WEB DAY 2025

You don't really know HTTP

Erik Pellizzon
Software Engineer

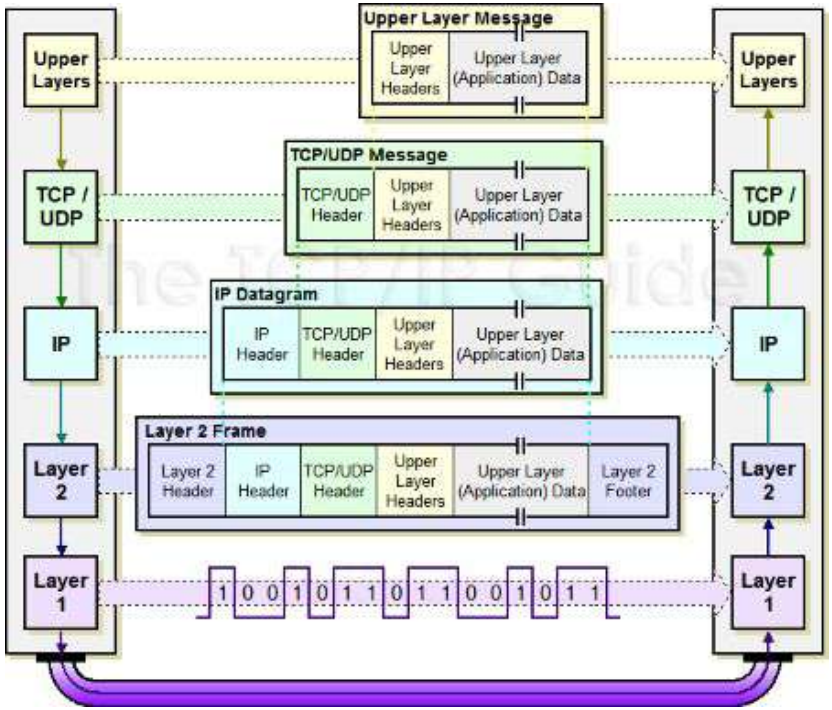
improove

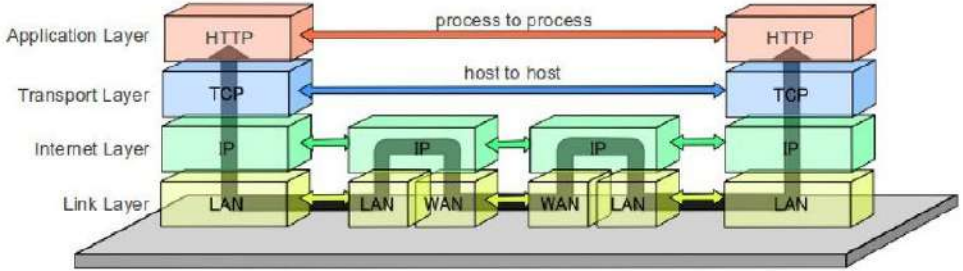
WEB DAY 2025

Kudos

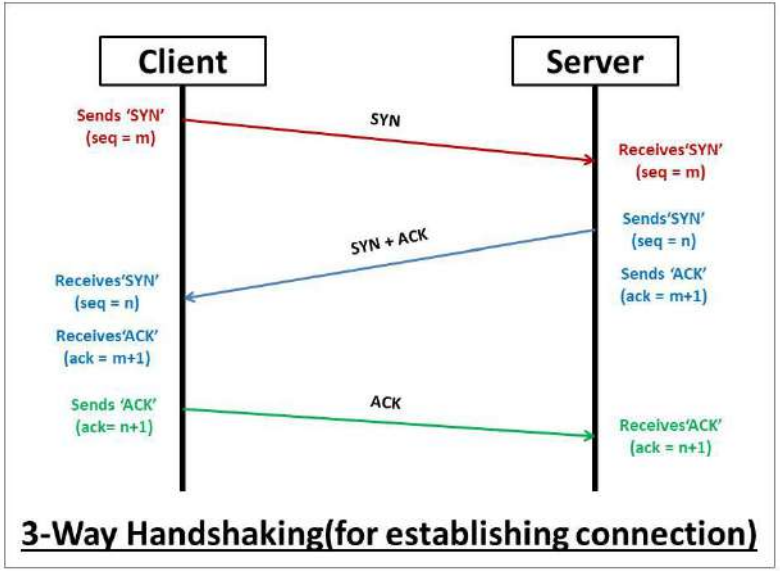
improove

Network


Upper Layers
TCP / UDP
IP
Layer 2
Layer 1
Upper Layer Message
Upper Layer Headers
Upper Layer (Application) Data
TCP/UDP Message
TCP/UDP Header
Upper Layer Headers
Upper Layer (Application) Data
IP Datagram
IP Header
TCP/UDP Header
Upper Layer Headers
Upper Layer (Application) Data
Layer 2 Frame
Layer 2 Header
IP Header
TCP/UDP Header
Upper Layer Headers
Upper Layer (Application) Data
Layer 2 Footer
1 0 0 1 0 1 1 0 1 1 0 0 1 0 1 1
Upper Layers
TCP / UDP
IP
Layer 2
Layer 1


Application Layer
Transport Layer
Internet Layer
Link Layer
HTTP
process to process
HTTP
TCP
host to host
TCP
IP
IP
IP
IP
LAN
LAN
WAN
WAN
LAN
LAN

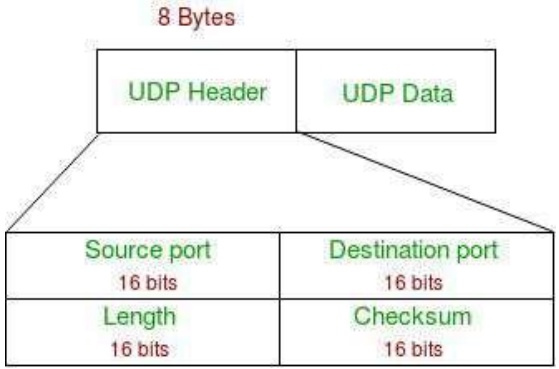
TCP (1974)



TCP

UDP

UDP (1980)



HTTP

HTTP History

Version	Year introduced	Current status	Usage in August 2024	Support in August 2024
HTTP/0.9	1991	Obsolete	0	100%
HTTP/1.0	1996	Obsolete	0	100%
HTTP/1.1	1997	Standard	33.8%	100%
HTTP/2	2015	Standard	35.3%	66.2%
HTTP/3	2022	Standard	30.9%	30.9%

First web server (1990) by Tim Berners Lee

HTTP/0.9 is based on it:

- only GET request
- only HTML response
- no headers or status codes
- designed to serve static files

HTTP/1.0 (Introduction from its paper)

The Hypertext Transfer Protocol (HTTP) is an **application**-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification reflects **common usage** of the protocol referred to as "HTTP/1.0". This specification describes the features that seem to be **consistently implemented** in most HTTP/1.0 clients and servers. Those features of HTTP for which implementations are usually consistent are described in the main body of this document. Those features which have few or inconsistent implementations are listed in **Appendix D**.

HTTP/1.0

Request

POST / HTTP/1.0**\r\n**

headers

Accept: */***r\n**

Referer: https://www.google.com/**\r\n**

User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/133.0.0.0 Safari/537.36**\r\n**

Content-Length: 13**\r\n**

Content-Type: text/html**\r\n**

\r\n

<h1>ping</h1>

body

Response

200 OK**\r\n**

Server: Apache**\r\n**

\r\n

time

1) TCP connection open

2) Client Request

3) Server Response

4) TCP connection close

1XX	Informational codes	The server acknowledges and is processing the request.
2XX	Success codes	The server successfully received, understood, and processed the request.
3XX	Redirection codes	The server received the request, but there's a redirect to somewhere else (or, in rare cases, some additional action other than a redirect must be completed).
4XX	Client error codes	The server couldn't find (or reach) the page or website. This is an error on the site's side.
5XX	Server error codes	The client made a valid request, but the server failed to complete the request.

