uc3m | Universidad Carlos III de Madrid

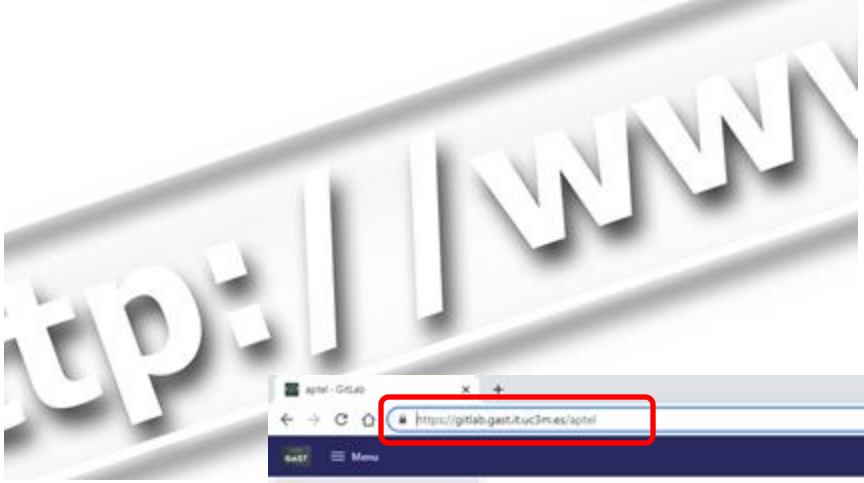
Telematics Applications

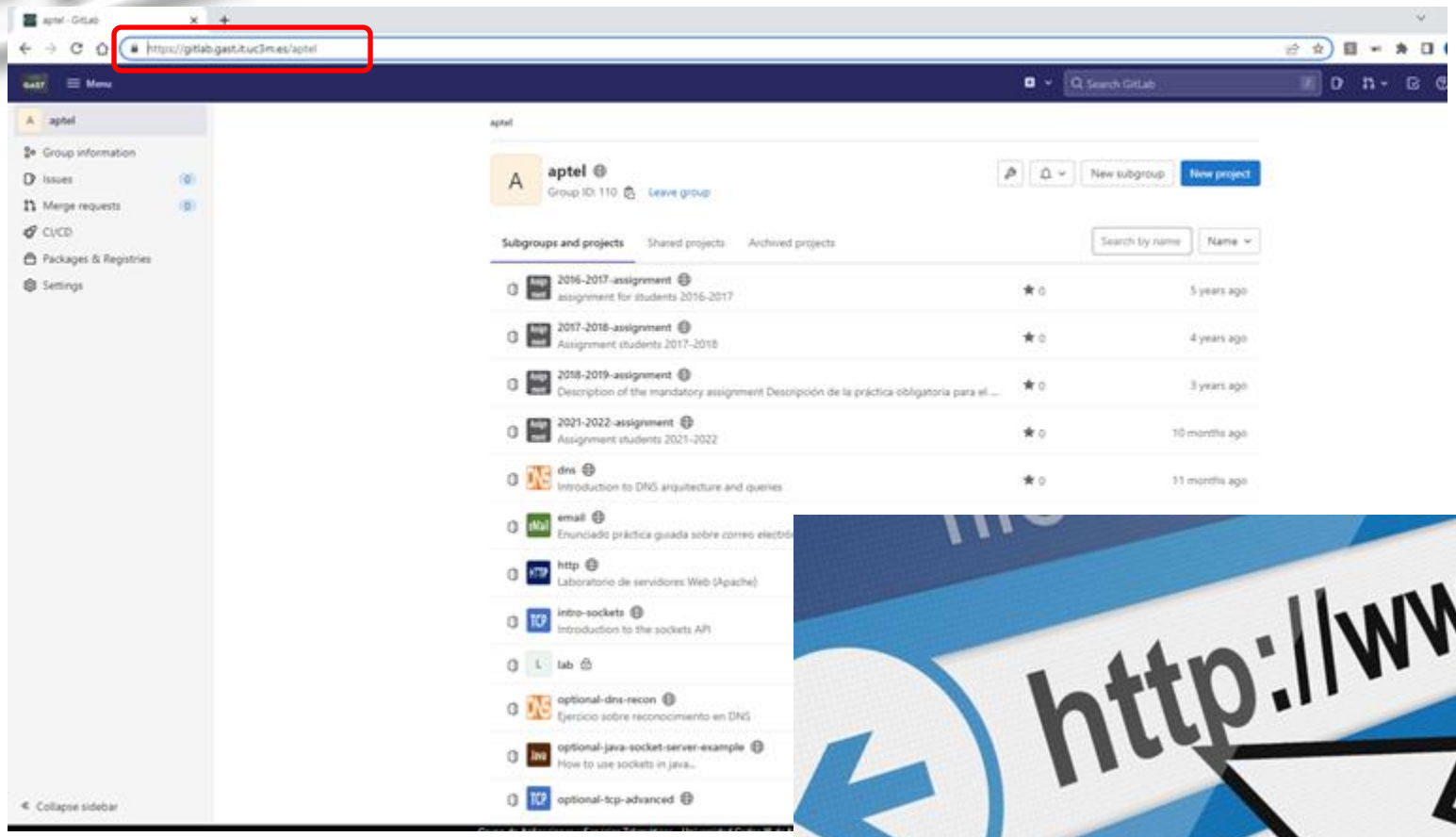
Web: HTTP (*HyperText Transfer Protocol*) and other related protocols

Telematics Applications

Bachelor in Telecommunication Technologies Engineering

Slides generated by Florina Almenarez, Daniel Díaz, Andrés Marín, Patricia Callejo


http://www


aptel - GitLab
https://gitlab.gast.it.uc3m.es/aptel
Menu
Search GitLab
aptel
Group information
Issues
Merge requests
CI/CD
Packages & Registries
Settings
Collapse sidebar
aptel
A
aptel
Group ID: 110
Leave group
New subgroup
New project
Subgroups and projects
Shared projects
Archived projects
Search by name
Name
2016-2017-assignment
assignment for students 2016-2017
5 years ago
2017-2018-assignment
Assignment students 2017-2018
4 years ago
2018-2019-assignment
Description of the mandatory assignment Descripción de la práctica obligatoria para el ...
3 years ago
2021-2022-assignment
Assignment students 2021-2022
10 months ago
dns
Introduction to DNS arquitecture and queries
11 months ago
email
http
Laboratorio de servidores Web (Apache)
intro-sockets
Introduction to the sockets API
lab
optional-dns-recon
Ejercicio sobre reconocimiento en DNS
optional-java-socket-server-example
How to use sockets in java..
optional-tcp-advanced

http://www.

Index

Bibliography

- **Basic**:
  - Forouzan, Behrouz A.. TCP/IP Protocol Suite. McGraw-Hill Higher Education. 2006 (Capítulo 22).

- Complementary:
  - "Internetworking with TCP/IP Volume I. Principles, Protocols and Architecture", 5a Ed. D.E. Comer and D.L. Stevens, Prentice-Hall Int., 2006 (Capitulo 28).
  - "Learning HTTP/2: A Practical Guide for Beginners", Stephen Ludin & Javier Garza. O'reilly, June 2017.

- **RFCs**:
  - RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1, 1999
  - RFC 9113: HTTP/2, 2022
    - RFC 7540: Hypertext Transfer Protocol Version 2 (HTTP/2), 2015
  - RFC 9114: HTTP/3, 2022

Objectives

- Know advanced aspects of the most **popular Internet services**: SMTP, **HTTP**, etc.
- Understand the basic concepts and components related to the WWW and its architecture.
- Learn about HTTP and the differences between the different versions.
- Define the fields of an HTTP **request** and **response**
- Define non-persistent and persistent **connections** in HTTP
- Understand Web **caching** and **proxying**
- Learn about **cookies** and the usage in HTTP
- Learn how to make HTTP requests using command line tools, e.g., `telnet`, `curl`…

Content >> Introduction

1. Introduction
   - A. Web
   - B. Definitions
   - C. General operation

Index

Introduction

- The World Wide **Web** (WWW) is defined by the **W3C** as "*an **information space** in which the items of interest, referred to as **resources**, are identified by global identifiers called Uniform Resource Identifiers* (**URI**)".
  - **resources ⇨ hypertext/hypermedia** documents or services
    - **hypertext is a text that contains links to other texts,** Ted Nelson, ~1965
    - **hypermedia** extension of "hypertext", e.g., audio, video, graphics....
  - **URI** ⇨ provides a simple and extensible way to identify a resource on the web, through a variety of naming schemes and **access methods** ⇨ **URLs**, *Uniform Resource Locator*
    - Access method, e.g., **HTTP**, FTP, email…
    - a locator, a name or both
    - `URI = scheme ":" hier-part [ "?" query ] [ "#" fragment ]`

      `hier-part = "//" authority path-abempty`

      `/path-absolute`

      `/path-rootless`

      `/path-empty`
    - Example: `http://www.rfc-editor.org/rfc/rfc3986#section-3`

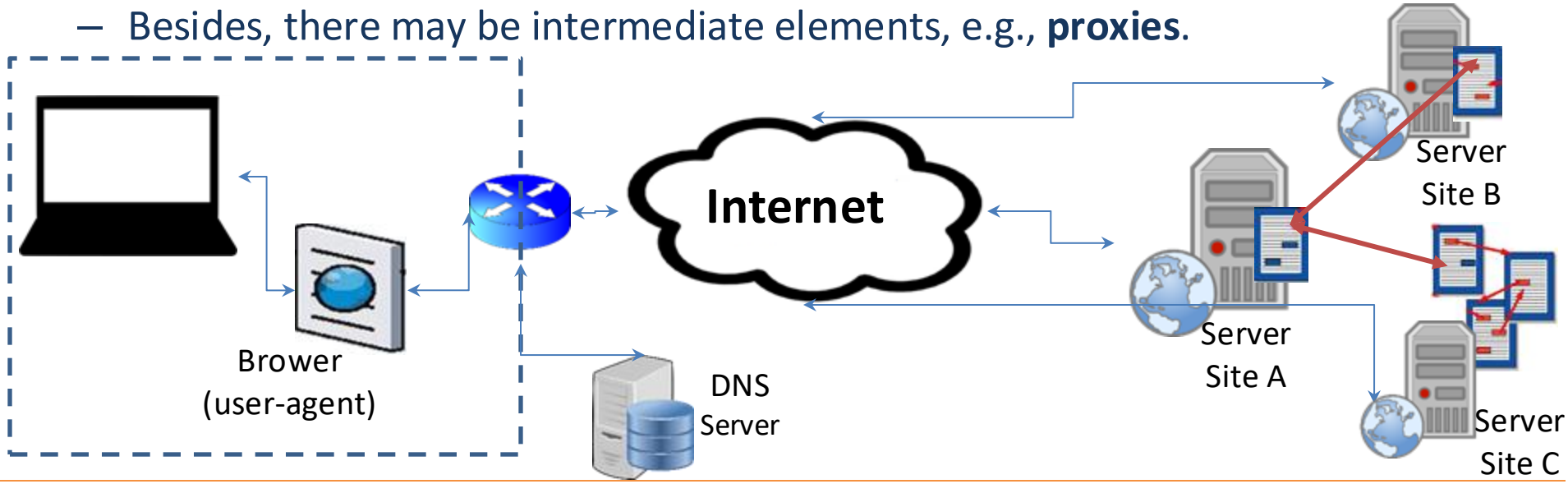
      `scheme` `authority` `path` `fragment`

Introduction >> Web Content

- The content can be grouped into two main categories:
  - **Static**
    - its content does not change, e.g., documents HTML, XML, XSL, XHTML, CSS, etc.
  - **Dynamic**
    - Execute some code (script) on the **server** side, e.g., CGI, Java, Python, PHP, ASP, etc.
    - Execute some code (script) on the **client** side, e.g., Javascript, AJAX, etc. Also known as **active**.
- The websites can have embedded resources (e.g., small base64 images, scripts...).

Content >> WWW Arquitecture

- The WWW is a distributed service **client-server**.
  - **client** uses a browser (or other type of program, e.g spiders) to access a **service** provided by a **server.**
  - the service is distributed over different locations called "**sites**".
  - each site hosts one or more hypertext documents (resources), called "**web pages**".
  - each web page may contain one or more links to other web pages
  - Besides, there may be intermediate elements, e.g., **proxies**.

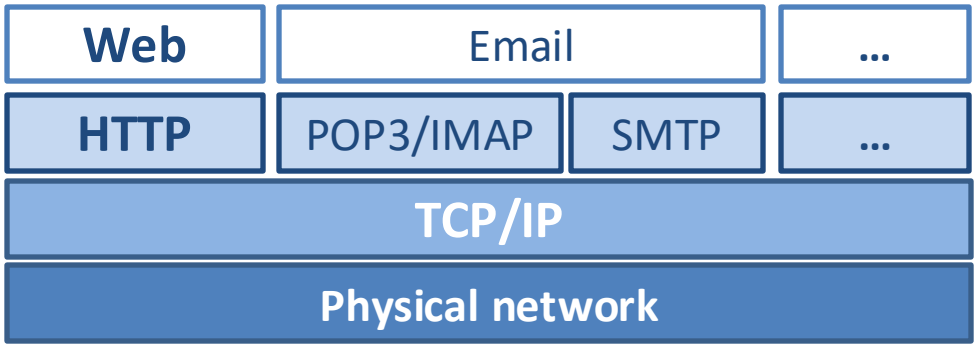


Introduction >> Definitions

- *Hypertext Transfer Protocol (**HTTP**)* ⇨ TCP/IP application layer protocol for distributed, collaborative hypermedia information systems [IETF, RFC2616, 1999]
  - Invented by Tim Berners-Lee, W3C, 1989-1991
- Allows a client's device (computer, smartphone, household appliances, etc.) request a server to send a resource.…
  - **hypertext/hypermedia** transfer protocol
  - nowadays, it can be used for other tasks, e.g. in distributed object management systems
  - allows systems to be built independently of the data being transferred.
- The protocol allows communicating servers, proxies, gateways and clients through the **request/response** scheme.
  - it is based on plaintext messages
  - readable, easy to debug

Introduction >> Definitions

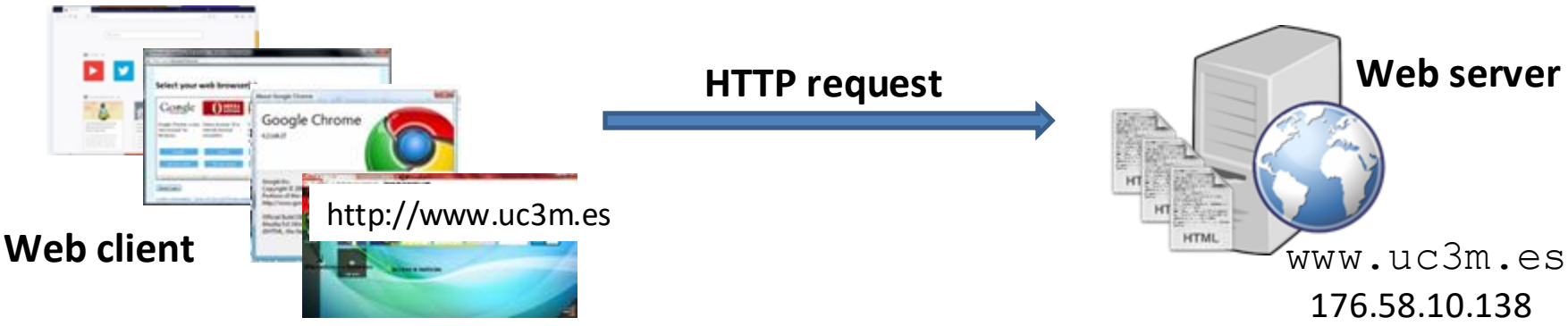
- **Connection-oriented** and **stateless** protocol
  - **connection-oriented**, it uses the **TCP** transport protocol for its operation (end-to-end between a client and a Web server)*.
  - **port 80**, default for non-secure connections (`http://`)
  - **port 443**, default for secure connections (`https://`)



  - **No maintains the state** after each request-response pair, as each connection is independent of the previous one.
    - there is no "session" concept.

