Tema 2. Web

Ingeniería de Servicios

2023-2024

Introducción

En este tema **servicios web** se refiere a aquellos prestados por un servidor web y "consumidos" por un navegador web.

> No nos referimos al acceso a servicios software (métodos) a través de interfaces Web como SOAP
> Esto es objeto de otras asignaturas

Resumen de este tema

1. Arquitectura básica (recursos, URIs, transacciones)
2. Sintaxis de los mensajes HTTP (métodos, códigos de estado)
3. Cabeceras HTTP
4. Proxies y cachés
5. Páginas generadas dinámicamente
6. Sesiones, cookies
7. Autenticación

1. Arquitectura básica de la web

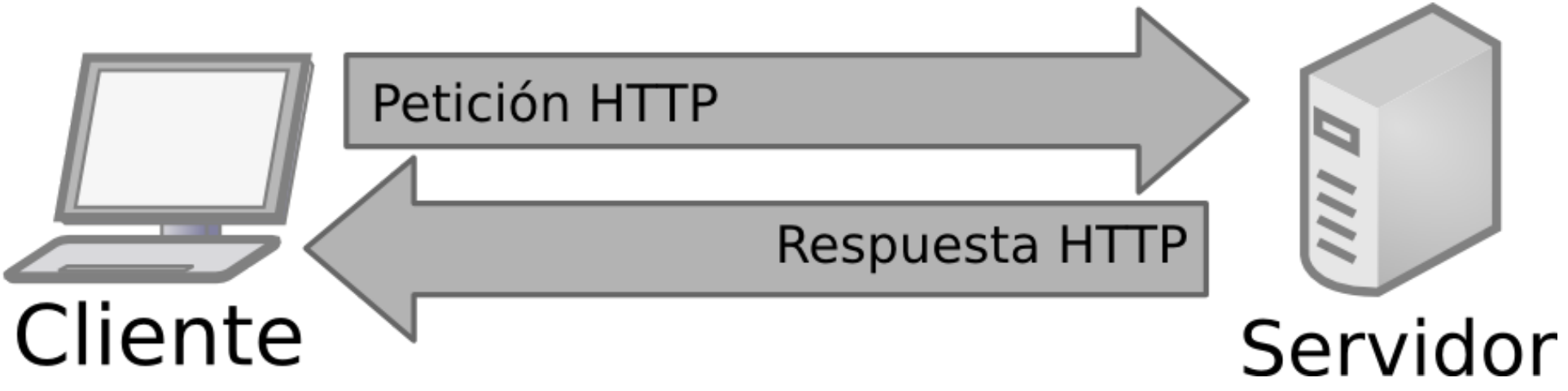
En su forma más simple, consiste en:

- Un servidor que almacena *recursos*
- Un cliente que los solicita al servidor y los muestra
- Un protocolo para solicitar y transmitir esos recursos
  - Basado en TCP
  - Cierre de la conexión tras cada envío de la respuesta
- Un lenguaje de marcado (HTML) en el que codificar recursos de tipo hipertexto.

Pero esta versión simple se fue complicando.

Protocolo HTTP

- Cliente y servidor interactúan mediante **transacciones**
- Que son una pareja **petición/respuesta**
  - El cliente pide **recursos**
  - Por cada petición debe haber una respuesta



Petición y respuesta se estructuran en:

- **Sobre**: Línea inicial + cabeceras estandarizadas
- **Contenido**: Cualquier secuencia de bytes

Las peticiones actúan sobre Recursos:

- Un **recurso** es un **contenido binario** que lleva asociado un **tipo**.
- Originalmente se trataba de archivos en disco (ej: una imagen `.jpg`)
- En la actualidad pueden no existir realmente en disco, sino ser generados cuando se piden (*recursos dinámicos*)

Tipos MIME

Son un estándar usado por HTTP para etiquetar el tipo de los contenidos.

Ejemplos:

Tipo	Tipos MIME
Texto	`text/html`, `text/plain`, `text/css`, `text/xml`
Imagen	`image/png`, `image/jpeg`, `image/gif`
Audio	`audio/mpeg`, `audio/ogg`,
Vídeo	`video/mpeg`, `video/webm`, `video/x-ms-wmv`
Ejecutables	`text/javascript`, `application/x-shockwave-flash`
Otros	`application/zip`, `application/pdf`, `application/vnd.ms-excel`

URI

Un URI (*Universal Resource Identifier*) es el nombre que el cliente usa para referirse a un **recurso**. Puede ser:

- **URL** (*Universal Resource Locator*) es un nombre que indica dónde encontrar el recurso.
- **URN** (*Universal Resource Name*) es un nombre independiente de la localización del recurso.

Ejemplos

- URL `http://www.uniovi.es/estudios/grados`
- URN `urn:isbn:978-84-239-4277-0`

Partes de una URL

❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾

`http://juan:1xZ3@www.sitio.com:1234/ruta/hacia/el/recurso.php;gr=true?q=23&r=12#fin`

Parte	Ejemplo
❶ Esquema	`http`
❷ Usuario	`juan` (opcional)
❸ Clave	`1xZ3` (opcional)
❹ Nodo	`www.sitio.com`
❺ Puerto	`1234` (opcional, 80 por defecto)
❻ Ruta	`ruta/hacia/el/recurso.php`
❼ Parámetros	`gr=true` (opcional)
❽ Query	`q=23&r=12` (opcional)
❾ Trozo	`fin` (opcional)

Caracteres en una URL

La sintaxis de la URL se definió de modo que:

- Pueda transmitirse bajo diferentes protocolos sin que se "desfiguren" caracteres.
  - Implica restringirse a ASCII
- Sea legible para el usuario
  - Implica prohibir caracteres invisibles o no imprimibles (espacios!)
- Sea completa, de modo que se puedan especificar en ellas cualquier carácter.
  - Implica diseñar un modo de "escapar" los no permitidos

Son especiales los caracteres invisibles (espacio, tabulador, etc.) y los siguientes:

```
: ? # [ ] @ ! $ & ' ( ) * + , ; =
```

Tampoco se permiten caracteres no ASCII.

