HTTP/2.0

Half-Blood Prince
Kito Mam
Sandy Yi
Paul Diaz

Computer Networks I
Prof. Robert Cartelli
San Jose State University

Abstract

This paper covers in depth of HTTP/2.0, the newest version of HTTP network protocol. The first part of this paper examines the history and evolution of HTTP in order to get familiarized with the different versions and added features from one version to another. This paper also talks about the differences of the last three versions of HTTP–which are HTTP/1.1, SPDY (pronounced as *speedy*) by Google, and HTTP/2.0–by comparing and contrasting them in terms of performance. The main emphasis of this research will be about the improvements and enhancements found in HTTP/2.0. Specifically, we talk about HTTP/2.0's binary framing, streams, messages, frames, request and response multiplexing, stream prioritization, flow control, server push, and header compression. The last part of this research is about a web security policy mechanism called HTTP Strict Transport Security (HSTS).

1. Introduction

One of the core technologies powering the web is Hypertext Transfer Protocol (HTTP). It has been more than 25 years since Tim Berners-Lee initiated the development of HTTP at CERN, the European Organization for Nuclear Research. Standard development of HTTP was through the collaboration of Internet Engineering Task Force (IETF) together with the World Wide Web Consortium (W3C), resulting in the formation and publication of different versions of Requests for Comments (RFCs). HTTP has undergone many changes and improvements through the years; its simplicity is preserved and most importantly flexibility enhanced. HTTP has gradually developed, from being just a simple protocol for the purpose of exchanging files and data in a somewhat reliable laboratory environment, to a new protocol that can carry different types of images, as well as high resolution and 3D videos into the modern complex network of the Internet.

2. Evolution of HTTP

In 1991, the first version of HTTP was specified and introduced by Berners-Lee. While he was working at CERN, he wrote a proposal for the purpose of building a hypertext system over the Internet, which was first called *Mesh* and later called *WorldWideWeb.* This proposal was built over the existing Transfer Control Protocol (TCP) and Internet Protocol (IP) protocols (Lester). The foundation of the first version were as follows: a client to display HTML documents, a server to use Hypertext Markup Language (HTML) in order to provide access to the documents, HTML to represent hypertext documents, and HyperText Transfer Protocol (HTTP) for exchanging HTML documents (Grigorik).

2.1. HTTP/0.9 The One-Line Protocol

In 1991, the very first version of HTTP was documented. Initially, this version had no version name nor number; however, when developers decided to update the first version (HTTP/1.0), they decided to name it HTTP/0.9 in order to distinguish between the two and future versions (Lester). Developers called the first version the “one-line protocol” because each request consisted of a single line of code, which starts with the only method GET, a slash, and followed by the path to the resource (Grigorik).

The primary goals of Berners-Lee was file transfer capability, client-server referencing, format negotiation, and a functionality for requesting an index search of any hypertext document archive (Grigorik). In order to implement all of these goals, he built a simple prototype that executed a small-scale branch of the proposed features and functionalities. These are as follows: request-response protocol (client and server), ASCII protocol running over a link (TCP/IP), a functionality capable of hypertext documents (HTML) making the connection close every completed document request transfer (Grigorik).

HTTP/0.9 specifically had no HTTP headers, which means that HTML files were plainly meant to be transmitted, but no other types of documents (Lester). One of the disadvantages of this version is that it did not contain any status nor error codes. Consequently, whenever a problem occurs, the specific HTML file that aimed to be transmitted will be sent back to the client and attached a description of the problem (Grigorik).

2.2. HTTP/1.0 Rapid Growth and Information RFC

Between the years 1991 and 1995, HTML specification and the “web browser” evolved drastically. This is due to the prolific numbers of web experimentations conducted between these years (Grigorik). It was the time of progressive emergence and fast development of the public Internet infrastructure (consumer-oriented). With HTTP/0.9’s limited features and functionalities, public Internet needed a protocol that could function as more than just hypertext documents. It needed a protocol that could supply richer and better request and response metadata and allow content negotiation (Grigorik). Subsequently, in May 1996, the HTTP Working Group (HTTP-WG) published RFC 1945: HTTP/1.0. However, it is important to note that this specification is not a formal nor an Internet standard, but only an informational RFC (Lester).