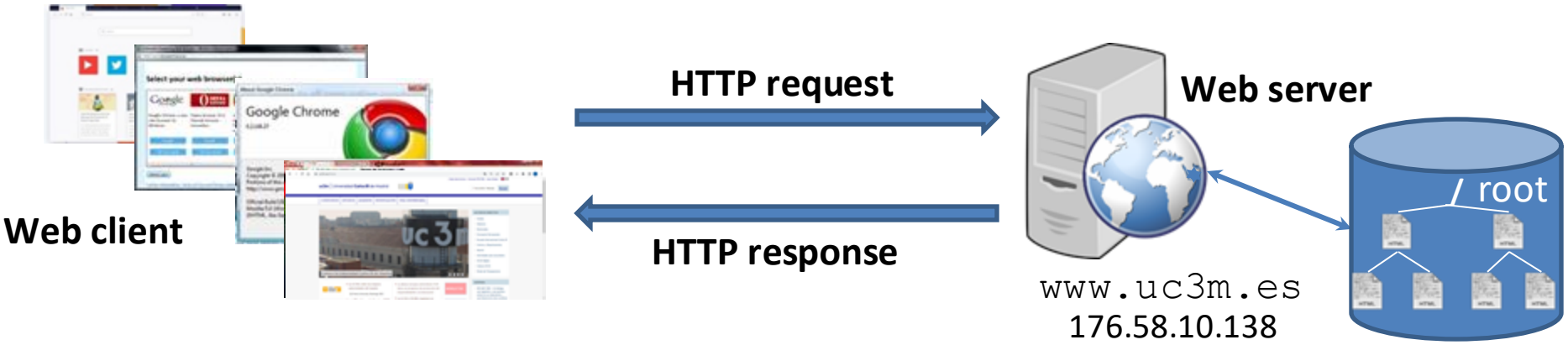
* Other transport protocols are currently supported

Introduction >> General Operation



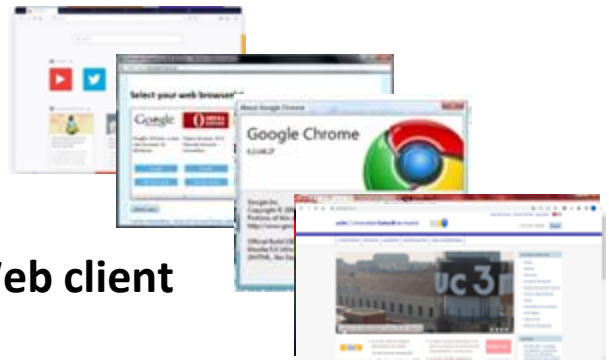
- The user types in the address bar of the browser the resource (URL) he/she wants to access:
  - **http://www.uc3m.es/**
- The browser breaks the URL into 3 parts:
  - ***Scheme*: protocol ("`http`")**
  - ***Authority*: name of the server/domain ("`www.uc3m.es`")**
  - ***Path* to the resource: ("/")**
- The browser communicates with the DNS to translate the name "`www.uc3m.es`" into an IP address, which is used to connect to the Web server machine (176.58.10.138).

Introduction >> General Operation



- The Web server, listening on port 80, accepts the connection.
- Client sends the **request**, e.g.: `GET /`
- The HTTP server receives the request message, creates a response message including the HTML text of the requested page (e.g. root resource) and **send the response**.
- The HTTP server **closes*** the TCP **connection** (depends on the version).
- The client receives the response message, and renders the web page in the browser.

Introduction >> General Operation



Web client

HTTP request

HTTP response

Web server

www.uc3m.es
176.58.10.138

How were or are the request and response messages?

How would the full page be retrieved if it contains 20 images that are part of the page but not "embedded" in it?

Depends on the protocol version

HTTP 0.9 **HTTP 1.0**

HTTP 1.1 **HTTP/2** **HTTP/3**

Content >> First versions

2. First HTTP versions
   - A. HTTP 0.9
   - B. HTTP 1.0
   - C. Exercise

Index

First versions >> HTTP 0.9

- Named after the original version, defined in 1991, W3C
- The **request** consists of a single line of ASCII characters, starting with the **GET** method, followed by the resource to be requested and ending with <CRLF>, e.g.,

```
GET /mywebpage.html
```

- The **response** consists of the requested resource ("*byte stream*" of ASCII characters) and ends with the closing of the connection by the server, e.g.,

```
<html>
  A very simple and obsolete web page
</html>
```

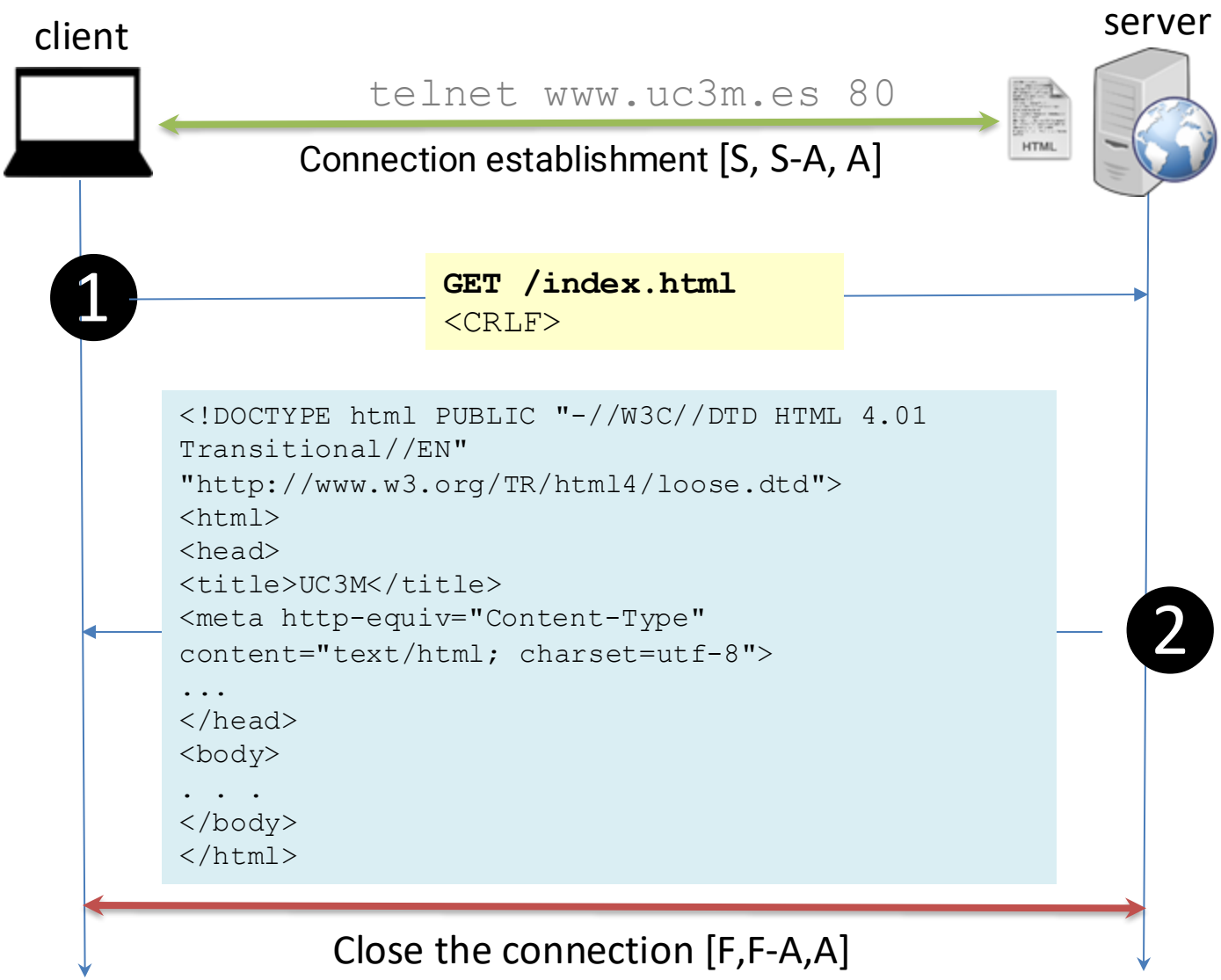
- One connection is used for each resource

No HTTP headers

Only HTML files could be transmitted

No response status information or error codes

First versions >> Example HTTP 0.9



First versions >> HTTP 1.0

- HTTP/1.0, RFC 1945, May 1996
- The protocol version is sent in each request
- Different methods are defined for the request: `GET`, `HEAD`, `POST`
- The concept of headers is added, both for requests and responses.
- Allows the use of different types of resources
  - messages are exchanged in a format similar to those used by RFC 822 and **MIME** (*Multipurpose Internet Mail Extensions*)
- A status code is sent at the beginning of the response.

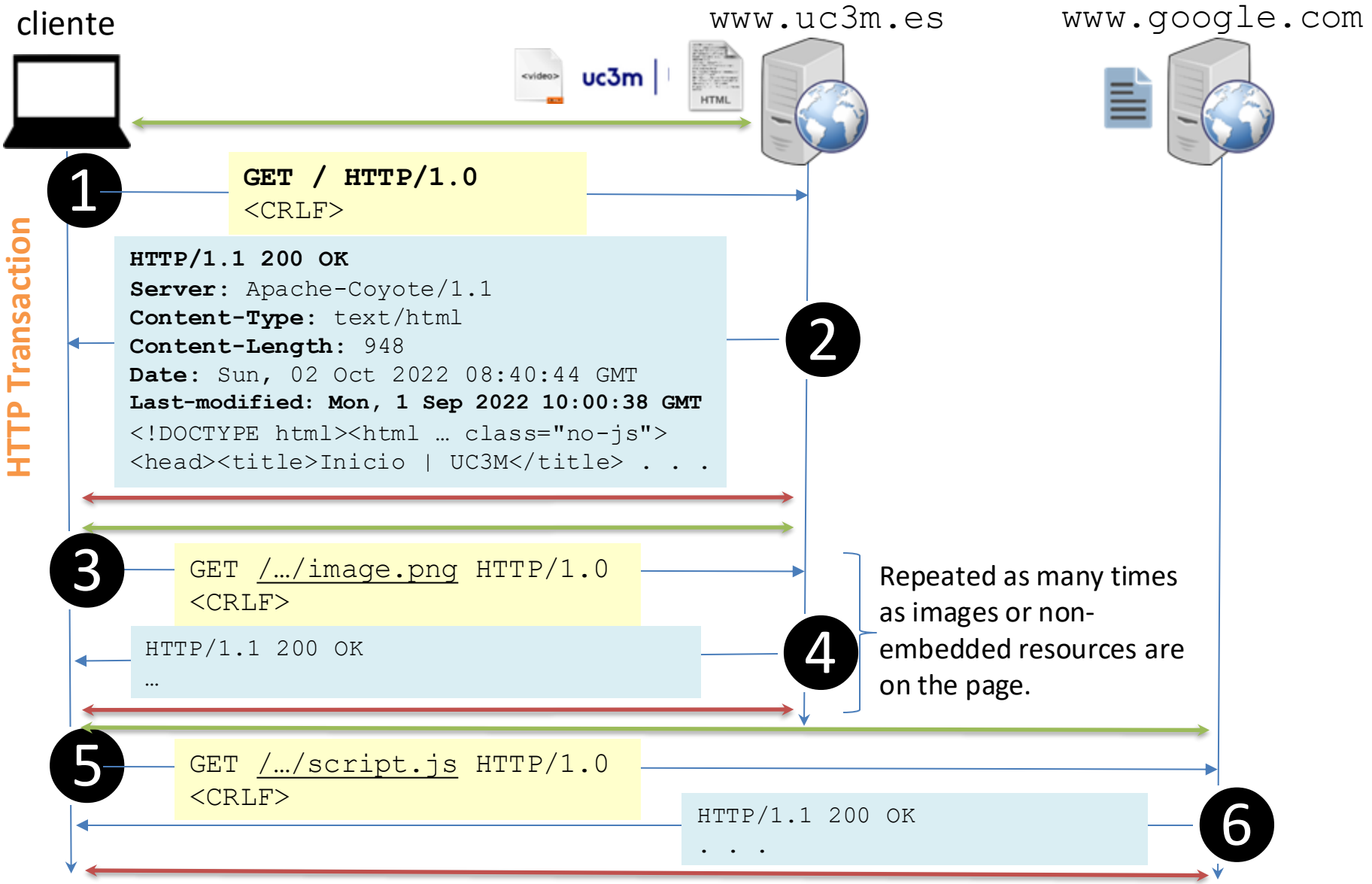
Full request

```
Method Request-URI HTTP-Version CRLF
*( General-Header
| Request-Header
| Entity-Header )
CRLF (an empty line)
[ Entity-Body ]
```

Full Answer

```
HTTP-Version Status-Code Reason-Phrase CRLF
*( General-Header
| Response-Header
| Entity-Header )
 CRLF (an empty line)
[ Entity-Body ]
```

First versions >> Example HTTP/1.0



First versions >> Exercise

- What is the difference between request-response messages in HTTP/1.0 compared to HTTP/0.9?
- What type of documents can you retrieve with HTTP/0.9?
- What is an HTTP transaction?