Methods

GET

HEAD

POST

Appendix D)

PUT

DELETE

~~LINK~~ (removed in 1.1)

~~UNLINK~~ (removed in 1.1)

HTTP/1.1

Methods:

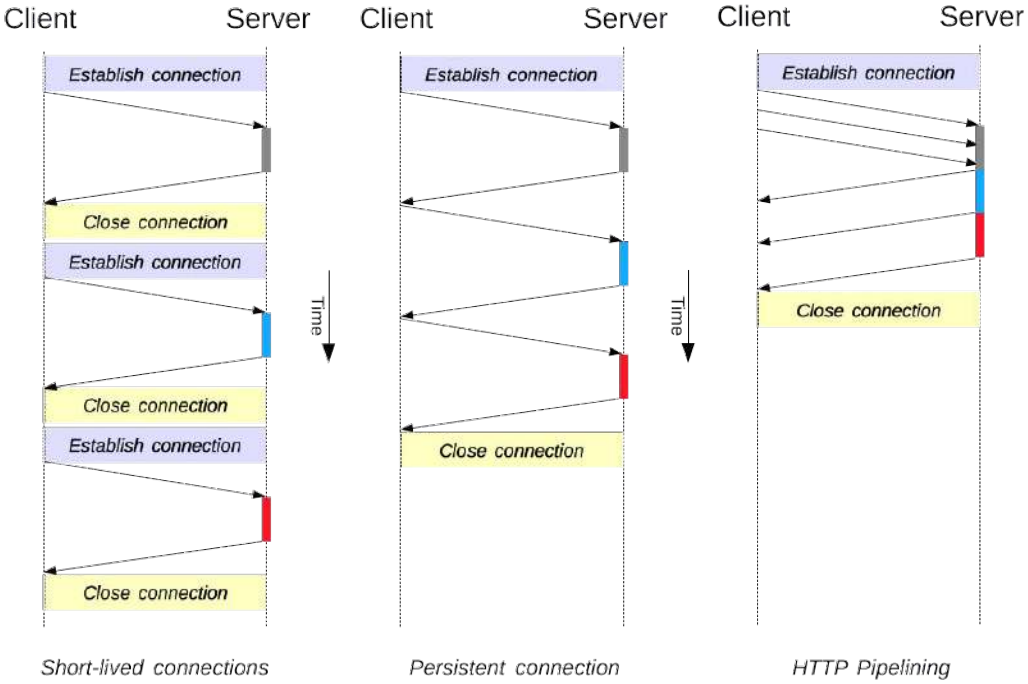
- OPTIONS (e.g. CORS)
- GET
- HEAD (no body in response)
- POST (create)
- PUT (full update)
- PATCH (partial update)
- DELETE
- TRACE (ping, not always implemented)
- CONNECT (e.g. HTTP proxies)

Header names are now case insensitive.

Host header is now mandatory, we can start hosting multiple websites in the same server (*Host: www.google.it*).

There are many new standard HTTP headers (here the list), the result of 18 years of additions.

Persistent connections



Connection: close

Connection: keep-alive (default)

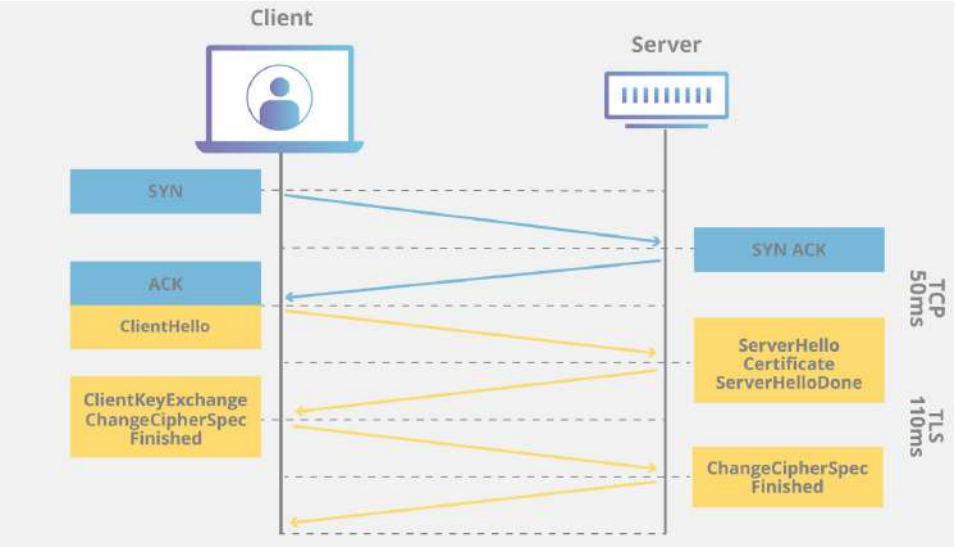
Disabled by default, implemented by HTTP/2 as multiplexing

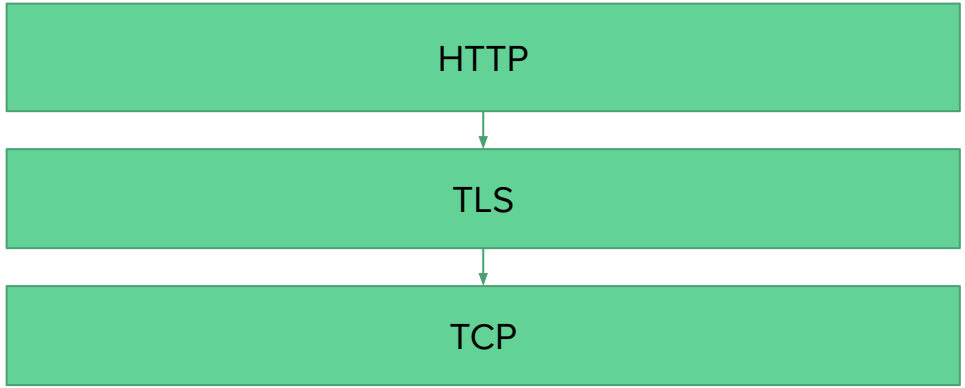
HTTPS (optional)

We encrypt the data sent over the connection so so they can't be read by third parties.

The protocol for the data format remains the same (HTTP), but everything is sent (and received) securely using SSL (deprecated) / TLS.

TLS requires a **reliable connection** in order to work (such as TCP).





THE END?

Where common knowledge ends

HTTP/2 (2015): Security & Performance

- **TLS 1.2+** is mandatory (security)
- h2 vs h2c (h2c is almost never used)
- Based on SPDY, an experimental Google project
- Binary, **frame**-based, supports multiple **concurrent requests**
- Only 1 TCP connection
- Header names must be lowercase (to avoid problems related to case sensitivity in non-compliant implementations)
- Not (directly) human readable anymore

HTTP/2.0 request:

```
00 00 9D 01 25 00 00 00  01 00 00 00 00 B6 41 8A    ..%  .    .A.
90 B4 9D 7A A6 35 5E 57  21 E9 82 00 84 B9 58 D3   ...z.5^W!.. ..X.
3F 85 61 09 1A 6D 47 87  53 03 2A 2F 2A 50 8E 9B   ?.a..mG.S.*/*P..
D9 AB FA 52 42 CB 40 D2  5F A5 11 21 27 51 8B 2D   ...RB.@._..!'Q.-
4B 70 DD F4 5A BE FB 40  05 DE 7A DA D0 7F 66 A2   Kp..Z..@..z...f.
81 B0 DA E0 53 FA D0 32  1A A4 9D 13 FD A9 92 A4   ....S..2........
96 85 34 0C 8A 6A DC A7  E2 81 04 41 04 4D FF 6A   ..4..j.....A.M.j
43 5D 74 17 91 63 CC 64  B0 DB 2E AE CB 8A 7F 59   C]t..c.d.......Y
B1 EF D1 9F E9 4A 0D D4  AA 62 29 3A 9F FB 52 F4   .....J...b):..R.
F6 1E 92 B0 D3 AB 81 71  36 17 97 02 9B 87 28 EC   .......q6.....(.
33 0D B2 EA EC B9
```

HTTP/1.1 request:

```
GET / HTTP/1.1
Host: demo.nginx.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Chrome/47.0.2518.0
```

Preface

Each HTTP/2 connection initially sends this data (preface):
PRI * HTTP/2.0\r\n\r\n**SM**\r\n\r\n

If the server supports HTTP/2, it recognizes the sequence and prepares to read HTTP/2 binary data. Usually a server that supports HTTP/2 also supports **HTTP/1.1**, to maintain **compatibility** with older browsers.

