HTTP1.1 vs HTTP2

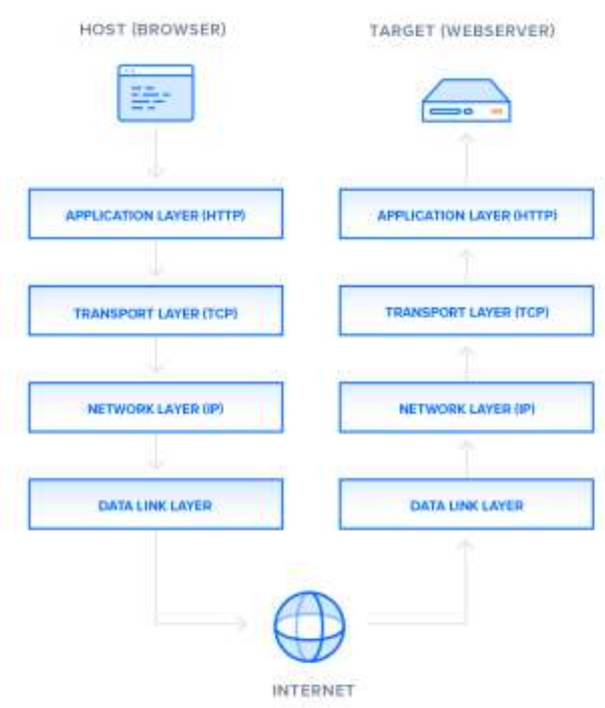
Understanding how HTTP2 is different from HTTP1.1 and how it helps improve performance

Introduction

The Hypertext Transfer Protocol, or HTTP, is an application protocol that has been the de facto standard for communication on the World Wide Web since its invention in 1989. From the release of HTTP/1.1 in 1997 until recently, there have been few revisions to the protocol. However, in 2015, a reimagined version called HTTP/2 came into use, which offered several methods to decrease latency, especially when dealing with mobile platforms and server-intensive graphics and videos.

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 the beginning, many browsers supported this standardization effort, including Chrome, Opera, Internet Explorer, and Safari. Due in part to this browser support, there has been a significant adoption rate of the protocol since 2015, with especially high rates among new sites.

Typical HTTP Connection



A typical request say “GET” starts from Host and goes to the HTTP application layer where the GET request is understood and the message is configured in HTTP format. Then it moves to TCP layer, which establishes the connection via the network and data link layer. It now moves from the internet to the data link and network layer and enters to Target’s TCP layer and then into the HTTP layer where the message is deciphered. Our focus is going to be mainly on the TCP and HTTP layer.

Understanding the differences

We will analyse the two protocols in terms of their delivery method, buffer management, predicting resource requests and compression.

Delivery Method

HTTP1.1	**HTTP2**
Transfers in ***plain text messages***	Transfers by ***encoding into binary frames***
Manages multiple requests ***by persistence connection and pipelining*** i.e. keeping TCP connection open unless directly told to close and keeping a pipeline of requests to be solved one after another. However, if one is unfulfilled then it can result into ***"Head of line blocking".*** The way to solve this optimization issue, HTTP1.1 proposes parallel TCP connections which require significant resources on both ends	The communication consists of bunch of binary-encoded frames tagged to a particular stream. These tags help in un-assembling at client end and re-assembling at server end. This way requests and ***responses run in parallel using multiplexing***. Hence, a single TCP connection is used. Now, HTTP2 uses ***stream prioritization*** to solve for multiple requests hitting a resource and overloading the same. Each stream is given a weight (1 to 256, higher the no. earlier should it be processed) and a dependency i.e. stream ID which should be run before it. This creates a node tree of requests and are resolved as per the tree.

Buffer Overflow

In a TCP connection, both sides have a buffer space to hold multiple requests. These buffers may overflow, if cache on that side is not cleared locally.

HTTP1.1	**HTTP2**
Uses "ACK Packet" i.e. receive window, which describes what size of the buffer is available to the sender. The sender will send only what the buffer size allows and then stop transfer. Transfer will again start when the buffer again becomes available. Each TCP connection will have it's own buffer management.	While the receive window method is similar to HTTP1.1 but the buffer management is moved to HTTP layer where for each multiplexed stream sets it's buffer. The application layer communicates the available buffer space, allowing the client and server to set the receive window on the level of the multiplexed streams. What this means is that an image stream can send first scan and then other responses are sent and then the remaining image information can be sent.

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.

HTTP1.1	HTTP2
Uses Resource Inlining: Puts in all the CSS and JS in the HTML document as inline. This increases the size of HTML document. It also means that each HTML document will be of a larger size thereby eating more time and resources.	Uses ***"Server Push"*** approach. The CSS and JS are kept separate from the HTML document. The developer decides that the additional document which are required are pushed to the client before the client requests. However, there is a PUSH_PROMISE sent first which the client assess and reverts to send or will say no as it already has the resource in it's cache. But this has some perils as many browsers may not recognize the PUSH_PROMISE which will result in more resources being sent than required. Hence, it is purely a developer call to use or not.

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.

HTTP1.1	HTTP2
The data files of CSS and JS are compressed using gzip but the header is always sent as plain text. With multiple requests this also become heavy on resources	Uses ***"HPACK"*** compression. The first request will have all the info, and the following requests will have only the parts which have changed. This is done by keeping a track of previously conveyed metadata fields.

Conclusion

HTTP2 significantly improves the performance of the web applications using binary-encoded frames, multiplexing, stream prioritization, buffer optimization, server push and HPACK header compression.