

What is HTTP?

If you are new to this theme, continue to read this block, but if you're familiar with this, skip this paragraph and go straight forward to the next one.

For those that are new to this theme, Hypertext Transfer Protocol (HTTP) is an application protocol that is, currently, **the foundation** of data communication for the World Wide Web.

HTTP is based on the Client/Server model. Client/Server model can be explained as two computers, Client (receiver of service) and Server (provider of service) that are communicating via requests and responses.

A simple and abstract example would be a **restaurant guest and a waiter**. The guest (**Client**) asks (**sends request**) waiter

(**Server**) for a meal, then the waiter gets the meal from the restaurant chef (**your application logic**) and brings the meal to the guest.

This is a very simplistic example, but it is also the one that will help you understand the concept.

There are many more interesting HTTP concepts and utilities to discuss, but the star of this post is (not enough) famous **HTTP/2**.

What is HTTP/2?

In 2015, Internet Engineering Task Force (IETF) release HTTP/2, the second major version of the most useful internet protocol, HTTP. It was derived from the earlier experimental SPDY protocol.

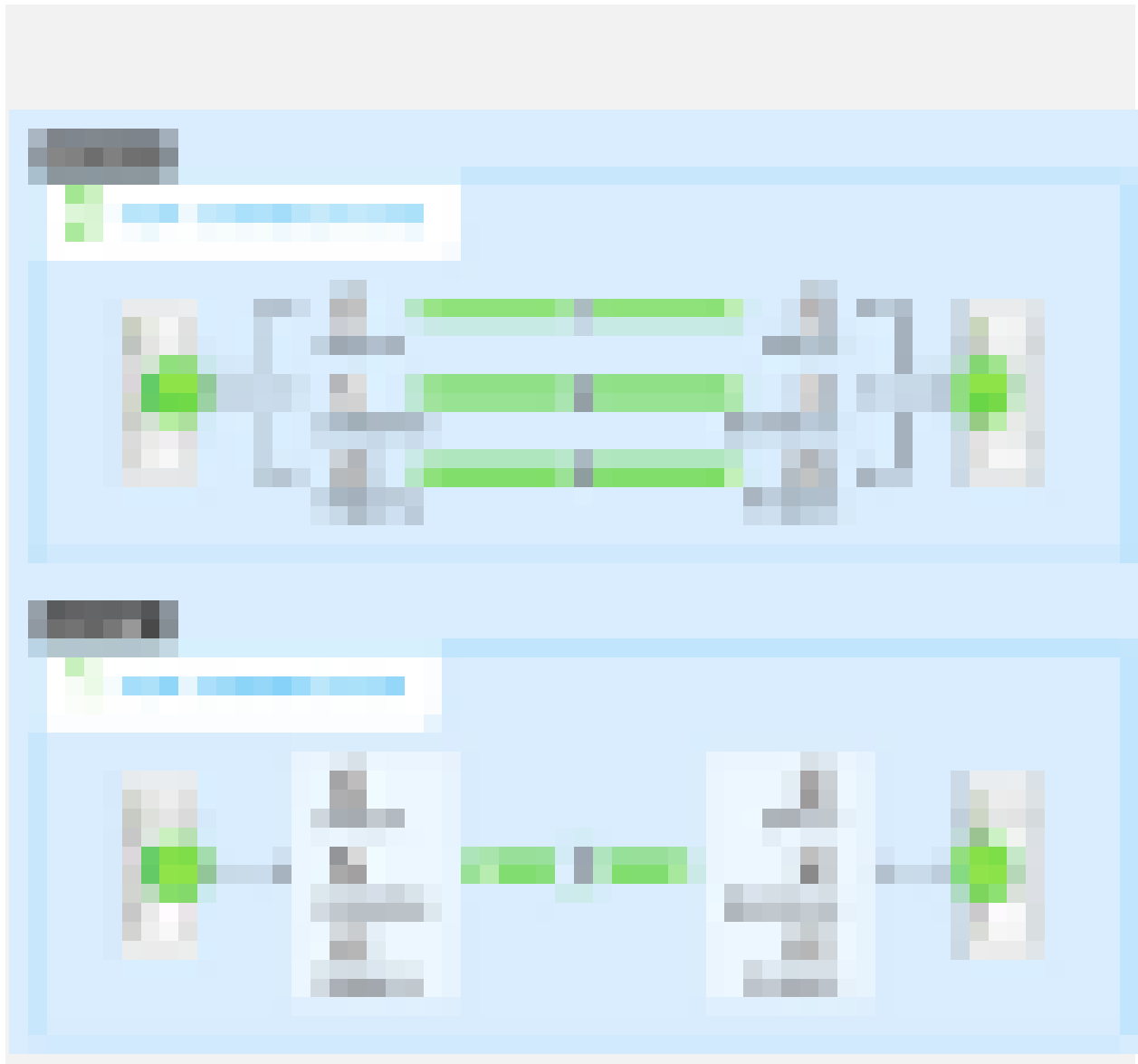
Main goals of developing HTTP/2 was:

- Protocol negotiation mechanism — protocol electing, eg. HTTP/1.1, HTTP/2 or other.
- High-level compatibility with HTTP/1.1 — methods, status codes, URIs and header fields.
- Page load speed improvements through:
 - Compression of request headers
 - Binary protocol
 - HTTP/2 Server Push
 - Request multiplexing over a single TCP connection
 - Request pipelining
 - HOL blocking (Head-of-line) — Package blocking

Request multiplexing

HTTP/2 can send **multiple requests** for data in parallel over a **single** TCP connection. This is **the most advanced feature** of the HTTP/2 protocol because it **allows you to download web**

files asynchronously from one server. Most modern browsers limit TCP connections to one server.