Content >> HTTP 1.1

3. HTTP 1.1

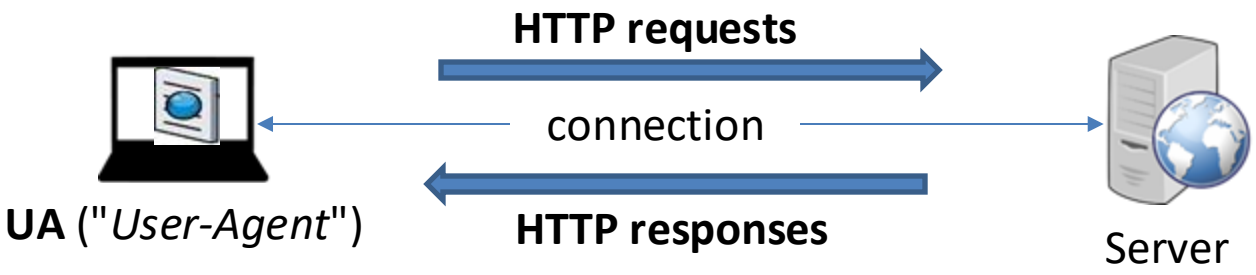
- A. General information
- B. Connection types
- C. Proxy and cache
- D. Request-response messages
- E. Persistent connections
- F. Head of line blocking
- G. Status management
- H. Exercise

Index

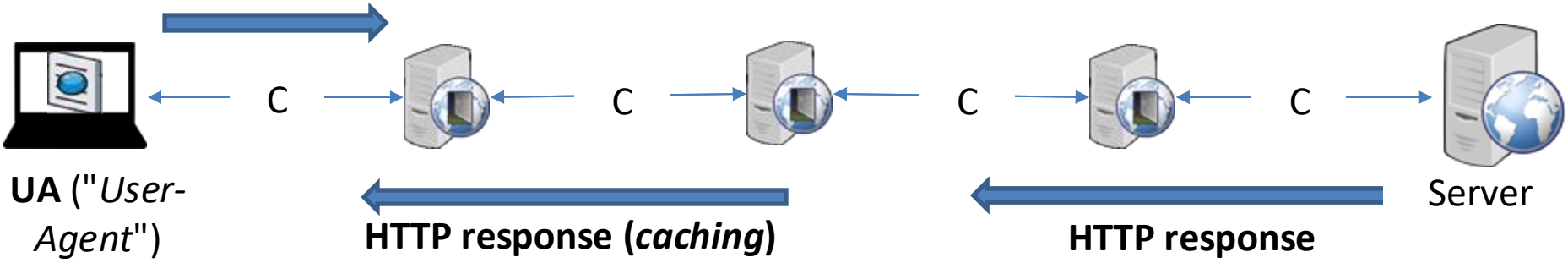
HTTP 1.1

- HTTP/1.1, RFC 2616, June 1999
- Maintains features of HTTP/1.0, e.g., use of MIME, specify protocol version, response status code.
- Extends other features such as **methods** (optional) and request and response **headers** (from 16 in HTTP/1.0 to 46 in HTTP/1.1).
- Takes into account the need for **persistent connections**, virtual hosts and the effects of hierarchical proxies and **cache** usage.
  - A **proxy** is an intermediary program that acts as a server to the client or as a client to the server for the purpose of making requests on behalf of other clients.
  - **Cache** is the local store of response messages and the subsystem that controls the storage, retrieval and deletion of messages.
    - reduce response times and network bandwidth consumption on future requests

HTTP/1.1 >> Client-Server connection types



Separate connections / HTTP request (absolute URI, p.ej. GET http://www.w3.org/index.html HTTP/1.1**)**



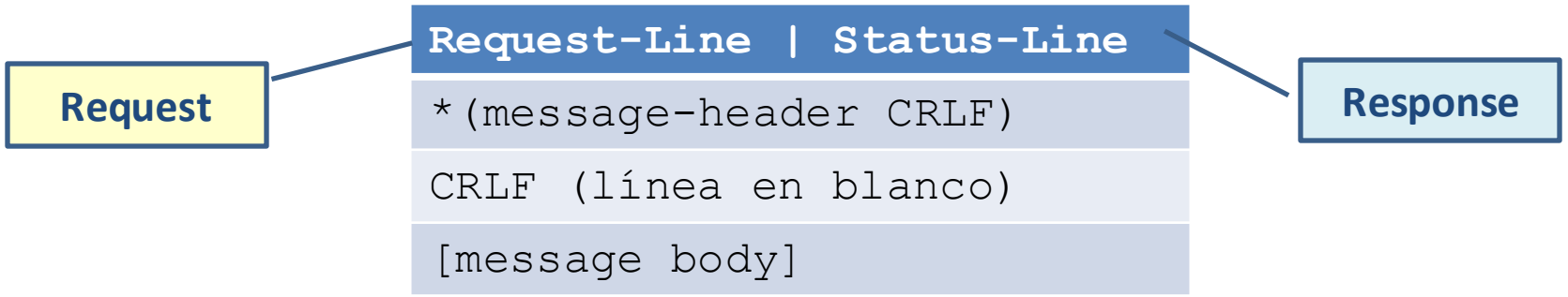
HTTP/1.1 >> *Proxies*

- Uses:
  - **Security**
    - Client > proxy > firewall > Internet
    - proxy: machine authorized to make HTTP connections
  - **Transform** requests and responses
    - translates HTTP versions
    - compress responses to suit low-capacity clients
  - Establish a **cache** of requests and responses
    - prioritize proxy traffic
    - save bandwidth (BW)
    - reduce user-perceived latency

HTTP/1.1 >> Proxy-cache

- Explicit use:
  - users configure their browsers to redirect their access through a proxy-cache, rather than to the end server hosting the resources they are seeking
  - drawbacks:
    - browser configuration required
    - if the proxy is not available, the browser must be reconfigured.
- Transparent:
  - perimeter systems intercept and redirect requests ⇨ TCP SYN lookups (without ACK) to ports 80
  - located at the edge of the ISP ("*Internet Service Provider*")
  - users are unaware of their existence
- It is essential to have the resources updated in cache
  - `expiry` or validation mechanisms
  - time to modify, time in cache, etc.
  - cache parameters in HTML (HTTP/1.0 y 1.1)

HTTP/1.1 >> Request-Response Messages



Request-Line = **Method** Request-URI HTTP-version CRLF

Methods	
OPTIONS	Check the available options
GET	**Request a recourse (`Request-URI`) to the server**
HEAD	Requests information about a resource, without body (only headers = metadata)
POST	**Sends data from the client to the server**
PUT	Creates/updates/replaces a document under the provided URI
DELETE	Deletes a web page (`Request-URI`)
TRACE	Performs a message loop test along the path to the requested resource
CONNECT	Reserved for proxies

HTTP/1.1 >> Response message

Status-Line = HTTP-Version **Status-Code Reason-Phrase** CRLF

Code	Reason-phrase	Description
		INFORMATIONAL - 1XX
100	*Continue*	Partial response to an incomplete request that should be continued
101	**Switching Protocols**	**The server is willing to change the protocol to the one requested by a client (Upgrade header).**
		SUCCESS – 2XX
200	***OK***	**The request has been successful**
201	*Created*	New resource has been created as a result of successful request
202	*Accepted*	Request received
203	*Non-Authoritative Information*	Content was not obtained from the originally requested source
204	*No Content*	The response has no content
205	*Reset Content*	The UA must initiate the page from which the request was made.
206	*Partial Content*	Requested content partially served
		REDIRECTION – 3XX
300	*Multiple Choices*	The request has more than one possible response
301	***Moved Permanently***	**The URI of the resource has changed**
302	***Found***	**The URI of the resource has temporarily changed**
303	*See Other*	Redirects client to a new resource requested at another address
304	*Not Modified*	Response has not been modified (caching purposes)
305	*Use Proxy*	Must be accessed from a proxy (deprecated)
306	*Temporary Redirect*	Redirects the client to another URI with the same method as the origin request

HTTP/1.1 >> Response message (II)

Status-Line = HTTP-Version **Status-Code Reason-Phrase** CRLF

Code	Reason-phrase	Description
	CLIENT ERROR - 4XX (400-417)	
400	***Bad request***	**Invalid request syntax**
401	***Unauthorized***	**Requires authentication to get the requested response**
402	*Payment Required*	Reserved for future use
403	***Forbidden***	**Customer does not have the required permissions**
404	***Not Found***	**The server could not find the requested resource**
405	*Method Not Allowed*	The method cannot be used
407	*Proxy Authentication Required*	Authentication must be done from a proxy (401)
408	*Request Time-out*	Inactive connection
413	*Request Entity Too Large*	The request entity is longer than the defined limits.
414	*Request-URI Too Large*	The URI is too large
415	*Unsupported Media Type*	Server does not support the media format of the requested data
	SERVER ERROR – 5XX	
500	***Internal Server Error***	**Internal server error**
501	***Not Implemented***	**The requested method is not supported by the server**
502	***Bad Gateway***	**The server, as gateway, got an invalid response**
503	*Service Unavailable*	The server is not ready to handle the request
504	*Gateway Time-out*	The server, as a gateway, cannot have a response in time
505	***HTTP Version not supported***	**The HTTP version used in the request is not supported.**

HTTP/1.1 >> Message headers

- Different types of headers are defined ⇨ `header: value`
  - **General headers** ⇨ apply to both requests and responses, unrelated to the data being transmitted in the body.
  - **Request headers**
  - **Response headers**
  - **Entity headers** ⇨ Define meta-information about the content.

Some general headers	
Cache-Control	**Cache-Control: no-cache | Cache-Control: max-age=0** **Cache-Control: public** **Cache-Control: private, community="UC3M"**
Connection	**Connection: close | Connection: upgrade**
Date	**Date: Sun, 04 Nov 2022 08:49:37 GMT**
Pragma	Pragma: no-cache
Transfer-Encoding	**Transfer-Encoding: chunked** **Transfer-Encoding: gzip**
Upgrade	Upgrade: HTTP/2.0, HTTP/3.0 | Upgrade: websockets
Via	Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)

"|" - in the examples - indicates a separation of different and independent headers with the possible values that can be defined

HTTP/1.1 >> General headers >> Cache-Control

- Cache-Control = "Cache-Control" ":" 1#cache-directive
- cache-directive = cache-request-directive | cache-response-directive
- cache-request-directive = "no-cache" ["=" <"> 1#field-name <">]
  | "no-store"
  | "max-age" "=" delta-seconds
  | "max-stale" ["=" delta-seconds]
  | "min-fresh" "=" delta-seconds
  | "only-if-cached"
  | cache-extension
- cache-response-directive = "public"
  | "private" ["=" <"> 1#field-name <">]
  | "no-cache" ["=" <"> 1#field-name <">]
  | "no-store"
  | "no-transform"
  | "must-revalidate"
  | "proxy-revalidate"
  | "max-age" "=" delta-seconds
  | cache-extension

Cache-control options

Option	Description
max-age=[seconds]	Number of seconds that the resource is considered valid
s-maxage=[seconds]	The same as max-age but only for public/shared caches
public	Marks the resource as cacheable by any cache
private	Marks the resource as cacheable except for public/shared caches
no-cache	Revalidate the resource with the server before it serves it from its cache
no-store	The cache must not store a copy of the resource
must-revalidate	Cache must revalidate expired resource. i.e not serve if expired
proxy-revalidate	Same as must-revalidate header excluding private caches
no-transform	Instructs cache not to transform body of message

HTTP/1.1 >> General headers >> Fragmented encoding

- `Transfer-encoding: chunked` means fragmented encoding, so the body of a message is sent as a series of *chunks*.
  - allows a client/server to send data without knowing in advance the total length of data to be sent (typical in dynamic pages)
- Chunks are sent:
  - Line with size in bytes (encoded in hex), followed by ";"
    - after the size there may be optional parameters that can be ignored
  - The chunk ends with <CRLF>
  - To indicate the end of the data, a line with "0" is sent
  - More headers (optional)
  - <CRLF>
- Optionally can include at the end of the general header `Trailer`

