**Difference between HTTP 1.1 vs HTTP 2:**

**HTTP/1.1**

Developed by Timothy Berners-Lee in 1989 as a communication standard for the World Wide Web, HTTP is a top-level application protocol that exchanges information between a client computer and a local or remote web server. In this process, a client sends a text-based request to a server by calling a method like GET or POST. In response, the server sends a resource like an HTML page back to the client.

HTTP request in the form of a text-based message:

GET /index.html HTTP/1.1

Host: [www.example.com](http://www.example.com)

This request uses the GET method, which asks for data from the host server listed after Host:. In response to this request, the example.com web server returns an HTML page to the requesting client, in addition to any images, stylesheets, or other resources called for in the HTML. Note that not all of the resources are returned to the client in the first call for data. The requests and responses will go back and forth between the server and client until the web browser has received all the resources necessary to render the contents of the HTML page on your screen. We can think of this exchange of requests and responses as a single application layer of the internet protocol stack, sitting on top of the transfer layer (usually using the Transmission Control Protocol, or TCP) and networking layers (using the Internet Protocol, or IP):

**HTTP/2**

HTTP/2 began as the SPDY protocol, developed primarily at Google with the intention of reducing web page load latency by using techniques such as compression, multiplexing, and prioritization. This protocol served as a template for HTTP/2 when the Hypertext Transfer Protocol working group httpbis of the IETF (Internet Engineering Task Force) put the standard together, culminating in the publication of HTTP/2 in May 2015.

From a technical point of view, one of the most significant features that distinguishes HTTP/1.1 and HTTP/2 is the binary framing layer, which can be thought of as a part of the application layer in the internet protocol stack. As opposed to HTTP/1.1, which keeps all requests and responses in plain text format, HTTP/2 uses the binary framing layer to encapsulate all messages in binary format, while still maintaining HTTP semantics, such as verbs, methods, and headers. An application level API would still create messages in the conventional HTTP formats, but the underlying layer would then convert these messages into binary. This ensures that web applications created before HTTP/2 can continue functioning as normal when interacting with the new protocol.

The conversion of messages into binary allows HTTP/2 to try new approaches to data delivery not available in HTTP/1.1, a contrast that is at the root of the practical differences between the two protocols. The next section will take a look at the delivery model of HTTP/1.1, followed by what new models are made possible by HTTP/2.

**HTTP/1.1 — Pipelining and Head-of-Line Blocking**

In HTTP/1.0, the client had to break and remake the TCP connection with every new request, a costly affair in terms of both time and resources. HTTP/1.1 takes care of this problem by introducing persistent connections and pipelining. With persistent connections, HTTP/1.1 assumes that a TCP connection should be kept open unless directly told to close. This allows the client to send multiple requests along the same connection without waiting for a response to each, greatly improving the performance of HTTP/1.1 over HTTP/1.0.

Unfortunately, there is a natural bottleneck to this optimization strategy. Since multiple data packets cannot pass each other when traveling to the same destination, there are situations in which a request at the head of the queue that cannot retrieve its required resource will block all the requests behind it. This is known as head-of-line (HOL) blocking, and is a significant problem with optimizing connection efficiency in HTTP/1.1. Adding separate, parallel TCP connections could alleviate this issue, but there are limits to the number of concurrent TCP connections possible between a client and server, and each new connection requires significant resources.

**HTTP/2 — Binary Framing Layer**

In HTTP/2, the binary framing layer encodes requests/responses and cuts them up into smaller packets of information, greatly increasing the flexibility of data transfer.

Let’s take a closer look at how this works. As opposed to HTTP/1.1, which must make use of multiple TCP connections to lessen the effect of HOL blocking, HTTP/2 establishes a single connection object between the two machines. Within this connection there are multiple streams of data. Each stream consists of multiple messages in the familiar request/response format. Finally, each of these messages split into smaller units called frames

**HTTP/2 — Stream Prioritization**

Stream prioritization not only solves the possible issue of requests competing for the same resource, but also allows developers to customize the relative weight of requests to better optimize application performance.

The binary framing layer organizes messages into parallel streams of data. When a client sends concurrent requests to a server, it can prioritize the responses it is requesting by assigning a weight between 1 and 256 to each stream. The higher number indicates higher priority. In addition to this, the client also states each stream’s dependency on another stream by specifying the ID of the stream on which it depends. If the parent identifier is omitted, the stream is considered to be dependent on the root stream. This is illustrated in the following figure:

**Stream Prioritization**

All other streams have some parent ID marked. The resource allocation for each stream will be based on the weight that they hold and the dependencies they require. The server uses this information to create a dependency tree, which allows the server to determine the order in which the requests will retrieve their data.

As an application developer, we can set the weights in our requests based on our needs. For example, you may assign a lower priority for loading an image with high resolution after providing a thumbnail image on the web page. The protocol also allows the client to change dependencies and reallocate weights at runtime in response to user interaction. It is important to note, however, that a server may change assigned priorities on its own if a certain stream is blocked from accessing a specific resource.

**Buffer Overflow**

In any TCP connection between two machines, both the client and the server have a certain amount of buffer space available to hold incoming requests that have not yet been processed. These buffers offer flexibility to account for numerous or particularly large requests, in addition to uneven speeds of downstream and upstream connections.

