Difference between

HTTP1.1 Vs HTTP2

**I**

n 1989, Tim Berners-Lee invented HTTP. HTTP1.1 was its 1st standardised version that was available for use in the year 1997 for the end-users. This version presented considerable performance optimisation over its precursors and changed how communication was handled between clients and servers. However, its key qualities opened the doors to many performance and API security loopholes.

HTTP1.1 was known to have poor response time. With websites becoming more resource-intensive, the protocol was losing its efficiency. It progressively became essential to minimise latency and boost page load speeds.

Google looked into these problems. And as expected, SPDY - an experimental project to end troubles with HTTP/1.x – was put into trial in the year 2010.

Years later, the IETF, Google, Microsoft, and Facebook released the fully-comprehensive and well-tested newer version of HTTP in 2015.

HTTP2, based on SPDY protocol, was developed to address the inherent limitations of HTTP1.1 and further progress the Internet.Image

Main Differences

# **1.The Background**

For better contextualisation of the certain alterations that HTTP2 made to its precursor, we’ll take a quick look at their basic functionalities and development details first.

**HTTP/1.1**

HTTP protocol was developed in 1989 as the common language that enables client and server machines’ interaction. Process steps are as enlisted:

* The client (browser) has to send a request to the server using the method (GET/POST).
* Server responds with the requested resource, for example – image, alongside the status of what it did to the client’s request.

This is not a one-time process. Such requests and responses needs to be transferred between both these machines until the client receives all the resources, essential to load a web page on the end-user’s (your) screen.

This request-response exchange can be regarded as an IP stack being handled by transfer layer and networking layers before finally reaching to the application layer.

**HTTP/2**

HTTP/2 was released at Google as the significant improvement of its predecessor. It was initially modelled after the SPDY protocol and went through significant changes to include features like multiplexing, header compression, and stream prioritisation to minimise page load latency. After its release, Google announced that it would not provide support for SPDY in favour of HTTP/2.

The major feature that differentiates HTTP/2 from HTTP/1.1 is the binary framing layer. Unlike HTTP/1.1, HTTP/2 uses a binary framing layer. This layer encapsulates messages – converted to its binary equivalent – while making sure that its HTTP semantics (method details, header information, etc.) remain untamed. This feature of HTTP/2 enables gRPC to use lesser resources.

# **2. Delivery Models**

HTTP/1.1 sends messages as plain text, and HTTP/2 encodes them into binary data and arranges them carefully. This implies that HTTP/2 can have various delivery models.

Most of the time, a client's initial response in return for an HTTP GET request is not the fully-loaded page. Fetching additional resources from the server requires that the client send repeated requests, break or form the TCP connection repeatedly for them.

As you can conclude already, this process will consume lots of resources and time.

**HTTP/1.1**

HTTP/1.1 addresses this problem by creating a persistent connection between server and client. Until explicitly closed, this connection will remain open. So, the client can use one TCP connection throughout the communication sans interrupting it again and again.

This approach surely ensures good performance, but it also is problematic.

For example – If a request at the queue head cannot retrieve its required resources, it can block all requests behind it. This phenomenon is called head-of-line blocking (HOL blocking).

From the above, we can conclude that multiple TCP connections are essential.

**HTTP/2**

Considering the bottleneck in the previous scenario, the HTTP/2 developers introduced a binary framing layer. This layer partitions requests and responses in tiny data packets and encodes them. Due to this, multiple requests and responses become able to run parallelly with HTTP/2 and chances of HOL blocking are bleak.

Not only has it solved the HOL blocking problem in HTTP/1.1, but it also concurrent message exchange between the client and the server. This way, both of them can have more control while the connection management quality is boosted too.

The problems of HTTP/1.1 looks resolved to a great extent here. However, at times, multiple data streams demanding the same resource can hinder HTTP/2’s performance. To achieve better performance, HTTP/2 has another way. It has the capability of stream prioritisation.

When sending streams in parallel, the client can assign weights (1-256) to its stream to prioritise the responses it demands. Here, the higher the weight, the higher the priority. The serve sets the data retrieval order as per the request’s weight. Programmers can enjoy better control on page rendering process with stream prioritisation ability.

