

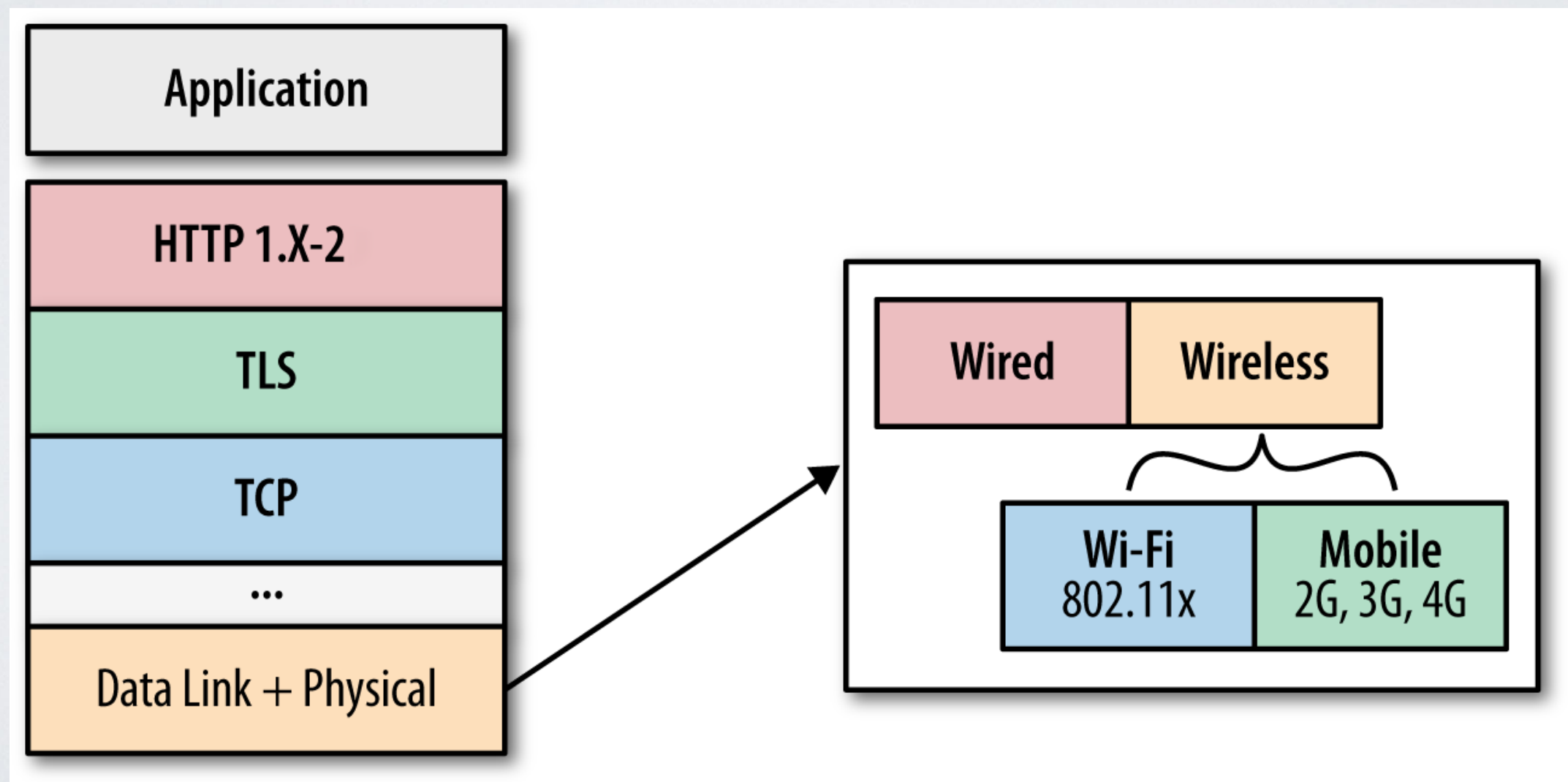
EL CAMINO HACIA HTTP/2

@luisddm_

HTTP

EN LA PILA DE PROTOCOLOS TCP/IP

- Protocolo de **extremo a extremo**, sobre un canal **orientado a conexión** y **libre de errores** de transmisión.
- La **morfología** de la red es **transparente** para HTTP.



HTTP/0.9 (NO OFICIAL)

EL PROTOCOLO DE UNA SOLA LÍNEA (1991)

ASCII, cliente-servidor, petición-respuesta, funciona sobre **TCP/IP**.

- La **petición** del cliente es cadena de texto ASCII que termina con un retorno de carro (CRLF).
- La **respuesta** del servidor es un *stream* ASCII formateado como HTML.
- La **conexión** termina después de cada petición, una vez que la transferencia del documento ha finalizado.

```
$ telnet localhost 80
```

```
Connected to localhost.
```

```
GET /
```

```
<html><body><h1>It works!</h1></body></html>
```

```
Connection closed by foreign host.
```


HTTP/1.0

RFC INFORMATIVO (1996)

Se creó para documentar las “**características comunes**” de las implementaciones de HTTP/1.0 que se encontraron por ahí.

No es una especificación formal de un estándar.

```
$ telnet localhost 80
Connected to localhost.

GET /resource HTTP/1.0
UserAgent: FakeUA
Accept: */*

HTTP/1.0 200 OK
Content-Type: text/plain
Content-Length: 137582
Server: Apache 2.4.8

(payload of any type)
Connection closed by foreign host.
```

- La **petición** puede consistir en múltiples líneas separadas por retornos de carro.
- La **respuesta** viene encabezada por un estado seguido de varias cabeceras, cada una también en su propia línea.
- La **conexión** entre servidor y cliente se cierra después de cada petición.

HTTP/1.0

RFC INFORMATIVO (1996)

- Tanto la **petición** como las **cabeceras** de la respuesta siguen siendo **ASCII**.

- Pero el **objeto** que lleva la respuesta ahora puede ser de **cualquier tipo**.

HTML
Texto plano
Imagen
...

- El RFC documenta también unas cuantas **capacidades nuevas**.

codificación de contenido
selección de charset
clave de autorización
cacheado
comportamiento de proxies
formato de fechas
...

- Prácticamente todos los servidores **actuales** en funcionamiento pueden funcionar en HTTP/1.0.
- Sigue haciendo falta **una conexión TCP por cada petición**.