Cualquier caracter no válido se codifica como `%XX`
(XX es el valor del byte en hexadecimal)

Ejemplos:

- Espacio → `%20`
- Eñe (UTF-8) → `%C3%B1`
- Tanto por ciento → `%25`

2. Sintaxis de los mensajes

Sintaxis de los mensajes HTTP

Son el contenido de las **peticiones** y **respuestas**. Su estructura es la misma para ambas:

- **Primera línea** terminada por `\r\n`
  - En la petición contiene el **método** solicitado
  - En la respuesta contiene el **código de estado**
- **Campos de cabecera** (la mayoría son opcionales)
  - Cada uno en una línea terminada por `\r\n`
  - Una línea en blanco (`\r\n`) marca el fin de la cabecera
- **Contenido** (formato libre, según su tipo).
  - Puede estar ausente

Sintaxis de la primera línea de un mensaje

En el mensaje de **petición**:

```
<Método HTTP> <URL> <Versión HTTP soportada por el cliente>
```

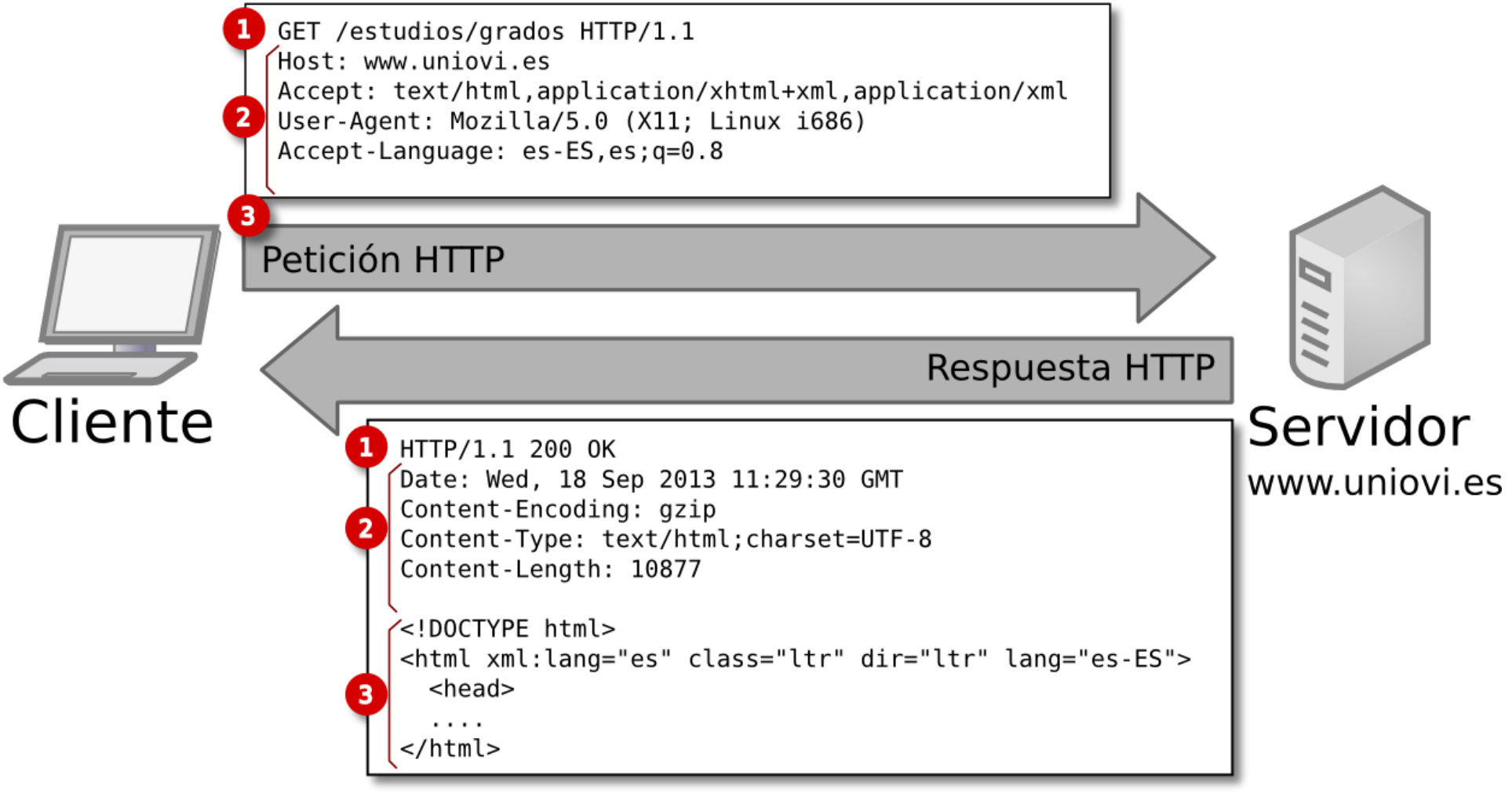
En el mensaje de **respuesta**:

```
<Versión HTTP servidor> <Código de estado> <Frase explicativa>
```

Ejemplos:

```
Petición:    GET /estudios/grados HTTP/1.1
Respuesta:   HTTP/1.0 200 OK
```

Ejemplo de mensajes



Métodos HTTP

Son las distintas acciones que el cliente puede solicitar:

Método	Acción
`GET`	Pide un recurso por su nombre
`PUT`	Envía datos al servidor para que los almecene en el recurso nombrado
`DELETE`	Borra el recurso nombrado del servidor
`POST`	Pide al servidor que envíe datos a una aplicación de entrada (*gateway*)
`HEAD`	Pide sólo las cabeceras de respuesta para el recurso nombrado
`OPTIONS`	El cliente solicita qué opciones existen sobre un recurso dado.

Y otros definidos en extensiones (ej, WebDAV)

Códigos de estado

Son códigos numéricos estandarizados que indican si la acción tuvo éxito o no, y la causa.

Grupos:

Rango	Significado
100-199	Información (ej: 100 Continue)
200-299	Exito (ej: 200 OK)
300-399	Redirección (ej: 301 Moved permanently)
400-499	Error del cliente (ej: 404 Not found)
500-599	Error del servidor (ej: 501 Not implemented)

Algunos códigos de estado (100 y 200)

- `100 Continue`
  Respuesta a una petición en que el cliente se dispone a enviar muchos datos

- `200 OK`
  La petición se ha completado con éxito. El cuerpo incluye el contenido solicitado.

