APUNTES, MANUALES, PRESENTACIONES

Universidad de Jaén

Escuela Politécnica Superior de Linares

Tema 2. Protocolo HTTP y el servicio www

Autor: Juan Carlos Cuevas Martínez

Fecha: 10/2024

Asignatura: Protocolos de transporte

Tema 2. Protocolo HTTP y el servicio www

El servicio web a través de su protocolo fundamental: HTTP

Asignatura: Protocolos de Transporte
Grado en Ingeniería Telemática

Autor:
Juan Carlos Cuevas Martínez

Dpto. de Ingeniería de Telecomunicación
Área de Ingeniería Telemática
Universidad de Jaén
Escuela Politécnica Superior de Linares

Versión 4.2
2024

Contenido

Bibliografía

Bibliografía básica

- Ludin S y Garza, J., **“Learning HTTP/2. A practical guide for beginners”**. O’Reilly, 2017.
- Pollard B., **“HTTP/2 in action”**. Manning, 2019.
- Grigorik I., **“High Performance Browser Networking”**. O’Reilly, 2013. Disponible en: https://hpbn.co/.
- TANENBAUM, Andrew S. Computer Networks, Global Edition. Pearson Universidad, 2021. ISBN 9781292374062.
- Forouzan, B., **“TCP IP Protocol Suite”**, 4ª Ed, Mc Graw Hill.
- Kozierok, C., **“The TCP IP guide: a comprehensive , illustrated internet protocols reference”**, No Starch Press.
- Reference For Comments (**RFC**).

Bibliografía

Bibliografía complementaria

- Kurose, J., Ross, K., **“Redes de Computadores. Un enfoque descendente basado en Internet”**. 4ª Edición, Pearson Education.
- Comer, D., **“Computer networks and internets”**, Prentice Hall.
- Stallings W. **“Comunicaciones y redes de computadores”**, 7ª Edición. Prentice Hall.

1. Introducción

Principales componentes funcionales de la Web

- La World Wide Web, WWW o simplemente Web es un fenómeno que pocos pudieron prever.
- Todo nació del hiper-texto: simplemente una forma de enlazar unos documentos con otros y transferirlos.
- Conjuntamente al éxito de las redes TCP/IP ha llevado a lo que es hoy en día.
- Así pues los componentes funcionales más importantes de la web son:
  - Hypertext Markup Language (HTML).
  - Hypertext Transfer Protocol (HTTP).
  - Uniform Resource Identifiers (URIs).

1. Introducción

Principales componentes funcionales de la Web

- **Hypertext Markup Language (HTML):**
  - El lenguaje HTML está formado por texto y marcas sencillas llamadas etiquetas (tags).
  - Permite enlazar documentos, así como de dotarles de estructura y de la capacidad de incorporar todo tipo de contenido.
  - HTML se ha convertido en el estándar de hiper-texto, así como ha propiciado la aparición de otros lenguajes, como XML, XHTML, etc.
- **Hypertext Transfer Protocol (HTTP):** HTTP es el protocolo de aplicación de la torre TCP/IP que implementa el servicio Web a través de la transferencia de los documentos HTML y de otros muchos tipos.
  - HTTP comenzó como un protocolo muy simple para enviar solamente documentos HTML.
  - En la actualidad posibilita gran variedad de funciones sofisticadas como la transferencia de todo tipo de documentos, streaming de varios ficheros por una conexión, uso de proxy, acceso a recursos con autenticación y caché, entre otras.

1. Introducción

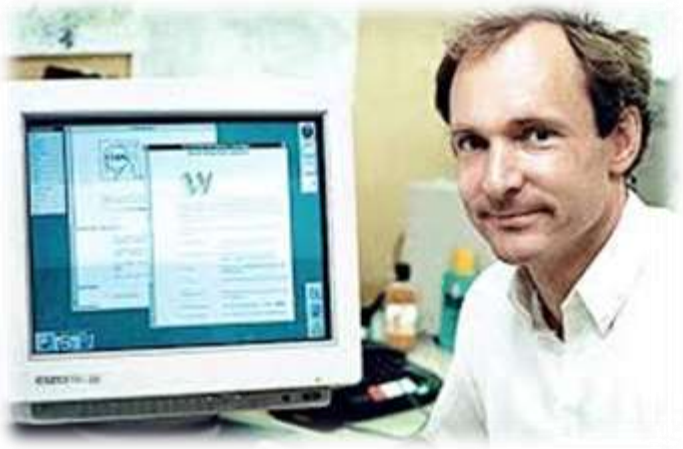
Principales componentes funcionales de la Web

- **Uniform Resource Identifiers (URIs):** Los URIs son usados para definir etiquetas que identifiquen de manera global y unívoca cualquier recurso en Internet.
  - Actualmente los URI no se limitan al servicio Web.
  - Existen dos categorías: los Uniform Resource Locators (URLs) y los Uniform Resource Names (URNs).
  - Tras la aparición de los URIs, los URLs y los URNs pasaron a integrarse como URIs
- Además, como parte fundamental también están:
  - Servidores WEB.
  - Navegadores WEB.
  - Los usuarios.

1. Introducción

Hypertext Markup Language (HTML)

- Inventado en 1990 por el inventor de la web Tim Berners-Lee.
- Es una aplicación del Standard Generalized Markup Language (SGML) que es el estándar ISO 8879:1986.
- HTML se organiza a través de la inclusión de etiquetas (tags) que deben tener un formato concreto en texto plano y que van encerradas entre los símbolos '<' y '>'.
  - Ejemplo <html>,<title> o
- Correspondiéndose normalmente una etiqueta de cierre </…> para una de las anteriores (esto es obligado en XHTML).
  - Ejemplo: Esto es un párrafo



```
<html>
<title>Hello</title>
<body>
<p>Hello</p>
</body>
</html>
```

1. Introducción

¿Qué es una aplicación web?

La respuesta evoluciona en función de las prestaciones.

- **En un principio:** conjunto de documentos, imágenes, etc., relacionados entre sí a través de hipervínculos.
- **Posteriormente:** Junto a los documentos HTML, imágenes, etc., aparecen programas que realizan una determinada función y que son ajenos al servidor http → concepto de aplicación de servidor
- Por su parte, los navegadores evolucionan, permitiendo la ejecución de aplicaciones:

Aplicaciones en el cliente + aplicaciones en el servidor

1. Introducción

¿Qué es una aplicación web?

- Las aplicaciones web se fundamentan en ideas y protocolos como:
  - CLIENTE-SERVIDOR.
  - TCP/IP (HTTP).
  - MIME.
  - LENGUAJE DE MARCAS (HTML, XML).
- Necesidad de un direccionamiento para identificar un recurso.
  - Para ello se introduce el *Indentificador de Recurso Uniforme* (URI).

2. Direccionamiento

- El direccionamiento a nivel de aplicación puede ser visto como redundante: con tan sólo con una dirección IP y un puerto ya se puede acceder a un servicio.

¿Y entonces?…

- Realmente existe una necesidad añadida:
  - ¿Cómo podemos referenciar o acceder a recursos concretos (texto, imágenes, vídeos, etc.) que están disponibles a través de una aplicación?
- Se necesita una dirección de aplicación:
  - IP: Máquina
  - Proceso: Servidor web
  - Recurso: URI

2. Direccionamiento

- Así, paralelo al crecimiento de la web y el desarrollo de HTTP, surgen los Identificadores Uniformes de Recurso o **URIs (Uniform Resource Identifier)**.
- Los **URI** permiten identificar de manera unívoca a un recurso del servicio web sin necesitar enviar varias instrucciones.
- Aunque surgieron de la tecnología web, se usan hoy en día en muchas otras aplicaciones y en sí han cobrado importancia en sí mismos como un ente separado de la web y HTTP.

2. Direccionamiento

Ficha Uniform Resource Identifier (URI): Generic Syntax

- RFC 3986 Uniform Resource Identifier (URI): Generic Syntax
  - Estándar 66.
  - Autores: T. Berners-Lee, R. Fielding, L. Masinter.
  - Fecha: Enero 2005.
  - ASCII.
  - Deja obsoletas: RFC 2732, RFC 2396 y RFC1808.
  - Actualiza: RFC1738.
  - La actualizan:
    - RFC 6874, Representing IPv6 Zone Identifiers in Address Literals and Uniform Resource Identifiers, febrero 2013.
    - BCP 190 RFC 8820 URI Design and Ownership, June 2020.

BCP: BEST CURRENT PRACTICE

2. Direccionamiento

Categorías de URIs

- **Uniform Resource Locators (URLs):** Una URL es una URI que permiten localizar un recurso completamente en una red, a través de la indicación del protocolo de comunicaciones y cualquier otro aspecto necesario, tal como el host, usuario y clave, ruta, nombre e incluso parámetros de funcionamiento.
- **Uniform Resource Names (URNs) [RFC8141]:** Un URN es una forma global y única de nombrar a un recurso sin especificar la forma en la que se accede a él ni donde está.

La RFC 3986 establece que en las especificaciones se debe emplear URI en vez de los otros dos términos, los cuales son más restrictivos.

2. Direccionamiento

URL (Uniform Resource Locators)

Formato

- Definido inicialmente en la RFC 1738, Uniform Resource Locators (URL) (diciembre 1994).
- Esta RFC ha sufrido diversas modificaciones y actualizaciones, las cuales han afectado a veces a solo una parte de la misma:
- La dejan obsoleta:
  - RFC 4248. The telnet URI Scheme, octubre 2005
  - RFC 4266. The gopher URI Scheme, noviembre 2005
- La actualizan
  - RFC 1808, dejada obsoleta también por la RFC 3986.
  - RFC 2368, dejada obsoleta por la RFC 6068 The 'mailto' URI Scheme, octubre 2010.
  - RFC 2396, dejada obsoleta también por la RFC 3986.
  - **RFC 3986, Uniform Resource Identifier (URI): Generic Syntax, enero 2005**
  - RFC 6196, Moving mailserver: URI Scheme to Historic, marzo 2011
  - RFC 6270, The 'tn3270' URI Scheme, junio 2011

2. Direccionamiento

URL (Uniform Resource Locators)

Formato

- La sintaxis genérica de los URI consiste en una secuencia jerárquica de componentes:
  - Esquema (scheme).
  - Autoridad (authority)
  - Ruta (path)
  - Petición (query)
  - Fragmento (fragment)

- **RFC 3986, Uniform Resource Identifier (URI): Generic Syntax, enero 2005**
  - Actualizada por las RFC6874, RFC7320 y RFC8820.

2. Direccionamiento

URL (Uniform Resource Locators)

Sintaxis ABNF

```
URI = scheme ":" hier-part[ "?" query ][ "#" fragment ]

hier-part = "//" authority path-abempty
 / path-absolute
 / path-rootless
 / path-empty

authority = [ userinfo "@" ] host [ ":" port ]

userinfo = *( unreserved / pct-encoded / sub-delims / ":" )
```

2. Direccionamiento

URL (Uniform Resource Locators)

Formato

<scheme>://<user>:<password>@<host>:<port>/<url-path>?<query>#fragment

- `scheme`: define el plan (URL scheme) de acceso al recurso, normalmente el protocolo a usar.
- `user` y `password`: es una aplicación de `userinfo` [RFC3986 3.2.1], no se debe emplear, están declarados obsoletos.
- `host`: lugar en la red, normalmente un nombre de dominio o una dirección IP.
- `port`: puerto a través del cual se accede al servicio.

2. Direccionamiento

URL (Uniform Resource Locators)

Formato

- `url-path`: camino del recurso dentro del Host.
- `query`: información adicional opcional aportada al servidor cuando el recurso es accedido. Se pueden enviar varias separándolas por el carácter '**&**'.
- `fragment`: identifica una parte concreta del recurso a la que se quiere acceder u otro tipo de representación alternativa. Realmente no es parte de la URL y no se envía al servidor. Lo retiene el programa cliente para saber como organizar la información a mostrar. El fragmento se marca con el carácter '#'.

Ejemplos

- ftp://usuario:pwd@192.168.1.100/archivo.exe
- https://platea.ujaen.es/course/view.php?id=5763

2. Direccionamiento

URL (Uniform Resource Locators)

Caracteres reservados, no reservados e inseguros en las URIs

- Las URIs se forman, como se ha visto, conforme a una estructura formada por caracteres delimitados por otros caracteres especiales.
- Caracteres reservados: sirven como delimitadores, si un carácter entra en conflicto con un delimitador, no siendo su función cumplir con la estructura de la URI se debe emplear su codificación con '%' y el código ASCII en hexadecimal .

```
reserved    = gen-delims / sub-delims
gen-delims  = ":" / "/" / "?" / "#" / "[" / "]" / "@"
sub-delims  = "!" / "$" / "&" / "'" / "(" / ")"
                  / "*" / "+" / "," / ";" / "="
```

- Los caracteres no reservados son aquéllos que están permitidos en la URI pero no tienen reservado un propósito concreto:

```
unreserved  = ALPHA / DIGIT / "-" / "." / "_" / "~"
```

2. Direccionamiento

URL (Uniform Resource Locators)

Caracteres reservados, no reservados e inseguros en las URIs

- Algunos caracteres, si se emplearan para formar una URI, podrían romper la coherencia de la misma.
- Estos caracteres se denominan caracteres inseguros y se sustituyen por el símbolo '**%**' seguido por el código ASCII en hexadecimal.
  - Por ejemplo el espacio en blanco no está permitido y se sustituye por %20 (20h = 32 que es el código ASCII del espacio en blanco).

```
pct-encoded = "%" HEXDIG HEXDIG
```

2. Direccionamiento

URL (Uniform Resource Locators)

Tabla con los códigos ASCII de caracteres inseguros para las URLs

Carácter	Codificación	Carácter	Codificación	Carácter	Codificación
<space>	%20	<	%3C	>	%3E
#	%23	%	%25	{	%7B
}	%7D	|	%7C	\	%5C
^	%5E	[	%5B	]	%5D
&	%26	`	%60	;	%3B
/	%2F	?	%3F	:	%3A
@	%40	=	%3D		

2. Direccionamiento

URL (Uniform Resource Locators)

URL del protocolo HTTP

- Normalmente la URL para el servicio web es muy sencilla, pero en su mayor detalle sería como sigue:

  http://<host>:<port>/<url-path>?<query>#<bookmark>

- El puerto para el esquema **http** es por defecto el **80**.
  - Para el esquema **https** (http sobre SSL/TLS) el puerto es el **443.**
- No se usan los parámetros, y las peticiones sirven para comunicar aspectos adicionales al servidor.
- Se han reemplazado los fragmentos por las referencias que se usan dentro de los documentos HTML.
- Generalmente una URL del servicio web es así de sencilla:

  http://<host>/<url-path>

3. Evolución del Protocolo HTTP

- Protocolo sencillo surgido de la necesidad de enviar y recibir documentos de hipertexto en el CERN que es el Instituto Europeo de Investigación Nuclear.
- La primera versión creada por estos investigadores se denominó posteriormente como HTTP/0.9.
- Aún hoy sigue siendo fundamentalmente un protocolo de pregunta/respuesta.
- Las versiones posteriores han sido la HTTP/1.0, la HTTP/1.1, la HTTP/2 y la HTTP/3.

3. Evolución del Protocolo HTTP

Versión HTTP/0.9

- La primera versión desarrollada (1991).
- No fue estandarizada.
- No tuvo una RFC o un número de versión.
- Era tan sólo una especificación de un par de páginas y que permitía enviar solamente documentos de hipertexto.
- La petición era una sola línea con el método (GET) y la ruta al recurso.
- La respuestas era un texto ASCII.
- No había códigos de error (eran parte del texto legible).
- No permitía cabeceras en las peticiones.

3. Evolución del Protocolo HTTP

Versión HTTP/1.0

- Durante los primeros años de uso de la primera versión, muchas ideas fueron surgiendo dando lugar a HTTP/1.0.
- La versión HTTT/1.0 aunó las características más comunes y significativas de la evolución que había existido en los primeros años de la WorldWideWeb (*en un principio WorldWideWeb no se separaba*).
- El primer documento publicado fue en mayo de 1996 en la RFC 1945 (aunque realmente es un memorándum)
  - Esta especificación se usó durante años antes de su publicación en la RFC 1945.
- Se convirtió en un protocolo de envío de mensajes completo:
  - Especificación de las peticiones de cliente y respuestas de servidor.
  - Nuevos métodos: HEAD y POST.
  - Se añadieron los códigos de estado a las respuestas: 1xx, 2xx, 3xx, 4xx y 5xx.
  - Peticiones condicionales: permitían no descargar un recurso si no había cambiado en el servidor.
  - Soporte a todo tipo de contenido (adaptó ciertos aspectos de las extensiones MIME diseñadas para el correo electrónico).

3. Evolución del Protocolo HTTP

Versión HTTP/1.0

Limitaciones

- Sin embargo dado al auge de Internet en la segunda mitad de los años 90 del siglo XX pronto quedaron de manifiesto sus limitaciones:
  - Cada sitio web tenía que estar en un servidor diferente (contar con una IP diferente o un puerto diferente, lo cual era inviable para un servicio global).
  - Cada petición (conexión TCP) tan sólo podía obtener un recurso web a la vez.
  - Falta de soporte para características de mejora del funcionamiento como caché, proxy y descarga parcial de recursos.

3. Evolución del Protocolo HTTP

Versión HTTP/1.1

- Para solucionar los problemas de HTTP/1.0 y adaptarse al creciente uso de la web, el primer borrador apareció en la RFC 2068 en enero de 1997.
- Fue revisado y posteriormente publicado en junio de 1999 en la RFC 2616 como “Hypertext Transfer Protocol –HTTP/1.1”.
- Mantiene compatibilidad hacia atrás con las versiones HTTP/1.0 y HTTP/0.9.
- Esta versión se acompaña de la RFC 2617 que versa sobre seguridad y autenticación.

3. Evolución del Protocolo HTTP

Versión HTTP/1.1

Evolución de las RFC

- La RFC 2616 quedó obsoleta tras la publicación de las siguientes RFCs en junio de 2014:
  - RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing
  - RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content
  - RFC 7232: Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests
  - RFC 7233: Hypertext Transfer Protocol (HTTP/1.1): Range Requests
  - RFC 7234: Hypertext Transfer Protocol (HTTP/1.1): Caching
  - RFC 7235: Hypertext Transfer Protocol (HTTP/1.1): Authentication
- Este trabajo ha sido llevado a cabo por el grupo **Hypertext Transfer Protocol Bis (httpbis)** del **IETF** que fuera iniciado en **2006.**

3. Evolución del Protocolo HTTP

Versión HTTP/1.1

Evolución de las RFC

- Se complementa con las siguientes nuevas RFCs:
  - RFC7236: Initial HTTP Authentication Scheme Registrations
  - RFC7237: Initial HTTP Method Registrations
- Otras especificaciones relacionadas:
  - RFC7238: The HTTP Status Code 308 (Permanent Redirect)
  - RFC7239: Forwarded HTTP Extension
  - RFC7240: Prefer Header for HTTP

3. Evolución del Protocolo HTTP

Versión HTTP/1.1

Evolución de las RFC

- En 2022, tras la estandarización de HTTP/3 se vuelven a unificar muchas de las RFCs anteriores en una sola la RFC 9110 HTTP Semantics, además de aparecer otras actualizaciones y una nueva RFC para HTTP/1.1.
- RFC 9110 HTTP Semantics
  - Deja obsoletas:
    - RFC 2818: HTTP Over TLS.
    - RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.
    - RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.
    - RFC 7232: Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests.
    - RFC 7233: Hypertext Transfer Protocol (HTTP/1.1): Range Requests.
    - RFC 7235: Hypertext Transfer Protocol (HTTP/1.1): Authentication.
    - RFC 7238: The Hypertext Transfer Protocol Status Code 308 (Permanent Redirect).
    - RFC 7615: HTTP Authentication-Info and Proxy-Authentication-Info Response Header Fields.
    - RFC 7694: Hypertext Transfer Protocol (HTTP) Client-Initiated Content-Encoding.
  - Actualiza:
    - RFC 3864: Registration Procedures for Message Header Fields
- RFC 9111 HTTP Caching: deja obsoleta la RFC 7234 Hypertext Transfer Protocol (HTTP/1.1): Caching.

3. Evolución del Protocolo HTTP

Versión HTTP/1.1

Evolución de las RFC

- La nueva especificación de HTTP/1.1 es la **RFC 9112 HTTP/1.1**, que deja obsoleta partes de la RFC 7230.
- Por lo tanto, la nueva especificación de HTTP/1.1 queda conformada por:
  - **RFC9110 HTTP Semantics.**
  - **RFC9111 HTTP Caching.**
  - **RFC9112 HTTP/1.1.**

3. Evolución del Protocolo HTTP

Versión HTTP/1.1

Mejoras introducidas en la versión HTTP/1.1

- **Soporte de Múltiples nombres de host:** En la versión HTTP/1.0 no había manera de especificar el nombre del host al que quería conectarse el cliente.
  - Cada servidor web tenía que estar en una IP diferente.
  - Se consumían demasiadas direcciones IP para el servicio.
  - Con HTTP/1.1 se permite a un servidor web manejar peticiones para un gran número (centenares) de servidores virtuales diferentes (se consigue gracias a la cabecera **`host`**).

