**What happens when...**

This repository is an attempt to answer the age-old interview question "What happens when you type google.com into your browser's address box and press enter?"

Except instead of the usual story, we're going to try to answer this question in as much detail as possible. No skipping out on anything.

This is a collaborative process, so dig in and try to help out! There are tons of details missing, just waiting for you to add them! So send us a pull request, please!

This is all licensed under the terms of the [Creative Commons Zero](https://creativecommons.org/publicdomain/zero/1.0/) license.

Read this in [简体中文](https://github.com/skyline75489/what-happens-when-zh_CN) (simplified Chinese), [日本語](https://github.com/tettttsuo/what-happens-when-JA) (Japanese), [한국어](https://github.com/SantonyChoi/what-happens-when-KR) (Korean) and [Spanish](https://github.com/gonzaleztroyano/what-happens-when-ES). NOTE: these have not been reviewed by the alex/what-happens-when maintainers.

**Table of Contents**

* [The "g" key is pressed](https://github.com/alex/what-happens-when?tab=readme-ov-file#the-g-key-is-pressed)
* [The "enter" key bottoms out](https://github.com/alex/what-happens-when?tab=readme-ov-file#the-enter-key-bottoms-out)
* [Interrupt fires [NOT for USB keyboards]](https://github.com/alex/what-happens-when?tab=readme-ov-file#interrupt-fires-not-for-usb-keyboards)
* [(On Windows) A WM\_KEYDOWN message is sent to the app](https://github.com/alex/what-happens-when?tab=readme-ov-file#on-windows-a-wm-keydown-message-is-sent-to-the-app)
* [(On OS X) A KeyDown NSEvent is sent to the app](https://github.com/alex/what-happens-when?tab=readme-ov-file#on-os-x-a-keydown-nsevent-is-sent-to-the-app)
* [(On GNU/Linux) the Xorg server listens for keycodes](https://github.com/alex/what-happens-when?tab=readme-ov-file#on-gnu-linux-the-xorg-server-listens-for-keycodes)
* [Parse URL](https://github.com/alex/what-happens-when?tab=readme-ov-file#parse-url)
* [Is it a URL or a search term?](https://github.com/alex/what-happens-when?tab=readme-ov-file#is-it-a-url-or-a-search-term)
* [Convert non-ASCII Unicode characters in the hostname](https://github.com/alex/what-happens-when?tab=readme-ov-file#convert-non-ascii-unicode-characters-in-the-hostname)
* [Check HSTS list](https://github.com/alex/what-happens-when?tab=readme-ov-file#check-hsts-list)
* [DNS lookup](https://github.com/alex/what-happens-when?tab=readme-ov-file#dns-lookup)
* [ARP process](https://github.com/alex/what-happens-when?tab=readme-ov-file#arp-process)
* [Opening of a socket](https://github.com/alex/what-happens-when?tab=readme-ov-file#opening-of-a-socket)
* [TLS handshake](https://github.com/alex/what-happens-when?tab=readme-ov-file#tls-handshake)
* [If a packet is dropped](https://github.com/alex/what-happens-when?tab=readme-ov-file#if-a-packet-is-dropped)
* [HTTP protocol](https://github.com/alex/what-happens-when?tab=readme-ov-file#http-protocol)
* [HTTP Server Request Handle](https://github.com/alex/what-happens-when?tab=readme-ov-file#http-server-request-handle)
* [Behind the scenes of the Browser](https://github.com/alex/what-happens-when?tab=readme-ov-file#behind-the-scenes-of-the-browser)
* [Browser](https://github.com/alex/what-happens-when?tab=readme-ov-file#browser)
* [HTML parsing](https://github.com/alex/what-happens-when?tab=readme-ov-file#html-parsing)
* [CSS interpretation](https://github.com/alex/what-happens-when?tab=readme-ov-file#css-interpretation)
* [Page Rendering](https://github.com/alex/what-happens-when?tab=readme-ov-file#page-rendering)
* [GPU Rendering](https://github.com/alex/what-happens-when?tab=readme-ov-file#gpu-rendering)
* [Window Server](https://github.com/alex/what-happens-when?tab=readme-ov-file#window-server)
* [Post-rendering and user-induced execution](https://github.com/alex/what-happens-when?tab=readme-ov-file#post-rendering-and-user-induced-execution)

**The "g" key is pressed**

The following sections explain the physical keyboard actions and the OS interrupts. When you press the key "g" the browser receives the event and the auto-complete functions kick in. Depending on your browser's algorithm and if you are in private/incognito mode or not various suggestions will be presented to you in the dropdown below the URL bar. Most of these algorithms sort and prioritize results based on search history, bookmarks, cookies, and popular searches from the internet as a whole. As you are typing "google.com" many blocks of code run and the suggestions will be refined with each keypress. It may even suggest "google.com" before you finish typing it.

**The "enter" key bottoms out**

To pick a zero point, let's choose the Enter key on the keyboard hitting the bottom of its range. At this point, an electrical circuit specific to the enter key is closed (either directly or capacitively). This allows a small amount of current to flow into the logic circuitry of the keyboard, which scans the state of each key switch, debounces the electrical noise of the rapid intermittent closure of the switch, and converts it to a keycode integer, in this case 13. The keyboard controller then encodes the keycode for transport to the computer. This is now almost universally over a Universal Serial Bus (USB) or Bluetooth connection, but historically has been over PS/2 or ADB connections.

*In the case of the USB keyboard:*

* The USB circuitry of the keyboard is powered by the 5V supply provided over pin 1 from the computer's USB host controller.
* The keycode generated is stored by internal keyboard circuitry memory in a register called "endpoint".
* The host USB controller polls that "endpoint" every ~10ms (minimum value declared by the keyboard), so it gets the keycode value stored on it.
* This value goes to the USB SIE (Serial Interface Engine) to be converted in one or more USB packets that follow the low-level USB protocol.
* Those packets are sent by a differential electrical signal over D+ and D- pins (the middle 2) at a maximum speed of 1.5 Mb/s, as an HID (Human Interface Device) device is always declared to be a "low-speed device" (USB 2.0 compliance).
* This serial signal is then decoded at the computer's host USB controller, and interpreted by the computer's Human Interface Device (HID) universal keyboard device driver. The value of the key is then passed into the operating system's hardware abstraction layer.