In HTTP/1.0, some of the key changes are as follows: a request can contain numerous newline separated header fields, a response object now has its own set of newline separated header fields, response status line is utilized, and the connection between client and server is closed every after every request (Grigorik). HTTP/1.0 is evidence that the headers of request and response have employed to be ASCII encoded. Furthermore, the response object has extended its flexibility where it can now respond a type from an HTML file to plain text file, images, or any other type. Consequently, HTTP/1.0 sometimes called it *hypermedia transport* (Grigorik).

2.3. HTTP/1.1 Internet Standard

HTTP/1.1 has evolved in two different distinct phases. The first evolution phase of HTTP/1.1, called RFC 2068, happened on January 1999, six months after the publication of HTTP/1.0. With some few updates, in this phase, HTTP/1.0 was officially standardized. Two and half years later, HTTP/1.1 evolution phase two called RFC 2616 followed. This phase added more functionalities and updates, where it explains some of issues and ambiguities in RFC 2068.

The first improvement is the ability to reuse connection: this is to address the issue of slow process in displaying resources contained in every document transmitted. The second improvement is for the purpose of decreasing the latency of the communication, HTTP/1.1 provides the ability to implement pipelining. This feature allows transmitting the next request before the response to the first request finishes its transmission. Third is the ability to transmit responses in chucks. This is in the connection to pipelining feature. The fourth improvement is the addition of cache control mechanism, which makes website connection and searching even faster. The fifth improvement allows a client and a server to conform on content exchange negotiation. Finally, the last improvement is the ability to allow server collocation, where it can host multiple domains at a single/same IP address (Lester).

2.4. SPDY Google

In June 2015, as part of the initiative to make the performance of web faster and better at transporting web content, Google (Mike Belshe and Robert Peon) conducted numerous experiments that aimed to decrease the latency of web pages. One of these experiments is an application layer protocol called SPDY. This protocol was tested by Google developers and found to load up a page 64% faster compared to HTTP/1.1 (Levin, Engel).

SPDY supplements HTTP/1.1 with a few speed performance features that can considerably decrease the time to load pages. SPDY's first feature enables client and server to compress request and response headers. This could result in a smaller amount of bandwidth usage when similar headers are transmitted numerous times over for numerous numbers of requests (Yue-Herng). Secondly, it allows multiple and simultaneously multiplexed streams requests over a single connection. This means that unlimited concurrent streams are permitted over a single TCP connection. TCP efficiency is increased since there are fewer network connections needed to be made. Though packets are more dense, the packets needed to be transmitted has become fewer. This can prevent on round trips between client and server, preventing low-priority resources from blocking higher priority requests. Another feature is that it allows the server to actively push resources to the client that it knows the client will need without waiting for the client to request them, allowing the server to make efficient use of unutilized bandwidth (Levin, Engel).

A few months after June of 2015, SPDY added an extra feature called request prioritization. Although the serialization problem is resolved by immeasurable amount of parallel streams, there is a potential problem that could happen (Levin, Engel). For example, when the bandwidth on the channel is restricted, by default,the client may block requests in order to maintain the channel from having too much traffic; however, this strategy could potentially lose tracking of the requests and may end up not responding to some of the client's requests. To resolve this problem, with SPDY, it assigns a priority to every request preventing requests drop and congested channel (Levin, Engel).

3. Improvements and Enhancements of HTTP/2.0

SPDY was a success for sometime. In fact, it was supported and adopted by servers and and many browsers: servers such as NGINX (pronounced as engine x) and browsers like Chrome, Firefox, and Opera, and others; however, Microsoft's Edge browser has dropped its support due to Microsoft implementing support for the creation of HTTP/2.0 (Andrew). In 2015, many browsers still supported SPDY, but this year, 2016, Chrome has removed its support as well, and expects other browsers to follow. In this case, HTTP/2.0 has been created and built by HTTP Working Group (HTTP-WG) (Andrew). The starting point of HTTP/2.0 was due to the success of SPDY: on February 2015, HTTP/2.0 specification has been finalized and published.

