Question No-1

Difference between HTTP1.1 Vs HTTP2

**HTTP1.1:**

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 in lining 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, in lining 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.

But there are a few problems with resource in lining. Including the resource in the HTML document is a viable solution for smaller, text-based resources, but larger files in non-text formats can greatly increase the size of the HTML document, which can ultimately decrease the connection speed and nullify the original advantage gained from using this technique. Also, since the in lined resources are no longer separate from the HTML document, there is no mechanism for the client to decline resources that it already has, or to place a resource in its cache. If multiple pages require the resource, each new HTML document will have the same resource in lined in its code, leading to larger HTML documents and longer load times than if the resource were simply cached in the beginning.

A major drawback of resource in lining, 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.

**HTTP2:**

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 in lining 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 in lining.

In HTTP/2, this process begins when the server sends a 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 frame. The frame also saves the client from sending a duplicate request to the server, since it knows which resources the server is going to push.

It is important to note here that the emphasis of server push is client control. If a client needed to adjust the priority of server push, or even disable it, it could at any time send a setting frame to modify this HTTP/2 feature.

Question No-2

HTTP Version History

The Hypertext Transfer Protocol (HTTP) is one of the most ubiquitous and widely adopted application protocols on the Internet: it is the common language between clients and servers, enabling the modern web. From its simple beginnings as a single keyword and document path, it has become the protocol of choice not just for browsers, but for virtually every Internet-connected software and hardware application.

We will take a brief historical tour of the evolution of the HTTP protocol. A full discussion of the varying HTTP semantics is outside the scope of this lesson, but an understanding of the key design changes of HTTP, and the motivations behind each, will give us the necessary background for our discussions on HTTP performance, especially in the context of the many upcoming improvements in HTTP/2.

**HTTP 0.9: The One-Line Protocol**

The original HTTP proposal by Tim Berners-Lee was designed with simplicity in mind as to help with the adoption of his other nascent idea: the World Wide Web. The strategy appears to have worked: aspiring protocol designers, take note.

In 1991, Berners-Lee outlined the motivation for the new protocol and listed several high-level design goals: file transfer functionality, ability to request an index search of a hypertext archive, format negotiation, and an ability to refer the client to another server. To prove the theory in action, a simple prototype was built, which implemented a small subset of the proposed functionality:

* Client request is a single ASCII character string.
* Client request is terminated by a carriage return (CRLF).
* Server response is an ASCII character stream.
* Server response is a hypertext markup language (HTML).
* Connection is terminated after the document transfer is complete.

However, even that sounds a lot more complicated than it really is. What these rules enable is an extremely simple, Telnet-friendly protocol, which some web servers support to this very day:

The request consists of a single line: GET method and the path of the requested document. The response is a single hypertext document—no headers or any other metadata, just the HTML. It really couldn’t get any simpler. Further, since the previous interaction is a subset of the intended protocol, it unofficially acquired the HTTP 0.9 label. The rest, as they say, is history.

From these humble beginnings in 1991, HTTP took on a life of its own and evolved rapidly over the coming years. Let us quickly recap the features of HTTP 0.9:

* Client-server, request-response protocol.
* ASCII protocol, running over a TCP/IP link.
* Designed to transfer hypertext documents (HTML).
* The connection between server and client is closed after every request

**HTTP/1.0: Rapid Growth and Informational RFC**

The period from 1991 to 1995 is one of rapid coevolution of the HTML specification, a new breed of software known as a “web browser,” and the emergence and quick growth of the consumer-oriented public Internet infrastructure.

Building on Tim Berner-Lee’s initial browser prototype, a team at the National Centre of Supercomputing Applications (NCSA) decided to implement their own version. With that, the first popular browser was born: NCSA Mosaic. One of the programmers on the NCSA team, Marc Andreessen, partnered with Jim Clark to found Mosaic Communications in October 1994. The company was later renamed Netscape, and it shipped Netscape Navigator 1.0 in December 1994. By this point, it was already clear that the World Wide Web was bound to be much more than just an academic curiosity.