HTTP/1.1

ESTÁNDAR DE INTERNET (1997 -1999)

```
$ telnet website.org 80
Connected to xxx.xxx.xxx.xxx
```

```
GET /index.html HTTP/1.1
Host: website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
10_11_6)... (snip)
Accept: text/html,application/xhtml+xml,application/
xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: __qca=P0-800083390... (snip)
```

```
HTTP/1.1 200 OK
Server: nginx/1.9.10
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Via: HTTP/1.1 GWA
Date: Wed, 25 Jul 2012 20:23:35 GMT
Expires: Wed, 25 Jul 2012 20:23:35 GMT
Cache-Control: max-age=0, no-cache
Transfer-Encoding: chunked
```

```
100
<!doctype html>
(snip)
```

```
100
(snip)
```

```
0
```

```
GET /favicon.ico HTTP/1.1
Host: www.website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
10_11_6)... (snip)
Accept: */*
Referer: http://website.org/
Connection: close
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: __qca=P0-800083390... (snip)
```

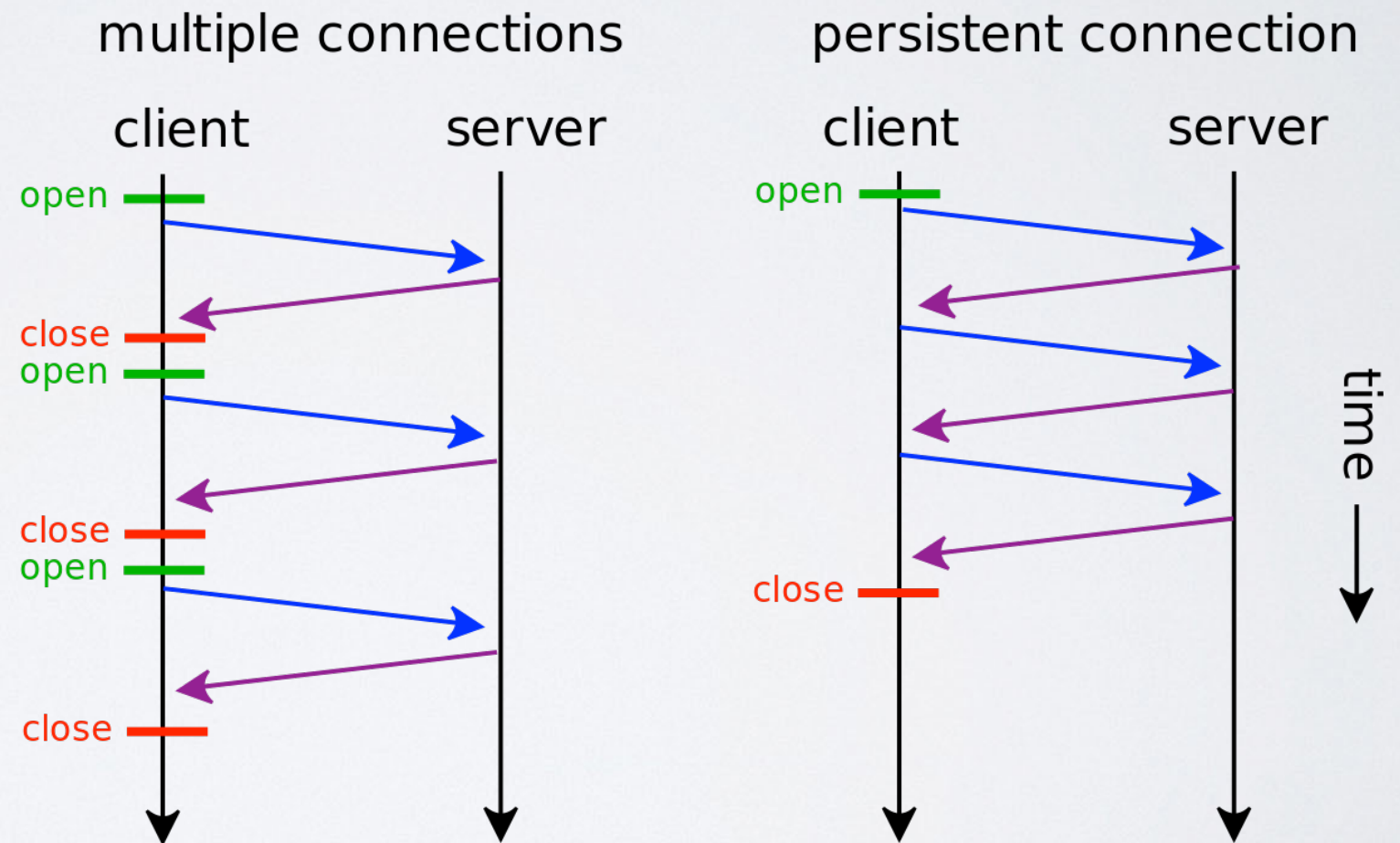
```
HTTP/1.1 200 OK
Server: nginx/1.9.10
Content-Type: image/x-icon
Content-Length: 3638
Connection: close
Last-Modified: Thu, 19 Jul 2012 17:51:44 GMT
Cache-Control: max-age=315360000
Accept-Ranges: bytes
Via: HTTP/1.1 GWA
Date: Sat, 21 Jul 2012 21:35:22 GMT
Expires: Thu, 31 Dec 2037 23:55:55 GMT
Etag: W/PSA-GAu26oXbDi
```

```
(icon data)
(connection closed)
```


HTTP/1.1

ESTÁNDAR DE INTERNET (1997 -1999)

- El estándar HTTP/1.1 introduce **optimizaciones** de rendimiento, como las conexiones con *keepalive*, peticiones con un rango de bytes determinado, mecanismos adicionales de cacheo y compresión, *pipelining*, *cookies*, etc.
- El **keepalive** permite reutilizar una misma conexión TCP **persistente** para hacer varias peticiones en serie, **evitando** varios **RTT** (Round Trip delay Time).
- Para terminar la conexión hay que **cerrarla de forma explícita** con un *Connection: close*.