*In the case of Virtual Keyboard (as in touch screen devices):*

* When the user puts their finger on a modern capacitive touch screen, a tiny amount of current gets transferred to the finger. This completes the circuit through the electrostatic field of the conductive layer and creates a voltage drop at that point on the screen. The screen controller then raises an interrupt reporting the coordinate of the keypress.
* Then the mobile OS notifies the currently focused application of a press event in one of its GUI elements (which now is the virtual keyboard application buttons).
* The virtual keyboard can now raise a software interrupt for sending a 'key pressed' message back to the OS.
* This interrupt notifies the currently focused application of a 'key pressed' event.

**Interrupt fires [NOT for USB keyboards]**

The keyboard sends signals on its interrupt request line (IRQ), which is mapped to an interrupt vector (integer) by the interrupt controller. The CPU uses the Interrupt Descriptor Table (IDT) to map the interrupt vectors to functions (interrupt handlers) which are supplied by the kernel. When an interrupt arrives, the CPU indexes the IDT with the interrupt vector and runs the appropriate handler. Thus, the kernel is entered.

**(On Windows) A WM\_KEYDOWN message is sent to the app**

The HID transport passes the key down event to the KBDHID.sys driver which converts the HID usage into a scancode. In this case, the scan code is VK\_RETURN (0x0D). The KBDHID.sys driver interfaces with the KBDCLASS.sys (keyboard class driver). This driver is responsible for handling all keyboard and keypad input in a secure manner. It then calls into Win32K.sys (after potentially passing the message through 3rd party keyboard filters that are installed). This all happens in kernel mode.

Win32K.sys figures out what window is the active window through the GetForegroundWindow() API. This API provides the window handle of the browser's address box. The main Windows "message pump" then calls SendMessage(hWnd, WM\_KEYDOWN, VK\_RETURN, lParam). lParam is a bitmask that indicates further information about the keypress: repeat count (0 in this case), the actual scan code (can be OEM dependent, but generally wouldn't be for VK\_RETURN), whether extended keys (e.g. alt, shift, ctrl) were also pressed (they weren't), and some other state.

The Windows SendMessage API is a straightforward function that adds the message to a queue for the particular window handle (hWnd). Later, the main message processing function (called a WindowProc) assigned to the hWnd is called in order to process each message in the queue.

The window (hWnd) that is active is actually an edit control and the WindowProc in this case has a message handler for WM\_KEYDOWN messages. This code looks within the 3rd parameter that was passed to SendMessage (wParam) and, because it is VK\_RETURN knows the user has hit the ENTER key.

**(On OS X) A KeyDown NSEvent is sent to the app**

The interrupt signal triggers an interrupt event in the I/O Kit kext keyboard driver. The driver translates the signal into a key code which is passed to the OS X WindowServer process. Resultantly, the WindowServer dispatches an event to any appropriate (e.g. active or listening) applications through their Mach port where it is placed into an event queue. Events can then be read from this queue by threads with sufficient privileges calling the mach\_ipc\_dispatch function. This most commonly occurs through, and is handled by, an NSApplication main event loop, via an NSEvent of NSEventType KeyDown.

**(On GNU/Linux) the Xorg server listens for keycodes**

When a graphical X server is used, X will use the generic event driver evdev to acquire the keypress. A re-mapping of keycodes to scancodes is made with X server specific keymaps and rules. When the scancode mapping of the key pressed is complete, the X server sends the character to the window manager (DWM, metacity, i3, etc), so the window manager in turn sends the character to the focused window. The graphical API of the window that receives the character prints the appropriate font symbol in the appropriate focused field.

**Parse URL**

* The browser now has the following information contained in the URL (Uniform Resource Locator):
  + *Protocol "http"*

Use 'Hyper Text Transfer Protocol'

* + *Resource "/"*

Retrieve main (index) page

**Is it a URL or a search term?**

When no protocol or valid domain name is given the browser proceeds to feed the text given in the address box to the browser's default web search engine. In many cases the URL has a special piece of text appended to it to tell the search engine that it came from a particular browser's URL bar.

**Convert non-ASCII Unicode characters in the hostname**

* The browser checks the hostname for characters that are not in a-z, A-Z, 0-9, -, or ..
* Since the hostname is google.com there won't be any, but if there were the browser would apply [Punycode](https://en.wikipedia.org/wiki/Punycode) encoding to the hostname portion of the URL.

**Check HSTS list**

* The browser checks its "preloaded HSTS (HTTP Strict Transport Security)" list. This is a list of websites that have requested to be contacted via HTTPS only.
* If the website is in the list, the browser sends its request via HTTPS instead of HTTP. Otherwise, the initial request is sent via HTTP. (Note that a website can still use the HSTS policy *without* being in the HSTS list. The first HTTP request to the website by a user will receive a response requesting that the user only send HTTPS requests. However, this single HTTP request could potentially leave the user vulnerable to a [downgrade attack](http://en.wikipedia.org/wiki/SSL_stripping), which is why the HSTS list is included in modern web browsers.)

**DNS lookup**

* Browser checks if the domain is in its cache. (to see the DNS Cache in Chrome, go to chrome://net-internals/#dns).
* If not found, the browser calls gethostbyname library function (varies by OS) to do the lookup.
* gethostbyname checks if the hostname can be resolved by reference in the local hosts file (whose location [varies by OS](https://en.wikipedia.org/wiki/Hosts_%28file%29#Location_in_the_file_system)) before trying to resolve the hostname through DNS.
* If gethostbyname does not have it cached nor can find it in the hosts file then it makes a request to the DNS server configured in the network stack. This is typically the local router or the ISP's caching DNS server.
* If the DNS server is on the same subnet the network library follows the ARP process below for the DNS server.
* If the DNS server is on a different subnet, the network library follows the ARP process below for the default gateway IP.

**ARP process**

In order to send an ARP (Address Resolution Protocol) broadcast the network stack library needs the target IP address to lookup. It also needs to know the MAC address of the interface it will use to send out the ARP broadcast.

The ARP cache is first checked for an ARP entry for our target IP. If it is in the cache, the library function returns the result: Target IP = MAC.

If the entry is not in the ARP cache:

* The route table is looked up, to see if the Target IP address is on any of the subnets on the local route table. If it is, the library uses the interface associated with that subnet. If it is not, the library uses the interface that has the subnet of our default gateway.
* The MAC address of the selected network interface is looked up.
* The network library sends a Layer 2 (data link layer of the [OSI model](https://en.wikipedia.org/wiki/OSI_model)) ARP request:

ARP Request:

Sender MAC: interface:mac:address:here

Sender IP: interface.ip.goes.here

Target MAC: FF:FF:FF:FF:FF:FF (Broadcast)

Target IP: target.ip.goes.here

Depending on what type of hardware is between the computer and the router:

Directly connected:

* If the computer is directly connected to the router the router response with an ARP Reply (see below)

Hub:

* If the computer is connected to a hub, the hub will broadcast the ARP request out of all other ports. If the router is connected on the same "wire", it will respond with an ARP Reply (see below).