Most browsers these days work around the above the issues by opening multiple connections to a server. But this can lead to other problems notably increased load, especially on the server. HTTP/2.0 aims to address the issue of increased page load, specifically on the server side; and this is done by changing the client and server communication process and technique: the concept of frames and streams (Levin, Engel).

3.1. Binary Framing

Binary framing is one of the important performance features of HTTP/2.0. Binary framing precepts how HTTP messages are being encapsulated and transmitted from client to server, and vice versa. Before HTTP messages are transmitted, they will be divided into frames, which will be encoded into binary format (Grigorik). Both client and server are required to negotiate and approve the type of mechanism to be used for binary encoding. This is crucial to happen for the client and server, otherwise, they will not understand each other and therefore HTTP message transmission will never work. However, on the bright side, whatever application currently running does not need to know about the encoding mechanism. In fact, the running application is unaware of the mechanism because the client and the server will execute all the required framing work for our benefit (Reilly).

If we compare the textual protocols used in earlier HTTP versions, HTTP/2.0 binary framing protocols is very efficient in decomposing and parsing HTTP messages. Moreover, since messages are in frames and binary encoded, messages are much more compact and therefore easier and faster in transmission. In addition, HTTP/2.0 binary framing exhibits a much robust mechanism in terms of error occurrence. Although, binary framing is typically much complicated to implement accurately, it is easier to inspect and debug errors (Grigorik). Tools such as the Wireshark plugin are available to incorporate with HTTP/2.0 for inspecting and debugging (Reilly).

3.2. Request and Response Multiplexing

In HTTP/1.0 version, in order to enhance the delivery performance, a client needs to create multiple TCP connections when sending multiple requests. This is because each response can only be transmitted in one response per connection. Though it allows sending multiple requests, this strategy however was problematic because it created a complication called head-of-line blocking, a situation where multiple packets in line are blocked by the first packet (Andrew).

In response to this problem, HTTP/1.1 introduced a new feature called pipelining, where numerous HTTP requests are constantly transmitted over one TCP connection without waiting for responses. Pipelining did not fully solve this issue because there are still apparent blockages of packets happening, especially with large packets and/or slow response packets (Grigorik). One of the main problems of pipelining is that there is much overhead in implementing and it can be very strenuous to utilize: this is due to the fact that the many servers and intermediaries processes are weak and often done inaccurately. Consequently, in order to load a page, clients have the problem of deciding which request must be transmitted to which connection (Reilly). Again, this creates another overhead and issue; according to Grigorik, loading a page requires more than 10 times the total number of accessible connections. Therefore, blockage of multiple requests affects the entire delivery and page loading performance (Grigorik).

HTTP/2.0 solves HTTP/1.1's pipelining problem by eliminating the creation of multiple connections and allowing the continuous transmission of numerous requests simultaneously in one dedicated connection. This could result an easier, simpler, and cheaper deployment process and making the whole loading operation easier and faster.

3.3. Server Push

Whenever we visit a website over the Internet, we use browsers to request for a page to load. When the request is sent, HTML will be transmitted in the response from the server and browser, then the HTML received will need to be decomposed and parsed. Before the browser can start transmitting files such as JavaScript and CSS, including images, it needs to release requests for all of the embedded resources

(Grigorik). This process can be cumbersome at times and can compromise the performance. The back-and-forth HTTP message transmission creates a round-trip delay, making the whole process slower (Grigorik). To avoid all these issues, HTTP/2.0 allows one dedicated connection to be open for the entire communication process and lets the server push all the required resources to the client. This feature is called the server push (Reilly).