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 2012 23:59:59 GMT
Content-Type: text/plain
Transfer-Encoding: chunked
1a; ignore this parameter
abcdefghijklmnopqrstuvwxyz
10
1234567890abcdef
0
some-header: some-value
another-header: another-value
[this is a blank line] (<CRLF>)
```

HTTP/1.1 >> Request headers

Header	Example
Accept | Accept-Charset | Accept-Encoding | Accept-Language	**Accept: text/plain; q=0.5, text/html, text/x-dvi; q=0.8, text/x-c** **Accept: application/json, text/json** **Accept-Charset: "iso-8859-5"**
Authorization | Proxy-Authorization	**Authorization: Basic A29X3DF13KLGF2U=**
From	From: webmaster@w3.org
Host [mandatory]	**Host: www.w3.org** **Host: www.example.com:8080**
If-Match | If-None-Match | If-Range | If-Modified-Since | If-Unmodified-Since	Conditional requests If-Modified-Since: Sat, 29 Oct 2022 19:43:31 GMT
Max-Forwards (TRACE, OPTIONS)	Max-forwards: 1
Range	Range: bytes=0-499, -500
Referer	Referer: http://www.w3.org/hypertext/view.html
TE	TE: deflate | TE: trailers
User-Agent	User-Agent: Mozilla/5.0

HTTP/1.1 >> Request headers >> `Authorization`

- Allows sending credentials to authenticate a user:
  - **Basic** authentication schema
  - **Digest** uses the value of a “nonce”
- The server can define different security zones (“realms”).
  - If a client requests a resource in one of this zones, a challenge is sent back in the response:

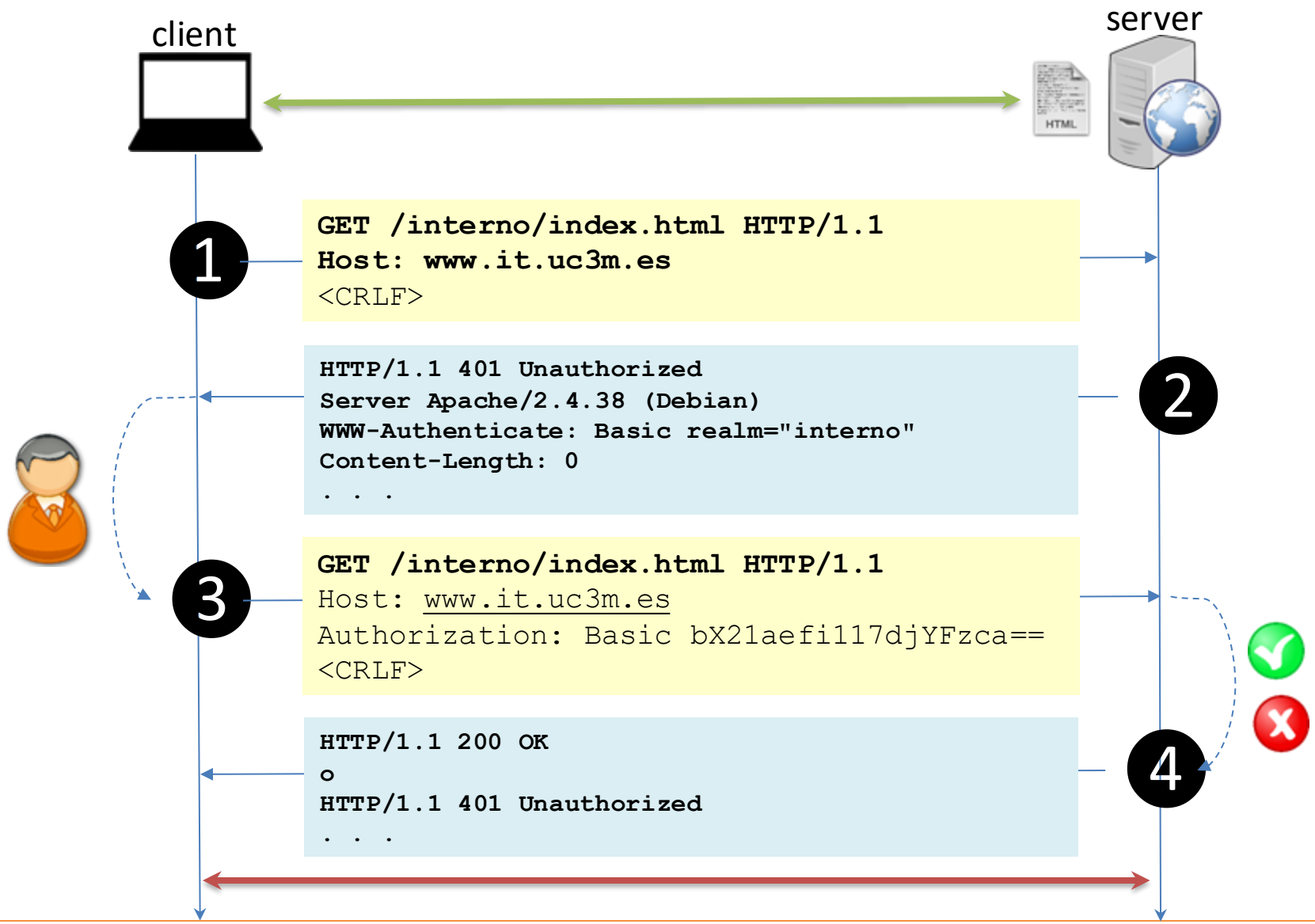
```
401 Unauthorized
WWW-Authenticate: Basic/Digest realm="interno"
[…]
```

- The client sends back the same request but includes a header with the security type, realm, username and password.

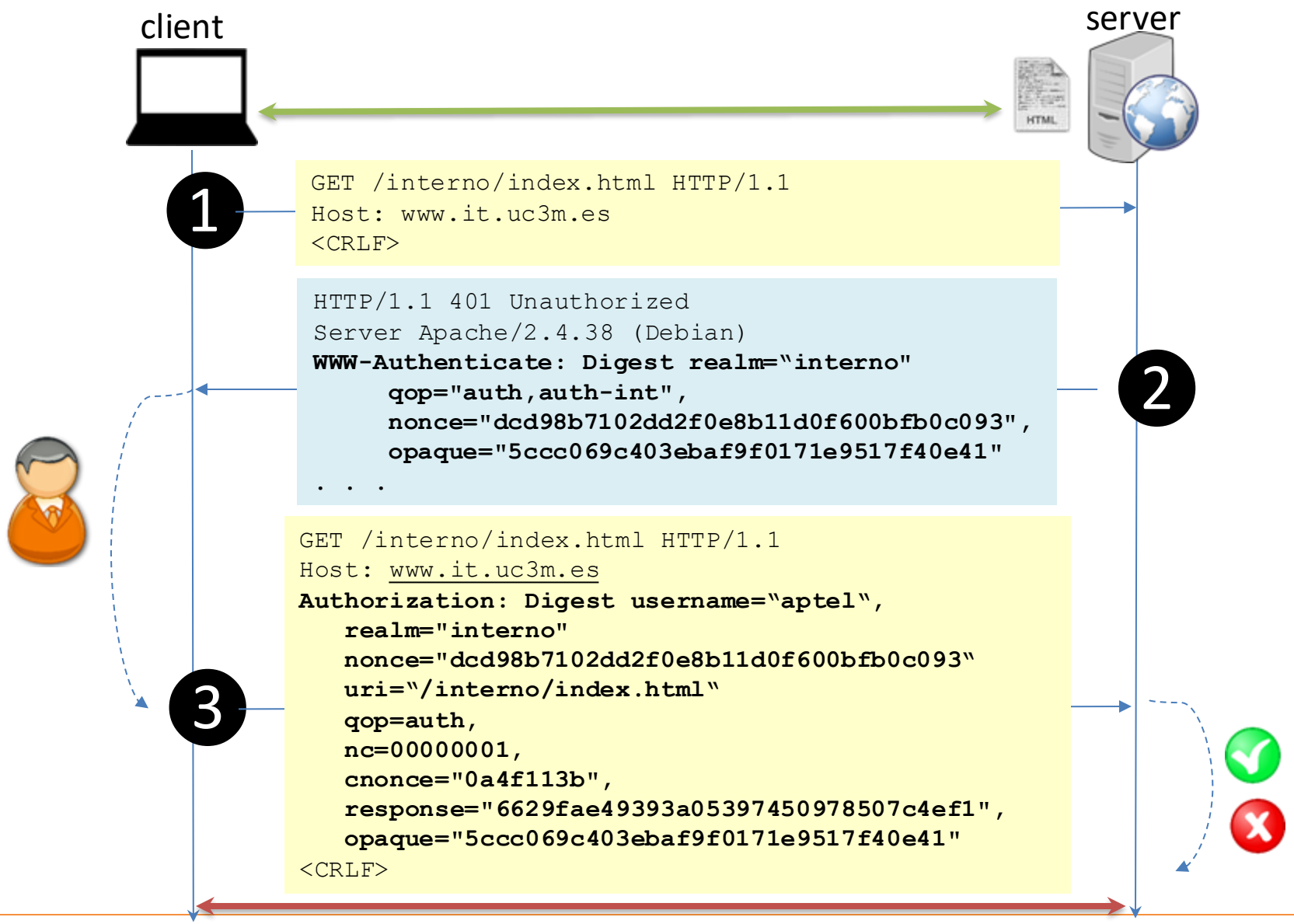
HTTP/1.1 >> Types of authentication

- They are specified in RFC 2617 (updated in RFC 7616 and 7617)
- In the **basic** security scheme user:password is base64-encoded.
  - `Authorization: Basic djpvcGVuIHNlc2FtZQ==`
    - No confidentiality, nor message integrity
    - A trusted channel is assumed https (SSL, TLS)
- The **Digest** authentication scheme uses a nonce and other parameters
  - SHA-512/256 hash function is used, but allows MD5 for backward compatibility: hash(a1:nonce:a2), where
    a1=hash("username:realm:password") and
    a2=hash("reqMethod:reqURI")
  - prevents replay attacks and mitigates nonce count attacks
  - defines a quality of protection parameter ("qop") for entity integrity (request body, not headers)

HTTP/1.1 >> Example basic authentication



HTTP/1.1 >> Example Digest authentication

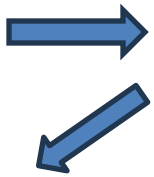


HTTP/1.1 >> Sending content-heavy requests

Bandwidth optimisation ⇨ avoid that the client sends in a request (POST or PUT) a body of many bytes, which the server will not be able to process or satisfy the requirements.

- **Example request:**

```
POST /docs HTTP/1.1
Host: www.example.com
Content-Type: application/pdf
Content-Length:99000
Expect: 100-continue
```



Response:

```
HTTP/1.1 100 Continue
```


PDF file content is sent

```
HTTP/1.1 200 OK
```

(or the corresponsding response)

- If not accepted, response `417 Expectation Failed`
- -If the client does not receive a response after a while, send the body.
- Compatibility may vary between implementations
  - Not all implementations can support or correctly handle this header.

HTTP/1.1 >> Request headers >> `Host`

- **Mandatory** in **HTTP/1.1** requests
  - if it does not appear the server responds with a *400 Bad Request* message.
- The value corresponds to the name of the server in the URL (even the port if different from 80)
  - It is a clue for the server to know to which machine the client tried to connect
  - Allows the same machine (IP) to serve different web sites (virtual hosts)
- For example, if we request the resource file.html through the following URL http://w.host1.com:8080/path/file.html, the request would be:

```
GET /path/file.html HTTP/1.1
Host: w.host1.com:8080<CRLF>
<CRLF>
```

HTTP/1.1 >> Example of a request GET

```
GET /index.html HTTP/1.1
Date:  Thu, 17 Nov 2022 17:00:00 GMT
Connection: close
Host: www.server.com
From: admin@foo.com
Accept: text/html; text/plain
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/105.0.0.0
Safari/537.36
```

⇨ *Line of request*

⇨ *General headers*

⇨ *Request headers*

⇨ *Body of the request (empty) <CRLF>*

HTTP/1.1 >> Example of a request POST

```
POST /form HTTP/1.1
Date:  Thu, 17 Nov 2022 17:00:00 GMT
Connection: close
Host: www.server.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 19

nombre=pepe&edad=25
```

⇨ *General headers*

⇨ *Request headers*

⇨ *Body of the request*

HTTP/1.1 >> Response headers

Header	Example
Accept-Ranges	Accept-Ranges: bytes | Accept-Ranges: none
Age	Age: 0 | Age: 3600
Etag	ETag: W/"948-1501098468000"
Location	Location: http://www.w3.org/pub/people.html
Proxy-Authenticate | **WWW-Authenticate**	WWW-Authenticate: Basic realm="interno"
Retry-After	Retry-After: Thu, 17 Nov 2022 23:00 GMT | Retry-After: 120
Server	Server: Apache/2.4.38 (Debian)
Vary	Vary: User-Agent | Vary: Accept-Encoding

HTTP/1.1 >> Entity headers

Entity header	Example
Allow	`Allow: GET, POST, HEAD`

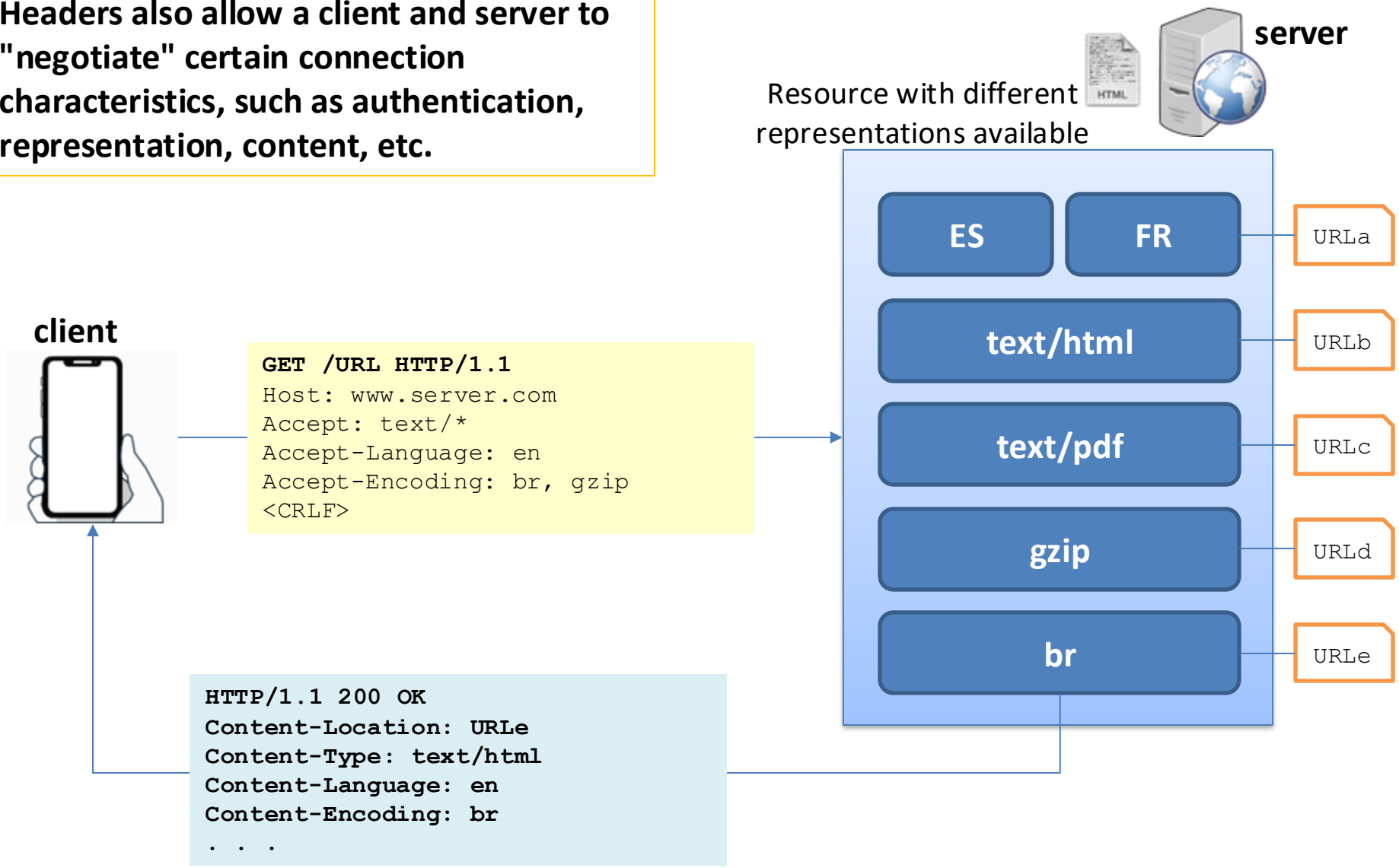
Content-Encoding
Content-Language
Content-Length
Content-Location
Content-MD5
Content-Range
	`Content-Encoding: gzip | Content-Encoding: br`
	`Content-Language: en-US`
	`Content-Length: 68137 | Content-Length: 948`
	`Content-Location: /documents/foo.json`
	`Content-Range: bytes 200-1000/67589`

Content-Type	`Content-Type: text/html; charset="ISO-8859-1"`
	`Content-Type: text/html; charset=utf-8`
	`Content-Type: multipart/form-data; boundary=some`

Expires	`Expires: Sat, 31 Oct 2022 23:59:00 GMT`
Last-Modified	`Last-Modified: Wed, 28 Jul 2021 19:47:48 GMT`

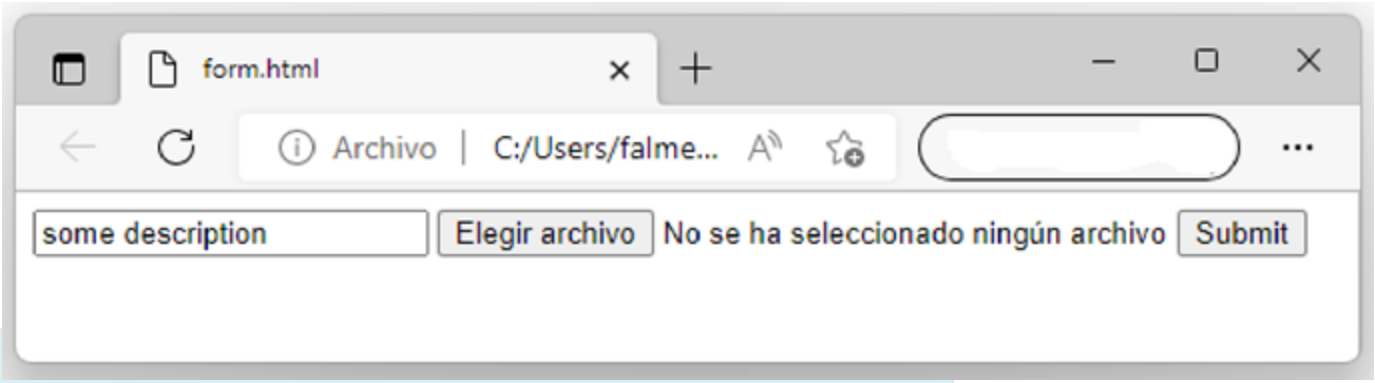
HTTP/1.1 >> Entity headers >> Content negotiation

Headers also allow a client and server to "negotiate" certain connection characteristics, such as authentication, representation, content, etc.



HTTP/1.1 >> Entity headers >> `Content-Type: multipart`

