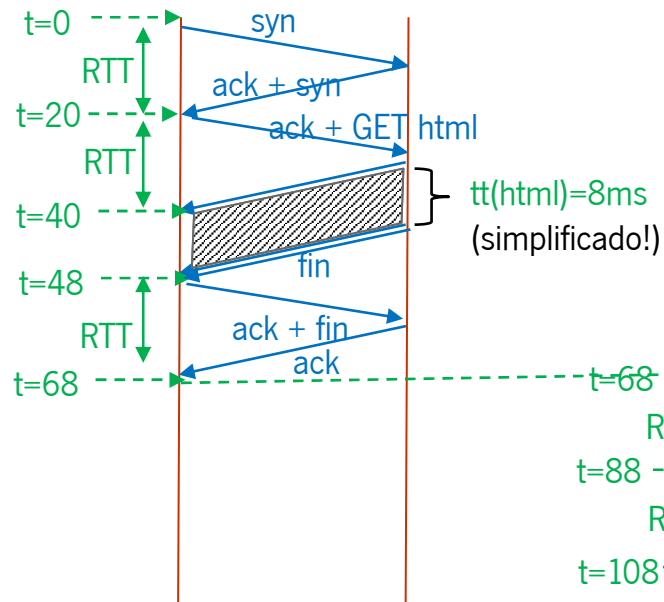
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8\text{ ms}$

$tt(\text{img}) = 80\text{ ms}$

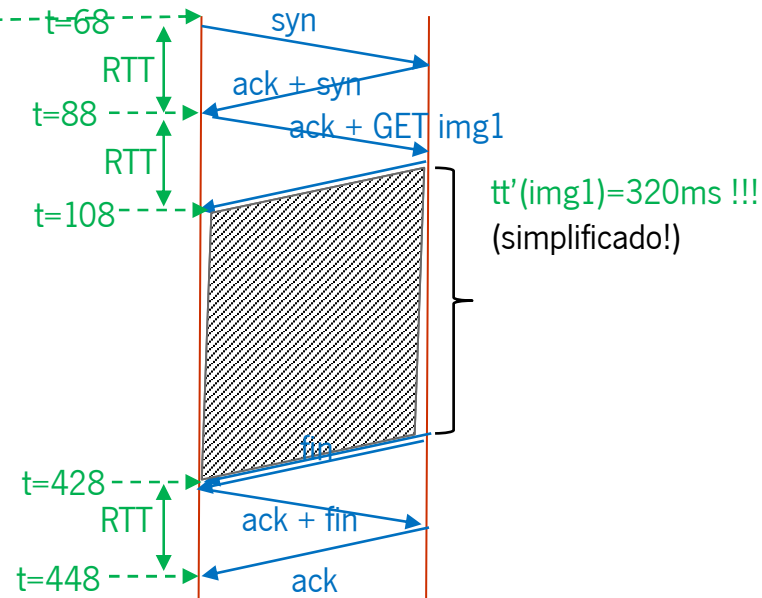


a) HTTP/1.0 Não persistente, com 4 conexões em paralelo



Mas... há que dividir a largura de banda pelos 4!
como $tt = L/R$ e agora temos $R' = R/4$
então $tt' = L/R' = L/(R/4) = 4 * L/R \Rightarrow tt' = 4 tt$

4 conexões em paralelo (4 threads)



Dados: $RTT = 20\text{ ms}$

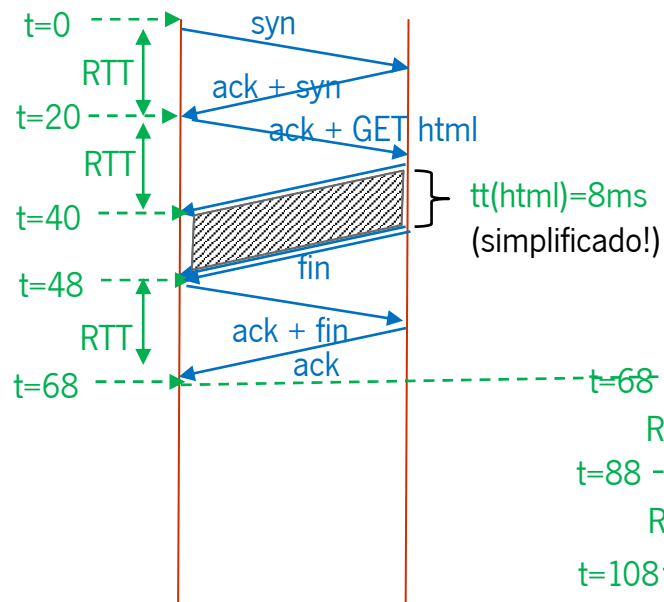
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8\text{ ms}$

$tt(\text{img}) = 80\text{ ms}$



a) HTTP/1.0 Não persistente, com 4 conexões em paralelo

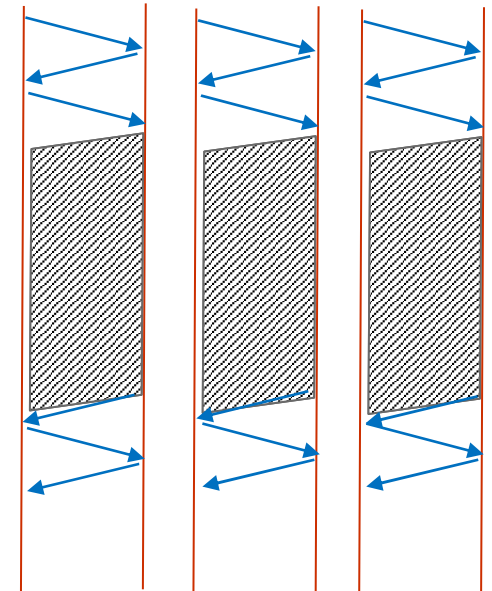
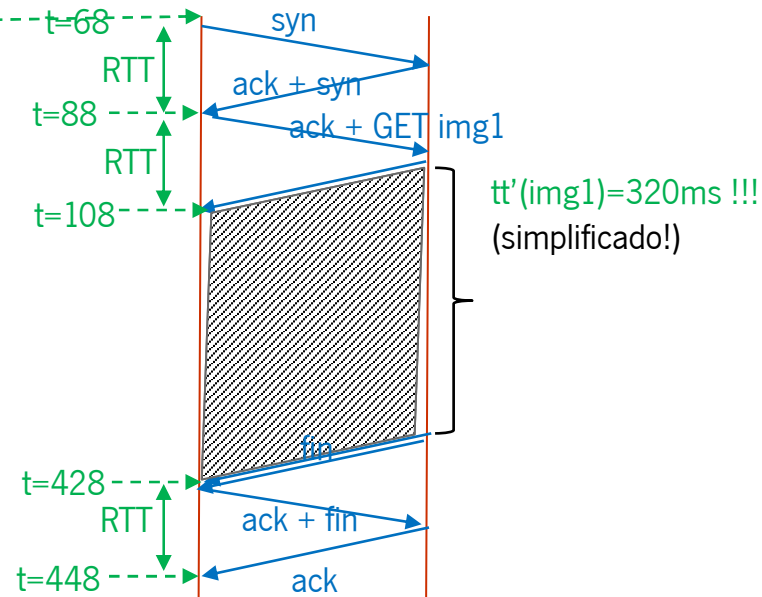


Mas... há que dividir a largura de banda pelos 4!

como $tt = L/R$ e agora temos $R' = R/4$

então $tt' = L/R' = L/(R/4) = 4 * L/R \Rightarrow tt' = 4 tt$

4 conexões em paralelo (4 threads)



... ainda falta uma imagem **img5** \rightarrow

Dados: $RTT = 20 \text{ ms}$

6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



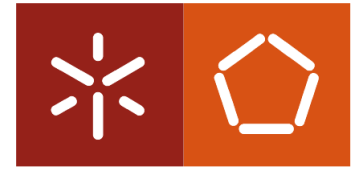
a) ... continuação: descarregar a ultima imagem numa nova conexão

Dados: $RTT = 20 \text{ ms}$

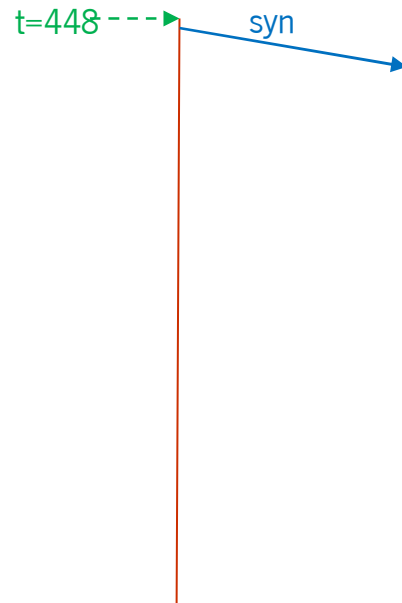
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



a) ... continuação: descarregar a ultima imagem numa nova conexão



Dados: $RTT = 20\text{ ms}$

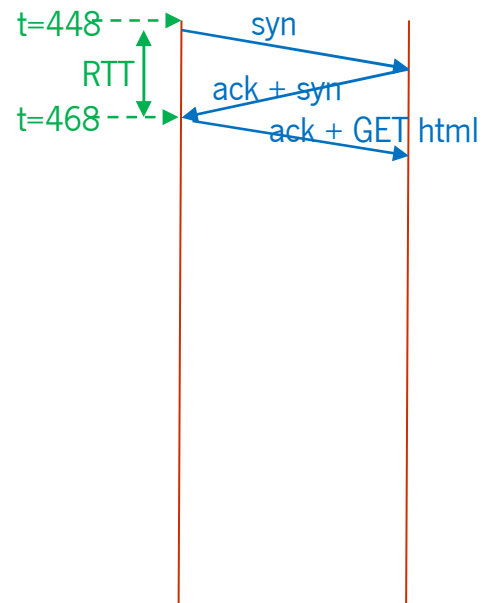
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8\text{ ms}$

$tt(\text{img}) = 80\text{ ms}$



a) ... continuação: descarregar a ultima imagem numa nova conexão



Dados: $RTT = 20\text{ ms}$

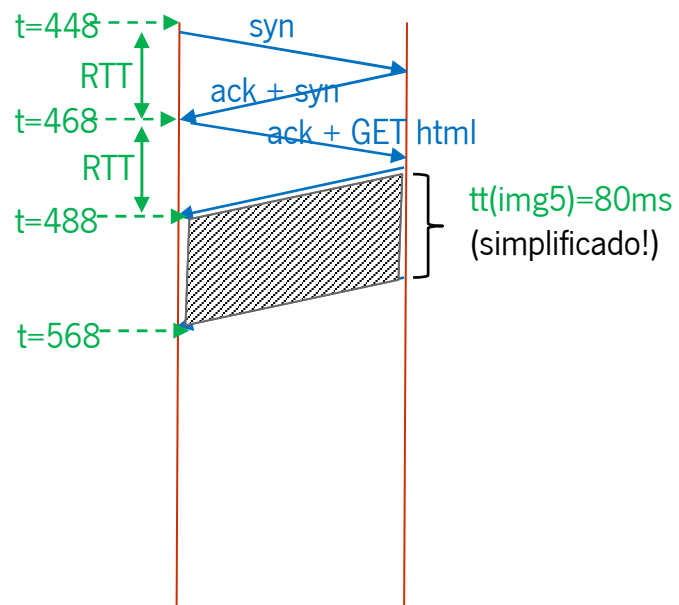
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8\text{ ms}$

$tt(\text{img}) = 80\text{ ms}$



a) ... continuação: descarregar a ultima imagem numa nova conexão



Dados: $RTT = 20\text{ ms}$

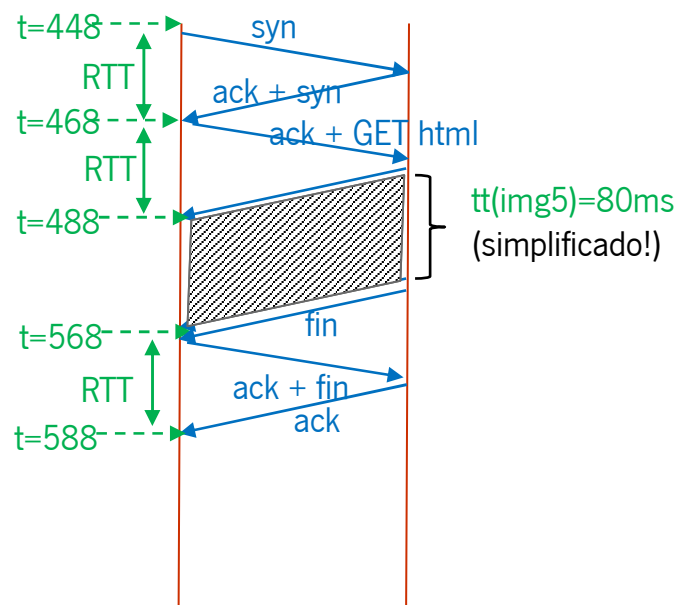
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8\text{ ms}$

$tt(\text{img}) = 80\text{ ms}$



a) ... continuação: descarregar a ultima imagem numa nova conexão



Dados: RTT = 20 ms

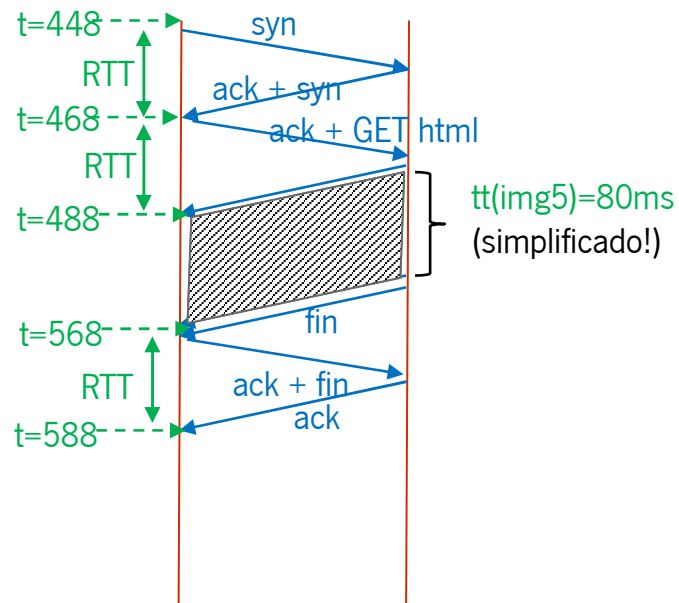
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



a) ... continuação: descarregar a ultima imagem numa nova conexão



T.a.total = 568 ms (ou 588 ms contando com fecho conexão)