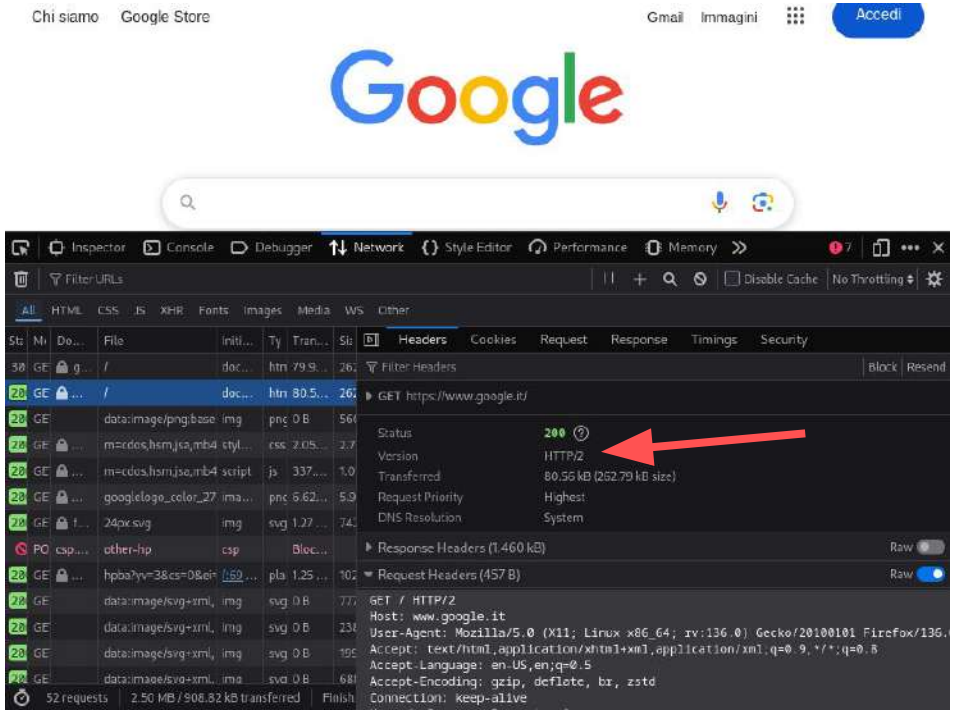
If the server supports only HTTP/1.1, it will return an error similar to **“PRI method is not supported”**.
This sequence is similar to an HTTP/1.1 request and is designed to return a readable error without breaking the HTTP/1.1 parser.

Compatibility

HTTP/2, despite differences in implementation, keeps the same HTTP basic logic:

- Methods (GET, POST, etc.)
- Status codes (200, 404, etc.)
- Body
- Headers
- Request & Response

Clients that implement it use the same public interface (e.g. *fetch()*) to perform the HTTP request, and the underlying protocol chosen is hidden from the user.



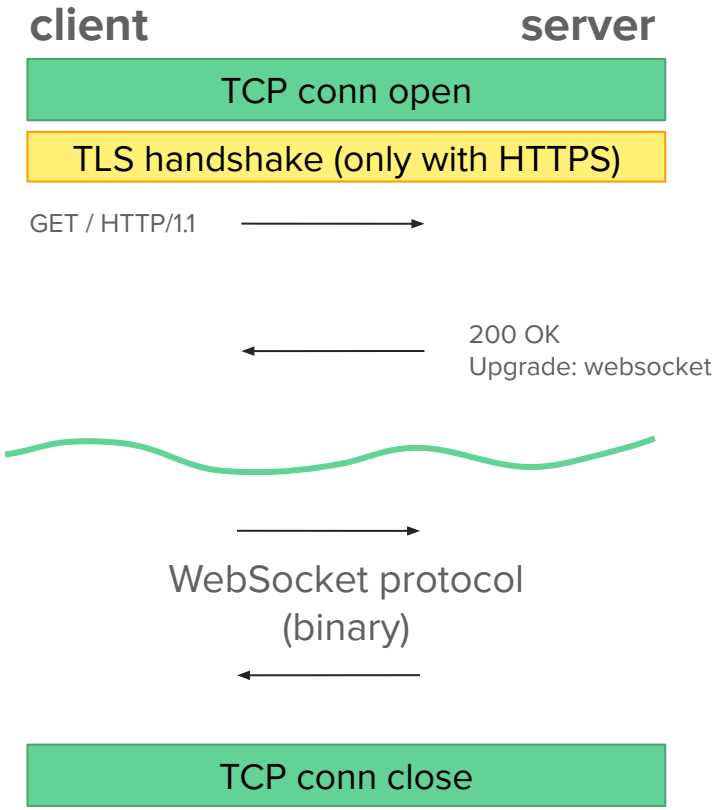
This “Raw” request is fake, generated by Firefox using request data

“Upgrade” header

This header is supported only by HTTP/1.1.

Initially a normal HTTP/1.1 request is made, if the server includes this header in the response, from there on the underlying connection switches to the specified protocol.

It’s not used for an HTTP/2 handshake because the **handshake overhead is very high**. Moreover, if it’s used with an HTTP connection (instead of HTTPS), TLS encryption will be absent, and in this case only h2c (not h2) is supported.

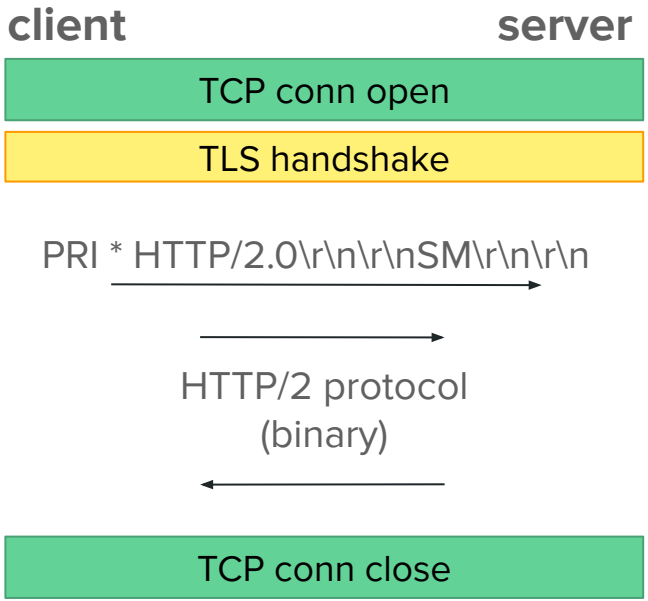


Direct HTTP/2

In this mode we send HTTP/2 data directly to the server, without knowing if the server supports the protocol.

It's obviously not currently used because it isn't acceptable to have an error because we haven't checked whether the new protocol is supported.

Maybe in the future, when every server will implement it?

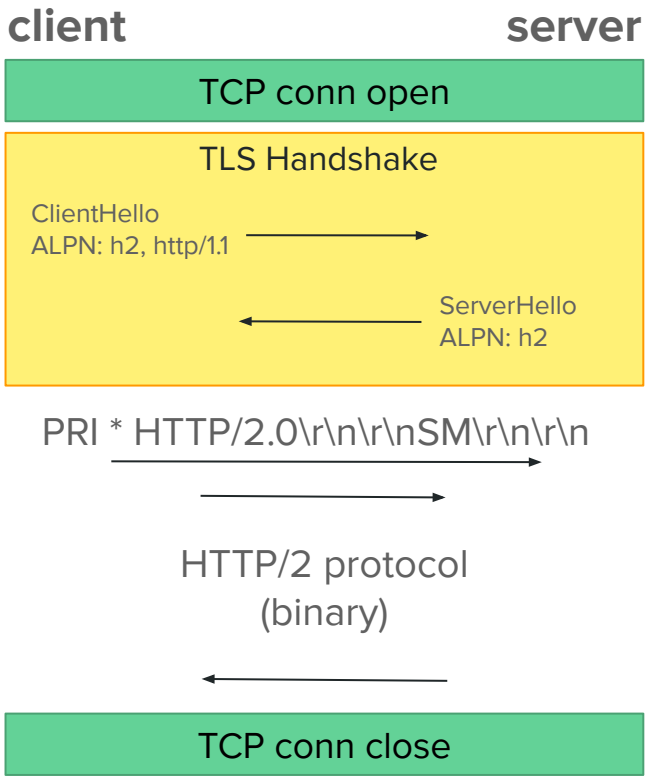


ALPN (TLS extension)