3. Evolución del Protocolo HTTP

Versión HTTP/1.1

Mejoras introducidas en la versión HTTP/1.1

- **Conexiones persistentes:** HTTP/1.1 permite a un cliente enviar múltiples peticiones para obtener recursos del servidor en una sola conexión TCP.
  - Con HTTP/1.0 cada petición tenía que usar una conexión TCP diferente.
  - Las peticiones se resuelven (responden) en el mismo orden que se han hecho.
- **Selección de recurso parcial:** En HTTP/1.1 un cliente puede solicitar solamente una parte de un recurso sin tener que descargarlo en su totalidad.
  - Mejora considerablemente el aprovechamiento del ancho de banda, sobre todo en la descarga de ficheros de gran tamaño.
  - Esto se consigue gracias a la cabecera **`range`**.
- Implementación de la cabecera **Transfer-Encoding**: establece la forma en la que se envía el mensaje. Particularmente permite el envío de contenidos que no se sabe su longitud de antemano con el tipo “chunked” o “por trozos”.

3. Evolución del Protocolo HTTP

Versión HTTP/1.1

Mejoras introducidas en la versión HTTP/1.1

- **Mejora del uso de caché y proxy:** Se introducen mejoras para aumentar el rendimiento y seguridad de estas técnicas.
- **Negociación del contenido:** En HTTP/1.1 se permite tanto al cliente como al servidor intercambiar información para ayudar a seleccionar el recurso más apropiado, versión o variante del mismo.
  - Ejemplo que un cliente en un móvil pueda ver las páginas adaptadas a terminales móviles en vez de las que están diseñadas para equipos de escritorio, o si lo prefiere ver éstas últimas.
- **Mejora general en la seguridad de HTTP/1.1 frente a HTTP/1.0.**

3. Evolución del Protocolo HTTP

Versión HTTP/2

- En 2015 se lanzó la versión HTTP/2.0, fruto del trabajo del grupo HTTPbis del IETF.
- Su intención no es dejar obsoleta la versión HTTP/1.1, sino ofrecer una alternativa.
- Ya es toda una realidad y algunos de los servicios más usados ya van sobre HTTP/2: Google, Facebook, etc.

Fue publicado en la RFC 7540 “Hypertext Transfer Protocol Version 2 (HTTP/2)“ en mayo de 2015 y dejada obsoleta en 2022 por la RFC 9113.

3. Evolución del Protocolo HTTP

Versión HTTP/2

Origen

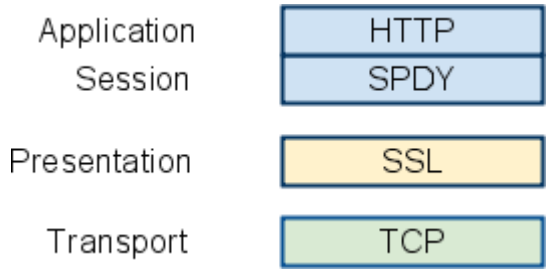
- HTTP/2 parte de los esfuerzos por mejorar el rendimiento del protocolo HTTP/1.1 en el nuevo entorno del servicio web e Internet del siglo XXI.
  - Incluso se intentó reemplazar TCP por SCTP como protocolo de transporte para aprovechar la capacidad de éste de gestionar varios flujos de datos.
- Problemas de HTTP/1.1:
  - Las peticiones se tienen que resolver en orden.
  - Cabeceras repetitivas y demasiado largas.
  - Las carencias de HTTP/1.1 obligaban a buscar soluciones “ingeniosas” para solventarlas (por ejemplo el “**domain sharding**”).

3. Evolución del Protocolo HTTP

Versión HTTP/2

Origen

- En 2012 se inició el trabajo en el grupo de trabajo de HTTP para crear HTTP/2 partiendo de SPDY (o Speedy), desarrollado por Google para el navegador Chrome, que tenía muy buenas innovaciones:
  - Compresión de cabeceras
  - Varios flujos simultáneos.
  - Priorización de peticiones.
  - Mensajes de servidor.



Torre de protocolos de SPDY
Fuente: www.chromium.org/spdy/spdy-whitepaper

3. Evolución del Protocolo HTTP

Versión HTTP/2

- Algunas de las nuevas funcionalidades que aporta son:
  - Mezcla de mensajes de petición y respuesta en la misma conexión.
  - Codificación eficiente de las cabeceras.
  - Priorización de peticiones, para conseguir que aquellas con más prioridad se envíen antes, aumentando la tasa de transferencia para éstas.
- En resultado usará menos conexiones TCP.

3. Evolución del Protocolo HTTP

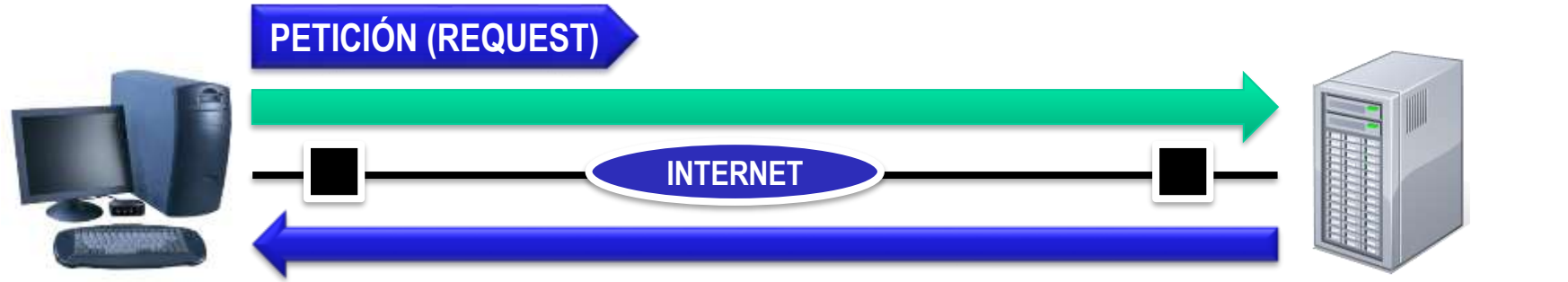
Versión HTTP/3

- Es fruto del grupo de trabajo QUIC del IETF.
- Surge de la necesidad de conseguir una menor latencia en las comunicaciones y solventar algunos problemas que presenta TCP cuando se pierde un segmento (***Head of line blocking***): nace QUIC.
- HTTP/3 comienza como HTTP-over-QUIC. Es en 2018 cuando empieza a denominarse como en la actualidad.
  - El primer borrador de HTTP-over-QUIC data de noviembre de 2016.
- Hoy en día es ya soportado en la mayoría de navegadores.

4. Características generales del protocolo HTTP

Modelo de operación y comunicación Cliente/Servidor

- HTTP es un protocolo cliente/servidor basado en peticiones y respuestas.
- Su objetivo principal es intercambiar **recursos** (no hay limitación en cuanto qué considerar un recurso), la mayoría de ellos identificados mediante una URI.



4. Características generales del protocolo HTTP

Modelo de operación y comunicación Cliente/Servidor

- **Petición del cliente:** Las peticiones del cliente son formateadas según la especificación de HTTP en un **mensaje de petición**:
  - Incluyen la información necesaria para identificar la operación a realizar y el recurso deseado.
  - Parámetros adicionales necesarios para el servidor: **campos** (*fields*) enviados en la cabecera, también conocidos como **cabeceras** (*header fields*).
  - Pueden, en ciertos casos, incluir también un recurso.
- **Respuesta del servidor:** El servidor interpreta la petición del cliente, la procesa, toma las acciones relevantes para dicha petición y elabora un **mensaje de respuesta**. El mensaje de respuesta contendrá:
  - Un código que indicará si la petición ha tenido éxito o no.
  - Nuevos campos para completar la respuesta y/o el recurso incluido.
  - Generalmente, un recurso si la petición así lo solicitó y ésta ha podido ser atendida correctamente por el servidor.

4. Características generales del protocolo HTTP

Modelo de operación y comunicación Cliente/Servidor

Cabeceras de petición

- Cabeceras Generales:
  - Se refieren al mensaje en sí mismo, no dependen de si es una petición o una respuesta.
  - No dependen del tipo de recurso solicitado.
  - Aportan información de cómo debe ser procesada la petición o proporcionan información adicional.
- Cabeceras de Petición:
  - Aportan detalles de la petición del cliente.
  - Informan de aspectos más concretos de cómo se tiene que procesar la petición.
- Cabeceras de Entidad:
  - Describen la entidad contenida en el cuerpo de la petición si ésta existe.

4. Características generales del protocolo HTTP

Modelo de operación y comunicación Cliente/Servidor

Cabeceras de respuesta

- Cabeceras Generales:
  - Igual que en la petición, si bien algunas son más comunes de peticiones y otras de respuestas.
- Cabeceras de Respuesta:
  - Aportan información adicional que complementa a la línea de estado.
  - El servidor puede añadir información adicional, incluso en el cuerpo de la respuesta, aunque esto es más habitual en caso de error.
- Cabeceras de Entidad:
  - Describen la entidad contenida en el cuerpo de la petición si ésta existe.
  - Son las mismas que pueden aparecer en una petición, pero son más habituales en los mensajes de respuestas.
  - Es posible que en la respuesta haya cabeceras de entidad pero no esté presente el recurso, como ocurre cuando es una respuesta a una petición **HEAD**.

4. Características generales del protocolo HTTP

Modelo de operación y comunicación Cliente/Servidor con intermediarios

- HTTP soporta que entidades intermediarias cambien/modifiquen las peticiones en función de su tarea.
- Estas entidades pueden ser: **proxis**, **pasarelas (*gateways*)** o **túneles**.

4. Características generales del protocolo HTTP

Extensibilidad de HTTP

- Aparte de lo definido en las RFC estándar, HTTP tiene diversas formas de poder ser extendido sin que esto implique la modificación de estos documentos o la creación de una nueva versión.
  - Esto se debe a que la semántica del protocolo no es dependiente de la versión.
  - Puede ser extendido en métodos, códigos de estado, nombres de campos de cabecera o tráiler, e incluso, dentro de los valores de cabeceras existentes como los esquemas de autenticación y las directivas de caché.
  - Se desaconseja que las extensiones sean dependientes de una versión en concreto, pero se admite cuando es inevitable.

5. Métodos HTTP

- Existen 8 métodos, y siempre se deben escribir (y enviar) en mayúsculas.
- Los más usados son **GET**, **POST** y **HEAD**.
- Los otros cinco son: OPTIONS, PUT, DELETE, TRACE y CONNECT.
- Un servidor de propósito general debe dar soporte a **GET** y **HEAD**, los demás son opcionales.

Nota: El motivo de que se denominen métodos, y no comandos (lo cual a todos los efectos sería igual), es por la herencia de la definición del protocolo en la versión 1.0 (RFC 1945). Ésta define a HTTP en su introducción como: *«un protocolo genérico, sin estados, orientado a objetos el cuál puede ser usado para muchas tareas, …»* Así pues, de la **OOP** (*object-oriented programming*), se deriva que los objetos ejecutan su cometido a través de **métodos**.

5. Métodos HTTP

Métodos más usados

- **GET:** Solicita la transferencia desde el servidor HTTP de la representación actual del recurso objetivo especificado en el URI.
- **HEAD**: Como GET, salvo que el servidor **no envía** el recurso en concreto, tan sólo las cabeceras. Se usa para comprobar la existencia de recursos, estado o tamaño.
- **POST**: Permite al cliente enviar información al servidor, normalmente para que sea procesada por este o por alguna aplicación en dicho servidor. La URL contiene el recurso que debe recibir y procesar los datos enviados por el cliente.

5. Métodos HTTP

Métodos no habituales

- **OPTIONS**: Informa acerca de las capacidades de comunicación de un recurso, del servidor o de un intermediario. Si se especifica una URI, se obtendrá información relevante sobre cómo accederlo. Si se especifica un asterisco '*', la petición será sobre el servidor mismo.
- **PUT:** Permite enviar un recurso objetivo especificado en la URI al servidor (no es común, es más empleado en APIs REST), creándose o reemplazándose el recurso destino.
- **DELETE:** Permite borrar un recurso en el servidor (nada habitual, más empleado en API REST).

5. Métodos HTTP

Métodos no habituales

- **TRACE:** Permite al cliente solicitar una copia de la petición enviada al servidor, normalmente para diagnosis.
- **CONNECT:** Informa al receptor del método de que debe establecer un túnel con el servidor identificado en la URI. Se emplea normalmente para que un proxy se pueda convertir en un túnel dinámicamente, lo que también se puede asegurar con TLS. Ejemplo:

```
CONNECT server.example.com:80 HTTP/1.1
Host: server.example.com
```

5. Métodos HTTP

Métodos HTTP

Ejemplo: comando OPTIONS

```
OPTIONS * HTTP/1.1
HOST:WWW10.UJAEN.ES



HTTP/1.1 200 OK
Date: Thu, 03 Oct 2013 07:32:06 GMT
Server: Apache
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Length: 0
Content-Type: text/plain
```

5. Métodos HTTP

Propiedades comunes de los métodos

- La documentación de HTTP agrupa en una serie de categorías sus métodos, en función del impacto que pueden ocasionar en el servidor:
  - Métodos seguros.
  - Métodos idempotentes.
- El IANA mantiene una lista con todos los métodos en https://www.iana.org/assignments/http-methods/http-methods.xhtml.

5. Métodos HTTP

Propiedades comunes de los métodos

Métodos seguros

- Son aquellos cuyo comportamiento básico es esencialmente de **solo lectura**, por lo que se les presupone que su impacto en un servidor no es perjudicial (en un principio) ya que no pueden ocasionar una alteración de los recursos proporcionados por éste, ni producir efectos colaterales indeseados.
  - Los métodos seguros son **GET**, **HEAD**, **OPTIONS** y **TRACE**.
  - Por el contrario, los que solicitan que el servidor procese algo, o pueden conducir a cambios en el mismo, se denominan inseguros, como lo son **CONNECT**, **POST**, **PUT** y **DELETE**.
  - Que sean considerados inseguros no significa que los servidores no los permitan nunca (de hecho POST es un método ampliamente usado, y fundamental actualmente).

5. Métodos HTTP

Propiedades comunes de los métodos

Métodos idempotentes

- Un método es **idempotente** cuando la ejecución de éste, independientemente de las veces que se ejecute, arroja el mismo resultado que si se hubiera ejecutado una sola vez.
  - Inherentemente, **todos los métodos seguros, junto con PUT y DELETE** son idempotentes (quedan fuera CONNECT y POST).
  - **POST**, al requerir, normalmente, del procesado de cierta información por parte del servidor, puede generar diferentes respuestas para la misma petición.
  - Sin embargo, secuencias de métodos idempotentes pueden conllevar un comportamiento no idempotente (una secuencia es idempotente cuando la ejecución de ella al completo, o de una parte, siempre arrojará los mismos resultados). **Ejemplo:** PUT y GET sobre el mismo recurso.
  - Además, a veces, un método idempotente como un **GET** sobre un ***SCRIPT*** que modifica el servidor, puede llevar a un comportamiento no idempotente.

6. Códigos de estado y frases de respuesta

- Los códigos de estado están formados por tres dígitos (xyy).
  - El primero (x) indica el tipo de respuesta y los otros dos (yy) se agrupan para generar 100 posibles razones diferentes en ese grupo.
  - Los códigos de estado válidos van del 100 al 599, ambos inclusive, un código de estado fuera de este rango debe ser tratado como un 5xx (Server Error).
  - Todos los códigos están definidos en el Hypertext Transfer Protocol (HTTP) Status Code Registry del IANA (https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml).
- La frase de respuesta es para facilitar la interpretación del código de respuesta por humanos.
  - En el estándar se especifican algunos textos de ejemplo para cada respuesta.
  - Un administrador puede editar estos textos como desee.
  - Las versiones por encima de la HTTP/1.1 ya no envían frase de respuesta.
  - A veces, las respuestas de error pueden contener un recurso HTML donde se muestra el error, y posiblemente información adicional sobre el mismo. En la actualidad es habitual que estén presentes, pero es a discreción del administrador del servidor.

6. Códigos de estado y frases de respuesta

Ejemplo de configuración – fichero httpd.conf de Apache

```
# Customizable error responses come in three flavors:
# 1) plain text 2) local redirects 3) external redirects
#
# Some examples:
#ErrorDocument 500 "The server has a problem."
ErrorDocument 404 /404.html
#ErrorDocument 404 "/cgi-bin/missing_handler.pl"
#ErrorDocument 402 http://localhost/subscription_info.html
#
```

Esta es la opción activa

6. Códigos de estado y frases de respuesta

Códigos de estado – Interpretación del primer dígito

FORMATO	SIGNIFICADO	DESCRIPCIÓN
1yy	Mensaje de información	Informa de que la petición se ha recibido y el proceso continua o es la respuesta parcial del estado de una conexión. No puede llevar ni contenido, ni trailers.
2yy	Éxito	El método fue aceptado, comprendido y aceptado por el servidor.
3yy	Redirección	La petición no ha fallado, pero necesita más información para poder ser completada.
4yy	Error de cliente	La petición es inválida, tiene una mala sintaxis o no puede ser completada por alguna razón, sin embargo el servidor estima que es un fallo del cliente.
5yy	Error de servidor	La petición es correcta pero no puede ser completada debido a un fallo del propio servidor.

6. Códigos de estado y frases de respuesta

Códigos de estado – Ejemplos

CÓDIGO	FRASE	DESCRIPCIÓN
100	Continue	*Indica que el servidor ha recibido la parte inicial de la petición y que aún ésta no ha sido rechazada. El servidor intentará enviar una respuesta final cuando la petición haya sido recibida y procesada completamente.*
101	Switching protocols	*El cliente solicita a través de la cabecera Update el uso de un protocolo alternativo y el servidor está de acuerdo.*
200	OK	*Es la respuesta genérica de operación realizada con éxito. El contenido en una respuesta con 200 depende de la petición.*
201	Created	*La petición ha sido realizada correctamente y como consecuencia se ha creado un nuevo recurso. Es la respuesta típica al método PUT cuando el recurso no existía y se crea.*
206	Partial Content	*El servidor ha completado satisfactoriamente una petición GET parcial que empleó la cabecera Range.*

6. Códigos de estado y frases de respuesta

Códigos de estado – Ejemplos

CÓDIGO	FRASE	DESCRIPCIÓN
300	Multiple choices	*El recurso solicitado está representado de más de una manera en el servidor. En la respuesta el servidor informa de las diferentes representaciones para que el cliente pueda elegir (parte de la negociación guiada por el cliente).*
301	Moved Permanently	*El recurso solicitado ha sido movido permanentemente a una nueva URL. Esta es la forma apropiada de informar de que un recurso se ha renombrado, y no que se muestre el error 404. El servidor informaría de la nueva ubicación con la cabecera Location.*
302	Found	*Indica que el recurso reside temporalmente en una URI diferente. Dado que la redirección podría ser alterada ocasionalmente, el cliente debería mantener la URI en peticiones futuras.*
304	Not Modified	*Es una respuesta típica a peticiones condicionales no llevadas a cabo, como cuando el cliente envía un GET condicionado para obtener el recurso solamente si ha cambiado. Con este código el servidor no envía de nuevo el recurso al no haber sido modificado.*
308	Permanent Redirect	*Informa de que el recurso solicitado se ha movido permanentemente a una nueva URI con la cabecera Location y en peticiones futuras debería usarse dicha nueva URI.*

6. Códigos de estado y frases de respuesta

Códigos de estado – Ejemplos

CÓDIGO	FRASE	DESCRIPCIÓN
400	Bad Request	*Es una respuesta genérica empleada cuando la petición de cliente no se entiende o no se puede llevar a cabo debido a un problema en el lado del cliente.*
401	Unauthorized	*El cliente no está autorizado para acceder a un recurso. Normalmente este es el caso de cuando el recurso está protegido con una clave o por cualquier otro tipo de credenciales. El servidor generará una cabecera WWW-Authenticate en la respuesta.*
402	Payment Required	*Reservado para uso futuro.*
403	Forbidden	*La petición ha sido entendida por el servidor pero el servidor rechaza el llevarla a cabo. En la respuesta el servidor puede describir el motivo de la prohibición. Puede ser una respuesta a una petición de autenticación con información insuficiente.*
404	Not Found	*El error más común en HTTP. Se devuelve cuando el servidor no puede encontrar el recurso solicitado, ya sea porque se ha movido, no existe o el usuario puso mal la URL.*
405	Method Not Allowed	*El método solicitado no está permitido para ese recurso en concreto. En la respuesta se incluye la cabecera Allow que indica los métodos que el servidor permite.*

6. Códigos de estado y frases de respuesta

Códigos de estado – Ejemplos

CÓDIGO	FRASE	DESCRIPCIÓN
500	Internal Server Error	*Es un mensaje genérico de error que indica que la petición no puede ser completada debido a un problema inesperado del servidor.*
501	Not Implemented	*El servido no soporta la funcionalidad requerida para llevar a cabo la petición recibida, por lo que no podrá atenderla.*
502	Bad Gateway	*El servidor, actuando como proxy o pasarela, recibe una respuesta errónea del otro servidor que intentó acceder en nombre del cliente.*
503	Service Unavailable	*El servidor no podrá atender la petición temporalmente, probablemente debido a que esté sobrecargado o en mantenimiento.*
504	Gateway Timeout	*La espera por la respuesta ha expirado cuando el servidor, actuando como proxy o pasarela, ha solicitado un recurso de un servidor remoto en nombre del cliente.*
505	HTTP Version Not Supported	*La petición enviada usa una versión de HTTP que el servidor no soporta o reúsa soportar.*

7. Cabeceras de HTTP

Cabeceras generales

- **Cache-Control:** Afecta a cómo se debe tratar una petición o respuesta por cualquier dispositivo en la cadena de peticiones.
  - Prevalecen sobre la configuración de caché establecida en el dispositivo.
  - Existen 7 para peticiones y 10 para respuestas. Pueden verse con más detalle en la RFC 9111.
  - Algunos valores son: no-cache o no-store.
- **Connection:** especifica cómo se tratará la conexión, el valor más típico es `Close` o también `Keep-Alive`.
- **Upgrade:** El cliente informa al servidor de la disponibilidad de otros protocolos de aplicación, pudiendo el servidor acordar el uso de estos.