In fact, that same year the first World Wide Web conference was organized in Geneva, Switzerland, which led to the creation of the World Wide Web Consortium (W3C) to help guide the evolution of HTML. Similarly, a parallel HTTP Working Group (HTTP-WG) was established within the IETF to focus on improving the HTTP protocol. Both of these groups continue to be instrumental to the evolution of the Web.

Finally, to create the perfect storm, CompuServe, AOL, and Prodigy began providing dial-up Internet access to the public within the same 1994–1995time frame. Riding on this wave of rapid adoption, Netscape made history with a wildly successful IPO on August 9, 1995—the Internet boom had arrived, and everyone wanted a piece of it!

The growing list of desired capabilities of the nascent Web and their use cases on the public Web quickly exposed many of the fundamental limitations of HTTP 0.9: we needed a protocol that could serve more than just hypertext documents, provide richer metadata about the request and the response, enable content negotiation, and more. In turn, the nascent community of web developers responded by producing a large number of experimental HTTP server and client implementations through an ad hoc process: implement, deploy, and see if other people adopt it.

From this period of rapid experimentation, a set of best practices and common patterns began to emerge, and in May 1996 the HTTP Working Group (HTTP-WG) published RFC 1945, which documented the “common usage” of the many HTTP/1.0 implementations found in the wild. Note that this was only an informational RFC: HTTP/1.0 as we know it is not a formal specification or an Internet standard!

* Request line with HTTP version number, followed by request headers
* Response status, followed by response headers
* The preceding exchange is not an exhaustive list of HTTP/1.0 capabilities, but it does illustrate some of the key protocol changes:
* Request may consist of multiple new line separated header fields.
* Response object is prefixed with a response status line.
* Response object has its own set of new line separated header fields.
* Response object is not limited to hypertext.

The connection between server and client is closed after every request.

Both the request and response headers were kept as ASCII encoded, but the response object itself could be of any type: an HTML file, a plain text file, an image, or any other content type. Hence, the “hypertext transfer” part of HTTP became a misnomer not long after its introduction. In reality, HTTP has quickly evolved to become a hypermedia transport, but the original name stuck.

In addition to media type negotiation, the RFC also documented a number of other commonly implemented capabilities: content encoding, character set support, multi-part types, authorization, caching, proxy behaviour, date formats, and more.

HTTP/1.1: Internet Standard

The work on turning HTTP into an official IETF Internet standard proceeded in parallel with the documentation effort around HTTP/1.0 and happened over a period of roughly four years: between 1995 and 1999. In fact, the first official HTTP/1.1 standard is defined in RFC 2068, which was officially released in January 1997, roughly six months after the publication of HTTP/1.0. Then, two and a half years later, in June of 1999, a number of improvements and updates were incorporated into the standard and were released as RFC 2616.

The HTTP/1.1 standard resolved a lot of the protocol ambiguities found in earlier versions and introduced a number of critical performance optimizations: keepalive connections, chunked encoding transfers, byte-range requests, additional caching mechanisms, transfer encodings, and request pipelining.

With these capabilities in place, we can now inspect a typical HTTP/1.1 session as performed by any modern HTTP browser and client:

* Request for HTML file, with encoding, charset, and cookie metadata
* Chunked response for original HTML request
* Number of octets in the chunk expressed as an ASCII hexadecimal number (256 bytes)
* End of chunked stream response
* Request for an icon file made on same TCP connection
* Inform server that the connection will not be reused
* Icon response, followed by connection close

Phew, there is a lot going on in there! The first and most obvious difference is that we have two object requests, one for an HTML page and one for an image, both delivered over a single connection. This is connection keepalive in action, which allows us to reuse the existing TCP connection for multiple requests to the same host and deliver a much faster end-user experience.

To terminate the persistent connection, notice that the second client request sends an explicit close token to the server via the Connection header. Similarly, the server can notify the client of the intent to close the current TCP connection once the response is transferred. Technically, either side can terminate the TCP connection without such signal at any point, but clients and servers should provide it whenever possible to enable better connection reuse strategies on both sides.

HTTP/2: Improving Transport Performance

Since its publication, RFC 2616 has served as a foundation for the unprecedented growth of the Internet: billions of devices of all shapes and sizes, from desktop computers to the tiny web devices in our pockets, speak HTTP every day to deliver news, video, and millions of other web applications we have all come to depend on in our lives.