```
<form action="/" method="post" enctype="multipart/form-data">
  <input type="text" name="description" value="some description">
  <input type="file" name="someFile">
  <button type="submit">Submit</button>
</form>
```




```
POST /foo HTTP/1.1
Content-Length: 68137
Content-Type: multipart/form-data; boundary=-----859678012

-------859678012
Content-Disposition: form-data; name="description"

some description
-------859678012
Content-Disposition: form-data; name="someFile"; filename="example.txt"
Content-Type: text/plain

(content of the uploaded file example.txt)
-------859678012--
```

HTTP/1.1 >> Response example

```
HTTP/1.1 200 OK
```
⇨ *Status line*

```
Date:  Thu, 17 Nov 2022 17:01:00 GMT
Connection: close
```
⇨ *General headers*

```
Server: Apache/2.4
Accept-Ranges: bytes
Etag: "5ccca0c574cbebc38f6e95063286d500"
```
⇨ *Response headers*

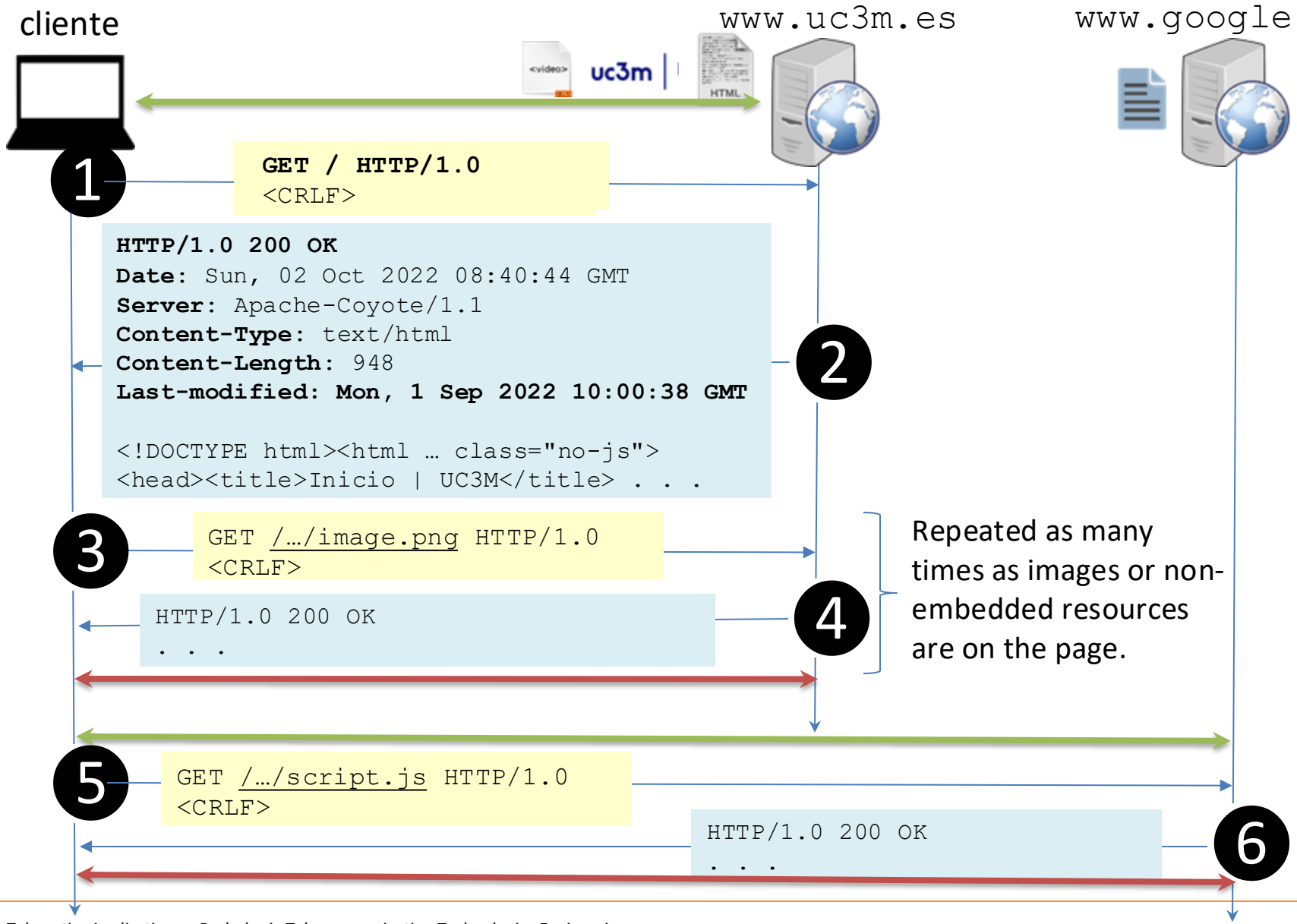
```
Content-Type: text/html
Content-length: 144
Last-Modified: Mon, 14 Nov 2022 11:30:37 GMT
```
⇨ *Entity headers*