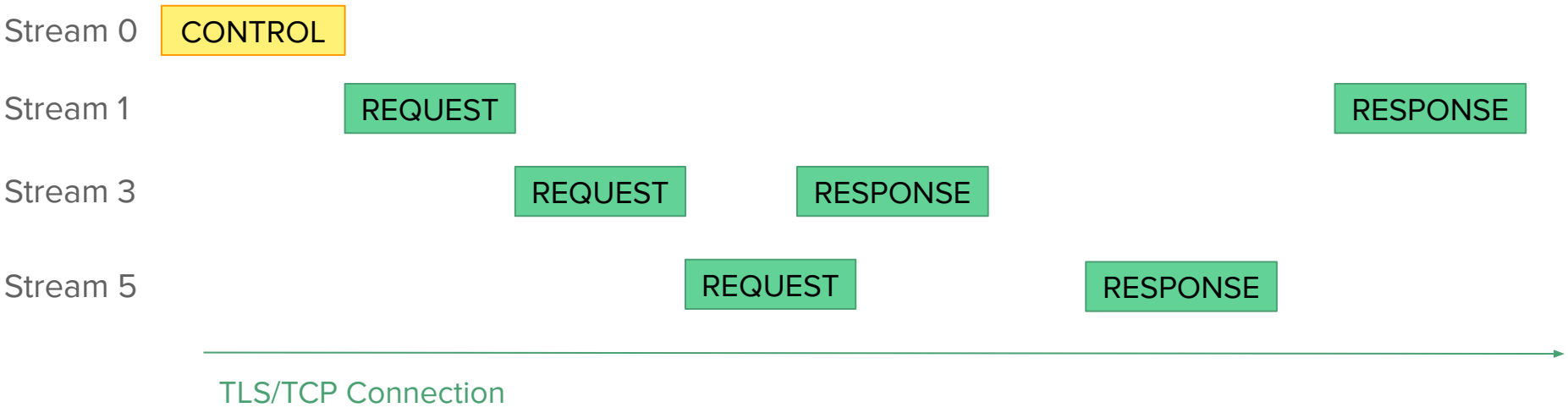
TLS is flexible and supports additional data inside extensions included in the ClientHello.

Along with HTTP/2, ALPN has been created, to agree on a common protocol directly during the TLS handshake, basically eliminating the overhead required by Upgrade.

If ALPN is not supported by the server, the browser makes an HTTP/1.1 request (and it may become HTTP/2 later if the server specifies the “Upgrade: h2” header).

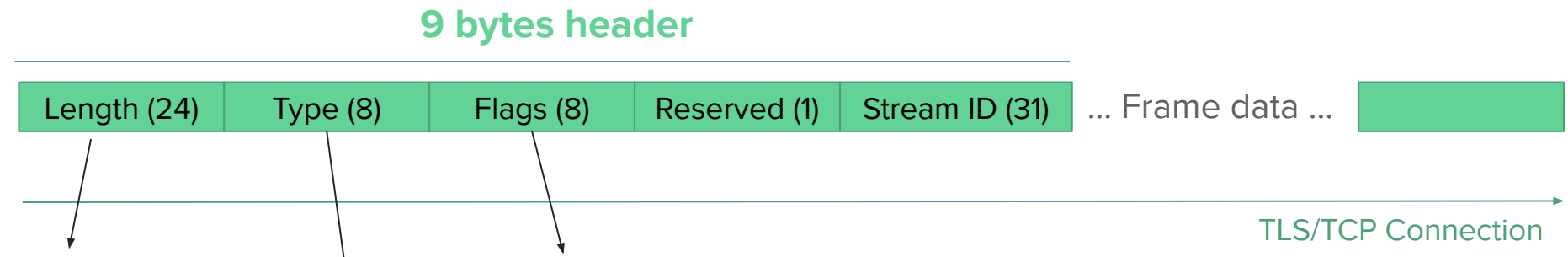


Multiplexing



0 = Control stream
Odd = Client created streams
Even = Server created streams

Frames



Up to 2^24 -1 bytes, ~16MB (the length of the frame data payload only)

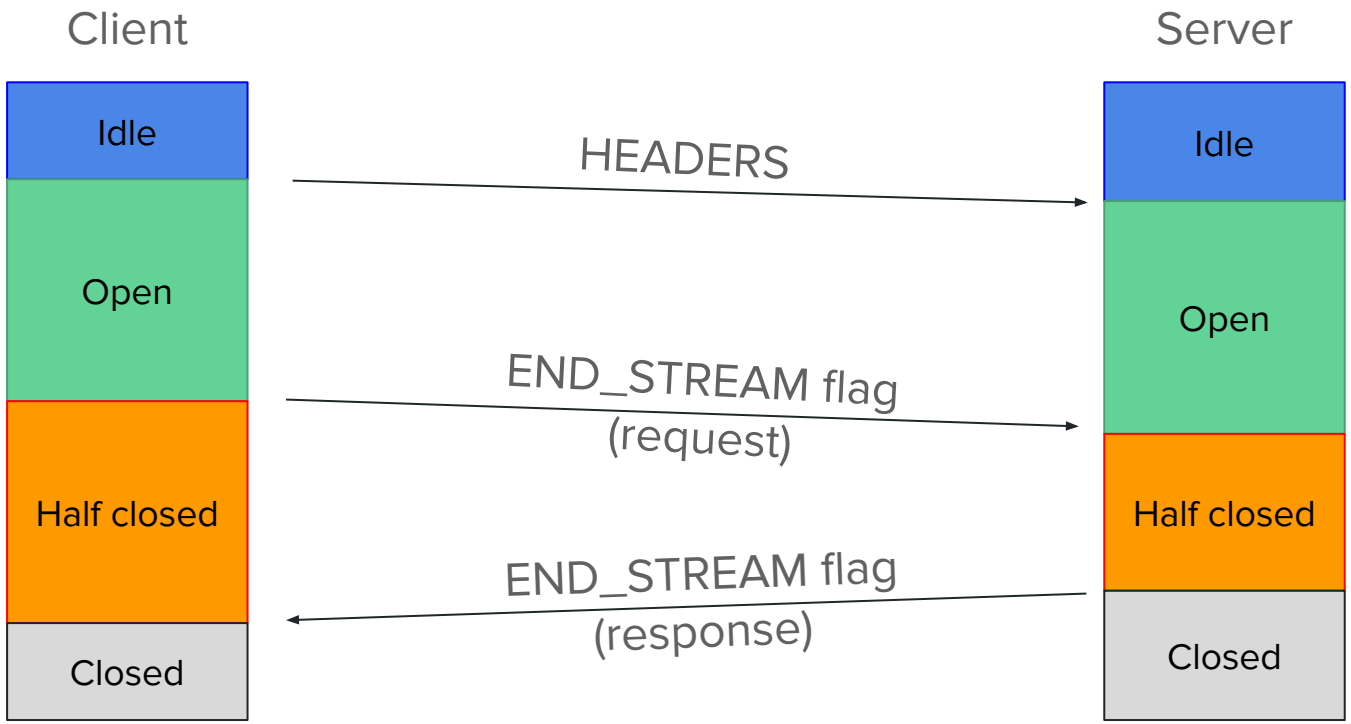
DATA
HEADERS
CONTINUATION

Their meaning depend on the frame type

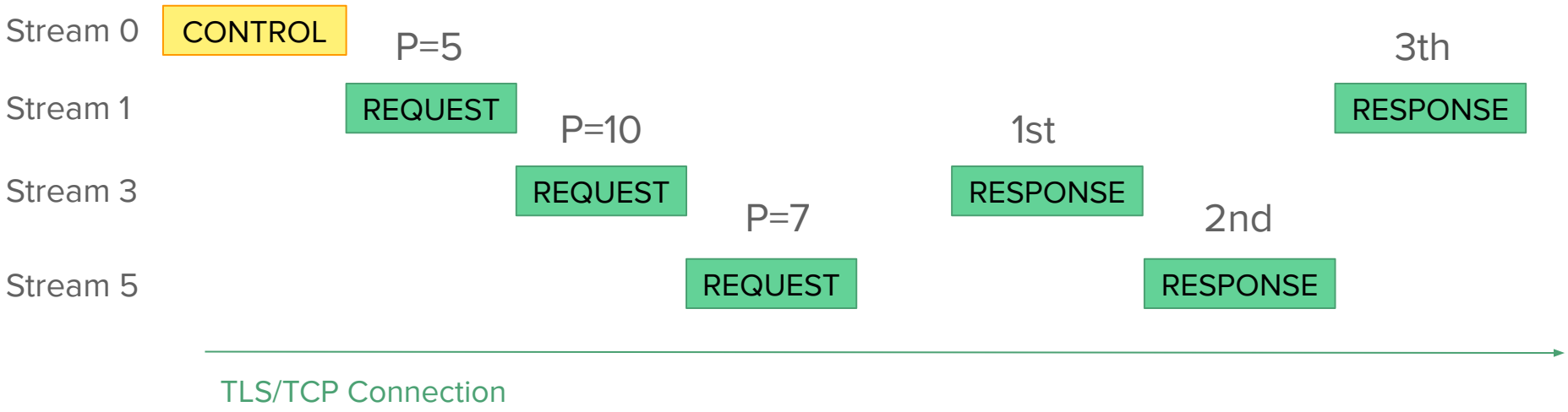
RST_STREAM, terminate specified stream ID
PRIORITY_UPDATE, RFC9218
PRIORITY, deprecated
PUSH_PROMISE, deprecated