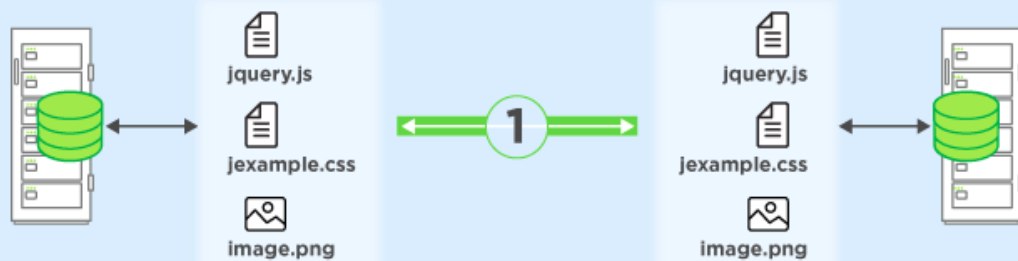
HTTP 1.1

3 TCP CONNECTIONS



HTTP/2

1 TCP CONNECTION

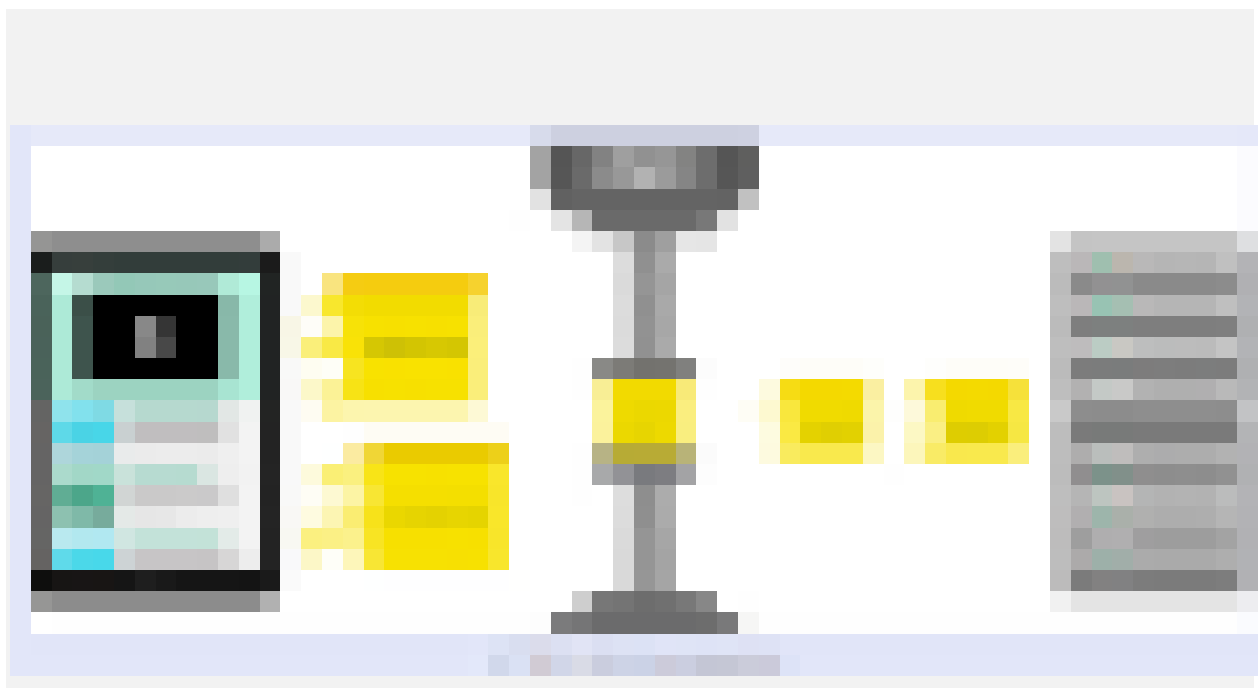


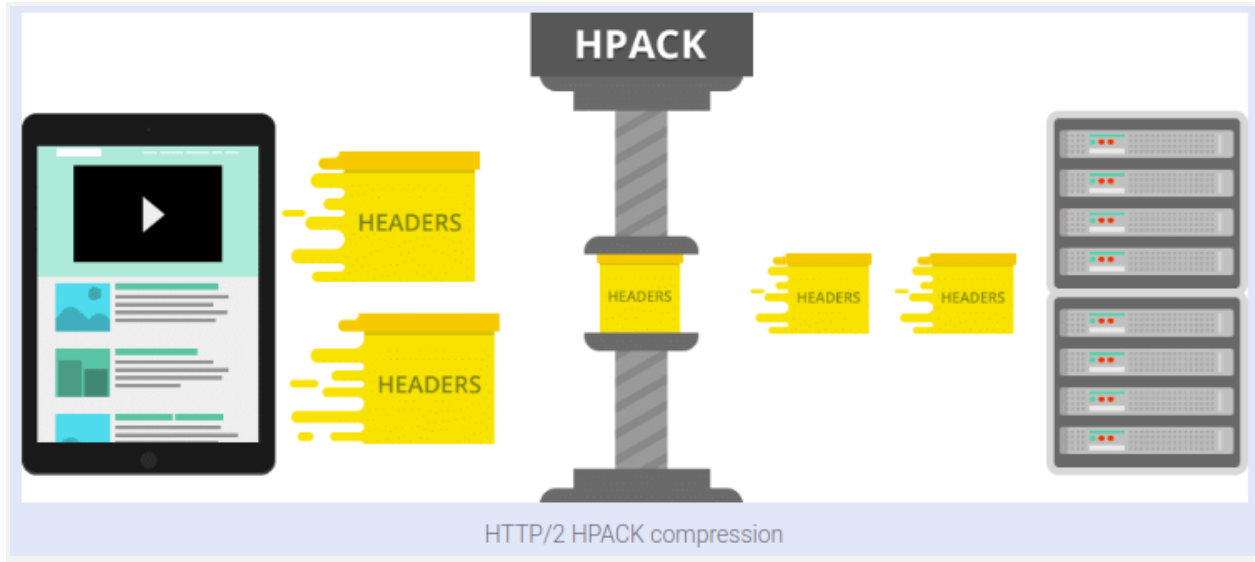
This reduces additional round trip time (RTT), **making your website load faster** without any optimization, and makes domain sharding unnecessary.

Header compression

HTTP/2 compress a large number of redundant header frames. It uses the HPACK specification as a simple and secure approach to header compression. Both client and server maintain a list of headers used in previous client-server requests.

HPACK compresses the individual value of each header before it is transferred to the server, which then looks up the encoded information in a list of previously transferred header values to reconstruct the full header information.

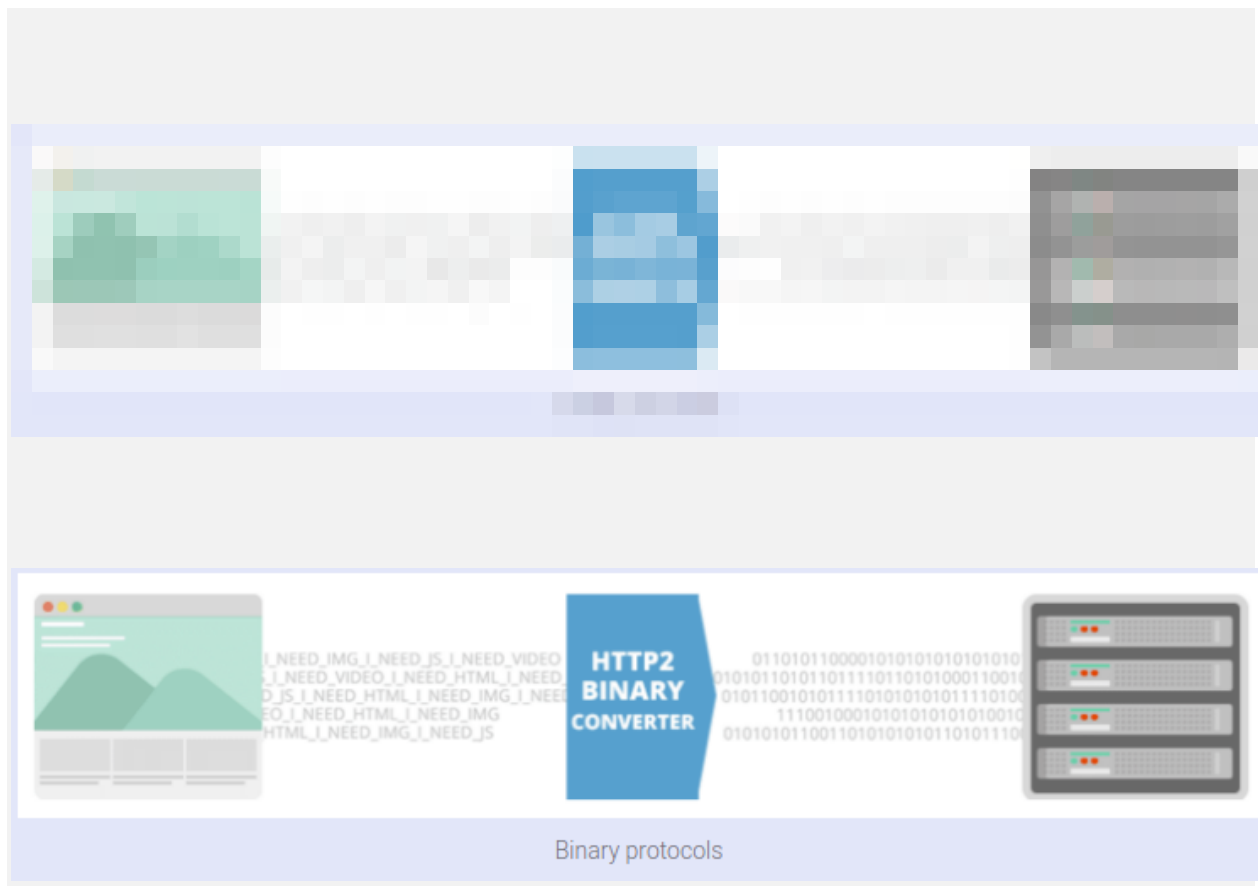




Binary protocol

The latest HTTP version has evolved significantly in terms of capabilities and attributes such as transforming from a text protocol to a binary protocol. HTTP1.x used to process text commands to complete request-response cycles. HTTP/2 will use binary commands (in 1s and 0s) to execute the same tasks. This attribute eases complications with framing and simplifies implementation of commands that were confusingly intermixed due to commands containing text and optional spaces.

Browsers using HTTP/2 implementation will convert the same text commands into binary before transmitting it over the network.



Benefits:

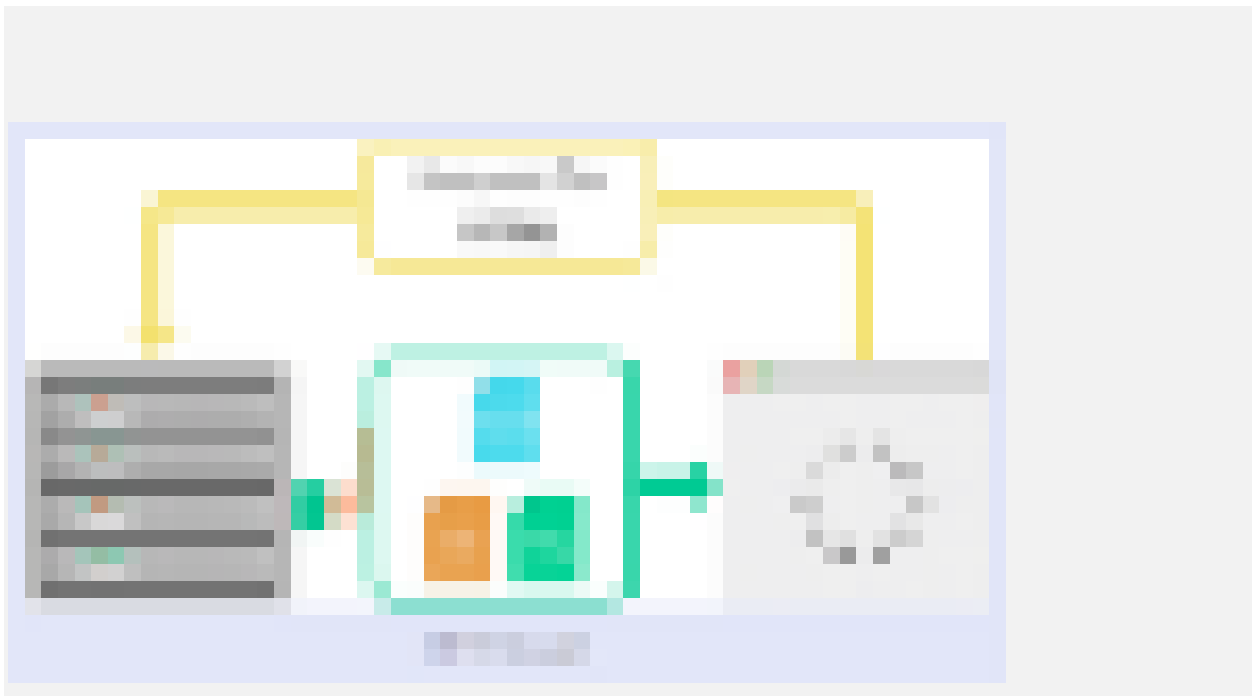
- Low overhead in parsing data — a critical value proposition in HTTP/2 vs HTTP1.
- Less prone to errors.