```
<html>
<head>
<title>Web: HTTP</title>
</head>
<body>
<h1>HTTP Lab</h1>
<p>Aplicaciones Telemáticas: Servidores Web (HTTP)</p>
</body>
</html>
```
⇨ *Body of the response*

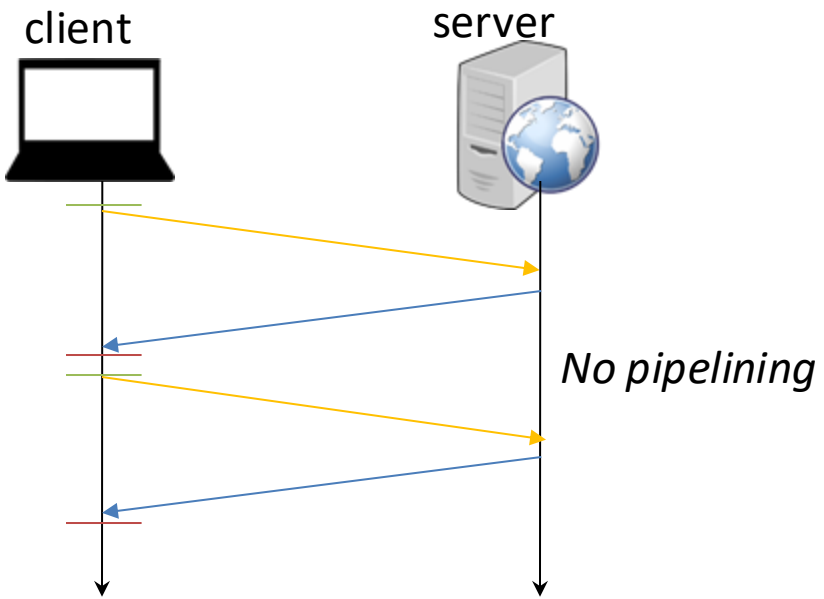
HTTP/1.1 >> Persistent connections

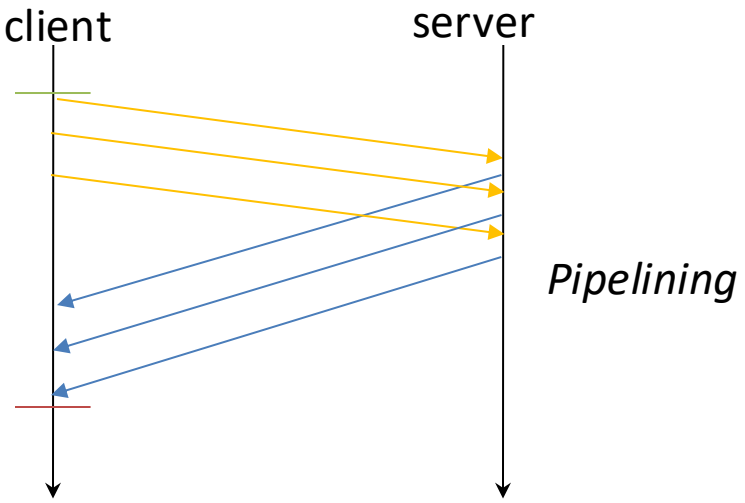
- Allows multiple requests to reuse the same connection
  - more than one transaction per request
  - several requests may be sent together without waiting for the responses (***HTTP pipelining***)
    - **responses** should be sent in the same order
    - the client must be prepared to resend the requests if the server close the connection before sending all the corresponding responses
  - The client can indicate that no more requests will be sent using the "`Connection: close`" header.
  - So the server may use the "`Connection: close`" header to close the connection (with or without sending all the responses)

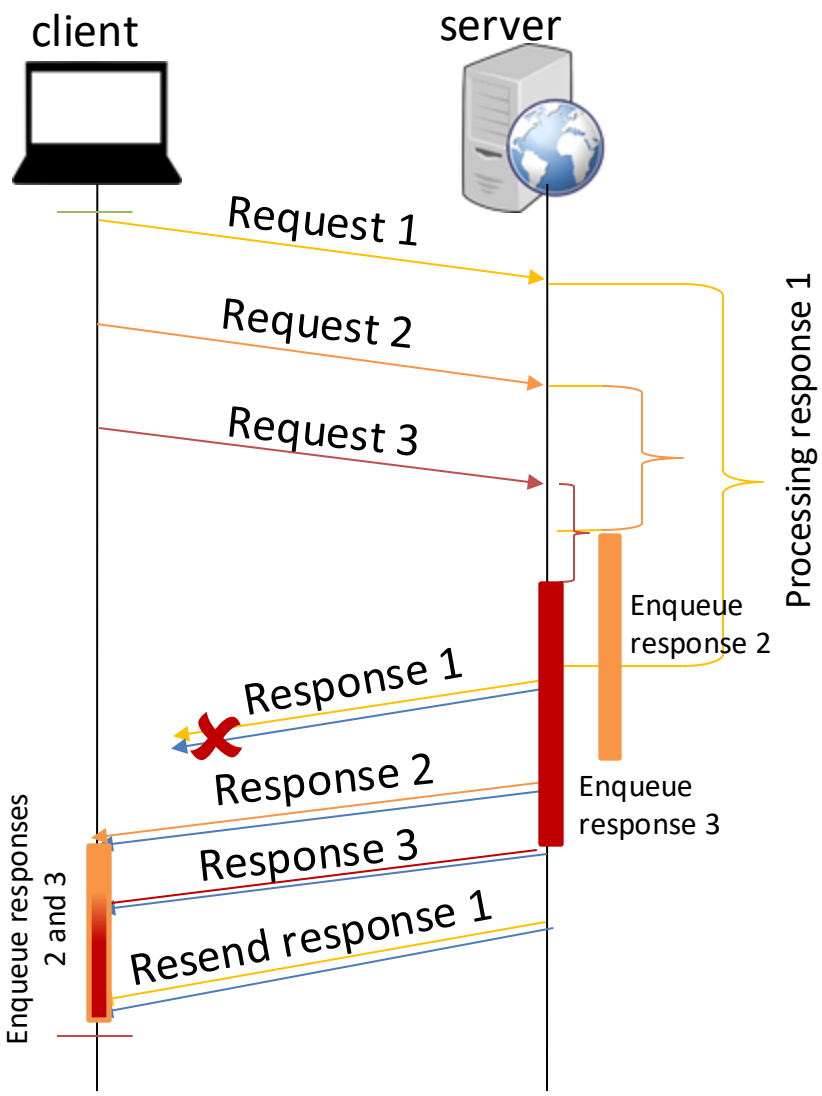
HTTP/1.1 >> Example persistent connections



HTTP/1.1 >> HOLB

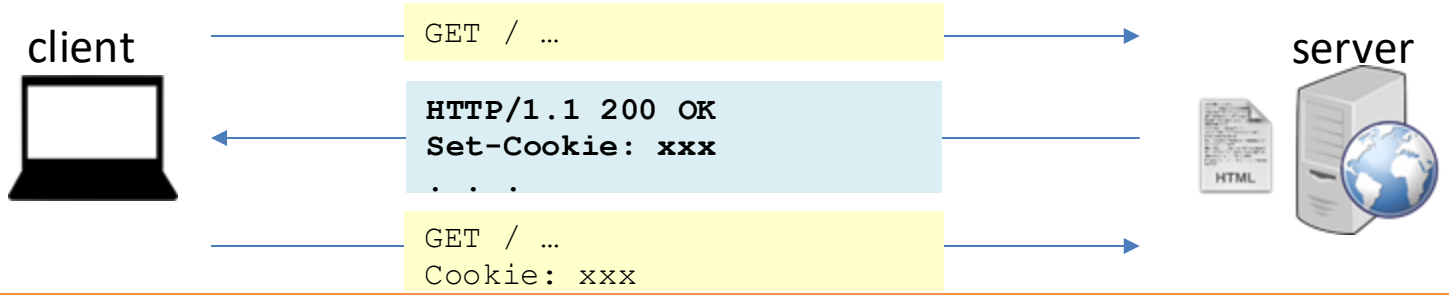

client
server
No pipelining


client
server
Pipelining

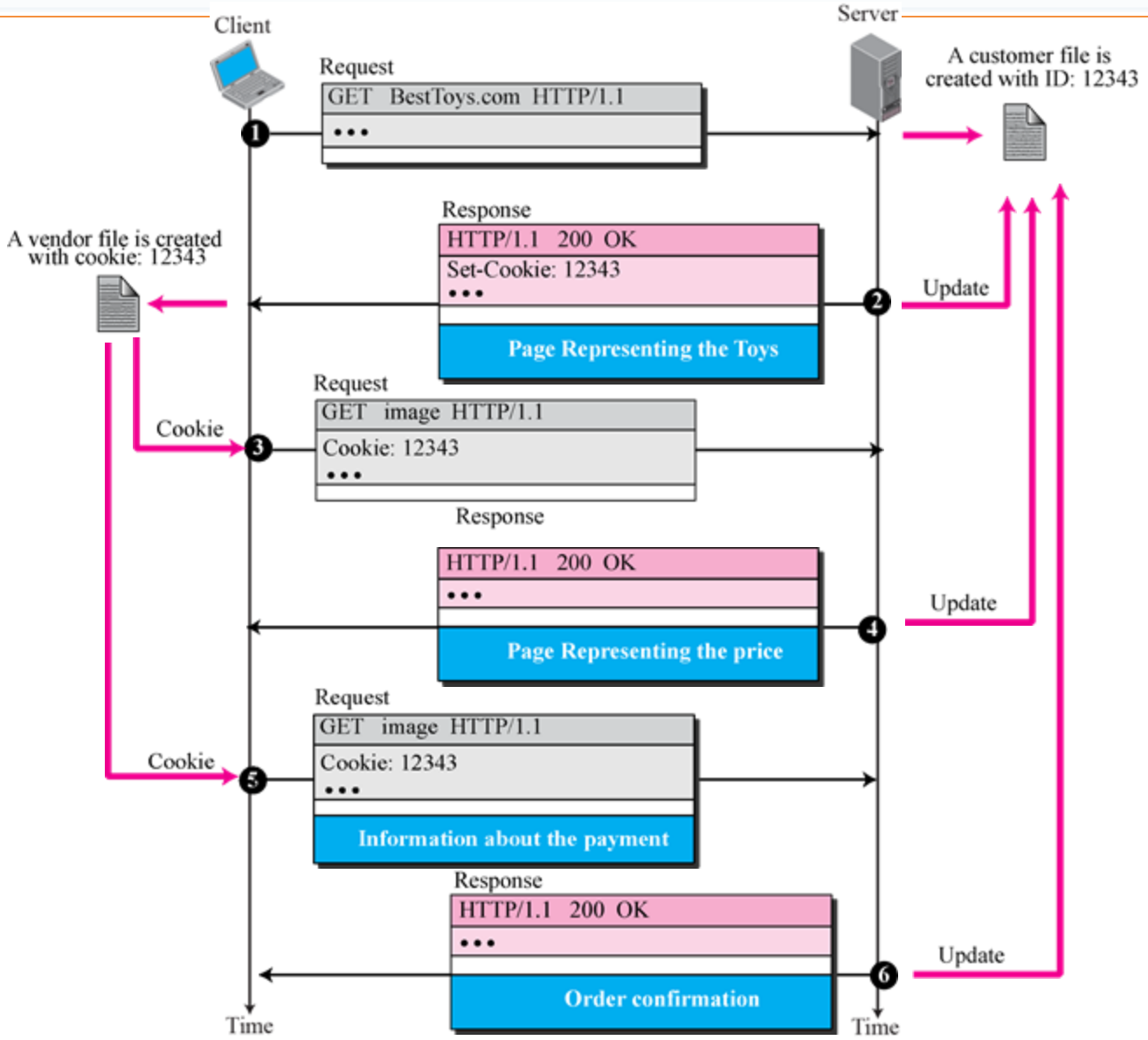

client
server
Request 1
Request 2
Request 3
Processing response 1
Enqueue response 2
Response 1
Response 2
Enqueue response 3
Response 3
Resend response 1
Enqueue responses 2 and 3

HTTP/1.1 >> Status management >> Cookies

- Since HTTP is stateless, it can use **cookies**:
  - The server sends the cookie(s) with the HTTP response and the client stores them (header **`Set-Cookie`**).
  - Future requests from the client will be accompanied by the cookie(s) so that the server recognizes the client (and the session).
- Cookies were introduced by Netscape in 1994 (RFC 2965, updated with **RFC 6265**)
- A cookie allows to store context information about the session, user state, etc.
  - Browser does not interpret them, merely returns them to the server in future requests.
  - The client can impose restrictions or limits, e.g. number of cookies per server, totals, minimum length...



HTTP/1.1 >> Status management >> Cookies



HTTP/1.1. >> Exercises >> requests and responses

- Write the HTTP/1.1 request to retrieve an image with the URI `/www/image1.png` on the server `www.it.uc3m.es`

- Which header do we add to indicate that the client can accept images in GIF, JPEG and PNG format?

- Write the answer defining the headers `Date`, `Server`, `Content-Type`, `Content-Length` of the image. Note: you do not need to include the body of the response.

Content >> HTTP/2

4. HTTP/2

A. Introduction & Background
B. General overview
C. Frame and frame types
D. Requests and responses
E. Exercise
F. Compression and decompression
G. Streams
H. “PUSH”
I. Conclusions

Index

HTTP/2 >> Differences HTTP/1.1 vs HTTP/2

From HTTP/2 IN ACTION by BARRY POLLARD, Copyright 2018.

© https://freecontent.manning.com/animation-http-1-1-vs-http-2-vs-http-2-with-push/

HTTP/2 >> Differences HTTP/1.1 vs HTTP/2

From HTTP/2 IN ACTION by BARRY POLLARD, Copyright 2018.

© https://freecontent.manning.com/animation-http-1-1-vs-http-2-vs-http-2-with-push/

HTTP/2 is the future of the Web, and it is here!

Your browser supports HTTP/2!

This is a demo of HTTP/2's impact on your download of many small tiles making up the Akamai Spinning Globe.

HTTP/2 >> Introduction

- HTTP/2 was initially defined in **RFC 7540** (May 2015) and updated in **RFC 9113**, June 2022
  - derived from the open communication protocol that was developed for transporting web content, called **SPDY,** approx. 2012
- **SPDY**, protocol announced by Google in 2009 and developed in 2010
  - Purpose: reduce the loading time of web pages and improve web security
  - Main features:
    - Uses TCP and TLS
    - Allows **concurrent requests** in a single TCP session
      - client can assign a **priority** to each request, to prevent network congestion with non-critical resources
      - client keeps the **connection open** until it navigates away from the web pages or the server closes the connection

HTTP/2 >> Background >> SPDY

- Main features of SPDY:
  - Reduces bandwidth (BW) by **compressing headers** (zlib) and eliminates unnecessary headers.
  - Defines message formats that are easy to parse ("**framing**")
    - "*control frames*" and "*data frames*"
  - Introduces **streams** that are independent bidirectional data streams divided into **frames**
    - 3 types of control frames to manage the stream lifecycle
  - Allows the server to initiate communications with the client and send data wherever possible
    - via a *stream* containing an "`associated-stream-id`", which indicates the request *stream* to which the sent data is related.

HTTP/2 >> General overview

- Optimize HTTP semantics, while maintaining backward compatibility (methods, URIs, codes, TCP/TLS...).
- Enable a more **efficient** use of network **resources**
  - clients establish a **single** network **connection** with the server
  - headers, data, messages are packaged in **binary data** structures
- Reduce **latency** by introducing header **compression** and allowing concurrent exchanges over the same connection
  - **multiplexing** and **prioritization** of requests ⇨ requests and responses associated with their own "**stream**".
  - avoid connection congestion and the problem of **head of line blocking** (HOLB) at HTTP level, i.e:
    - clients must limit the number of simultaneous connections
    - each client can make several requests on the same connection (HTTP/1.1) and the responses must be received in the same order

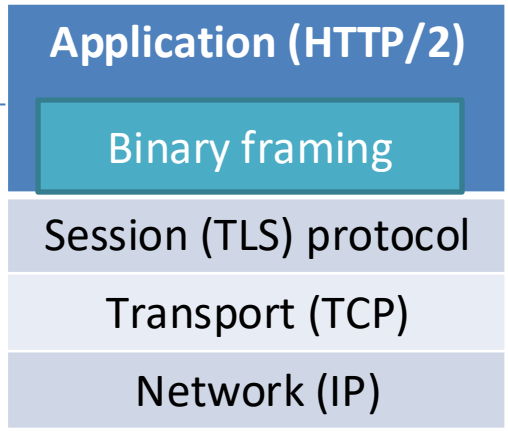
HTTP/2 >> General overview

- Optimize HTTP semantics, while maintaining backward compatibility (methods, URIs, codes, TCP/TLS...).
- Enable a more **efficient** use of network **resources**
  - clients establish a **single** network **connection** with the server
  - headers, data, messages are packaged in **binary data** structures
- Reduce **latency** by introducing header **compression** and allowing concurrent exch
  - **multiplexing** responses a
  - avoid conne **blocking** (H
    - clients must limit the number of simultaneous connections
    - each client can make several requests on the same connection (HTTP/1.1) and the responses must be received in the same order

> *"Clients that use persistent connections SHOULD limit the number of simultaneous connections that they maintain to a given server. A single-user client SHOULD NOT maintain more than 2 connections with any server or proxy… These guidelines are intended to improve HTTP response times and avoid congestion". [RFC2616]*

HTTP/2 >> General overview (II)

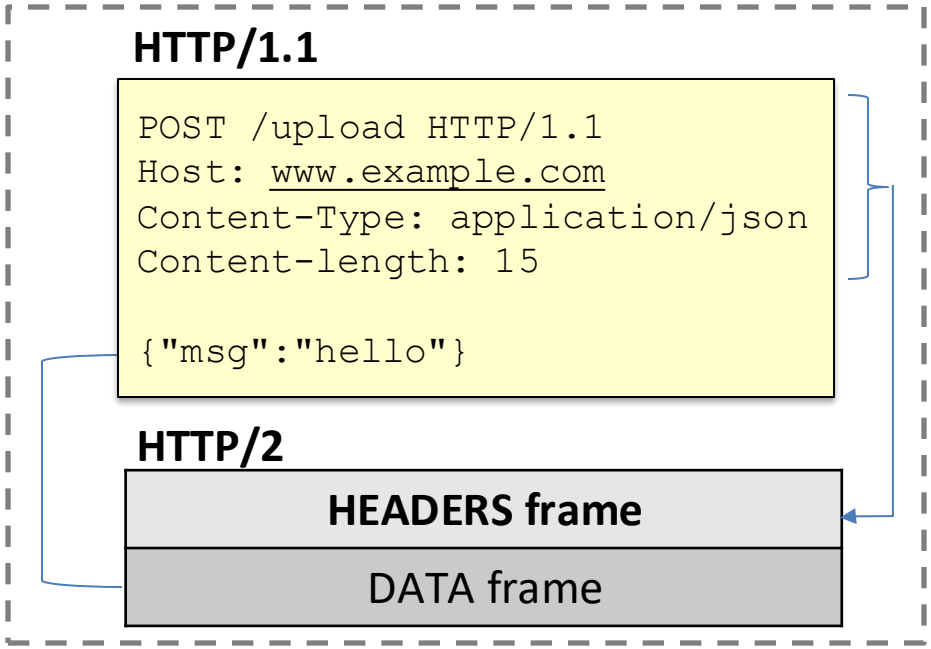
- It can be generalized into two parts: the framing layer and the data or http layer.
- The binary framing layer
  - prioritization and flow control
  - introduce *push* messages from server to client
- **Frames** are the basic unit of the protocol.
  - have 9-octets headers including:



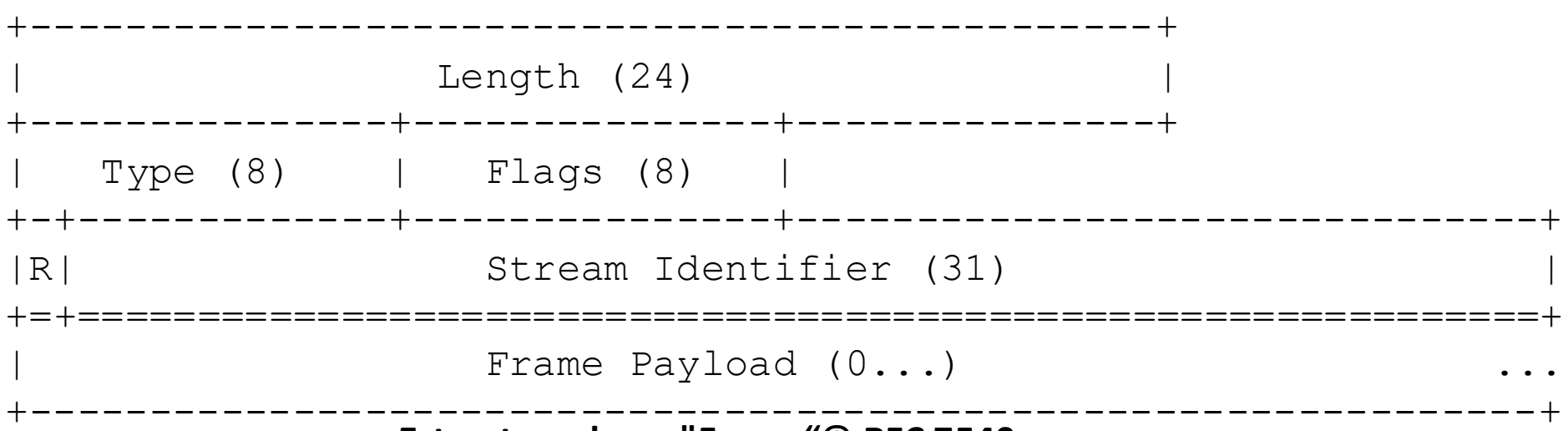