Connection Control frames (Stream 0)
SETTINGS
PING
GOAWAY, terminate connection
WINDOW_UPDATE, flow control

Stream state

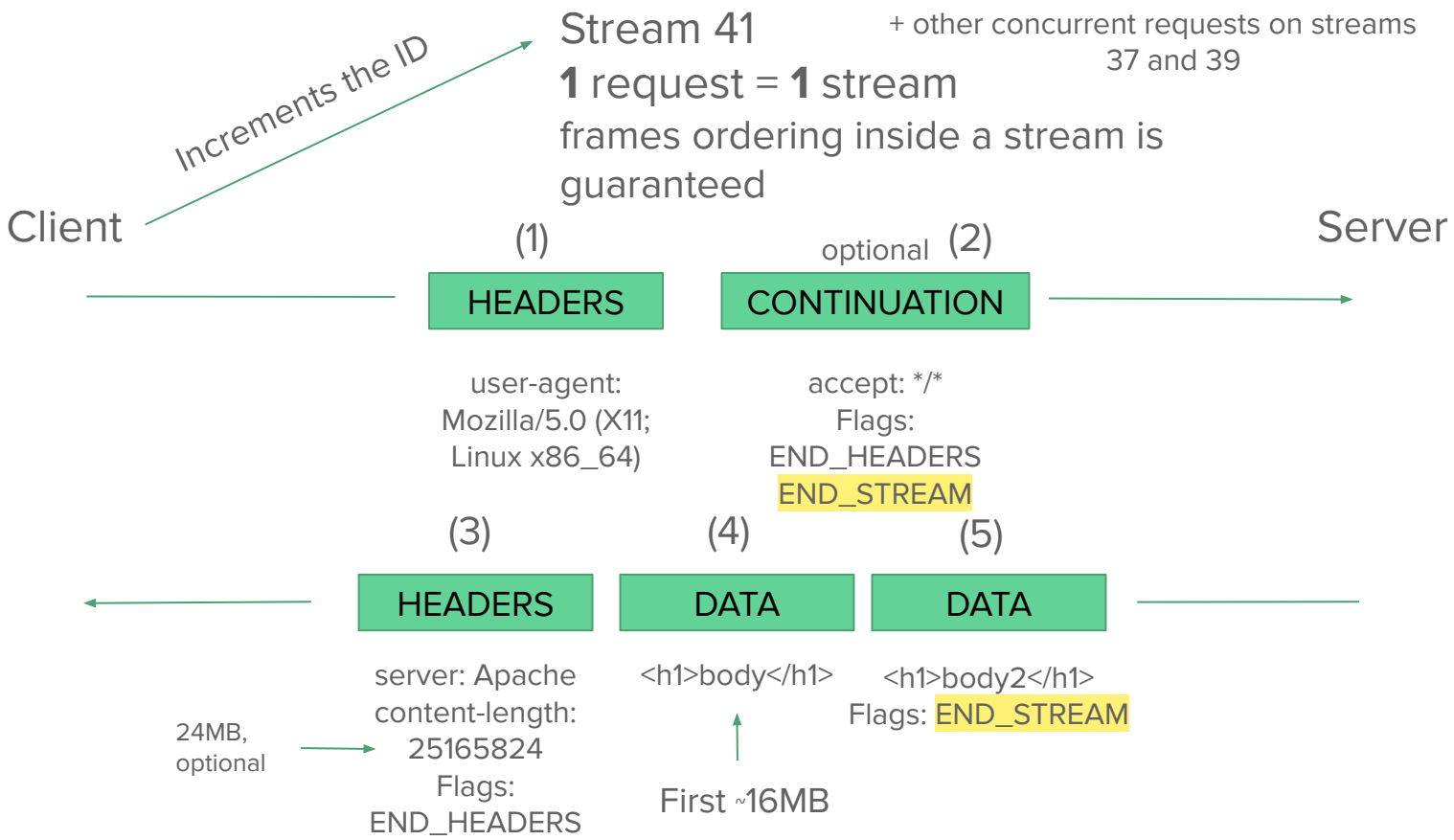


Stream Priority



Client priority is just a suggestion,
server could skip it

Request & Response



Headers

Pseudo Headers

Name	Value
:method	GET
:path	/
:authority	google.it
:scheme	https
user-agent	Mozilla/5.0 (Android 15; Mobile; rv:133.0) Gecko/133.0 Firefox/133.0
accept	text/html
accept-language	it-IT

HPACK compression

Client

Encoder
&
Decoder

ID	Name	Value
2	:method	GET

Static Table

Server

Encoder
&
Decoder

Request

2, 62=header1: value1

62=header2: value2

Client Decoder
Dynamic Table, 4KB default

ID	Name	Value
62	header2	value2

Duplicated in client and server

Server Decoder
Dynamic Table, 4KB default

ID	Name	Value
62	header1	value1

Assigned by client encoder

Assigned by server encoder

Request

2, 62

62

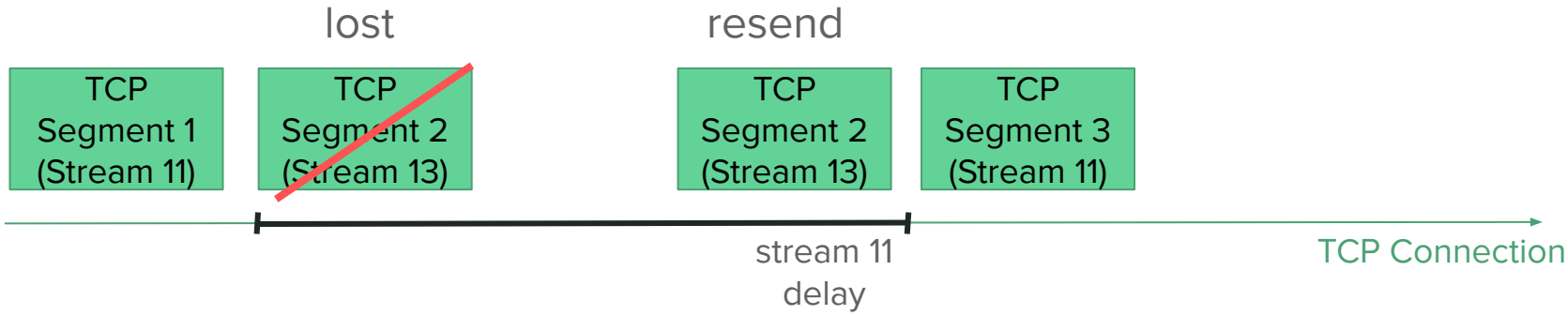
Downsides

TCP

- Issues with unstable connections (e.g. 4G/5G in mountain), a lot of TLS handshake round-trips

- Limited concurrency, since we use only one connection for all the streams

- Packet loss (TCP is ordered and every stream must wait for the resend)



