**HTTP (Hypertext Transfer Protocol)**

HTTP is essentially a request/response protocol in the client-server computing model, and the primary communication mode of the World Wide Web. The original version, proposed as an application protocol in 1989 by Tim Berners-Lee, was very limited, and quickly modified to support wider browser and server functionality.

**HTTP/0.9** – Experimental Version (1991)

**HTTP/1** – First official Version (1996)

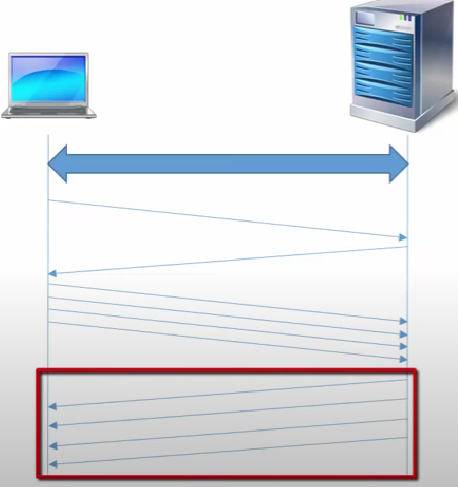
To overcome the severe limitations of HTTP/0.9, browsers and servers modified the protocol independently. Some key protocol changes:

* **Requests** were allowed to consist of **multiple header** fields separated by newline characters.
* The server sent a **response** consisting of a single **status** line.
* A **header** field **was** **added** **to the response**. The response header object consisted of header fields separated by newline characters.
* Servers could respond with **files other than HTML**.

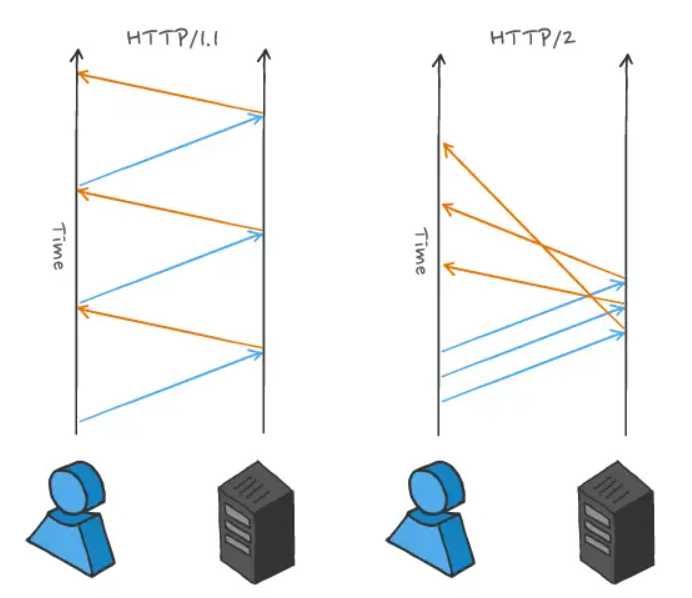
**HTTP/1.1** – Extended Version (1997)

HTTP/1.1 is the most widely supported version in web browsers and servers and its arrival was a big step forward, because it enabled some pretty important optimizations and enhancements, from persistent and pipelined connections, to new request/response header fields. Chief among them are two headers that are the basis for many of the improvements that have helped to enable a more dynamic, realtime web:

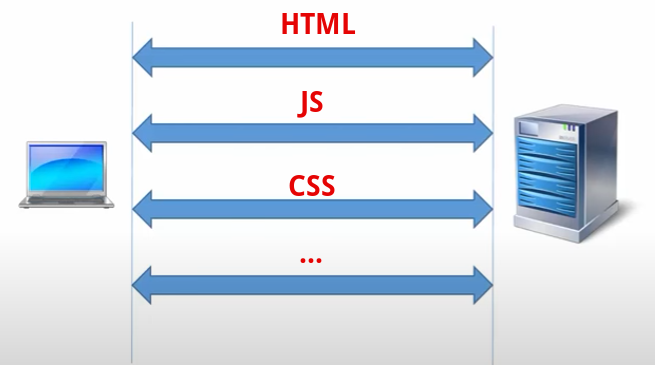
* **The Keep-Alive header**: Used to set up persistent communications between hosts. That means the connection can be reused for more than one request, which reduces request latency perceptibly because the client does not need to re-negotiate the TCP 3-Way-Handshake connection after the first request has been sent. Another positive side effect is that, in general, the connection becomes faster with time due to TCP's slow-start-mechanism. Prior to HTTP/1.1, you had to open a new connection for every single request/response pair
* **The Upgrade header**: Used to upgrade the connection to an enhanced protocol mode (such as WebSockets).
* **Pipelined connections** - is a technique in which clients send multiple requests to the server without waiting for a response. However, the server must respond to the requests in order