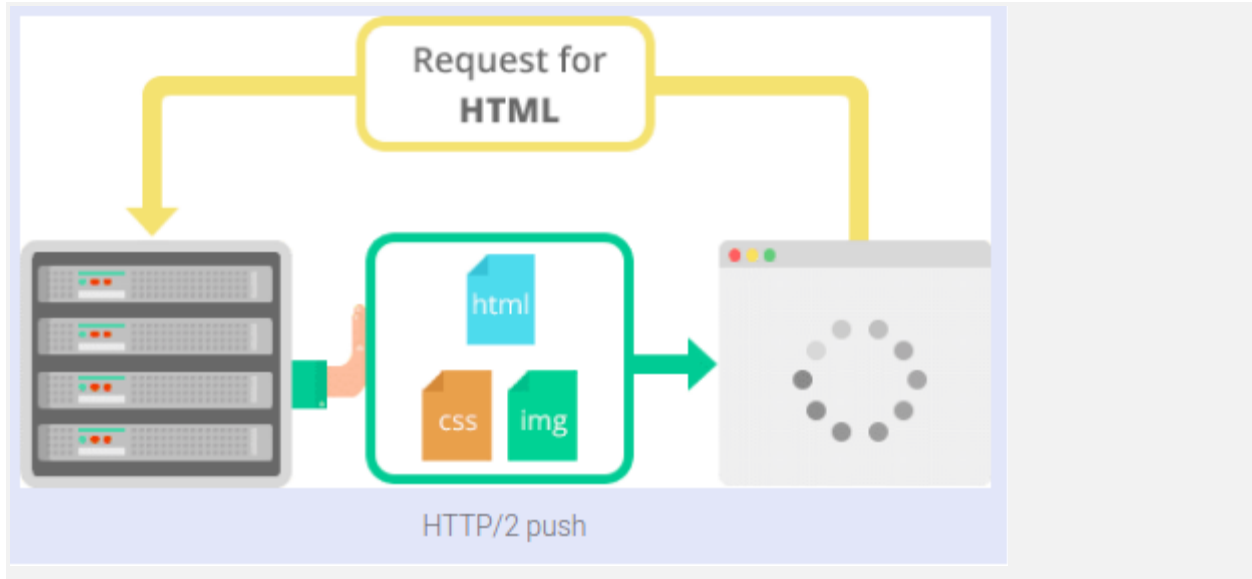
- Lighter network footprint.
- Effective network resource utilization.
- Eliminating security concerns associated with the textual nature of HTTP1.x such as response splitting attacks.
- Enables other capabilities of the HTTP/2 including compression, multiplexing, prioritization, flow control and effective handling of TLS.
- Compact representation of commands for easier processing and implementation.
- Efficient and robust in terms of processing of data between client and server.
- Reduced network latency and improved throughput.

HTTP/2 Server Push

This capability allows the server to send additional cacheable information to the client that isn't requested but is anticipated in

future requests. For example, if the client requests for the resource X and it is understood that the resource Y is referenced with the requested file, the server can choose to push Y along with X instead of waiting for an appropriate client request.





Benefits:

- The client saves pushed resources in the cache.
- The client can reuse these cached resources across different pages.
- The server can multiplex pushed resources along with originally requested information within the same TCP connection.
- The server can prioritize pushed resources — a key performance differentiator in HTTP/2 vs HTTP1.

- The client can decline pushed resources to maintain an effective repository of cached resources or disable Server Push entirely.
- The client can also limit the number of pushed streams multiplexed concurrently.

If you remember the story about a guest in a restaurant and waiter, that would be an example

for HTTP/1.1 and HTTP/2 protocol with a slight difference.

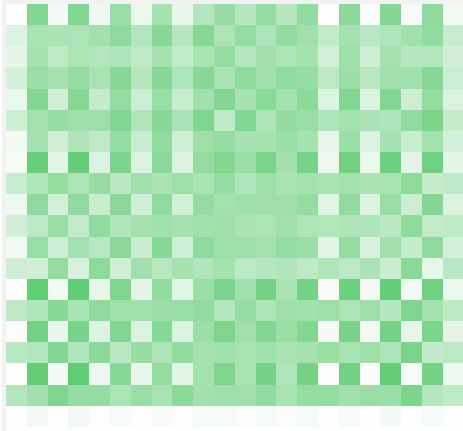
Imagine that waiters are TCP connections and you want to order your meal and a bottle of water. For HTTP/1.1 that would mean that you ask one waiter for your meal and another one for water, hence you would allocate two TCP connections. For HTTP/2 that would mean that you ask only one waiter for both, but he brings them separately. You only allocate one TCP connection and that

will already result with lower server load, plus the server would have one extra free connection (waiter) for the next client (guest).

The real difference between HTTP/1.1 and HTTP/2 comes with server push example.

Imagine that the guest (Client) asks (sends request) waiter (Server) for a meal, then the waiter gets the meal from the restaurant chef (your application logic), but the waiter also thinks you would need a bottle of water so he brings that too with your meal. The end result of this would be only one TCP connection and only one request that will significantly lower the server load.

As a simple showcase of those mechanics, I made a simple page example.