- `204 No content`
  La petición se ha completado con éxito, pero no hay contenido para devolver (ej: tras DELETE)

Algunos códigos de estado (300)

- `301 Moved permanently`
  El recurso no está, pero se proporciona un URL de dónde encontrarlo
- `302 Found`
  Movido temporalmente, pero usado bajo HTTP/1.0 en respuesta a un POST para indicarle al cliente que debe hacer un GET (y la nueva URL)
- `303 See other`
  Lo mismo que el uso anterior, pero para HTTP/1.1
- `307 Temporary redirect`
  El recurso temporalmente está en otro lugar (y se da el URL), para HTTP/1.1
- `304 Not changed`
  Respuesta a una petición GET del cliente que incluye fecha condicional. Si el contenido no cambió desde la fecha dada, se envía este estado en lugar de un `200 OK`

Ejemplo de redirección

- Petición del cliente

```
GET / HTTP/1.1
Host: es.wikipedia.org
```

- Respuesta del servidor:

```
HTTP/2.0 301 Moved Permanently
Date: Tue, 04 Dec 2018 07:17:05 GMT
Content-Type: text/html; charset=utf-8
Content-length: 0
location: https://es.wikipedia.org/wiki/Wikipedia:Portada
```

Algunos códigos de estado (400)

- `400 Bad request`
  La petición enviada por el cliente no es correcta sintácticamente
- `401 Unauthorized`
  El recurso solicitado está protegido. El cliente debe enviar credenciales.
- `403 Forbiden`
  El cliente no puede acceder a ese recurso (incluso si se ha autenticado)
- `404 Not found`
  El recurso no se encuentra, ni se sabe de él

Algunos códigos de estado (500)

- `500 Internal server error`
  El servidor ha encontrado un error al tratar de servir la petición, pero no hay más detalles

- `501 Not implemented`
  El método solicitado no está implementado en este servidor

3. Cabeceras HTTP

Restantes líneas del sobre

Las restantes líneas son **cabeceras**, todas con la misma estructura:

```
Nombre-de-la-cabecera: valor de la cabecera<CRLF>
```

- El **nombre** no puede contener espacios. Muchos nombres posibles están estandarizados.
- El **valor** depende de cada caso (veremos algunos)
- `<CRLF>` es una forma de representar la secuencia de bytes 13, 10 (retorno de carro y nueva línea, o `"\r\n"`)
- Las cabeceras terminan cuando aparece una línea en blanco `<CRLF><CRLF>`

Tipos de cabecera

- **Generales**. Pueden aparecer tanto en peticiones como en respuestas
- **De petición**. Sólo aparecen en los mensajes de petición.
- **De respuesta**. Sólo aparecen en los mensajes de respuesta.
- **De entidad**. Describen en cuerpo (tamaño, tipo)
- **De extensión**. Ampliaciones y otros usos no estandarizados.

Ejemplos de cabeceras

Cabecera	Descripción
`Date: Tue, 3 Oct 1997 02:16:03 GMT`	Fecha y hora en que se genera la respuesta
`Content-length: 15040`	Tamaño en bytes del cuerpo
`Content-type: image/gif`	Tipo MIME del cuerpo
`Accept: image/gif, image/jpeg, text/html`	Tipos que admite el cliente

Algunas cabeceras generales

- `Date:`
  Fecha y hora en que se genera el mensaje. El formato es espefíco de HTTP.

- `Connection:`
  Tipo de conexión. En una petición el cliente indica qué tipo prefiere, en la respuesta el servidor le dice qué tipo tiene. El valor típico es `keep-alive` que dica que no se cierre el *stream* TCP. Cuando el cliente quiere cerrar, envía en su última petición un `Connection: close`

- `Cache-control:`
  Política de cache (si se aceptan contenidos *cacheados* o no). Veremos más después.

Algunas cabeceras de la petición

- `Host:`
  Es obligatorio. Debe incluir el nombre del host a que se envía la petición. Importante para *hosts virtuales*.

- `User-Agent:`
  Cadena de texto que identifica a la aplicación cliente que hace la petición (nombre del navegador, operativo, etc.)

- `Accept:`
  Lista de tipos MIME que el cliente puede aceptar

- `Accept-Encoding:`
  Lista de métodos de codificación del contenido que el cliente admite. Por ejemplo `gzip`, `identity`.

Cabeceras en la petición para usos más avanzados

- `If-None-Match:`
  Optimización que permite al cliente no descargar un recurso que ya tiene (ya lo veremos)

- `Cookie:`
  El cliente envía las *cookies* para ese servidor (ya lo veremos)

- `Authorization:`
  El cliente aquí envía sus credenciales para recursos que los pidan (ya lo veremos)

Cabeceras en la respuesta

- `Server:`
  Cadena de texto que identifica al servidor y su versión (ej, apache)
- `WWW-Authenticate:`
  Especifica qué tipo de autenticación debe usar el cliente (ya lo veremos)
- `Set-Cookie:`
  Envía una *cookie* al cliente (ya lo veremos)
- `Transfer-Encoding:`
  Método usado por el servidor para enviar el cuerpo. Puede ser `chunked`.

- **Por trozos**, cada trozo va precedido por un número (*no cabecera*) que indica su longitud. El último con `0`
- **De una vez**, la cabecera `Content-Length`, indica en ascii el número de bytes del cuerpo.