7. Cabeceras de HTTP

Cabeceras generales

- **Transfer-Encoding:** Indica el tipo de codificación usada en el cuerpo del mensaje para que no haya errores en la interpretación.
- **Date:** fecha y hora en la que se originó el mensaje.
- **Trailer:** Complementa el uso del envío de datos en trozos (chunked data).
- **Via:** Introducida por los dispositivos que gestionan proxys, pasarelas o túneles para informar al receptor de su presencia.

7. Cabeceras de HTTP

Cabeceras de petición

- Son sólo usadas en las cabeceras de petición HTTP y permiten al cliente:
  - Proporcionar información sobre sí mismo.
  - Aportar detalles sobre la petición que está enviando.
  - Controlar mejor la manera en la que la petición será tratada y respondida.
- A continuación se verá una breve descripción de algunas de las cabeceras de petición.
- Otras cabeceras son: Expect, From, If-Match, If-Modified-Since, If-None-Match, If-Range, If-Unmodified-Since, Max-fordwards, Proxy-Authorization, Range, Referer, TE (*transfer encodings*) o User-Agent.

7. Cabeceras de HTTP

Cabeceras de petición

- **Host**: Indica el dominio y puerto del host destinatario de la petición. Es la única cabecera de obligatoria presencia en HTTP.
  - En HTTP/2 y HTTP/3 la cabecera **host** es a veces suplantada por la pseudo-cabecera **:authority**.
- **Accept:** El cliente informa qué tipos de contenido puede soportar.

Accept: text/html,application/xhtml+xml,application/xml;**q**=0.9,*/*;**q**=0.8

- **Accept-Charset:** Similar a la anterior, ya que especifica qué juego de caracteres espera recibir en las respuestas, en vez de un tipo de contenido concreto. También permite realizar el control de preferencia con el parámetro 'q'. Por defecto los clientes no la envían.

Accept-Charset: iso-8859-5, unicode-1-1;**q**=0.8

7. Cabeceras de HTTP

Cabeceras de petición

- **Accept-Encoding:** El cliente especifica qué tipo de codificación del contenido admite (normalmente comprimidas). Algunos ejemplos son: gzip, compress, deflate o sdch (Shared Dictionary Compression over HTTP de Google).
- **Accept-Language:** Como las anteriores pero para idiomas.

  Accept-Language: es-ES,es;**q**=0.8,en-US;**q**=0.5,en;**q**=0.3
- **Authorization:** Usada para proporcionar las credenciales de un usuario ante el requerimiento de autenticación del servidor (como al enviar al cliente un `401 No autorizado`).

7. Cabeceras de HTTP

Cabeceras respuesta

- Son solamente usadas en las cabeceras de respuesta HTTP y son tan sólo enviadas por servidores o intermediarios.
  - Proporcionan información adicional a la línea de estado.
  - Su uso está vinculado a la recepción de algunas peticiones concretas.
  - Existen nueve cabeceras.
- Algunos ejemplos son:
  - **Accept-Range:** indica si el servidor acepta peticiones para descargar un rango concreto de un recurso (valores habituales none o bytes).
  - **ETag:** Etiqueta de entidad generada para un recurso e incluida en la respuesta. Se usa para distinguir representaciones de los recursos, como por los clientes para hacer peticiones condicionales posteriores como con la etiqueta If-match.
  - Otras: Location, Proxy-Autenticate, Retry-After, Server, Vary y WWW-Autenticate.

7. Cabeceras de HTTP

Cabeceras de entidad

- Aportan información acerca del recurso enviado en el cuerpo del mensaje HTTP, el cual, en la especificación del protocolo en la RFC 2616, se denominó **entidad, en la nueva RFC 9110 se emplea el término representación**:
  - Una “representación” es información que pretende reflejar un estado pasado, actual o deseado de un recurso dado, en un formato que pueda comunicarse fácilmente a través del protocolo.
    - Una representación consiste en un conjunto de metadatos de representación y un flujo potencialmente ilimitado de datos de representación.
- Deben proporcionar la información necesaria para que la representación sea procesada correctamente por el receptor.
- Normalmente las representaciones están presentes en los mensajes de respuesta y son enviadas por el servidor, pero pueden estar presentes también en las peticiones POST o PUT.

7. Cabeceras de HTTP

Cabeceras de entidad

- **Allow:** Informa de todos los métodos que se pueden aplicar a un recurso.
- **Content-Encoding:** Codificación de la entidad (compresión).
- **Content-Language:** El lenguaje natural humano que usa la entidad.
- **Content-Length:** Tamaño en octetos de la entidad.
- **Content-Location:** URL donde se encuentra la entidad.
- **Content-Range:** rango de la entidad enviado cuando se envía por partes, por ejemplo los 100 bytes de un recurso de 500 se pondría: 0-99/500.
- **Content-Type:** tipo MIME de la entidad.
- **Last-Modified:** Fecha y hora de cuando cree el servidor que el recurso fue modificado por última vez.

7. Cabeceras de HTTP

Tipos y subtipos de contenido

- HTTP se convirtió a principio de la década de los 90 del siglo XX en un protocolo de Hiper-media, adoptando buena parte de la especificación MIME, entre ella, la definición de tipos de contenido. HTTP también soporta tipos MIME compuestos, como el `multipart`.

`<tipo>/<subtipo> [;parámetro1; parámetro 2…; parémetro N]`

Ejemplo: **`text/html;charset=utf-8`**

- Se usan habitualmente en la cabecera de entidad **Content-Type**, la cual está presente siempre cualquier mensaje que lleve una entidad.
- También se usan en la cabecera **Accept**, donde el cliente indica que tipos de entidades está dispuesto a aceptar.
  - Se puede usar el '*' para informar de que se admite cualquier tipo o subtipo o combinación de ambos.
  - Ejemplo: **`text/*`**, para cualquier entidad de tipo texto, o **`*/*`**, que simboliza cualquier entidad capaz de ser enviada.

7. Cabeceras de HTTP

Tipos y subtipos de contenido

Diferencias entre HTTP, MIME y RFC 822

- HTTP no incluye la cabecera **MIME-Version**.
- La cabecera **Content-Encoding** tiene una misión diferente en HTTP, indica el tipo de compresión usada, y en MIME indica el formato en el que se representa la información, como BASE64 o ASCII.
- HTTP no sigue tampoco estrictamente el formato “RFC 822”, ya que permite el envío de entidades en los mensajes HTTP en formato binario, lo cual es más eficiente que codificarlos en BASE64.
- Así pues, HTTP puede enviar entidades sin ninguna codificación, debiendo ser el cliente quien interprete el contenido a partir de lo que aparezca en la cabecera **Content-Type**.

7. Cabeceras de HTTP

Tipos y subtipos de contenido

Esquema de codificación de dos niveles de HTTP

- Para añadir flexibilidad HTTP aporta:
  - **Codificación de contenido:** Esta codificación se aplica a la entidad a enviar en un mensaje HTTP
    - Es extremo a extremo.
    - El formato usado se especifica en la cabecera **Content-Encoding**.
    - El cliente informa de los tipos que soporta con **Accept-Encoding**.
  - **Codificación de transferencia:** Su función es asegurar que todo el mensaje se transmite adecuadamente ente dispositivos.
    - Esta codificación se aplica a todo el mensaje HTTP, y no solo a una entidad.
    - La codificación no es extremo a extremo, sino salto a salto.
    - Se informa de la codificación en la cabecera **Transfer-Encoding**.
    - Normalmente se usa más para identificar el final de mensajes HTTP.

8. Funcionamiento del protocolo HTTP

Peticiones condicionales

- HTTP permite hacer peticiones condicionales en base a una serie de cabeceras.
- El objetivo principal de estas peticiones es hacer más eficiente la actualización de recursos, ya sea en el cliente como en el servidor.
- Se basan en la comparación del valor de las etiquetas de entidad generadas para cada recurso (intercambiadas con la cabecera ETag).
- Comparaciones:
  - **If-Match**: más empleada con POST, PUT y DELETE, la operación se lleva a cabo si la etiqueta de entidad enviada en la cabecera coincide con la del recurso seleccionado.
  - **If-None-Match**: principalmente empleada con GET, se realiza la operación si la etiqueta de entidad del recurso no coincide con la enviada en la cabecera.
  - **If-Modified-Since**: empleada con GET y HEAD, el servidor realiza la operación si la fecha de última actualización del recurso es posterior a la fecha enviada en esta cabecera.
  - **If-Unmodified-Since**: tiene una función similar a If-Match para cuando no se dispone de una etiqueta de entidad. Igualmente está pensada para POST, PUT y DELETE y la operación se lleva a cabo cuando la fecha aportada en la cabecera es más reciente que la del recurso.
  - **If-Range**: permite enviar, o una etiqueta de entidad, o una fecha, para obtener el rango deseado si el recurso no ha cambiado (coinciden la etiqueta o la fecha) o el recurso entero si éste ha cambiado.

8. Funcionamiento del protocolo HTTP

Negociación de contenido en HTTP

- Los recursos disponibles a través del protocolo HTTP pueden tener diversos formatos, lenguajes o codificaciones y representar, aún así, la misma información.
  - Así mismo, los agentes de usuario pueden tener diferentes capacidades, características o preferencias para elegir una representación de la información entre todas las disponibles.
- HTTP aporta mecanismos para que se pueda obtener el recurso que mejor se adapta a las necesidades del cliente (siempre que sea posible).
  - Si a pesar de enviar cabeceras de negociación de contenidos no se pueden satisfacer las necesidades del cliente, un servidor puede enviar en código de estado 406 (Not Acceptable), o enviar un contenido sin tener en cuenta las cabeceras (lo cual no implica que ese contenido pueda ser interpretado por el cliente).

8. Funcionamiento del protocolo HTTP

Negociación de contenido en HTTP

- Existen tres técnicas para la negociación de contenidos:
  - Negociación dirigida por el servidor o proactiva.
  - Negociación dirigida por el cliente o reactiva.
  - Negociación por contenido solicitado: el servidor envía en la respuesta a una petición las cabeceras `Accept` que estime oportunas para influenciar las subsecuentes peticiones al mismo recurso.

8. Funcionamiento del protocolo HTTP

Negociación de contenido en HTTP

Negociación dirigida por el servidor

- Esta es la recomendada y más usada actualmente.
- Es el servidor, a través de la información aportada por el cliente, quien elige el recurso más apropiado para éste último.
- Se apoya en las cabeceras de petición: **Accept**, **Accept-Charset**, **Accept-Encoding**, **Accept-Language** y **User-Agent**.
- El servidor usa también la cabecera **Vary** para informar del criterio usado para la elección del recurso enviado, así como informa a la caché de si esta respuesta es válida para siguientes peticiones.

8. Funcionamiento del protocolo HTTP

Negociación de contenido en HTTP

Negociación dirigida por el servidor

- El cliente informa a través de esas cabeceras sus preferencias a través del parámetro ‘q’ de calidad (*quality,* aunque no tiene relación directa con la calidad real del documento).
- El valor de q es un número real que va de **‘1’** (máximo), que indica máxima predilección (si se omite q se toma como valor ‘1’), a **‘0.001’** (mínimo) que informa de la mínima predilección. Un valor de **0** informa de la negativa a aceptar ese formato (no aceptable).

8. Funcionamiento del protocolo HTTP

Negociación de contenido en HTTP

Negociación dirigida por el servidor

Ejemplos

Accept-Language: es-ES,es;**q**=0.8,en-US;**q**=0.5,en;**q**=0.3,de;q=0

Accept: text/*; q=0.5, text/html, text/x-dvi; q=0.8, text/x-c

Accept-Charset: iso-8859-5, unicode-1-1; q=0.8 (está obsoleta y no se debe emplear dado que la codificación más común UTF-8 es usada prácticamente en todos los contenidos textuales)

Accept-Encoding: compress;q=0.5, gzip;q=1.0

Accept-Encoding: gzip;q=1.0, identity; q=0.5, *;q=0

Vary: accept-encoding, accept-language

8. Funcionamiento del protocolo HTTP

Negociación de contenido en HTTP

Negociación dirigida por el cliente

- Ante la petición de un recurso disponible en varios formatos el servidor envía una respuesta 300 (múltiples elecciones) con información de las variantes disponibles.
- El servidor puede devolver el mensaje 406 (No aceptación) si no dispone de una representación del recurso especificado según los criterios del cliente.
- El cliente envía una nueva petición con el recurso concreto.
- Es un proceso más selectivo y predecible, pero menos eficiente en cuanto a tiempo y comunicaciones, ya que necesita de dos mensajes extra.

8. Funcionamiento del protocolo HTTP

La caché en HTTP

- La caché es un almacén local (ya sea en el cliente o el servidor) de mensajes previos y un subsistema de HTTP que permite el almacenamiento, la recuperación y el borrado de mensajes en él.
- El funcionamiento de la caché en HTTP está definida en la RFC 9111 y aunque hoy en día es algo empleado en todos los navegadores, es una característica OPCIONAL de HTTP, aunque manifiestamente deseable.
- El objetivo fundamental de la caché es reutilizar información ya recibida en respuestas previas para **reducir el ancho de banda empleado por una comunicación y el tiempo de respuesta en la obtención de recursos**.
  - Una labor también importante de la caché es determinar qué se debe almacenar y qué no.

8. Funcionamiento del protocolo HTTP

La caché en HTTP

Funcionamiento de la caché

- Como mínimo, la cache almacena el **método** y la **URI** para recuperar una respuesta almacenada previamente (el conjunto de método y URI se denomina *cache key*).
  - En la actualidad, fundamentalmente, la caché almacena respuestas con **código de estado 200** del método **GET**.
    - Aunque también es posible almacenar respuestas negativas (404 Not found), redirecciones, respuestas incompletas (206 Partial Content) o respuestas a otros métodos diferentes de GET si en su definición se permite el cacheado.
  - La caché debe incluir todas las cabeceras recibidas (incluso las no reconocidas), aunque hay algunas excepciones (como la cabecera Connection, Proxy-Connection, TE, Transfer-Encoding o Upgrade) y otras vinculadas a un proxy (Proxy-Authenticate, Proxy-Authentication-Info y Proxy-Authorization).
    - En posteriores respuestas recibidas, los valores de las cabeceras almacenadas deben ser actualizados (también hay ciertas excepciones como Content-Length).

8. Funcionamiento del protocolo HTTP

La caché en HTTP

Funcionamiento de la caché

- Cuando una caché tiene una o más respuestas almacenadas para una URI, pero no puede servir ninguna de ellas, puede emplear el mecanismo de peticiones condicionales. Las respuestas a estas peticiones podrían ser:
  - 304 (Not modified): indica que la respuesta almacenada en la caché puede ser actualizada y reusada.
  - Una respuesta completa: ninguna de las condiciones aportadas era apropiada, por lo que se debe emplear la respuesta completa para satisfacer la petición y podría ser almacenada en la caché.
  - Se recibe un error 5xx: se puede tanto devolver una respuesta almacenada, como repetir la petición condicional.

8. Funcionamiento del protocolo HTTP

La caché en HTTP

Funcionamiento de la caché

Campos de cabecera vinculados con la caché

- **Age:** tiempo en segundos que se estima que la respuesta ha estado en una caché desde que dicha respuesta se generó o se validó.
- **Cache-Control:** lleva una serie de directivas para controlar el comportamiento de la caché en la cadena de petición/respuesta. Estas directivas son unidireccionales y no tienen por qué ser iguales en la petición que en la respuestas (hay directivas diferentes para peticiones y respuestas). Ejemplos: max-age, no-cache, no-store, public, private, etc.
- **Expires:** fecha y hora a partir de la cual una respuesta es considerada obsoleta.

8. Funcionamiento del protocolo HTTP

La caché en HTTP

Tipos de cachés y problemas funcionales

- Ubicación de las cachés
  - Cliente de usuario.
  - En un intermediario. Por ejemplo las cachés compartidas, que guardan información de más de un usuario.
    - No se deben almacenar las respuestas marcadas con Cache-Control: private.
  - En el servidor.
- Problemas:
  - Complejidad añadida al protocolo, clientes y servidores.
  - Necesidad de control de las versiones.
  - Problemas de privacidad: trazabilidad de la actividad o las respuestas almacenadas con cookies.
  - Problemas de seguridad en cachés intermedias.

8. Funcionamiento del protocolo HTTP

Seguridad y privacidad en HTTP

- HTTP es en sí un protocolo no seguro ya que no aporta privacidad de los contenidos por sí mismo aunque sí implementa mecanismos de autenticación:
  - Entran en juego las cabeceras:
    - `WWW-Authenticate` normalmente en un mensaje de estado 401 (Unauthorized) del servidor o `Proxy-Authenticate` enviada con el mensaje de estado 407 (Proxy Authentication Required) generado por un proxy.
    - Y `Authorization` o `Proxy-Authorization` enviadas por el cliente.
    - En las respuestas a los clientes también se pueden incluir las cabeceras `Authentication-Info` y `Proxy-Authentication-Info`, con Información adicional sobre una autenticación correcta.
  - Los mecanismos de autenticación empleados en HTTP se detallan en el *Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry* del IANA (https://www.iana.org/assignments/http-authschemes/http-authschemes.xhtml).

8. Funcionamiento del protocolo HTTP

Seguridad y privacidad en HTTP

Ejemplos de esquemas de autenticación.

- Basic [RFC7617]: identificador de usuario y clave separados por “:” y codificado en BASE64.
  - Ejemplo RFC 7617 para userid=“Aladdin” y clave=“open sesame”:
    `Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==`
- Digest [RFC7616]: basado en resumen de mensaje, se emplea encriptado y algoritmos complejos, pero no es tan fiable como los mecanismos basados en clave pública.
- Bearer [RFC6750]: empleada en esquemas de autorización con testigos (*tokens*) como Oauth 2.0.

8. Funcionamiento del protocolo HTTP

Seguridad y privacidad en HTTP

- La seguridad y privacidad en HTTP recaen en otros niveles y en buenas prácticas (en la RFC 9110 se dan algunas de estas recomendaciones):
  - La principal hoy en día: emplear la versión más reciente del protocolo TLS (*Transport Layer Security*) entre el protocolo de transporte y el de aplicación, que es lo que indica el esquema (*scheme*) **https**.
  - No enviar credenciales de cliente en métodos GET, ya que irían en la URI y si se está en un navegador sería publicadas en la barra de dirección o en ficheros de registro en proxys.
  - Encriptado de los recursos. Aparte de que la conexión sea cifrada, cuando el recurso llega al servidor o cliente destino estará desprotegido, por lo que si éste se cifra se le añadirá una nueva capa de seguridad.
  - Un adecuada gestión y administración de los servidores web, como anonimizar los registros de actividad de los servidores (*logs*).

8. Funcionamiento del protocolo HTTP

Control de estado en HTTP

- A pesar de sus nuevas versiones, HTTP sigue siendo un protocolo sin estados.
  - Cada nueva petición no tiene en cuenta las peticiones o respuestas que se intercambiaron con anterioridad.
- Sin embargo, las nuevas aplicaciones web que fueron surgiendo en los años 90 del siglo XX hicieron necesaria la incorporación de un control de estado:
  - Inicialmente propuesto por Netscape ➞ aparecen las ***cookies***.
  - Publicado por primera vez en 1997 como RFC 2109 “HTTP State Management Mechanism”. Se definen dos nuevas cabeceras: **Cookie** y **Set-Cookie.**
  - Actualizado en la RFC 2965 (ahora en la RFC 6265) en el año 2.000: tres cabeceras: **Cookie**, **Cookie2** y **Set-Cookie2** (estas dos últimas han quedado obsoletas por la RFC 6265).
- Problemas:
  - Transmisión de información privada o sensible en una cookie.
  - Uso no deseado.
  - Cookies de terceras partes.

8. Funcionamiento del protocolo HTTP

Control de estado en HTTP

Funcionamiento

- Es un proceso iniciado por el servidor:
  - La aplicación de servidor crea una cookie, comunicándola al cliente a través de la cabecera **Set-Cookie** como un par de nombre/valor.
    - Para establecer varias cookies se deben enviar en cabeceras Set-Cookie por separado.
  - Así pues, en posteriores peticiones al mismo servidor el cliente enviará en la petacón la cabecera **Cookie** con todos los valores que recibió del servidor en anteriores cabeceras Set-Cookie, de manera que éste podrá emplear esta información, por ejemplo, seguir reconociendo al cliente y al proceso que llevaba a cabo (autenticación), establecer valores de configuración o rastrear sus preferencias.
  - Las cookies tienen que tener siempre un nombre (`name-value-pair`) para identificarlas, y luego pueden tener algún parámetro como los de validez temporal (`Expires`, `Max-Age`), el host (`Domain`) y otros como `Path`, `Secure` o `HTTPOnly`.

8. Funcionamiento del protocolo HTTP

Control de estado en HTTP

Ejemplo de funcionamiento

- Establecer una cookie:

```
== Server -> User Agent ==
   Set-Cookie: SID=31d4d96e407aad42

== User Agent -> Server ==
   Cookie: SID=31d4d96e407aad42
```

- Establecer varias cookies:

```
== Server -> User Agent ==
   Set-Cookie: SID=31d4d96e407aad42; Path=/; Secure; HttpOnly
   Set-Cookie: lang=en-US; Path=/; Domain=example.com

== User Agent -> Server ==
   Cookie: SID=31d4d96e407aad42; lang=en-US
```

8. Funcionamiento del protocolo HTTP

Servidores PROXY

- Se comportan como servidores (para el cliente web) y como clientes (para el servidor web) simultáneamente.
  - Transparentes: no modifican las peticiones del cliente.
  - No transparentes: pueden modificar las peticiones del cliente.
- Los clientes web deben configurarse específicamente para usar un determinado proxy.
- Beneficios:
  - Seguridad.
  - Compatibilidad con cachés.
  - Rendimiento.