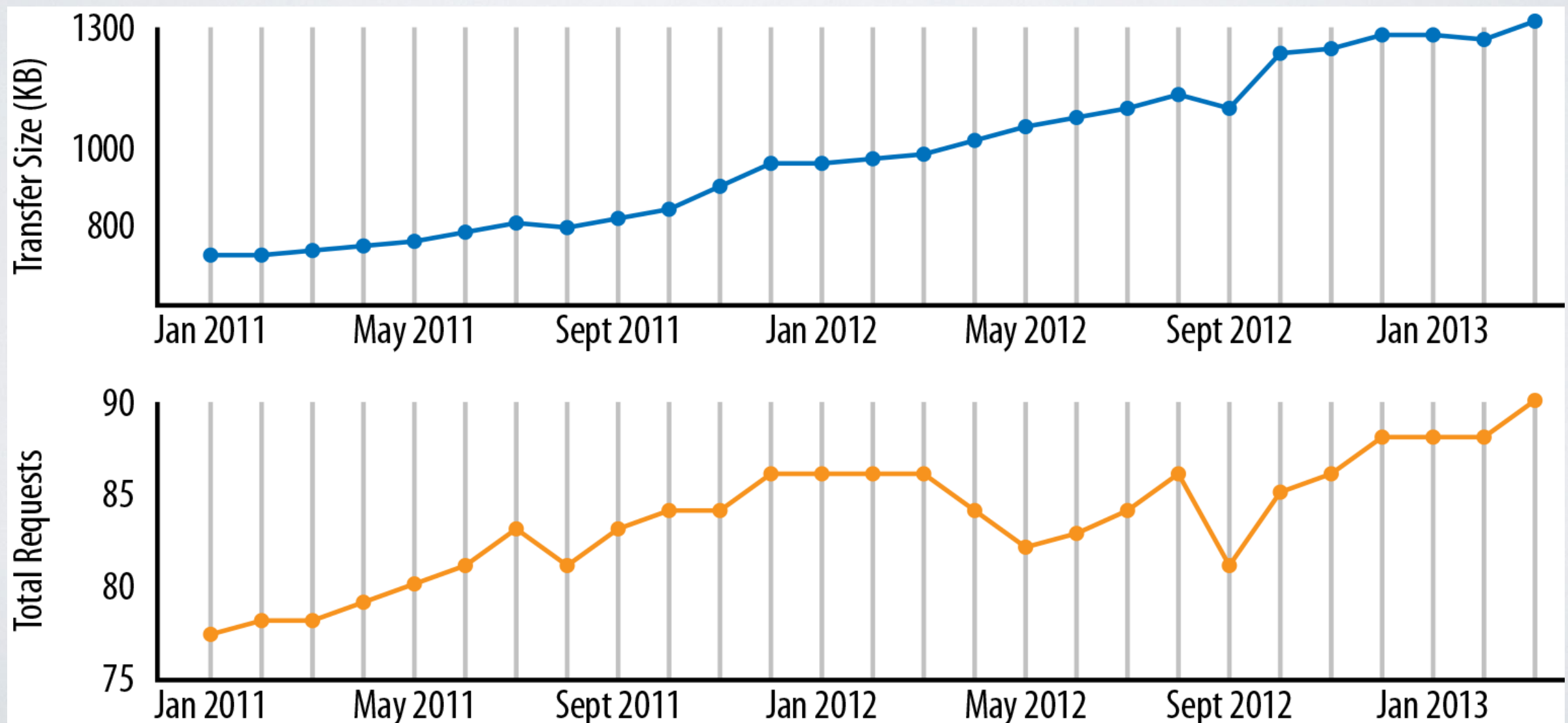


ANATOMÍA

DE UNA APLICACIÓN WEB MODERNA

90 peticiones a **15 hosts**, con un tamaño de 1311 KB.

- **HTML:** 10 peticiones, 52 KB.
- **Imágenes:** 55 peticiones, 812 KB.
- **Javascript:** 15 peticiones, 216 KB.
- **CSS:** 5 peticiones, 36 KB.
- **Otros:** 5 peticiones, 195KB.

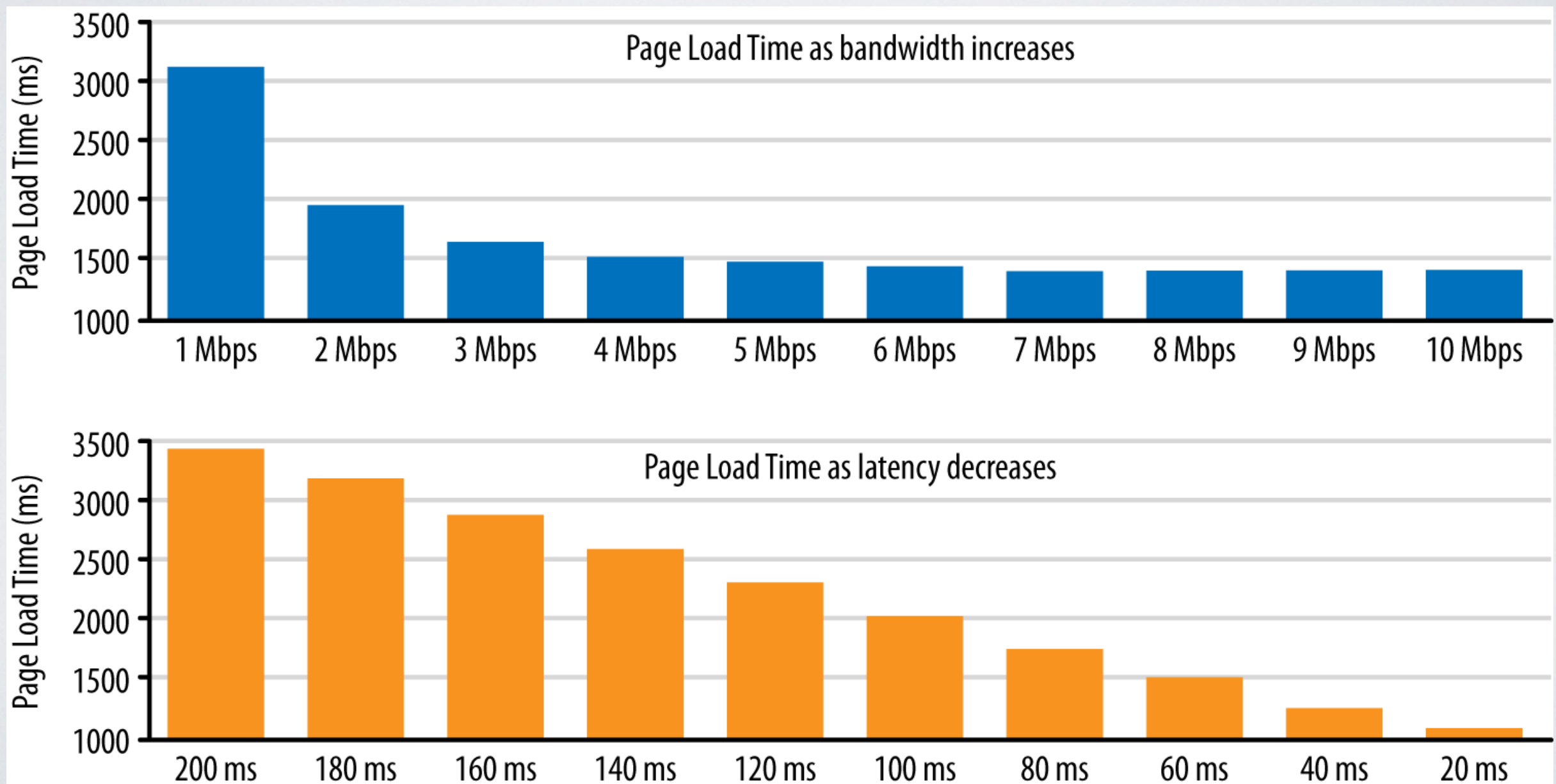


VELOCIDAD Y RENDIMIENTO

Y PERCEPCIÓN HUMANA

El ancho de banda no importa demasiado (a partir de cierto punto).