Cabeceras de entidad

- `Content-Type:`
  Tipo MIME del cuerpo
- `Content-length:`
  Longitud (en bytes) del cuerpo
- `Content-Language:`
  Idioma del cuerpo (si es texto)
- `Etag:`
  Identificador del recurso usado para optimizaciones de caché (ya lo veremos)

4. Proxies y cachés

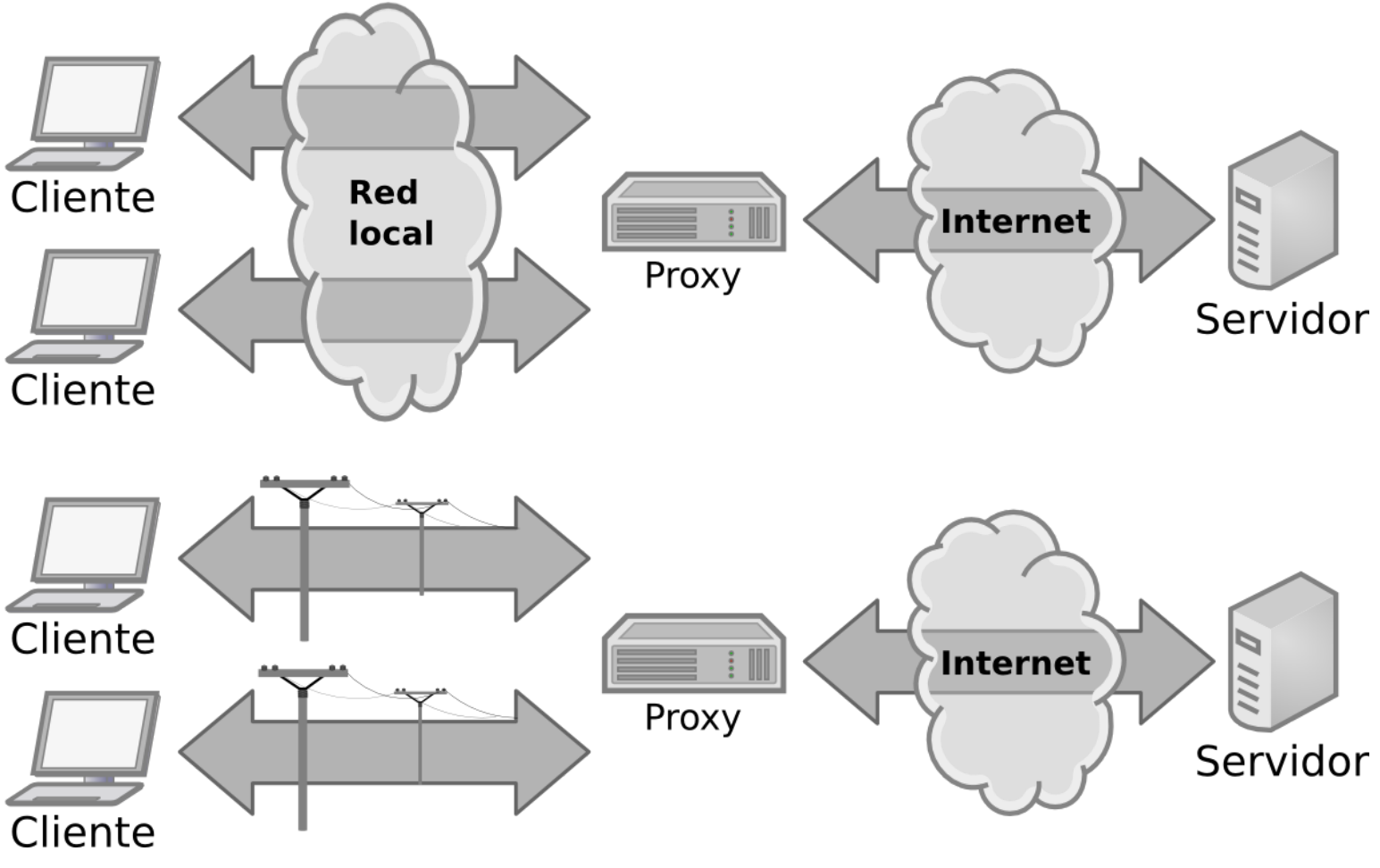
Proxies

- Cliente : Programa que solicita el recurso
- Servidor : Programa que suministra el recurso

Proxies
Servidores intermediarios entre el cliente y otros servidores

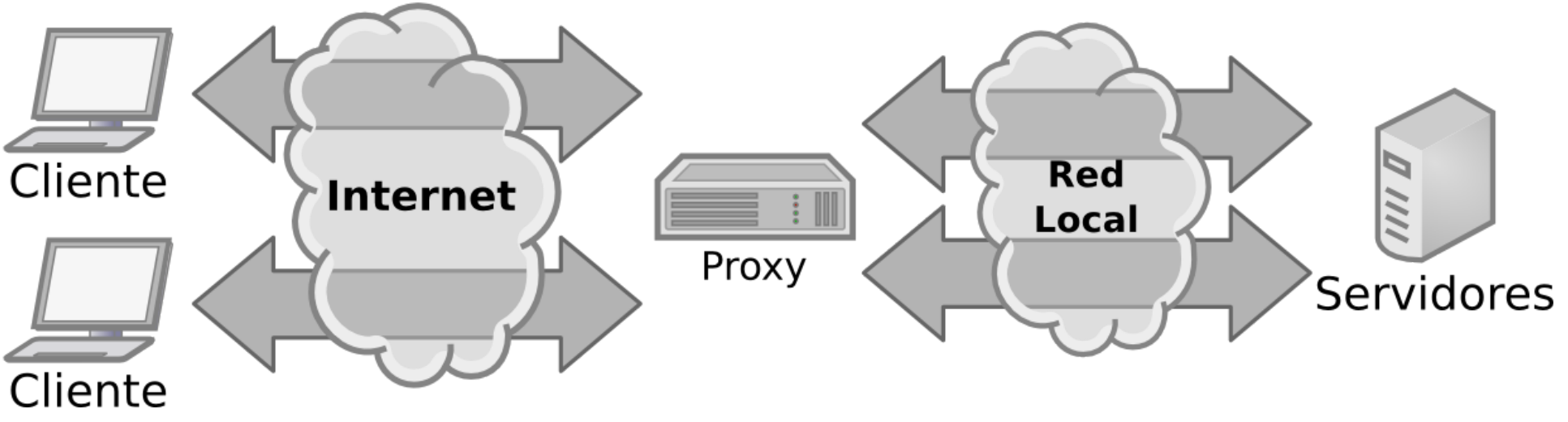
- Proxy directo
  Sirve peticiones dirigidas otros servidores (pidiendo el recurso al servidor que corresponda)
- Proxy inverso (surrogado)
  Ante el cliente parece un servidor normal, pero obtiene el recurso de otros servidores

Ejemplos de proxy directo



Puede filtrar contenidos, mejorar la eficiencia (caché). Puede ser *transparente* (el usuario no requiere configurar su existencia).