Switch:

* If the computer is connected to a switch, the switch will check its local CAM/MAC table to see which port has the MAC address we are looking for. If the switch has no entry for the MAC address it will rebroadcast the ARP request to all other ports.
* If the switch has an entry in the MAC/CAM table it will send the ARP request to the port that has the MAC address we are looking for.
* If the router is on the same "wire", it will respond with an ARP Reply (see below)

ARP Reply:

Sender MAC: target:mac:address:here

Sender IP: target.ip.goes.here

Target MAC: interface:mac:address:here

Target IP: interface.ip.goes.here

Now that the network library has the IP address of either our DNS server or the default gateway it can resume its DNS process:

* The DNS client establishes a socket to UDP port 53 on the DNS server, using a source port above 1023.
* If the response size is too large, TCP will be used instead.
* If the local/ISP DNS server does not have it, then a recursive search is requested and that flows up the list of DNS servers until the SOA is reached, and if found an answer is returned.

**Opening of a socket**

Once the browser receives the IP address of the destination server, it takes that and the given port number from the URL (the HTTP protocol defaults to port 80, and HTTPS to port 443), and makes a call to the system library function named socket and requests a TCP socket stream - AF\_INET/AF\_INET6 and SOCK\_STREAM.

* This request is first passed to the Transport Layer where a TCP segment is crafted. The destination port is added to the header, and a source port is chosen from within the kernel's dynamic port range (ip\_local\_port\_range in Linux).
* This segment is sent to the Network Layer, which wraps an additional IP header. The IP address of the destination server as well as that of the current machine is inserted to form a packet.
* The packet next arrives at the Link Layer. A frame header is added that includes the MAC address of the machine's NIC as well as the MAC address of the gateway (local router). As before, if the kernel does not know the MAC address of the gateway, it must broadcast an ARP query to find it.

At this point the packet is ready to be transmitted through either:

* [Ethernet](http://en.wikipedia.org/wiki/IEEE_802.3)
* [WiFi](https://en.wikipedia.org/wiki/IEEE_802.11)
* [Cellular data network](https://en.wikipedia.org/wiki/Cellular_data_communication_protocol)

For most home or small business Internet connections the packet will pass from your computer, possibly through a local network, and then through a modem (MOdulator/DEModulator) which converts digital 1's and 0's into an analog signal suitable for transmission over telephone, cable, or wireless telephony connections. On the other end of the connection is another modem which converts the analog signal back into digital data to be processed by the next [network node](https://en.wikipedia.org/wiki/Computer_network#Network_nodes) where the from and to addresses would be analyzed further.

Most larger businesses and some newer residential connections will have fiber or direct Ethernet connections in which case the data remains digital and is passed directly to the next [network node](https://en.wikipedia.org/wiki/Computer_network#Network_nodes) for processing.

Eventually, the packet will reach the router managing the local subnet. From there, it will continue to travel to the autonomous system's (AS) border routers, other ASes, and finally to the destination server. Each router along the way extracts the destination address from the IP header and routes it to the appropriate next hop. The time to live (TTL) field in the IP header is decremented by one for each router that passes. The packet will be dropped if the TTL field reaches zero or if the current router has no space in its queue (perhaps due to network congestion).

This send and receive happens multiple times following the TCP connection flow:

* Client chooses an initial sequence number (ISN) and sends the packet to the server with the SYN bit set to indicate it is setting the ISN
* *Server receives SYN and if it's in an agreeable mood:*
  + Server chooses its own initial sequence number
  + Server sets SYN to indicate it is choosing its ISN
  + Server copies the (client ISN +1) to its ACK field and adds the ACK flag to indicate it is acknowledging receipt of the first packet
* *Client acknowledges the connection by sending a packet:*
  + Increases its own sequence number
  + Increases the receiver acknowledgment number
  + Sets ACK field
* *Data is transferred as follows:*
  + As one side sends N data bytes, it increases its SEQ by that number
  + When the other side acknowledges receipt of that packet (or a string of packets), it sends an ACK packet with the ACK value equal to the last received sequence from the other
* *To close the connection:*
  + The closer sends a FIN packet
  + The other sides ACKs the FIN packet and sends its own FIN
  + The closer acknowledges the other side's FIN with an ACK

**TLS handshake**

* The client computer sends a ClientHello message to the server with its Transport Layer Security (TLS) version, list of cipher algorithms and compression methods available.
* The server replies with a ServerHello message to the client with the TLS version, selected cipher, selected compression methods and the server's public certificate signed by a CA (Certificate Authority). The certificate contains a public key that will be used by the client to encrypt the rest of the handshake until a symmetric key can be agreed upon.
* The client verifies the server digital certificate against its list of trusted CAs. If trust can be established based on the CA, the client generates a string of pseudo-random bytes and encrypts this with the server's public key. These random bytes can be used to determine the symmetric key.
* The server decrypts the random bytes using its private key and uses these bytes to generate its own copy of the symmetric master key.
* The client sends a Finished message to the server, encrypting a hash of the transmission up to this point with the symmetric key.
* The server generates its own hash, and then decrypts the client-sent hash to verify that it matches. If it does, it sends its own Finished message to the client, also encrypted with the symmetric key.
* From now on the TLS session transmits the application (HTTP) data encrypted with the agreed symmetric key.

**If a packet is dropped**

Sometimes, due to network congestion or flaky hardware connections, TLS packets will be dropped before they get to their final destination. The sender then has to decide how to react. The algorithm for this is called [TCP congestion control](https://en.wikipedia.org/wiki/TCP_congestion_control). This varies depending on the sender; the most common algorithms are [cubic](https://en.wikipedia.org/wiki/CUBIC_TCP) on newer operating systems and [New Reno](https://en.wikipedia.org/wiki/TCP_congestion_control#TCP_New_Reno) on almost all others.

* Client chooses a [congestion window](https://en.wikipedia.org/wiki/TCP_congestion_control#Congestion_window) based on the [maximum segment size](https://en.wikipedia.org/wiki/Maximum_segment_size) (MSS) of the connection.
* For each packet acknowledged, the window doubles in size until it reaches the 'slow-start threshold'. In some implementations, this threshold is adaptive.
* After reaching the slow-start threshold, the window increases additively for each packet acknowledged. If a packet is dropped, the window reduces exponentially until another packet is acknowledged.