Dados: $RTT = 20 \text{ ms}$

6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo

Dados: $RTT = 20 \text{ ms}$

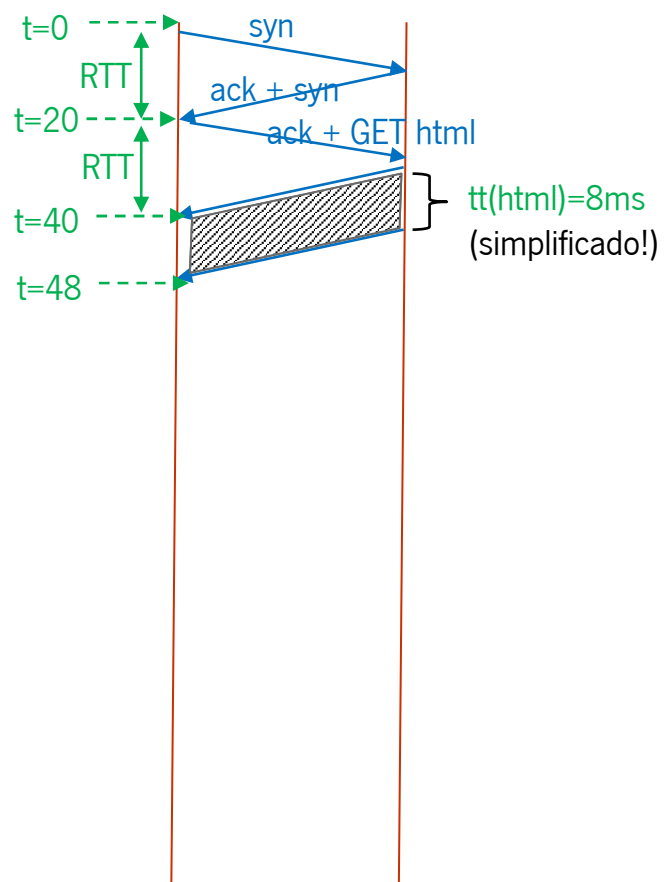
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo



Dados: $RTT = 20\text{ ms}$

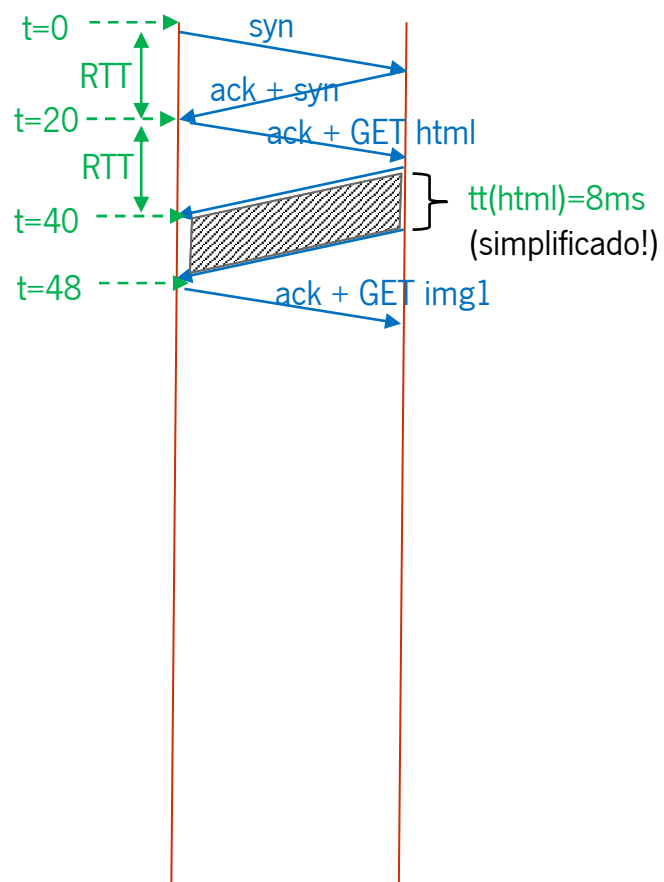
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8\text{ ms}$

$tt(\text{img}) = 80\text{ ms}$



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo



Dados: RTT = 20 ms

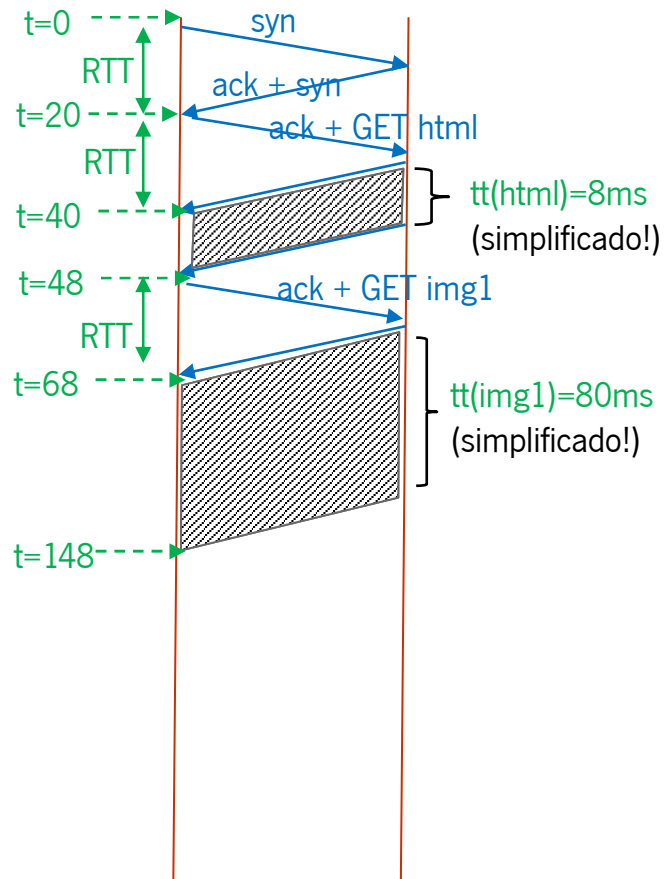
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt(html) = 8 ms

tt(img) = 80 ms



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo



Dados: RTT = 20 ms

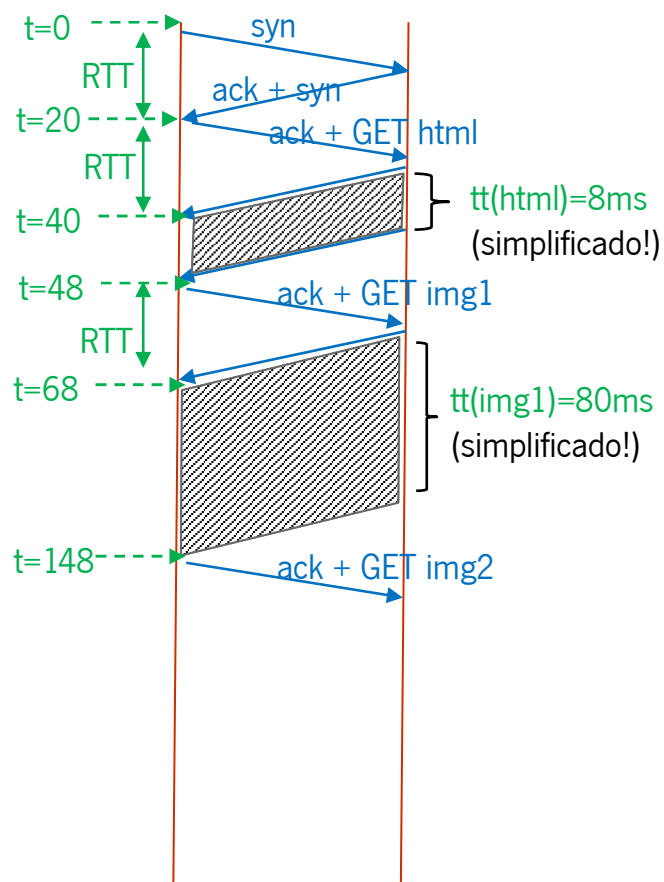
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo



Dados: RTT = 20 ms

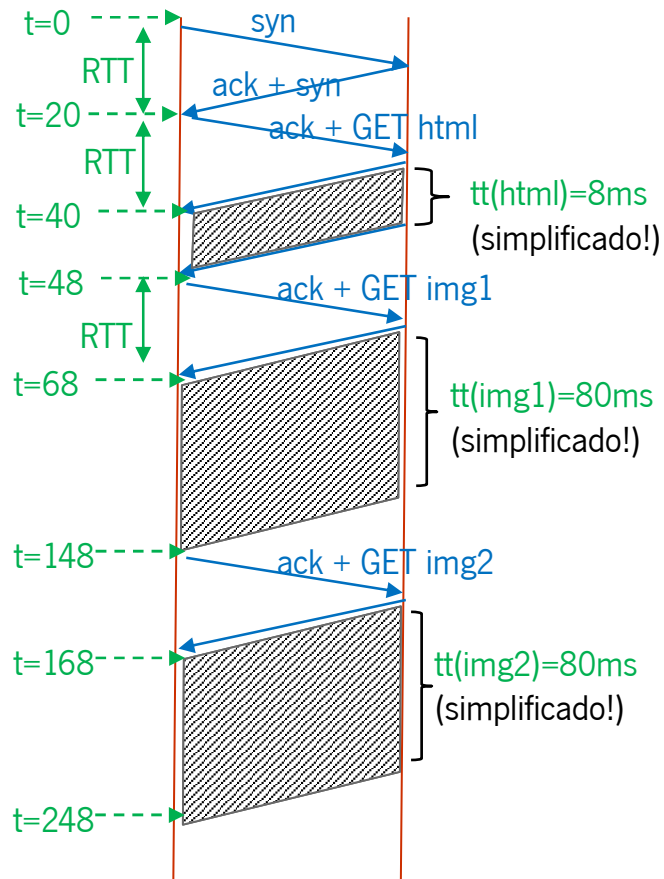
6 Objectos \rightarrow { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo



Dados: RTT = 20 ms

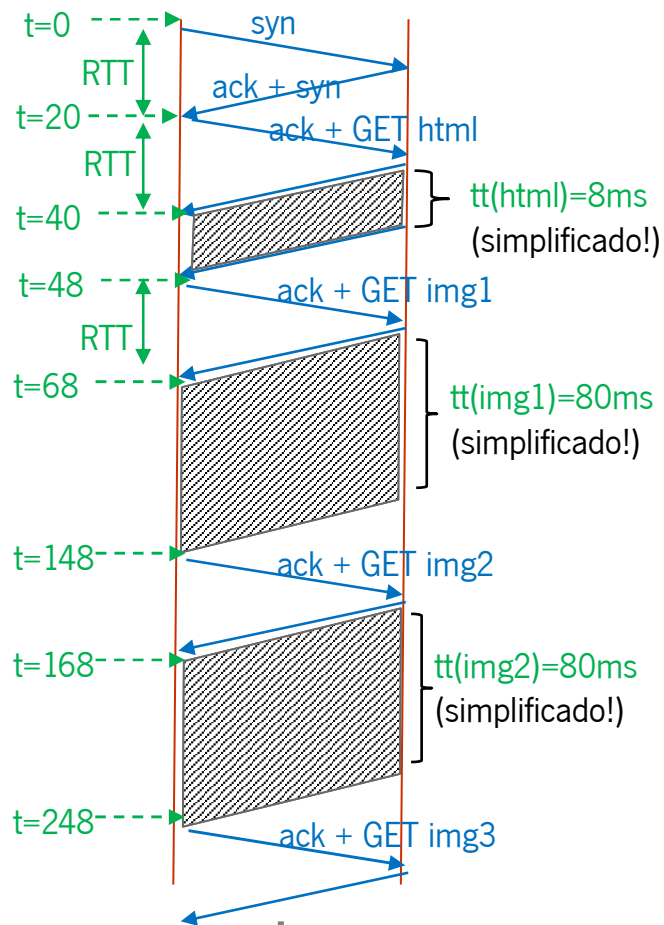
6 Objectos \rightarrow { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo



Dados: RTT = 20 ms

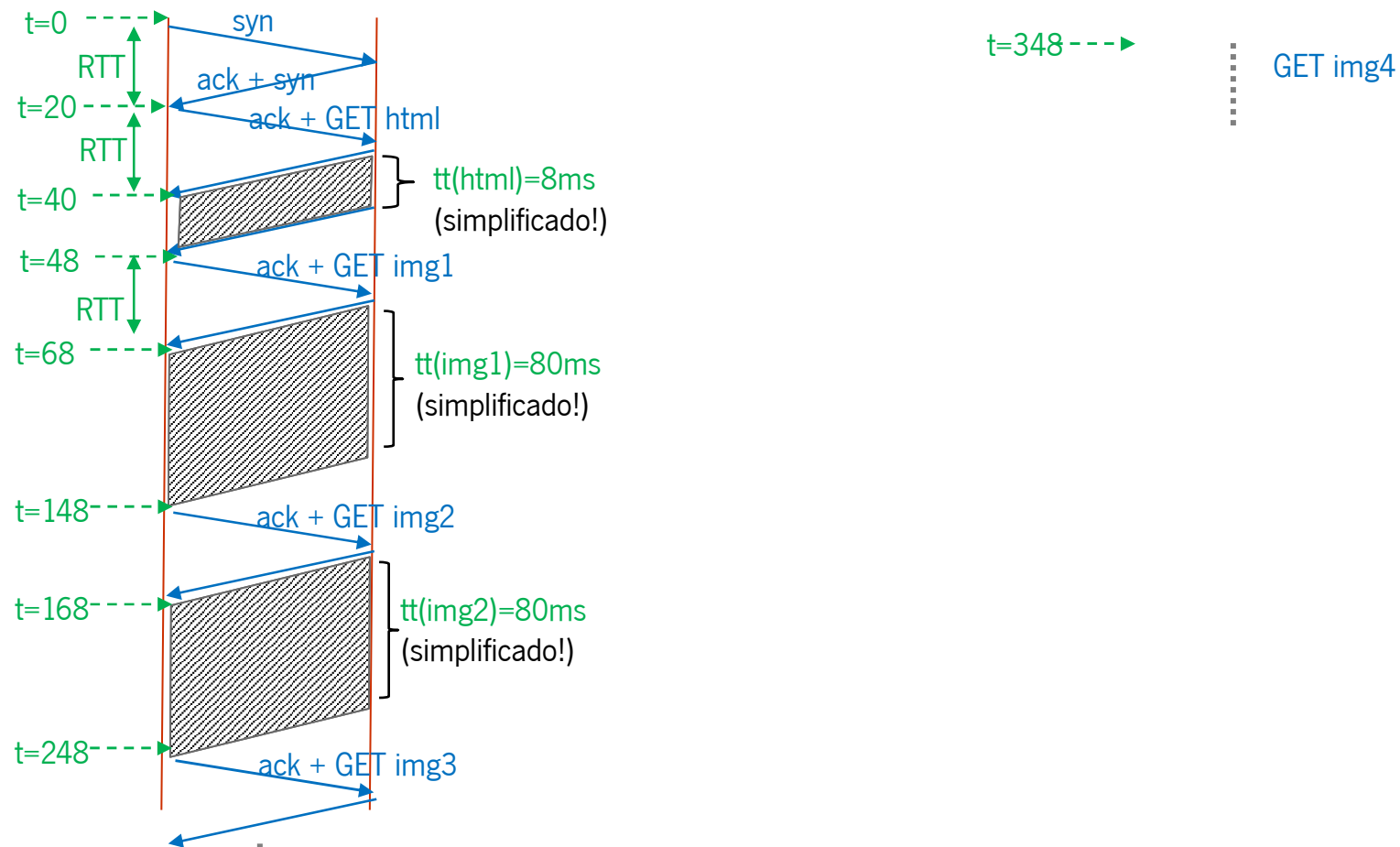
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo



