Brief History of HTTP

HTTP, CHAPTER 9

The Hypertext Transfer Protocol (HTTP) is one of the most ubiquitous and widely adopted application protocols on the Internet: it is the common language between clients and servers, enabling the modern web. From its simple beginnings as a single keyword and document path, it has become the protocol of choice not just for browsers, but for virtually every Internet-connected software and hardware application.

In this chapter, we will take a brief historical tour of the evolution of the HTTP protocol. A full discussion of the varying HTTP semantics is outside the scope of this book, but an understanding of the key design changes of HTTP, and the motivations behind each, will give us the necessary background for our discussions on HTTP performance, especially in the context of the many upcoming improvements in HTTP/2.

HTTP 0.9: The One-Line Protocol §

The original HTTP proposal by Tim Berners-Lee was designed with *simplicity in mind* as to help with the adoption of his other nascent idea: the World Wide Web. The strategy appears to have worked: aspiring protocol designers, take note.

In 1991, Berners-Lee outlined the motivation for the new protocol and listed several high-level design goals: file transfer functionality, ability to request an index search of a hypertext archive, format negotiation, and an ability to refer the client to another server. To prove the theory in action, a simple prototype was built, which implemented a small subset of the proposed functionality:

- Client request is a single ASCII character string.
- Client request is terminated by a carriage return (CRLF).
- Server response is an ASCII character stream.
- Server response is a hypertext markup language (HTML).
- Connection is terminated after the document transfer is complete.

However, even that sounds a lot more complicated than it really is. What these rules enable is an extremely simple, Telnet-friendly protocol, which some web servers support to this very day:

```
$> telnet google.com 80

Connected to 74.125.xxx.xxx
```

```
(hypertext response)
(connection closed)
```

The request consists of a single line: GET method and the path of the requested document. The response is a single hypertext document—no headers or any other metadata, just the HTML. It really couldn't get any simpler. Further, since the previous interaction is a subset of the intended protocol, it unofficially acquired the HTTP 0.9 label. The rest, as they say, is history.

From these humble beginnings in 1991, HTTP took on a life of its own and evolved rapidly over the coming years. Let us quickly recap the features of HTTP 0.9:

- Client-server, request-response protocol.
- ASCII protocol, running over a TCP/IP link.
- Designed to transfer hypertext documents (HTML).
- The connection between server and client is closed after every request.

Note

Popular web servers, such as Apache and Nginx, still support the HTTP 0.9 protocol—in part, because there is not much to it! If you are curious, open up a Telnet session and try accessing google.com, or your own favorite site, via HTTP 0.9 and inspect the behavior and the limitations of this early protocol.

HTTP/1.0: Rapid Growth and Informational RFC

The period from 1991 to 1995 is one of rapid coevolution of the HTML specification, a new breed of software known as a "web browser," and the emergence and quick growth of the consumer-oriented public Internet infrastructure.

The Perfect Storm: Internet Boom of the Early 1990s

Building on Tim Berner-Lee's initial browser prototype, a team at the National Center of Supercomputing Applications (NCSA) decided to implement their own version. With that, the first popular browser was born: NCSA Mosaic. One of the programmers on the NCSA team, Marc Andreessen, partnered with Jim Clark to found Mosaic Communications in October 1994. The company was later renamed Netscape, and it shipped Netscape Navigator 1.0 in December 1994. By this point, it was already clear that the World Wide Web was bound to be *much more* than just an academic curiosity.

In fact, that same year the first World Wide Web conference was organized in Geneva, Switzerland, which led to the creation of the World Wide Web Consortium (W3C) to help guide the evolution of

> focus on improving the HTTP protocol. Both of these groups continue to be instrumental to the evolution of the Web.
>
> Finally, to create the perfect storm, CompuServe, AOL, and Prodigy began providing dial-up Internet access to the public within the same 1994–1995 time frame. Riding on this wave of rapid adoption, Netscape made history with a wildly successful IPO on August 9, 1995—the Internet boom had arrived, and everyone wanted a piece of it!

The growing list of desired capabilities of the nascent Web and their use cases on the public Web quickly exposed many of the fundamental limitations of HTTP 0.9: we needed a protocol that could serve more than just hypertext documents, provide richer metadata about the request and the response, enable content negotiation, and more. In turn, the nascent community of web developers responded by producing a large number of experimental HTTP server and client implementations through an ad hoc process: implement, deploy, and see if other people adopt it.