Pushing all related and required resources ahead of time prevents round-trip delay. This is true because the server already knows in advance what resources the client/browser will need to load the website visited (Reilly). An example of this is when developers insert CSS and/or JavaScript into an HTML document. However, if server push is not done correctly, it can compromise the performance process. There are still continuing experimentation and research on this subject area (Reilly).

3.4. Header Compression

In the earlier versions of HTTP, every transmission of request or response is embedded by headers that detail information about them. According to Patrick McManus, the principal engineer at Mozilla and current Co-Chair for IETF HTTPBIS Working Group, each header typically contains 1400 bytes (plus few kilobytes for cookies) and requires at least 80 assets per page. McManus calculated that it will take seven to eight round trips (doesn’t include response time) containing between 500 to 800 bytes of bandwidth per transfer (Reilly). This compromises the performance and bandwidth latency. HTTP/2.0 resolves all of these problems implementing header compression using the Huffman Code compression format (Grigorik).

The main goal of HTTP/2.0 header compression is to limit the trips of transfer (containing uncompressed headers – huge bytes) by preventing the headers that are already transmitted from retransmitting. In order to accomplish this, first, all headers are converted into a compressed form before they get transferred using Huffman Code. HTTP/2.0 header compression forces client and server to maintain an index table that records the compressed headers received (Grigorik). If there is a retransmission of header, instead of transmitting the entire compressed header, it will only send the index referenced to the same actual header, avoiding retransmission of the same exactly header. HTTP/2.0, not only decreases the overall length of headers (compressed headers), but also prevents header retransmission, which essentially enhances the HTTP request and response transmission (Grigorik).

3.5. Flow Control

Flow control is a mechanism to help reduce the chances of the client sending useless requests to the server so that the server does not have to fetch and buffer unnecessary data. To fix the problem, HTTP/2.0 implemented a set of building blocks to allow the client and the server to implement their own stream and connection level flow control. Notably, first, flow control in HTTP/2.0 is bidirectional; the client could choose the specified window size for each stream or entire connection. Second, flow control is credit-based; the client initializes the connection and stream window, so that with the server response, the frames are reduced via a WINDOW_UPDATE. Flow control can not be disabled. For example, when the connection is established, the window size has to be exchanged in SETTING frame to 65,535 bytes (Reilly). In addition, the client can update it to a larger size by sending WINDOW_UPDATE frame whenever data is received. As described above, HTTP/2.0 doesn't really implement any solid algorithm, but instead offers simple building blocks and defers the implementation to the client and the server (Grigorik).

3.6. Streams, Messages, Frames

HTTP/2.0 introduced the binary framing mechanism which changes how data is transferred between the client and server. It consists of streams, messages, and frames. The stream is a bidirectional data flow within one established connection. It can transfer one or more messages. The message is a full message of sequence that maps to one logical request or response message. The frame is the smallest unit in HTTP/2.0. Each frame contains the frame header. Every communication in HTTP/2.0 is executed only on a single TCP connection. It can send bidirectional streams. Every stream has a unique identifier, and it is used to carry out the bidirectional messages. Lastly, every message consists of a logical HTTP message. It contains either a request or response which consists of frames (Grigorik).

4. Performance Comparison Between HTTP/1.1, SPDY, and HTTP/2.0

4.1. Size of Request and Response Headers

As of now, there are not many browsers that support HTTP/2.0 by default. Only a few, such as Google and Firefox, have enabled the support of HTTP/2.0. It is not the final version, but rather the Draft 14 version of HTTP/2. In the performance comparison, HTTP/1.1, SPDY and HTTP/2.0 will be compared in four areas. They are: size of request and response headers, size of the response message, number of TCP connects and SSL handshakes required during page load, and page load time. First, let's talk about the performance comparison in the size of request and response headers. The test is conducted with HTTPWatch, a site that logs the loading time when processing HTML on Firefox which tests on simple web pages on the Google UK homepage using HTTP/1.1, SPDY/3.1 and HTTP/2.0 (Trekner). A lot of websites out there compressed its textual content for a optimal performance. However, the HTTP/1.1 doesn't have a compression algorithm in the header; this results in a bigger header size. For SPDY/3.1 and HTTP/2.0, it introduced the DEFLATE and HPACK algorithms respectively. These new algorithms compress the header and are created to fix the shortcoming of using different types of compression. As for the results, the HTTP/2.0 HPACK algorithm works the best. Its header size is significantly smaller than that of HTTP/1.1 and SPDY/3.1, which uses DEFLATE algorithm to compress the header (Trekner).