Dados: RTT = 20 ms

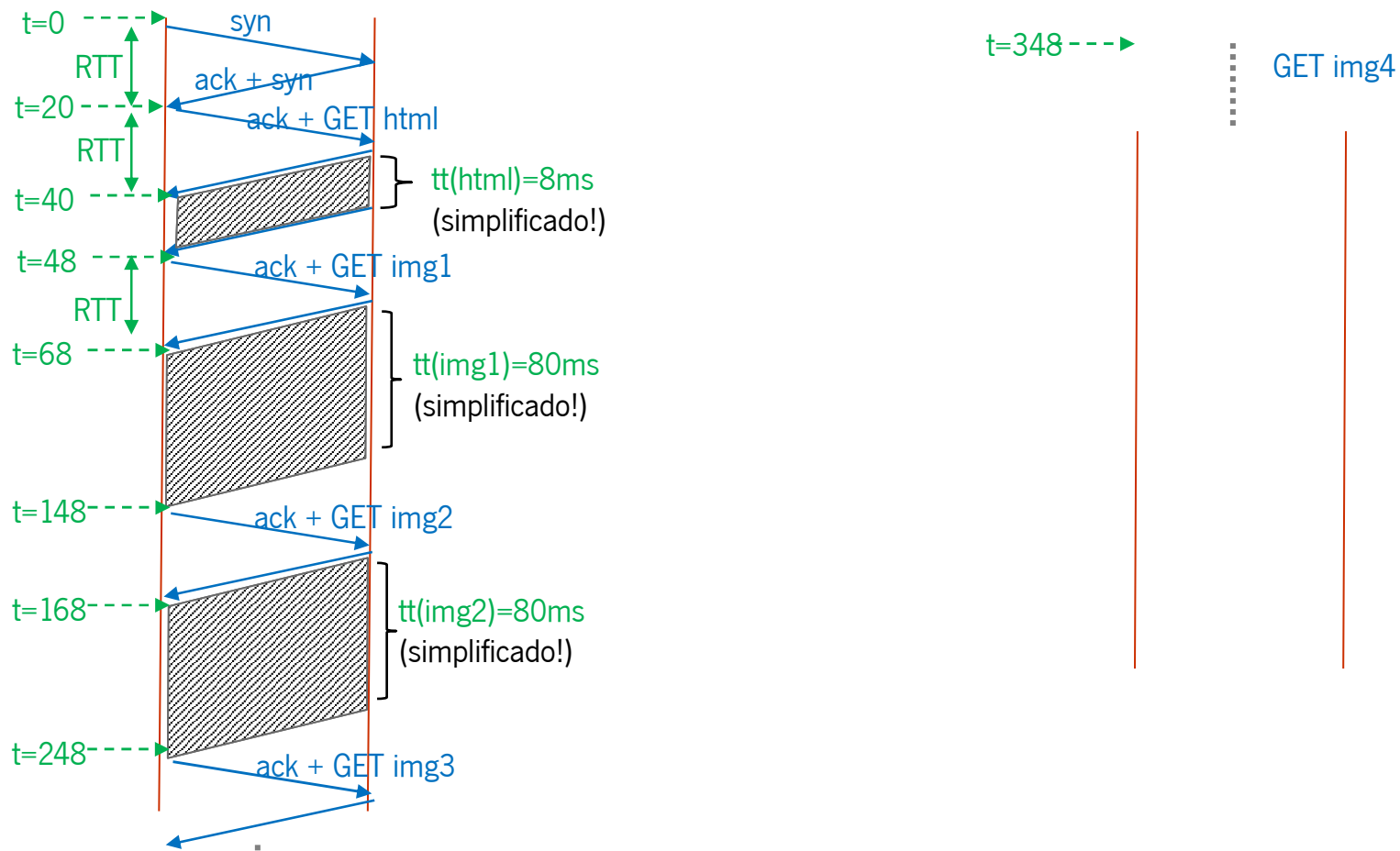
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo



Dados: RTT = 20 ms

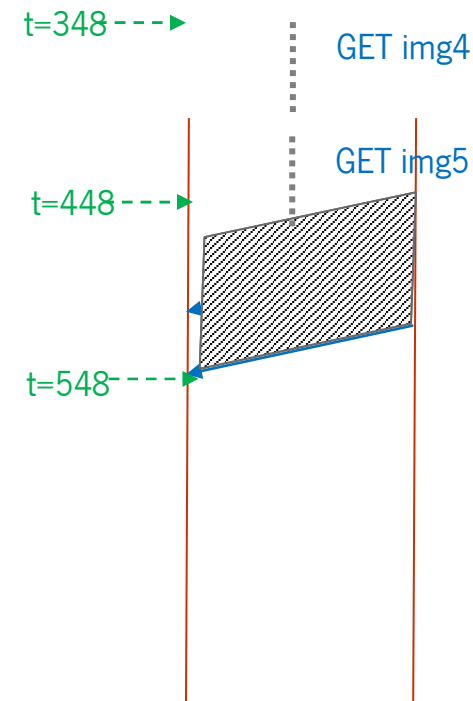
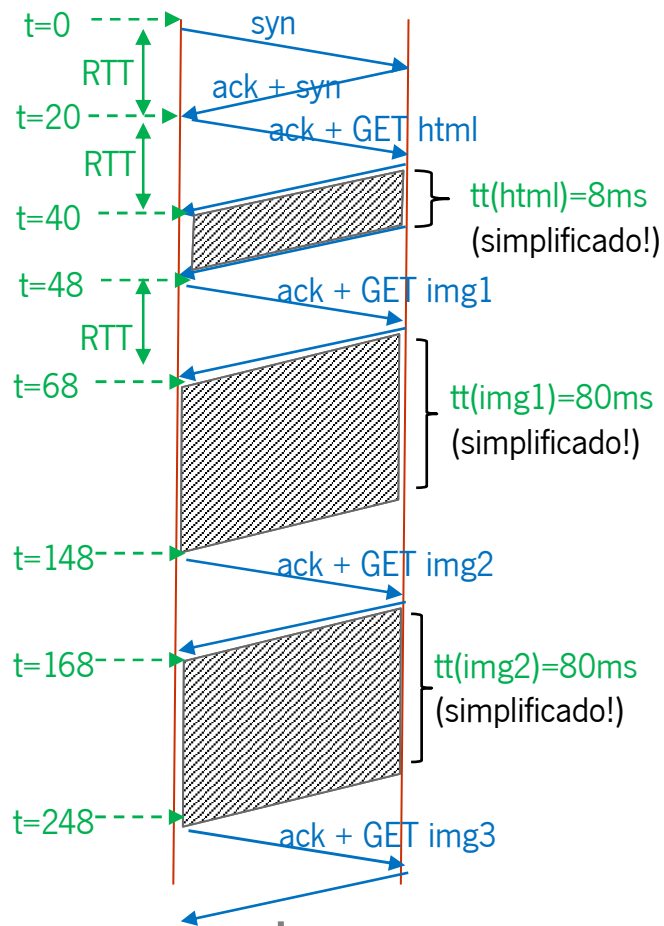
6 Objectos \rightarrow { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo



Dados: RTT = 20 ms

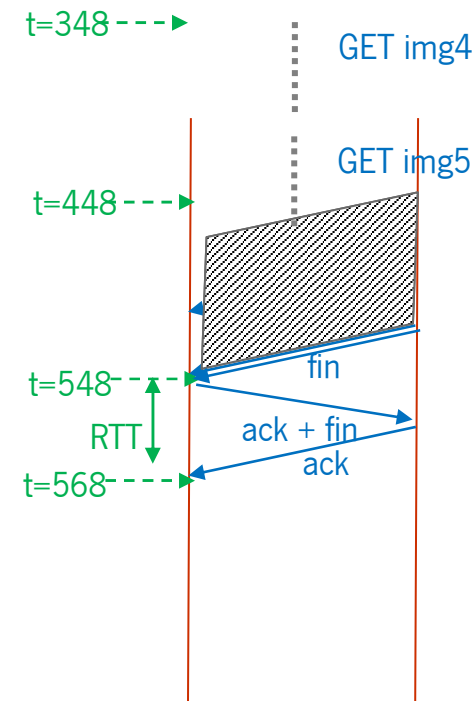
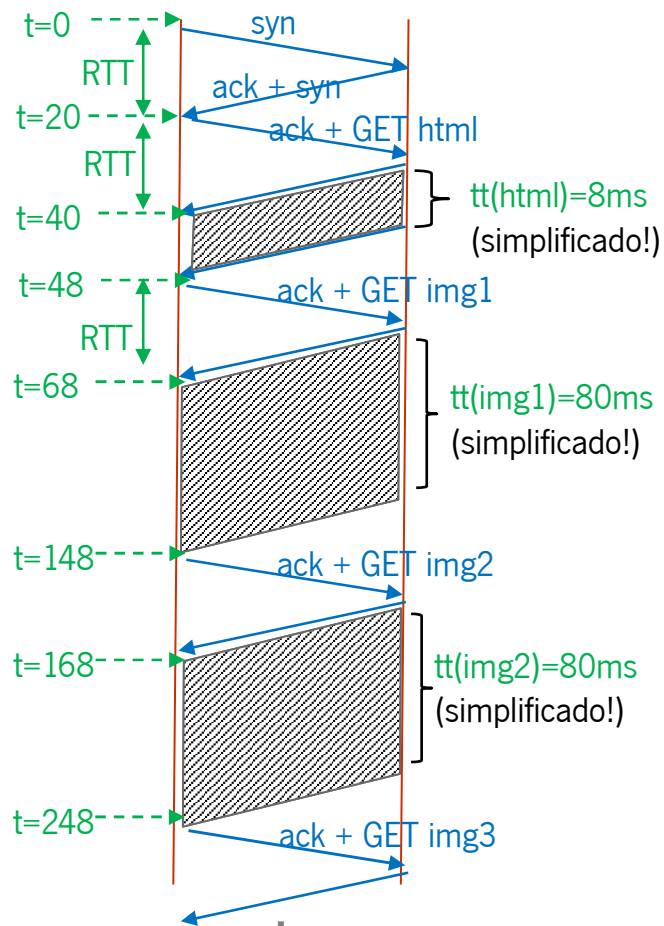
6 Objectos \rightarrow { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo

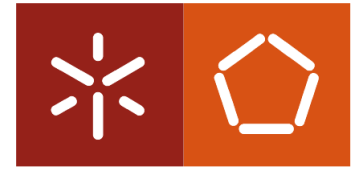


Dados: RTT = 20 ms

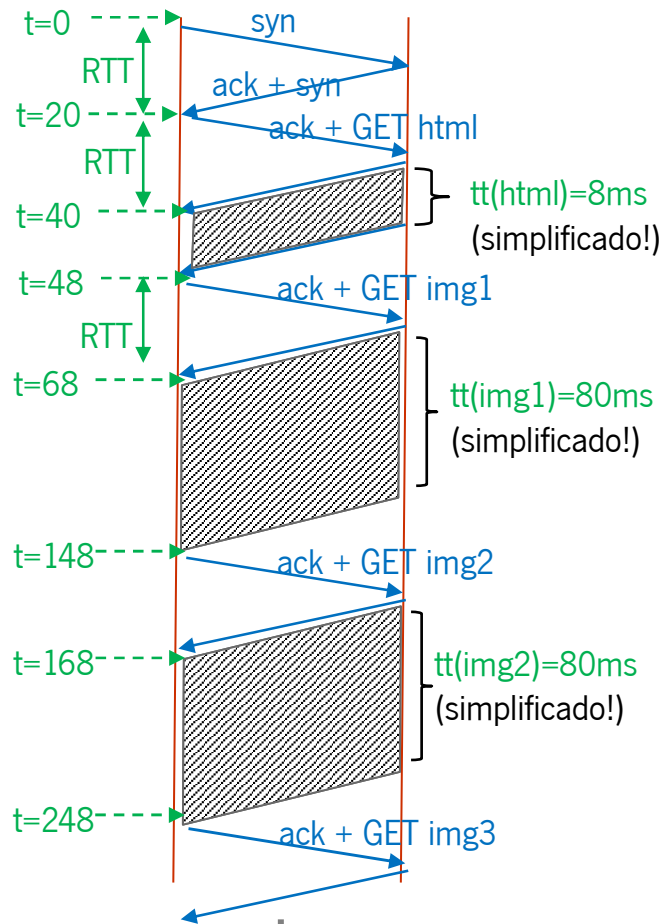
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



b) HTTP/1.1 persistente, sem pipeline, sem conexões em paralelo



T.b.total = 548 ms (ou 568 ms contando com fecho conexão)

Dados: $RTT = 20 \text{ ms}$

6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



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

Dados: $RTT = 20 \text{ ms}$

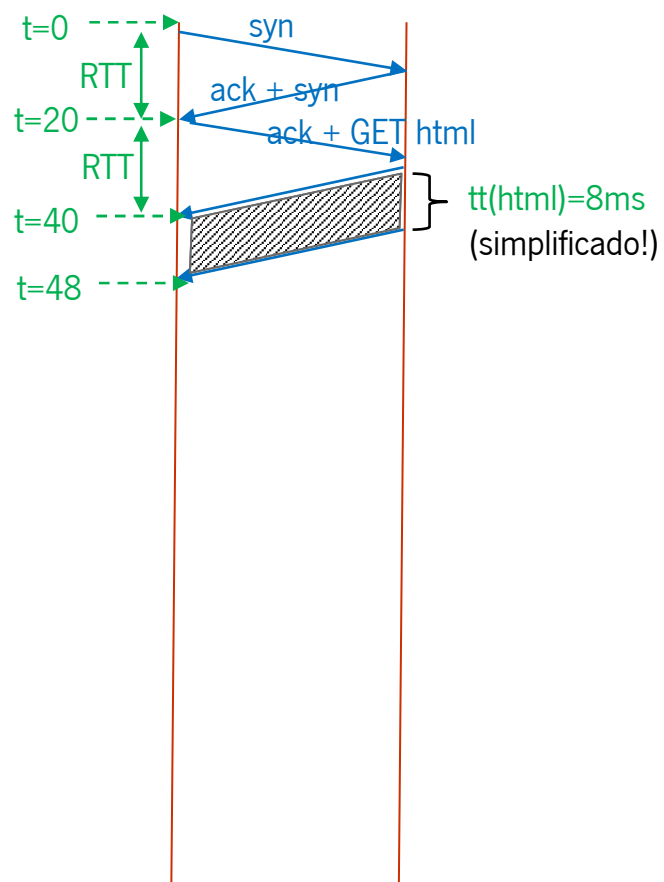
6 Objectos $\rightarrow \{ \text{html}, \text{img1}, \text{img2}, \text{img3}, \text{img4}, \text{img5} \}$

$tt(\text{html}) = 8 \text{ ms}$

$tt(\text{img}) = 80 \text{ ms}$



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



Dados: RTT = 20 ms

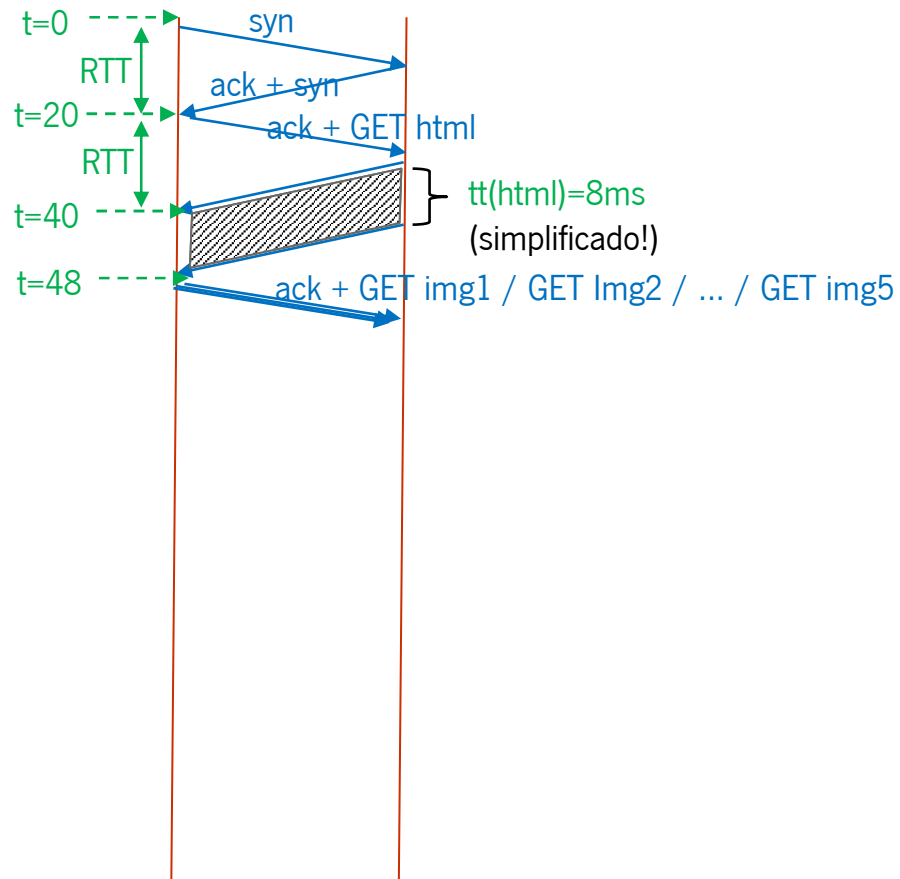
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