****

A large or slow response can break pipelining by blocking the responses behind it. Pipelining was challenging to implement because many intermediaries and servers didn’t process it correctly.



Browser can also open multiple connections for data transferring (first connection for html, second for JS, etc). Inside each connection we can use HTTP pipelines



* **Virtual hosting**
* **Content negotiation, chunked transfer, compression and decompression, transfer encoding, and caching directives**
* **Character set and language tags**
* **Client identification and cookies**
* **Basic authentication, digest authentication, and secure HTTP**

**HTTP/2.0** – Modern Version (2015)

HTTP/2.0 evolved from an experimental protocol – SPDY – which was originally announced by Google in 2009. By 2015, the HTTP Working Group had published HTTP/2.0 as a Proposed Standard, having taken the SPDY specification as its starting point.

At a high level, HTTP/2 was designed to resolve the issues of HTTP/1.1. Let’s take a look at how HTTP/2 works. An important aspect to remember is HTTP/2 is an extension of, not a replacement for HTTP/1.1. It retains the application semantics of HTTP/1.1. Functionality, HTTP methods, status codes, URIs, and header fields remain the same.

**Structure of HTTP/2 messages**

HTTP/2 has a highly structured format with HTTP messages formatted into packets (called frames) with each frame assigned to a stream. HTTP/2 frames have a specific format, including length, declared at the beginning of each frame and several other fields in the frame header.

In many ways, the HTTP frame is similar to a TCP packet. Reading an HTTP/2 frame follows a defined process: the first 24 bits are the length of this packet, followed by 8 bits which define the frame type, and so on.

After the frame header comes the payload. This could be HTTP headers or the body. These also have a specific format, known in advance. An HTTP/2 message can be sent in one or more frames.

In contrast, HTTP/1.1 is an unstructured format consisting of lines of ASCII text. Ultimately, it’s a stream of characters rather than being specifically broken into separate pieces/frames (apart from the lines distinction).

HTTP/1.1 messages (or at least the first HTTP Request/Response line and HTTP Headers) are parsed by reading in one character at a time, until a new line character is reached. This is messy as you don’t know in advance how long each line is.

In HTTP/1.1, the HTTP body’s length is handled slightly differently, as it is typically defined in the Content-Length HTTP header. An HTTP/1.1 message must be sent in its entirety as one continuous stream of data and the connection cannot be used for anything but transmitting that message until it is completed.

Every HTTP/2 connection starts as HTTP/1.1 and the connection upgrades if the client supports HTTP/2. HTTP/2 uses a single TCP connection between the client and the server, which remains open for the duration of the interaction

**HTTP/2 design features**

It was essentially a performance update designed to improve the speed of web communications. The main developments in the context of realtime communication were:.