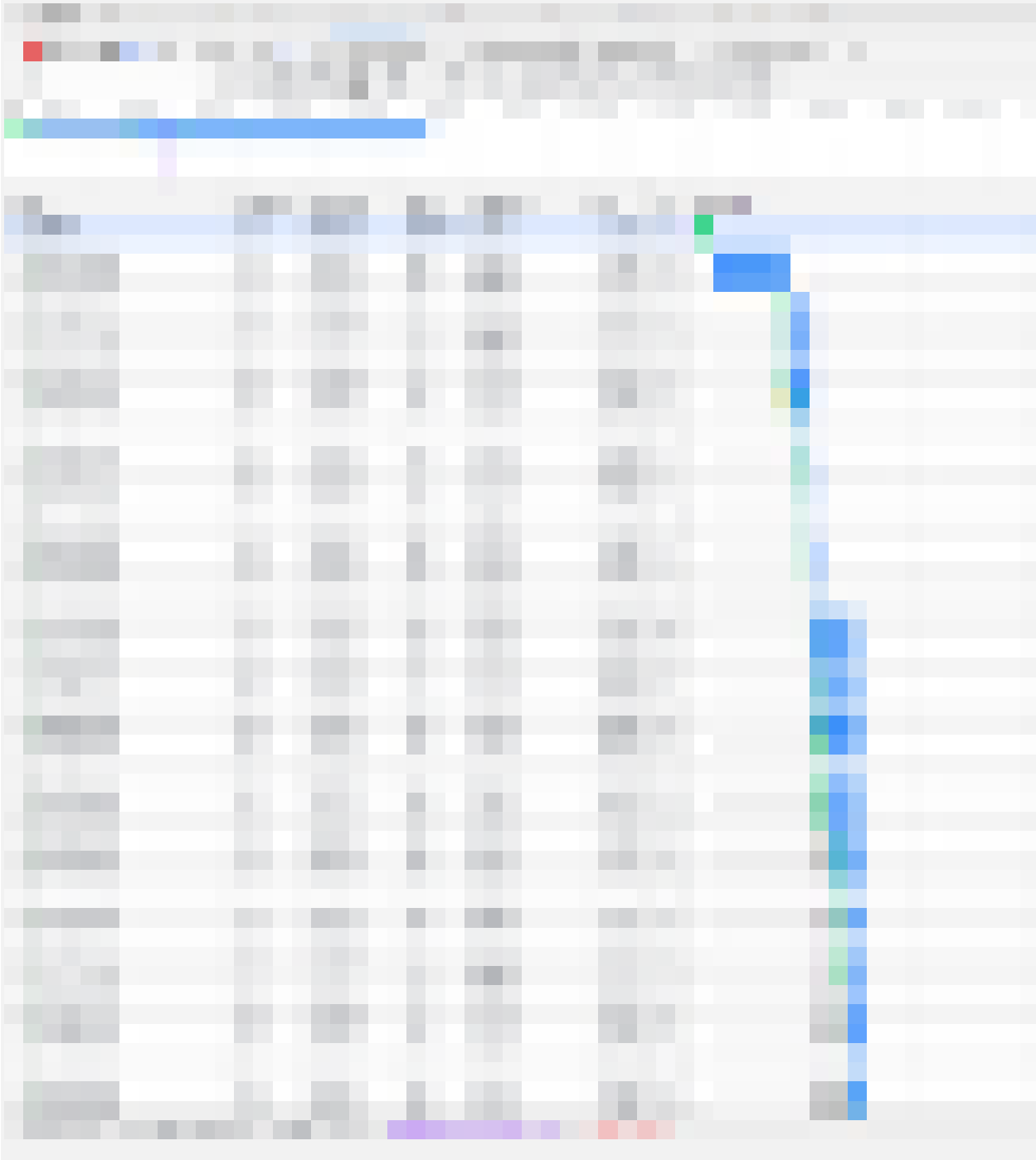


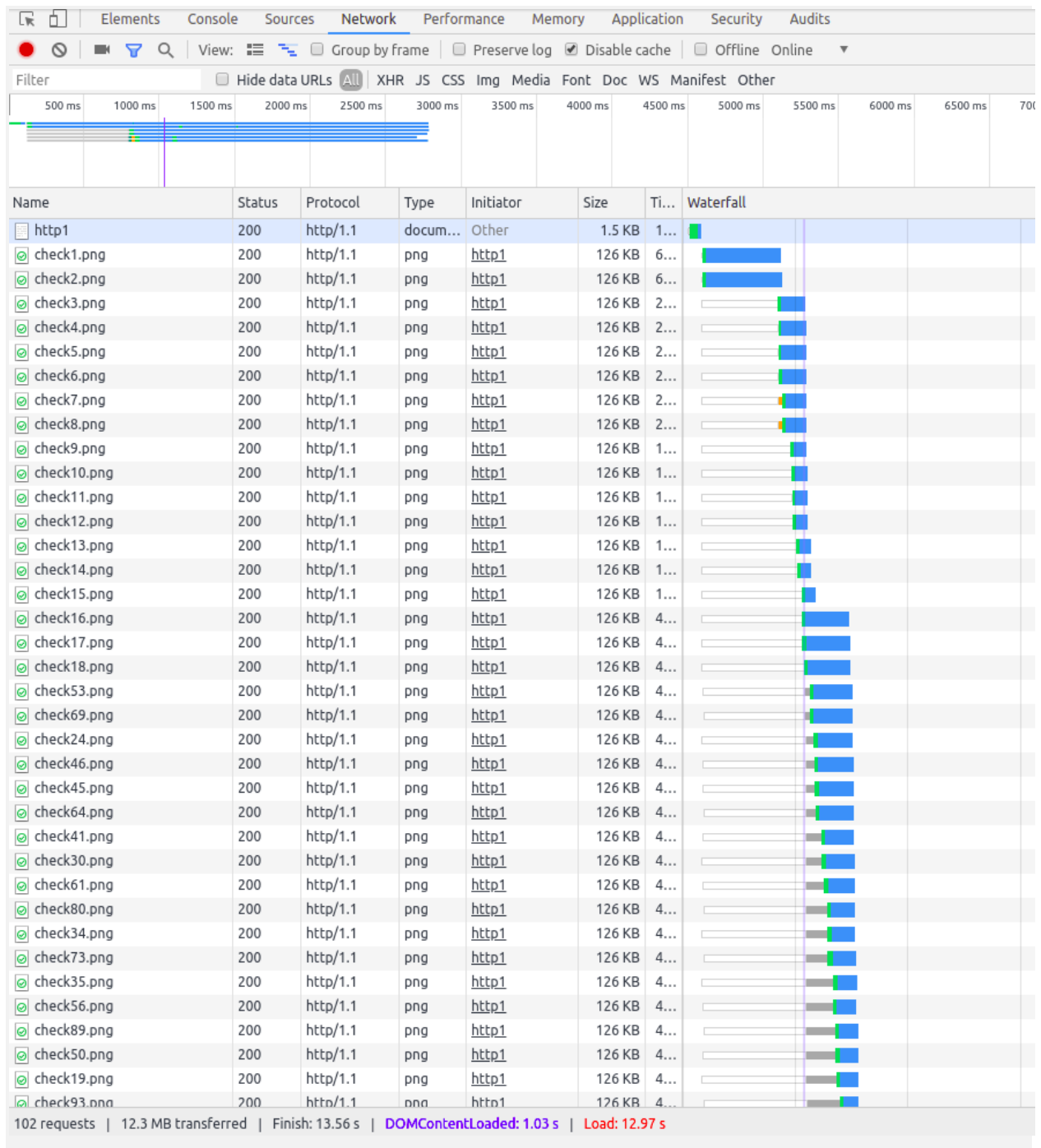
What we have here is a simple page with 100 images of checks which I'll use to demonstrate HTTP/1.1, HTTP/2 AND HTTP/2 server push.

What is important to note in the picture above are number of requests, load time, protocol column, initiator column and waterfall diagram itself (we can see how requests are made

through multiple batches, unfortunately, it is hard to see other data from it except TTFB and content download time; eg. resource scheduling and connection start time).

For HTTP/1.1:





Number of requests: 102

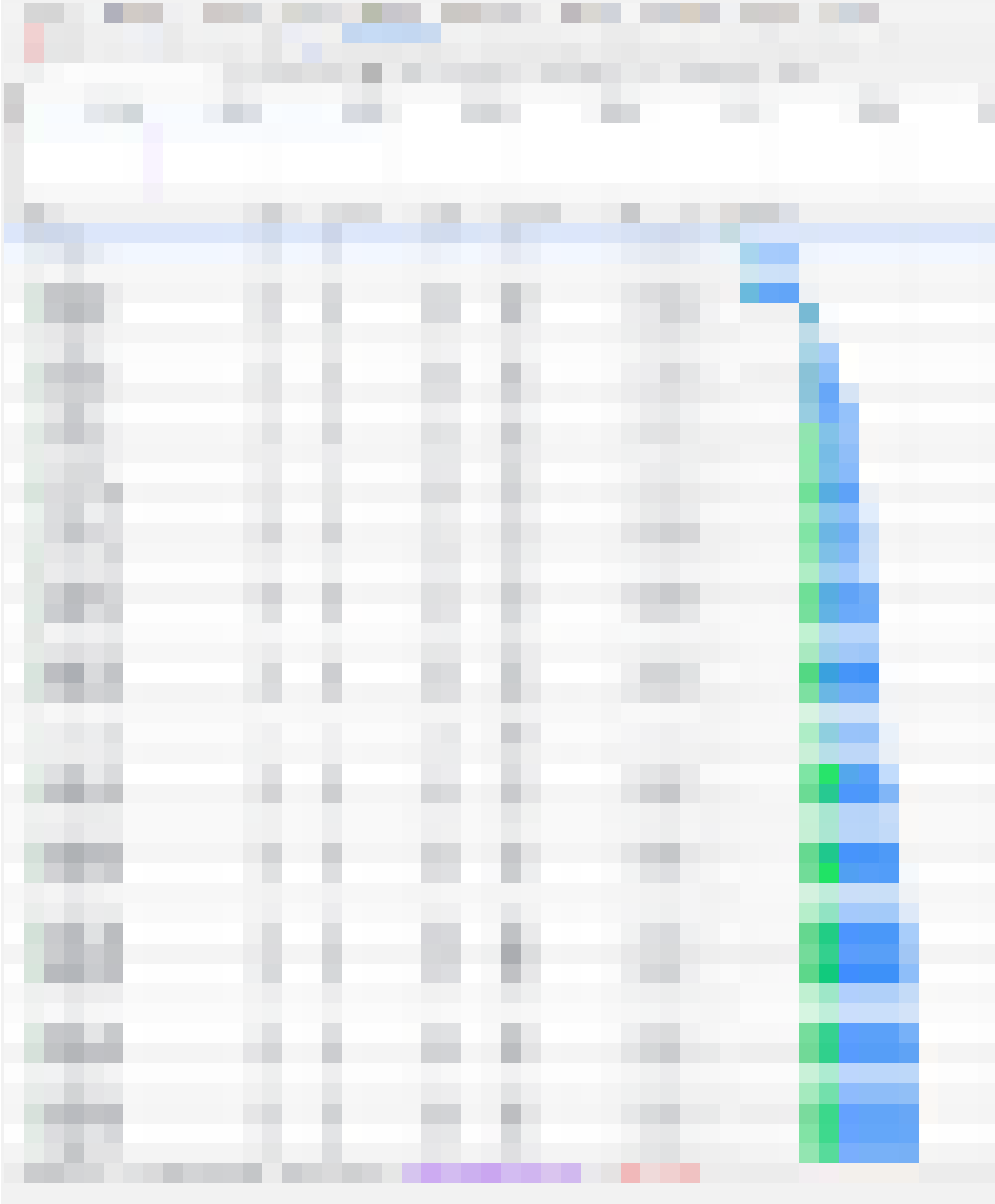
Load time: 12.97s

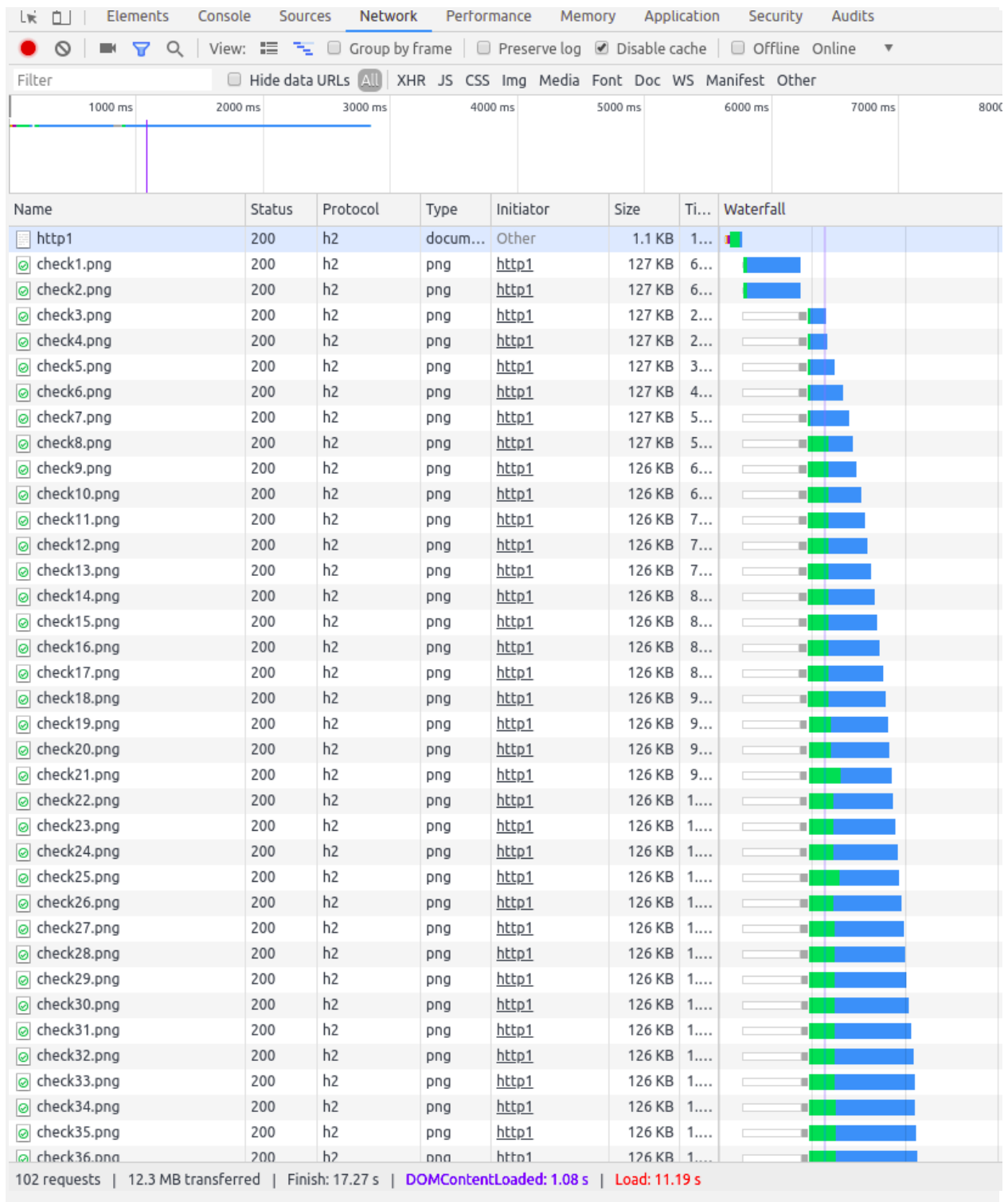
Protocol: “http/1.1”

Initiator column: Initiator of the first one is user/client and the rest of the requests are initiated by the response to client who realizes he needs some other resources (in this case, images).

