Difference between HTTP1.1 vs HTTP2

HTTP

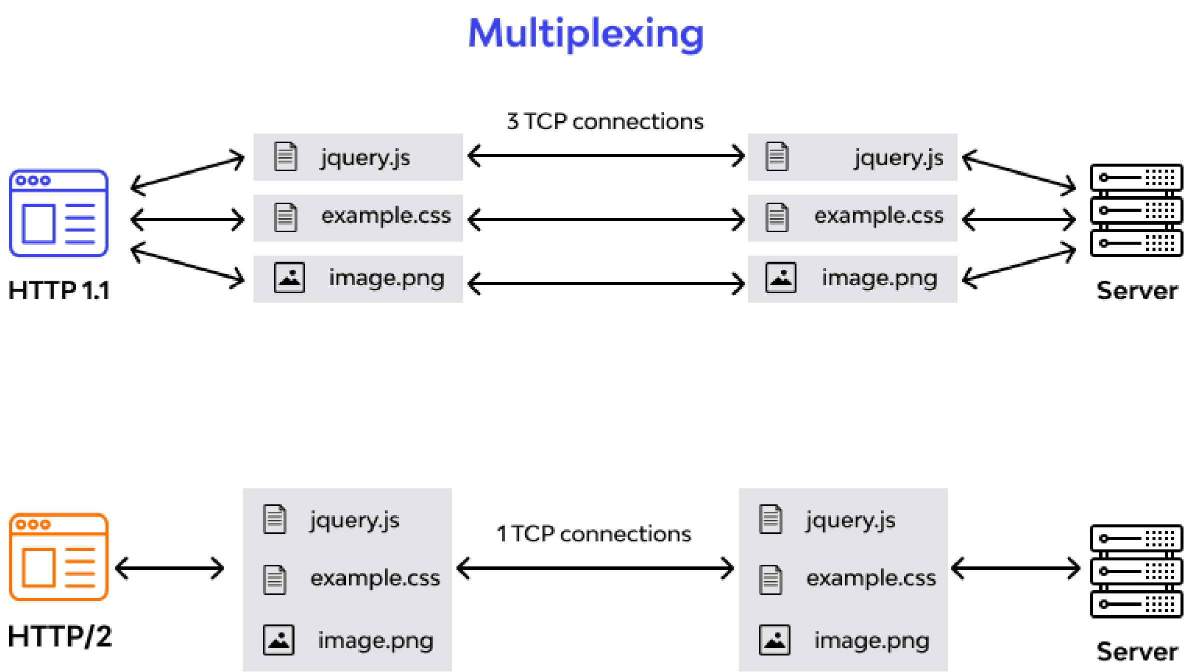
Hypertext Transfer Protocol (HTTP) is an application-layer protocol for transmitting hypermedia documents, such as HTML. It was designed for communication between web browsers and web servers, but it can also be used for other purposes. HTTP follows a classical client-server model, with a client opening a connection to make a request, then waiting until it receives a response. HTTP is a stateless protocol, meaning that the server does not keep any data (state) between two requests.

HTTP/1.1 and HTTP/2 Main Differences

The Background

For better contextualization of the certain alterations that HTTP/2 made to its precursor, we'll take a quick look at their basic functionalities and development details first.

HTTP / 1.1	HTTP / 2
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. Keep in mind that 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. Now, let's see how HTTP/2 handles the same scenario.	HTTP/2 was released at Google as the significant improvement of its predecessor. It was initially modeled after the SPDY protocol and went through significant changes to include features like multiplexing, header compression, and stream prioritization to minimize page load latency. After its release, Google announced that it would not provide support for SPDY in favor 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.



Delivery Models

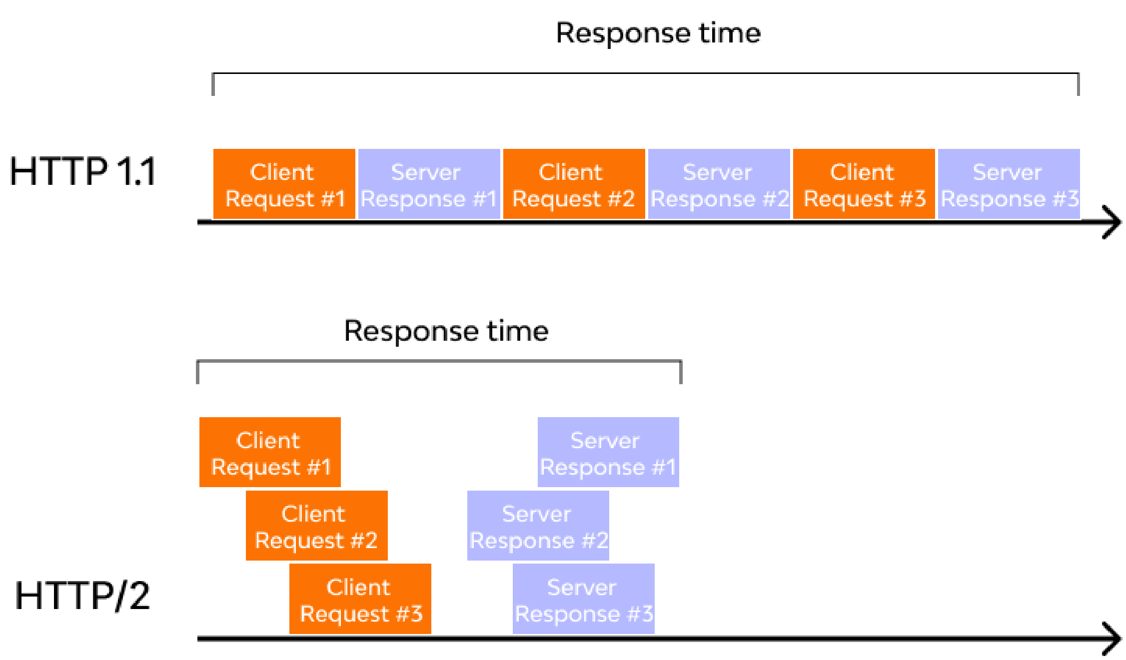
As discussed before, HTTP/1.1 sends messages as plain text, and HTTP/2 encodes them into binary data and arranges them carefully. This implies that HTTP/2 can have various delivery models.

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 / 2
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	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

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.	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 prioritization. When sending streams in parallel, the client can assign weights (1-256) to its stream to prioritize 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 prioritization ability.



Predicting Resource Requests

As already discussed, 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	HTTP / 2
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 optimization – a need that HTTP/2 meets with server push.	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.

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	HTTP / 2
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.	It multiplexes data streams utilizing 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.

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 minimizing 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 / 2
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.	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.