**Difference between HTTP1.1 vs HTTP2**

**Introduction**

The Hypertext Transfer Protocol, or HTTP, is an application protocol that has been the standard for communication on the World Wide Web since its invention in 1989.

**Background**

**HTTP/1.1**:

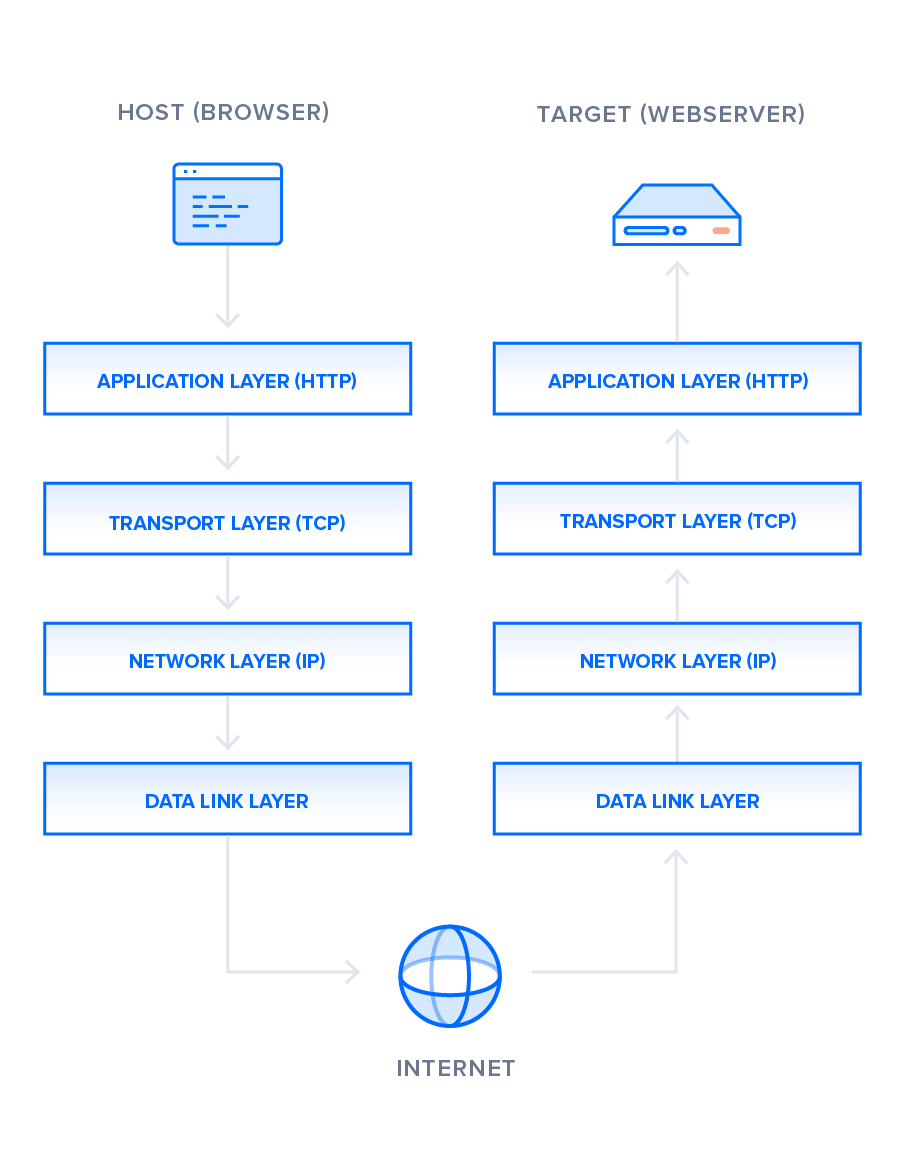
Developed by Timothy Berners-Lee in 1989 as a communication standard for the World Wide Web, HTTP is a top-level application protocol that exchanges information between a client computer and a local or remote web server. In this process, a client sends a text-based request to a server by calling a method like GET or POST. In response, the server sends a resource like an HTML page back to the client.

For example, let’s say you are visiting a website at the domain [www.guvi.com](http://www.guvi.com) When you navigate to this URL, the web browser on your computer sends an HTTP request in the form of a text-based message, similar to the one shown here:

GET /index.html HTTP/1.1

Host: www.guvi.com

This request uses the GET method, which asks for data from the host server listed after Host:. In response to this request, the guvi.com web server returns an HTML page to the requesting client, in addition to any images, stylesheets, or other resources called for in the HTML. Note that not all of the resources are returned to the client in the first call for data. The requests and responses will go back and forth between the server and client until the web browser has received all the resources necessary to render the contents of the HTML page on your screen.

****

**HTTP/2:**

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.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.

Most significant feature that distinguishes HTTP/1.1 and HTTP/2 is the binary framing layer, which can be thought of as a part of the application layer in the internet protocol stack.