HTTP/3

2022

Erik Pellizzon

https://erikpelli.pp.ua

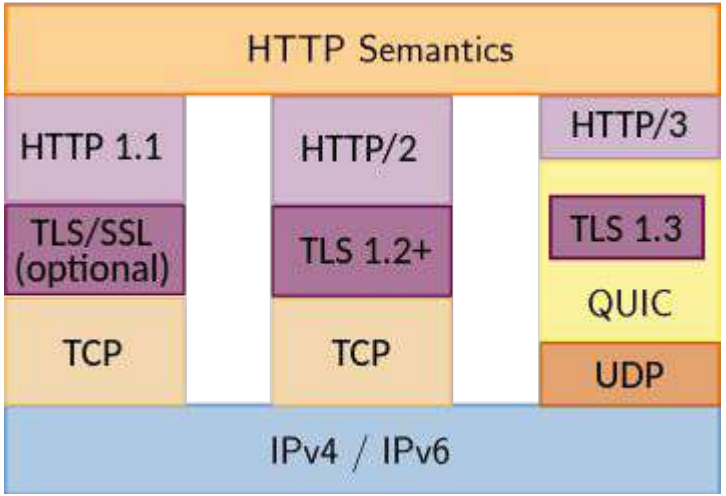
Go Developer

Freelancer

LinkedIn

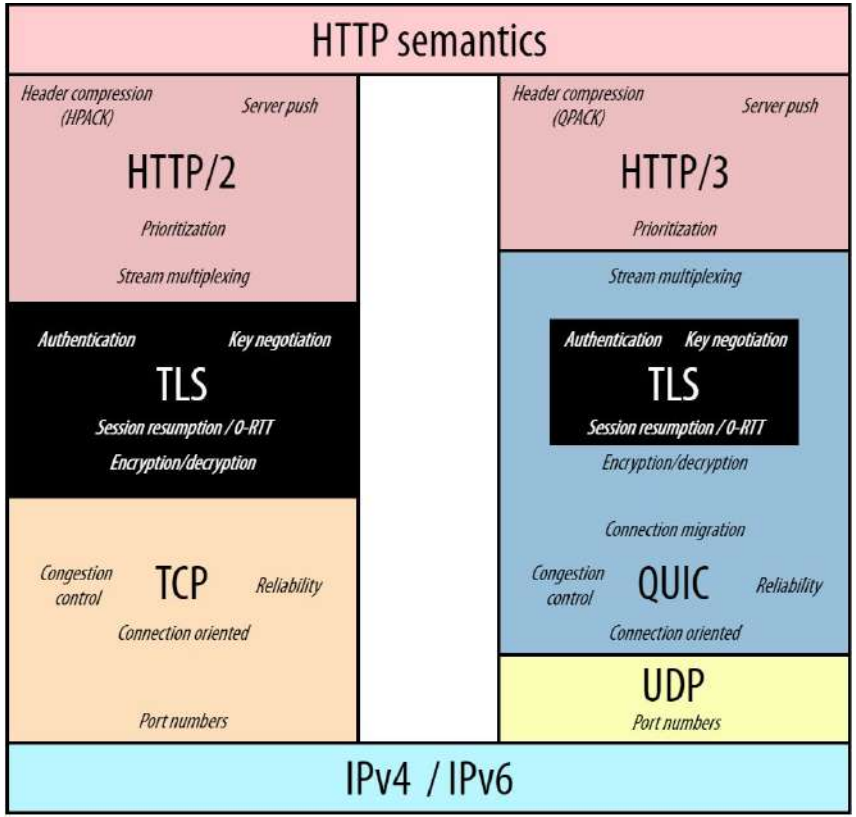
Looking for collaborations

HTTP/3: HTTP/2 over QUIC

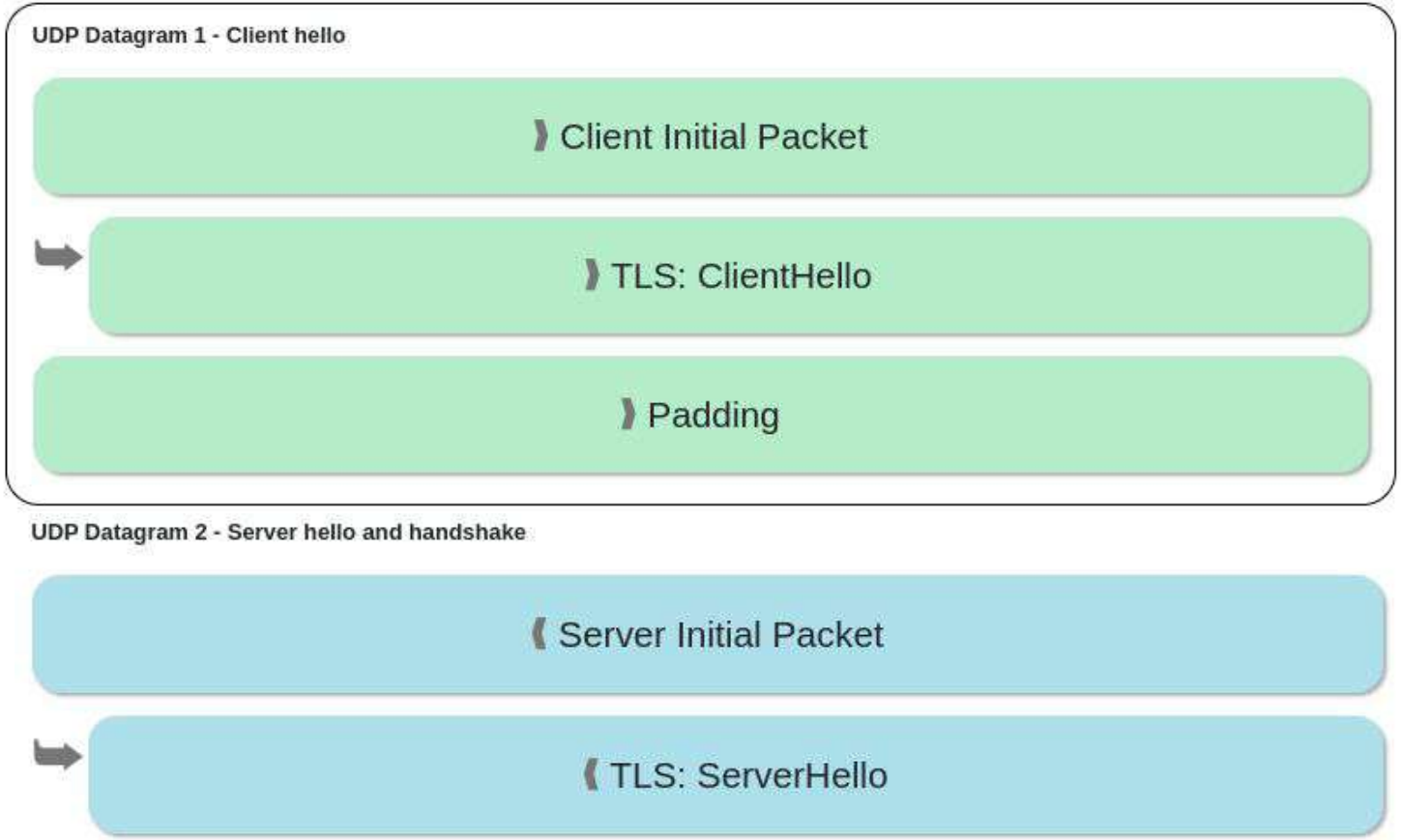


QUIC

- Application-level protocol (can be easily updated, unlike TCP)
- Reliable connection built over UDP (which is unreliable)
- Everything is encrypted using TLS 1.3