¡Es la latencia, estúpido!



VELOCIDAD Y RENDIMIENTO

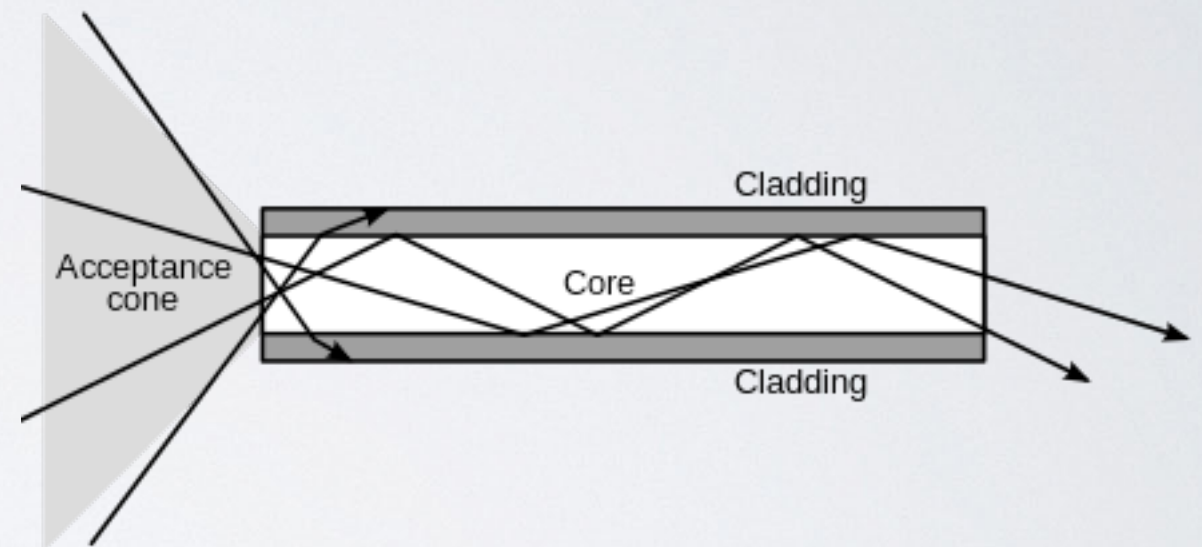
Y PERCEPCIÓN HUMANA

Mejorar el ancho de banda es fácil.

- Muchísimos cables de fibra óptica desplegados en redes troncales sin usar.
- “Tú echa otro cable y ya está...”.

Mejorar la latencia es caro... ¿o imposible?

- Limitación: la velocidad de la luz. La fibra óptica ya transmite señales a unos $2/3$ de c , y la necesidad de refracción para la propagación impide que vaya más rápido.
- ¿Cables más cortos?



Y en redes móviles bajar la latencia se complica muuuuucho más.

OPTIMIZACIÓN: MENOS ARCHIVOS, MÁS GRANDES

HACKS DE HTTP/1.1

Concatenar todos los recursos en un paquete.

- Concatenamos todos los archivos de texto del mismo tipo entre sí, y las imágenes en *sprites*.
- Eliminamos la sobrecarga de cabeceras (mejor un archivo grande que 100 pequeños).



Embeber los recursos pequeños en el documento principal (***inlining***).

```

```

- Por ejemplo, incrustar el código binario de una imagen codificándolo en base64.

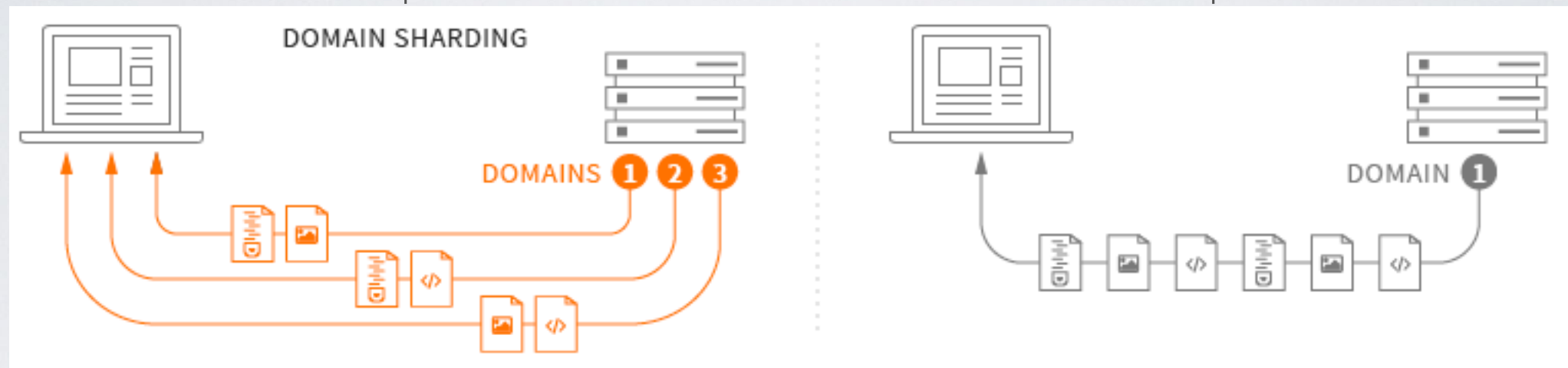
¡Pero hay que esperar a recibir todo el *bundle* para renderizar!

OPTIMIZACIÓN: SHARDING

HACKS DE HTTP/1.1

Establecer un **sharding de (sub) (pseudo)dominios**.

- Distribuimos los recursos para evitar el límite de 6 conexiones TCP por dominio.