**HTTP protocol**

If the web browser used was written by Google, instead of sending an HTTP request to retrieve the page, it will send a request to try and negotiate with the server an "upgrade" from HTTP to the SPDY protocol.

If the client is using the HTTP protocol and does not support SPDY, it sends a request to the server of the form:

GET / HTTP/1.1

Host: google.com

Connection: close

[other headers]

where [other headers] refers to a series of colon-separated key-value pairs formatted as per the HTTP specification and separated by single newlines. (This assumes the web browser being used doesn't have any bugs violating the HTTP spec. This also assumes that the web browser is using HTTP/1.1, otherwise it may not include the Host header in the request and the version specified in the GET request will either be HTTP/1.0 or HTTP/0.9.)

HTTP/1.1 defines the "close" connection option for the sender to signal that the connection will be closed after completion of the response. For example,

Connection: close

HTTP/1.1 applications that do not support persistent connections MUST include the "close" connection option in every message.

After sending the request and headers, the web browser sends a single blank newline to the server indicating that the content of the request is done.

The server responds with a response code denoting the status of the request and responds with a response of the form:

200 OK

[response headers]

Followed by a single newline, and then sends a payload of the HTML content of www.google.com. The server may then either close the connection, or if headers sent by the client requested it, keep the connection open to be reused for further requests.

If the HTTP headers sent by the web browser included sufficient information for the webserver to determine if the version of the file cached by the web browser has been unmodified since the last retrieval (ie. if the web browser included an ETag header), it may instead respond with a request of the form:

304 Not Modified

[response headers]

and no payload, and the web browser instead retrieve the HTML from its cache.

After parsing the HTML, the web browser (and server) repeats this process for every resource (image, CSS, favicon.ico, etc) referenced by the HTML page, except instead of GET / HTTP/1.1 the request will be GET /$(URL relative to www.google.com) HTTP/1.1.

If the HTML referenced a resource on a different domain than www.google.com, the web browser goes back to the steps involved in resolving the other domain, and follows all steps up to this point for that domain. The Host header in the request will be set to the appropriate server name instead of google.com.

**HTTP Server Request Handle**

The HTTPD (HTTP Daemon) server is the one handling the requests/responses on the server-side. The most common HTTPD servers are Apache or nginx for Linux and IIS for Windows.

* The HTTPD (HTTP Daemon) receives the request.
* *The server breaks down the request to the following parameters:*
  + HTTP Request Method (either GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS, or TRACE). In the case of a URL entered directly into the address bar, this will be GET.
  + Domain, in this case - google.com.
  + Requested path/page, in this case - / (as no specific path/page was requested, / is the default path).
* The server verifies that there is a Virtual Host configured on the server that corresponds with google.com.
* The server verifies that google.com can accept GET requests.
* The server verifies that the client is allowed to use this method (by IP, authentication, etc.).
* If the server has a rewrite module installed (like mod\_rewrite for Apache or URL Rewrite for IIS), it tries to match the request against one of the configured rules. If a matching rule is found, the server uses that rule to rewrite the request.
* The server goes to pull the content that corresponds with the request, in our case it will fall back to the index file, as "/" is the main file (some cases can override this, but this is the most common method).
* The server parses the file according to the handler. If Google is running on PHP, the server uses PHP to interpret the index file, and streams the output to the client.

**Behind the scenes of the Browser**

Once the server supplies the resources (HTML, CSS, JS, images, etc.) to the browser it undergoes the below process:

* Parsing - HTML, CSS, JS
* Rendering - Construct DOM Tree → Render Tree → Layout of Render Tree → Painting the render tree

**Browser**

The browser's functionality is to present the web resource you choose, by requesting it from the server and displaying it in the browser window. The resource is usually an HTML document, but may also be a PDF, image, or some other type of content. The location of the resource is specified by the user using a URI (Uniform Resource Identifier).

The way the browser interprets and displays HTML files is specified in the HTML and CSS specifications. These specifications are maintained by the W3C (World Wide Web Consortium) organization, which is the standards organization for the web.

Browser user interfaces have a lot in common with each other. Among the common user interface elements are:

* An address bar for inserting a URI
* Back and forward buttons
* Bookmarking options
* Refresh and stop buttons for refreshing or stopping the loading of current documents
* Home button that takes you to your home page

**Browser High-Level Structure**

The components of the browsers are:

* **User interface:** The user interface includes the address bar, back/forward button, bookmarking menu, etc. Every part of the browser display except the window where you see the requested page.
* **Browser engine:** The browser engine marshals actions between the UI and the rendering engine.
* **Rendering engine:** The rendering engine is responsible for displaying requested content. For example if the requested content is HTML, the rendering engine parses HTML and CSS, and displays the parsed content on the screen.
* **Networking:** The networking handles network calls such as HTTP requests, using different implementations for different platforms behind a platform-independent interface.
* **UI backend:** The UI backend is used for drawing basic widgets like combo boxes and windows. This backend exposes a generic interface that is not platform-specific. Underneath it uses operating system user interface methods.
* **JavaScript engine:** The JavaScript engine is used to parse and execute JavaScript code.
* **Data storage:** The data storage is a persistence layer. The browser may need to save all sorts of data locally, such as cookies. Browsers also support storage mechanisms such as localStorage, IndexedDB, WebSQL and FileSystem.

**HTML parsing**

The rendering engine starts getting the contents of the requested document from the networking layer. This will usually be done in 8kB chunks.

The primary job of the HTML parser is to parse the HTML markup into a parse tree.

The output tree (the "parse tree") is a tree of DOM element and attribute nodes. DOM is short for Document Object Model. It is the object presentation of the HTML document and the interface of HTML elements to the outside world like JavaScript. The root of the tree is the "Document" object. Prior to any manipulation via scripting, the DOM has an almost one-to-one relation to the markup.

**The parsing algorithm**

HTML cannot be parsed using the regular top-down or bottom-up parsers.

The reasons are:

* The forgiving nature of the language.
* The fact that browsers have traditional error tolerance to support well known cases of invalid HTML.
* The parsing process is reentrant. For other languages, the source doesn't change during parsing, but in HTML, dynamic code (such as script elements containing document.write() calls) can add extra tokens, so the parsing process actually modifies the input.

Unable to use the regular parsing techniques, the browser utilizes a custom parser for parsing HTML. The parsing algorithm is described in detail by the HTML5 specification.

The algorithm consists of two stages: tokenization and tree construction.

**Actions when the parsing is finished**

The browser begins fetching external resources linked to the page (CSS, images, JavaScript files, etc.).