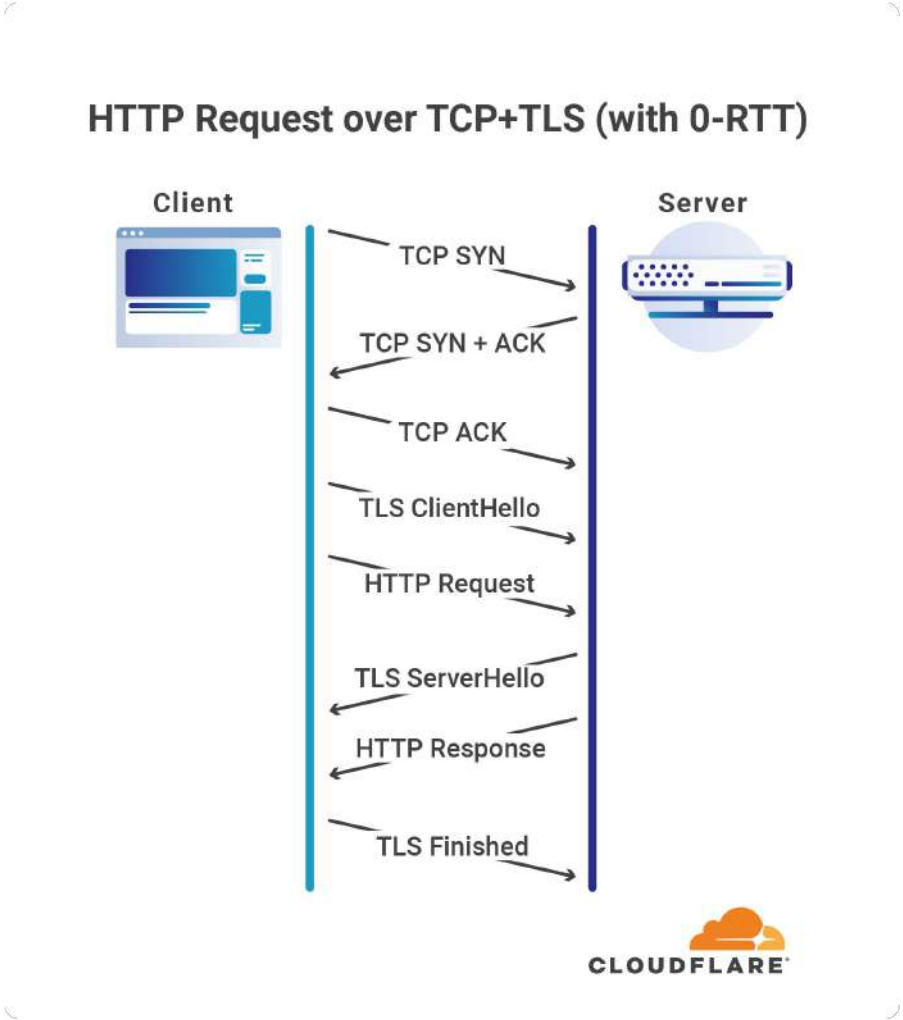


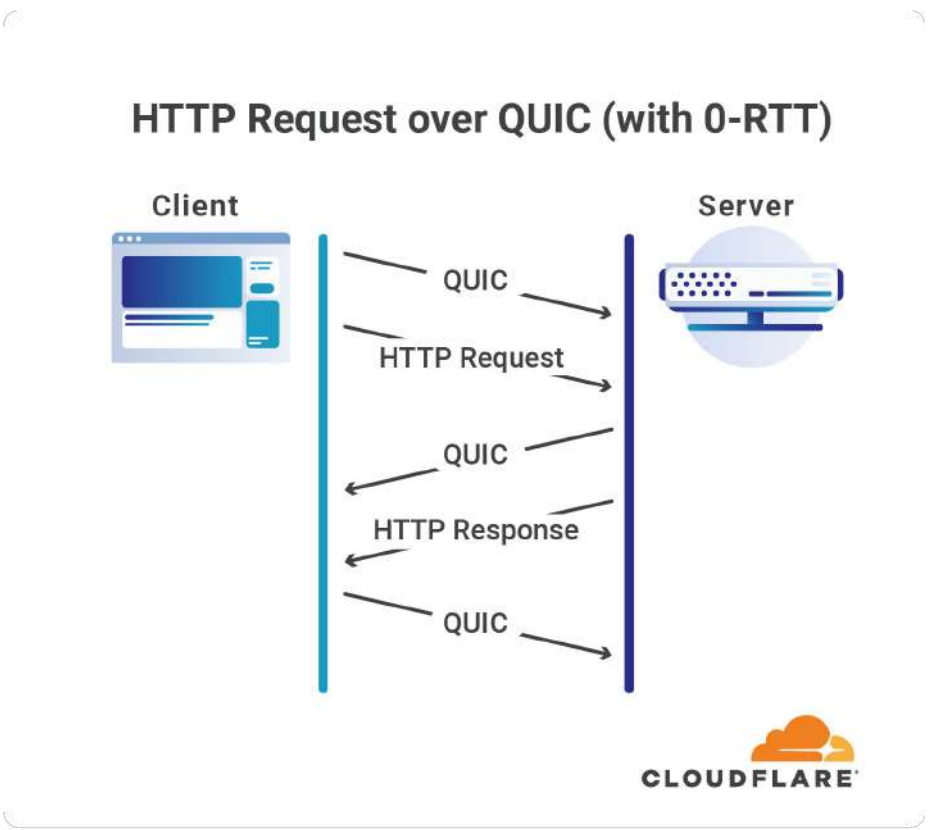
QUIC Initial Packet



0RTT (TLS 1.3)

To reduce the time required to establish a new connection, a client that has previously connected to a server may cache certain parameters from that connection and subsequently set up a 0-RTT connection with the server. This allows the client to send data immediately, without waiting for a handshake to complete.

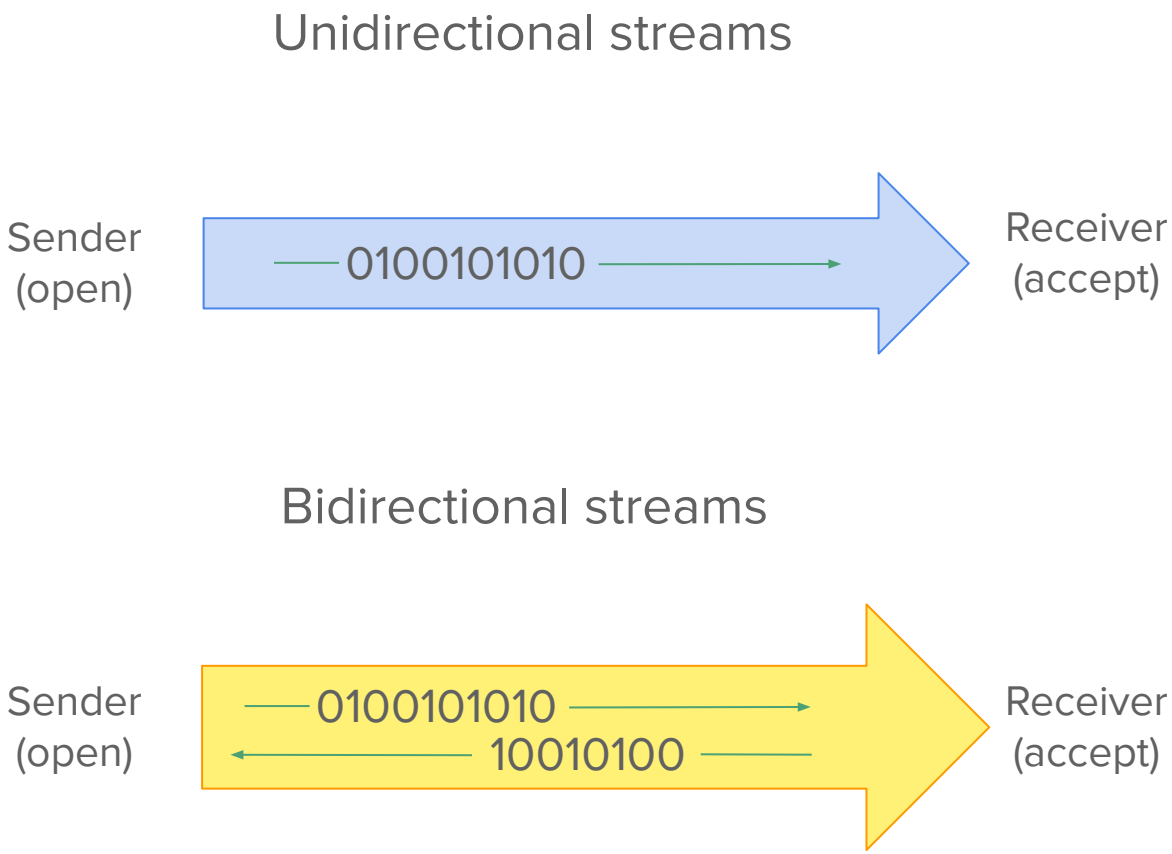

HTTP Request over TCP+TLS (with 0-RTT)
Client
Server
TCP SYN
TCP SYN + ACK
TCP ACK
TLS ClientHello
HTTP Request
TLS ServerHello
HTTP Response
TLS Finished
CLOUDFLARE


HTTP Request over QUIC (with 0-RTT)
Client
Server
QUIC
HTTP Request
QUIC
HTTP Response
QUIC
CLOUDFLARE

Source

QUIC streams

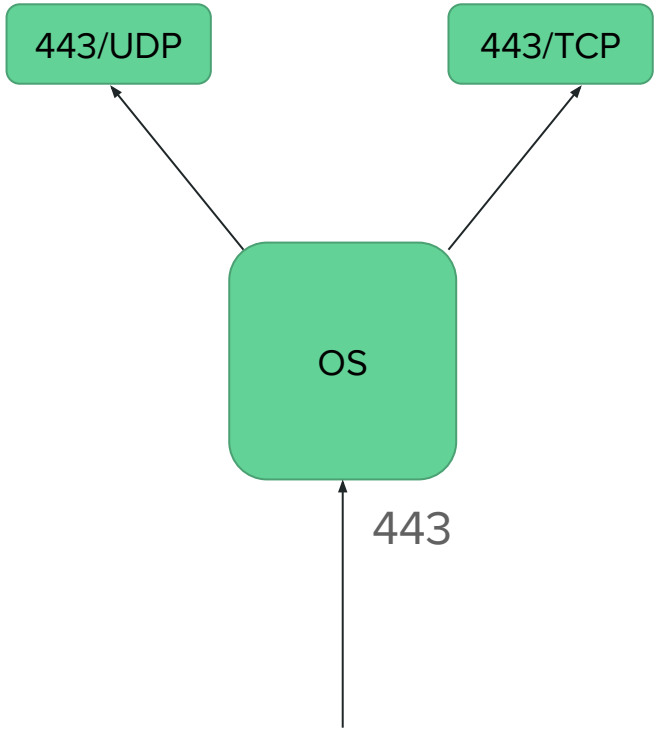
Data sent within a stream are ordered, but each stream is independent from the others