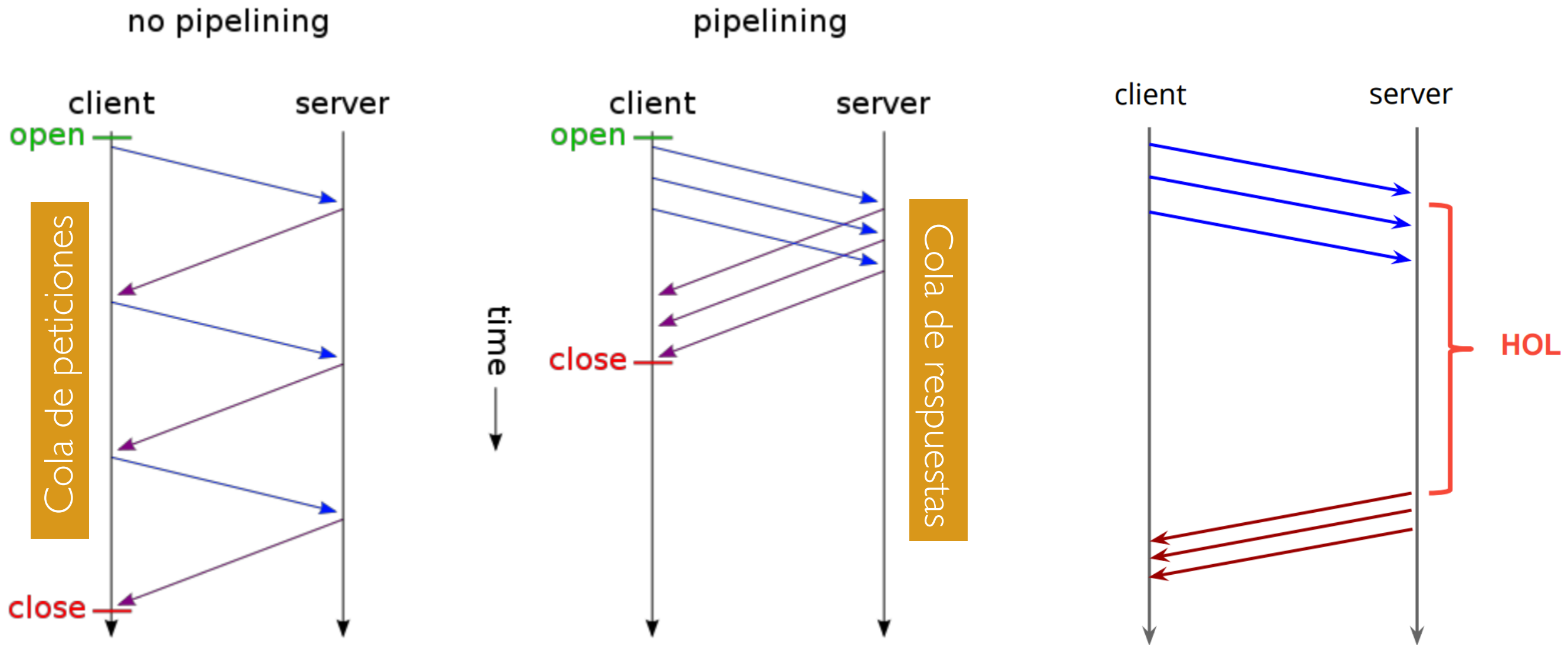
200	GET	174.jpg	w.cdn-expressen.se	jpeg	6.14 KB	→ 105 ms
200	GET	174.jpg	y.cdn-expressen.se	jpeg	4.19 KB	→ 172 ms
200			z.cdn-expressen.se	jpeg	4.48 KB	→ 223 ms
200			x.cdn-expressen.se	jpeg	4.58 KB	→ 173 ms
200			y.cdn-expressen.se	jpeg	35.18 KB	→ 56 ms
200			w.cdn-expressen.se	jpeg	12.97 KB	→ 165 ms
200			y.cdn-expressen.se	jpeg	4.83 KB	→ 56 ms
200			z.cdn-expressen.se	jpeg	9.54 KB	→ 228 ms
200			w.cdn-expressen.se	jpeg	182.50 KB	→ 285 ms
200			y.cdn-expressen.se	jpeg	5.66 KB	→ 104 ms
200			z.cdn-expressen.se	jpeg	12.24 KB	→ 287 ms
200			y.cdn-expressen.se	jpeg	6.85 KB	→ 225 ms
200			z.cdn-expressen.se	jpeg	7.50 KB	→ 173 ms
200			y.cdn-expressen.se	gif	2.85 KB	→ 227 ms
200			z.cdn-expressen.se	jpeg	50.07 KB	→ 188 ms

Interfiere con el control de congestión e invierte tiempo en varios DNS *lookup* extra

OPTIMIZACIÓN: PIPELINING

HACKS DE HTTP/1.1

Enviar **varias peticiones a la vez** para intentar reducir la latencia.



RFC: “el servidor **DEBE** enviar las respuestas a las peticiones en el **mismo orden** que fueron recibidas”.

Head-of-line blocking: una cola FIFO permanece bloqueada hasta que no se envíe el primer paquete.

OPTIMIZACIÓN: PIPELINING

HACKS DE HTTP/1.1

- Con lo cual, el pipelining, aunque a priori parezca potente, **no funciona** empíricamente.
- **Deshabilitado por defecto** (y a veces ni siquiera implementado) en los principales navegadores.

HTTP/2

¿PARA QUÉ UN NUEVO PROTOCOLO?

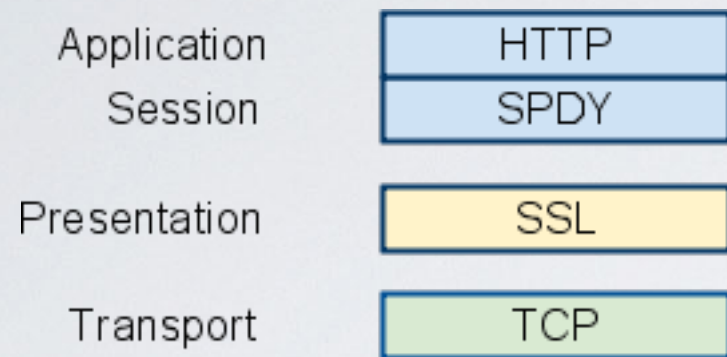
- Debido a su simplicidad y a la **ubicuidad** de los navegadores, HTTP puede llegar a **morir de éxito**.



- Hemos **migrado prácticamente todo a la web**, incluyendo aplicaciones con todo tipo de recursos diferentes.

EL PREDECESOR: SPDY

CREADO POR GOOGLE



Soportado por casi todos los navegadores...

¡excepto por Chrome! (deprecado en v51).

- SPDY añade una **capa de sesión** sobre SSL y por debajo de HTTP.
- Permite peticiones **concurrentes** y **multiplexadas** sobre una sola conexión TCP.
- SSL obligatorio, aunque añade una pequeña penalización a la latencia.
- Los mensajes habituales de tipo GET y POST siguen siendo iguales, pero SPDY especifica un **nuevo formato de trama** para codificar y transmitir los datos.
- Reduce el ancho de banda **comprimiendo** cabeceras y eliminando las innecesarias.
- **Fácil de implementar** y más sencillo.
- Permite que al servidor iniciar la transferencia de forma autónoma mediante mensajes **PUSH**.

