**differences between HTTP 1.1 and HTTP 2**

\* Binary framing layer:

One of the most significant features 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. As opposed to HTTP/1.1,

which keeps all requests and responses in plain text format, 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. An application level API would still create messages in the conventional HTTP formats,

but the underlying layer would then convert these messages into binary.

This ensures that web applications created before HTTP/2 can continue functioning as normal when interacting with the new protocol.

**\*Delivery models**

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

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.

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

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.

Multiplexing:

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.

HTTP/2 — Stream Prioritization

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.The server uses this information to create a dependency tree,

which allows the server to determine the order in which the requests will retrieve their data.

**\*Buffer over-flow**

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.

Because HTTP/1.1 relies on the transport layer to avoid buffer overflow, each new TCP connection requires a separate flow control mechanism.

HTTP/2, however, multiplexes streams within a single TCP connection, and will have to implement flow control in a different manner.

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.

**Predicting Resource Requests**

HTTP/1.1 — Resource Inlining

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 — Server Push

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.

**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.

In order to solve this bottleneck, HTTP/2 uses HPACK compression to shrink the size of headers, a topic discussed further in the next section.

HTTP/2

One of the themes that has come up again and again in HTTP/2 is its 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 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.