**Delivery Models**

HTTP/1.1 and HTTP/2 share semantics, ensuring that the requests and responses traveling between the server and client in both protocols reach their destinations as traditionally formatted messages with headers and bodies, using familiar methods like GET and POST.

HTTP/1.1 transfers these in plain-text messages, HTTP/2 encodes these into binary, allowing for significantly different delivery model possibilities.

**HTTP/1.1 — Pipelining and Head-of-Line Blocking**

The first response that a client receives on an HTTP GET request is often not the fully rendered page. Instead, it contains links to additional resources needed by the requested page. The client discovers that the full rendering of the page requires these additional resources from the server only after it downloads the page. Because of this, the client will have to make additional requests to retrieve these resources. In HTTP/1.0, the client had to break and remake the TCP connection with every new request, a costly affair in terms of both time and resources.

HTTP/1.1 takes care of this problem by introducing persistent connections and pipelining. With persistent connections, HTTP/1.1 assumes that a TCP connection should be kept open unless directly told to close. This allows the client to send multiple requests along the same connection without waiting for a response to each, greatly improving the performance of HTTP/1.1 over HTTP/1.0.

Unfortunately, there is a natural bottleneck to this optimization strategy. Since multiple data packets cannot pass each other when traveling to the same destination, there are situations in which a request at the head of the queue that cannot retrieve its required resource will block all the requests behind it. This is known as head-of-line (HOL) blocking, and is a significant problem with optimizing connection efficiency in HTTP/1.1. Adding separate, parallel TCP connections could alleviate this issue, but there are limits to the number of concurrent TCP connections possible between a client and server, and each new connection requires significant resources.

**HTTP/2 — Advantages of the Binary Framing Layer**

In HTTP/2, the binary framing layer encodes requests/responses and cuts them up into smaller packets of information, greatly increasing the flexibility of data transfer.

Let’s take a closer look at how this works. As opposed to HTTP/1.1, which must make use of multiple TCP connections to lessen the effect of HOL blocking, HTTP/2 establishes a single connection object between the two machines. Within this connection there are multiple streams of data. Each stream consists of multiple messages in the familiar request/response format. Finally, each of these messages split into smaller units called frames:

the communication channel consists of a bunch of binary-encoded frames, each tagged to a particular stream. The identifying tags allow the connection to interleave these frames during transfer and reassemble them at the other end. The interleaved requests and responses can run in parallel without blocking the messages behind them, a process called *multiplexing*. Multiplexing resolves the head-of-line blocking issue in HTTP/1.1 by ensuring that no message has to wait for another to finish. This also means that servers and clients can send concurrent requests and responses, allowing for greater control and more efficient connection management.

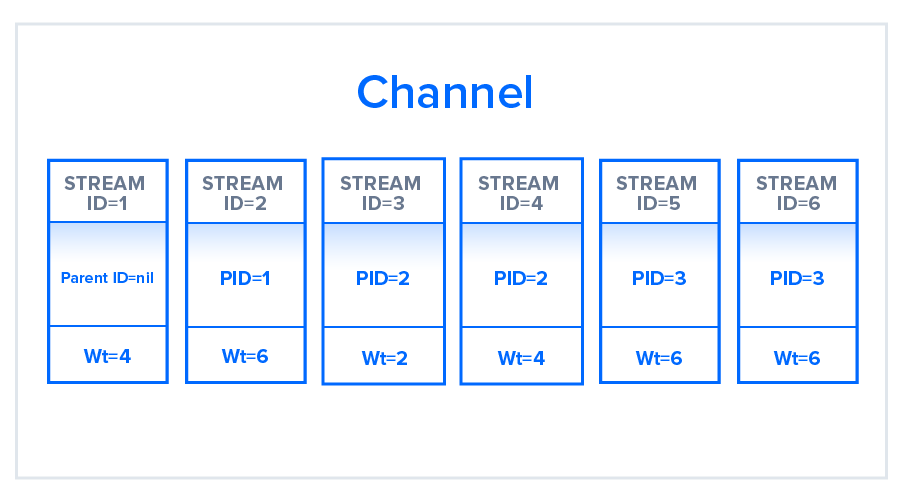
A single TCP connection also improves the performance of the HTTPS protocol, since the client and server can reuse the same secured session for multiple requests/responses. In HTTPS, during the TLS or SSL handshake, both parties agree on the use of a single key throughout the session. If the connection breaks, a new session starts, requiring a newly generated key for further communication. Thus, maintaining a single connection can greatly reduce the resources required for HTTPS performance. Note that, though HTTP/2 specifications do not make it mandatory to use the TLS layer, many major browsers only support HTTP/2 with HTTPS.

Although the multiplexing inherent in the binary framing layer solves certain issues of HTTP/1.1, multiple streams awaiting the same resource can still cause performance issues. The design of HTTP/2 takes this into account,

### **HTTP/2 — Stream Prioritization**

Stream prioritization not only solves the possible issue of requests competing for the same resource, but also allows developers to customize the relative weight of requests to better optimize application performance. In this section, we will break down the process of this prioritization in order to provide better insight into how you can leverage this feature of HTTP/2.