COMPATIBILIDAD HACIA ATRÁS

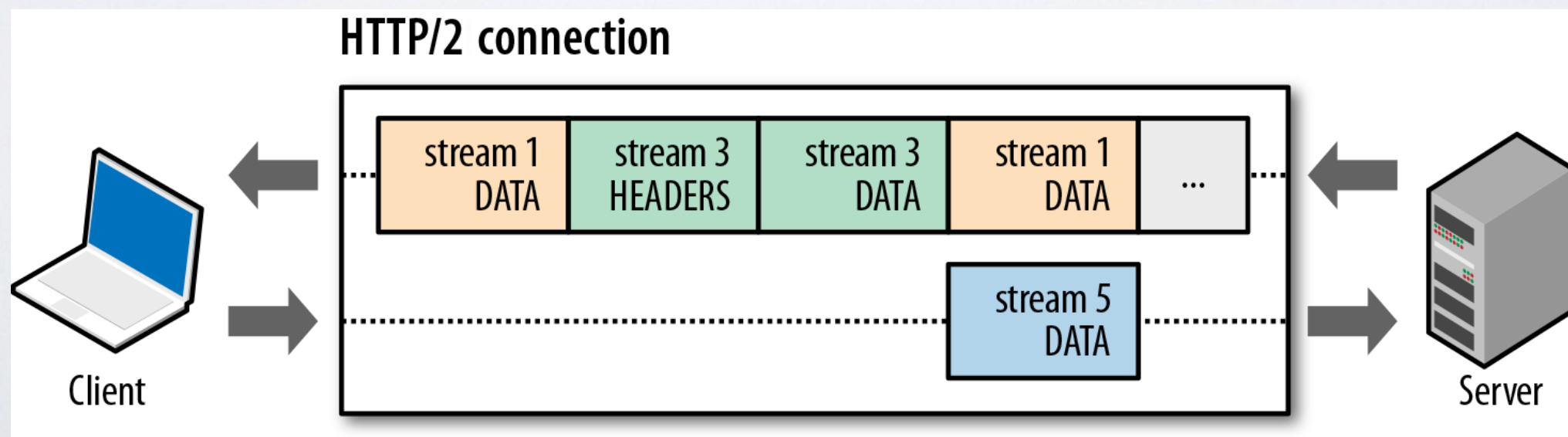
EL PROTOCOLO MANTIENE LA API

- HTTP/2 conserva casi **toda la sintaxis de alto nivel** de HTTP/1.1, como los métodos, los códigos de estado, cabeceras, URIs...
- Lo que **cambia** es cómo se **empaquetan los datos** y cómo se **transportan** de extremo a extremo.

USO DE *STREAMS*

MULTIPLEXADO E INTERCALADO DE TRAMAS

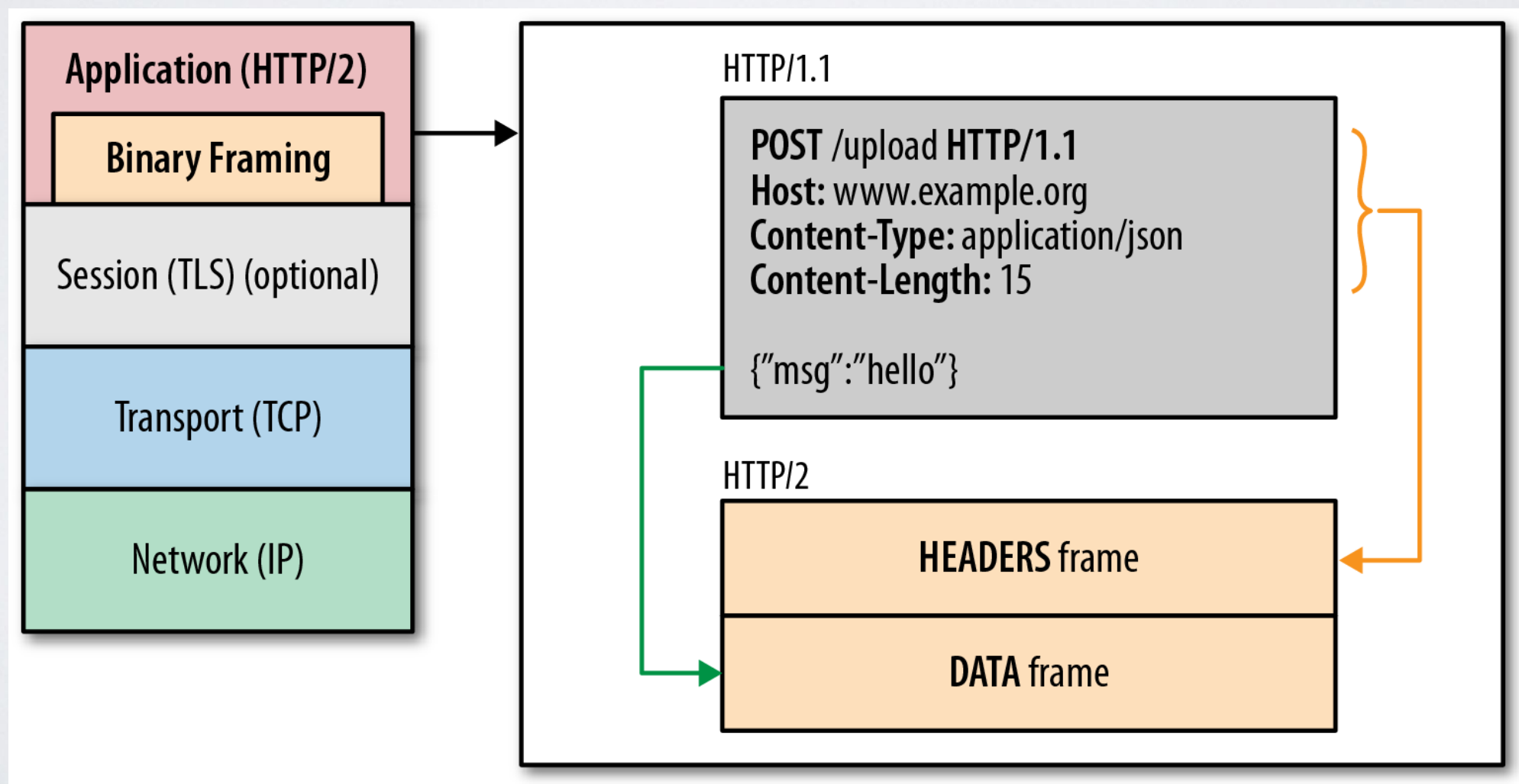
- El **cliente** puede **pedir recursos en cualquier momento**, y el **servidor** también **enviarlos por sí mismo** si lo considera necesario.
- **No hay que esperar** a que terminen las anteriores transacciones.
- La conexión puede llevar cualquier número de ***streams* bidireccionales**.
- Cada ***stream*** tiene un **id único** y una cierta **prioridad**.
- Cada **mensaje** HTTP es una petición, o una respuesta, compuesto de **una o varias tramas**.
- La **trama** es la **unidad más pequeña** de comunicación, y cada una lleva un tipo específico de datos (cabeceras, *payload*, etc).
- Las **tramas de diferentes *streams*** se **intercalan** en la transmisión y se **reconstruyen** en el receptor, gracias al id del *stream* que lleva cada una.