From this period of rapid experimentation, a set of best practices and common patterns began to emerge, and in May 1996 the HTTP Working Group (HTTP-WG) published RFC 1945, which documented the "common usage" of the many HTTP/1.0 implementations found in the wild. Note that this was only an informational RFC: HTTP/1.0 as we know it is not a formal specification or an Internet standard!

Having said that, an example HTTP/1.0 request should look very familiar:

```
$> telnet website.org 80

Connected to xxx.xxx.xxx.xxx

GET /rfc/rfc1945.txt HTTP/1.0 ①
User-Agent: CERN-LineMode/2.15 libwww/2.17b3
Accept: */*

HTTP/1.0 200 OK ②
Content-Type: text/plain
Content-Length: 137582
Expires: Thu, 01 Dec 1997 16:00:00 GMT
Last-Modified: Wed, 1 May 1996 12:45:26 GMT
Server: Apache 0.84

(plain-text response)
(connection closed)
```

① Request line with HTTP version number, followed by request headers

② Response status, followed by response headers some of the key protocol changes:

- Request may consist of multiple newline separated header fields.
- Response object is prefixed with a response status line.
- Response object has its own set of newline separated header fields.
- Response object is not limited to hypertext.
- The connection between server and client is closed after every request.

Both the request and response headers were kept as ASCII encoded, but the response object itself could be of any type: an HTML file, a plain text file, an image, or any other content type. Hence, the "hypertext transfer" part of HTTP became a misnomer not long after its introduction. In reality, HTTP has quickly evolved to become a *hypermedia transport*, but the original name stuck.

In addition to media type negotiation, the RFC also documented a number of other commonly implemented capabilities: content encoding, character set support, multi-part types, authorization, caching, proxy behaviors, date formats, and more.

Note

Almost every server on the Web today can and will still speak HTTP/1.0. Except that, by now, you should know better! Requiring a new TCP connection per request imposes a significant performance penalty on HTTP/1.0; see Three-Way Handshake, followed by Slow-Start.

HTTP/1.1: Internet Standard

The work on turning HTTP into an official IETF Internet standard proceeded in parallel with the documentation effort around HTTP/1.0 and happened over a period of roughly four years: between 1995 and 1999. In fact, the first official HTTP/1.1 standard is defined in RFC 2068, which was officially released in January 1997, roughly six months after the publication of HTTP/1.0. Then, two and a half years later, in June of 1999, a number of improvements and updates were incorporated into the standard and were released as RFC 2616.

The HTTP/1.1 standard resolved a lot of the protocol ambiguities found in earlier versions and introduced a number of critical performance optimizations: keepalive connections, chunked encoding transfers, byte-range requests, additional caching mechanisms, transfer encodings, and request pipelining.

With these capabilities in place, we can now inspect a typical HTTP/1.1 session as performed by any modern HTTP browser and client:

```
Connected to xxx.xxx.xxx.xxx

GET /index.html HTTP/1.1 ①
Host: website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4)... (snip)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: __qca=P0-800083390... (snip)

HTTP/1.1 200 OK ②
Server: nginx/1.0.11
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Via: HTTP/1.1 GWA
Date: Wed, 25 Jul 2012 20:23:35 GMT
Expires: Wed, 25 Jul 2012 20:23:35 GMT
Cache-Control: max-age=0, no-cache
Transfer-Encoding: chunked

100 ③
<!doctype html>
(snip)

100
(snip)

0 ④

GET /favicon.ico HTTP/1.1 ⑤
Host: www.website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4)... (snip)
Accept: */*
Referer: http://website.org/
Connection: close ⑥
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: __qca=P0-800083390... (snip)

HTTP/1.1 200 OK ⑦
Server: nginx/1.0.11
Content-Type: image/x-icon
Content-Length: 3638
Connection: close
Last-Modified: Thu, 19 Jul 2012 17:51:44 GMT
Cache-Control: max-age=315360000
Accept-Ranges: bytes
Via: HTTP/1.1 GWA
Date: Sat, 21 Jul 2012 21:35:22 GMT
Expires: Thu, 31 Dec 2037 23:55:55 GMT
```

```
(icon data)
(connection closed)
```

1. Request for HTML file, with encoding, charset, and cookie metadata
2. Chunked response for original HTML request
3. Number of octets in the chunk expressed as an ASCII hexadecimal number (256 bytes)
4. End of chunked stream response
5. Request for an icon file made on same TCP connection
6. Inform server that the connection will not be reused
7. Icon response, followed by connection close

Phew, there is a lot going on in there! The first and most obvious difference is that we have two object requests, one for an HTML page and one for an image, both delivered over a single connection. This is connection keepalive in action, which allows us to reuse the existing TCP connection for multiple requests to the same host and deliver a much faster end-user experience; see Optimizing for TCP.