The binary framing layer organizes messages into parallel streams of data. When a client sends concurrent requests to a server, it can prioritize the responses it is requesting by assigning a weight between 1 and 256 to each stream. The higher number indicates higher priority. In addition to this, the client also states each stream’s dependency on another stream by specifying the ID of the stream on which it depends. If the parent identifier is omitted, the stream is considered to be dependent on the root stream



## **Buffer Overflow**

In any TCP connection between two machines, both the client and the server have a certain amount of buffer space available to hold incoming requests that have not yet been processed. These buffers offer flexibility to account for numerous or particularly large requests, in addition to uneven speeds of downstream and upstream connections.There are situations, however, in which a buffer is not enough.

### **HTTP/1.1**

In HTTP/1.1, flow control relies on the underlying TCP connection. When this connection initiates, both client and server establish their buffer sizes using their system default settings. If the receiver’s buffer is partially filled with data, it will tell the sender its receive window, i.e., the amount of available space that remains in its buffer. This receive window is advertised in a signal known as an ACK packet, which is the data packet that the receiver sends to acknowledge that it received the opening signal. If this advertised receive window size is zero, the sender will send no more data until the client clears its internal buffer and then requests to resume data transmission. It is important to note here that using receive windows based on the underlying TCP connection can only implement flow control on either end of the connection.

### **HTTP/2**

HTTP/2 multiplexes streams of data within a single TCP connection. As a result, receive windows on the level of the TCP connection are not sufficient to regulate the delivery of individual streams. HTTP/2 solves this problem by allowing the client and server to implement their own flow controls, rather than relying on the transport layer. 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. This fine-scale flow control can be modified or maintained after the initial connection via a WINDOW\_UPDATE frame.

Since this method controls data flow on the level of the application layer, the flow control mechanism does not have to wait for a signal to reach its ultimate destination before adjusting the receive window. Intermediary nodes can use the flow control settings information to determine their own resource allocations and modify accordingly. In this way, each intermediary server can implement its own custom resource strategy, allowing for greater connection efficiency.

This flexibility in flow control can be advantageous when creating appropriate resource strategies. For example, the client may fetch the first scan of an image, display it to the user, and allow the user to preview it while fetching more critical resources. Once the client fetches these critical resources, the browser will resume the retrieval of the remaining part of the image. Deferring the implementation of flow control to the client and server can thus improve the perceived performance of web applications.

## **Predicting Resource Requests**

### **HTTP/1.1 — Resource Inlining**

The client machine will need to render the page, they can use a technique called resource inlining to include the required resource directly within the HTML document that the server sends in response to the initial GET request. For example, if a client needs a specific CSS file to render a page, inlining that CSS file will provide the client with the needed resource before it asks for it, reducing the total number of requests that the client must send.

A major drawback of resource inlining, then, is that the client cannot separate the resource and the document

### **HTTP/2 — Server Push**

HTTP/2 enables multiple concurrent responses to a client’s initial GET request, a server can send a resource to a client along with the requested HTML page, providing the resource before the client asks for it. This process is called server push

HTTP/2 connection can accomplish the same goal of resource inlining while maintaining the separation between the pushed resource and the document. This means that the client can decide to cache or decline the pushed resource separate from the main HTML document, fixing the major drawback of resource inlining.

process begins when the server sends a PUSH\_PROMISE frame to inform the client that it is going to push a resource. This frame includes only the header of the message, and allows the client to know ahead of time which resource the server will push. If it already has the resource cached, the client can decline the push by sending a RST\_STREAM frame in response. The PUSH\_PROMISE frame also saves the client from sending a duplicate request to the server, since it knows which resources the server is going to push.

## **Compression**

### **HTTP/1.1**

Programs like gzip have long been used to compress the data sent in HTTP messages, especially to decrease the size of CSS and JavaScript files. The header component of a message, however, is always sent as plain text. Although each header is quite small, the burden of this uncompressed data weighs heavier and heavier on the connection as more requests are made, particularly penalizing complicated, API-heavy web applications that require many different resources and thus many different resource requests. Additionally, the use of cookies can sometimes make headers much larger, increasing the need for some kind of compression.

### **HTTP/2**

It ability to use the binary framing layer to exhibit greater control over finer detail. The same is true when it comes to header compression. HTTP/2 can split headers from their data, resulting in a header frame and a data frame. The HTTP/2-specific compression program [HPACK](https://tools.ietf.org/html/draft-ietf-httpbis-header-compression-12) can then compress this header frame. This algorithm can encode the header metadata using Huffman coding, thereby greatly decreasing its size. Additionally, HPACK can keep track of previously conveyed metadata fields and further compress them according to a dynamically altered index shared between the client and the server.