9. Protocolo HTTP/1.1

Conexiones transitorias y persistentes

- En las versiones HTTP/0.9 y HTTP/1.0 las conexiones eran transitorias:
  - Una conexión TCP para cada recurso (documento de hipertexto, imagen, clip multimedia, etc.).
  - Esto derivó en un gran desperdicio de recursos conforme los documentos de hipertexto fueron añadiendo enlaces a otros tipo de recursos, tales como imágenes.

9. Protocolo HTTP/1.1

Conexiones transitorias y persistentes

- Solución de la versión HTTP/1.1: Conexiones persistentes.
  - A través de una misma conexión el cliente puede solicitar todos los recursos necesarios.
  - Se reduce la congestión en la red debida a innecesarios establecimientos de conexión.
  - Se ahorran recursos en los servidores al no tener que mantener tantas conexiones TCP abiertas como antes.
  - Los servidores HTTP/1.1 mantienen las conexiones transitorias para dar soporte a clientes HTTP/0.9 y HTTP/1.0
  - Un cliente HTTP/1.1 puede especificar su deseo de usar las conexiones transitorias, en vez de las persistentes, enviando la cabecera **`Connection:Close`**.

9. Protocolo HTTP/1.1

Pipelining

- La técnica de ***pipelining*** permite el encadenamiento de peticiones HTTP al servidor sin tener que esperar a que las realizadas previamente sean respondidas.
  - Pero se deben responder en orden.
- Tanto las conexiones persistentes, como el **pipelining** añaden complejidad al cliente.
  - El cliente tiene que discriminar entre todas las respuestas del servidor qué recurso es el que ha recibido de entre todos los solicitados.

9. Protocolo HTTP/1.1

Establecimiento y gestión de una conexión

- Conexión TCP al puerto 80 (well known).
  - Otro puerto se puede especificar en la URL.
- Conexión establecida: se envía la primera petición indicando la versión (0.9, 1.0 o 1.1).
  - Si es 0.9 o 1.0 el servidor usará automáticamente conexiones transitorias.
  - Si es 1.1, las conexiones serán persistentes y se usará el *pipelining*.
  - Un cliente 1.1 puede desistir de usar las conexiones persistentes, enviando la cabecera en la petición **`Connection:close`**.
- El servidor procesa las peticiones del cliente en orden y le envía las respuestas, en el mismo orden.
  - Si el cliente envía muchas peticiones, los servidores suelen usar el control de flujo de TCP para evitar que envíen más, en vez de cortar la conexión.
- El cliente recibe las respuestas y muestra al usuario los resultados.
  - Si el cliente ya no va a enviar más peticiones puede indicarlo enviando la cabecera **`Connection:close`** en la última que tuviera que enviar.

9. Protocolo HTTP/1.1

Formato general de los mensajes

- El formato del mensaje HTTP-message sigue la RFC 822 y MIME en gran parte, pero no lo hace estrictamente:

```
HTTP-message   = start-line CRLF
                 *( field-line CRLF )
                 CRLF
                 [ message-body ]
```

- Formato:
  - línea inicial
  - campos de cabecera (o cabeceras) del mensaje
  - línea en blanco
  - [cuerpo del mensaje]
- Las líneas terminan con **<CRLF>**
- La línea en blanco es un **<CRLF>**
- El cuerpo del mensaje es opcional.

9. Protocolo HTTP/1.1

Formato general de los mensajes

Ejemplo de mensajes

PETICIÓN

```
GET / HTTP/1.1
Host: www10.ujaen.es
User-Agent: Mozilla/5.0
Accept: text/html
Accept-Language: es-es,es;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

```
HTTP/1.1 200 OK
Date: Wed, 03 Oct 2012 09:22:10 GMT
Server: Apache
Content-Length: 6581
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
```

RESPUESTA

9. Protocolo HTTP/1.1

Formato de las peticiones

- La especificación de la sintaxis de HTTP/1.1 sigue la notación **Augmented Backus-Naur Form (ABNF)** [RFC 5234].
- Están basadas en el formato general antes mostrado:
  - Línea de petición.
  - Cabeceras generales.
  - Cabeceras de petición.
  - Cabeceras de entidad.
  - Línea en blanco.
  - [Cuerpo del mensaje].

9. Protocolo HTTP/1.1

Formato de las peticiones

Línea de petición

```
request-line   = method SP request-target SP HTTP-version
```

- Ejemplo:

```
GET /index.html HTTP/1.1
```

- Para completar esta petición, es obligatorio en HTTP/1.1 que siempre esté presente la cabecera Host:<host>.
- Ejemplo (petición del recurso por defecto):

```
GET / HTTP/1.1
Host: www.ujaen.es
```

9. Protocolo HTTP/1.1

Formato de las respuestas

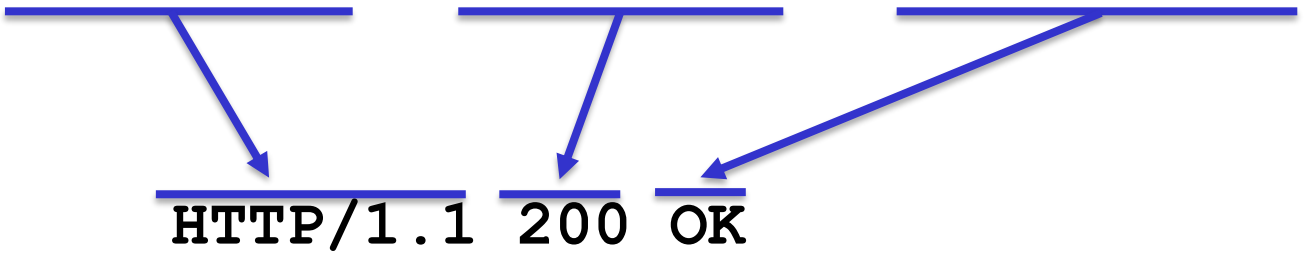
- Están basadas en el formato general antes mostrado:
  - Línea de estado
  - Cabeceras generales
  - Cabeceras de respuesta
  - Cabeceras de entidad
  - Línea en blanco
  - [cuerpo del mensaje]

9. Protocolo HTTP/1.1

Formato de las respuestas

Línea de estado

```
status-line = HTTP-version SP status-code SP [reason-phrase]
```



- Ejemplo:

```
HTTP/1.1 200 OK
```

9. Protocolo HTTP/1.1

Gestión de la transferencia de mensajes

- Se necesita una gestión especial dada la existencia de las conexiones persistentes de HTTP 1.1.
- En HTTP se informa de la longitud de una entidad a través de la cabecera Content-Length.
- ¿Qué hacer cuando la longitud no es conocida de antemano?
  - Una imagen de muy alta resolución.
  - Una respuesta a una búsqueda a una base de datos con posibilidad de ser muy extensa.
  - Un vídeo o audio.

- Transferencia por partes (*Chunked Transfer).*
- Cabeceras tráiler.

9. Protocolo HTTP/1.1

Gestión de la transferencia de mensajes

Transferencia fraccionada

- Los partes, o *chunks* (trozos) del recurso se van adjuntando al cuerpo de un mensaje HTTP.
- Estos mensajes se identifican con la cabecera **`Transfer-Encoding: chunked`**.
- Cada trozo va precedido por su longitud en hexadecimal codificada en ASCII.
- Cada longitud y datos de un trozo van terminados por la secuencia de caracteres **`CRLF`**.
- El final del envío se identifica con la longitud 0.

9. Protocolo HTTP/1.1

Gestión de la transferencia de mensajes

Transferencia fraccionada

- Definición en ABNF del cuerpo cuando es transferencia por trozos:

```
chunked-body    = *chunk
                  last-chunk
                  trailer-section
                  CRLF

chunk           = chunk-size [ chunk-ext ] CRLF
                  chunk-data CRLF
chunk-size      = 1*HEXDIG
last-chunk      = 1*("0") [ chunk-ext ] CRLF

chunk-data      = 1*OCTET ; a sequence of chunk-size octets

trailer-section = *( field-line CRLF )
```

9. Protocolo HTTP/1.1

Gestión de la transferencia de mensajes

Cabeceras tráiler

- Cuando se realiza un envío fraccionado en trozos, opcionalmente, siguiendo al último trozo, pueden ir cabeceras finales o tráiler.
- Esto está pensado para aquellas cabeceras cuyo contenido no podría haber sido generado de antemano.
- Si va a estar presente algún tráiler, se anuncia en las cabeceras de la entidad a través de cabecera **`Trailer`**.

9. Protocolo HTTP/1.1

Gestión de la transferencia de mensajes

Cabeceras tráiler

- Ejemplo de respuesta por trozos:

```
HTTP/1.1 200 OK
Date: Mon, 22 Mar 2004 11:15:03 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Trailer: Expires