There are situations, however, in which a buffer is not enough. For example, the server may be pushing a large amount of data at a pace that the client application is not able to cope with due to a limited buffer size or a lower bandwidth. Likewise, when a client uploads a huge image or a video to a server, the server buffer may overflow, causing some additional packets to be lost.

In order to avoid buffer overflow, a flow control mechanism must prevent the sender from overwhelming the receiver with data. This section will provide an overview of how HTTP/1.1 and HTTP/2 use different versions of this mechanism to deal with flow control according to their different delivery models.

**Predicting Resource Requests**

In a typical web application, the client will send a GET request and receive a page in HTML, usually the index page of the site. While examining the index page contents, the client may discover that it needs to fetch additional resources, such as CSS and JavaScript files, in order to fully render the page. The client determines that it needs these additional resources only after receiving the response from its initial GET request, and thus must make additional requests to fetch these resources and complete putting the page together. These additional requests ultimately increase the connection load time.

There are solutions to this problem, however: since the server knows in advance that the client will require additional files, the server can save the client time by sending these resources to the client before it asks for them. HTTP/1.1 and HTTP/2 have different strategies of accomplishing this, each of which will be described in the next section.

**HTTP/1.1 — Resource Inlining**

In HTTP/1.1, if the developer knows in advance which additional resources the client machine will need to render the page, they can use a technique called resource inlining to include the required resource directly within the HTML document that the server sends in response to the initial GET request.

A major drawback of resource inlining, then, is that the client cannot separate the resource and the document.

**HTTP/2 — Server Push**

Since HTTP/2 enables multiple concurrent responses to a client’s initial GET request, a server can send a resource to a client along with the requested HTML page, providing the resource before the client asks for it. This process is called server push. In this way, an HTTP/2 connection can accomplish the same goal of resource inlining while maintaining the separation between the pushed resource and the document. This means that the client can decide to cache or decline the pushed resource separate from the main HTML document, fixing the major drawback of resource inlining.

It is important to note here that the emphasis of server push is client control. If a client needed to adjust the priority of server push, or even disable it, it could at any time send a SETTINGS frame to modify this HTTP/2 feature.

**Compression**

A common method of optimizing web applications is to use compression algorithms to reduce the size of HTTP messages that travel between the client and the server. HTTP/1.1 and HTTP/2 both use this strategy, but there are implementation problems in the former that prohibit compressing the entire message.

**HTTP/1.1**

Programs like gzip have long been used to compress the data sent in HTTP messages, especially to decrease the size of CSS and JavaScript files. The header component of a message, however, is always sent as plain text. Although each header is quite small, the burden of this uncompressed data weighs heavier and heavier on the connection as more requests are made, particularly penalizing complicated, API-heavy web applications that require many different resources and thus many different resource requests. Additionally, the use of cookies can sometimes make headers much larger, increasing the need for some kind of compression.

**HTTP/2**

One of the themes that has come up again and again in HTTP/2 is its ability to use the binary framing layer to exhibit greater control over finer detail. HTTP/2 can split headers from their data, resulting in a header frame and a data frame. The HTTP/2-specific compression program HPACK can then compress this header frame. This algorithm can encode the header metadata using Huffman coding, thereby greatly decreasing its size. Additionally, HPACK can keep track of previously conveyed metadata fields and further compress them according to a dynamically altered index shared between the client and the server.

Using HPACK and other compression methods, HTTP/2 provides one more feature that can reduce client-server latency.

**HTTP version history**

|  |  |
| --- | --- |
| **Year** | **HTTP Version** |
| 1996 | 1.0 |
| 1997 | 1.1 |
| 2015 | 2.0 |
| Draft (2020) | 3.0 |

**Browser JS vs Node JS**

|  |  |
| --- | --- |
| Node JS | Browser JS |
| Node doesn't have a predefined "window" object cause it doesn't have a window to draw anything. | "window" is a predefined global object which has functions and attributes, that have to deal with window that has been drawn. |
| "location" object is related to a particular url; that means it is for page specific. So, node doesn't require that. | "location" is another predefined object in browsers, that has all the information about the url we have loaded. |
| Node doesn't have "document" object also, cause it never have to render anything in a page. | "document", which is also another predefined global variable in browsers, has the html which is rendered. |
| Node has "global", which is a predefined global object. It contains several functions that are not available in browsers, cause they are needed for server side works only. | Browsers may have an object named "global", but it will be the exact one as "window". |
| "require" object is predefined in Node which is used to include modules in the app. | Browsers don't have "require" predefined. You may include it in your app for asynchronous file loading. |
| In Node everything is a module. You must keep your code inside a module. | Moduling is not mandatory in client side JavaScript, i.e. in browsers. |
| Node is headless. | Browsers are not headless. |
| Node processes request object. | Browsers processes response objects. |

**What happens when we type an URL in address bar:**

1. Enter a URL into a web browser
2. The browser looks up the IP address for the domain name via DNS
3. The browser sends a HTTP request to the server
4. The server sends back a HTTP response
5. The browser begins rendering the HTML
6. The browser sends requests for additional objects embedded in HTML (images, css, JavaScript) and repeats steps 3-5.
7. Once the page is loaded, the browser sends further async requests as needed.