PROTOCOLO BINARIO

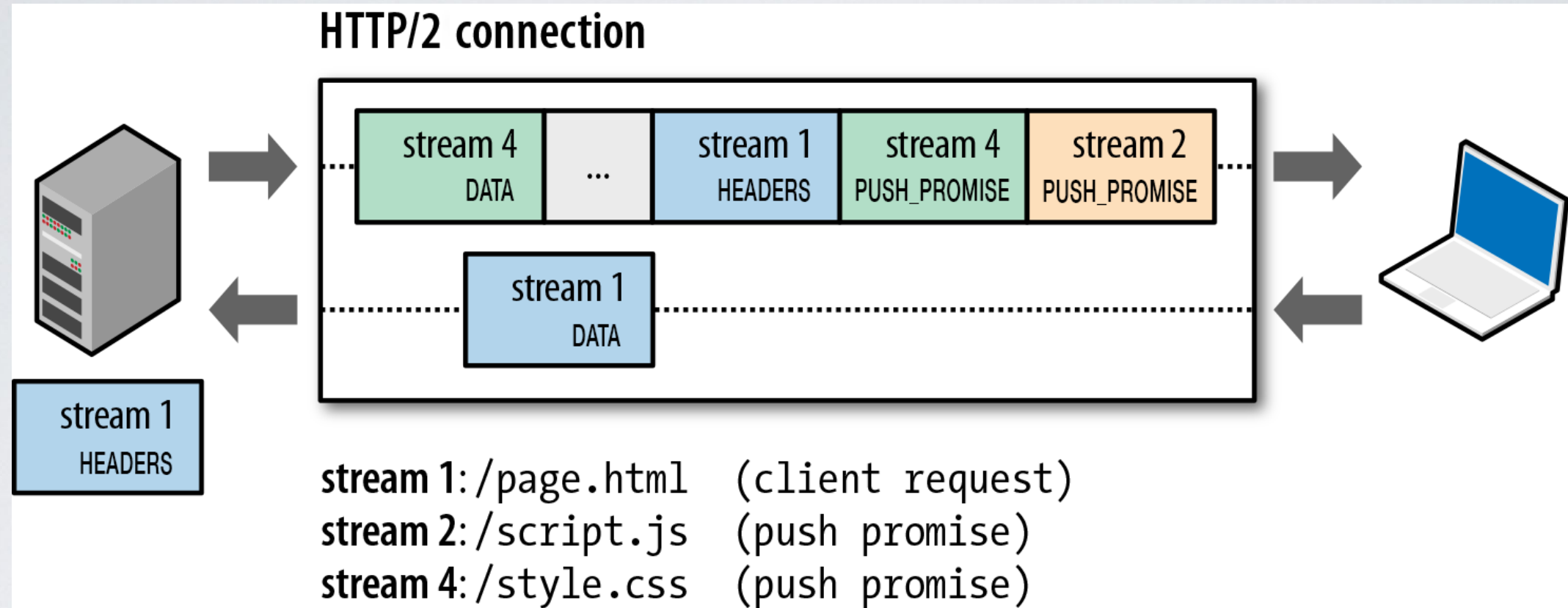
¿CÓMO LO LEO AHORA?

- **¡No más telnet!** No apto para humanos :(
- Se mejora la **interoperatibilidad** (ya no hay problema con espacios, tabs, saltos de línea...), mejora la **eficiencia** de parseo, facilita la **compresión**.
- Con **Wireshark** podremos seguir analizando el tráfico.



SERVER PUSH

RECIBIR RECURSOS QUE EL CLIENTE VA A NECESITAR



- El servidor ha de indicar que va a **enviar recursos no pedidos explícitamente** con una trama de tipo PUSH_PROMISE.
- Esto permite al servidor proveer por adelantado al cliente con recursos que sabe que va a necesitar para el renderizado, **ahorrando nuevos RTTs**.

CIFRADO

OPCIONAL PERO OBLIGATORIO DE FACTO

- El estándar HTTP/2 dice que el protocolo puede funcionar sin cifrado y con cifrado (TLS 1.2 o superior).
- Pero prácticamente **todas las implementaciones sólo soportan HTTP/2 sobre TLS**, así que es obligatorio de facto.

Críticas:

- **Coste computacional** mayor, **negociación** necesaria (más RTTs). Muchas aplicaciones actuales no necesitan cifrado.
- La arquitectura actual de las **autoridades certificadoras jerarquizadas** no gusta a mucha gente.
- Hasta ahora tener un certificado **costaba dinero**. Aunque han aparecido sitios donde obtener certificados gratuitos como letsencrypt.org

BUENAS PRÁCTICAS

¡TODO LO CONTRARIO QUE EN HTTP/1.1!

Dispersar todos los recursos en **archivos pequeños**.

- Ahora podemos enviar archivos pequeños bajo demanda sin penalización adicional.
- La modularización del código con los **import** de ES6 lo favorece.

El **sharding** de dominios es ahora **contraproducente**.

- Ahora tenemos una sola conexión eficiente y sin limitaciones ni sobrecarga de cabeceras.

La **multiplexación** deja **obsoleto** al ***pipelining***.

SOPORTE DE HTTP/2

EN SERVIDORES

Servidor	Versión	Comentarios
Apache	2.4.12	Con el módulo <code>mod_h2</code> y unos cuantos parches
	2.4.17	Con el módulo <code>mod_http2</code>
Apache Tomcat	8.5	
Microsoft IIS	10	Windows 10 y Windows Server 2016
nginx	1.9.5	
node.js	5.0	
lighttpd	-	Sin soporte planificado al menos en la rama 1.x

SOPORTE DE HTTP/2

EN NAVEGADORES

Navegador	Versión	Comentarios
Chrome	41	
Firefox	36	
Opera	28	
IE / EDGE	12	Sólo en Windows 10
lighttpd	9	Sólo en OSX 10.11

¡Todas las implementaciones anteriores son con **TLS obligatorio!**

IDENTIFICACIÓN DE HTTP/2

EN EL NAVEGADOR Y EN WIRESHARK

▼ HyperText Transfer Protocol 2

▼ Stream: HEADERS, Stream ID: 1, Length 20

Length: 20

Type: HEADERS (1)

▼ Flags: 0x05

.... 1 = End Stream: True

.... 1 = End Headers: True

.... 0 = Padded: False

..0. = Priority: False

00.0 ..0. = Unused: 0x00

0... = Reserved: 0x00000000

.000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1

[Pad Length: 0]

Header Block Fragment: 8682418aa0e41d139d09b8f01e078453032a2f2a

[Header Length: 100]

▶ Header: :scheme: http

▶ Header: :method: GET

▶ Header: :authority: localhost:8080

▶ Header: :path: /

▼ Header: accept: */*

Name Length: 6

Name: accept

Value Length: 3

Value: */*







Representation: Literal Header Field with Incremental Indexing – Indexed Name

Index: 19

common frame header

HPACK encoded headers

- **Wireshark** descodifica y muestra los campos de cada trama en el **mismo orden** que se supone que se han transmitido, comparando con la **plantilla** de la trama.
- En **Chrome** hay que activar la columna **Protocol** y buscar **h2**.