Waterfall diagram: We can see how requests are made through multiple batches (TCP connections).

For HTTP/2:





Number of requests: 102

Load time: 11.19s

Protocol: “h2” (HTTP/2)

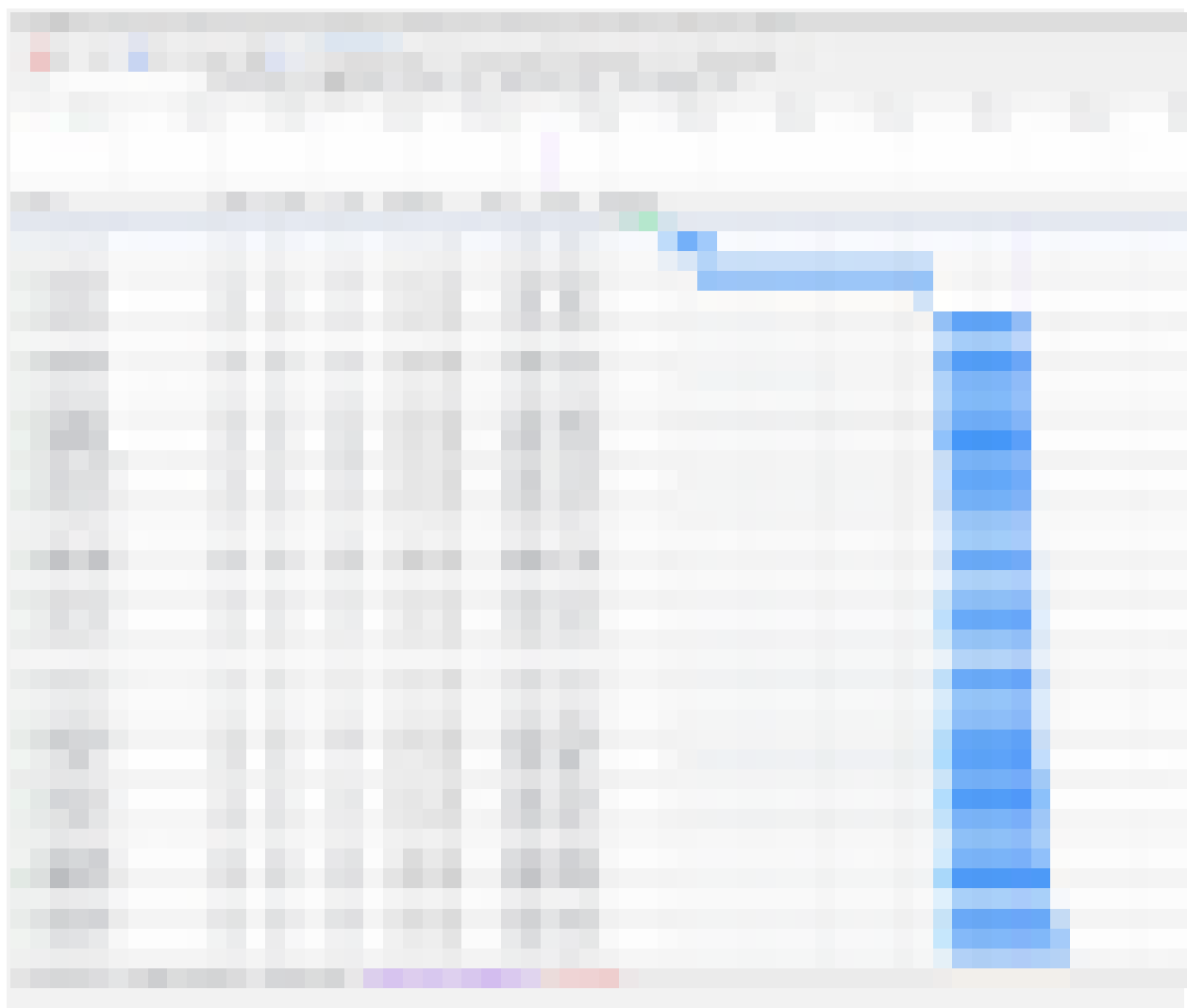
Initiator column: Initiator of the first one is user/client and the rest of the requests are initiated by response to client who realizes he needs some other resources (in this case, images).

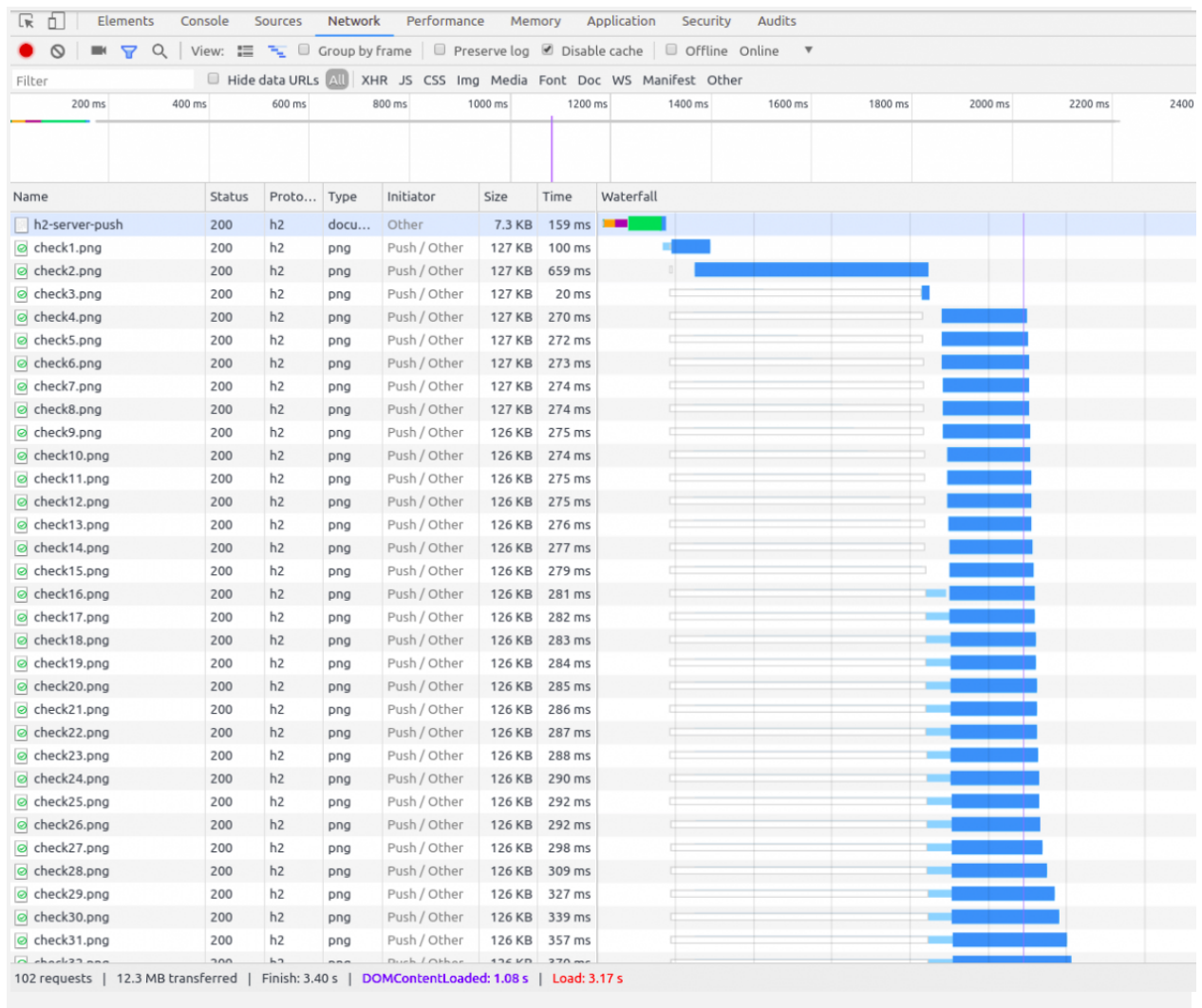
Waterfall diagram: We can see how requests are made through 2 batches (TCP connections).

Take note of the load time. In this case, it is a bit lower than the load time of HTTP/1.1 example but it doesn't have to be always.

This example shows the multiplexing of client requests.

HTTP/2 server push:





Number of requests: 102

Load time: 3.17 s

Protocol: “h2” (HTTP/2)

Initiator column: Initiator of the first one is user/client and the rest of requests are initiated by the push of the server (virtually one request/response cycle).

Waterfall diagram: We can see how requests are made through 1 batch (1 TCP connection).