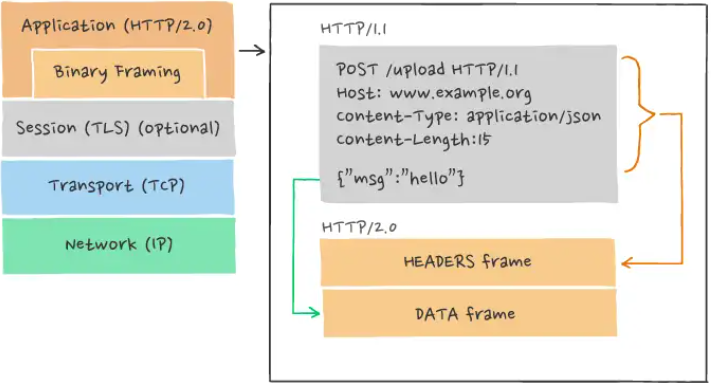
* Replacement of the textual protocol with a **binary one**. A binary framing layer creates an interleaved communication stream
* **Multiplexing**: Rather than transporting data in plaintext format, data is encoded as binary and encapsulated inside frames which can be multiplexed along bidirectional channels known as streams – all over a single TCP connection. This allows for many parallel requests/responses to take place concurrently
* **Server push**: Server push is a performance feature that allows a server to send responses to an HTTP/2-compliant client before the client requests them. This feature is useful when the server knows that the client needs the ‘pushed’ responses to process the original request fully. Pushing resources that are already in the client’s cache waste bandwidth. Wasting bandwidth has an opportunity cost because it could be used for relevant responses. The problem can be tackled by pushing on the first visit only and using “cache-digest”.
* **Flow control** to ensure that streams are non-blocking.
* **Stream prioritization** to tackle problems caused by chopping up and multiplexing of multiple requests and response
* **Header compression** to reduce overhead. HTTP requests contain headers and cookies data, which add performance overhead. HTTP/2 uses the HPACK compression format to compress request and response metadata. Transmitted header fields are encoded using Huffman coding. HTTP/2 requires the client and the server to maintain and update an indexed list of previously-seen header fields. The indexed list is used as a reference to encode previously transmitted values efficiently.

Despite those, and other, steps forward, the explosion of internet traffic driven by the massive uptake of mobile devices has seen HTTP/2.0 struggle to provide a smooth, transparent web browsing experience – especially under the ever-increasing demands of realtime applications and their users.

**Binary framing layer**

The binary framing layer is responsible for all performance enhancements in HTTP/2, setting out the protocol for encapsulation and transfer of messages between the client and the server.

The binary framing layer breaks the communication between the client and server into small chunks and creates an interleaved bidirectional stream of communication. Thanks to the binary framing layer, HTTP/2 uses a single TCP connection that remains open for the duration of the interaction.

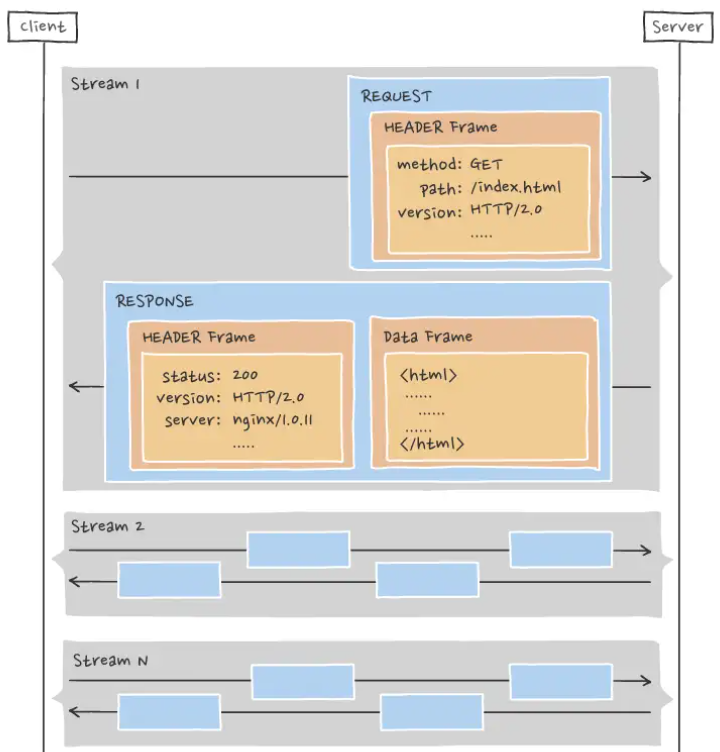


**Frame**. A frame is the smallest unit of communication in HTTP/2. A frame carries a specific type of data, such as HTTP headers or payloads. All frames begin with a fixed 9 octet header. The header field contains, amongst other things, a stream identifier field. A variable-length payload follows the header. The maximum size of a frame payload is 2^24 - 1 octets.