Name	Method	Status	Protocol	Type	Initiator
 google.es	GET	301	http/1.1	text/html	Other
 www.google.es	GET	302	http/1.1	text/html	http://google.es/
 ?gws_rd=ssl	GET	200	h2	document	http://www.google.es/
 nav_logo242_hr.png	GET	200	h2	png	?gws_rd=ssl:51
 googlelogo_color_272x92dp.png	GET	200	h2	png	?gws_rd=ssl:51
 i2_2ec824b0.png	GET	200	h2	png	?gws_rd=ssl:51

ADOPCIÓN GENERALIZADA

¿PARA CUÁNDO TODO ESTO?

También nos dijeron que IPv6 iba a estar a pleno rendimiento en la actualidad, pero...

- requiere actualizar no sólo los extremos, sino todos los routers y equipos electrónicos intermedios
- se implementa a nivel de SO

¡Muy costoso de actualizar!

Sin embargo, HTTP/2...

- protocolo de aplicación que funciona sobre el omnipresente TCP
- sólo afecta a cliente y servidor
- es sólo software

¡Muy fácil de actualizar!

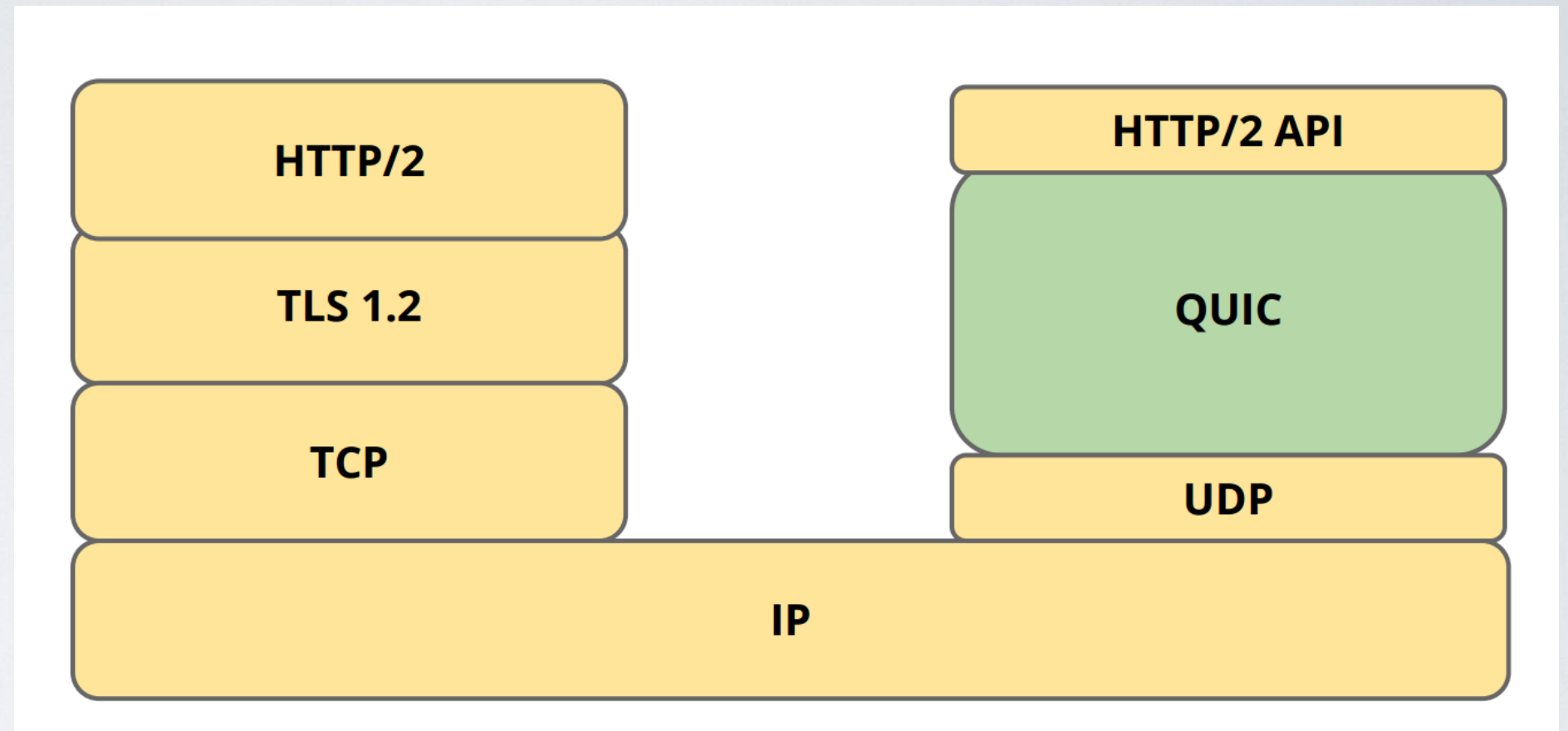
POSIBLE EVOLUCIÓN: QUIC

QUICK UDP INTERNET CONNECTION

HTTP sobre UDP

Capa de **transporte**
experimental (Google,
2012)

Borrador enviado a la
IETF para estandarización



- Detección y corrección de errores en el destino mediante **FEC** (Forward Error Correction)

El control de errores y de flujo de TCP se sacrifican para mejorar la latencia.

Cifrado SSL / TLS integrado.

BIBLIOGRAFÍA

Ilya Grigorik, High Performance Browser Networking

<https://www.nginx.com/http2-ebook/>

Ilya Grigorik, SPDY, err... HTTP 2.0

<https://www.igvita.com/slides/2012/http2-spdy-devconf.pdf>

Giuseppe Ciotta, The HTTP/2 protocol: kill that latency!

<https://www.youtube.com/watch?v=5udwe18ylgc>

RFC, Hypertext Transfer Protocol Version 2 (HTTP/2)

<https://www.rfc-editor.org/rfc/rfc7540.txt>

¡FIN!

¿Preguntas?