Browser Compatibility



IE	Edge *	Firefox	Chrome	Safari	iOS Safari *	Opera Mini *	Chrome for Android	UC Browser for Android	Samsung Internet
		49	49		10.3				4
		60	66		11.2				6.2
11	17	61	67	11.1	11.4	all	67	11.8	7.2
	18	62	68	12	12				
		63	69	TP					
			70						

Most of the modern browser fully support HTTP/2 protocol with an exception (red) of Opera mini (all versions) and UC Browser for Android. There are also the ones that have partial support (light green) like IE11.

You can find more details on browser support on this link

<https://caniuse.com/#feat=http2>

Use HTTP/2 and speed up your site

HTTP/2 provides us with **many new mechanics** that will mitigate HTTP/1.1 issues and ones that **will boost your web page performance**. Currently, it is widely supported by web clients so its implementation is painless. Although implementation of HTTP/2 protocol is easy **you should have in mind** that with it you will probably have to change the mechanics

(like serving assets to the client) to use the full potential of this protocol.

Evolution of HTTP

HTTP (HyperText Transfer Protocol) is the underlying protocol of the World Wide Web.

Developed by Tim Berners-Lee and his team between 1989-1991, HTTP has seen many changes, keeping most of the simplicity and further shaping its flexibility. HTTP has evolved from an early protocol to exchange files in a semi-trusted laboratory environment, to the modern maze of the Internet, now carrying images, videos in high resolution and 3D.

Invention of the World Wide Web

In 1989, while he was working at CERN, Tim Berners-Lee wrote a proposal to build a hypertext system over the Internet. Initially calling it the *Mesh*, it was later renamed to *World Wide Web* during its implementation in 1990. Built over the existing TCP and IP protocols, it consisted of 4 building blocks:

- A textual format to represent hypertext documents, the [*HyperText Markup Language*](#) (HTML).
- A simple protocol to exchange these documents, the *HypertText Transfer Protocol* (HTTP).

- A client to display (and accidentally edit) these documents, the first Web browser called *WorldWideWeb*.
- A server to give access to the document, an early version of *httpd*.

These four building blocks were completed by the end of 1990, and the first servers were already running outside of CERN by early 1991. On August 6th 1991, Tim Berners-Lee's [post](#) on the public *alt.hypertext* newsgroup is now considered as the official start of the World Wide Web as a public project.

The HTTP protocol used in those early phases was very simple, later dubbed HTTP/0.9, and sometimes as the one-line protocol.

HTTP/0.9 – The one-line protocol

The initial version of HTTP had no version number; it has been later called 0.9 to differentiate it from the later versions. HTTP/0.9 is extremely simple: requests consist of a single line and start with the only possible method [GET](#) followed by the path to the resource (not the URL as both the protocol, server, and port are unnecessary once connected to the server).

```
GET /mypage.html
```

The response is extremely simple too: it only consisted of the file itself.

```
<HTML>
```

A very simple HTML page

</HTML>

Unlike subsequent evolutions, there were no HTTP headers, meaning that only HTML files could be transmitted, but no other type of documents. There were no status or error codes: in case of a problem, a specific HTML file was send back with the description of the problem contained in it, for human consumption.

HTTP/1.0 – Building extensibility

HTTP/0.9 was very limited and both browsers and servers quickly extended it to be more versatile:

- Versioning information is now sent within each request (HTTP/1.0 is appended to the GET line)
- A status code line is also sent at the beginning of the response, allowing the browser itself to understand the success or failure of the request and to adapt its behavior in consequence (like in updating or using its local cache in a specific way)
- The notion of HTTP headers has been introduced, both for the requests and the responses, allowing metadata to be transmitted and making the protocol extremely flexible and extensible.

- With the help of the new HTTP headers, the ability to transmit other documents than plain HTML files has been added (thanks to the [Content-Type](#) header).

At this point, a typical request and response looked like this:

```
GET /mypage.html HTTP/1.0
```

```
User-Agent: NCSA_Mosaic/2.0 (Windows 3.1)
```

```
200 OK
```

```
Date: Tue, 15 Nov 1994 08:12:31 GMT
```

```
Server: CERN/3.0 libwww/2.17
```

```
Content-Type: text/html
```

```
<HTML>
```

```
A page with an image
```

```
<IMG SRC="/myimage.gif">
```

```
</HTML>
```

Followed by a second connection and request to fetch the image (followed by a response to that request):

GET /myimage.gif HTTP/1.0

User-Agent: NCSA_Mosaic/2.0 (Windows 3.1)

200 OK

Date: Tue, 15 Nov 1994 08:12:32 GMT

Server: CERN/3.0 libwww/2.17

Content-Type: text/gif

(image content)

These novelties have not been introduced as concerted effort, but as a try-and-see approach over the 1991-1995 period: a server and a browser added one feature and it saw if it got traction. A lot of interoperability problems were common. In November 1996, in order to solve these annoyances, an informational document describing the common practices has been published, [RFC 1945](#). This is the definition of HTTP/1.0 and it is notable that, in the narrow sense of the term, it isn't an official standard.

HTTP/1.1 – The standardized protocol

In parallel to the somewhat chaotic use of the diverse implementations of HTTP/1.0, and since 1995, well before the publication of HTTP/1.0 document the next year, proper

standardization was in progress. The first standardized version of HTTP, HTTP/1.1 was published in early 1997, only a few months after HTTP/1.0.

HTTP/1.1 clarified ambiguities and introduced numerous improvements:

- A connection can be reused, saving the time to reopen it numerous times to display the resources embedded into the single original document retrieved.
- Pipelining has been added, allowing to send a second request before the answer for the first one is fully transmitted, lowering the latency of the communication.
- Chunked responses are now also supported.
- Additional cache control mechanisms have been introduced.
- Content negotiation, including language, encoding, or type, has been introduced, and allows a client and a server to agree on the most adequate content to exchange.
- Thanks to the [Host](#) header, the ability to host different domains at the same IP address now allows server colocation.

A typical flow of requests, all through one single connection is now looking like this:

```
GET /en-US/docs/Glossary/Simple_header HTTP/1.1
```

```
Host: developer.mozilla.org
```


User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101
Firefox/50.0

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

Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate, br

Referer: https://developer.mozilla.org/en-US/docs/Glossary/Simple_header

200 OK

Connection: Keep-Alive

Content-Encoding: gzip

Content-Type: text/html; charset=utf-8

Date: Wed, 20 Jul 2016 10:55:30 GMT

Etag: "547fa7e369ef56031dd3bff2ace9fc0832eb251a"

Keep-Alive: timeout=5, max=1000

Last-Modified: Tue, 19 Jul 2016 00:59:33 GMT

Server: Apache

Transfer-Encoding: chunked

Vary: Cookie, Accept-Encoding

(content)

GET /static/img/header-background.png HTTP/1.1

Host: developer.mozilla.org

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101

Firefox/50.0

Accept: */*

Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate, br

Referer: https://developer.mozilla.org/en-US/docs/Glossary/Simple_header

200 OK

Age: 9578461

Cache-Control: public, max-age=315360000

Connection: keep-alive

Content-Length: 3077

Content-Type: image/png

Date: Thu, 31 Mar 2016 13:34:46 GMT

Last-Modified: Wed, 21 Oct 2015 18:27:50 GMT

Server: Apache

(image content of 3077 bytes)

HTTP/1.1 was first published as [RFC 2068](#) in January 1997.

More than 15 years of extensions

Thanks to its extensibility – creating new headers or methods is easy – and even if the

HTTP/1.1 protocol was refined over two revisions, [RFC 2616](#) published in June 1999

and the series of [RFC 7230-RFC 7235](#) published in June 2014 in prevision of the release of HTTP/2, this protocol has been extremely stable over more than 15 years.

Using HTTP for secure transmissions

The largest change that happened to HTTP was done as early as end of 1994. Instead of sending HTTP over a basic TCP/IP stack, Netscape Communications created an additional encrypted transmission layer on top of it: SSL. SSL 1.0 was never released outside the company, but SSL 2.0 and its successor SSL 3.0 allowed for the creation of e-commerce Web sites by encrypting and guaranteeing the authenticity of the messages exchanged between the server and client. SSL was put on the standards track and eventually became TLS, with versions 1.0, 1.1, 1.2, and 1.3 appearing successfully to close vulnerabilities.

During the same time, the need for an encrypted transport layer raised: the Web left the relative trustiness of a mostly academic network, to a jungle where advertisers, random individuals or criminals compete to get as much private information about people, try to impersonate them or even to replace data transmitted by altered ones. As the applications built over HTTP became more and more powerful, having access to more and more private information like address books, e-mail, or the geographic position of the user, the need to have TLS became ubiquitous even outside the e-commerce use case.

Using HTTP for complex applications

The original vision of Tim Berners-Lee for the Web wasn't a read-only medium. He envisioned a Web where people can add and move documents remotely, a kind of distributed file system. Around 1996, HTTP has been extended to allow authoring, and a standard called WebDAV was created. It has been further extended for specific applications like CardDAV to handle address book entries and CalDAV to deal with calendars. But all these *DAV extensions had a flaw: they had to be implemented by the servers to be used, which was quite complex. Their use on Web realms stayed confidential.

In 2000, a new pattern for using HTTP was designed: [representational state transfer](#) (or REST). The actions induced by the API were no more conveyed by new HTTP methods, but only by accessing specific URIs with basic HTTP/1.1 methods. This allowed any Web application to provide an API to allow retrieval and modification of its data without having to update the browsers or the servers: all what is needed was embedded in the files served by the Web sites through standard HTTP/1.1. The drawback of the REST model resides in the fact that each website defines its own non-standard RESTful API and has total control on it; unlike the *DAV extensions where clients and servers are interoperable. RESTful APIs became very common in the 2010s.

Since 2005, the set of APIs available to Web pages greatly increased and several of these APIs created extensions, mostly new specific HTTP headers, to the HTTP protocol for specific purposes:

- [Server-sent events](#), where the server can push occasional messages to the browser.
- [WebSocket](#), a new protocol that can be set up by upgrading an existing HTTP connection.