29
<html><body><p>The file you requested is
5
3,400
23
bytes long and was last modified:
1d
Sat, 20 Mar 2004 21:12:00 GMT
13
.</p></body></html>
0
Expires: Sat, 27 Mar 2004 21:12:00 GMT
```

10. Protocolo HTTP/2

Motivación

- HTTP/1.1 ha mostrado que no es el protocolo más eficiente en el entorno actual de los servicios y contenidos disponibles en la www.
- Persigue, entre otros objetivos:
  - Reducir la latencia de usuario de manera efectiva y medible.
  - Solucionar el conocido problema de "*head of line blocking*" de HTTP (bloqueo producido por la limitación de conexiones entre clientes y servidores y el tener que esperar por los recursos en orden).
  - Paralelismo a nivel de aplicación, sin requerir múltiples conexiones TCP.
  - Mantener la semántica de HTTP/1.1 y escalando la nueva información, además de definir claramente la interacción entre HTTP/2 y HTTP/1.x.
  - Clarificar nuevas mejoras, extensiones o políticas para que puedan ser apropiadamente aplicadas.

10. Protocolo HTTP/2

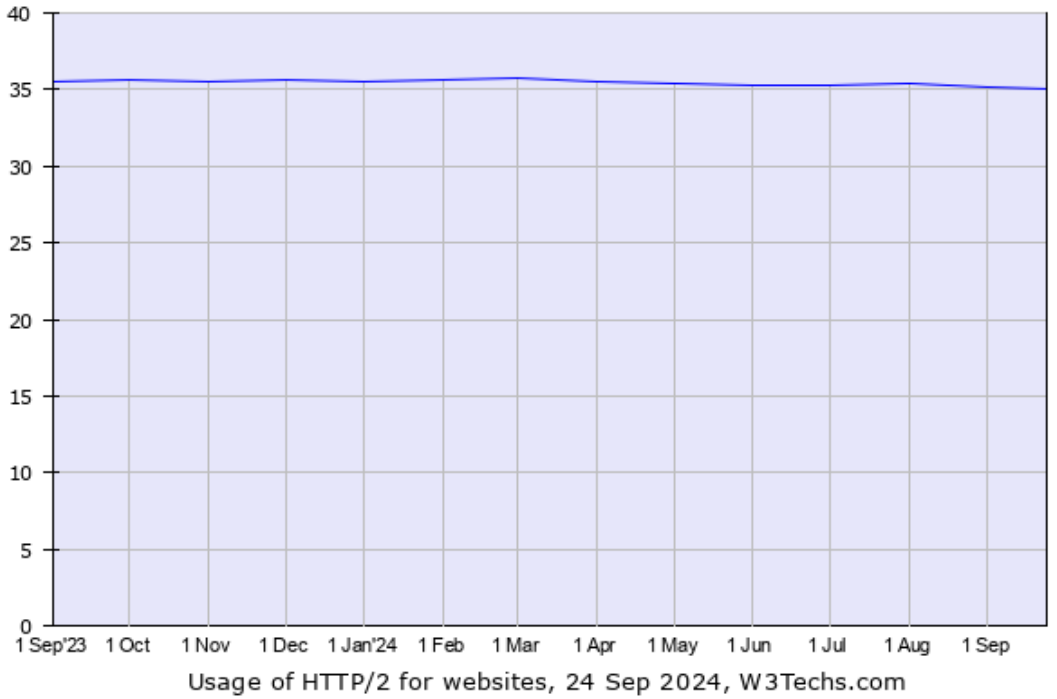
Los inicios del HTTP/2

- Sus especificaciones fueron publicadas en Mayo 2015 en la RFC 7540.
  - Fue desarrollado tomando como base el protocolo SPDY desarrollado por Google en 2012.
  - Los cambios propuestos no requieren ninguna modificación en el trabajo de las web existentes.
  - En 2022 se publicó la **RFC 9113 HTTP/2** que deja obsoletas a las RFC 7540 y a la RFC 8740 Using TLS 1.3 with HTTP/2.
- La estandarización fue apoyada por los navegadores Chrome, Safari, Internet Explorer 11, Opera, Firefox, Amazon Silk y Edge.
- Al final del 2015 la mayoría de los navegadores añadieron soporte para HTTP/2.

10. Protocolo HTTP/2

Estado actual

- Según W3Techs, en septiembre 2017 el **17%** de todos los servidores de Internet soportaban HTTP/2, en octubre de 2018 fue el **30,7%**, en julio de 2022 fue del **45 % y en septiembre de 2024 se mantiene en un 35% (debido a la irrupción de HTTP/3)**

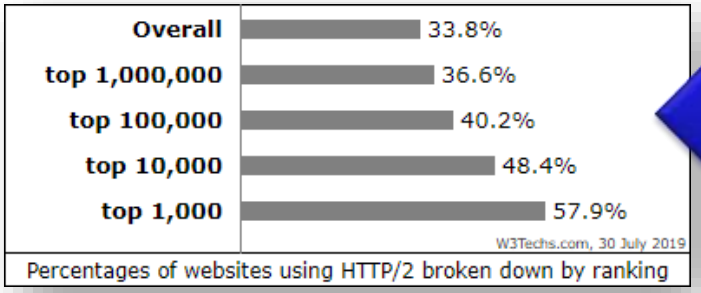


Fuente: https://w3techs.com/technologies/details/ce-http2/all/all

10. Protocolo HTTP/2

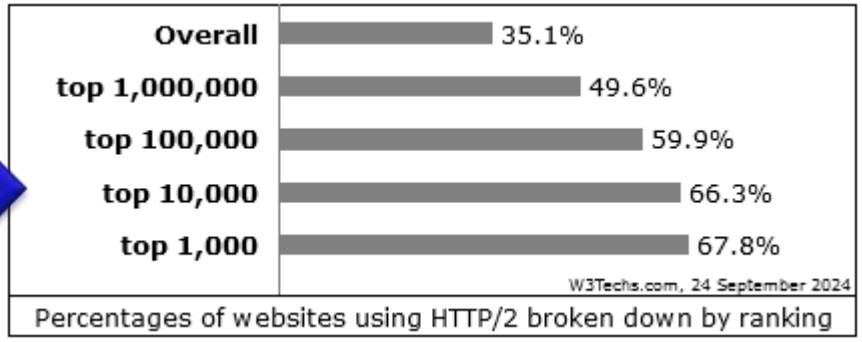
Estado actual

- El porcentaje de servidores que usan HTTP/2 se mantiene alrededor del 50% si se limita al millón de mejores servidores.



30 de julio de 2019

24 de septiembre de 2024



Fuente: https://w3techs.com/technologies/breakdown/ce-http2/ranking

10. Protocolo HTTP/2

Características generales

El objetivo del protocolo HTTP/2 es mejorar las carencias existentes en las anteriores versiones y que han sido puestas de manifiesto con el avance del servicio:

- **Mantiene la mayor parte de la sintaxis** de alto nivel de HTTP 1.1, como los métodos, los códigos de estado, los campos de cabecera y URI.
- Utiliza **una única conexión TCP** para ofrecer múltiples solicitudes y respuestas en paralelo.
- **Elimina información redundante**, evitando el envío de datos repetidos durante una misma conexión, consiguiendo que se consuman menos recursos, obteniendo una menor latencia.
- Permite **enviar y recibir varios mensajes al mismo tiempo** optimizando la comunicación, mediante la multiplexación.

10. Protocolo HTTP/2

Características generales

- Es un **protocolo binario**, siendo mucho más simple y por lo tanto menos propenso a tener errores respecto a los protocolos de texto.
  - La información se intercambia con tramas (*frames*).
- Las **cabeceras experimentan compresiones**, con lo que se obtienen mejores tiempos de respuesta y también se mejora la eficiencia.
  - El algoritmo empleado para la compresión es HPACK [RFC 7541].
- El **servidor puede enviar al cliente información no solicitada** que prevé que va a ser necesitada para que esté disponible en la caché (“server push”).
- Es un protocolo extensible (tipos de tramas, parámetros de configuración o códigos de errores).

10. Protocolo HTTP/2

Características generales

Uso de TLS y HTTP/2

- El uso de la encriptación TLS (*Transport Layer Security*) es opcional.
  - HTTP/2 debe emplear TLS 1.2 (RFC 5246) o versiones superiores, así que también puede emplear TLS 1.3 (RFC 8446).
  - El uso de TLS 1.2 soluciona problemas de conexión provocados por la cabecera UPGRADE, en la peticiones sobre el puerto 80, así como con los proxis, al emplear directamente el puerto 443 se evitan estos errores.
  - Hoy en día la mayoría de navegadores solamente soportan HTTP 2.0 sobre TLS.
  - TLS por sí solo no soluciona todos los problemas de seguridad, y en sí requiere de una constante vigilancia sobre técnicas de seguridad, actualizaciones de los servidores, etc.

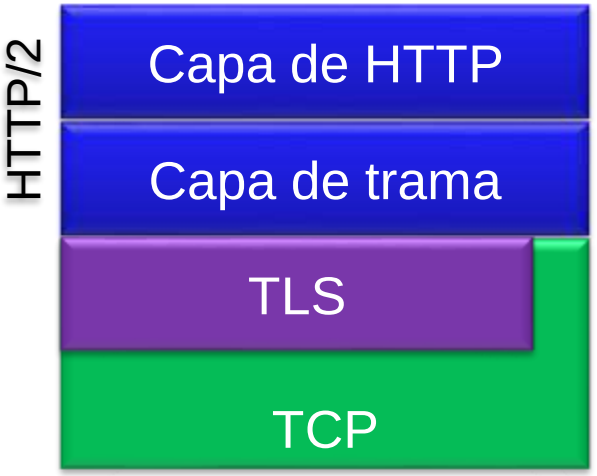
A pesar de no ser obligatorio, con HTTP/2 **se intentó que el uso de TLS sea global** para así extender el uso de comunicaciones cifradas por toda la web.

10. Protocolo HTTP/2

Estructura del protocolo HTTP/2

Fundamentos

- HTTP/2 organiza el comportamiento de HTTP en dos capas con funciones diferenciadas.
  - Capa de trama.
  - Capa de datos o capa de HTTP.
- HTTP/2 deja de ser un protocolo sin estados.
  - La conexión necesita guardar cierta información de la conexión: tramas, streams, tabla de cabeceras, etc.
  - Pero no se guarda información de la sesión.



Estructura de capas de HTTP/2

10. Protocolo HTTP/2

Estructura del protocolo HTTP/2

Fundamentos

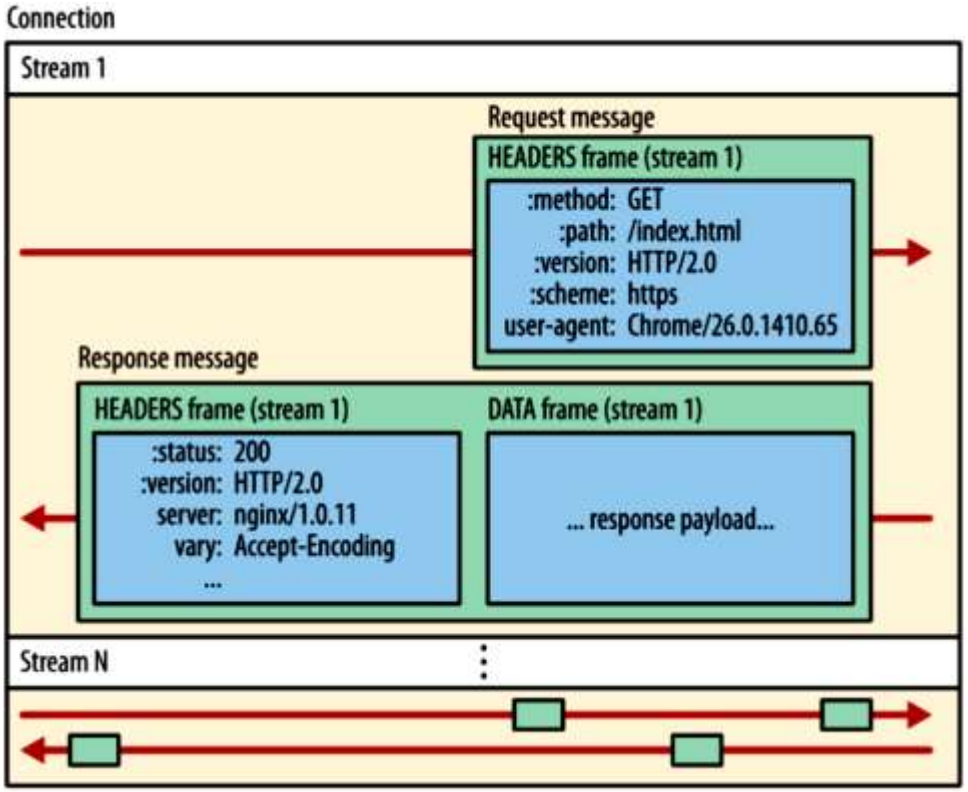
- El envío binario de HTTP/2 cambia completamente la forma de intercambiar información entre el cliente y el servidor.
- Aparecen nuevos conceptos:
  - **Stream**: un flujo bidireccional de bytes dentro de una conexión establecida que puede llevar una o más tramas.
  - **Mensaje**: una secuencia completa de tramas que corresponde con una petición o respuesta concreta.
  - **Trama**: La unidad mínima de comunicación de HTTP/2, cada una de las cuales contiene una cabecera de trama que, como mínimo, identifica al stream al que pertenece la trama.

10. Protocolo HTTP/2

Estructura del protocolo HTTP/2

Fundamentos

Streams, mensajes y tramas en HTTP/2



Fuente: High Performance Browser Networking, O'Reilly https://hpbn.co/http2/

10. Protocolo HTTP/2

Estructura del protocolo HTTP/2

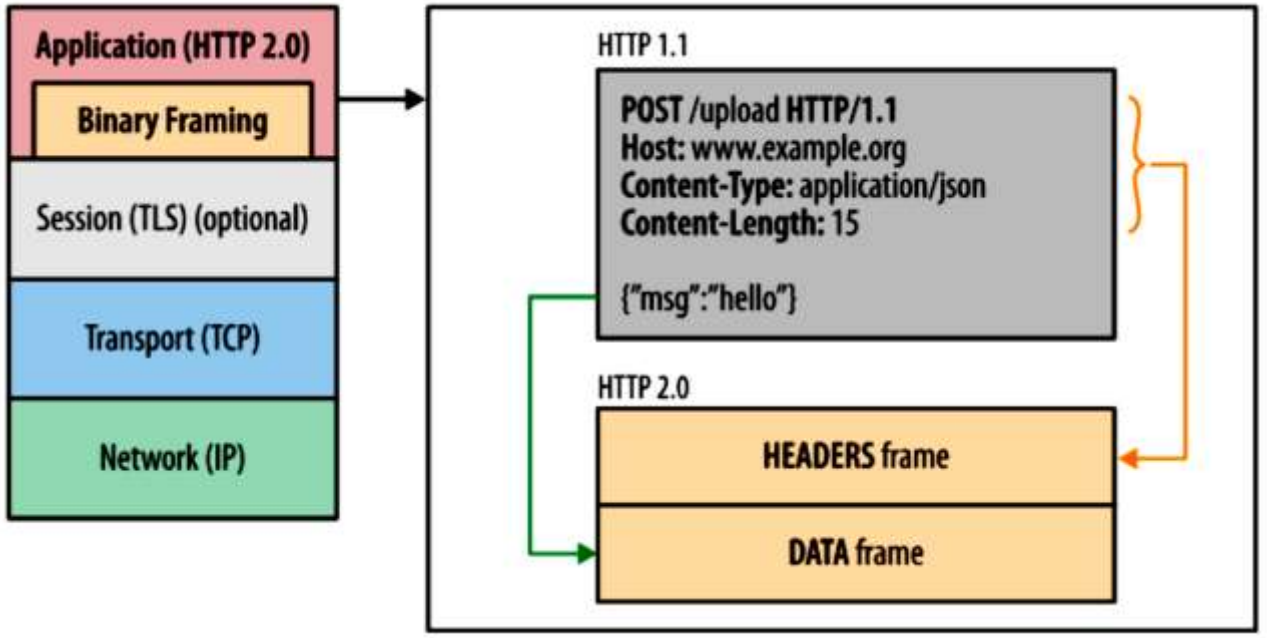
Capa de trama

- La capa de trama se encarga de proporcionar un mecanismo de intercambio de información HTTP entre cliente y servidor empleando tramas.
  - Las tramas siguen un formato binario.
  - Compresión de cabeceras, buscando una alta eficiencia.
  - Multiplexado: las peticiones y respuestas pueden ser intercaladas y se pueden completar cuando están listas, no teniendo que esperar a peticiones o respuestas anteriores.
  - Enfatiza el envío con mecanismos de encriptación (TLS).

10. Protocolo HTTP/2

Estructura del protocolo HTTP/2

Capa de trama



Fuente: High Performance Browser Networking, O'Reilly https://hpbn.co/http2/

Capa de envío de tramas binarias de HTTP/2

10. Protocolo HTTP/2

Estructura del protocolo HTTP/2

Capa de trama

Formato de trama

- Las tramas tienen tamaño variable.
  - El tamaño máximo por defecto es 2^{14} bytes.
  - Se pueden negociar tamaños mayores (2^{24}-1 bytes) con una trama SETTINGS.
  - El cliente y el servidor pueden llegar a un acuerdo para utilizar un tamaño de trama mayor (como máximo puede llegar a ser de 16 MB).
- Todas las tramas comienzan con una cabecera de 9 octetos fijos seguidos de la carga útil o datos, cuya longitud es variable.
  - Se facilita la sincronización de lectura y procesado por parte de clientes y servidores.

10. Protocolo HTTP/2

Formato del protocolo HTTP/2

Estructura

Formato de trama

- Las longitudes se expresan en bits (formato según RFC 9000 sección 1.3).
- Ejemplo:

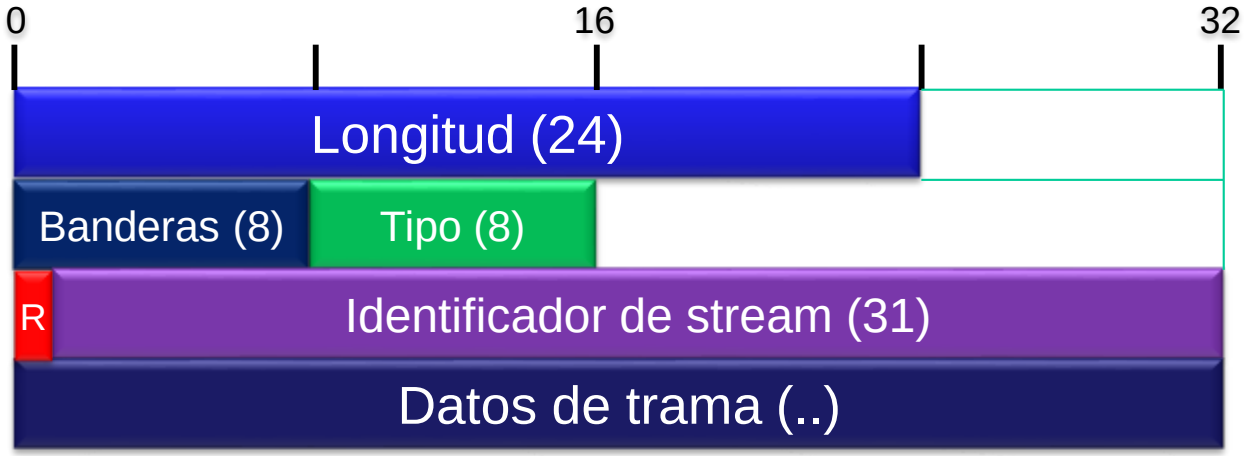
```
Example Structure {
  One-bit Field (1),
  7-bit Field with Fixed Value (7) = 61,
  Field with Variable-Length Integer (i),
  Arbitrary-Length Field (..),
  Variable-Length Field (8..24),
  Field With Minimum Length (16..),
  Field With Maximum Length (..128),
  [Optional Field (64)],
  Repeated Field (8) ...,
}
```

10. Protocolo HTTP/2

Formato del protocolo HTTP/2

Estructura

Formato de trama



10. Protocolo HTTP/2

Formato del protocolo HTTP/2

Estructura

Formato de trama

- **Cabecera (9 bytes):** es fija y siempre va al principio de cada trama. Sus campos son los siguientes:
  - Longitud (24 bits): entero sin signo que informa de los octetos de los datos de la trama.
    - Todas las implementaciones debe ser capaces de procesar tramas de hasta 2^{14} bytes.
  - Tipo (8 bits). Se debe descartar tramas de tipos desconocidos.
  - Banderas o *flags* (8 bits): marcan comportamientos concretos para los diferentes tipo de trama.
  - R (1 bit): reservado, el emisor lo debe poner a 0 y el receptor ignorarlo.
  - Identificador de stream (31 bits): entero sin signo que identifica una secuencia independiente de intercambio de información entre el cliente y el servidor dentro de una conexión HTTP/2.
- **Datos de trama (*payload*):** el contenido real de la trama, cuya longitud marca el campo del mismo nombre.

10. Protocolo HTTP/2

Formato del protocolo HTTP/2

Estructura

Tipos de trama

Nombre	ID	Descripción
DATA	00h	Lleva el contenido principal de un stream
HEADERS	01h	Lleva las cabeceras HTTP y opcionalmente prioridades
PRIORITY	02h	Indica cambios en la prioridades y dependencias
RST_STREAM	03h	Permite a un extremo finalizar un stream (normalmente en caso de error)
SETTINGS	04h	Comunicar parámetros a nivel de conexión
PUSH_PROMISE	05h	Informa al cliente que el servidor está a punto de enviarle algo

Tramas estandarizadas para HTTP/2: http://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml

10. Protocolo HTTP/2

Formato del protocolo HTTP/2

Estructura

Tipos de trama

Nombre	ID	Descripción
PING	06h	Chequea la conexión y mide el RTT (Round trip time)
GOAWAY	07h	Comunica al otro extremo que ya no aceptará nuevos streams
WINDOW_UPDATE	08h	Usada para el control de flujo, informa de cuantos bytes está dispuesto a aceptar un extremo
CONTINUATION	09h	Usado para extender tramas HEADERS o PUSH_PROMISE

Nuevas tramas (no incluidas en la RFC 7540)

Nombre	ID	Descripción
ALTSVC	0a	[RFC7838, Sección 4] Permite definir host, puerto o protocolos alternativos para un recurso
ORIGIN	0c	[RFC8336] Permite definir al servidor una lista de orígenes permitidos

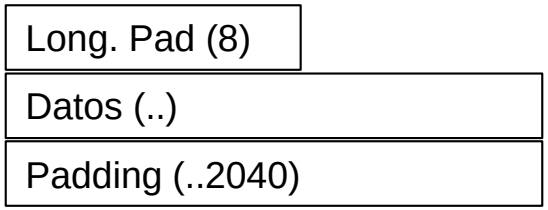
10. Protocolo HTTP/2

Formato del protocolo HTTP/2

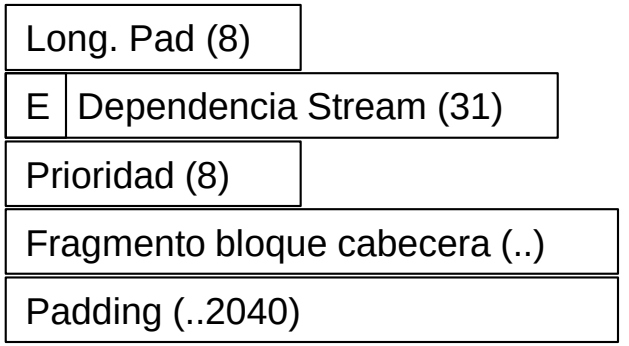
Estructura

Tipos de trama

- Trama **DATA** (type=0x00)
  - Flag=0x01 End_Stream (ultima trama que se mandará para ese stream)
  - Flag=0x08 Padded (el campo longitud de Padding y padding están presentes)



- Trama **HEADERS** (type=0x01)
  - E (Exclusivo) = flag que indica si la dependencia con el stream es exclusiva si se a especificado el flag PRIORITY (obsoleto RFC 9113)
  - Prioridad=1..256 si Flag=0x20 (PRIORITY)
  - Flags:
    - 0x01 END_STREAM: a 1 indica que no hay datos en la petición.
    - 0x04 END_HEADERS: contiene todas las cabeceras.
    - 0x08 PADDED.
    - 0x20 PRIORITY.



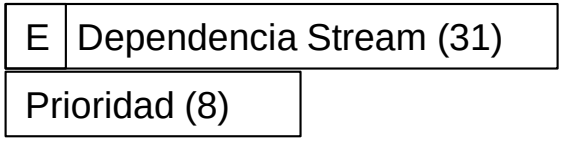
10. Protocolo HTTP/2

Formato del protocolo HTTP/2

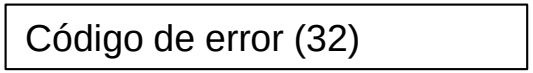
Estructura

Tipos de trama

- Trama **PRIORITY** (type=0x02) Obsoleta por la RFC 9113
  - E: flag que indica si la dependencia con el stream es exclusiva.
  - Prioridad (8b): establece el peso para un stream de 1 a 256.



- Trama **RST_STREAM** (type=0x03)
  - Cancelación inmediata del stream (se pasa el estado cerrado), puede ser debida a que ha habido un error, o por ejemplo a que la petición ya no se necesita.
    - Con HTTP/1.1 no era posible evitar que se enviara un recurso solicitado.
  - Puede indicar el código de error, el 0x00 es NO_ERROR.
  - El código de error a enviar en según qué situación no está definido, por lo que puede variar de una implementación a otra.



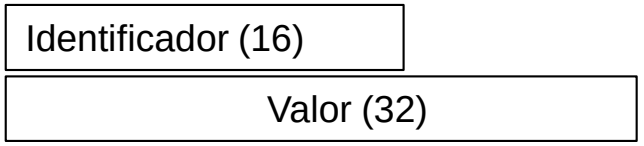
10. Protocolo HTTP/2

Formato del protocolo HTTP/2

Estructura

Tipos de trama

- Trama **SETTINGS** (type=0x04)
  - Es la primera trama a enviar tanto por el cliente como por el servidor y afectan a toda la conexión, no a streams concretos (usan el stream 0).
  - Establece los parámetros de configuración de la comunicación y deben ser asentidas.
  - Pueden ir varios bloques de identificador y valor (6 bytes).
  - Identificadores:
    - 0x01 SETTINGS_HEADER_TABLE_SIZE
    - 0x02 SETTINGS_ENABLE_PUSH
    - 0x03 SETTINGS_MAX_CONCURRENT_STREAMS
    - 0x04 SETTINGS_INITIAL_WINDOW_SIZE
    - 0x05 SETTINGS_MAX_FRAME_SIZE
    - 0x06 SETTINGS_MAX_HEADER_LIST_SIZE



Flag ACK (0x01): asiente una trama SETTINGS recibida previamente.

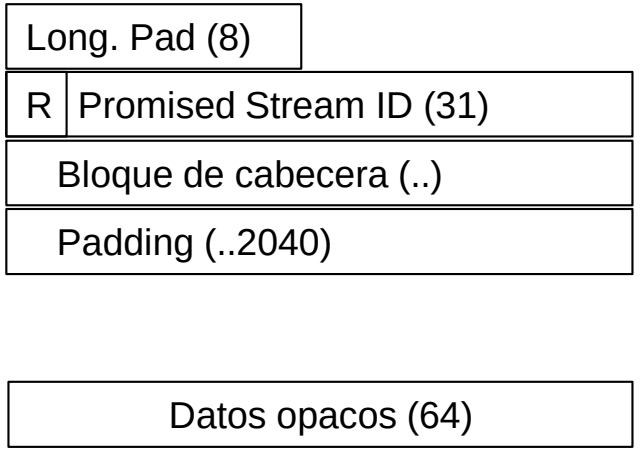
10. Protocolo HTTP/2

Formato del protocolo HTTP/2

Estructura

Tipos de trama

- Trama **PUSH_PROMISE** (type=0x05)
  - Permite al servidor avisar de que va a enviar un recurso al cliente por el *stream en promised stream ID*.
  - R = Reservado (1 bit).
  - El bloque de cabecera es igual que el de la trama HEADERS.
- Trama **PING** (type=0x06)
  - Se envía por el stream 0.
  - Se envía para medir el tiempo de ida y vuelta (RTT = *round trip time*) o también para comprobar si la conexión está activa ante periodos de inactividad.
  - Una trama PING debe ser respondida por otra trama PING con el flag ACK activo (0x01).



10. Protocolo HTTP/2

Formato del protocolo HTTP/2

Estructura

Tipos de trama

- Trama **GOAWAY** (type=0x07)
  - Inicia el cierre de la conexión y comunica el Id del último stream procesado (no admitirá nuevos streams) y un posible código de error (0x00 NO_ERROR), si lo hubiera.
  - R = Reservado (1 bit)

R	Id del último stream (31)
Código de error (32)	
Datos de depuración (..)	

- Trama **WINDOW_UPDATE** (type=0x08)
  - R = Reservado (1 bit)
  - Puede ser enviado en el stream 0 que afectaría a toda la conexión o en un stream concreto.
  - Incremento de tamaño de ventana
    - El tamaño de ventana máximo es de 2^{31}-1 bytes, el valor por defecto es 2^{16}-1 bytes (65.535 bytes).

R	Incremento del tamaño de ventana (31)

10. Protocolo HTTP/2

Formato del protocolo HTTP/2

Estructura

Tipos de trama

- Trama **CONTINUATION** (type=0x09)
  - Se emplea para continuar con la secuencia de campos de cabecera de una trama HEADERS o PUSH_PROMISE.
  - Debe seguir a una trama HEADERS o PUSH_PROMISE que no tengan activo el flag END_HEADERS.
  - Si una trama CONTINUATION no lleva activo el flag END_HEADERS debe estar seguida de otra trama CONTINUATION.
  - Solo define un flag:
    - 0x04 END_HEADERS: a 1 indica que no hay nuevas tramas CONTINUATION.

Fragmento bloque cabecera (..)

10. Protocolo HTTP/2

Formato del protocolo HTTP/2

Estructura

Explorar tramas

- Aparte de emplear WireShark, Chrome tiene una extensión que permite capturar las tramas intercambiadas con HTTP/2.
  - Poner en el campo de dirección: **`chrome://net-export/`**
  - Elegir las opciones y dar al botón de iniciar captura.
  - Ir a una nueva pestaña y navegar a la web deseada.
  - Volver a la pestaña de **`net-export`** y parar la captura cuando se haya cargado la página, esto generará un fichero local JSON.
  - Ir a **`https://netlog-viewer.appspot.com`** y cargar la captura y visualizar las tramas en la opción HTTP/2.

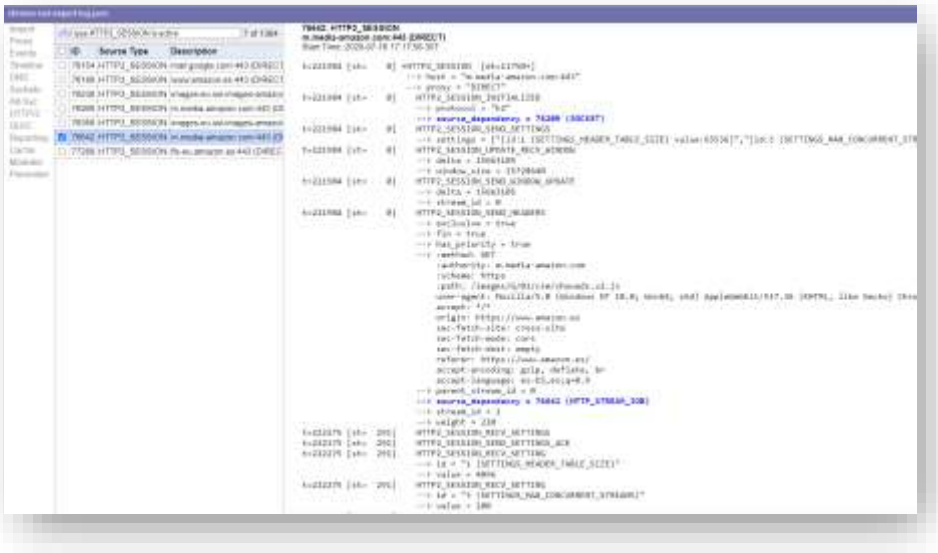


Imagen de una captura en https://netlog-viewer.appspot.com

10. Protocolo HTTP/2

Funcionamiento

Conexión

- Existen diferentes formas de iniciar una comunicación en HTTP/2
  - ALPN[1] desde TLS 1.2 o TLS 1.3. Se solicita el protocolo HTTP/2 o "**h2"** en el HANDSHAKE de TLS, evitando los RTT adicionales del caso anterior.
  - También es posible emplear el conocimiento previo que tiene el cliente de que el servidor soporta HTTP/2.
  - Cabecera UPGRADE (ha quedado obsoleta por la RFC 9113 ya que no fue apenas implementada):
    - El cliente enviaba con HTTP/1.1 una petición con la cabecera `UPGRADE` y el "**h2c**" token (h2c significa HTTP/2 *Cleartext*) y una cabecera `HTTP2-Settings` cuyo valor es una trama SETTINGS completa en Base64.

1: Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301

10. Protocolo HTTP/2

Funcionamiento

Conexión

Servicios alternativos

- Para iniciar una conexión en HTTP/2 existe otro mecanismo que es que el servidor puede informar de que soporta HTTP/2 a través de una nueva cabecera: *Alternate Services* (cabecera `Alt-Svc` en HTTP/1.1 o la trama `ALTSVC` en HTTP/2) [1].
  - Esta nueva funcionalidad permite definir un nuevo host, puerto o protocolo donde se puede encontrar un recurso solicitado al servidor, pero sin que instancias superiores tengan constancia de este cambio.
  - También puede informar de que se dispone de HTTP/3.

1: Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <https://www.rfc-editor.org/info/rfc7838>

10. Protocolo HTTP/2

Funcionamiento

Conexión

Prólogo de conexión

- Una vez se ha establecido la conexión y que la comunicación es sobre HTTP/2 ambos extremos deben intercambiar un prólogo de conexión.
  - Es diferente para cada extremo (cliente y servidor).
  - Para el cliente es la siguiente secuencia de 24 octetos:

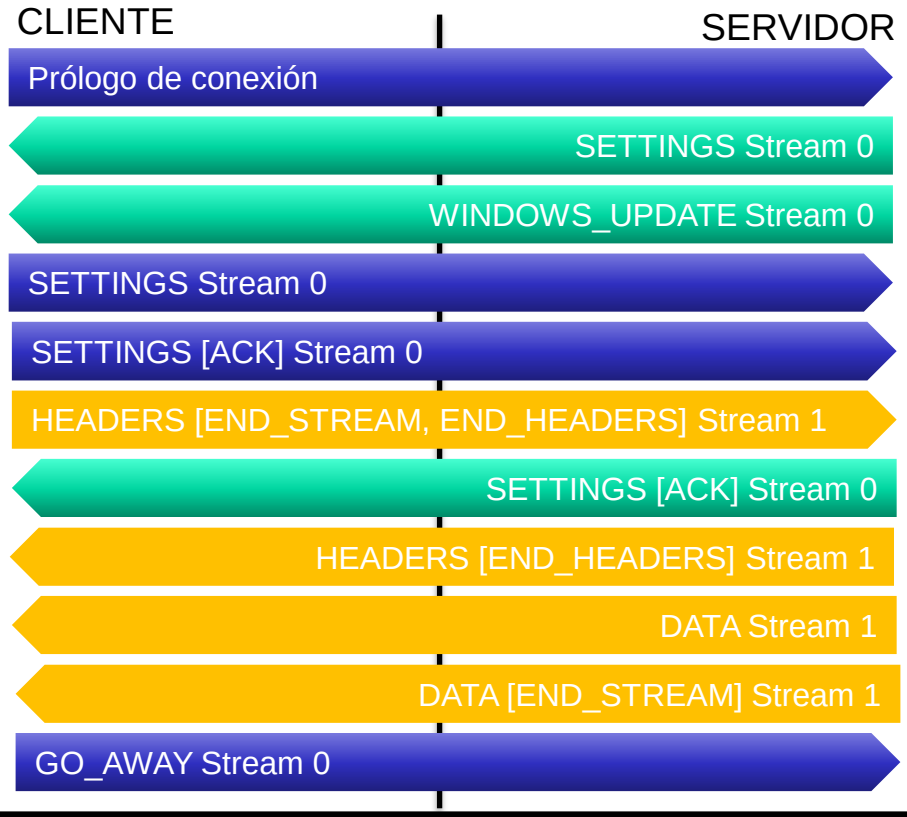
    `0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a`
    - Coincide con el mensaje siguiente: `PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n`
    - Con esto se consigue que un servidor que no entiende HTTP/2 devolverá error ante el método PRI, que no existe en HTTP/1.1 o HTTP/1.
    - A este prólogo le debe seguir una trama `SETTINGS`, la cual puede estar vacía.
  - El prólogo del servidor es una trama `SETTINGS`, la cual suele esta vacía y debe ser asentida.
- Para evitar una latencia innecesaria, los clientes pueden enviar tramas de petición inmediatamente después de enviar el prólogo sin tener que esperar a recibir el prólogo del servidor.

10. Protocolo HTTP/2

Funcionamiento

Conexión

Secuencia de conexión



- *En este ejemplo el cliente solicita un solo recurso (GET) y recibe la respuestas en dos tramas de datos, a continuación cierra la conexión.*
- Tanto el cliente como el servidor deben tratar un prólogo de conexión inválido como un error de conexión (PROTOCOL_ERROR). No se enviaría la trama GOAWAY dado que se supone que el otro extremo no usa HTTP/2.

10. Protocolo HTTP/2

Funcionamiento

Streams

- Un stream es una secuencia independiente y bidireccional de tramas intercambiadas entre el cliente y el servidor dentro de una conexión HTTP/2.
- Características de los streams:
  - Una simple conexión HTTP/2 puede tener múltiples streams simultáneos, con cualquiera de los extremos enviando tramas por múltiples de ellos.
  - Los streams pueden ser establecidos y usados unilateralmente o compartidos por cualquiera de los extremos.
  - Los streams pueden ser cerrados por cualquiera de los extremos.
  - El orden de las tramas enviadas es significativo, por lo que el receptor debe procesarlas en el orden recibido.
  - Cada stream es identificado por un entero que es asignado por el extremo que lo inicia.

10. Protocolo HTTP/2

Funcionamiento

Streams

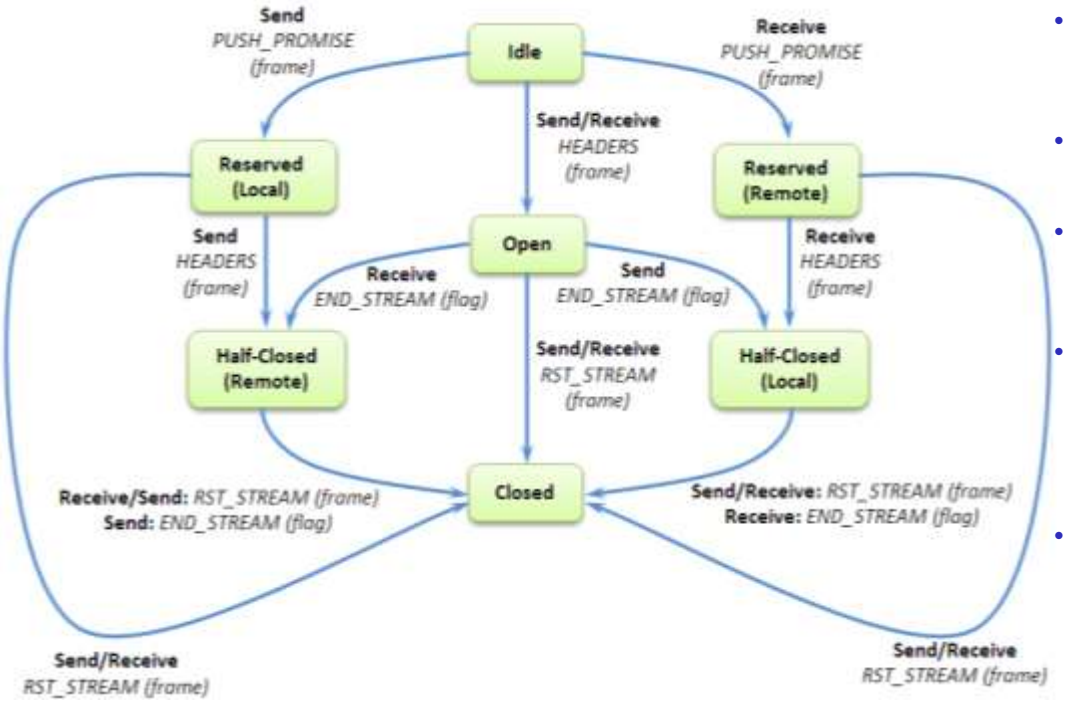
- En HTTP/2 cada petición se envía en un stream diferente.
  - El stream 0 es para **tramas asociadas con la conexión en su conjunto y no se puede emplear para peticiones de cliente, ni datos de servidor**.
  - Todas las nuevas peticiones deben emplear un identificador de stream mayor a todos los empleados con anterioridad y no se pueden reutilizar identificadores.
  - Los streams creados por el **cliente deben tener identificadores impares** y los **iniciados por el servidor serán siempre pares**.
  - En un mismo stream irán todas las peticiones y las correspondientes respuestas del servidor.
  - Cada extremo puede limitar el número de streams que el otro lado puede iniciar.
- Los streams tienen un ciclo de vida definido.
- Cada stream tiene un control de flujo independiente, en el que ambos extremos pueden comunicar actualizaciones de la ventana con WINDOW_UPDATE.

10. Protocolo HTTP/2

Funcionamiento

Streams

Ciclo de vida de un stream



Fuente: gRPC http://dist-prog-book.com/chapter/1/gRPC.html

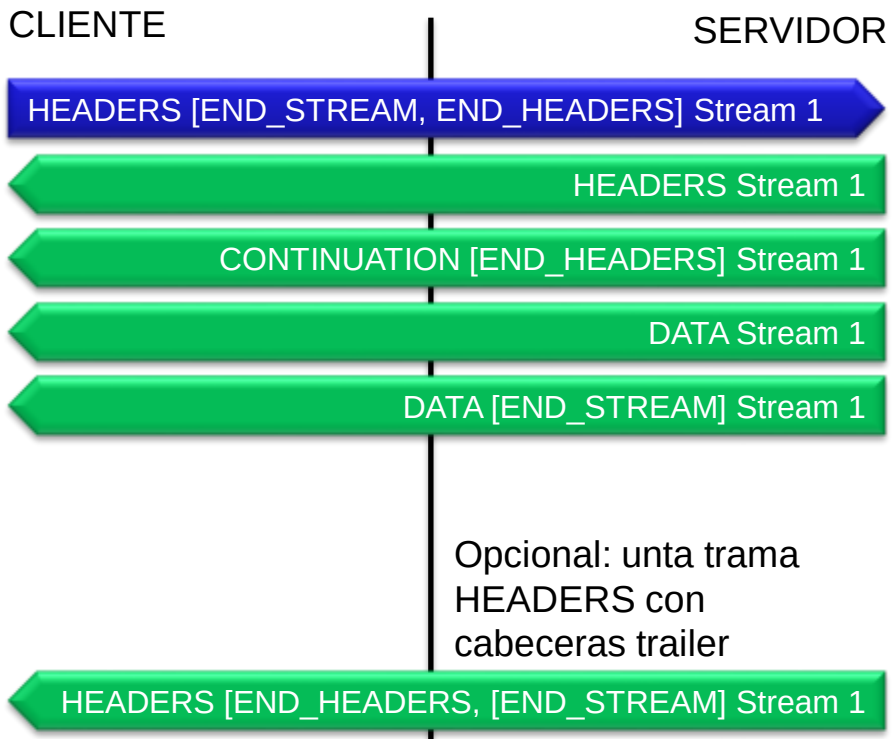
- **Idle:** cuando se acaba de crear o referenciar. Es un estado transitorio. Abrir un stream con un identificador mayor lleva automáticamente al stream al estado CLOSED.
- **Open:** se pasa a este estado cuando se envía una trama HEADERS y está listo para una comunicación bidireccional.
- **Reserved:** Se recibe una trama PUSH_PROMISE por lo que se supone que está disponible para recibir un recurso, pero no se conocerán los detalles hasta que se reciban una trama HEADERS.
- **Half-closed:** el cliente envía el flag END_STREAM por lo que indica que lo que solicita en la trama HEADERS es todo lo que necesita. Solo se enviarán tramas de control y los datos para esta petición.
  - Local: se pueden recibir tramas de cualquier tipo.
  - Remote: se pueden enviar tramas de cualquier tipo.
- **Closed:** un stream pasa a este estado cuando ambos extremos han enviado y recibido una trama con el flag END_STREAM o envía o recibe una trama con RST_STREAM. Este stream se considerará cerrado y no se debe volver a emplear.

10. Protocolo HTTP/2

Funcionamiento

Envío de mensajes

- Cada nueva petición el cliente la envía en un stream no usado previamente y el servidor le responde en ese mismo stream.
  - Un **intercambio completo de petición/respuesta consume completamente un stream**.
  - La transferencia por trozos de HTTP/1.1 no está permitida en HTTP/2.
- Un mensaje HTTP (petición o respuesta) consiste en:
  - Una trama HEADERS, seguida de ninguna o varias tramas CONTINUATION conteniendo la sección de cabecera.
  - Ninguna o varias tramas DATA que contienen el contenido del mensaje.
  - Opcionalmente, una trama HEADERS, seguida de ninguna o varias tramas CONTINUATION que contendría la sección de cabeceras tráiler, si estuviera presente.



10. Protocolo HTTP/2

Funcionamiento

Prioridad

- Que HTTP/2 sea capaz de priorizar unos streams frente a otros es fundamental para obtener un buen desempeño.
  - Prioridad en la versión de HTTP/2 de la RFC 7540: era un modelo rico y complejo **que no fue implementado** uniformemente y resulto infructuoso y **la RFC 9113 lo deja obsoleto**.
  - Aunque el modelo de la RFC 7540 se deja obsoleto, la señalización de prioridad sí se puede seguir empleando dado que aporta una información adicional a la comunicación que puede mejorar el rendimiento de la misma.
    - Cuando la señalización de prioridad de estima importante se recomienda emplear lo establecido en la RFC 9218 Extensible Prioritization Scheme for HTTP.
      - Define valores absolutos de prioridad en contra de los valores relativos de la RFC 7540.

10. Protocolo HTTP/2

Funcionamiento

PUSH_PROMISE

- Esta funcionalidad permite al servidor enviar datos al cliente sin que éste los solicite previamente.
  - Se puede desactivar poniendo a 0 el parámetro SETTINGS_ENABLE_PUSH.
- Se activa cuando el servidor envía al cliente una trama PUSH_PROMISE (por el stream 0).
  - La trama PUSH_PROMISE tiene un contenido similar a la trama HEADERS (petición) pero un parámetro adicional (`promised_stream_id`) que es el stream por el que el servidor enviará el recurso (un stream par).
  - El envío del recurso incluirá una trama HEADERS (respuesta) y las de datos necesarias.
  - Este mecanismo depende del servidor, por lo que deben configurarse qué recursos enviar con antelación, o el servidor revisar los recursos enviados para detectar dependencias.
- Además, plantea problemas en cuanto a adelantar recursos ya disponibles en caché, recursos que finalmente no se empleen, complejidad en la definición y análisis de qué recursos enviar,…

Ejemplo para activar SERVER_PUSH en la configuración del servidor web Apache para un recurso concreto.