```
HTTP Frame {
  Length (24),
  Type (8),

  Flags (8),
  Reserved (1),
  Stream Identifier (31),

  Frame Payload (0...),
}
```



HTTP/2 >> Frame header fields



Estructura de un "Frame“© RFC 7540

Field	Length	Description
`Length`	3 bytes	Payload length (**2^{14}** and 2^{24}-1 bytes)
`Type`	1 byte	*Frame* type
`Flags`	1 byte	Flags specific to the frame type
`R`	1 bit	Reserved
`Stream ID`	31 bits	Unique identifier for each stream
`Payload`	Variable	Data according to length specified in field `Length`

HTTP/2 >> Frame types

Frame	Functionality
DATA	Carry octets in a stream, for instance HTTP requests or responses
HEADERS	To open a stream, or carry header block fragment
PRIORITY	Sender advised priority of a stream (carries dependency and weight)
RST_STREAM	Requires immediate termination of a stream
SETTINGS	Configuration parameters like preferences and constraints on peer behavior (HEADER_TABLE_SIZE, ENABLE_PUSH, MAX_CONCURRENT_STREAMS, INITIAL_WINDOW_SIZE, MAX_FRAME_SIZE, MAX_HEADER_LIST_SIZE)
PUSH_PROMISE	Notifies the peer endpoint (client or server) in advance of streams the sender intends to initiate
PING	Measuring round trip time
GOAWAY	To initiate shutdown of a connection or to signal serious error conditions
WINDOW_UPDATE	To implement flow control, signals of window increments up to 2^31-1, that may apply to a given stream or to the whole connection (stream_id=0)
CONTINUATION	To continue a sequence of header block fragments, the END_HEADERS flag is set if this is the last fragment

HTTP/2 >> Connection management

- Connections are persistent
  - for performance, it is expected that clients will not close connections until it is determined that no further communication with a server is necessary (for example, when a user goes away from a particular web page) or until the server closes the connection.
  - Clients should not open more than one connection to a given "`host:port`" with the same configuration or under "normal" conditions.
- CONNECT method ⇨ converts an HTTP connection into a tunnel to a remote computer (used with proxies to establish a TLS session).
- HTTP/2 implementations must use TLS version 1.2 or higher when using a secure connection.
- Although the use of TLS is not mandatory in the specification, in practice most implementations (e.g., browsers) support HTTP/2 when used over TLS.

HTTP/2 >> Request-Response

- A client making an HTTP request for a URI, without prior knowledge about HTTP/2 support, uses the `Upgrade` header
  - "h2c" for “http” connections
  - “h2” for “https” connections

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding
of HTTP/2 SETTINGS payload>
```

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html

...
```

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[ HTTP/2 connection ...
```

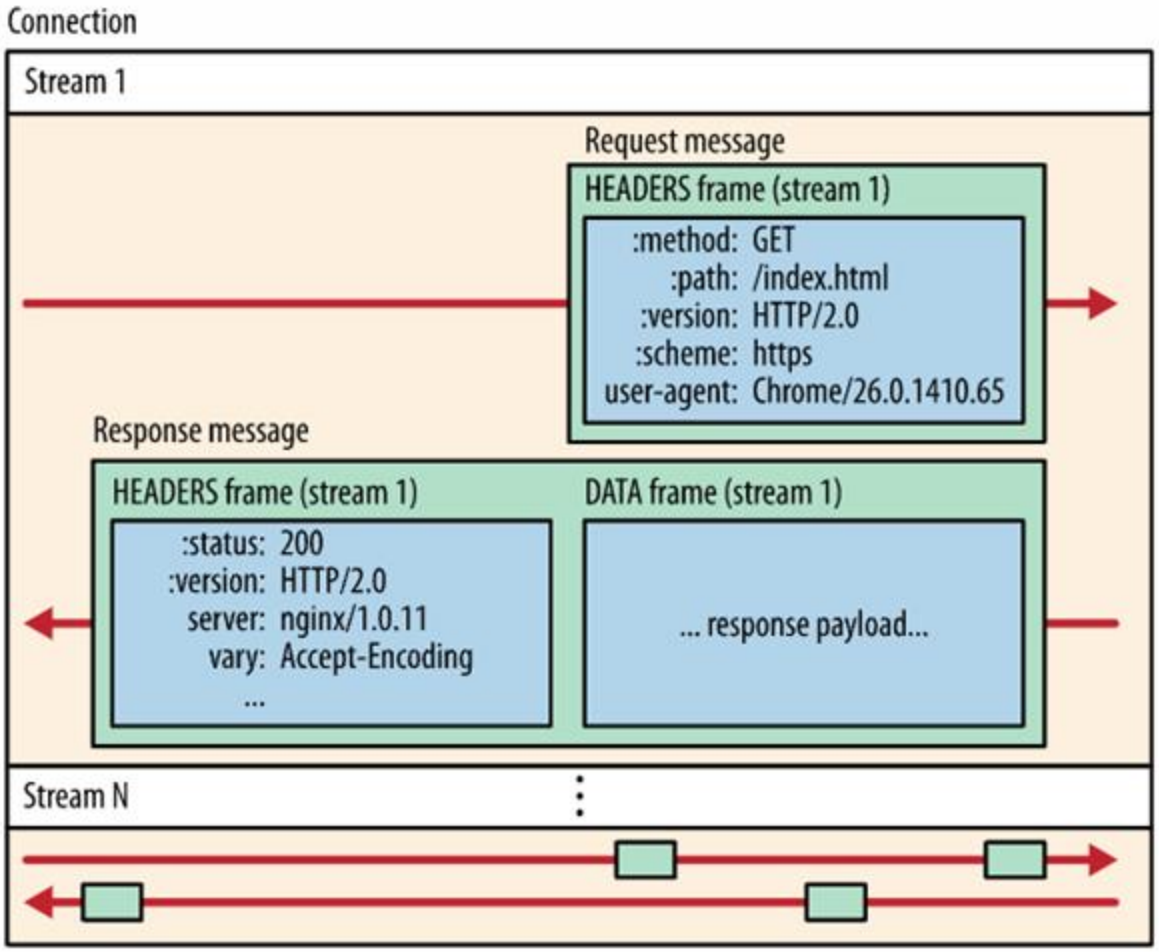
- If it has prior knowledge, then the client must send the preamble of the connection and immediately send frames (`SETTINGS`)

```
x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

HEX

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

HTTP/2 >>Request - response exchange



Source: https://hpbn.co/http2/#streams-messages-and-frames

HTTP/2 >> Example of a GET request - response

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

```
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 ...
```

HTTP/2 >> Example of a POST request - response

```
POST /resource HTTP/1.1          HEADERS
Host: example.org          ==>     - END_STREAM
Content-Type: image/jpeg           - END_HEADERS
Content-Length: 123                  :method = POST
                                     :path = /resource
{binary data}                        :scheme = https
                                 CONTINUATION
                                   + END_HEADERS
                                     content-type = image/jpeg

       host = example.org

                                     content-length = 123
                                 DATA
                                   + END_STREAM
                                 {binary data}
```

```
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
```

HTTP/2 >> Practical example of an HTTP request-response

```
$ curl -v --http2 http://gitlab.gast.it.uc3m.es/aptel
  * Trying 163.117.141.50:80...
  * Connected to gitlab.gast.it.uc3m.es (163.117.141.50) port 80 (#0)
> GET /aptel HTTP/1.1
> Host: gitlab.gast.it.uc3m.es
> User-Agent: curl/7.88.1
> Accept: */*
> Connection: Upgrade, HTTP2-Settings
> Upgrade: h2c
> HTTP2-Settings: AAMAAABkAAQCAAAAAAIAAAAA
>
< HTTP/1.1 301 Moved Permanently
< Server: nginx
< Date: Wed, 15 Nov 2023 13:23:07 GMT
< Content-Type: text/html
< Content-Length: 178
< Connection: keep-alive
< Location: https://gitlab.gast.it.uc3m.es/aptel
<
<html><head><title>301 Moved Permanently</title></head>
<body bgcolor="white"><center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx</center>
</body></html>
```

HTTP/2 >> Practical example of an HTTPS request-response