**Message**. A message is a complete HTTP request or response message. A message is made up of one or more frames.

**Stream**. A stream is a bidirectional flow of frames between the client and the server. Some important features of streams are:

* A single HTTP/2 connection can have multiple concurrently open streams. The client and server can send frames from different streams on the connection. Streams can be shared by both the client and server. A stream can also be established and used by a single peer.
* Either endpoint can close the stream.
* The order of frames on a stream is important. Receivers process frames in the order they are received. The order of headers and data frames has semantic significance.
* An integer identifies streams. The identifying integer is assigned by the endpoint which initiated the stream.



**HTTP/2.0 Pros**

* All browsers support HTTP/2 protocol over HTTPS with the installation of an SSL certificate.
* HTTP/2 allows the client to send all requests concurrently over a single TCP connection. Theoretically, the client should receive the resources faster.
* TCP is a reliable, stable connection protocol.

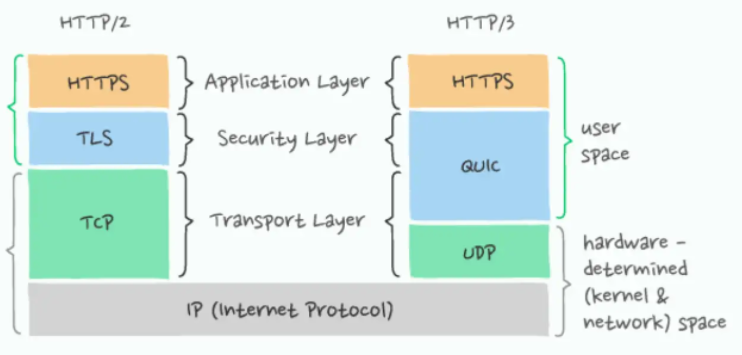
**HTTP/2.0 Cons**

* Concurrent requests can increase the load on the servers. HTTP/2 servers can receive requests in large batches, which can lead to requests timing out. The issue of server load spiking can be solved by inserting a load balancer or a proxy server, which can throttle the requests.
* Server support for HTTP/2 prioritization is not yet mature. Software support is still evolving. Some CDNs or load balancers may not support prioritization properly.
* The HTTP/2 push feature can be tricky to implement correctly.
* HTTP/2 addressed HTTP head-of-line blocking, but TCP-level blocking can still cause problems.

**HTTP/3.0** – Future Version

HTTP/3.0 is a new iteration of HTTP, which has been in development since 2018 and, even though it is still a draft standard at the time of writing (as of October 2020), some browsers are already making use of it.

The aim of HTTP/3 is to provide fast, reliable, and secure web connections across all forms of devices by straightening out the transport-related issues of HTTP/2. To do this, it uses a different transport layer network protocol called QUIC, which runs over the User Datagram Protocol (UDP) instead of TCP, which is used by all previous versions of HTTP



There are already some potential issues with HTTP/3 starting to emerge. For instance:

* **Transport layer ramifications**. Transitioning to HTTP/3 involves not only a change in the application layer, but also a change in the underlying transport layer. Hence, adoption of HTTP/3 is a bit more challenging compared to its predecessor.
* **Reliability and data integrity issues**. UDP is generally suitable for applications where packet loss is acceptable. That's because UDP does not guarantee that your packets will arrive in order. In fact, it does not guarantee that your packets will arrive at all. So if data integrity is important to your use case and you're using HTTP/3, you will have to build mechanisms to ensure message ordering and guaranteed delivery.

**HTTP/3.0 Pros**

* Introduction of new (different) transport protocol QUIC running over UDP means a decrease in latency both theoretically, and, for now, experimentally.
* Because UDP does not perform error checking and correction in the protocol stack, it is suitable for use cases where these are either not required or are performed in the application. This means UDP avoids any associated overhead. UDP is often used in time-sensitive applications, such as realtime systems, which cannot afford to wait for packet retransmission and therefore tolerate some dropped packets.

**HTTP/3.0 Cons**

* Transport layer ramifications. Transitioning to HTTP/3 involves not only a change in the application layer but also a change in the underlying transport layer. Hence, adoption of HTTP/3 is a bit more challenging compared to its predecessor.
* Reliability issues. UDP applications tend to lack reliability, it must be accepted there will be a degree of packet loss, re-ordering, errors, or duplication. It is up to the end-user applications to provide any necessary handshaking, such as real time confirmation that the message has been received.
* HTTP/3 is not yet fully standardized.