```
<Location /index.html>
    Header add Link "</styles.css>;rel=preload"
    Header add Link "</script.js>;rel=preload"
</Location>
```

10. Protocolo HTTP/2

Funcionamiento

Control de flujo

- El control de flujo intenta evitar que los extremos que están funcionando con pocos recursos se colapsen, además de que los streams se bloqueen entre ellos.
- Se hace necesario al tener varios streams que pueden evolucionar de manera diferente sobre la misma conexión TCP.
  - HTTP/1.1 no lo necesitaba porque solamente se estaba respondiendo un recurso a la vez y se podía relegar este control a TCP.
- El control de flujo se puede aplicar a toda la conexión o a streams concretos.

10. Protocolo HTTP/2

Funcionamiento

Control de flujo

Principios del control de flujo

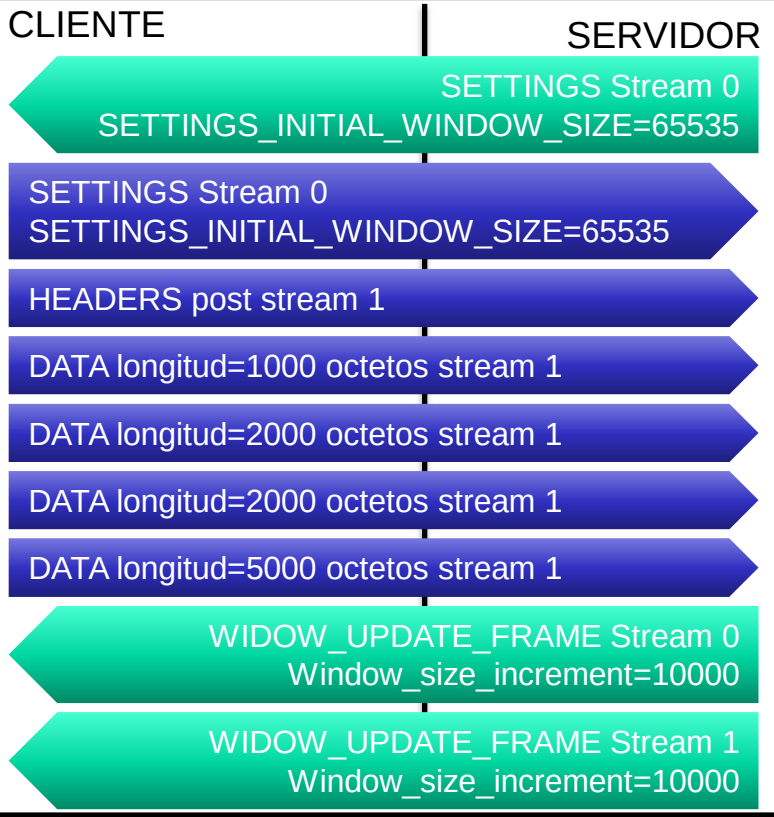
1. El control de flujo es salto a salto, no entre los extremos de toda la conexión.
2. Se basa en el intercambio de tramas WINDOW_UPDATE.
3. Es direccional, el receptor y el emisor pueden definir valores diferentes y deben ser respetados.
4. El tamaño inicial de la ventana de control de flujo se establece por defecto en 65.535 octetos tanto para nuevos streams como para la conexión en su conjunto.
5. Solamente las tramas DATA están limitadas por el control de flujo.
6. Un extremo puede deshabilitar su control de flujo, pero no puede ignorar los mensajes de control de flujo de su par.
7. HTTP/2 no define cuando un receptor debe enviar una trama WINDOWS_UPDATE, ni el valor que debe enviar, esto queda para las implementaciones.

10. Protocolo HTTP/2

Funcionamiento

Control de flujo

- Como se ha comentado, el mecanismo que proporciona HTTP/2 para comunicar que un extremo tiene disponibilidad para recibir datos es a través de la trama WINDOW_UPDATE.
  - El tamaño inicial se puede incrementar con la correspondiente trama WINDOW_UPDATE.
  - Las tramas WINDOW_UPDATE no se envían cada vez que se envía una trama DATA, se suelen acumular y una práctica habitual es enviarlas cuando se ha recibido la mitad del tamaño de la ventana.



10. Protocolo HTTP/2

Funcionamiento

Campos de cabecera

- Los campos de cabecera en HTTP/2 son transportados en las tramas HEADERS, CONTINUATION o PUSH_PROMISE.
- Validación de los campos de cabecera:
  - Solo pueden tener caracteres visibles ASCII y letras en minúscula y no pueden contener el espacio.
  - A excepción de las seudo-cabeceras (éstas no son en sí campos de cabecera, se verán más tarde), que comienzan con dos puntos: “:” (ejemplo: `:authority`, `:method` o `:scheme`) no pueden comenzar con dos puntos.
  - No pueden contener, en ninguna posición, el valor cero (ASCII Nul, 0x00), avance de línea (ASCII LF, 0x0a) o retorno de carro (ASCII CR, 0x0d).
  - No pueden empezar ni terminar con cualquier espacio en blanco (ASCII SP o HTAB, 0x20 o 0x09).

10. Protocolo HTTP/2

Funcionamiento

Datos de control - campos de seudo-cabecera

- Los **campos de seudo-cabecera** (comienzan con ":") se emplean para transportar información de control del mensaje.
  - No son cabeceras HTTP y los extremos solamente deben emplear los definidos en la RFC 9113 (aunque una extensión del protocolo podría añadir nuevos).
  - Solamente son válidos en el contexto para el que son definidos (los campos de seudo-cabecera de peticiones no pueden aparecer en respuestas y viceversa).
  - Tampoco pueden aparecer en secciones tráiler.
  - Todos deben aparecer al principio de un bloque de cabeceras (antes de todos los campos de cabecera).
  - No deben aparecer más de una vez.
- Tanto las peticiones como las respuestas de HTTP/2 tienen una versión implícita de 2.0.

10. Protocolo HTTP/2

Funcionamiento

Datos de control - campos de seudo-cabecera

Tipos de seudo-cabeceras de petición

- Petición:
  - **`:method`** incluye el método HTTP de la petición.
  - **`:scheme`** incluye la porción de esquema (*scheme*) de la URI del recurso objetivo. No se limita a “http” o “https”, pero son las más comunes.
  - **`:authority`** porta la porción de la autoridad de la URI destino (dominio o IP normalmente). Si está presente el receptor no debe emplear la cabecera Host, aunque sus valores deben coincidir si están presentes ambos.
  - **`:path`** incluye la parte de la ruta y peticiones (*queries*) de la URI destino. No puede estar vacío (al menos “/”) salvo en el método OPTIONS que valdrá “*” y en CONNECT que se omitirá.
  - Todas las peticiones HTTP/2, menos las CONNECT, deben incluir, al menos ":method", ":scheme" y ":path“.
- Respuestas:
  - **`:status`** porta el código de estado de HTTP y debe estar en todas las respuestas.

10. Protocolo HTTP/2

Funcionamiento

Compresión de cabecera

- Uno de los aspectos más importantes para proporcionar una mayor eficiencia en la transmisión, es el uso de compresión de cabeceras.
  - El algoritmo empleado es **HPACK** [RFC 7541].
  - Cada extremo mantiene dos tablas, una **dinámica** con cabeceras de la comunicación en curso, y otra **estática**, con las 61 entradas de cabeceras más comunes, como :method: GET
  - Existen diversas políticas para indexar o no las cabeceras.
  - Usa compresión entera.
  - Emplea codificación Huffman para mayor compresión de las cadenas de texto.

Índice	Cabecera	Valor
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
6	:scheme	http
7	:scheme	https
8	:status	200
9	:status	204
10	:status	206
11	:status	304
12	:status	400
13	:status	404
14	:status	500
15	accept-charset	
16	accept-encoding	gzip, deflate

Extracto de la tabla estática.
https://datatracker.ietf.org/doc/html/rfc7541#appendix-A

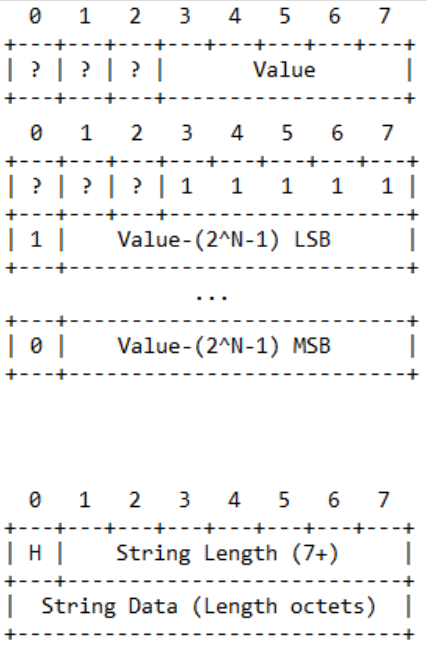
10. Protocolo HTTP/2

Funcionamiento

Compresión de cabecera

HPACK

- Los enteros menores de 32 se codifican en un solo byte, si son mayores se añaden bytes con el MSB a 1 menos para el último byte.
- Las cadenas se pueden representar como literales con la misma codificación original o con codificación Huffman (bit H=1).
  - https://tools.ietf.org/html/rfc7541



10. Protocolo HTTP/2

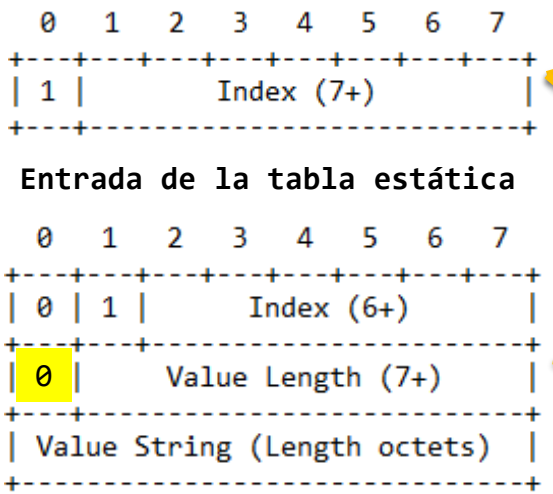
Funcionamiento

Compresión de cabecera. Ejemplo sin codificación Huffman

HPACK

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
```