Ejemplo de proxy inverso



Usado para mejorar la eficiencia (balanceo de carga, caché, aceleración de cifrado SSL) y la seguridad.

Caché

Es un almacén más cercano y rápido que contiene copias de recursos para no tener que pedirlas de nuevo

- La *caché local*, que es la copia que mantiene el navegador y que es privada (sólo el usuario con acceso a ese navegador podrá acceder a esa cache).
- La *caché remota*, que es la copia que mantienen los servidores intermedios, y que en principio es compartida por todos los usuarios.

¿Por qué se usan?

- Se reduce el uso de la red.
- Se reduce el tiempo necesario para obtener el recurso.

Pero...

¿Cómo sabe el cliente si la copia que mantiene en cache ha dejado de ser válida?

> Recordemos que en HTTP las peticiones siempre se inician en el cliente. El servidor no puede notificarle si un recurso cambia.

Códigos de respuesta relacionados con cachés

El cliente tiene diferentes formas (cabeceras de petición) para preguntar

> "¿Esta copia que tengo del recurso ha sido actualizada en el servidor?"

El código de respuesta del servidor será uno de estos:

- `304 Not changed` El recurso no ha sido actualizado (la copia que tiene el cliente en caché sigue siendo válida).
- Otros códigos en caso contrario (ej: el `200 OK` para indicar que se está enviando en el cuerpo el nuevo recurso actualizado).

Cabeceras relacionadas con cachés

Son las diferentes estrategias para que el cliente sepa cuándo descargar un recurso de nuevo.

- Basado en la fecha de modificación:
  - `If-Modified-Since`
  - `Last-Modified`
- Basado en el cambio del contenido:
  - `If-None-Match`
  - `ETag`
- Basado en fecha de caducidad:
  - `Cache-Control`

If-Modified-Since

Se basa en la fecha del último cambio.

- El cliente solicita por primera vez un recurso
- El servidor (tal vez) le devuelve en la cabecera el campo `Last-Modified`
- El cliente almacena en su caché local el recurso, junto con esa fecha de última modificación
- Cuando el cliente necesita de nuevo el recurso usa el campo `If-Modified-Since` con la fecha guardada
- Si no ha cambiado desde entonces, el servidor retorna `304`. Si ha cambiado, retorna el nuevo contenido junto con una nueva `Last-Modified`

Etags

- Se basa en el contenido del recurso

> Un *Etag* es un *hash* del contenido del recurso.
> Si el recurso cambia, cambiará su *Etag*

- Junto con el recurso el servidor envía ese *hash* en una cabecera `Etag:`
- El navegador almacena ese *hash* junto con el recurso
- Cuando necesita de nuevo el recurso, usa la cabecera `If-None-Match:` en la petición, dándole el valor del *hash* almacenado.
- El servidor compara el *hash* que viene en `If-None-Match:` con el que tiene el recurso. Si son iguales, devuelve un `304`. Si no, devuelve el recurso con su nuevo `Etag:`.

Caducidad (Frescura)

Se basa en el tiempo estimado entre cambios

- Junto con el recurso el servidor envía un campo `Cache-control:` donde especifica el tiempo de vida del recurso (`max-age`)
- El cliente no volverá a pedir ese recurso hasta que expire el tiempo
- Si el tiempo ha expirado, lo pedirá de nuevo

Evita el tráfico de las peticiones/respuesta 304

Pero es un problema si el recurso cambia antes

Es ineficiente si el tiempo ha expirado pero el recurso no ha cambiado

Cache remota

- El cliente pide un recurso
- Un proxy-caché intermedio gestiona la petición
- Puede que el proxy le devuelva una copia, en lugar de pedir a su vez el recurso

¿Puede controlar el cliente si recibirá una copia o el original?

El cliente puede usar la cabecera `Cache-Control: no-cache` en la petición

Control de la cache desde el HTML

El autor de una página web puede incluir en el HTML:

```
<meta http-equiv="Cache-control" content="no-cache">
```

- El servidor web incluirá esa cabecera en su respuesta.
- Lo que indicará a los proxies por los que pase (y al propio cliente), que no lo almacenen

También puede asignarse el valor:

- `public` si puede almacenarse en proxies intermedios
- `private` si puede almacenarse en el cliente (navegador)
- `max-age=3000` si puede almacenarse, pero caduca en 3000 segundos

5. Páginas generadas dinámicamente

¿Qué son?

- No son contenidos almacenados estáticamente en disco
  - ...sino el resultado de ejecutar un programa externo
  - ...cuya salida depende de los parámetros que se le pasen, o de la fecha y hora, etc
  - ...los cuales se obtienen de la URL o de los datos suministrados vía POST

Ejemplos:

- Tiendas online
- Buscadores web
- Redes sociales
- Blogs
- ...prácticamente cualquier página web hoy día.

Tecnologías para contenido dinámico

- CGI (*Common Gateway Interface*)
- Módulos en el servidor web
- Fast CGI, SCGI
- ASP.NET
- Tecnologías Java

CGI (*Common Gateway Interface*)

- Es la tecnología más antigua y la más simple
- El código no se ejecuta en el servidor Web
- ...sino en un proceso externo (lanzado con `fork()`)
- al que se le pasan en variables de entorno los parámetros apropiados
- La salida estándar de ese proceso es lo que el servidor Web envía como respuesta.

La sobrecarga de crear un proceso nuevo por cada petición lo hace inaceptable, y ha quedado obsoleto.

Módulos en el servidor Web

- El servidor web incluye intérpretes de ciertos lenguajes (ej: php, perl, python, ...)
- Por tanto el código se ejecuta como parte del proceso servidor-web
- Es un método muy común en Apache (ej: `mod_php`)
- El mismo servidor sirve contenidos estáticos y dinámicos

No permite tener máquinas separadas y optimizadas a cada caso (estático/dinámico)

En algunos lenguajes (ej php) el código va "embebido" en HTML y el servidor web se ocupa de las cabeceras HTTP. En otros lenguajes (ej python) no.

