**TASK 1**

1. **Difference between HTTP1.1 vs HTTP2**

HTTP1.1 and HTTP2 ensures 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. **But while HTTP/1.1 transfers these in plain-text messages, HTTP/2 encodes these into binary, allowing for significantly different delivery model possibilities.**

1. **Delivery Models**

**HTTP1.1**

* HTTP/1.1 uses 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. 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.**

HTTP/1.1, which makes use of multiple TCP connections to lessen the effect of HOL blocking. 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**

* **Multiplexing**-HTTP/2 establishes a single connection object between the two machines. 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.
* **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.

1. **Buffer Overflow**

* **HTTP1.1**

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

* **HTTP2**

HTTP/2 solves this problem by allowing the client and server to implement their own flow controls, rather than relying on the transport layer.

1. **Predicting Resource requests**

* **HTTP1.1**

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

* **HTTP2**

**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.

1. **Compression**

* **HTTP1.1**

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.

* **HTTP2**

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.

1. **Http version history**

|  |  |
| --- | --- |
| Year | HTTP Version |
| 1991 | 0.9 |
| 1996 | 1.0 |
| 1997 | 1.1 |
| 2015 | 2.0 |
| Draft (2020) | 3.0 |

1. **Difference between Browser JS and Node JS**

**Application**

Browser.js is used for frontend while Node.js is used for backend applications.

**System Access**

Node.js has full system access. It can read and write directly to the file system like any other application that also concludes that we can write complete software using Node.js while Browser.js is sandboxed for the safety purposes and have access limited to the browser.

**Running Engine**

Browser.js runs any engine like Spider monkey (Firefox), JavaScript Core (Safari), V8 (Google Chrome) accordingly to the browser while Node.js runs in a V8 engine which is mainly used by google chrome.

**Headless**

Node.js is headless without any GUI while Bowsers are not headless.

**Modularity**

In Node.js everything is a module, it is mandatory to keep everything inside a module while moduling is not mandatory for browser JavaScript.

1. **What happens when you type a URL in the address bar?**

When a URL is requested in browser (User interface), Networking module in web browser issues Http call to look for the resource in internet and bring backs the response. After obtaining the response the respective files are processed by Rendering Engine which holds HTML, CSS and JS parser. This is given back to the client/user in User Interface via Browser engine.