8286 8441 0f77 7777 2e65 7861 6d70 6c65 2e63 6f6d = 20 bytes




```
82 | == Indexed - Add ==
   | idx = 2
   | -> :method: GET
86 | == Indexed - Add ==
   | idx = 6
   | -> :scheme: http
84 | == Indexed - Add ==
   | idx = 4
   | -> :path: /
41 | == Literal indexed ==
   | Indexed name (idx = 1)
   | :authority
0f | Literal value (len = 15)
7777 772e 6578 616d 706c 652e 636f 6d | www.example.com
   | -> :authority:
   | www.example.com
```

HTTP/1.1
41 bytes

Sin comprimir (ASCII)
www.example.com

10. Protocolo HTTP/2

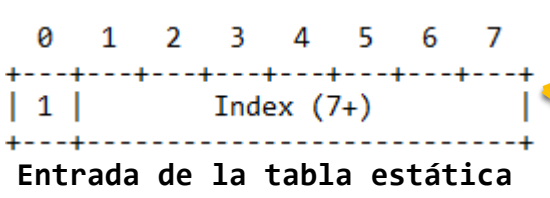
Funcionamiento

Compresión de cabecera. Ejemplo con codificación Huffman

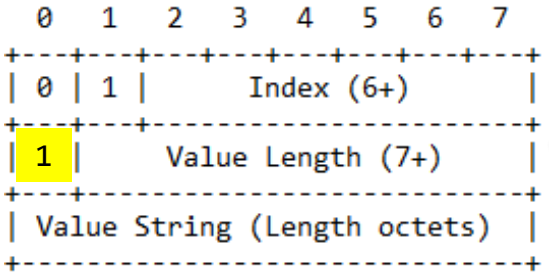
HPACK

```
:method: GET
:scheme: http
:path: /
:authority: www.example.com
```

8286 8441 8cf1 e3c2 e5f2 3a6b a0ab 90f4 ff = 17 bytes



Entrada de la tabla estática



Entrada en la tabla dinámica con indexado incremental

Códigos Huffman

82	'w' (119) |1111000	== Indexed - Add ==
	'.' (46) |010111	idx = 2
	'e' (101) |00101	-> :method: GET
86	'x' (120) |1111001	== Indexed - Add ==
	'a' (97) |00011	idx = 6
	'm' (109) |101001	-> :scheme: http
84	'p' (112) |101011	== Indexed - Add ==
	'l' (108) |101000	idx = 4
		-> :path: /
41		== Literal indexed ==
		Indexed name (idx = 1)
		:authority
8c		Literal value (len = 12)
		Huffman encoded:
f1e3 c2e5 f23a 6ba0 ab90 f4ff		www.example.com
		-> :authority:
		www.example.com

10. Protocolo HTTP/2

Funcionamiento

Compresión de cabecera. Ejemplo con codificación Huffman

Ejemplos de mensajes (RFC 9113)

Petición simple

```
GET /resource HTTP/1.1              HEADERS
Host: example.org           ==>       + END_STREAM
Accept: image/jpeg                    + END_HEADERS
                                        :method = GET
                                        :scheme = https
                                        :authority = example.org
                                        :path = /resource
                                       host = example.org
                                       accept = image/jpeg
```

Respuesta simple

```
HTTP/1.1 304 Not Modified           HEADERS
ETag: "xyzzy"               ==>       + END_STREAM
Expires: Thu, 23 Jan ...              + END_HEADERS
                                        :status = 304
                                        etag = "xyzzy"
                                        expires = Thu, 23 Jan ...
```

10. Protocolo HTTP/2

Funcionamiento

Compresión de cabecera. Ejemplo con codificación Huffman

Ejemplos de mensajes (RFC 9113)

Petición compleja

```
POST /resource HTTP/1.1
Host: example.org
Content-Type: image/jpeg
Content-Length: 123

{binary data}
```

==>

```
HEADERS
  - END_STREAM
  - END_HEADERS
    :method = POST
    :authority = example.org
    :path = /resource
    :scheme = https

CONTINUATION
  + END_HEADERS
    content-type = image/jpeg
    host = example.org
    content-length = 123

DATA
  + END_STREAM
{binary data}
```

10. Protocolo HTTP/2

Funcionamiento

Compresión de cabecera. Ejemplo con codificación Huffman

Ejemplos de mensajes (RFC 9113)

Respuesta con cuerpo

```
HTTP/1.1 200 OK                      HEADERS
Content-Type: image/jpeg    ==>        - END_STREAM
Content-Length: 123                    + END_HEADERS
                                         :status = 200
{binary data}                            content-type = image/jpeg
                                         content-length = 123

                                     DATA
                                       + END_STREAM
                                     {binary data}
```

10. Protocolo HTTP/2

Funcionamiento

Compresión de cabecera. Ejemplo con codificación Huffman

Ejemplos de mensajes (RFC 9113)

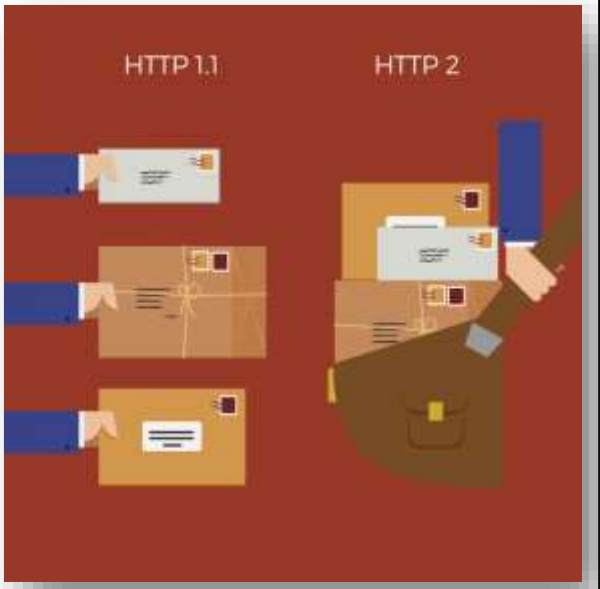
Respuestas informativas

```
HTTP/1.1 100 Continue                HEADERS
Extension-Field: bar        ==>        - END_STREAM
                                       + END_HEADERS
                                         :status = 100
                                         extension-field = bar

HTTP/1.1 200 OK                      HEADERS
Content-Type: image/jpeg    ==>        - END_STREAM
Transfer-Encoding: chunked           + END_HEADERS
Trailer: Foo                             :status = 200
                                         content-type = image/jpeg
123                                      trailer = Foo
{binary data}
0                                    DATA
Foo: bar                               - END_STREAM
                                     {binary data}

                                     HEADERS
                                       + END_STREAM
                                       + END_HEADERS
                                         foo = bar
```

10. Protocolo HTTP/2

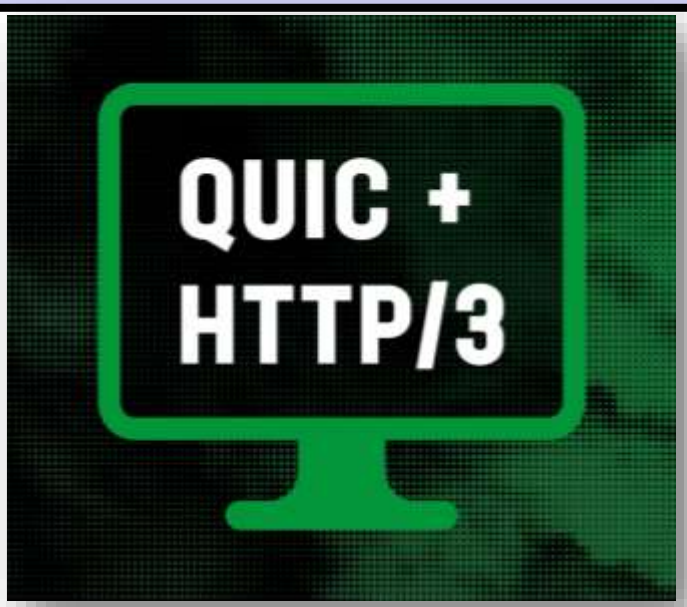
Ejemplos de comparación de HTTP/1.1 con HTTP/2

- Comprobar con HTTP/1.1 y HTTP/2 con www.webpagetest.org.
- Deshabilitar HTTP/2 en Firefox:
  - Escribir `about:config` en la línea de dirección.
  - Aceptar riesgos.
  - Buscar la propiedad: `network.http.spdy.enabled.http2` y ponerla a false (HTTP/1.1).



11. Protocolo HTTP/3

- HTTP/3 es una nueva versión de HTTP estandarizada junto al nuevo sistema de compresión de cabeceras QPACK:
  - RFC 9114 HTTP3 (Junio 2022)
  - RFC 9204 QPACK: Field Compression for HTTP/3 (Junio 2022)
- HTTP/3 sigue manteniendo los aspectos fundamentales de HTTP/2, pero otros, como los *streams*, los relega a la capa de transporte que emplea el protocolo QUIC.
- Siguen existiendo:
  - Cabeceras y cuerpo.
  - Peticiones y respuestas.
  - Métodos.
  - Cookies.
  - Cache
  - …



QUIC está especialmente diseñado para el tráfico web.

- QUIC ofrece conexión, recuperación de errores y control de congestión.
- QUIC se ha montado sobre UDP para facilitar su despliegue.
- Se verá más sobre QUIC en el Tema 3.

11. Protocolo HTTP/3

¿Por qué QUIC y HTTP/3?

- TCP posee muchas ventajas, pero un inconveniente: el **bloqueo de cabecera (*head of line blocking*)** en el caso de un segmento perdido o dañado, similar a lo que le sucedía a HTTP/1.1 y aunque emplee HTTP/2:
  - Si hay un segmento dañado o perdido, todos los posteriores que hayan llegado al destino no pueden ser procesados hasta que se recupere esa pérdida, lo cual paraliza, concretamente en HTTP/2, otros streams que podrían estar avanzando, mientras que el stream que ha perdido un segmento sí se podría quedar parado.
- Para eliminar este bloqueo se ha diseñado el protocolo de transporte QUIC, por lo que se hace necesaria una nueva versión de HTTP.

11. Protocolo HTTP/3

¿Por qué QUIC y HTTP/3?

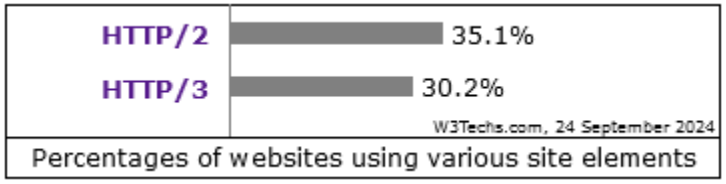
Bloqueo de cabecera (head of line blocking)

Bloqueados: los streams 3 y 4 no pueden continuar y ser enviados a la aplicación HTTP/2, a pesar de no tener errores, hasta que el segmento *n* sea retransmitido y recibido sin errores.

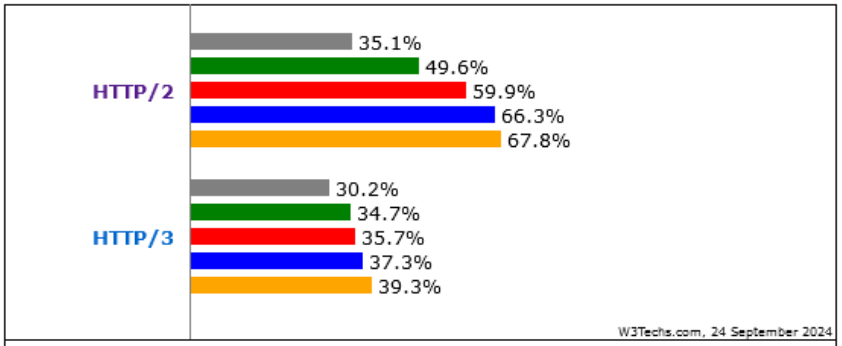
11. Protocolo HTTP/3

¿Por qué QUIC y HTTP/3?

Estado y evolución de HTTP/3 y HTTP/2



Percentages of websites using various site elements



Percentages of websites using the selected site elements broken down by ranking
Note: a website may use more than one site element

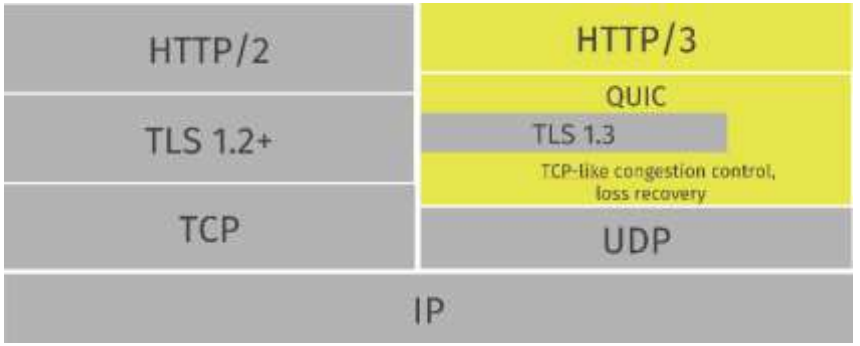
Usage of selected site elements for websites, 24 Sep 2024, W3Techs.com

Datos a septiembre de 2024.
Fuente: https://w3techs.com/technologies/comparison/ce-http2,ce-http3

11. Protocolo HTTP/3

Novedades en HTTP/3

- La novedades en HTTP/3 son las siguientes:
  - Los stream los proporciona QUIC a nivel de transporte por lo que HTTP/3 no aporta streams.
  - QUIC proporciona confiabilidad a nivel de stream y control de congestión para toda la conexión.
  - Los streams QUIC son ligeramente diferentes a como son en HTTP/2.
  - Ahora los streams son independientes, por lo que el protocolo de compresión de cabecera empleado en HTTP/2 no se puede emplear, se ha diseñado uno nuevo denominado QPACK.
  - QUIC también incorpora TLS 1.3.



Pila de protocolos de HTTP/3 sobre QUIC. Fuente: https://http3-explained.haxx.se/en/the-protocol

11. Protocolo HTTP/3

Comparativa con HTTP/2

Streams

- HTTP/3 permite un mayor número de streams (2^{62}-1) ya que los proporciona QUIC.
- La concurrencia de streams la aporta QUIC, que considera cerrado un stream cuando todos los datos han sido recibidos y asentidos por el par y no como en HTTP/2 que se indicaba con el flag END_STREAM.
- En HTTP/2 solo los datos (payload) de las tramas DATA se veían afectados por el control de flujo. En HTTP/3, como el control de flujo lo aporta QUIC para cada stream, todas las tramas están sujetas a él.

11. Protocolo HTTP/3

Comparativa con HTTP/2

Tipos de trama en HTTP

- Dado que QUIC gestiona los streams, no se hace necesario un identificador de stream en las tramas HTTP, así como un flag END-STREAM ya que, igualmente, QUIC se encarga de la finalización de los streams.
- Las tramas que no son necesarias en HTTP/3 (como para el control de flujo) son eliminadas, pero su identificadores se han reservado.
- Algunas tramas comunes entre HTTP/2 y HTTP/3 tienen semánticas diferenciadas. También difiere el formato de todas ellas.
- HTTP/3 no dispone de priorización entre streams, lo cual no quiere decir que no sea importante.
  - Esto se debe a la poca repercusión y falta de implementaciones del modelo de prioridad de HTTP/2 que, como se comentó, se eliminó en su segunda RFC.
- Las tramas en HTTP/3 no tienen un campo de flags, por lo que estos se tienen que añadir en la parte de carga útil de la trama y además, los campos siguen el formato de enteros de longitud variable que usa QUIC.

11. Protocolo HTTP/3

Comparativa con HTTP/2

Tipos de trama en HTTP

- **DATA(0x00):** no se define el relleno (*padding*).
- **HEADERS (0x01):** no se definen los campos vinculados a la prioridad, ni el relleno (*padding*).
- **RST_STREAM (0x03):** no se definen en HTTP/3 ya que QUIC proporciona el control sobre el ciclo de vida de un stream.
- **SETTINGS (0x04):** solamente se envían al principio de una conexión.
- **PUSH_PROMISE (0x05):** esta trama ya no se refiere a un stream, sino que incorpora un identificador push ID para referenciar el envío futuro.
- **PING (0x06):** no existen en HTTP/3, este comportamiento lo aporta QUIC.
- **GOAWAY (0x07):** no contiene un código de error.
- **WINDOW_UPDATE (0x08):** no existe en HTTP/3 dado que el control de flujo lo proporciona QUIC.
- **CONTINUATION (0x09):** no existe en HTTP/3 ya que se permiten tramas HEADERS/PUSH_PROMISE más grandes.

11. Protocolo HTTP/3

Características de HTTP/3

- HTTP/3 empleará siempre el esquema `https://` no tiene opción en texto claro.
- HTTP/3 tiene algunas tramas nuevas, pero mantiene varias de HTTP/2:
  - DATA, HEADERS, SETTINGS, PUSH_PROMISE,GOAWAY.
  - Nuevas: CANCEL_PUSH y MAX_PUSH_ID.
- Una **petición** se realiza sobre un stream bidireccional enviando una cabecera HEADERS, seguida por una o varias tramas DATA y, opcionalmente, por una nueva trama HEADERS con los tráileres.
- Las **respuestas** pueden tener las mismas tramas que una petición: HEADERS, y DATA o HEADERS para los tráileres.
- Las cabeceras comprimidas dentro de las tramas HEADERS emplean QPACK, el cual es similar a HPACK pero se sirve de dos streams unidireccionales para informar de las tablas dinámicas en ambas direcciones.

11. Protocolo HTTP/3

Características de HTTP/3

- SEVER_PUSH es similar a HTTP/2 pero solo ocurren si el cliente los ha aceptado, informando cuantos aceptará con la trama MAX_PUSH_ID.
  - Los push stream se pueden cancelar con la trama CANCEL_PUSH.
- Las conexiones en HTTP/3 pueden ser reutilizadas, para lo que el cliente debe guardar convenientemente las credenciales de seguridad del servidor.

11. Protocolo HTTP/3

Conexiones en HTTP/3

Establecimiento de una conexión HTTP/3

- Como HTTP/3 emplea QUIC versión 1 como capa de transporte se deben dar ciertos pasos:
  - QUIC emplea TLS 1.3, por lo que los clientes HTTP/3 tienen que tener una forma de comunicar el host destino al servidor durante el saludo (*handshake*) de TLS.
    - Si el servidor se identifica por un nombre de dominio el cliente debe emplear la extensión de TLS Server Name Indication (SNI; [RFC6066]), a menos que haya otro mecanismo alternativo para indicar el host.
    - Las conexiones QUIC se establecen como se definen en el protocolo y el empleo de HTTP/3 se especifica con el token “h3” en el protocolo ALPN (Application-Layer Protocol Negotiation) del saludo de TLS.
    - Una vez QUIC ha establecido la conexión, cada extremo intercambia una trama SETTINGS por el stream de control.