Image

# **3. Predicting Resource Requests**

The client receives an HTML page on sending a GET request. While examining the page contents, the client determines that it needs additional resources for rendering the page and makes further requests to fetch these resources. As a consequence of these requests, the connection load time increases. Since the server already knows that the client needs additional files, it can save the client time by sending these resources before requesting; thus, offering a great solution to the problem.

**HTTP/1.1**

To accomplish this, HTTP/1.1 has a different technique called resource inlining, wherein the server includes the required source within the HTML page in response to the initial GET request. Though this technique reduces the number of requests that the client must send, the larger, non-text format files increase the size of the page.

As a result, the connection speed decreases, and the primary benefit obtained from it also nullifies. Another drawback is the client cannot separate the inlined resources from the HTML page. For this, a deeper level of control is required for connection optimisation – a need that HTTP/2 meets with server push.

**HTTP/2**

As HTTP/2 supports multiple simultaneous responses to the client’s initial GET request, the server provides the required resource along with the requested HTML page. This is called the server push process, which performs the resource inlining like its precursor while keeping the page and the pushed resource separate. This process fixes the main drawback of resource inlining by enabling the client machine to decide to cache/decline the pushed resource separate from the HTML page.

# **4. Buffer Overflow**

Server and client machine TCP connection requires both of these to have a certain buffer space for holding incoming requests.

Though these buffers can hold numerous or large requests, they may also lack space due to small or limited buffer size. It causes buffer overflow at receiver’s end, resulting in data packet loss. For example, packets received after the buffer is full, will be lost.

To prevent it from happening, a flow control mechanism stops the sender from transmitting an overwhelming amount of data to the receiver side.

**HTTP/1.1**

The flow control mechanism in HTTP/1.1 relies on the basic TCP connection. In beginning itself, both the machines set their buffer sizes automatically. If the receiver’s buffer is full, it shares the receive window details, telling how much available space is left. The receiver acknowledges the same and sends an opening signal.

Note that flow control can only be implemented on either end of the connection. Moreover, since HTTP/1.1 uses a TCP connection, each connection demands an individual flow control mechanism.

**HTTP/2**

It multiplexes data streams utilising the same (one) TCP connection. So, in this case, both machines can implement their flow controls instead of using the transport layer. The application layer shares the available buffer size data, after which, both machines set their receive window details on the multiplexed streams level. In addition, the flow control mechanism does not need to wait for the signal to reach its destination before modifying the receive window.

# **5. Compression**

Every HTTP transfer contains headers that describe the sent resource and its properties. This metadata can add up to 1KB or more of overhead per transfer, impacting the overall performance. For minimising this overhead and boosting performance, compressions algorithms must be used to reduce the size of HTTP messages that travels between the machines.

**HTTP/1.1**

HTTP/1.x uses formats like gzip to compress the data transferred in the messages. However, the header component of the message is always sent as plain text. Though the header itself is small, it gets larger due to the use of cookies or an increased number of requests.

**HTTP/2**

To deal with this bottleneck, HTTP/2 uses HPACK compression to decrease the average size of the header. This compression program encodes the header metadata using Huffman coding, which significantly reduces its size as a result. In addition, HPACK keeps track of previously transferred header values and further compresses them as per a dynamically modified index shared between client and server.

**Request #1**

|  |
| --- |
| method: Get |
| scheme: https |
| host: result.com |
| path: /resource |
| accept: /image/png |
| user-agent: Mozilla/5.0… |

‍**Request #2**

|  |
| --- |
| method: Get |
| scheme: https |
| host: result.com |
| path: /new\_resource |
| accept: /image/png |
| user-agent: Mozilla/5.0… |