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



Dados: RTT = 20 ms

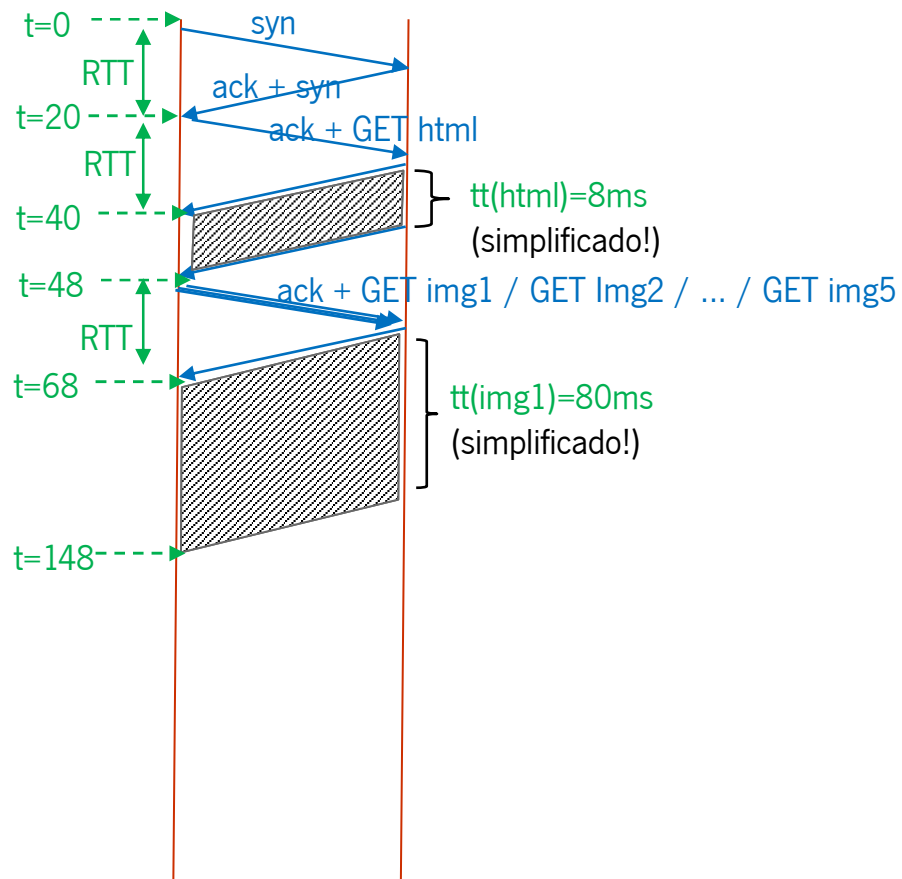
6 Objectos \rightarrow { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



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



Dados: RTT = 20 ms

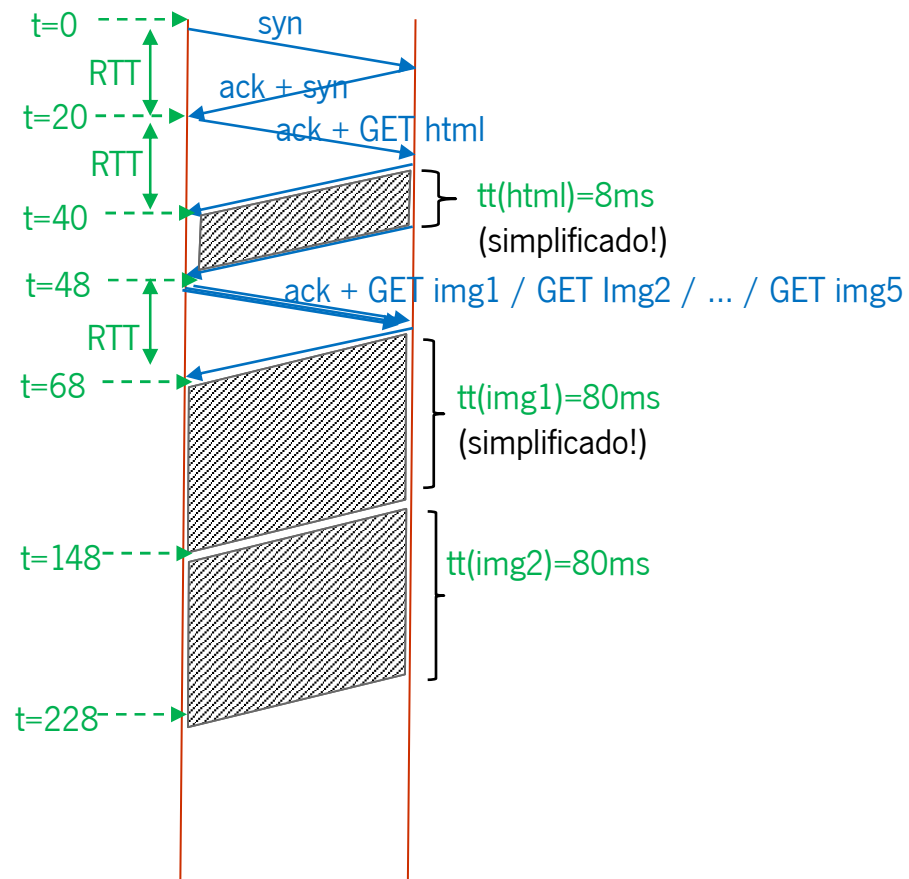
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



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



Dados: RTT = 20 ms

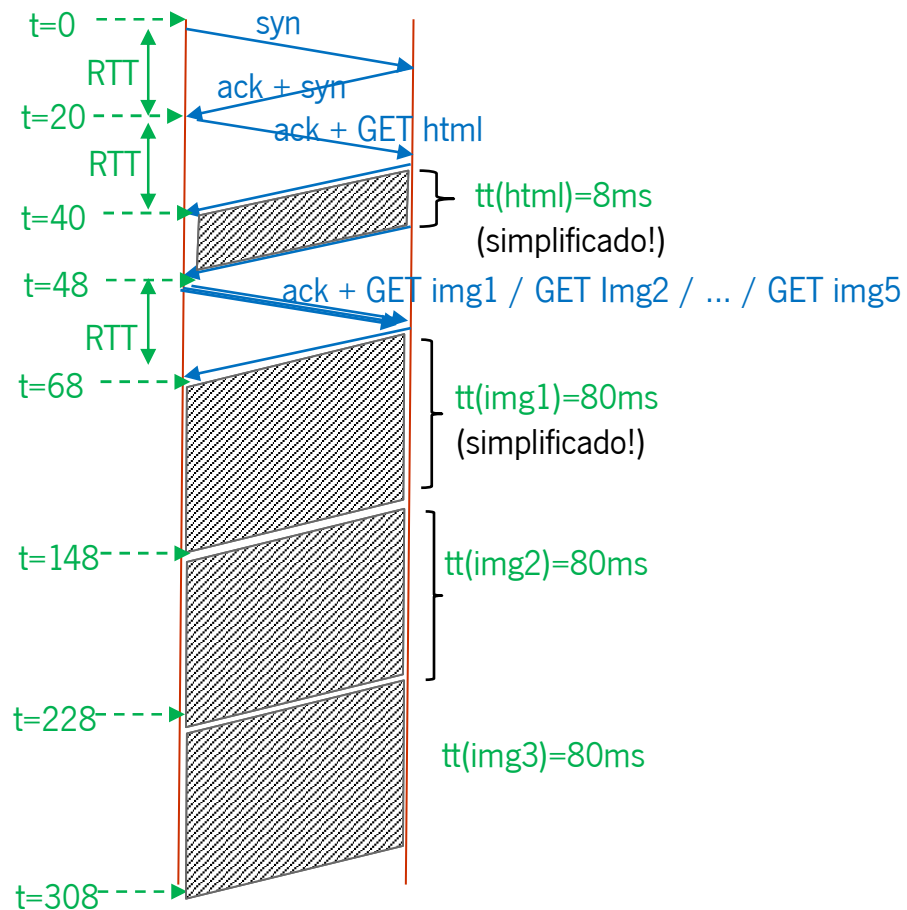
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



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



Dados: RTT = 20 ms

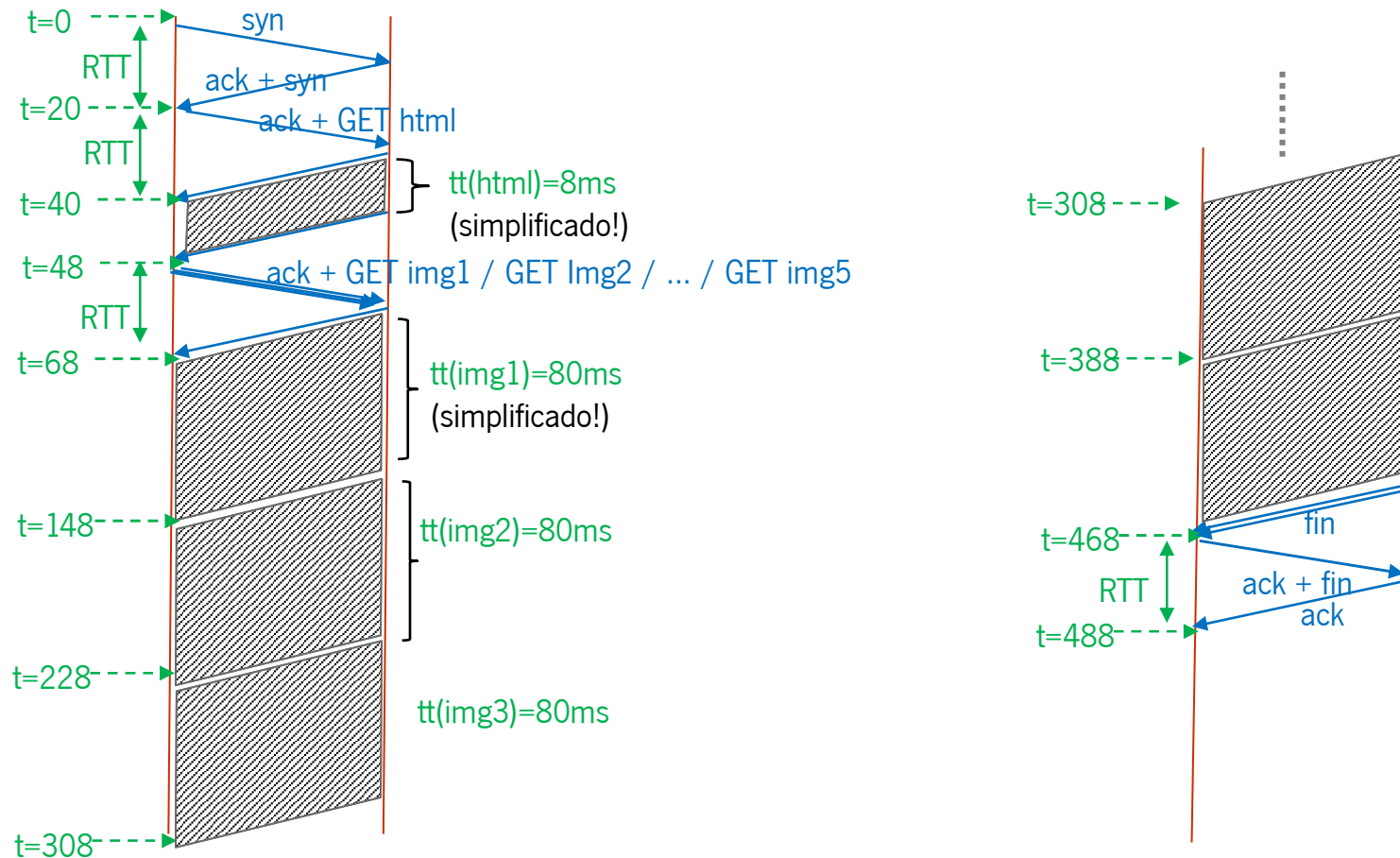
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



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



Dados: RTT = 20 ms

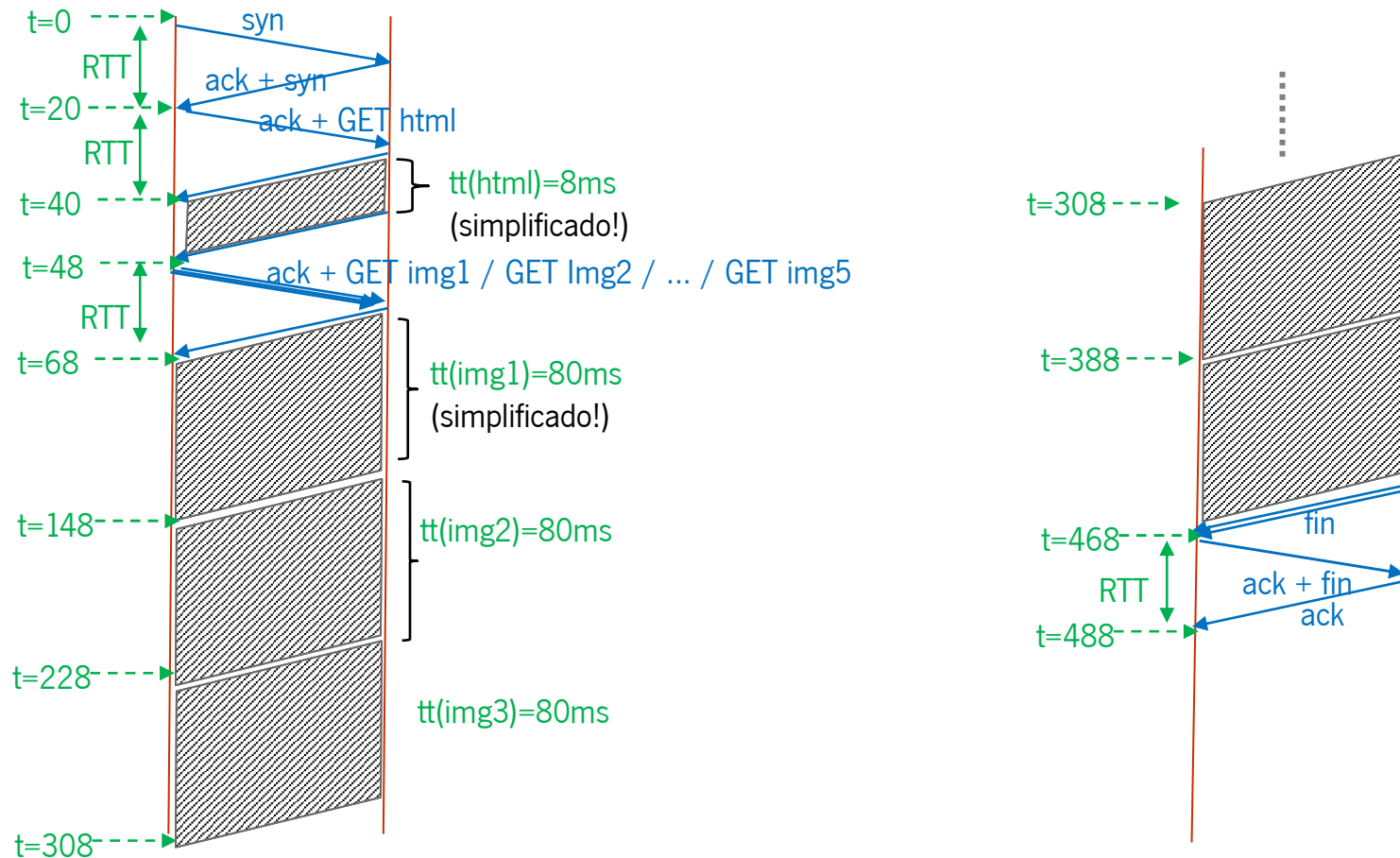
6 Objectos → { html, img1, img2, img3, img4, img5 }