11. Protocolo HTTP/3

Conexiones en HTTP/3

Cierre de una conexión HTTP/3

- **Conexiones sin actividad:** en el saludo de una conexión QUIC se establece un tiempo de expiración por inactividad, por lo que si se alcanza ese periodo sin recibir paquetes, la conexión se considerará cerrada.
- **Cierre de la conexión:** aún en una conexión no inactiva, cualquiera de los extremos puede iniciar un cierre acordado de conexión enviando una trama GOAWAY.
  - La trama GOAWAY informa del rango de peticiones o *pushes* que se han procesado en la conexión, todas las peticiones con identificadores mayores serán rechazadas o descartadas.
- **Cierre inmediato de aplicación:** una implementación HTTP/3 puede cerrar inmediatamente una conexión QUIC en cualquier momento enviando una trama QUIC denominada CONNECTION_CLOSE.
- **Cierre en la capa de transporte:** QUIC puede indicar a la capa de aplicación que ha cerrado la conexión por múltiples motivos (error de transporte, cambios la red que pierden la conectividad, etc.).

11. Protocolo HTTP/3

Semántica HTTP en HTTP/3

Envío de mensajes

- Un cliente enviará una sola petición HTTP en un stream de petición, el cual será un stream bidireccional de QUIC iniciado por el cliente.
  - Un cliente enviará solamente una petición, y el servidor enviará, ninguna o más respuestas interinas (códigos 1xx) seguidas de una respuesta HTTP final.
  - En estos streams pueden intercalarse con la respuesta tramas PUSH_PROMISE enviadas por el servidor.
- Un servidor enviará respuestas no solicitadas (PUSHED) en un stream unidireccional QUIC iniciado por el servidor.
  - El servidor enviará, ninguna o más respuestas interinas (códigos 1xx) seguidas de una respuesta HTTP final, como para una respuesta estándar.

11. Protocolo HTTP/3

Semántica HTTP en HTTP/3

Envío de mensajes

Mensajes HTTP

- Un mensaje HTTP (petición o respuesta) consiste en:
  - La sección de cabeceras, incluyendo los datos de control, enviados como una sola trama HEADERS.
  - Opcionalmente, el contenido, si está presente, enviado como una serie de tramas DATA, y
  - Opcionalmente, la sección tráiler, si está presente, enviada como una sola trama HEADERS.
- Un intercambio de petición/respuesta de HTTP consume completamente un stream bidireccional QUIC iniciado por el cliente:
  - El cliente debe cerrar el stream para envío una vez haya enviado la petición.
  - El servidor cerrará el stream para envío una vez haya enviado la respuesta final, quedando el stream completamente cerrado.
- Las peticiones pueden ser rechazadas (no se han procesado) o canceladas (se han procesado total o parcialmente) por el servidor.

11. Protocolo HTTP/3

Semántica HTTP en HTTP/3

Campos de cabecera y datos de control (pseudo cabeceras)

- Los nombres de los campos de cabecera deben enviarse en minúscula.
- Los campos de cabecera se comprimen con el algoritmo QPACK (RFC 9204).
- Una implementación puede imponer límite al tamaño de los campos de cabecera. Emplea el parámetro SETTINGS_MAX_FIELD_SECTION_SIZE para informar.
- Las **pseudo cabeceras** (comienzan con ":") , al igual que en HTTP/2 llevan la información de control de un mensaje.
  - No son campos de cabecera.
  - Siguen las mismas condiciones que en HTTP/2.
  - Existen las mismas que en HTTP/2.
- En HTTP/3 no se define una forma de incluir el identificador de la versión, como sí lo hace HTTP/1.1 por lo que se asume implícitamente la versión "3.0"

11. Protocolo HTTP/3

Semántica HTTP en HTTP/3

Server push

- Al igual que en HTTP/2, un envío desde el servidor (*server push*) permite anticipar un recurso antes de que el cliente lo solicite.
  - Un *server push* supone un mayor uso de la red en favor de conseguir una potencial mejora en la latencia.
  - En general en HTTP/3 un *server push* es similar a lo descrito en HTTP/2 pero emplea diferentes mecanismos.
  - Solo se puede hacer push si se cumple que la petición sea cacheable, segura y no incluya datos de petición ni tráiler.
- A cada server push se le asigna un push ID y comienzan en 0.
  - El número máximo de *server push* se limita con la trama MAX_PUSH_ID.
  - El servidor no puede hacer *push* hasta recibir del cliente MAX_PUSH_ID.
- El servidor emplea el *push* ID en la trama PUSH_PROMISE para enviar los datos de control y campos de cabecera del mensaje de petición.
- La trama CANCEL_PUSH permite a un cliente rechazar un *push* y al servidor informar de que finalmente no completará un *push*.

11. Protocolo HTTP/3

Mapeado y uso de los streams

- Un stream QUIC proporciona a HTTP/3 un envío de bytes ordenado y confiable, pero no garantiza el orden de envío entre diferentes streams.
  - Los streams de QUIC pueden ser unidireccionales o bidireccionales y pueden ser iniciados tanto por el cliente como por el servidor.
- **Streams bidireccionales (request stream):** solo son iniciados por el cliente para enviar peticiones HTTP y recibir respuestas. La primera petición será por el stream QUIC 0 y las siguientes por el 4, 8,…
- **Streams unidireccionales:** pueden ser iniciados tanto por el cliente como por el servidor para diversos propósitos. Por el momento solamente streams de control y de push, y dos tipos más definidos por QPACK.
  - Cada extremo abre un solo stream de control (tipo 0x00), fundamentalmente para enviar tramas SETTINGS.
  - Streams push (tipo 0x01) iniciados por el servidor para enviar una respuesta antes de que se haga la petición.
  - QPACK requiere dos streams adicionales.

11. Protocolo HTTP/3

Capa de trama de HTTP/3

- Las tramas de HTTP/3 son transportadas por streams QUIC de los tres tipo vistos: petición, control y push.
- Formato de la trama:
  - Tipo de trama (type): entero de longitud de variable.
  - Longitud de los datos (length): entero de longitud variable que describe la longitud en bytes de la carga útil de la trama (payload).
  - Carga útil de la trama (payload): datos transportados por la trama cuyo significado está determinado por el tipo.

```
HTTP/3 Frame Format {
  Type (i),
  Length (i),
  Frame Payload (..),
}
```

2MSB	Long	Bits usables	Rango
00	1	6	0-63
01	2	14	0-16383
10	4	30	0-1073741823
11	8	62	0-4611686018427387903

Codificación de enteros de longitud variable definida en la RFC 9000 de QUIC

11. Protocolo HTTP/3

Capa de trama de HTTP/3

- Las tramas de HTTP/3 son transportadas por streams QUIC de los tres tipo vistos

Trama	Control Stream	Request Stream	Push Stream
DATA	No	Sí	Sí
HEADERS	No	Sí	Sí
CANCEL_PUSH	Sí	No	No
SETTINGS	Sí (solo al principio)	No	No
PUSH_PROMISE	No	Sí	No
GOAWAY	Sí	No	No
MAX_PUSH_ID	Sí	No	No
Reserved	Sí	Sí	Sí

11. Protocolo HTTP/3

Capa de trama de HTTP/3

Tipos de trama

- **DATA (tipo=0x00):** lleva datos arbitrarios de cualquier longitud.
- **HEADERS (tipo=0x01):** lleva una sección de campos de cabecera codificada con QPACK.
- **CANCEL_PUSH (tipo=0x02):** se emplea para cancelar un *server push* con un *push* ID concreto, antes de que el *stream push* se reciba. Puede enviarla el cliente o el servidor.
- **SETTINGS (tipo=0x04):** lleva parámetros de configuración que afectan a cómo se comunican ambos extremos y siempre se aplican a toda la conexión HTTP/3 y no solamente a un stream. Solo se debe enviar como la primera trama de un stream de control.

11. Protocolo HTTP/3

Capa de trama de HTTP/3

Tipos de trama

- **PUSH_PROMISE (tipo=0x05):** lleva una sección de cabecera de una petición que se promete al cliente desde el servidor y se identifica con un *push* ID. Se envía en el *request stream* en el que se envió la petición que da lugar al *push*.
- **GOAWAY (tipo=0x07):** se emplea para iniciar un cierre acordado de la conexión HTTP/3 por cualquiera de los extremos. Siempre se envía por el stream de control.
- **MAX_PUSH_ID (tipo=0x0d):** el cliente informa al servidor la cantidad de *server push* que puede hacer el servidor. Siempre se envía en un stream de control.

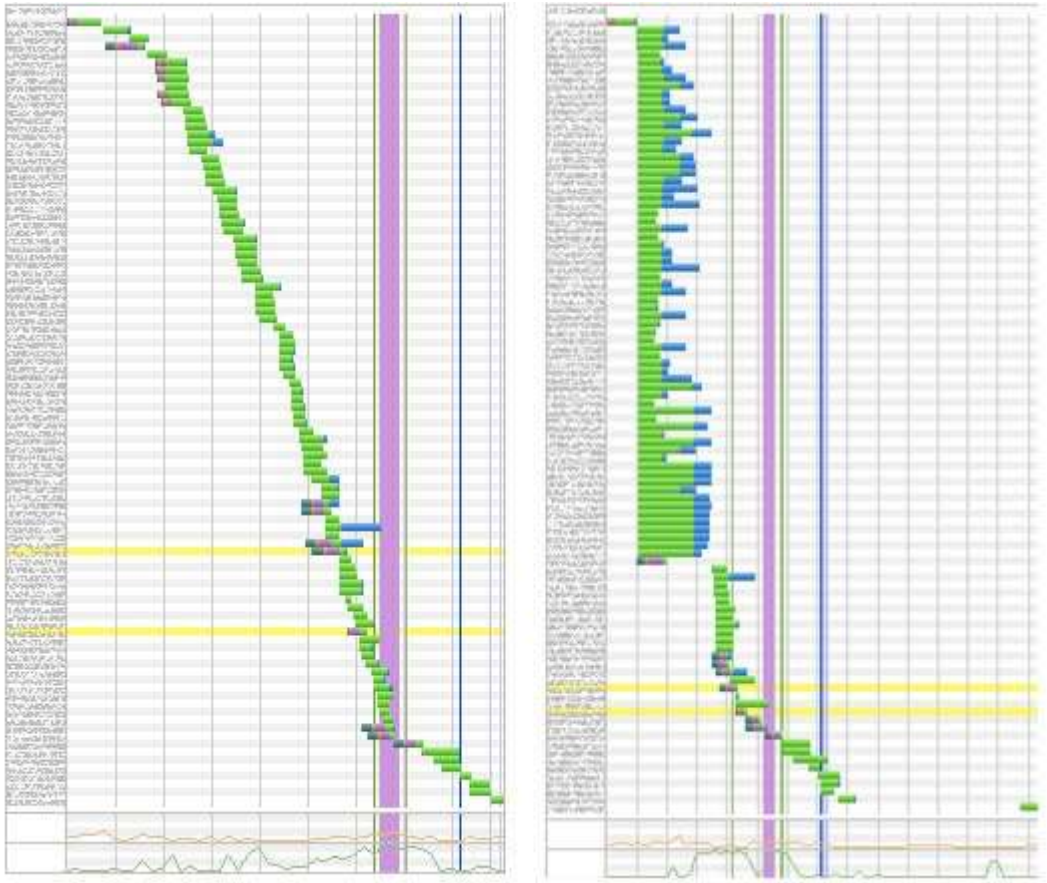
11. Protocolo HTTP/3

Manejo de errores en HTTP/3

- En HTTP/3, aparte de los códigos de estado de error, que son resultados de peticiones completamente procesadas, existen errores de stream y errores de conexión, los cuales los comunica QUIC.
- Una aplicación puede decidir terminar abruptamente un stream y comunicar una razón: esto se conoce como un **error de stream**.
- Igualmente, si la conexión al completo tiene que ser terminada, QUIC proporciona mecanismos para comunicarlo, lo cual se refiere como un **error de conexión**.

12. Rendimiento en comunicaciones web

- Conseguir un adecuado rendimiento de las comunicaciones entre cliente y servidor web depende de:
  - Los protocolos que se emplean a nivel de transporte y aplicación.
  - Cómo se usan estos protocolos.
  - Cómo se diseña la información que se intercambia.
  - Si hay accesos a la información de terceros.
- Existen recomendaciones para conseguir un mejor rendimiento.
- No es igual emplear HTTP/3, HTTP/2 o HTTP/1.1



HTTP/1.1 HTTP/2

12. Rendimiento en comunicaciones web

Comparación de medidas de optimización de rendimiento

- **Concatenación**: no es tan útil en HTTP/2 o HTTP/3 dado que la sobrecarga de cabeceras es mucho menor y se hace innecesario.
  - Además, concatenar archivos es complejo de especificar y puede derivar a descargar código que no se emplea en la página.
- **Minimización** del código: es una buena práctica a mantener en HTTP/2 o HTTP/3.
- ***Domain sharding*** (fragmentación de conexiones): HTTP/2 está optimizado para aprovechar al máximo una única conexión TCP (aunque algunos estudios han probado que en ciertas condiciones la multiplicidad de conexiones puede ser útil).
  - Con HTTP/1.1 los navegadores solían lanzar seis conexiones paralelas para evitar el bloqueo de peticiones (HOL – *Head of Line*) y conseguir mayor rendimiento en el ancho de banda disponible limitado al principio por TCP.
  - Con HTTP/3 al emplear QUIC, más flexible que TCP tampoco necesitaría esta medida.

12. Rendimiento en comunicaciones web

Comparación de medidas de optimización de rendimiento

- **Dominios sin cookies** para recursos estáticos: deben evitarse al igual que el anterior, ya que además la compresión de cabecera reduce el impacto de las cookies.
- ***Spriting*** *(Las imágenes de la página se incluyen en una imagen más grande y son obtenidas a través de CSS)*: debe evitarse en HTTP/2 o HTTP/3, dado que introduce una complejidad innecesaria en el diseño de CSS.

Sprite empleado en la web de la UJA: https://www.ujaen.es/themes/custom/ujaen_base/images/sprite.png

13. Websockets

- Los websockets permiten que un cliente web y un servidor web inicien una comunicación bidireccional interactiva sobre un socket TCP sin las limitaciones de HTTP.
- Los websockets están definidos por un API y un protocolo:
  - Hickson, I., "The WebSocket API", W3C Working Draft WD-websockets-20110929, September 2011, http://www.w3.org/TR/2011/WD-websockets-20110929/. Última versión disponible en http://www.w3.org/TR/websockets/.
  - RFC 6455: The WebSocket Protocol: http://tools.ietf.org/html/rfc6455
    - Actualizaciones RFC7936, RFC 8307 y RFC 8441

13. Websockets

Trasfondo

- En muchas aplicaciones web (como juegos o aplicaciones de mensajería), para obtener la información actualizada del servidor era necesario realizar peticiones constantes al servidor (*poll* o *polling*).
- Además:
  - Cada petición requería de una nueva conexión TCP.
  - El protocolo empleado (HTTP) añade una gran sobrecarga dada la necesidad en seguir enviando sus cabeceras.
  - La complejidad en el cliente aumentaba ya que era necesario mantener muchas conexiones TCP para esas peticiones.
- Sin embargo:
  - Esto podría hacerse con conexiones TCP a nivel de transporte, pero se perdería la versatilidad de las comunicaciones sobre HTTP (proxys, autenticación, filtrado, etc.).

13. Websockets

Características

- El protocolo WebSocket asimila los beneficios de las comunicaciones bidireccionales existentes con HTTP aprovechando su infraestructura:
  - Funciona sobre los puertos 80 y 443, así como soporta proxys HTTP e intermediarios.
  - El protocolo tiene dos fases fundamentales: saludo (*handshake*) y transmisión de datos (*base data framing*).
  - Es un protocolo basado en el intercambio de tramas que añade direccionamiento y nombrado para permitir varios servicios sobre un puerto y varios nombres de host en una única IP.
  - Incorpora un mecanismo de seguridad basando en el origen de las comunicaciones.
  - Incluye un mecanismo de cierre de la comunicación “en banda” que funciona incluso en la presencia de proxys y otros intermediarios.

13. Websockets

Características

URI de un websocket

- Los websockets definen dos nuevos esquemas de URI [RFC3986] que se definen en ABNF:

```
ws-URI = "ws:" "//" host [ ":" port ] path [ "?" query ]
wss-URI = "wss:" "//" host [ ":" port ] path [ "?" query ]

host = <host, definido en [RFC3986], Sección 3.2.2>
port = <port, definido en [RFC3986], Sección 3.2.3>
path = <path-abempty, definido en [RFC3986], Sección 3.3>
query = <query, definido en [RFC3986], Sección 3.4>
```

- <port> es opcional; el valor por defecto para “ws” es el Puerto 80, mientras que para "wss" es el puerto 443.
- Los fragmentos no tienen significado en las URIs de WebSocket y no se deben usar, además, si el carácter "#“ apareciese sin indicar fragmento, se debe sustituir por %23.

13. Websockets

Características

Handshake

- El cliente y el servidor intercambian peticiones y respuestas HTTP completas, aunque con algunas cabeceras especiales de websockets.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

CLIENTE

Solamente una respuesta con 101 es considerada como un establecimiento de la conexión WebSocket correcta.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

SERVIDOR

13. Websockets

Características

Handshake

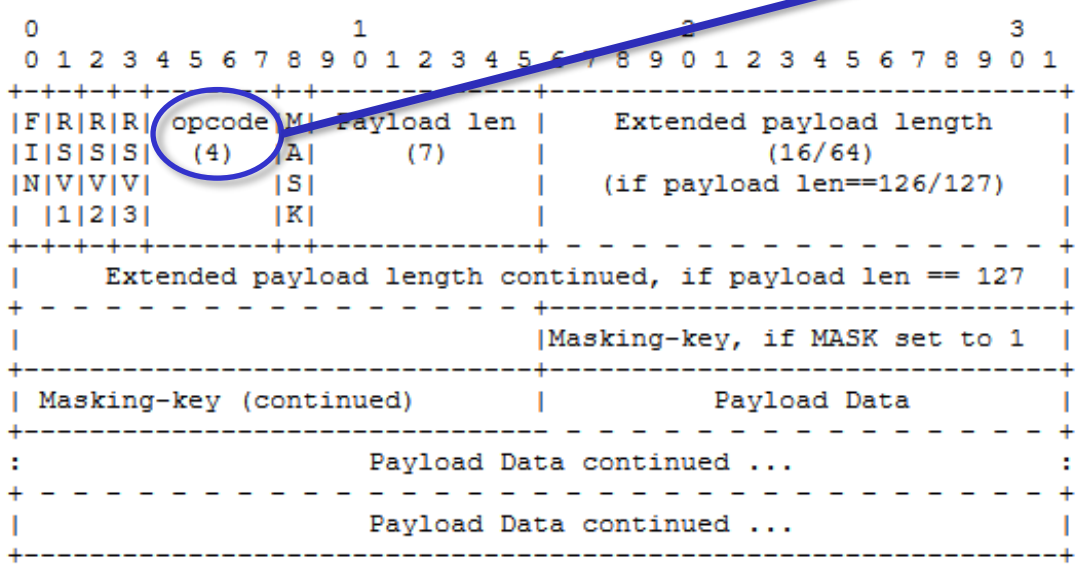
- Aparte de la línea de petición, las cabeceras más importante para establecer una conexión WebSocket son:
  - `Origin`: para evitar peticiones de dominios que no acepta el servidor.
  - `Sec-WebSocket-Key`: comunica al servidor una cadena a la que el servidor debe añadir la cadena "258EAFA5-E914-47DA-95CA-C5AB0DC85B11“ que es es un identificador global único (*Globally Unique Identifier* o GUID). Después devuelve un hash SHA-1 en BASE64 de los datos concatenados en la cabecera `Sec-WebSocket-Accept`.
    - El cliente una vez recibido Sec-WebSocket-Accept debe calcularlo también para comprobar que coinciden y que efectivamente está en comunicación con un servidor WebSocket.
  - `Sec-WebSocket-Protocol`: informa del sub-protocolo a nivel de aplicación que se va a emplear, uno de los más típicos es “chat”.
  - `Sec-WebSocket-Version`: informa de la versión preferida por el cliente. Puede ser aceptada o modificada por el servidor.

13. Websockets

Características

Base framing protocol

- Encapsula la información de la aplicación en tramas, las cuales se pueden fragmentar.
- Existen tramas de control (no se fragmentan) y deben tener un *payload* de 125 bytes o menos.



Datos (opcode=0xxx):
- 0x01 (Text UTF-8)
- 0x02 Datos (Binario)
- 0x03-0x07 Reservados

Control (opcode=1xxx):
- 0x08 Close
- 0x09 Ping
- 0x0A Pong
- 0x0B-0x0F Reservados

Formato de las tramas de datos
Fuente: RFC 6455 The WebSocket Protocol. December 2011

13. Websockets

Características

Base framing protocol

- `FIN`: indica si es “1” el fragmento final de un mensaje, lo cual implica que el primer fragmento puede ser también el último.
- `RSV1, RSV2, RSV3`: Deben ir a “0” salvo que alguna extensión le de significado al valor “1”.
- `MASK`: si vale “1” los datos irán enmascarados.
- `Payload len`: indica la longitud de los datos si esta lleva entre 0 y 125 bytes. Si es 126, los siguientes 16 bits se interpretan como un entero sin signo como longitud de los datos. Si es 127 son los siguientes 8 bytes (64 bits) los que se interpretan como un entero sin signo como la longitud de los datos.
- `Masking key (`0 o 4 bytes): contiene la clave, no predecible, con la que el cliente enmascara toda la información que envía.
  - La operación de enmascarado en una XOR en bloques de 32 bits de la máscara y los datos del payload.
- `Payload data`: es el conjunto de los datos de una extensión (si los hay) y los datos de aplicación.