Relaxing the security-model of the Web

HTTP is independent of the security model of the Web, the [same-origin policy](#). In fact, the current Web security model has been developed after the creation of HTTP! Over the years, it has proved useful to be able to be more lenient, by allowing under certain constraints to lift some of the restriction of this policy. How much and when such restrictions are lifted is transmitted by the server to the client using a new bunch of HTTP headers. These are defined in specifications like [Cross-Origin Resource Sharing](#) (CORS) or the [Content Security Policy](#) (CSP).

In addition to these large extensions, numerous other headers have been added, sometimes experimentally only. Notable headers are Do Not Track ([DNT](#)) header to control privacy, [X-Frame-Options](#), or [Upgrade-Insecure-Requests](#) but many more exist.

HTTP/2 – A protocol for greater performance

Over the years, Web pages have become much more complex, even becoming applications in their own right. The amount of visual media displayed, the volume and

size of scripts adding interactivity, has also increased: much more data is transmitted over significantly more HTTP requests. HTTP/1.1 connections need requests sent in the correct order. Theoretically, several parallel connections could be used (typically between 5 and 8), bringing considerable overhead and complexity. For example, HTTP pipelining has emerged as a resource burden in Web development.

In the first half of the 2010s, Google demonstrated an alternative way of exchanging data between client and server, by implementing an experimental protocol SPDY. This amassed interest from developers working on both browsers and servers. Defining an increase in responsiveness, and solving the problem of duplication of data transmitted, SPDY served as the foundations of the HTTP/2 protocol.

The HTTP/2 protocol has several prime differences from the HTTP/1.1 version:

- It is a binary protocol rather than text. It can no longer be read and created manually. Despite this hurdle, improved optimization techniques can now be implemented.
- It is a multiplexed protocol. Parallel requests can be handled over the same connection, removing the order and blocking constraints of the HTTP/1.x protocol.
- It compresses headers. As these are often similar among a set of requests, this removes duplication and overhead of data transmitted.

- It allows a server to populate data in a client cache, in advance of it being required, through a mechanism called the server push.

Officially standardized, in May 2015, HTTP/2 has had much success. By July 2016, 8.7% of all Web sites^[1] were already using it, representing more than 68% of all requests^[2]. High-traffic Web sites showed the most rapid adoption, saving considerably on data transfer overheads and subsequent budgets.

This rapid adoption rate was likely as HTTP/2 does not require adaptation of Web sites and applications: using HTTP/1.1 or HTTP/2 is transparent for them. Having an up-to-date server communicating with a recent browser is enough to enable its use: only a limited set of groups were needed to trigger adoption, and as legacy browser and server versions are renewed, usage has naturally increased, without further Web developer efforts.

Post-HTTP/2 evolution

HTTP didn't stop evolving upon the release of HTTP/2. Like with HTTP/1.x previously, HTTP's extensibility is still being used to add new features. Notably, we can cite new extensions of the HTTP protocol appearing in 2016:

- Support of [Alt-Svc](#) allows the dissociation of the identification and the location of a given resource, allowing for a smarter [CDN](#) caching mechanism.

- The introduction of [Client-Hints](#) allows the browser, or client, to proactively communicate information about its requirements, or hardware constraints, to the server.
- The introduction of security-related prefixes in the [Cookie](#) header, now helps guarantee a secure cookie has not been altered.

This evolution of HTTP proves its extensibility and simplicity, liberating creation of many applications and compelling the adoption of the protocol. The environment in which HTTP is used today is quite different from that seen in the early 1990s. HTTP's original design proved to be a masterpiece, allowing the Web to evolve over a quarter of a century, without the need of a mutiny. By healing flaws, yet retaining the flexibility and extensibility which made HTTP such a success, the adoption of HTTP/2 hints at a bright future for the protocol.

Difference between Nodejs and JavaScript :

S.N

0

Javascript

NodeJS

- | | | |
|----|---|---|
| 1. | Javascript is a programming language that is used for writing scripts on the website. | NodeJS is a Javascript runtime environment. |
| 2. | Javascript can only be run in the browsers. | NodeJS code can be run outside the browser. |
| 3. | It is basically used on the client-side. | It is mostly used on the server-side. |
| 4. | Javascript is capable enough to add HTML and play with the DOM. | Nodejs does not have capability to add HTML tags. |

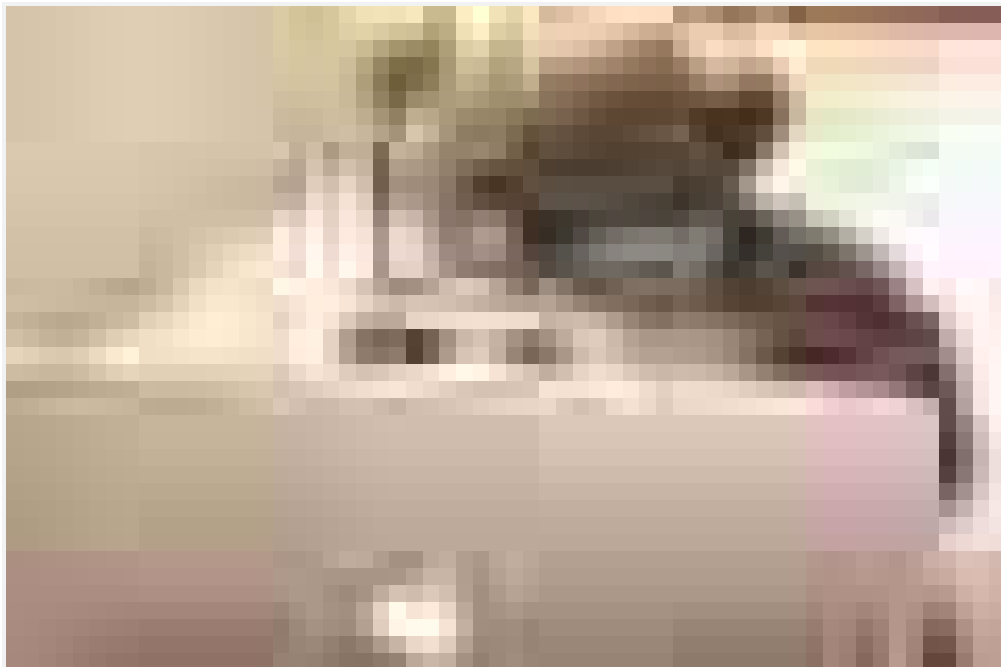
5 Javascript can run in any browser engine as
like JS core in safari and Spidermonkey in
Firefox. Nodejs can only run in
V8 engine of google
chrome.

What happens when you type a URL in the browser and press enter?



Maneesha Wijesinghe

Apr 26, 2017 · 7 min read



If you are in any technical profession, I am sure someone at some point has asked you this question. Whether you are an engineer, developer, marketer, or even in sales, it is always good to have a basic understanding of what is going on behind our browsers and how information is transferred to our computers via the internet.

Let's imagine that you want to access maps.google.com to check the exact time it would take for you to get to your dinner reservation from work.

1. You type maps.google.com into the address bar of your browser.
2. The browser checks the cache for a DNS record to find the corresponding IP address of maps.google.com.

DNS(Domain Name System) is a database that maintains the name of the website (URL) and the particular IP address it links to. Every single URL on the internet has a unique IP address assigned to it. The IP address belongs to the computer which hosts the server of the website we are requesting to access. For example, www.google.com has an IP address of 209.85.227.104. So if you'd like, you can reach www.google.com by typing <http://209.85.227.104> on your browser. DNS is a list of URLs, and their IP addresses, like how a phone book is a list of names and their corresponding phone numbers.

The primary purpose of DNS is human-friendly navigation. You can easily access a website by typing the correct IP address for it on your browser, but imagine having to remember different sets of numbers for all the sites we regularly access? Therefore, it is easier to remember the name of the website using a URL and let DNS do the work for us by mapping it to the correct IP.

To find the DNS record, the browser checks four caches.

- First, it checks the browser cache. The browser maintains a repository of DNS records for a fixed duration for websites you have previously visited. So, it is the first place to run a DNS query.
- Second, the browser checks the OS cache. If it is not in the browser cache, the browser will make a system call (i.e., *gethostname* on Windows) to your underlying computer OS to fetch the record since the OS also maintains a cache of DNS records.
- Third, it checks the router cache. If it's not on your computer, the browser will communicate with the router that maintains its' own cache of DNS records.

- Fourth, it checks the ISP cache. If all steps fail, the browser will move on to the ISP. Your ISP maintains its' own DNS server, which includes a cache of DNS records, which the browser would check with the last hope of finding your requested URL.

You may wonder why there are so many caches maintained at so many levels. Although our information being cached somewhere doesn't make us feel very comfortable when it comes to privacy, caches are essential for regulating network traffic and improving data transfer times.