Fast CGI, SCGI

- Además del servidor Web, existe otro servidor "de aplicaciones"
- Ambos se comunican por sockets,
- Se pasan parámetros y resultados con un protocolo específico binario (ej, Fast CGI)

Ventajas:

- El servidor de aplicaciones está permanentemente cargado, no se crea un proceso por cada petición.
- El servidor de aplicaciones puede estar programado en cualquier lenguaje
- Puede ejecutarse en diferente máquina que el servidor web

Desventaja: es más complejo de configurar que el "módulo en el servidor"

ASP.NET

- Es equivalente al "módulo en el servidor" (similar a php), pero:
  - Propietario de Microsoft
  - Limitado al servidor Microsoft Internet Information Services
  - El lenguaje de programación es uno de la familia .NET

Inconveniente: cerrado

Tecnologías Java

- Java Servlets
  - Es una clase Java que ejecuta el código que genera la página dinámica
  - Se ejecuta dentro de un *contenedor de servlets* (también en Java)
  - El contenedor suele ser un servidor web escrito en Java (ej: Tomcat)
- JSP (JavaServer Pages)
  - Es una mezcla de HTML con java (tipo php)
  - Las partes en java son ejecutadas por un *contenedor de servlets*

Inconveniente: cerrado en torno a Java. Peor respuesta ante sobrecarga que Fast CGI o `mod_*`

6. Sesiones y cookies

El problema

Muchas aplicaciones (ej: carrito de la compra) requieren que el servidor guarde un estado del cliente.

Una sesión es un conjunto de peticiones HTTP que tienen como denominador común el estar dirigidas a un mismo servidor, desde un mismo usuario.

Problema

En HTTP cada petición es una conexión TCP distinta.

El servidor no tiene forma de saber si es el mismo cliente u otro.

El protocolo no implementa el concepto de *sesión*, que debe implementarse de alguna forma haciendo uso de las cabeceras.

Token de sesión HTTP

- Es un identificador único para cada sesión:
  - Generado por el servidor en la primera interacción con el cliente para esa sesión
  - Que se envía a dicho cliente de alguna forma
  - Que el cliente debe proporcionar de nuevo al servidor en cada interacción de esa sesión

Gracias al *token* el servidor "reconoce" al cliente y puede acceder al estado del cliente, típicamente almacenado en una base de datos.

- Mecanismos de transmisión del token:
  - En la url de los métodos HTTP (raro)
  - En *cookies* (habitual)

Token en la URL

Ejemplo:

1. El cliente accede a la URL `http://www.example.com/prueba.php`
2. El servidor responde con una redirección `302 Found` y le da al cliente una nueva URL
3. La nueva URL incluye el token, por ejemplo: `http://www.example.com/prueba.php?sid=a35669ddd8dc45b7aefb705616f99833`
4. El cliente hace un `GET` a esa nueva URL
5. El servidor, analizando la URL, determina el *id* del cliente y sabe que es el mismo de antes.

Cuidado! Si alguien ve la URL y copia el *id* puede hacerse pasar por el cliente a menos que el servidor tome medidas

Cookies

Una *cookie* es una información (una cadena de texto) que el servidor envía al cliente a través de la cabecera `Set-Cookie`.

Se espera del cliente que:

- Almacene esa información (en RAM, en disco, ...) asociada con el servidor que se la envió. *No necesita entender su contenido*.
- La envíe en cada una de las peticiones que vuelva a hacerle a ese servidor, a través de otra cabecera `Cookie`

Esto puede usarse para implementar **sesiones**

Y más cosas...

Sintaxis de las Cookies

La sintaxis general de `Set-Cookie:` (en los mensajes del servidor al cliente) es:

```
Set-Cookie:nombre=valor; atributos
```

- **nombre** cualquier identifiador alfanumérico
- **valor** cualquier secuencia de caracteres (salvo punto y coma). Es "opaco" para el cliente (no significa nada para él)
- **atributos** son parejas `clave=valor;` que definen aspectos de la cookie, como el servidor, la caducidad, etc.
- Si el servidor quiere enviar varias cookies, puede poner varios `Set-Cookie:`

Ejemplo:

```
Set-Cookie:JSESSIONID=a35669ddd8dc45b7aefb705616f99833;
  Expires=Sat, 20-Sep-2014 23:10:00 GMT; Path=/
```

Sintaxis de las Cookies

La sintaxis general de `Cookie:` (en los mensajes del cliente al servidor) es:

```
Cookie:nombre=valor;nombre=valor;...
```

Por tanto el cliente puede devolver varias *cookies* en el mismo campo.

Ejemplo:

```
Cookie:JSESSIONID=a35669ddd8dc45b7aefb705616f99833;
  GUEST_LANGUAGE_ID=es_ES; COOKIE_SUPPORT=true
```

Cookie como token de sesión

El servidor puede usar las Cookies para enviar al cliente una cadena que identifica la sesión.

No confundir con *credenciales*. El Token de sesión puede identificar al cliente aún si el usuario *no tiene cuenta* en el servidor.

Usos de las Cookies

- Identificación de la sesión, para
  - "Carrito de la compra"
  - Mantener al usuario "*logueado*"
- Personalización
  - Recordar las preferencias del usuario
  - Incluso si éste no se identificó
- Seguimiento (estadísticas)
  - Conocer qué páginas del sitio visita el usuario
  - En qué orden lo hace
  - Cuánto tiempo consume en cada una, etc.

En la Unión Europea, por ley, una web no puede usar *cookies* de seguimiento sin antes obtener la aprobación del usuario.