At this stage the browser marks the document as interactive and starts parsing scripts that are in "deferred" mode: those that should be executed after the document is parsed. The document state is set to "complete" and a "load" event is fired.

Note there is never an "Invalid Syntax" error on an HTML page. Browsers fix any invalid content and go on.

**CSS interpretation**

* Parse CSS files, <style> tag contents, and style attribute values using ["CSS lexical and syntax grammar"](http://www.w3.org/TR/CSS2/grammar.html)
* Each CSS file is parsed into a StyleSheet object, where each object contains CSS rules with selectors and objects corresponding CSS grammar.
* A CSS parser can be top-down or bottom-up when a specific parser generator is used.

**Page Rendering**

* Create a 'Frame Tree' or 'Render Tree' by traversing the DOM nodes, and calculating the CSS style values for each node.
* Calculate the preferred width of each node in the 'Frame Tree' bottom-up by summing the preferred width of the child nodes and the node's horizontal margins, borders, and padding.
* Calculate the actual width of each node top-down by allocating each node's available width to its children.
* Calculate the height of each node bottom-up by applying text wrapping and summing the child node heights and the node's margins, borders, and padding.
* Calculate the coordinates of each node using the information calculated above.
* More complicated steps are taken when elements are floated, positioned absolutely or relatively, or other complex features are used. See <http://dev.w3.org/csswg/css2/> and <http://www.w3.org/Style/CSS/current-work> for more details.
* Create layers to describe which parts of the page can be animated as a group without being re-rasterized. Each frame/render object is assigned to a layer.
* Textures are allocated for each layer of the page.
* The frame/render objects for each layer are traversed and drawing commands are executed for their respective layer. This may be rasterized by the CPU or drawn on the GPU directly using D2D/SkiaGL.
* All of the above steps may reuse calculated values from the last time the webpage was rendered, so that incremental changes require less work.
* The page layers are sent to the compositing process where they are combined with layers for other visible content like the browser chrome, iframes and addon panels.
* Final layer positions are computed and the composite commands are issued via Direct3D/OpenGL. The GPU command buffer(s) are flushed to the GPU for asynchronous rendering and the frame is sent to the window server.

**GPU Rendering**

* During the rendering process the graphical computing layers can use general purpose CPU or the graphical processor GPU as well.
* When using GPU for graphical rendering computations the graphical software layers split the task into multiple pieces, so it can take advantage of GPU massive parallelism for float point calculations required for the rendering process.

**Window Server**

**Post-rendering and user-induced execution**

After rendering has been completed, the browser executes JavaScript code as a result of some timing mechanism (such as a Google Doodle animation) or user interaction (typing a query into the search box and receiving suggestions). Plugins such as Flash or Java may execute as well, although not at this time on the Google homepage. Scripts can cause additional network requests to be performed, as well as modify the page or its layout, causing another round of page rendering and painting.

Internet hoạt động như thế nào?  
Internet hoạt động thông qua một mạng lưới định tuyến gói tin theo giao thức Internet (IP), Giao thức Kiểm soát Truyền tải (TCP) và các giao thức khác.

Giao thức là gì?  
Giao thức là một tập hợp các quy tắc quy định cách các máy tính giao tiếp với nhau qua mạng. Ví dụ, Giao thức Kiểm soát Truyền tải có một quy tắc là nếu một máy tính gửi dữ liệu đến một máy tính khác, máy tính nhận phải thông báo cho máy tính gửi biết nếu có dữ liệu bị thiếu để máy tính gửi có thể gửi lại. Hay Giao thức Internet quy định cách các máy tính định tuyến thông tin đến các máy tính khác bằng cách đính kèm địa chỉ vào dữ liệu mà nó gửi.

Gói tin là gì?  
Dữ liệu gửi qua Internet được gọi là một thông điệp. Trước khi thông điệp được gửi, nó sẽ được chia thành nhiều mảnh nhỏ gọi là gói tin. Các gói tin này được gửi độc lập với nhau. Kích thước tối đa của một gói tin thường dao động từ 1000 đến 3000 ký tự. Giao thức Internet quy định cách các thông điệp nên được đóng gói.

Mạng định tuyến gói tin là gì?  
Đó là một mạng lưới định tuyến các gói tin từ máy tính nguồn đến máy tính đích. Internet được tạo thành từ một mạng lưới khổng lồ các máy tính chuyên dụng gọi là bộ định tuyến (router). Nhiệm vụ của mỗi bộ định tuyến là biết cách chuyển gói tin từ nguồn đến đích. Một gói tin sẽ đi qua nhiều bộ định tuyến trong hành trình của nó.

Khi một gói tin di chuyển từ bộ định tuyến này sang bộ định tuyến khác, đó được gọi là một "bước nhảy" (hop). Bạn có thể sử dụng công cụ dòng lệnh traceroute để xem danh sách các bước nhảy mà gói tin đi qua giữa bạn và một máy chủ.

Giao thức Internet quy định cách các địa chỉ mạng nên được gắn vào phần tiêu đề của gói tin, một không gian được chỉ định trong gói tin chứa các siêu dữ liệu của nó. Giao thức Internet cũng quy định cách các bộ định tuyến nên chuyển tiếp các gói tin dựa trên địa chỉ trong tiêu đề.

Những bộ định tuyến Internet này đến từ đâu? Ai sở hữu chúng?  
Những bộ định tuyến này bắt nguồn từ những năm 1960 với ARPANET, một dự án quân sự có mục tiêu tạo ra một mạng máy tính phân tán để chính phủ có thể truy cập và phân phối thông tin trong trường hợp xảy ra sự kiện thảm khốc. Kể từ đó, nhiều công ty Cung cấp Dịch vụ Internet (ISP) đã thêm các bộ định tuyến vào các bộ định tuyến ARPANET.

Không có một chủ sở hữu duy nhất của các bộ định tuyến Internet này, mà thay vào đó là nhiều chủ sở hữu: các cơ quan chính phủ và các trường đại học liên quan đến ARPANET từ những ngày đầu và các công ty ISP như AT&T và Verizon sau này.

Hỏi ai sở hữu Internet giống như hỏi ai sở hữu tất cả các đường dây điện thoại. Không có một thực thể nào sở hữu tất cả; nhiều thực thể khác nhau sở hữu các phần của chúng.