```
$ curl -v --http2 https://gitlab.gast.it.uc3m.es/aptel
  * Trying 163.117.141.50:443...
  * Connected to gitlab.gast.it.uc3m.es (163.117.141.50) port 443 (#0)
  * ALPN: offers h2,http/1.1
  * TLSv1.3 (OUT), TLS handshake, Client hello (1):
  * . . .
  * SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
  * ALPN: server accepted h2
  * . . .
  * Using Stream ID: 1 (easy handle 0x5630d704fc70)
> GET /aptel HTTP/2
> Host: gitlab.gast.it.uc3m.es
> user-agent: curl/7.88.1
> accept: */*
>
< HTTP/2 200
< server: nginx/1.14.2
< date: Wed, 15 Nov 2023 13:17:20 GMT
< content-type: text/html; charset=utf-8
< content-length: 32611
< cache-control: max-age=0, private, must-revalidate
< etag: W/"6e19af91479c8bbba703be9fa61cd46c"
< referrer-policy: strict-origin-when-cross-origin
< vary: Accept
<
<!DOCTYPE html>
<html class="with-header with-system-footer" lang="en">
<head prefix="og: http://ogp.me/ns#"> . . .
```

HTTP/2 >> Exercise

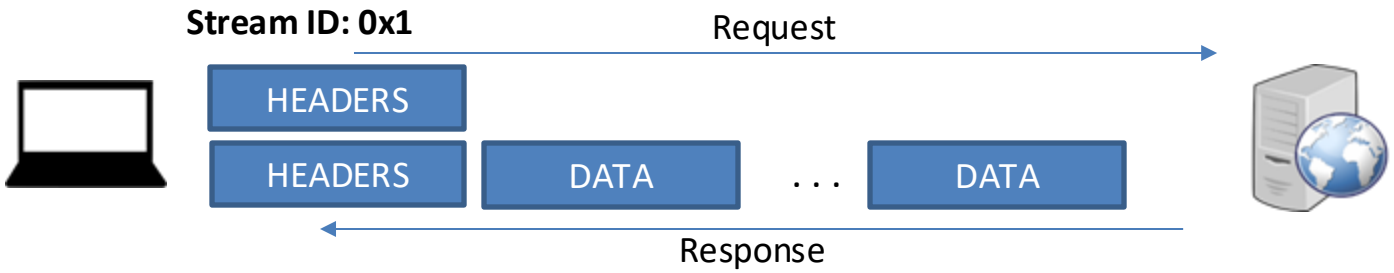
- Access to `aulavirtual.lab.it.uc3m.es`
- Open a terminal
- Access to https://www.google.com using the command line `curl`. Note: you can change the "`User-agent`", e.g, `--user-agent "Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0"`
- Indicates the request and response obtained (without the body)

HTTP/2 >> Compression

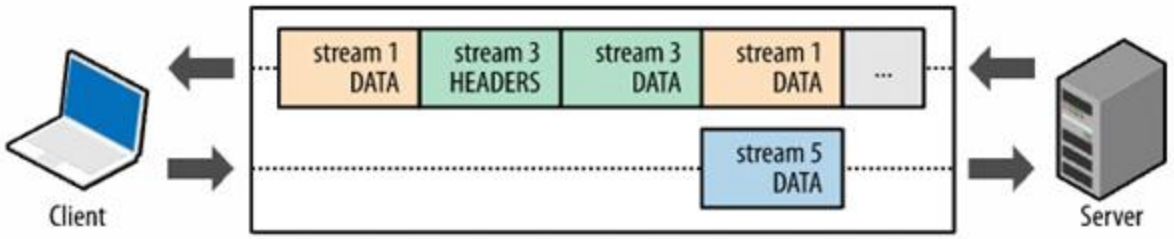
- HTTP message headers can contain large amounts of data, the "*frames*" contained in these messages are compressed (**HPACK**).
  - the main advantage lies in the size of the requests.
- Headers are used in both requests and responses in *push* messages.
- Header block fragments can be sent as payload of the frames HEADERS, PUSH_PROMISE or CONTINUATION
  - who receives these frames organizes the header blocks and decompresses them.

HTTP/2 >> Streams and Multiplexing

- A stream is a bidirectional, independent sequence of frames.



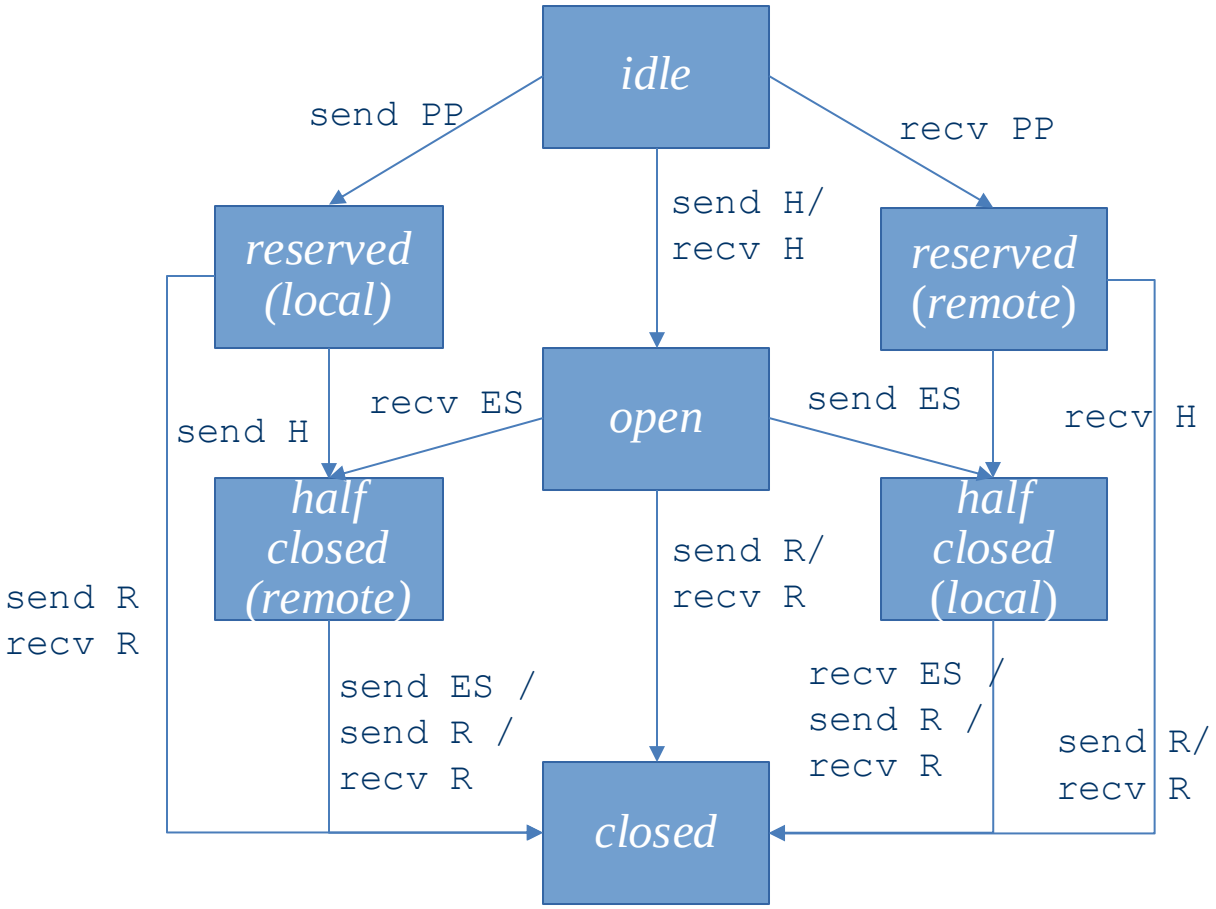
- One connection can have multiple streams open concurrently.
- Streams can be set up and used unilaterally or shared by client or server.
  - can be closed by either party
- The order in which the frames are sent is relevant:
  - processed in the order they are received
  - identified by an integer assigned by the initiator of the "stream"



HTTP/2 request and response multiplexing within a TCP shared connection

Image source: https://hpbn.co/http2/#streams-messages-and-frames

HTTP/2 >> Life cycle of a stream



H: HEADERS frame (with implied CONTINUATION)
PP: PUSH_PROMISE frame (with implied CONTINUATION)
ES: END_STREAM flag
R: RST_STREAM frame

In the `reserved` state, `WINDOW_UPDATE` or `PRIORITY` can be received, and `PRIORITY` can be sent

In the open state, any frame may be sent and received

In the `half closed (local)` state, frames of any type may be received

In the `half closed (remote)` state, frames of any type may be sent

HTTP/2 >> Priority of "streams"

- A client can assign a priority for a new stream by including prioritization information in the HEADERS frames that open the stream.
- Later the stream priority can be changed with a PRIORITY frame
- **Priority** ⇨ How would you prefer the server (or the client) to allocate resources to this flow when managing concurrent streams? In other words, when the capacity to send is limited, which streams can transmit frames?
- Streams can be prioritized by marking them as dependent on the completion of other streams
  - a tree is formed with parent and child streams
- Each dependency is assigned a relative weight
- The processing of concurrent streams in a particular order cannot be forced using this priority. This is only a suggestion.

HTTP/2 >> Error handling and connection closure

- Two types of errors can be handled:
  - Connection error ⇨ disables the entire connection
    - should send a GOAWAY frame with the “stream-id” of the last *stream* from which messages were successfully received.
    - After the transmission, the TCP connection must be closed.
  - Stream error ⇨ in an individual stream
    - Sends a RST_STREAM frame with the “stream-id”, including an error code
- A list of error codes is defined indicating the cause, e.g., PROTOCOL_ERROR, INTERNAL_ERROR, SETTINGS_TIMEOUT, REFUSED_STREAM, HTTP_1_1_REQUIRED…

- A connection can be closed by either party at any time, influencing the streams that are in state “*open*” or “*half-closed*”

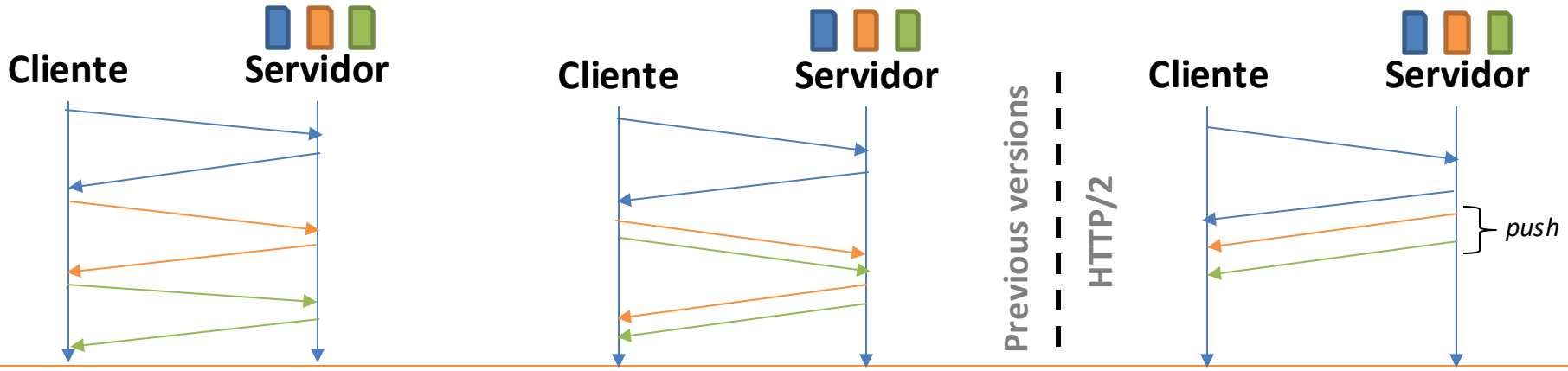
HTTP/2 >> PUSH from the server

- It is a new interaction model, where the **server** sends responses/data to a client without an explicit request ("**push**"), anticipating the client's needs
  - e.g., when the main resource requested includes other resources that are not embedded
  - therefore, the push is associated with a previous request initiated by the client.
- It uses a PUSH_PROMISE frame.
- A client can disable that push (SETTINGS_ENABLE_PUSH)
  - it is negotiated at each hop independently.

From HTTP/2 IN ACTION by BARRY POLLARD, Copyright 2018.

HTTP/2 >> PUSH from the server

- It is a new interaction model, where the **server** sends responses/data to a client without an explicit request ("**push**"), anticipating the client's needs
  - e.g., when the main resource requested includes other resources that are not embedded
  - therefore, the push is associated with a previous request initiated by the client.
- It uses a PUSH_PROMISE frame.
- A client can disable that push (SETTINGS_ENABLE_PUSH)
  - it is negotiated at each hop independently.



HTTP/2 >> Conclusions

- The most widely used browsers and servers have full HTTP/2 support for a few years now.
- HTTP/2 improves performance over HTTP/1.1, in terms of latency and flow control of TCP connections.
  - avoids HOLB at HTTP level
- However, HOLB at TCP level is still present.
  - **QUIC** protocol ("*Quick UDP Internet Connections*")
- **HTTP/3** has been defined in RFC 9114, September 2022, based on QUIC, instead of TCP/TLS.

Content >> HTTP/2

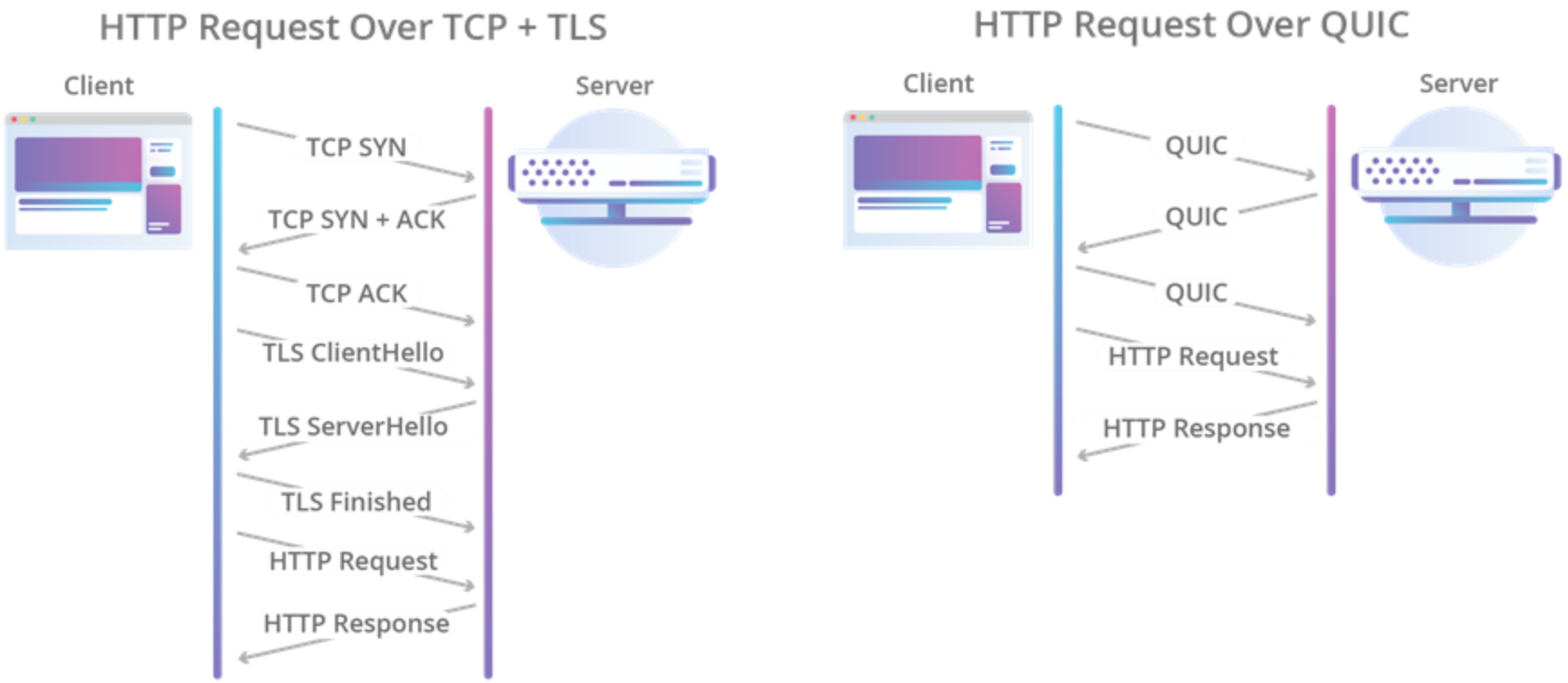
4. Other protocols
   A. QUIC
   B. HTTP/3

Index

QUIC >> General overview

- Quick UDP Internet Connections emerged as an experimental protocol over the transport layer designed by J. Roskind (Google), 2013
  - submitted to the IETF for consideration in 2015
- Standardized by the IETF in May 2021, **RFC 9000**, as a protocol for **multiplexed and secure** connections over **UDP** (User Datagram Protocol).
- Similar to HTTP/2, but emerges as an alternative to TCP
- Designed to provide security equivalent to TLS, with lower connection and transport latency.
- It is enabled by default in Chromium and introduced in Opera 16.
- When the client accesses a web page, certificates and keys are sent.
- Client and server exchange QUIC packets, which contain **frames** with **control** information and application **data**.

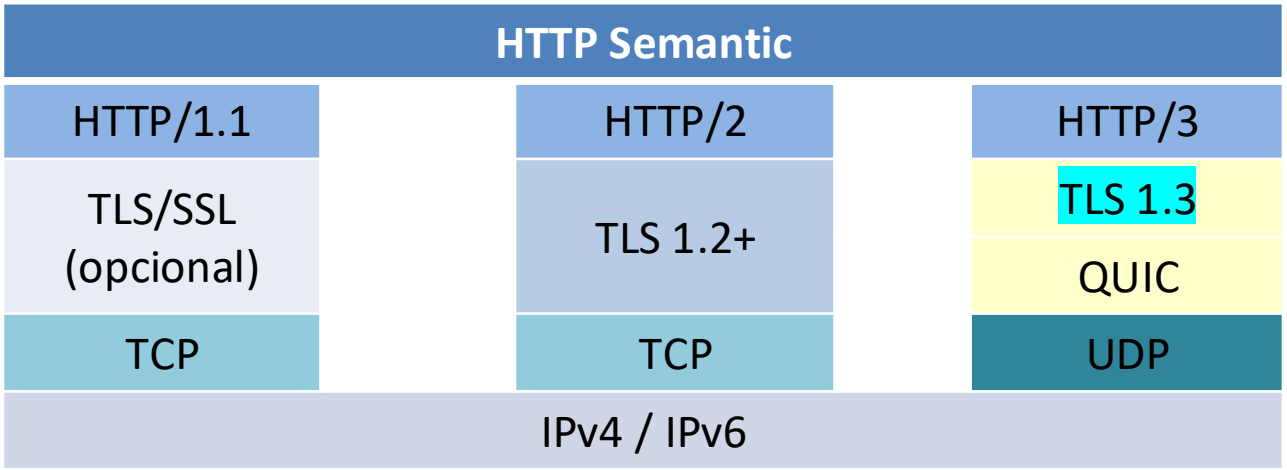
QUIC >> General overview



Source: cloudflare.com

HTTP/3 >> General overview

- **RFC 9114** describes how to map HTTP over QUIC, identifying features of HTTP/2 that are assumed by QUIC and how extensions can be transferred to HTTP/3.
- When a client knows that the server implements HTTP/3, it opens a QUIC connection.
  - QUIC provides protocol negotiation, stream-based multiplexing and flow control.



- A client can attempt to access a resource with "https", using the "h3" token in the Application-Layer Protocol Negotiation (ALPN, *Application-Layer Protocol Negotiation*) in the TLS handshake.

HTTP/3 >> General overview (II)

- **Similarities** with **HTTP/2**: connection management, HTTP semantics
  - persistent connections across multiple requests
  - HTTP messages consist of: header section, content (optional) and "trailer" section
    - request fields: `:method`, `:scheme`, `:path`
    - response fields: `:status`
- A client sends an HTTP request on a request stream, which is a bidirectional QUIC stream initiated by the client.
  - a single request must be sent on a given stream
  - the server sends zero or more HTTP responses on the same stream, followed by a single final HTTP response
- Push messages are sent on a one-way QUIC stream initiated by the server.
- **Differences** with **HTTP/2**: does not support the "Upgrade" mechanism, does not provide a mechanism to indicate priority, compression type, frame format: `Type, Length, Payload (..)`

HTTP/3 >> Example HTTP/3 request