What began as a simple, one-line protocol for retrieving hypertext quickly evolved into a generic hypermedia transport, and now a decade later can be used to power just about any use case you can imagine. Both the ubiquity of servers that can speak the protocol and the wide availability of clients to consume it means that many applications are now designed and deployed exclusively on top of HTTP.

Need a protocol to control your coffee pot? RFC 2324 has you covered with the Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)—originally an April Fools’ Day joke by IETF, and increasingly anything but a joke in our new hyper-connected world.

*The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol that can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.*

The simplicity of the HTTP protocol is what enabled its original adoption and rapid growth. In fact, it is now not unusual to find embedded devices—sensors, actuators, and coffee pots alike—using HTTP as their primary control and data protocols. But under the weight of its own success and as we increasingly continue to migrate our everyday interactions to the Web—social, email, news, and video, and increasingly our entire personal and job workspaces—it has also begun to show signs of stress. Users and web developers alike are now demanding near real-time responsiveness and protocol performance from HTTP/1.1, which it simply cannot meet without some modifications.

To meet these new challenges, HTTP must continue to evolve, and hence the HTTP bis working group announced a new initiative for HTTP/2 in early 2012:

*There is emerging implementation experience and interest in a protocol that retains the semantics of HTTP without the legacy of HTTP/1.x message framing and syntax, which have been identified as hampering performance and encouraging misuse of the underlying transport.*

*The working group will produce a specification of a new expression of HTTP’s current semantics in ordered, bi-directional streams. As with HTTP/1.x, the primary target transport is TCP, but it should be possible to use other transports.*

Question No-3

List 5 Difference between Browser JS(Console) vs Node JS

| S.no | JavaScript | NodeJS |
| --- | --- | --- |
| 1. | JavaScript is a programming language that is used for writing scripts on the website. | NodeJS is a JavaScript runtime environment. |
| 2. | JavaScript can only be run in the browsers. | NodeJS code can be run outside the browser. |
| 3. | It is basically used on the client-side. | It is mostly used on the server-side. |
| 4. | JavaScript is capable enough to add HTML and play with the DOM. | Nodejs does not have capability to add HTML tags. |
| 5. | JavaScript can run in any browser engine as like JS core in safari and Spider monkey in Firefox. | Nodejs can only run in V8 engine of google chrome. |
| 6. | JavaScript is used in frontend development. | Nodejs is used in server-side development. |
| 7. | Some of the JavaScript frameworks are Ramada JS, Type JS, etc. | Some of the Nodejs modules are Lod ash, express etc. These modules are to be imported from npm. |
| 8. | It is the upgraded version of ECMA script that uses Chrome’s V8 engine written in C++. | Nodejs is written in C, C++ and JavaScript. |

Question No-4

what happens when you type a URL in the address bar in the browser?

If you are in any technical profession, I am sure someone at some point has asked you this question. Whether you are an engineer, developer, marketer, or even in sales, it is always good to have a basic understanding of what is going on behind our browsers and how information is transferred to our computers via the internet.

Let’s imagine that you want to access maps.google.com to check the exact time it would take for you to get to your dinner reservation from work.

**1. You type maps.google.com into the address bar of your browser.**

**2. The browser checks the cache for a DNS record to find the corresponding IP address of maps.google.com.**

DNS(Domain Name System) is a database that maintains the name of the website (URL) and the particular IP address it links to. Every single URL on the internet has a unique IP address assigned to it. The IP address belongs to the computer which hosts the server of the website we are requesting to access. For example, [www.google.com](http://www.google.com/) has an IP address of 209.85.227.104. If you’d like, you can reach [www.google.com](http://www.google.com/) by typing [http://209.85.227.104](http://209.85.227.104/) on your browser. DNS is a list of URLs, and their IP addresses, like how a phone book is a list of names and their corresponding phone numbers.