Các gói tin có luôn đến đích theo đúng thứ tự không? Nếu không, thông điệp sẽ được tái cấu trúc như thế nào?  
Các gói tin có thể đến đích ngoài thứ tự. Điều này xảy ra khi một gói tin sau tìm thấy một lộ trình nhanh hơn đến đích so với gói tin trước. Tuy nhiên, phần tiêu đề của gói tin chứa thông tin về thứ tự của gói tin trong mối quan hệ với toàn bộ thông điệp. Giao thức Kiểm soát Truyền tải sử dụng thông tin này để tái cấu trúc thông điệp tại đích.

Các gói tin có luôn đến đích không?  
Giao thức Internet không đảm bảo rằng các gói tin luôn đến đích. Khi điều này xảy ra, nó được gọi là mất gói tin. Điều này thường xảy ra khi một bộ định tuyến nhận được nhiều gói tin hơn khả năng xử lý của nó. Bộ định tuyến không có lựa chọn nào khác ngoài việc bỏ qua một số gói tin.

Tuy nhiên, Giao thức Kiểm soát Truyền tải xử lý mất gói tin bằng cách thực hiện gửi lại. Nó làm điều này bằng cách yêu cầu máy tính đích gửi các gói tin xác nhận trở lại máy tính nguồn để chỉ ra lượng thông điệp mà nó đã nhận và tái cấu trúc. Nếu máy tính đích phát hiện có gói tin bị thiếu, nó sẽ gửi yêu cầu đến máy tính nguồn yêu cầu gửi lại các gói tin bị mất.

Khi hai máy tính giao tiếp thông qua Giao thức Kiểm soát Truyền tải, chúng ta nói rằng có một kết nối TCP giữa chúng.

Địa chỉ Internet trông như thế nào?  
Các địa chỉ này được gọi là địa chỉ IP và có hai tiêu chuẩn.

Tiêu chuẩn địa chỉ đầu tiên gọi là IPv4 và có dạng như 212.78.1.25. Tuy nhiên, vì IPv4 chỉ hỗ trợ 2³² (khoảng 4 tỷ) địa chỉ khả dụng, Lực lượng Tác chiến Internet đã đề xuất một tiêu chuẩn địa chỉ mới gọi là IPv6, có dạng như 3ffe:1893:3452:4:345:f345:f345:42fc. IPv6 hỗ trợ 2¹²⁸ địa chỉ khả dụng, cho phép nhiều thiết bị mạng hơn, đủ để đáp ứng nhu cầu khi số lượng thiết bị mạng trên Internet tính đến năm 2017 đã vượt quá 8 tỷ.

Do đó, có một sự chuyển đổi một-một giữa địa chỉ IPv4 và IPv6. Lưu ý rằng quá trình chuyển từ IPv4 sang IPv6 vẫn đang được thực hiện và sẽ mất thời gian dài. Đến năm 2014, Google tiết lộ lưu lượng IPv6 của họ chỉ đạt 3%.

Làm sao có thể có hơn 8 tỷ thiết bị mạng trên Internet trong khi chỉ có khoảng 4 tỷ địa chỉ IPv4?  
Điều này là vì có các địa chỉ IP công cộng và riêng tư. Nhiều thiết bị trong một mạng cục bộ kết nối với Internet sẽ chia sẻ cùng một địa chỉ IP công cộng. Trong mạng cục bộ đó, các thiết bị này được phân biệt với nhau bằng các địa chỉ IP riêng tư, thường có dạng 192.168.xx, 172.16.x.x hoặc 10.x.x.x, trong đó "x" là một số từ 1 đến 255. Các địa chỉ IP riêng tư này được cấp phát bởi Giao thức Cấu hình Host Động (DHCP).

Ví dụ, nếu một máy tính xách tay và một điện thoại thông minh trong cùng một mạng cục bộ đều yêu cầu truy cập [www.google.com](http://www.google.com/), trước khi các gói tin rời khỏi modem, modem sẽ sửa đổi tiêu đề gói tin và gán một trong các cổng của mình cho gói tin đó. Khi máy chủ Google phản hồi các yêu cầu, nó sẽ gửi dữ liệu trở lại modem qua cổng cụ thể này, để modem có thể biết chuyển tiếp các gói tin tới máy tính xách tay hay điện thoại thông minh.

Theo cách này, địa chỉ IP không phải là đặc trưng cho một máy tính mà là đặc trưng cho kết nối mà máy tính đó sử dụng để kết nối với Internet. Địa chỉ duy nhất cho máy tính của bạn là địa chỉ MAC, và địa chỉ này không bao giờ thay đổi trong suốt vòng đời của máy tính.

Giao thức ánh xạ địa chỉ IP riêng tư sang địa chỉ IP công cộng này được gọi là Giao thức Chuyển đổi Địa chỉ Mạng (NAT). Đây chính là cách giúp hỗ trợ hơn 8 tỷ thiết bị mạng với chỉ 4 tỷ địa chỉ IPv4 khả dụng.

Làm sao các bộ định tuyến biết gửi gói tin đến đâu? Liệu chúng có cần phải biết tất cả các địa chỉ IP trên Internet không?  
Mỗi bộ định tuyến không cần phải biết tất cả các địa chỉ IP. Nó chỉ cần biết phải chuyển tiếp mỗi gói tin tới đâu, qua liên kết ra ngoài (outbound link), tức là một trong những "hàng xóm" của nó. Lưu ý rằng các địa chỉ IP có thể được chia thành hai phần: tiền tố mạng và định danh máy chủ. Ví dụ, địa chỉ 129.42.13.69 có thể được chia thành:

* Tiền tố mạng: 129.42
* Định danh máy chủ: 13.69

Tất cả các thiết bị kết nối mạng qua cùng một kết nối Internet (ví dụ như một trường đại học, một công ty, hoặc một ISP trong khu vực đô thị) sẽ chia sẻ cùng một tiền tố mạng.

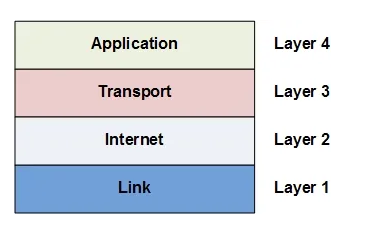
Các bộ định tuyến sẽ chuyển tất cả các gói tin có dạng 129.42.*.* đến cùng một vị trí trong mạng. Vì vậy, thay vì phải theo dõi hàng tỷ địa chỉ IP, các bộ định tuyến chỉ cần theo dõi ít hơn một triệu tiền tố mạng.