To terminate the persistent connection, notice that the second client request sends an explicit `close` token to the server via the `Connection` header. Similarly, the server can notify the client of the intent to close the current TCP connection once the response is transferred. Technically, either side can terminate the TCP connection without such signal at any point, but clients and servers should provide it whenever possible to enable better connection reuse strategies on both sides.

> Note
>
> *HTTP/1.1 changed the semantics of the HTTP protocol to use connection keepalive by default. Meaning, unless told otherwise (via `Connection: close` header), the server should keep the connection open by default.*
>
> *However, this same functionality was also backported to HTTP/1.0 and enabled via the `Connection: Keep-Alive` header. Hence, if you are using HTTP/1.1, technically you don’t need the `Connection: Keep-Alive` header, but many clients choose to provide it nonetheless.*

Additionally, the HTTP/1.1 protocol added content, encoding, character set, and even language negotiation, transfer encoding, caching directives, client cookies, plus a dozen other capabilities that can be negotiated on each request.

dedicated book, and many great ones have been written already. Instead, the previous example serves as a good illustration of both the quick progress and evolution of HTTP, as well as the intricate and complicated dance of every client-server exchange. There is a lot going on in there!

Note

For a good reference on all the inner workings of the HTTP protocol, check out O'Reilly's HTTP: The Definitive Guide by David Gourley and Brian Totty.

HTTP/2: Improving Transport Performance

Since its publication, RFC 2616 has served as a foundation for the unprecedented growth of the Internet: billions of devices of all shapes and sizes, from desktop computers to the tiny web devices in our pockets, speak HTTP every day to deliver news, video, and millions of other web applications we have all come to depend on in our lives.

What began as a simple, one-line protocol for retrieving hypertext quickly evolved into a generic hypermedia transport, and now a decade later can be used to power just about any use case you can imagine. Both the ubiquity of servers that can speak the protocol and the wide availability of clients to consume it means that many applications are now designed and deployed exclusively on top of HTTP.

Need a protocol to control your coffee pot? RFC 2324 has you covered with the Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)—originally an April Fools' Day joke by IETF, and increasingly anything but a joke in our new hyper-connected world.

> *The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol that can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.*
>
> *RFC 2616: HTTP/1.1, June 1999*

The simplicity of the HTTP protocol is what enabled its original adoption and rapid growth. In fact, it is now not unusual to find embedded devices—sensors, actuators, and coffee pots alike—using HTTP as their primary control and data protocols. But under the weight of its own success and as we increasingly continue to migrate our everyday interactions to the Web—social, email, news, and video, and increasingly our entire personal and job workspaces—it has also begun to show signs of stress. Users and web developers alike are now demanding near real-time some modifications.

To meet these new challenges, HTTP must continue to evolve, and hence the HTTPbis working group announced a new initiative for HTTP/2 in early 2012:

> *There is emerging implementation experience and interest in a protocol that retains the semantics of HTTP without the legacy of HTTP/1.x message framing and syntax, which have been identified as hampering performance and encouraging misuse of the underlying transport.*
>
> *The working group will produce a specification of a new expression of HTTP's current semantics in ordered, bi-directional streams. As with HTTP/1.x, the primary target transport is TCP, but it should be possible to use other transports.*
>
> *HTTP/2 charter, January 2012*

The primary focus of HTTP/2 is on improving transport performance and enabling both lower latency and higher throughput. The major version increment sounds like a big step, which it is and will be as far as performance is concerned, but it is important to note that none of the high-level protocol semantics are affected: all HTTP headers, values, and use cases are the same.

Any existing website or application can and will be delivered over HTTP/2 without modification: you do not need to modify your application markup to take advantage of HTTP/2. The HTTP servers will have to speak HTTP/2, but that should be a transparent upgrade for the majority of users. The only difference if the working group meets its goal, should be that our applications are delivered with lower latency and better utilization of the network link!

Having said that, let's not get ahead of ourselves. Before we get to the new HTTP/2 protocol features, it is worth taking a step back and examining our existing deployment and performance best practices for HTTP/1.1. The HTTP/2 working group is making fast progress on the new specification, but even if the final standard was already done and ready, we would still have to support older HTTP/1.1 clients for the foreseeable future—realistically, a decade or more.

« Back to the Table of Contents