The primary purpose of DNS is human-friendly navigation. You can easily access a website by typing the correct IP address for it on your browser, but imagine having to remember different sets of numbers for all the sites we regularly access? Therefore, it is easier to remember the name of the website using a URL and let DNS do the work for us by mapping it to the correct IP.

To find the DNS record, the browser checks four caches.

● First, it checks the browser cache. The browser maintains a repository of DNS records for a fixed duration for websites you have previously visited. So, it is the first place to run a DNS query.

● Second, the browser checks the OS cache. If it is not in the browser cache, the browser will make a system call (i.e*.,*get host name on Windows) to your underlying computer OS to fetch the record since the OS also maintains a cache of DNS records.

● Third, it checks the router cache. If it’s not on your computer, the browser will communicate with the router that maintains its’ own cache of DNS records.

● Fourth, it checks the ISP cache. If all steps fail, the browser will move on to the ISP. Your ISP maintains its’ own DNS server, which includes a cache of DNS records, which the browser would check with the last hope of finding your requested URL.

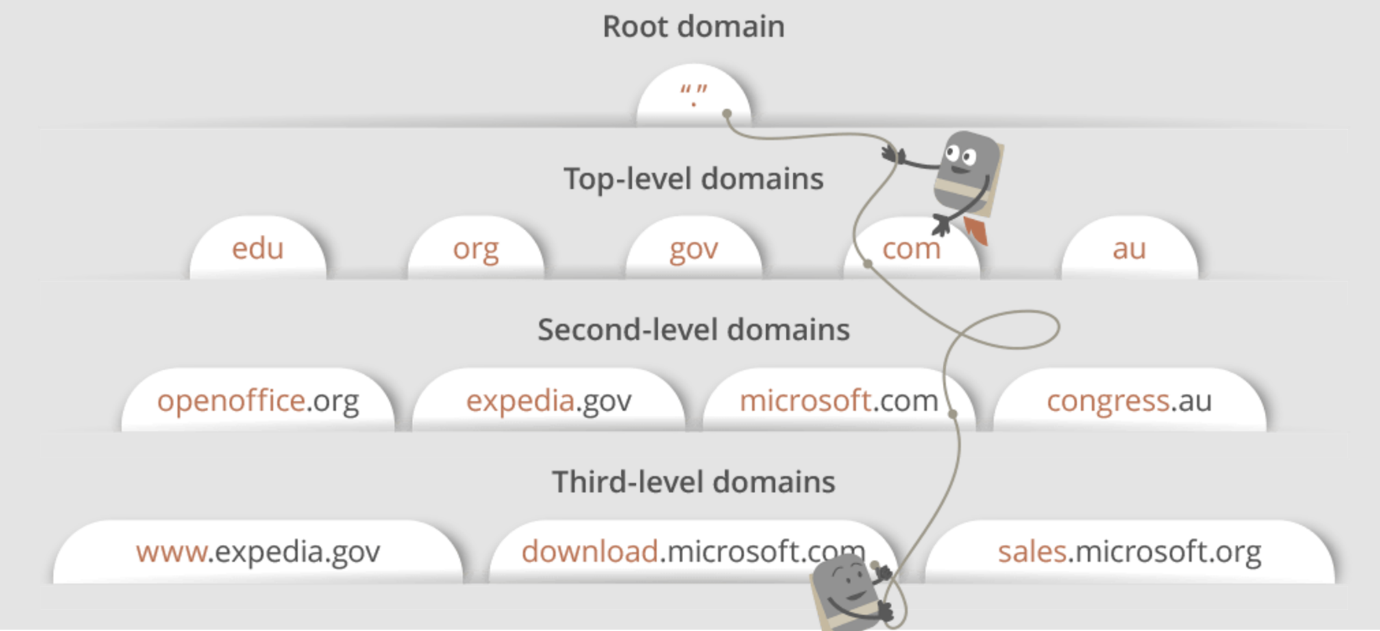
You may wonder why there are so many caches maintained at so many levels. Although our information being cached somewhere doesn’t make us feel very comfortable when it comes to privacy, caches are essential for regulating network traffic and improving data transfer times.