tt (html) = 8 ms

tt(img) = 80 ms



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



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

HTTP

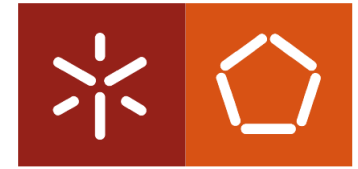
Exercícios



2. Pretende-se estimar o atraso na receção de um documento web usando o protocolo HTTP. Sabemos que o atraso de ida-e-volta (RTT) entre cliente e servidor é de 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 devem ser ilustradas com diagramas temporais.
- a) Se um documento consistir num único objeto base com 2048 *bytes*, a memória de receção TCP for ilimitada e o TCP utilizar os mecanismos de SS e CA com 4 segmentos de valor de *threshold*, estime 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.
 - b) Assuma que outro documento web contém 4 imagens que são referenciadas no objeto base. Cada imagem contém 1024 *bytes* e a versão de HTTP usada é não-persistente suportando um máximo de 2 conexões paralelas. Determine uma estimativa do tempo decorrido até à receção do documento, considerando que a largura de banda disponível é repartida equitativamente entre conexões paralelas.
 - c) Determine o tempo decorrido no contexto da alínea anterior, mas tendo agora em consideração a utilização da versão 1.1 do protocolo HTTP, com e sem possibilidade de pedidos em sequência.

HTTP

Informação de estado – *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*

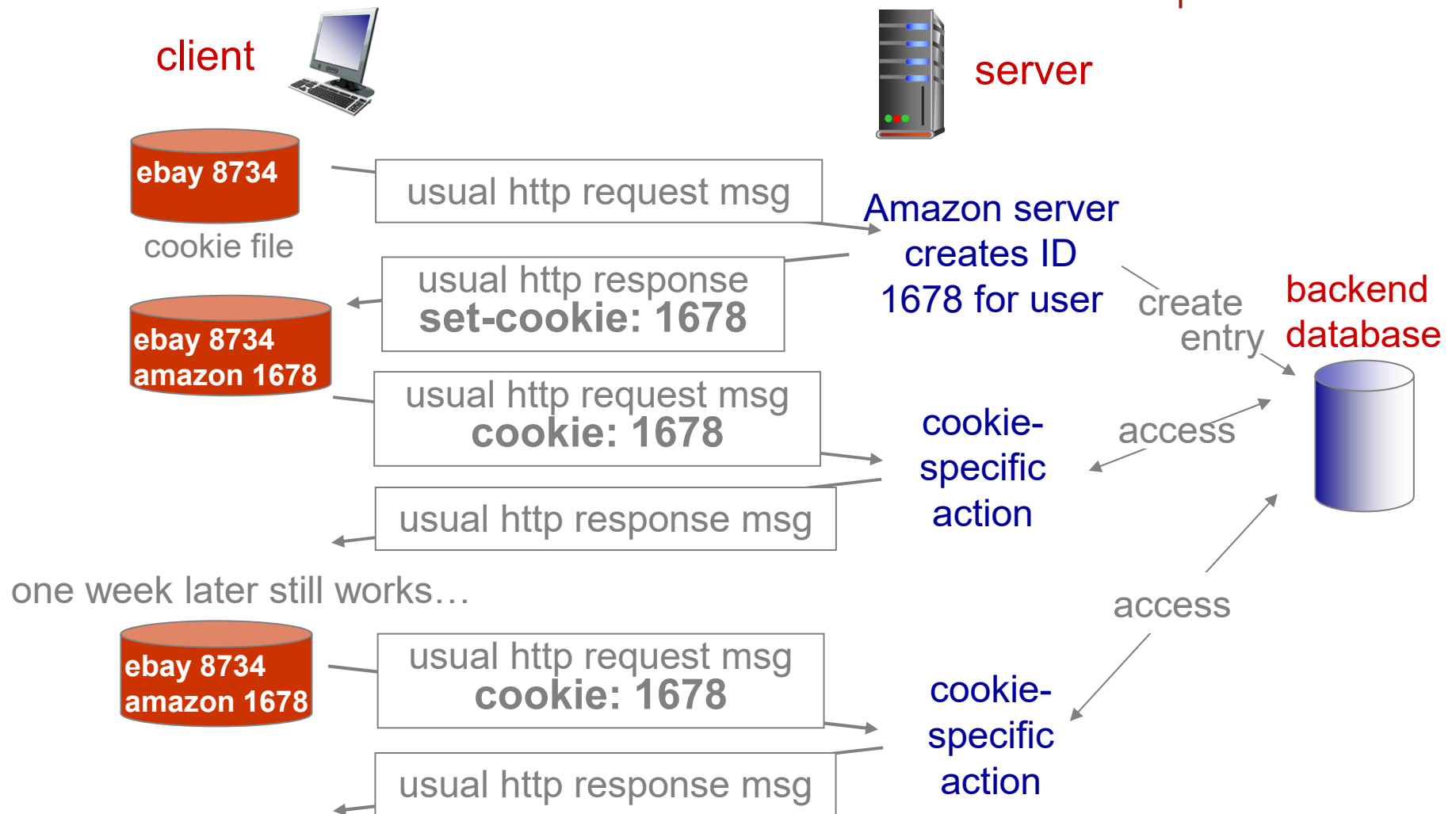
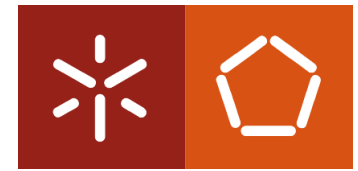
Exemplo:

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.

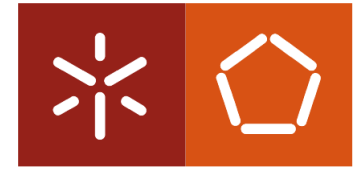
HTTP

Informação de estado – *Cookies*



HTTP

Informação de estado – *Cookies*



efeitos colaterais

O que os cookies permitem:

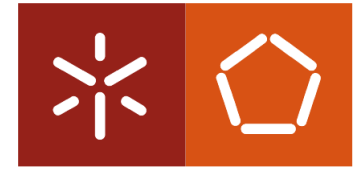
- autenticar/autorizar
- implementar cabaz de compras
- fazer sugestões ao utilizador
- manter informação da sessão por cada utilizador (ex: *Web e-mail*)
- etc...

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?

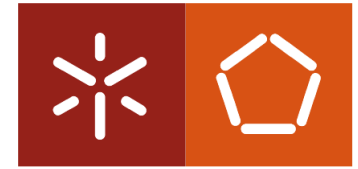
- reduz o tempo de resposta para os pedidos dos clientes
- reduz o tráfego nos *links* de acesso ao exterior (os mais problemáticos para a instituição).
- a Internet está povoada de *caches* e que permitem que fornecedores de conteúdos mais “pobres” disponibilizem efetivamente os seus conteúdos (um pouco 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.)

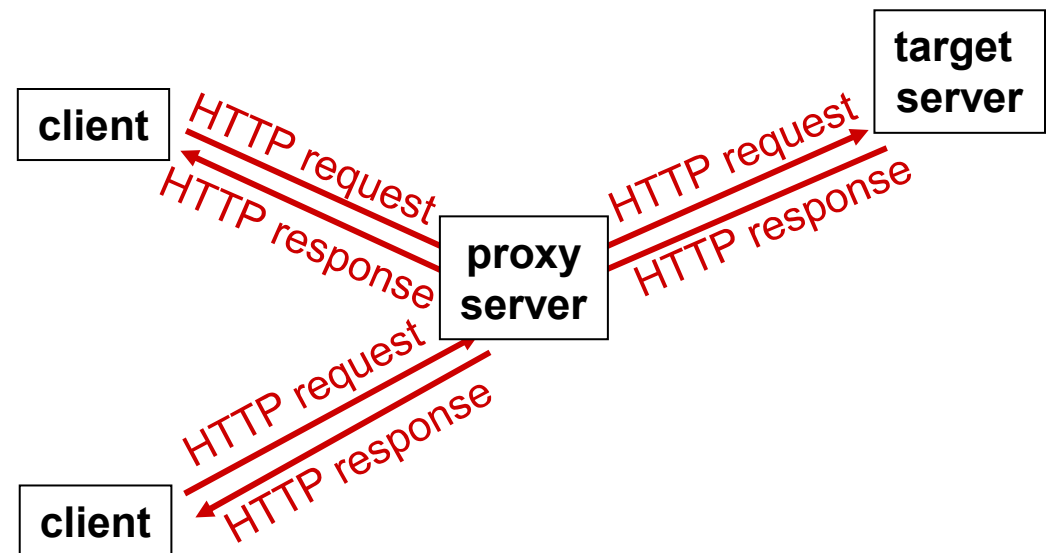
HTTP

Servidores Proxy – *Web Cache*



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 de *Web Caching*

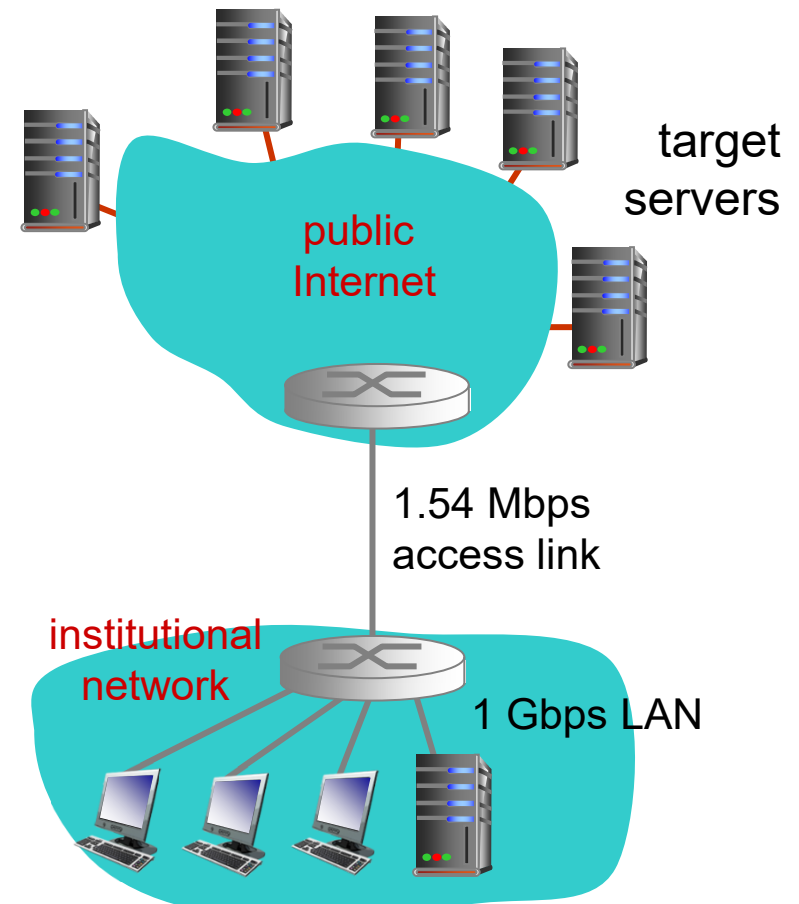


Pressupostos:

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

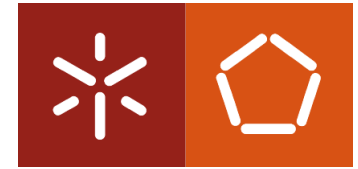
Consequências:

- Utilização da LAN = 15%
 $(15 \text{ pedidos/seg}) \times (100\text{Kbits/pedido}) / (10\text{Mbps})$
- Utilização do *Link* de acesso = 99%
 $(15 \text{ pedidos/seg}) \times (100\text{Kbits/pedido}) / (1.54\text{Mbps})$
- Total do atraso =
= atraso Internet + atraso acesso + atraso LAN
= 2 segundos + minutes + milissegundos



HTTP

Exemplo de *Web Caching*

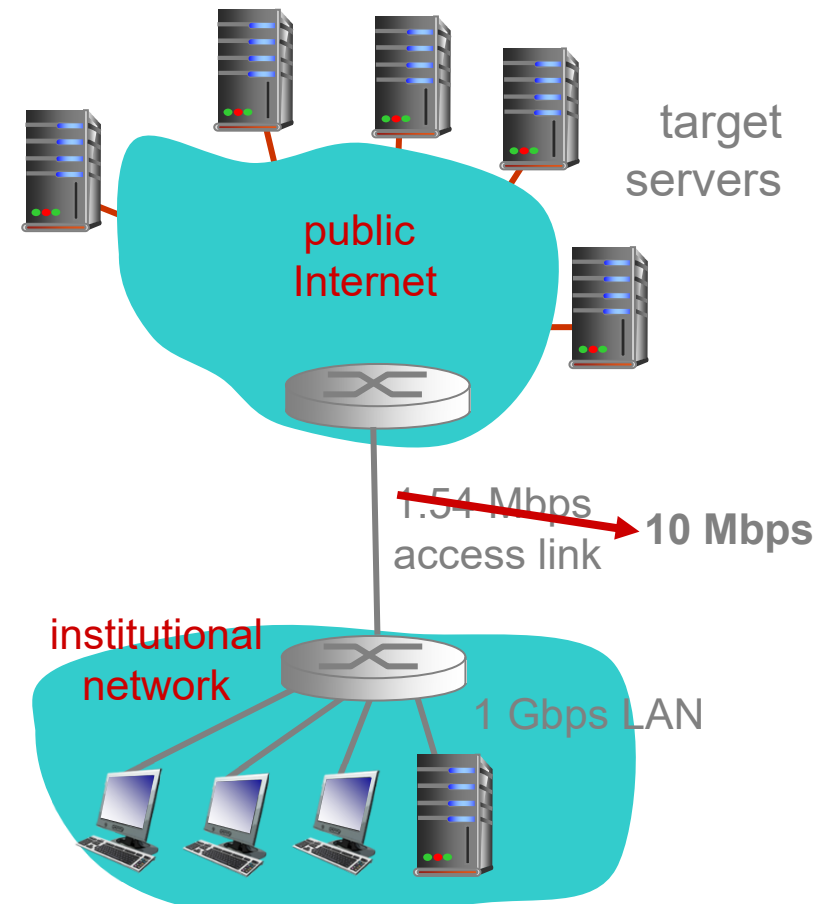


Solução possível:

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

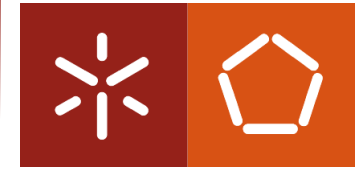
Consequência:

- Utilização da LAN = 15%
- Utilização do Link de Acesso = 15%
- Atraso Total = 2 seg + ms + ms
- É habitualmente muito dispendioso fazer o upgrade do *link* de acesso de uma instituição nesta proporção...