Tipos de Cookies

- **Cookies de sesión**
  - No deben especificar caducidad
  - El navegador debe borrarlas al cerrar
  - Típicamente almacenan un *Token* de sesión
- **Cookies persistentes**
  - Especifican caducidad
  - Se almacenan en disco y no se borran al reiniciar
  - Pueden almacenar preferencias del usuario (idioma, etc)

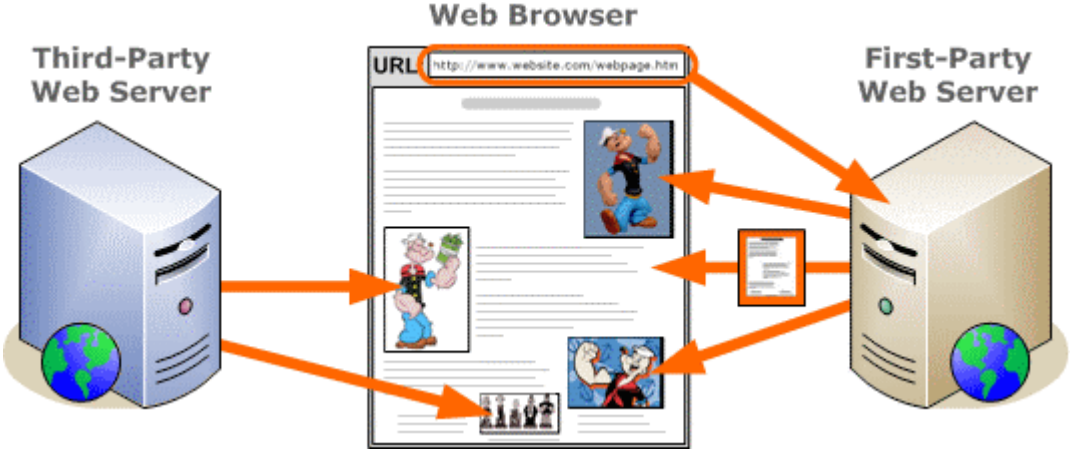
Cualquiera de las dos puede usarse también para seguimiento

Cookies de terceros

Son cookies que se envían a servidores *distintos* del que aparece en la barra de navegación del navegador.

Por ejemplo, estamos navegando por "La Voz de Asturias", y el navegador está enviando *cookies* a Google y a Facebook.

Pero ¿cómo puede ser esto?



7. Autenticación

Recursos protegidos

El servidor puede estar configurado para que algunos recursos sólo puedan ser accedidos por ciertos usuarios, luego necesita conocer quién se lo está pidiendo.

Mecanismos

- Basados en la cabecera `Authorization:`
- Basados en Tokens gestionados por la capa de aplicación
  - En cookies
  - En la URL
  - En el cuerpo de los mensajes

que deben ser usados **en todas las peticiones**.

Cabecera `Authorization`

Al acceder a un recurso protegido, el servidor responderá con:

- `401 Unauthorized`
- Cabecera `WWW-Authenticate` especificando qué método debe usarse

El cliente deberá

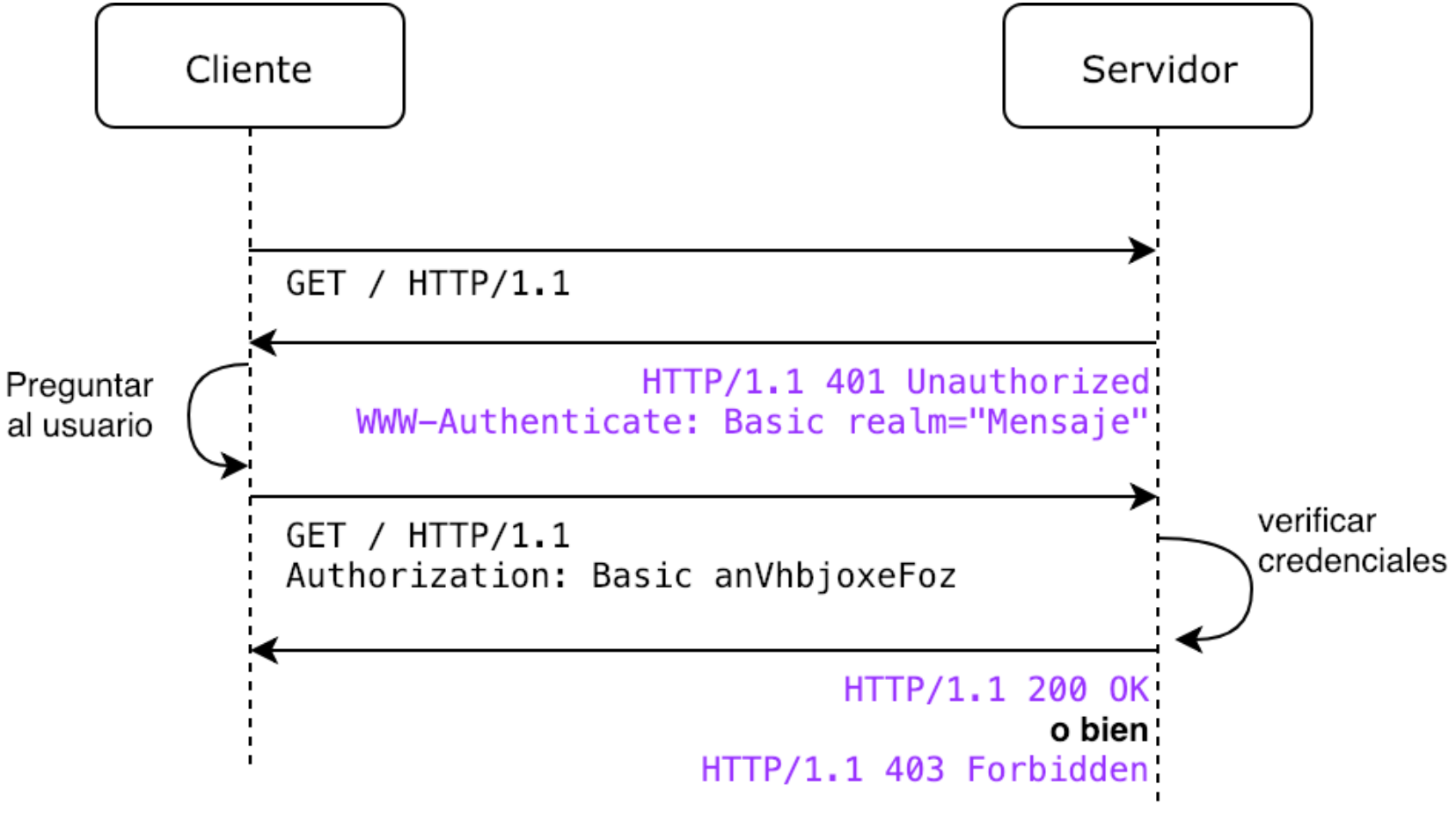
- Solicitar credenciales al usuario
- Repetir la petición con una cabecera `Authorization` con un valor adecuado.