**3. If the requested URL is not in the cache, ISP’s DNS server initiates a DNS query to find the IP address of the server that hosts maps.google.com.**

As mentioned earlier, for my computer to connect with the server that hosts maps.google.com, I need the IP address of maps.google.com. The purpose of a DNS query is to search multiple DNS servers on the internet until it finds the correct IP address for the website. This type of search is called a recursive search since the search will repeatedly continue from a DNS server to a DNS server until it either finds the IP address we need or returns an error response saying it was unable to find it.

In this situation, we would call the ISP’s DNS server a DNS re-Cursor whose responsibility is to find the proper IP address of the intended domain name by asking other DNS servers on the internet for an answer. The other DNS servers are called name servers since they perform a DNS search based on the domain architecture of the website domain name.

Without further confusing you, I’d like to use the following diagram to explain the domain architecture.



Many website URLs we encounter today contain a third-level domain, a second-level domain, and a top-level domain. Each of these levels contains their own name server, which is queried during the DNS lookup process.

For maps.google.com, first, the DNS re-cursor will contact the root name server. The root name server will redirect it to the **.com** domain name server. **.com** name server will redirect it to the **google.com** name server. The **google.com** name server will find the matching IP address for maps.google.com in its’ DNS records and return it to your DNS re-cursor, which will send it back to your browser.

These requests are sent using small data packets that contain information such as the content of the request and the IP address it is destined for (IP address of the DNS re-cursor). These packets travel through multiple networking equipment between the client and the server before it reaches the correct DNS server. This equipment uses routing tables to figure out which way is the fastest possible way for the packet to reach its’ destination. If these packets get lost, you’ll get a request failed error. Otherwise, they will reach the correct DNS server, grab the correct IP address, and come back to your browser.

**4. The browser initiates a TCP connection with the server.**

Once the browser receives the correct IP address, it will build a connection with the server that matches the IP address to transfer information. Browsers use internet protocols to build such connections. There are several different internet protocols that can be used, but TCP is the most common protocol used for many types of HTTP requests.

To transfer data packets between your computer(client) and the server, it is important to have a TCP connection established. This connection is established using a process called the TCP/IP three-way handshake. This is a three-step process where the client and the server exchange SYN (synchronize) and ACK (acknowledge) messages to establish a connection.

1. The client machine sends a SYN packet to the server over the internet, asking if it is open for new connections.

2. If the server has open ports that can accept and initiate new connections, it’ll respond with an Acknowledgment of the SYN packet using a SYN/ACK packet.

3. The client will receive the SYN/ACK packet from the server and will Acknowledge it by sending an ACK packet.

Then a TCP connection is established for data transmission!

**5. The browser sends an HTTP request to the webserver.**

Once the TCP connection is established, it is time to start transferring data! The browser will send a GET request asking for maps.google.com web page. If you’re entering credentials or submitting a form, this could be a POST request. This request will also contain additional information such as browser identification (User-Agent header), types of requests that it will accept (Accept header), and connection headers asking it to keep the TCP connection alive for additional requests. It will also pass information taken from cookies the browser has in store for this domain.

(If you’re curious about what’s going on behind the scenes, you can use tools such as Firebug to take a look at HTTP requests. It is always fun to see the information passed between clients and servers without us knowing).

**6. The server handles the request and sends back a response.**

The server contains a webserver (i.e., Apache, IIS) that receives the request from the browser and passes it to a request handler to read and generate a response. The request handler is a program (written in ASP.NET, PHP, Ruby, etc.) that reads the request, its’ headers, and cookies to check what is being requested and also update the information on the server if needed. Then it will assemble a response in a particular format (JSON, XML, HTML).

**7. The server sends out an HTTP response.**

The server response contains the web page you requested as well as the status code, compression type (Content-Encoding), how to cache the page (Cache-Control), any cookies to set, privacy information, etc.

If you look at the above response, the first line shows a status code. This is quite important as it tells us the status of the response. There are five types of statuses detailed using a numerical code.

● 1xx indicates an informational message only

● 2xx indicates success of some kind

● 3xx redirects the client to another URL

● 4xx indicates an error on the client’s part

● 5xx indicates an error on the server’s part

So, if you encountered an error, you can take a look at the HTTP response to check what type of status code you have received.