**HTTP/1.1 :**

* 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.
* HTTP/1.1, which keeps all requests and responses in plain text format,
* 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.
* 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.
* In HTTP/1.1, if the developer knows in advance which additional resources 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. A finer level of control is needed to optimize the connection, a need that HTTP/2 seeks to meet with server push.

**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.
* HTTP/2 uses the binary framing layer to encapsulate all messages in binary format, while still maintaining HTTP semantics, such as verbs, methods, and headers.
* 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.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. At the most granular level, 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. Since multiplexing allows the client to construct multiple streams in parallel, these streams only need to make use of a single TCP connection. Having a single persistent connection per origin improves upon HTTP/1.1 by reducing the memory and processing footprint throughout the network. This results in better network and bandwidth utilization and thus decreases the overall operational cost.
* 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.
* Since 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. In this way, an 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.

<https://www.digitalocean.com/community/tutorials/http-1-1-vs-http-2-what-s-the-difference>