Nhưng một bộ định tuyến vẫn cần phải biết rất nhiều tiền tố mạng. Nếu một bộ định tuyến mới được thêm vào Internet, nó làm thế nào để biết cách xử lý các gói tin cho tất cả các tiền tố mạng này?  
Một bộ định tuyến mới có thể đi kèm với một số tuyến đường đã được cấu hình sẵn. Nhưng nếu nó gặp phải một gói tin mà nó không biết cách định tuyến, nó sẽ truy vấn một trong các bộ định tuyến láng giềng của mình. Nếu bộ định tuyến láng giềng biết cách định tuyến gói tin đó, nó sẽ gửi thông tin đó trở lại bộ định tuyến yêu cầu. Bộ định tuyến yêu cầu sẽ lưu thông tin này để sử dụng trong tương lai. Bằng cách này, một bộ định tuyến mới sẽ xây dựng bảng định tuyến của riêng mình, một cơ sở dữ liệu các tiền tố mạng và các liên kết ra ngoài tương ứng. Nếu bộ định tuyến láng giềng không biết, nó sẽ truy vấn các bộ định tuyến láng giềng của nó và cứ như vậy.

Làm sao các máy tính mạng biết cách xác định địa chỉ IP từ tên miền?  
Chúng ta gọi việc tra cứu địa chỉ IP của một tên miền có thể đọc được của con người, như [www.google.com](http://www.google.com/), là "giải quyết địa chỉ IP". Các máy tính giải quyết địa chỉ IP thông qua Hệ thống Tên Miền (DNS), một cơ sở dữ liệu phân tán chứa các ánh xạ từ tên miền đến địa chỉ IP.

Để giải quyết một địa chỉ IP, máy tính đầu tiên kiểm tra bộ nhớ cache DNS cục bộ của mình, nơi lưu trữ địa chỉ IP của các trang web mà nó đã truy cập gần đây. Nếu nó không tìm thấy địa chỉ IP ở đó hoặc bản ghi địa chỉ IP đã hết hạn, máy tính sẽ truy vấn các máy chủ DNS của ISP, những máy chủ chuyên dụng để giải quyết địa chỉ IP. Nếu các máy chủ DNS của ISP không thể giải quyết địa chỉ IP, chúng sẽ truy vấn các máy chủ tên gốc, những máy chủ có thể giải quyết mọi tên miền cho một tên miền cấp cao nhất (TLD) nhất định. Các tên miền cấp cao nhất là các từ phía bên phải dấu chấm cuối cùng trong tên miền, ví dụ như .com, .net, .org.

Các ứng dụng giao tiếp qua Internet như thế nào?  
Giống như nhiều dự án kỹ thuật phức tạp khác, Internet được chia thành các thành phần nhỏ độc lập, làm việc với nhau thông qua các giao diện được định nghĩa rõ ràng. Các thành phần này được gọi là Các Lớp Mạng Internet và chúng bao gồm: Lớp Liên kết, Lớp Internet, Lớp Truyền tải và Lớp Ứng dụng. Chúng được gọi là các lớp vì chúng được xây dựng chồng lên nhau; mỗi lớp sử dụng các khả năng của các lớp dưới nó mà không phải lo lắng về chi tiết triển khai của nó.



Các ứng dụng Internet hoạt động ở Lớp Ứng dụng và không cần phải lo lắng về các chi tiết trong các lớp dưới. Ví dụ, một ứng dụng kết nối với một ứng dụng khác trên mạng qua TCP bằng một cấu trúc gọi là socket, giúp trừu tượng hóa các chi tiết như định tuyến gói tin và tái cấu trúc các gói tin thành thông điệp.

Mỗi lớp Internet làm gì?  
Ở mức thấp nhất là Lớp Liên kết, là "lớp vật lý" của Internet. Lớp Liên kết chịu trách nhiệm truyền tải các bit dữ liệu qua một phương tiện vật lý như cáp quang hoặc tín hiệu sóng vô tuyến wifi.

Lớp Internet nằm trên Lớp Liên kết. Lớp Internet chịu trách nhiệm định tuyến các gói tin đến đích của chúng. Giao thức Internet được nhắc đến trước đó hoạt động trong lớp này (vì thế có tên gọi giống nhau). Giao thức Internet điều chỉnh và định tuyến lại các gói tin dựa trên tải mạng hoặc sự cố. Lưu ý rằng nó không đảm bảo các gói tin luôn đến đích, mà chỉ cố gắng làm tốt nhất có thể.

Lớp Truyền tải nằm trên Lớp Internet. Lớp này bù đắp cho việc dữ liệu có thể bị mất trong các lớp Internet và Liên kết dưới. Giao thức Kiểm soát Truyền tải (TCP) hoạt động ở lớp này, và nó chủ yếu giúp tái cấu trúc các gói tin thành thông điệp ban đầu và cũng gửi lại các gói tin bị mất.

Lớp Ứng dụng nằm trên cùng. Lớp này sử dụng tất cả các lớp dưới để xử lý các chi tiết phức tạp trong việc chuyển các gói tin qua Internet. Nó giúp các ứng dụng dễ dàng kết nối với các ứng dụng khác trên Internet thông qua các trừu tượng đơn giản như sockets. Giao thức HTTP, quy định cách các trình duyệt web và máy chủ web tương tác, hoạt động ở Lớp Ứng dụng. Giao thức IMAP, quy định cách các ứng dụng email truy cập email, cũng nằm trong Lớp Ứng dụng. Giao thức FTP, quy định một giao thức truyền tệp giữa các ứng dụng tải xuống tệp và các máy chủ lưu trữ tệp, cũng thuộc Lớp Ứng dụng.

Khách hàng và máy chủ là gì?  
Mặc dù cả khách hàng (client) và máy chủ (server) đều là các ứng dụng giao tiếp qua Internet, nhưng khách hàng là các ứng dụng "gần người dùng hơn", vì chúng là các ứng dụng giao diện người dùng như trình duyệt web, ứng dụng email, hoặc ứng dụng trên điện thoại thông minh. Máy chủ là các ứng dụng chạy trên máy tính từ xa, mà khách hàng giao tiếp qua Internet khi cần.

Một định nghĩa chính thức hơn là: ứng dụng khởi tạo kết nối TCP là khách hàng, trong khi ứng dụng nhận kết nối TCP là máy chủ.

Làm sao dữ liệu nhạy cảm như thẻ tín dụng có thể được truyền tải an toàn qua Internet?  
Vào những ngày đầu của Internet, chỉ cần đảm bảo các bộ định tuyến và liên kết mạng được bảo vệ vật lý ở các **vị trí** an toàn. Tuy nhiên, khi Internet phát triển và có thêm nhiều bộ định tuyến, có nghĩa là có nhiều điểm dễ bị tấn công hơn. Hơn nữa, với sự ra đời của các công nghệ không dây như WiFi, tin tặc có thể chặn các gói tin trong không khí; vì vậy, chỉ đảm bảo an toàn cho phần cứng mạng là không đủ. Giải pháp cho vấn đề này là mã hóa và xác thực qua SSL/TLS.