4.2. Size of Response Message

In the second performance test, the size of the response message is tested to see which protocol contains the smallest size. As reviewed above, HTTP/2.0 has the smallest header size, which could lead to smaller sized response message. This is true and not true. It is true when a site contains image resources. But HTTP/2.0 doesn't produce the smallest sized response message when the site contains textual resources. The reason behind this is because Google servers are adding to the data frames of textual resources (Trekner). The padding is being used to prevent the attack within the HTTP because the compressed content could contain both attacker controlled plaintext and secret data. This is not the case for image resources because they are already compressed. The winner for this test is SPDY/3.1. Even though it has larger header size, its response message size is consistently small (Trekner).

4.3. Number of TCP Connects and SSL Handshakes Required During Page Load

More on performance comparison - this phase will compare the number of TCP connections and SSL handshakes required during page load. In the traditional HTTP/1.1, it allows a better performance by running multiple connections per hostname. This also allows concurrency and hence boosts the website loading performance. But such a boost in performance comes with a cost. Extra TCP connections and SSL handshakes are used to archive this high performance (Trekner). SPDY/3.1 and HTTP/2.0 wants to archive this high performance with multiple connection, but also wants to eliminate the cost of extra TCP connection and SSL handshakes. It fixes this issue by using multiplexing to allow more than one request at a time to send and receive data on a single connection (Trekner).

4.4. Page Load Time

The last performance comparison is the page load time. Page load time can be said to truly measure the performance of these three protocols. Base from the test from HttpWatch, the clear winner is HTTP/2.0. For traditional HTTP/1.1, it takes so much time to load a website page due to its uncompressed header that it results in a bigger size header (Trekner). Also, the cost of maintaining more than a single connection on a host to run concurrency by using extra TCP and SSL connection, also slows the down the time of loading the webpage. For SPDY/3.1, it is still slower than HTTP/2.0. Even though HTTP/2.0 can have a larger a response message than SPDY/3.1, it is still faster than

SPDY/3.1 (Trekner). This is probably because HTTP/2.0 always produce the smaller GET request in the header.

5. HTTP Strict Transport Security (HSTS)

Transmission Control Protocol (TCP) is commonly used for HTTP transmission. Sending data with plain text will be bombarded by many hack attacks and potentially a process of collecting of metadata by unwanted connection/client/hacker. TCP uses Secure Sockets Layer (SSL) and Transport Layer Security (TLS), an advanced version of SSL for channel oriented security.; however, TCP does not provide channel integrity protection, confidentiality, nor secure host identification. To address this issue, Mark Nottingham, the chairperson for HTTPbis Working Group who developed HTTP/2.0, mandated HTTP over SSL or HTTP Secure (HTTPS) for HTTP/2.0 (Hodges, Jackson, Barth).

In HSTS, user agents converts URI references from being insecure into an HSTS host secure URI references and closes all secure transport connection executions when transport errors and warnings occur. The goal of HTTPS forces browsers to only communicate with other browsers that are using HTTPS. All websites must enforce and implement HTTPS to browsers every time they load a page. If a browser doesn’t support HSTS then the page simply fails; however, if a client uses HTTP but the website it's trying to connect support HTTPs, it will automatically transform all load page from HTTP to HTTPS (Hodges, Jackson, Barth).