HTTP

Exemplo de *Web Caching*

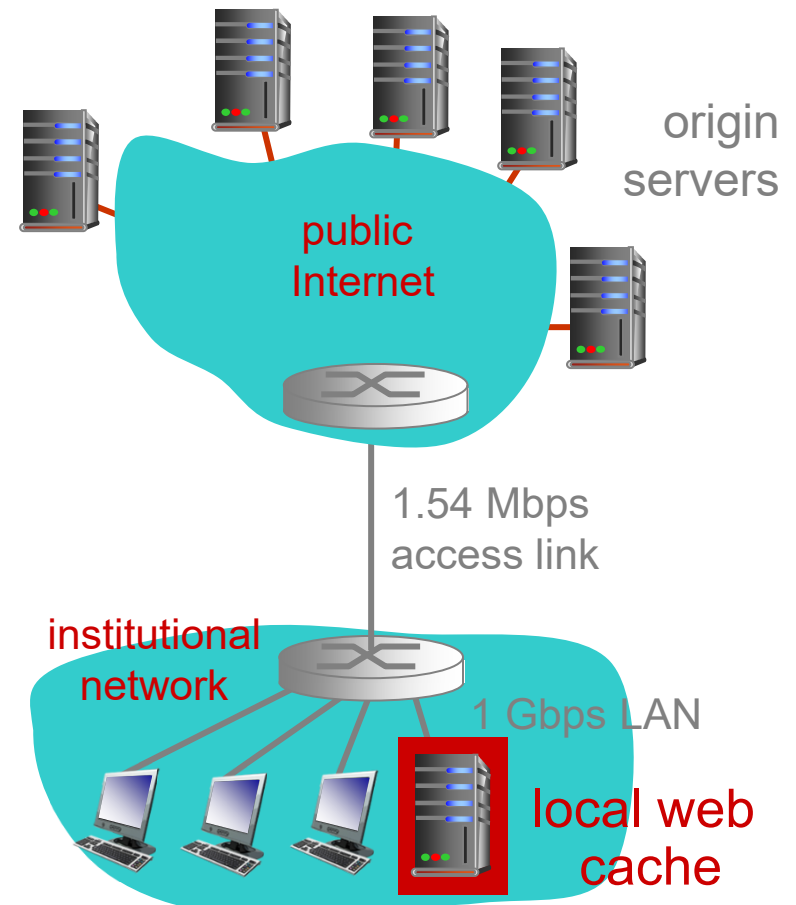


Outra solução é instalar um *Web Proxy*:

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

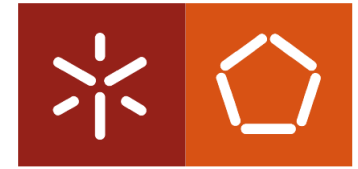
Então:

- 40% dos pedidos serão satisfeitos quase imediatamente
- 60% dos pedidos terão que ser redirecionados para o servidor HTTP alvo
- Os atrasos dos pedidos satisfeitos pelo proxy são negligenciáveis (10 ms)
- A utilização do *link* de acesso será reduzida 40%, i.e., passa a ser de $0.99 \cdot 0.6 \approx 0.6$
- Atraso Médio Total = $0.6 \cdot 2 + 0.4 \cdot 0.010 < 1.4$ seg.



HTTP

Web Caching – GET Condicional

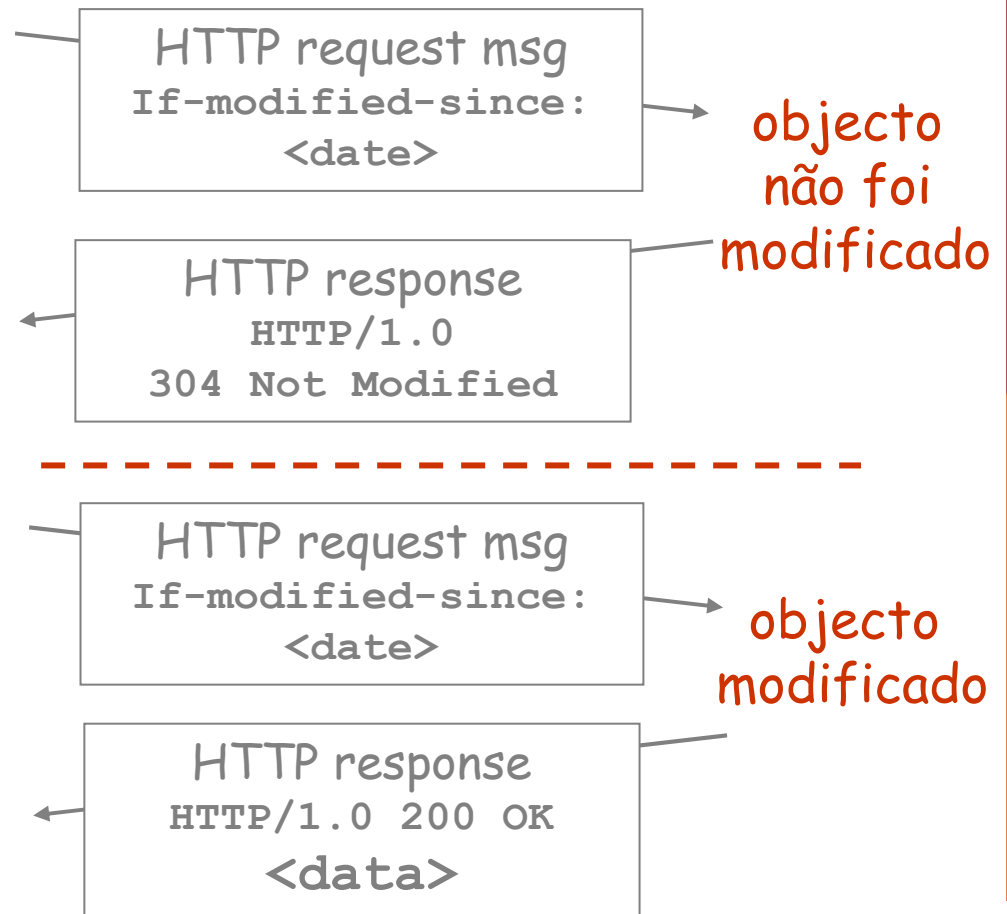


Objetivo: não enviar o objeto se a cópia mantida em *cache* está atualizada...

- O proxy inclui no cabeçalho do pedido HTTP a data da cópia guardada
- A resposta do servidor não contém nenhum objeto se o original não for mais recente.

proxy server

target server



HTTP/2

Baseado no documento

*HTTP/2, A New Excerpt from High Performance Browser Networking**

Ilya Grigorik

Comunicações por Computador

Licenciatura em Engenharia Informática

Universidade do Minho

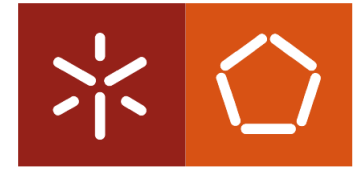


*Disponível gratuitamente em hpbrowser.com/http2

Slides respetivos disponíveis em bit.ly/http2-opt

HTTP/2

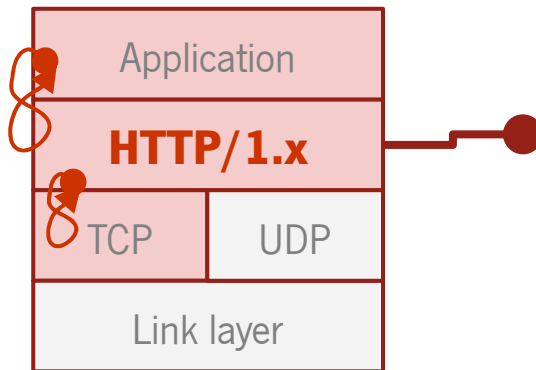
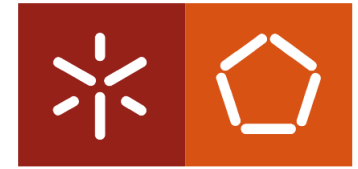
Motivação



- Como melhorar o desempenho das versões HTTP/1.*?
- Quais as melhores práticas, simples e eficazes, que têm sido usadas com regularidade?
 - Reduzir o número de consultas ao DNS (*DNS Lookups/Queries*)
 - Reutilizar as conexões TCP
 - Utilizar CDN (*Content Delivery Networks*)
 - Minimizar o número de redireccionamentos HTTP (*HTTP Redirects*)
 - Eliminar bytes desnecessários nos pedidos HTTP (redução dos cabeçalhos)
 - Comprimir os artefactos na transmissão (compressão dos dados no corpo)
 - *Cache* dos recursos do lado do cliente
 - Eliminar o envio de recursos desnecessários

HTTP/2

Motivação



Paralelismo limitado:

- O paralelismo está limitado ao número de conexões
- Na prática, mais ou menos 6 conexões por origem (HTTP/1.1 define um máximo de 2 por cliente, mas...)

Head-of-line blocking

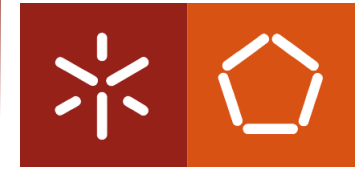
- Bloqueio do cabeça de fila, acumula pedidos em *queue* e atrasa o processamento da solicitação do cliente
- Servidor obrigado a responder pela ordem de chegada dos pedidos (ordem restrita)

Overhead protocolar é elevado

















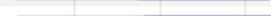

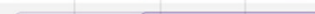



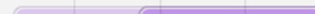




- Metadados do cabeçalho não são compactados
- Aproximadamente 800 bytes de metadados por pedido, mais os *cookies*...

HTTP/2

Motivação



- **Paralelismo é limitado pelo número de conexões...**

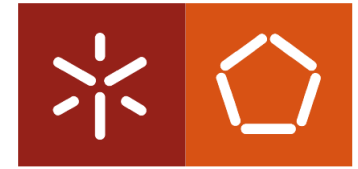
✕	Elements	Resources	Network	Sources	Timeline	Profiles	Audits	Console	PageSpeed		
Name	Method	Status	Type	Time	Start Time	302 ms	453 ms	604 ms	755 ms
 localhost	GET	200	text/html	17 ms					
 01.jpeg	GET	202	image/jpeg	242 ms					
 02.jpeg	GET	202	image/jpeg	243 ms					
 03.jpeg	GET	202	image/jpeg	242 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 client

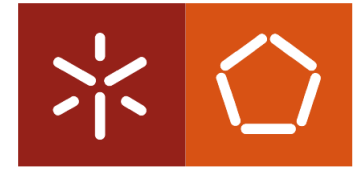
- Cada conexão implica *overhead* de *handshake* inicial
- Se for HTTPS ainda tem mais um *overhead* do *handshake* TLS
- Cada conexão gasta recursos do lado do servidor
- As conexões competem umas com as outras

HTTP/2

Mecanismos do lado do servidor...



- **Definir N subdomínios *web* em vez de se usar um único domínio por servidor (*domain sharding*)**
 - Aumenta o paralelismo — passamos a ter 6 conexões por subdomínio
 - Mas aumenta as consultas ao DNS
 - E exige mais servidores, competição nas conexões e complexidade nas aplicações
- **Reduzir pedidos → concatenar objetos (*concatenated assets*)**
 - Vários CSS ou vários JS num único objeto
 - Mas atrasa o processamento no cliente e dificulta o uso generalizado de *caching*
- **Incluir recursos em linha no HTML (*inline objects*)**
 - Os mesmos objetivos do anterior
 - E produz os mesmos problemas



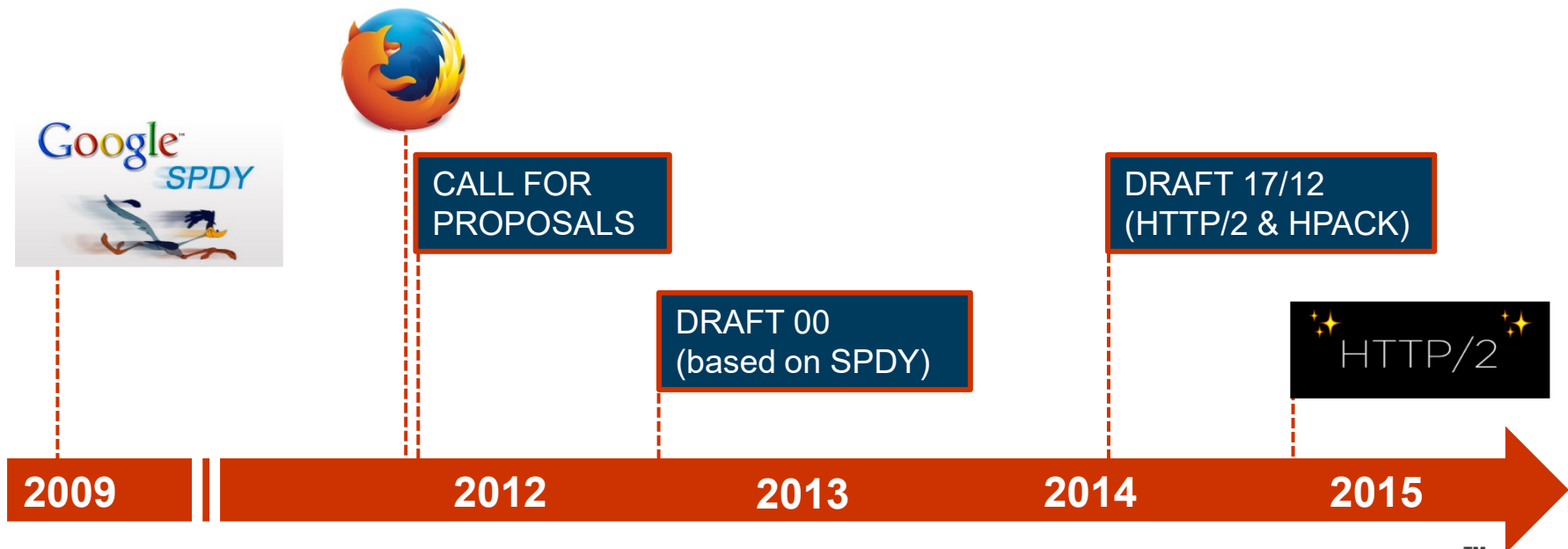
- **Em meados de 2009, a Google inicia o seu projeto SPDY:**
 - Definir e validar nova versão do protocolo que permitisse reduzir para 50% o tempo de carregamento duma página (*Page Load Time*)
 - Outros objetivos:
 - Evitar que os autores *web* tenham de “otimizar” os conteúdos
 - Minimizar o tempo de implantação e as alterações na infraestrutura
 - Desenvolver R&D 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 a própria Google...**
- **E o IETF?**
 - Teve de ir atrás, a reboque, e formar um grupo de trabalho HTTP/2...