El servidor le permitirá el acceso si el valor de esa cabecera es correcto.

Autenticación "Basic"

No es muy segura, pero es muy sencilla

Ejemplo:

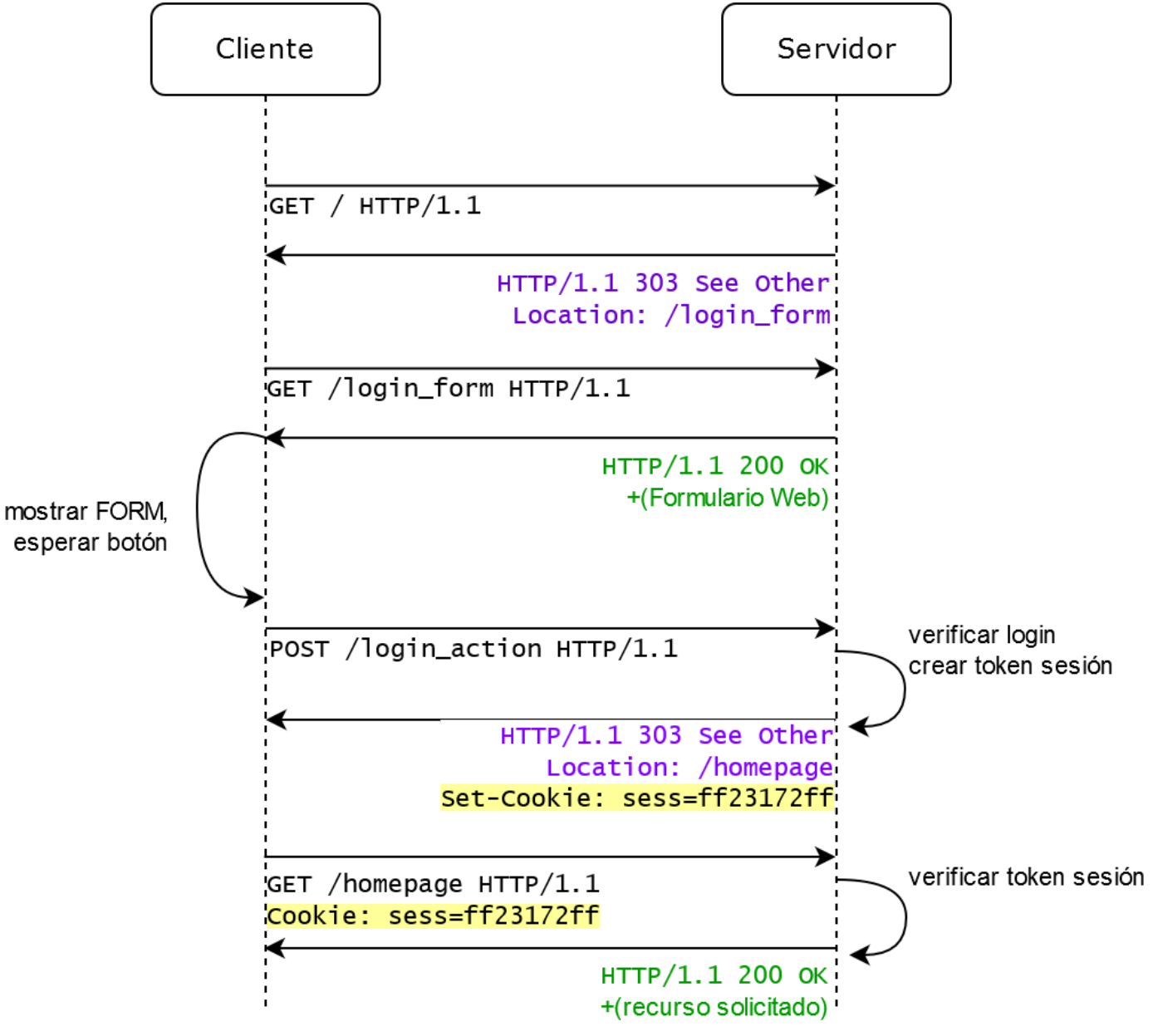


]

Explicación del ejemplo

1. El cliente pide un recurso mediante `GET url`
2. El servidor le responde con `401 Unauthorized`
3. El servidor incluye en la respuesta la cabecera `WWW-Authenticate:Basic`
4. El cliente solicita al usuario nombre y clave (ej: `juan` y `1xZ3`)
5. Junta ambos con dos puntos (`juan:1xZ3`) y codifica en *base64* el resultado (sale `anVhbjoxeFoz`)
6. El cliente repite el GET incluyendo en la cabecera `Authorization: Basic anVhbjoxeFoz`
7. El servidor responde con `200 OK` y sirve el recurso (o con `401` si falla la clave)

Autenticación "en capa de aplicación"



Explicación de la figura anterior

1. El navegador pide el recurso `/` mediante `GET`
2. El servidor le redirige a la página de login
3. El navegador pide el recurso `/login_form`
4. El servidor responde con una página HTML que contiene un formulario
5. El usuario rellena el formulario y pulsa el botón "Enviar"
6. El navegador hace un `POST` a la dirección especificada en el formulario
   - El POST contiene los datos del formulario rellenado (login, contraseña)
7. El servidor extrae los datos del formulario y verifica contra su base de datos que la contraseña esté bien
   - Si no lo está, redirige de nuevo a la página de formulario
8. Si todo fue bien, redirige a la página apropiada para el usuario, y le envía una Cookie de sesión
9. El navegador solicita la página en cuestión enviando la Cookie

Esa Cookie "demuestra" al servidor que el usuario se ha logueado antes.

Se enviará en todas las interacciones con ese servidor.