Aside from the path field, all other fields in these requests have the

same values. So, when sending the second request, the client machine

can use the HPACK format to transfer only the indexed values required

to regenerate the common fields and encode the path field in a new

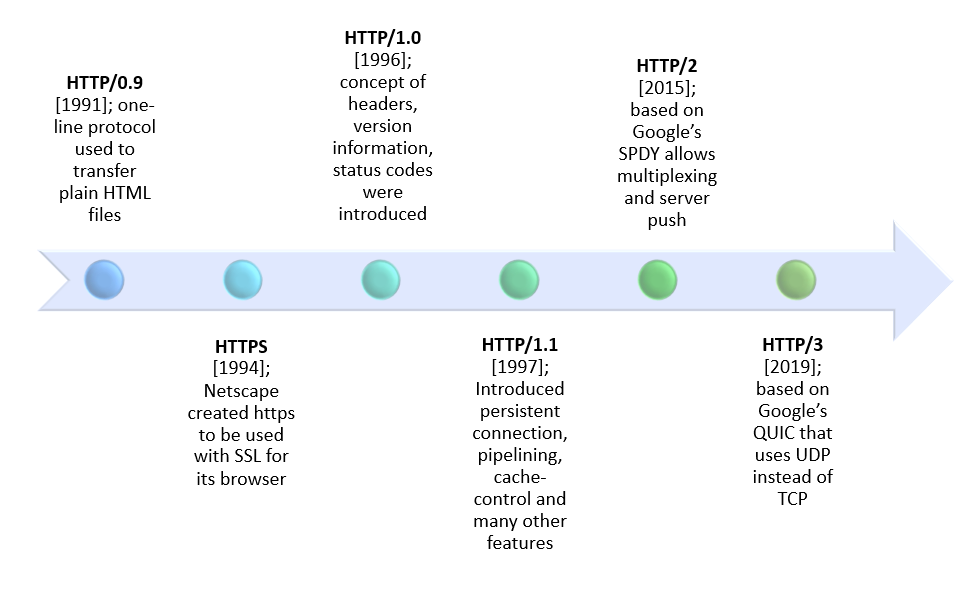
manner. The resulting header frames for both requests will look as

follows:

**HEADERS Frame for Request #1**

|  |
| --- |
| :method: Get |
| :scheme: https |
| :host: result.com |
| :path: /resource |
| accept: /image/png |
| user-agent: Mozilla/5.0… |
| HEADERS Frame **for** Request #2 |
| :path: /new\_resource |

**The below points out the differentiating factors between http2 vs http1:**

****

|  |  |  |
| --- | --- | --- |
| **DIFFERENTIATOR** | **HTTP1.1** | **HTTP2** |
| **Year** | 1997 | 2015 |
| **Key Features** | It supports connection reuse i.e. for every TCP connection there could be multiple requests and responses, and pipe-lining where the client can request several resources from the server at once. However, pipe-lining was hard to implement due to issues such as head-of-line blocking and was not a feasible solution. | Uses multiplexing, where over a single TCP connection resources to be delivered are interleaved and arrive at the client almost at the same time. It is done using streams which can be prioritized, can have dependencies and individual flow control. It also provides a feature called server push that allows the server to send data that the client will need but has not yet requested. |
| **Status Code** | Introduces a warning header field to carry additional information about the status of a message. Can define 24 status codes, error reporting is quicker and more efficient. | Underlying semantics of HTTP such as headers, status codes remains the same. |
| **Authentication Mechanism** | It is relatively secure since it uses digest authentication, NTLM authentication. | Security concerns from previous versions will continue to be seen in HTTP/2. However, it is better equipped to deal with them due to new TLS features like connection error of type Inadequate\_Security. |
| **Caching** | Expands on the caching support by using additional headers like cache-control, conditional headers like If-Match and by using entity tags. | HTTP/2 does not change much in terms of caching. With the server push feature if the client finds the resources are already present in the cache, it can cancel the pushed stream. |
| **Web Traffic** | HTTP/1.1 provides faster delivery of web pages and reduces web traffic as compared to HTTP/1.0. However, TCP starts slowly and with domain sharding (resources can be downloaded simultaneously by using multiple domains), connection reuse and pipe-lining, there is an increased risk of network congestion. | HTTP/2 utilizes multiplexing and server push to effectively reduce the page load time by a greater margin along with being less sensitive to network delays. |