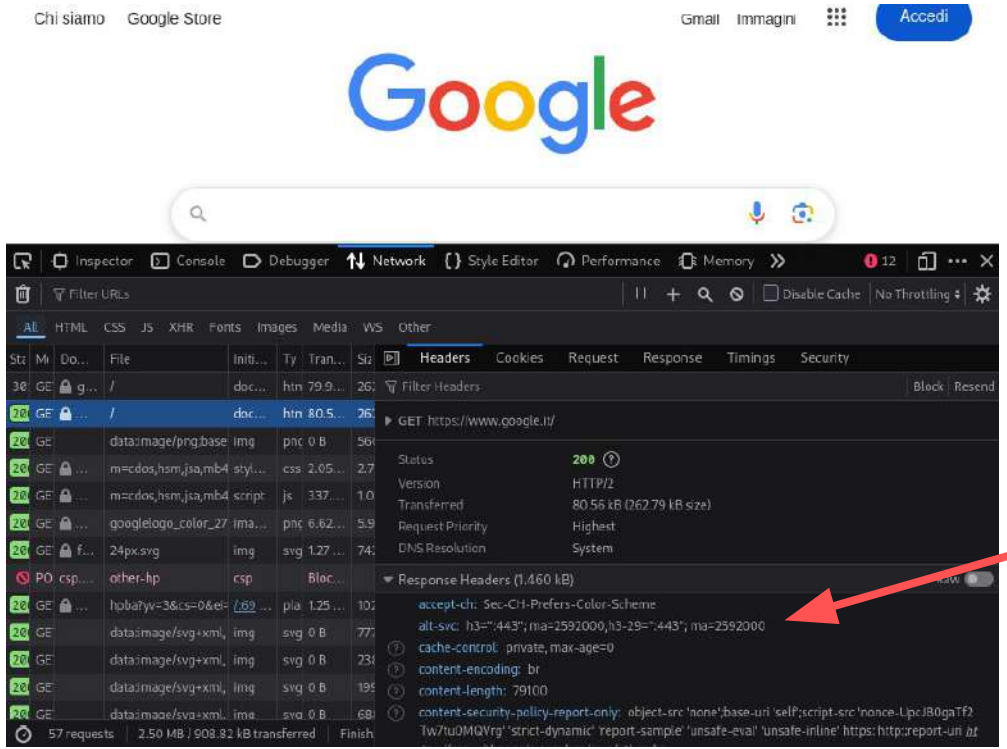
Compatibility

QUIC runs over UDP, we can use both TCP server (HTTP/1.1, HTTP/2) and UDP (HTTP/3) on the same port (443).

Like HTTP/2, the public interface for making requests is the same and the HTTP version used for the request is chosen by the implementation.



Alt-Svc header



1 month

HTTPS DNS record

How do HTTPS records work?

An HTTPS record can be set up for the domain `sample-test.com` as shown below:

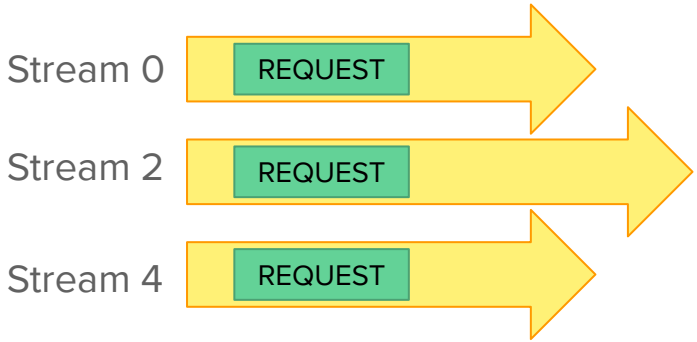
```
www.sample-test.com.        1800    IN  HTTPS   1 . alpn=h3,h3-29,h2 ipv4h l.
```

Here's what each element represents:

- `www.sample-test.com` is the domain name.
- `1800` is the Time To Live (TTL).
- `IN` represents the class.
- `HTTPS` signifies the record type.
- `1` is the priority, i.e., the number in the queue.
- `.` stands for the host if it is the same as the domain name.
- `alpn=h3,h3-29,h2` specifies the application protocol versions.
- `ipv4hint=1.2.3.4,9.8.7.6` specifies IPv4 addresses (this is optional.)
- `ipv6hint=2001:db8:3333:4444:5555:6666:7777:8888,2001:db8:3333:4444:CCCC:DDDD:EEEE:FFFF` specifies IPv6 addresses (this is also optional.)

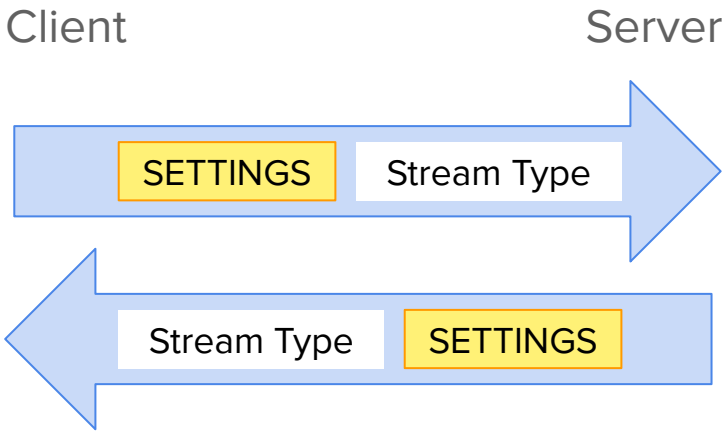
HTTP/3 streams

Bidirectional streams



Even = Streams created by client
Odd = server (not used by HTTP/3)

Unidirectional streams

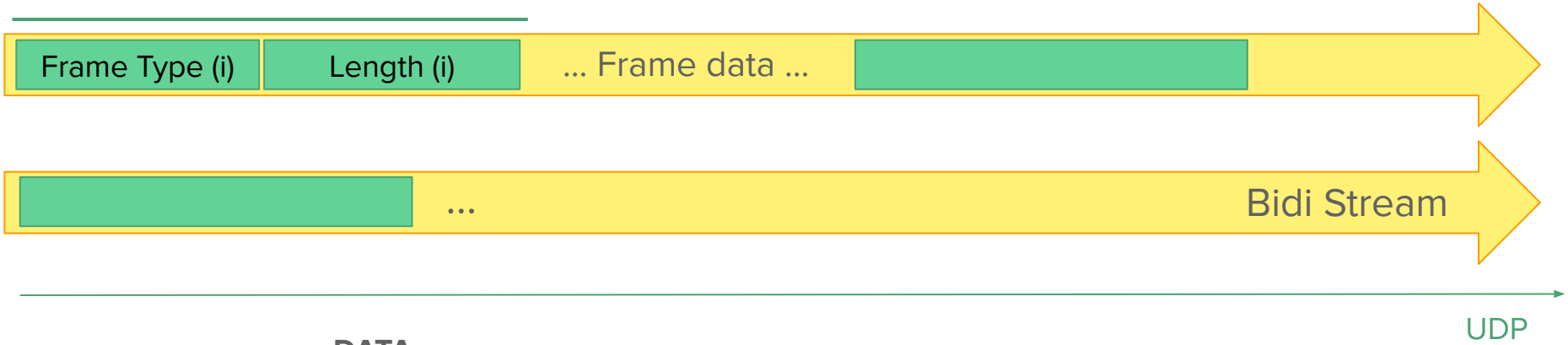


Two control unidirectional streams (type 0x00)

"This allows either peer to send data as soon as it is able."

Frames

variable length header
(up to 62 bits, aka 4,194,304TB)



DATA
HEADERS
~~CONTINUATION~~

not needed with variable length

PRIORITY_UPDATE
~~RST_STREAM~~
PUSH_PROMISE, deprecated
CANCEL_PUSH, deprecated
MAX_PUSH_ID, deprecated

handled by QUIC

~~"END_STREAM"~~ flag

Connection Control frames
SETTINGS
GOAWAY
~~PING~~
~~WINDOW_UPDATE~~

~~HPACK~~

QPACK

Based on unidirectional streams

Download the slides

WEB DAY 2025

Thanks!

improove