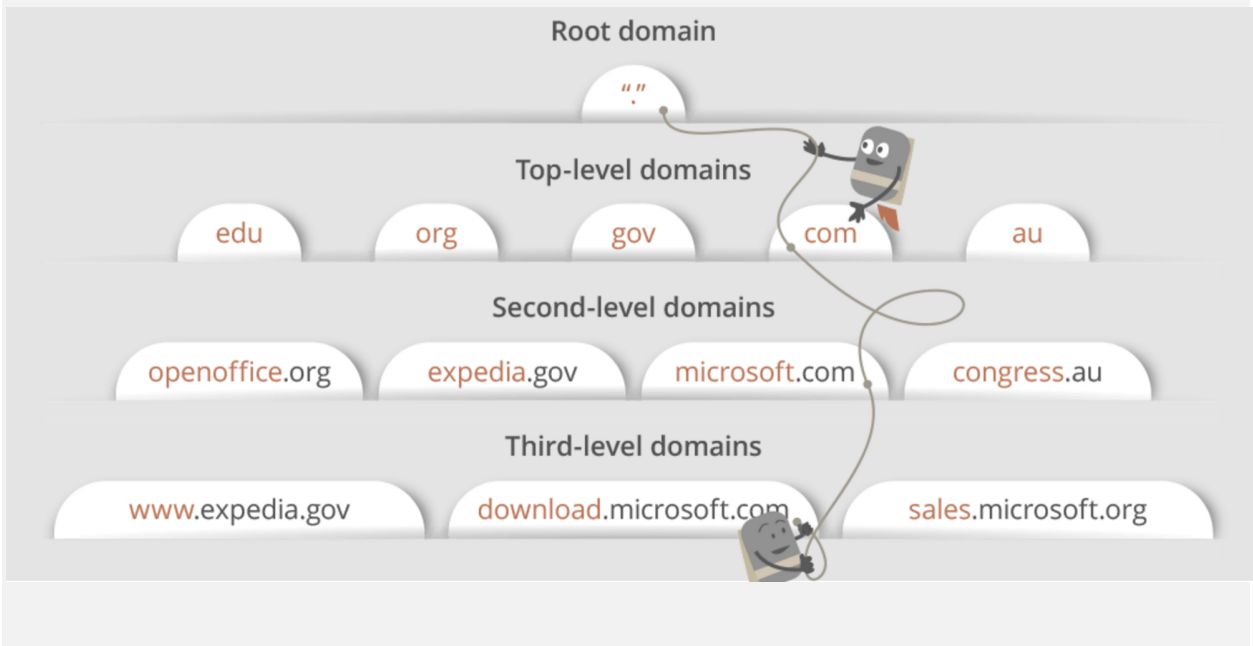
3. If the requested URL is not in the cache, ISP's DNS server initiates a DNS query to find the IP address of the server that hosts `maps.google.com`.

As mentioned earlier, for my computer to connect with the server that hosts `maps.google.com`, I need the IP address of

maps.google.com. The purpose of a DNS query is to search multiple DNS servers on the internet until it finds the correct IP address for the website. This type of search is called a recursive search since the search will repeatedly continue from a DNS server to a DNS server until it either finds the IP address we need or returns an error response saying it was unable to find it.

In this situation, we would call the ISP's DNS server a DNS recursor whose responsibility is to find the proper IP address of the intended domain name by asking other DNS servers on the internet for an answer. The other DNS servers are called name servers since they perform a DNS search based on the domain architecture of the website domain name.

Without further confusing you, I'd like to use the following diagram to explain the domain architecture.



<https://webhostinggeeks.com/guides/dns/>

Many website URLs we encounter today contain a third-level domain, a second-level domain, and a top-level domain. Each of these levels contains their own name server, which is queried during the DNS lookup process.

For `maps.google.com`, first, the DNS recursor will contact the root name server. The root name server will redirect it to the **.com** domain name server. **.com** name server will redirect it to the **google.com** name server. The **google.com** name server will find the matching IP address for `maps.google.com` in its' DNS records and return it to your DNS recursor, which will send it back to your browser.

These requests are sent using small data packets that contain information such as the content of the request and the IP address it is destined for (IP address of the DNS recursor). These packets travel through multiple networking equipment between the client

and the server before it reaches the correct DNS server. This equipment use routing tables to figure out which way is the fastest possible way for the packet to reach its' destination. If these packets get lost, you'll get a request failed error. Otherwise, they will reach the correct DNS server, grab the correct IP address, and come back to your browser.

4. The browser initiates a TCP connection with the server.

Once the browser receives the correct IP address, it will build a connection with the server that matches the IP address to transfer information. Browsers use internet protocols to build such connections. There are several different internet protocols that can be used, but TCP is the most common protocol used for many types of HTTP requests.

To transfer data packets between your computer(client) and the server, it is important to have a TCP connection established. This connection is established using a process called the TCP/IP three-way handshake. This is a three-step process where the client and the server exchange SYN(synchronize) and ACK(acknowledge) messages to establish a connection.

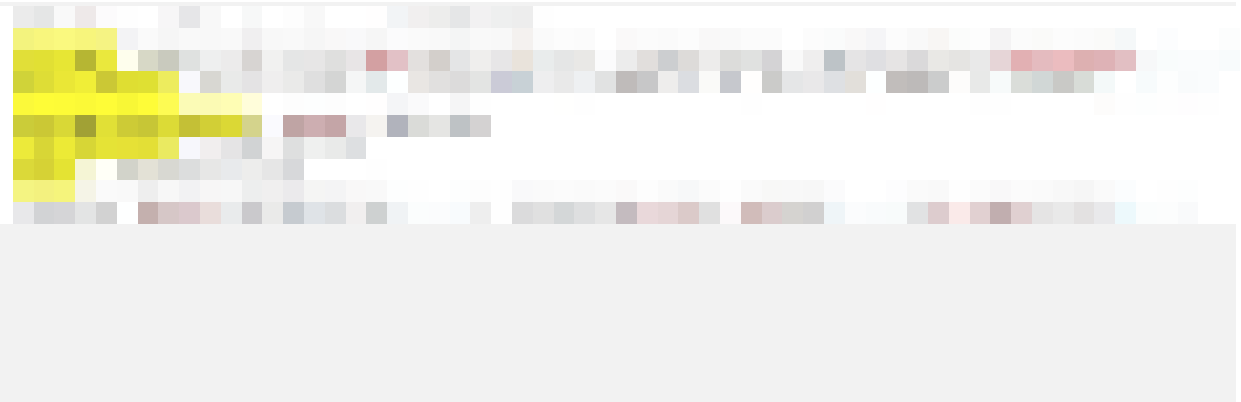
1. The client machine sends a SYN packet to the server over the internet, asking if it is open for new connections.
2. If the server has open ports that can accept and initiate new connections, it'll respond with an ACKnowledgment of the SYN packet using a SYN/ACK packet.
3. The client will receive the SYN/ACK packet from the server and will acknowledge it by sending an ACK packet.

Then a TCP connection is established for data transmission!

5. The browser sends an HTTP request to the webserver.

Once the TCP connection is established, it is time to start transferring data! The browser will send a GET request asking for maps.google.com web page. If you're entering credentials or submitting a form, this could be a POST request. This request will also contain additional information such as browser identification (*User-Agent* header), types of requests that it will accept (*Accept* header), and connection headers asking it to keep the TCP connection alive for additional requests. It will also pass information taken from cookies the browser has in store for this domain.

Sample GET request (Headers are highlighted):



```
GET http://facebook.com/ HTTP/1.1
Accept: application/x-ms-application, image/jpeg, application/xaml+xml, [...]
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; [...])
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
Host: facebook.com
Cookie: datr=1265876274-[...]; locale=en_US; lsd=WW[...]; c_user=2101[...]
```

(If you're curious about what's going on behind the scenes, you can use tools such as Firebug to take a look at HTTP requests. It is always fun to see the information passed between clients and servers without us knowing).

6. The server handles the request and sends back a response.

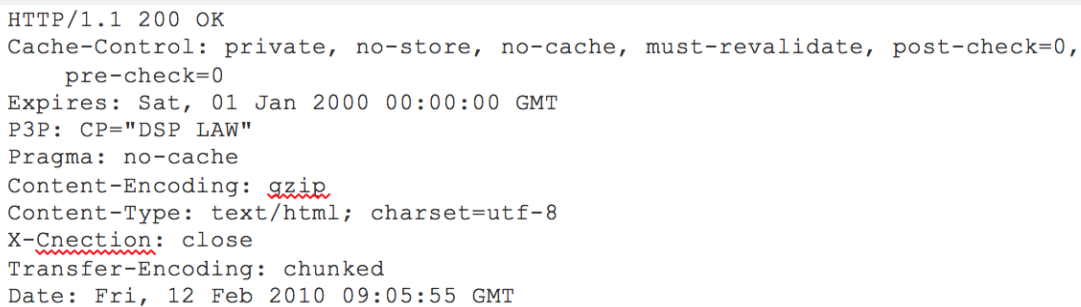
The server contains a webserver (i.e., Apache, IIS) that receives the request from the browser and passes it to a request handler to

read and generate a response. The request handler is a program (written in ASP.NET, PHP, Ruby, etc.) that reads the request, its' headers, and cookies to check what is being requested and also update the information on the server if needed. Then it will assemble a response in a particular format (JSON, XML, HTML).

7. The server sends out an HTTP response.

The server response contains the web page you requested as well as the status code, compression type (*Content-Encoding*), how to cache the page (*Cache-Control*), any cookies to set, privacy information, etc.

Example HTTP server response:



```
HTTP/1.1 200 OK
Cache-Control: private, no-store, no-cache, must-revalidate, post-check=0,
pre-check=0
Expires: Sat, 01 Jan 2000 00:00:00 GMT
P3P: CP="DSP LAW"
Pragma: no-cache
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
X-Cnection: close
Transfer-Encoding: chunked
Date: Fri, 12 Feb 2010 09:05:55 GMT
```

If you look at the above response, the first line shows a status code. This is quite important as it tells us the status of the response. There are five types of statuses detailed using a numerical code.

- 1xx indicates an informational message only
- 2xx indicates success of some kind

- 3xx redirects the client to another URL
- 4xx indicates an error on the client's part
- 5xx indicates an error on the server's part

So, if you encountered an error, you can take a look at the HTTP response to check what type of status code you have received.

8. The browser displays the HTML content (for HTML responses, which is the most common).

The browser displays the HTML content in phases. First, it will render the bare bone HTML skeleton. Then it will check the HTML tags and send out GET requests for additional elements on the web page, such as images, CSS stylesheets, JavaScript files, etc. These static files are cached by the browser, so it doesn't have

to fetch them again the next time you visit the page. In the end, you'll see `maps.google.com` appearing on your browser.

That's it!