SSL/TLS là gì?  
SSL (Secure Sockets Layer) là một giao thức bảo mật được phát triển lần đầu tiên bởi Netscape vào năm 1994, nhưng một phiên bản bảo mật hơn sau đó đã được phát triển và đổi tên thành TLS (Transport Layer Security). Chúng ta sẽ gọi chung là SSL/TLS.

SSL/TLS là một lớp tùy chọn nằm giữa Lớp Truyền tải và Lớp Ứng dụng. Nó cho phép giao tiếp an toàn qua Internet của các thông tin nhạy cảm thông qua mã hóa và xác thực.

**Mã hóa** có nghĩa là khách hàng có thể yêu cầu kết nối TCP đến máy chủ được mã hóa. Điều này có nghĩa là tất cả các thông điệp được gửi giữa khách hàng và máy chủ sẽ được mã hóa trước khi chia thành các gói tin. Nếu tin tặc chặn các gói tin này, họ sẽ không thể khôi phục thông điệp gốc.

**Xác thực** có nghĩa là khách hàng có thể tin tưởng rằng máy chủ là đúng như những gì nó tuyên bố. Điều này giúp bảo vệ khỏi các cuộc tấn công kiểu "man-in-the-middle", khi một bên xâm nhập vào kết nối giữa khách hàng và máy chủ để nghe lén và thay đổi thông tin truyền tải.

Chúng ta thấy SSL hoạt động mỗi khi truy cập các trang web hỗ trợ SSL trên trình duyệt hiện đại. Khi trình duyệt yêu cầu một trang web sử dụng giao thức **https** thay vì **http**, nó đang thông báo với máy chủ web rằng nó muốn một kết nối được mã hóa bằng SSL. Nếu máy chủ web hỗ trợ SSL, một kết nối an toàn và mã hóa sẽ được thiết lập và chúng ta sẽ thấy biểu tượng ổ khóa bên cạnh thanh địa chỉ trên trình duyệt.

SSL xác thực danh tính của máy chủ và mã hóa giao tiếp của chúng như thế nào?  
SSL sử dụng mã hóa bất đối xứng và chứng chỉ SSL.

**Mã hóa bất đối xứng** là một hệ thống mã hóa sử dụng khóa công khai và khóa riêng. Các khóa này thực chất chỉ là các số được sinh ra từ các số nguyên tố lớn. Khóa riêng được sử dụng để giải mã dữ liệu và ký tài liệu, trong khi khóa công khai được sử dụng để mã hóa dữ liệu và xác minh tài liệu đã ký. Khác với mã hóa đối xứng, mã hóa bất đối xứng có nghĩa là khả năng mã hóa không đồng nghĩa với khả năng giải mã. Điều này thực hiện nhờ các nguyên lý trong lý thuyết số, một nhánh của toán học.

**Chứng chỉ SSL** là một tài liệu số bao gồm một khóa công khai được gán cho một máy chủ web. Các chứng chỉ SSL này được cấp phát cho máy chủ bởi các cơ quan cấp chứng chỉ (Certificate Authorities - CAs). Các hệ điều hành, thiết bị di động và trình duyệt đều có cơ sở dữ liệu chứa một số cơ quan cấp chứng chỉ đáng tin cậy để chúng có thể xác minh chứng chỉ SSL.

Khi một khách hàng yêu cầu một kết nối được mã hóa SSL với một máy chủ, máy chủ sẽ gửi chứng chỉ SSL của mình. Khách hàng kiểm tra chứng chỉ SSL để xác nhận:

* Chứng chỉ được cấp cho máy chủ này.
* Chứng chỉ được ký bởi một cơ quan cấp chứng chỉ đáng tin cậy.
* Chứng chỉ chưa hết hạn.

Sau đó, khách hàng sử dụng khóa công khai của chứng chỉ SSL để mã hóa một khóa bí mật tạm thời được tạo ra ngẫu nhiên và gửi lại cho máy chủ. Vì máy chủ có khóa riêng tương ứng, nó có thể giải mã khóa bí mật tạm thời của khách hàng. Bây giờ cả khách hàng và máy chủ đều biết khóa bí mật tạm thời này, và họ có thể sử dụng nó để mã hóa đối xứng các thông điệp mà họ gửi cho nhau. Họ sẽ vứt bỏ khóa bí mật tạm thời này khi phiên làm việc kết thúc.

**Chuyện gì sẽ xảy ra nếu một hacker chặn một phiên làm việc được mã hóa SSL?**  
Giả sử hacker đã chặn mọi thông điệp được gửi giữa khách hàng và máy chủ. Hacker nhìn thấy chứng chỉ SSL mà máy chủ gửi đi, cũng như khóa bí mật tạm thời đã được mã hóa của khách hàng. Nhưng vì hacker không có khóa riêng, nó không thể giải mã khóa bí mật tạm thời. Và vì nó không có khóa bí mật tạm thời, nó không thể giải mã bất kỳ thông điệp nào giữa khách hàng và máy chủ.

**Tóm tắt**  
Internet bắt đầu như ARPANET vào những năm 1960 với mục tiêu xây dựng một mạng máy tính phân tán.  
Về mặt vật lý, Internet là một tập hợp các máy tính di chuyển dữ liệu dưới dạng các bit qua dây cáp, cáp quang và tín hiệu radio.  
Giống như nhiều dự án kỹ thuật phức tạp khác, Internet được chia thành nhiều lớp, mỗi lớp giải quyết một vấn đề nhỏ hơn. Các lớp này kết nối với nhau qua các giao diện rõ ràng.  
Có nhiều giao thức định nghĩa cách thức hoạt động của Internet và các ứng dụng của nó ở các lớp khác nhau: HTTP, IMAP, SSH, TCP, UDP, IP, v.v. Theo nghĩa này, Internet vừa là một tập hợp các quy tắc về cách máy tính và chương trình nên hành xử, vừa là một mạng máy tính vật lý.  
Với sự phát triển của Internet, sự ra đời của WiFi, và nhu cầu thương mại điện tử, SSL/TLS đã được phát triển để giải quyết các vấn đề bảo mật.

https://medium.com/@User3141592/how-does-the-internet-work-edc2e22e7eb8