HTTP/2

História



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



©2015 AKAMAI | FASTER FORWARD™

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 HTTP/2

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

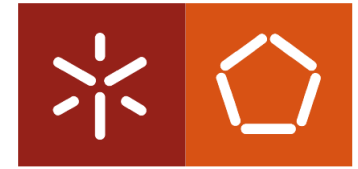
Aug, 2014: HTTP/2 draft-17 and HPACK draft-12 are published

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

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

HTTP/2

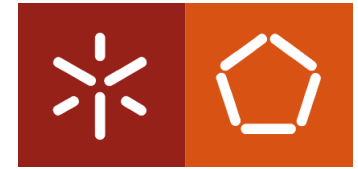
Extensão do HTTP/1.1



- **HTTP/2 é uma extensão e não uma substituição do HTTP/1.1!**
 - Não se mexe nos métodos, URL, *cabeçalhos*, códigos de resposta, etc. do HTTP/1.1
 - Semântica para a aplicação deve ser a mesma
 - Ou seja, não há alterações na API aplicacional
- **Atacar 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 de dados
 - Não há mecanismos que implementem prioridades
 - Uso desadequado das conexões TCP de suporte

HTTP/2

Resumo



1. Uma única conexão TCP!

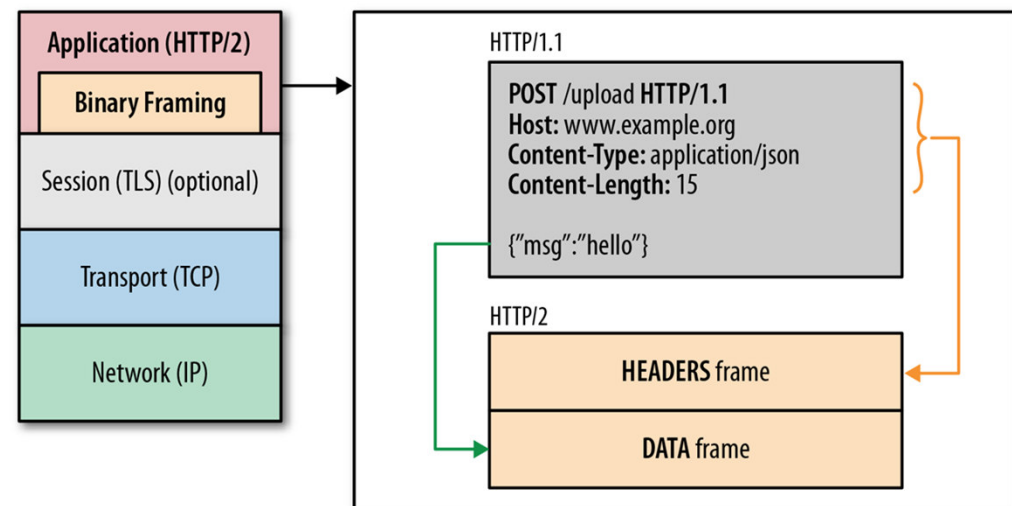
2. Pedidos → Sequências/Streams

- *Streams* são multiplexadas!
- *Streams* são priorizadas!

3. Camada de *framing* binário

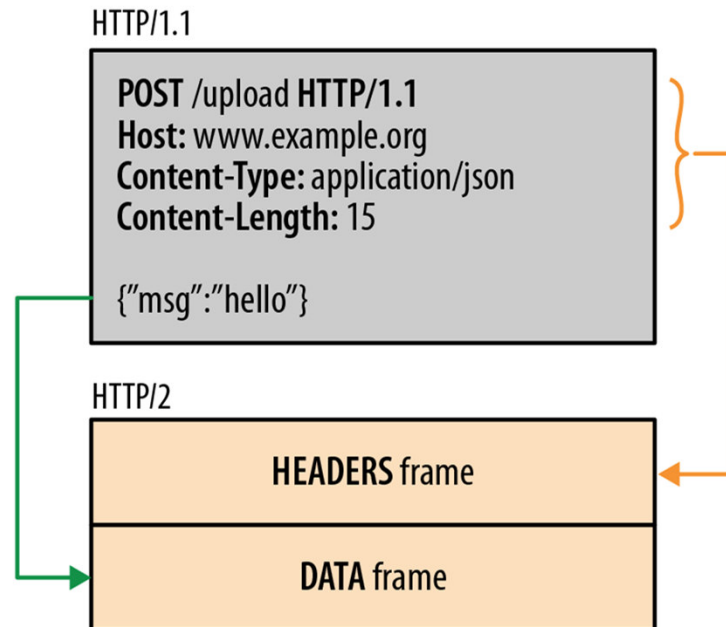
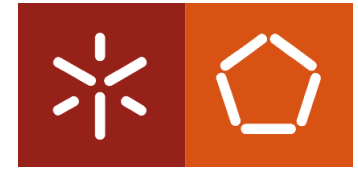
- Priorização
- Controlo de fluxo
- Server *push*

4. Compressão do cabeçalho (HPACK)



HTTP/2

Binary Framing

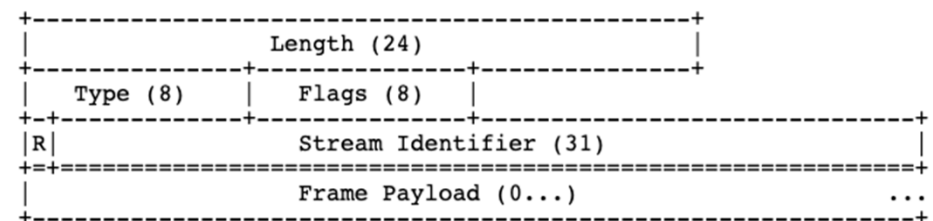


- **Mensagens HTTP são divididas em uma ou mais *frames***

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

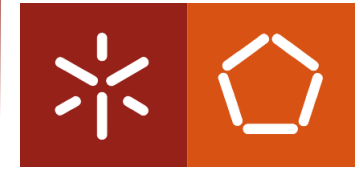
- **Cada *frame* tem um cabeçalho comum**

- 9 bytes (72 bits), com tamanho à cabeça
- De *parsing* fácil e eficiente



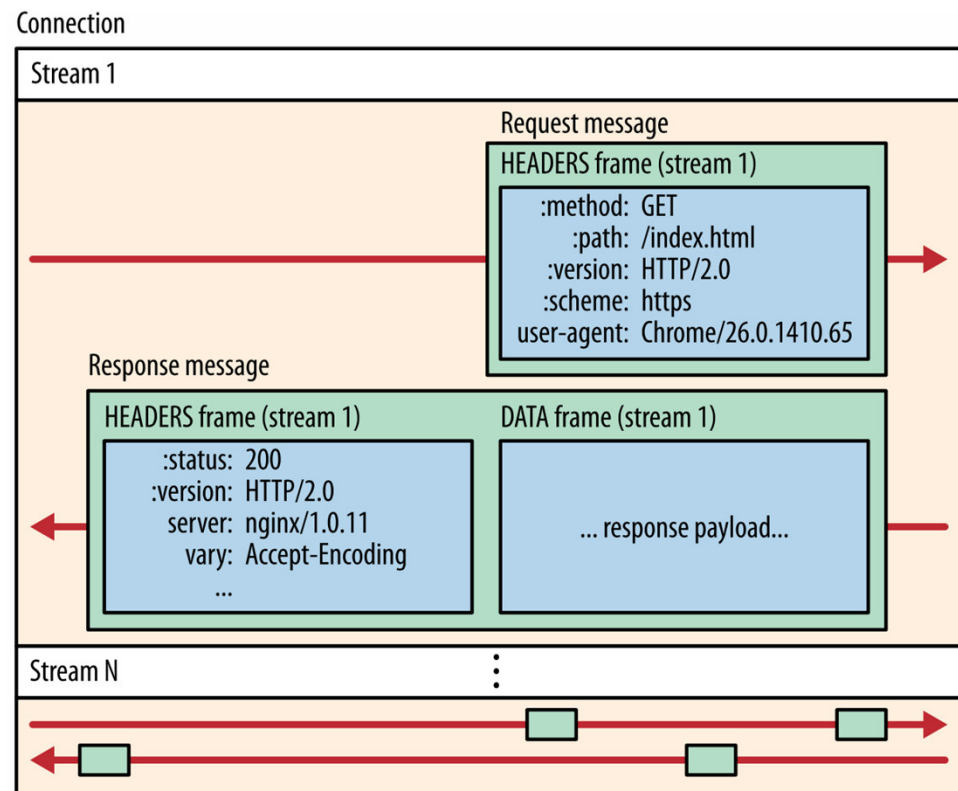
HTTP/2

Binary Framing



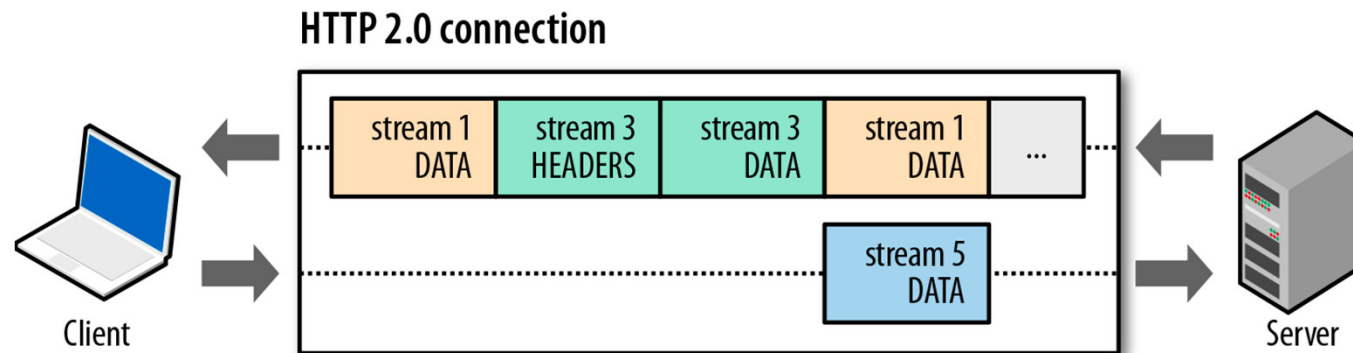
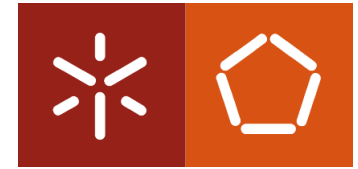
Terminologia:

- **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



HTTP/2

Fluxo de Dados

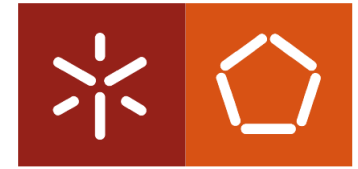


As *streams* são multiplexadas porque as *frames* pode ser intercaladas entre si:

- 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 TCP

HTTP/2

Tipos de *Frames*

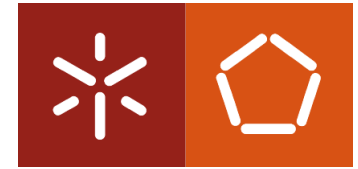


Algumas *frames* definidas no RFC7540:

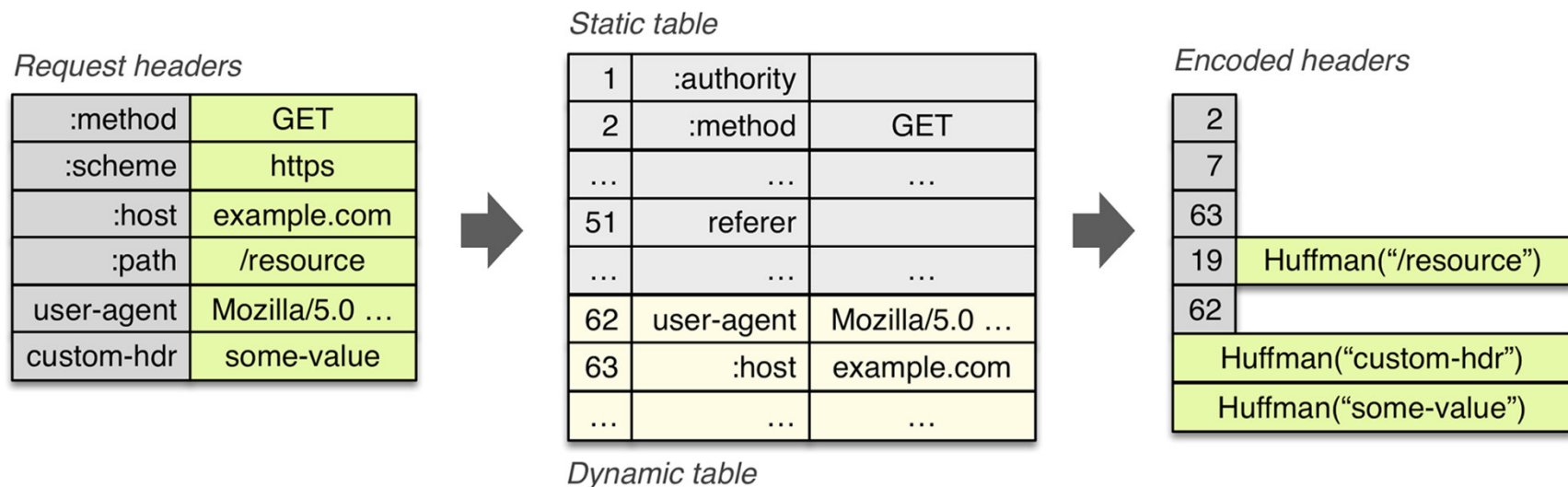
- HEADERS – cabeçalhos de um pedido ou de uma resposta
- DATA – corpo dos objetos (dados)
- PRIORITY – define a prioridade da *stream* para o originador
- RST_STREAM – permite o término imediato da *stream*
- SETTINGS – para definir parâmetros de configuração
SETTINGS_HEADER_TABLE_SIZE, SETTINGS_ENABLE_PUSH, 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, etc.

HTTP/2

HPACK – Compressão do cabeçalho

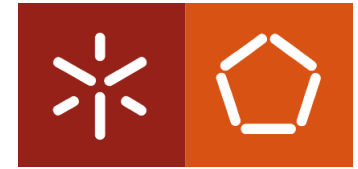


- Valores literais (texto) são comprimidos com codificação estatística Huffman
- Uma tabela de indexação estática → por ex: “2” corresponde a “method: GET”
- Uma tabela de indexação dinâmica → Valores enviados anteriormente são indexados!



HTTP/2

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/detalhe no envio de recursos:

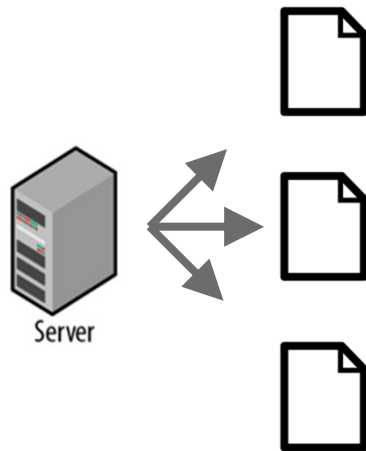
- Evita o *inlining* e permite *caching* eficiente dos recursos
- Permite multiplexar e definir prioridades no envio dos recursos
- É necessário permitir controlo de fluxo por parte dos clientes

HTTP/2

Server Push



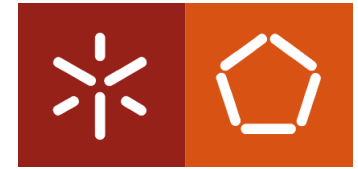
Devem ser implementadas estratégias inteligentes de “Server push”
(por exemplo, implementação Jetty)



1. Servidor observa o tráfego de entrada
 - a. Constrói um modelo de dependências baseado no campo *Referer* do cabeçalho (ou outros),
e.g. `index.html` → `{style.css, app.js}`
2. Servidor inicia um *push* inteligente de acordo com as dependências que aprendeu
 - a. `client` → `GET index.html`
 - b. `server` → **push** `style.css, app.js, index.html`

HTTP/2

Controlo de Fluxo



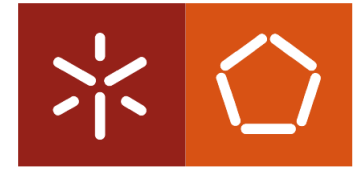
I want image geometry and preview, and I'll fetch the rest later...