6. Conclusion

In conclusion, this research has given insight on how the network protocol has advanced from the previous protocols such as HTTP/1.0 and SPDY. We reviewed the history of HTTP/1.0 and SPDY which allows us to easily spot and compare the innovative features introduced by the

HTTP/2.0 to fix the shortcomings of previous network protocols. To recap, there are many new improvements and enhancements to make network protocol even better and more secure in HTTP/2.0. In addition, with the HPACK algorithm, HTTP/2.0 is able to reduce the size of the header significantly, resulting in the shortest web page loading time. Also, it aids in HTTP/2.0 by making the content safer to prevent attacks in the compressed textual file. Lastly, flow control, server push stream prioritization, and the binary framing mechanism help make the HTTP/2.0 protocol the best and secure network protocol in the industry today.

References

Lester, D. (2016, July 20). Evolution of HTTP (C. Riecan, Ed.). Retrieved October 22, 2016, from MDN Mozilla Developer Network website: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP. This article aims to present the evolution of Hypertext Transfer Protocol (HTTP). Also, it talks about the invention of the World Wide Web including the early protocols; starting from HTTP/.09, HTTP/1.0, HTTP/1.1, SPDY (pronounced as *speedy*) of Google, and HTTP/2.0

Grigorik, I. (2015, November 3). HTTP/2. Retrieved October 29, 2016, from High Performance Browsing Networking website: https://hpbn.co/http2/. This article specifically talks about the design and technical goals of the latest HTTP/2.0 protocol. It also talks about the performance enhancements of HTTP/2.0 in terms of different aspects; such as binary framing layer, streams, messages, frames, request and response multiplexing, stream prioritization, flow control, server push, and header compression.

Andrew, R. (2016, February 16). Getting Ready for HTTP/2: A Guide for Web Designers And Developers. Retrieved October 29, 2016, from Smashing Magazine website: https://www.smashingmagazine.com/2016/02/getting-ready-for-http2/. This article looks at the basics of HTTP/2.0 and the application into web browsers. It explains the key features of this new protocol, browser and server compatibility, and the see how the adaptation of recent versions to HTTP/2.0.

Trekner, U. (2015, January 16). A Simple Performance Comparison of HTTPS, SPDY, and HTTP/2.0. Retrieved October 29, 2016, from HttpWatch website: http://blog.httpwatch.com/2015/01/16/a-simple-performance-comparison-of-https-spdy-and-http2/comment-page-1/. The contents of this article covers the performance of different HTTP versions; such as HTTPS, SPDY, and HTTP/2.0. This article compares these three versions in terms of the size of request and response headers, size of response message, number of TCP connects and SSL handshakes required during page load, and page load time.

Levin, S., & Engel, S. (2015, May 27). SPDY Overview. Retrieved November 3, 2016, from Google Developers website: https://developers.google.com/speed/spdy/ This page provides an overview of the new technology for networking protocol developed by Google called SPDY. It also provides the major features of this technology which are all in connection with web performance.

Yue-Herng, L. (2015, March 3). SPDY: An experimental protocol for a faster web (B. David, Ed.). Retrieved November 3, 2016, from The Chromium Projects website: https://www.chromium.org/spdy/spdy-whitepaper This article talks about the functionalities and features of SPDY networking protocol developed by Google. It talks about the background of SPDY, design goals, and features including advanced features. The last part of this article talks about SPDY implementation and its primary results.

Reilly, W. (2015, October 2). HTTP/2 A Quick Look. Retrieved November 7, 2016, from Scott Logic / Altogether Smarter website: http://blog.scottlogic.com/2014/11/07/http-2-a-quick-look.html This article starts talking about the basis and foundation of HTTP/2.0. The middle part of this article talks about features and enhancements offers by HTTP/2.0.

Hodges, J., Jackson, C., & Barth, A. (2012, November 1). HTTP Strict Transport Security (HSTS). Retrieved November 13, 2016, from IETF Tools website: https://tools.ietf.org/html/rfc6797 This website provides the publication of HSTS RFC 6797. It talks about the entire specification of HSTS reviewed and approved by Internet Engineering Task Force (IETF).