- **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."

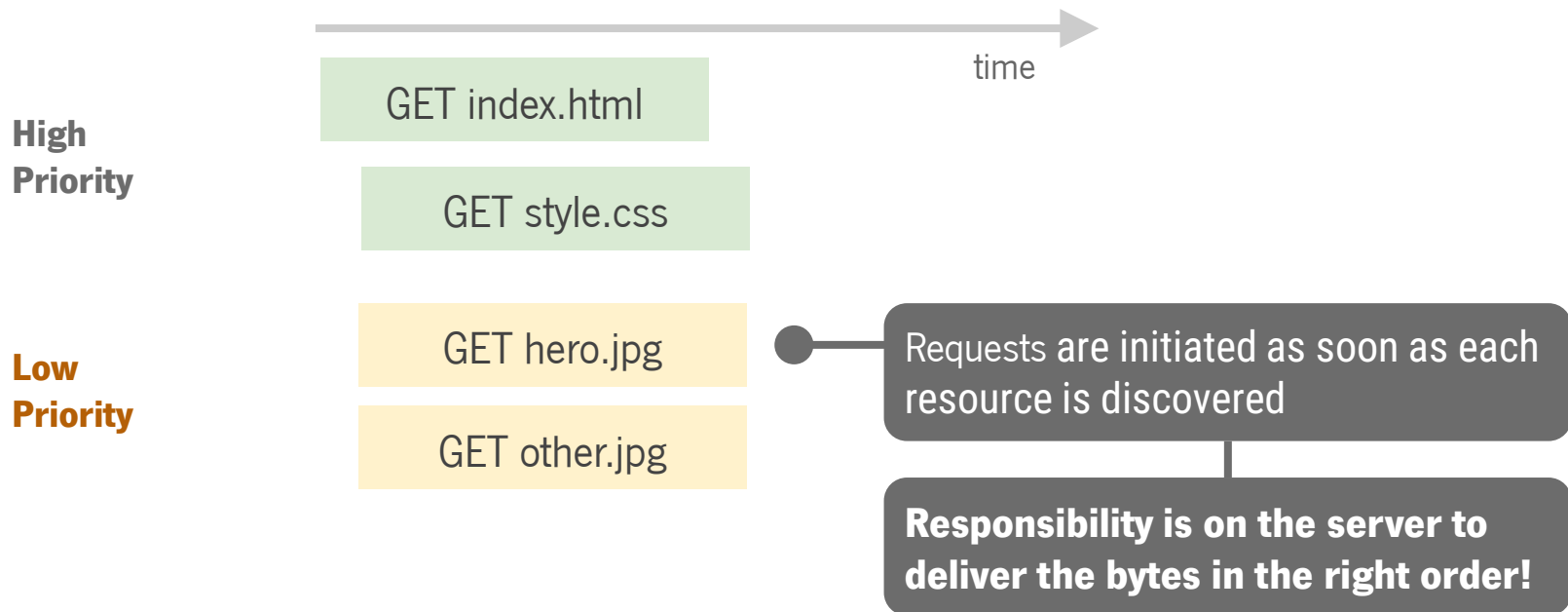
- Permite ao cliente fazer uma pausa na *stream* e retomar o envio mais tarde
- Controlo de fluxo é baseado num sistema de créditos numa janela:
 - Cada *frame* do tipo **DATA** decrementa o número de créditos
 - Cada *frame* do tipo **WINDOW_UPDATE** atualiza/repõe o valor máximo

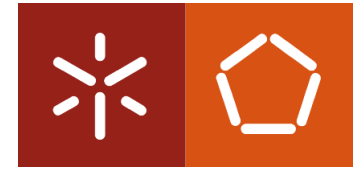
HTTP/2

Priorização

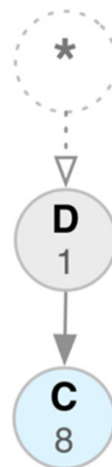
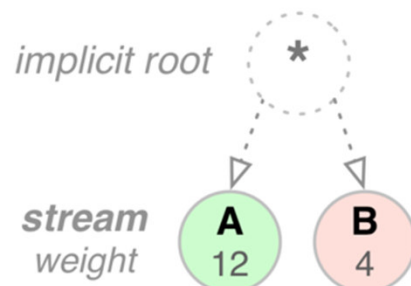


- A priorização é fundamental para um *rendering* eficiente/adequado
- Com HTTP/2, o cliente define as prioridades e faz logo os pedidos; cabe ao servidor entregar os conteúdos com a prioridade certa

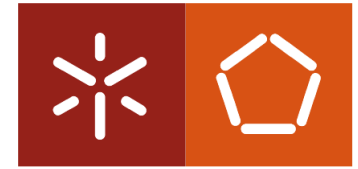




- Exemplo do uso de pesos: a *stream A* deve ter 12/16 e a *stream B* deve ter 4/16 dos recursos totais
- Exemplo do uso de dependências: a *stream D* deve ser entregue antes da *stream C*

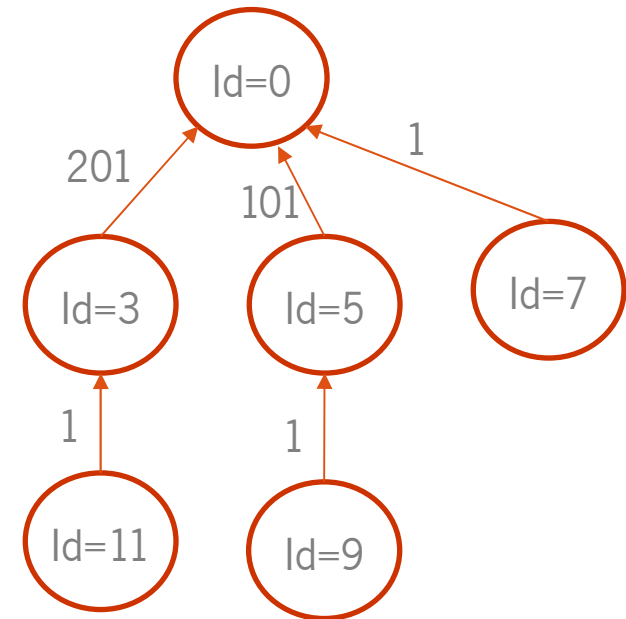


- Cada *stream* pode ter um peso (um número inteiro entre 1 e 256)
- Cada *stream* pode ter uma dependência de outra *stream*



Exemplos de 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 (raiz virtual)
- 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



HTTP/2

Negociação Protocolar



O cliente tem 3 formas para tentar usar o HTTP/2, não podendo assumir que todos os servidores o suportam:

1. Começando em HTTP/1.* e pedido "upgrade" da conexão (semelhante ao mecanismo usado para os *WebSockets*)
2. Usando HTTPS e negociando o protocolo HTTP/2 durante o *handshake* TLS inicial
3. Sabendo que o servidor é HTTP/2, envia logo sequência inicial HTTP/2

NOTA: A Google e outros defendiam que destes três mecanismos só se deveria usar o segundo mas o IETF impôs o suporte para os métodos restantes...

HTTP/2

Negociação Protocolar



Mecanismo de *upgrade* duma conexão HTTP/1.* para HTTP/2:

```
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 HTTP/1.1 e pede *upgrade* para HTTP/2

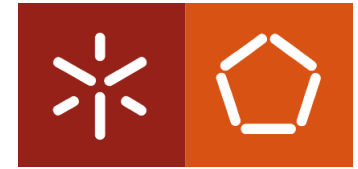
2. *Settings* codificados em BASE64

3. Servidor declina pedido, respondendo apenas em HTTP/1.1

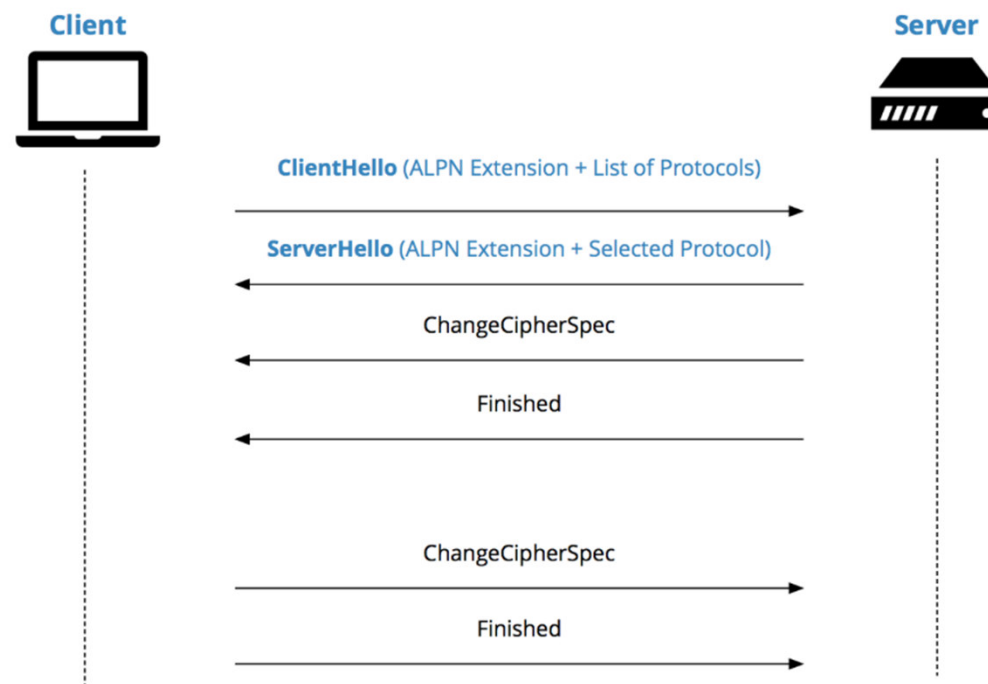
4. Servidor aceita pedido para HTTP/2 e começa *Framing* binário

HTTP/2

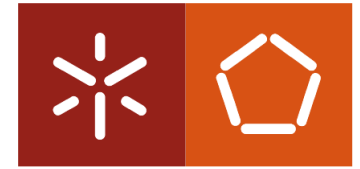
Negociação Protocolar



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 HTTP/2 se e só se o cliente já souber que o servidor suporta HTTP/2:

Enviar a sequência de 24 octetos

`0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a`

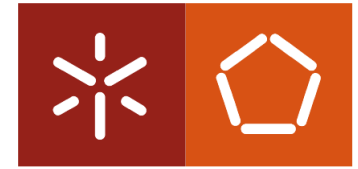
que corresponde a

```
PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n
```

logo seguido duma *frame* de SETTINGS para definir os parâmetros da conexão HTTP/2...

HTTP/2

Testar!



- **Experimentar URLs:**

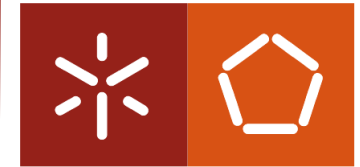
- `https://http2.akamai.com/demo`
- `http://www.http2demo.io/`
- `https://http2.golang.org/serverpush`

- **Usar o magnífico nghttp2:**

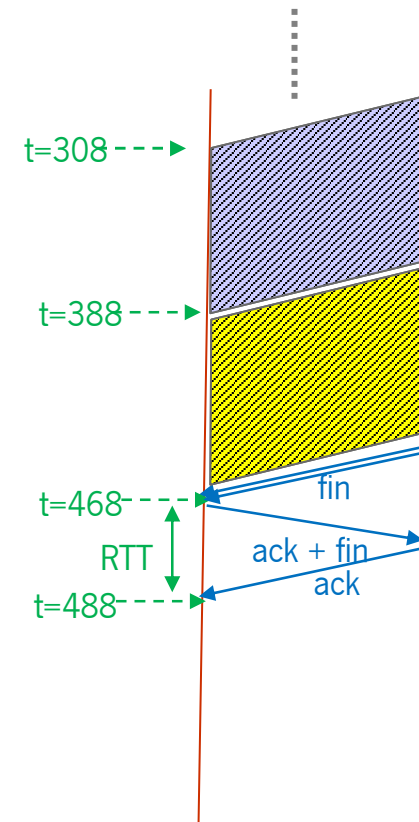
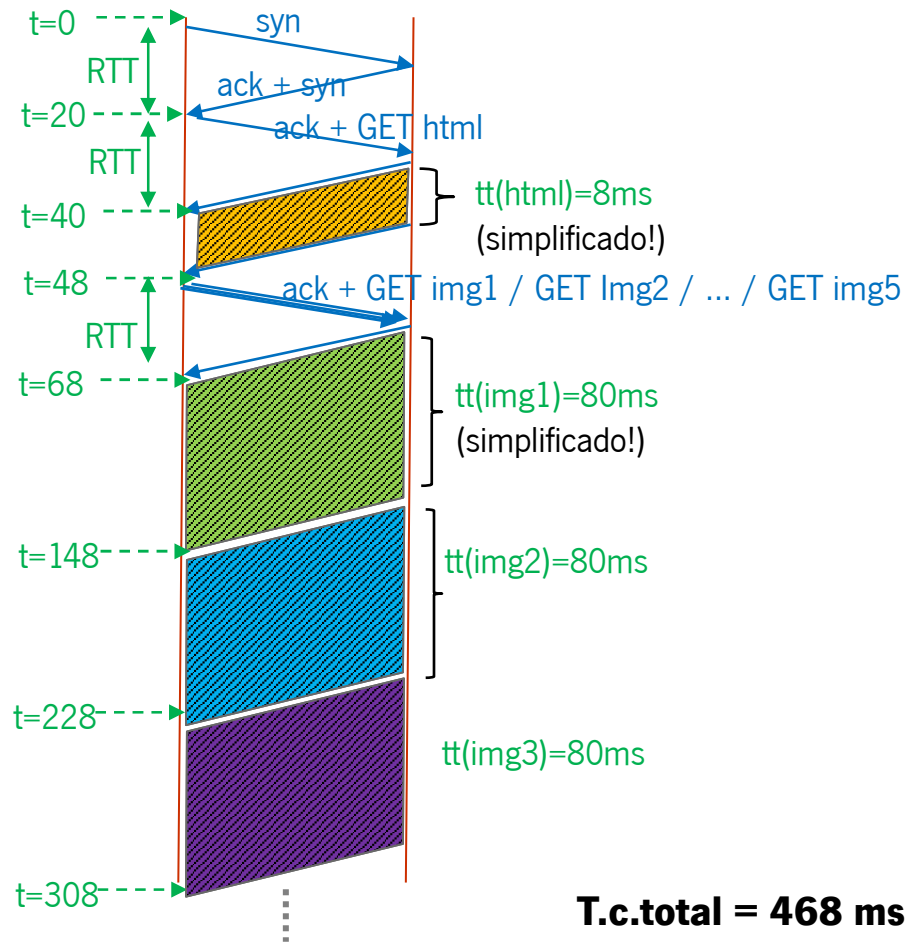
```
$ nghttp -vv -a -n -y -s https://http2.golang.org/serverpush
```

HTTP/2

Relembrar exercício...



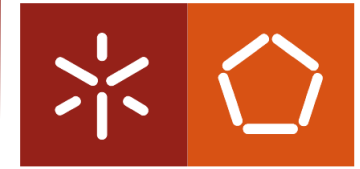
HTTP/1.1 persistente, com pipeline, sem conexões em paralelo:



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

HTTP/2

Relembrar exercício...



HTTP/2 múltiplas *streams* numa mesma conexão com server *push*:

