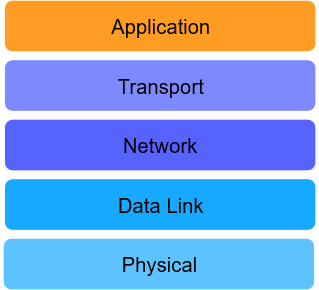
**What Is the Application Layer?**

### **The Post Analogy** [#](https://www.educative.io/courses/grokking-computer-networking/RMlKzPJZwGO#the-post-analogy)

Imagine you post a package across the world.

Presumably, the post system would hand it off to an airplane or ship to transport it across the world.

However, you would take it to the post office first to be shipped off. Carrying the package to the post office is what the application layer does in networks, except that it carries messages to the transport layer!



All of these applications(**e-mail**, **Instant messaging, voice over IP**, (WhatsApp calls), **video chat** (Skype), and **video streaming,** **Social media**) **run on application layer protocols**. Due to the presence of these standard protocols, client applications developed by various vendors can talk to server applications developed by others!

# Network Application Architectures

Before we start off with application layer protocols, **it’s important to understand how applications are structured across end systems**. This is called the network application’s **architecture** and it’s designed by application developers.

## Client-Server Architecture [#](https://www.educative.io/courses/grokking-computer-networking/R8MNDLDkALY#client-server-architecture)

In this architecture, a network application consists of two parts: **client-side** software and **server-side** software. These pieces of software are generally called **processes**, and they communicate with each other through **messages**.

### Servers [#](https://www.educative.io/courses/grokking-computer-networking/R8MNDLDkALY#servers)

The server process controls access to a centralized resource or service such as a website.

Servers have two important characteristics:

1. Generally, an attempt is made to keep servers online all the time, although 100% availability is impossible to achieve. Furthermore, servers set up as a hobby or as an experiment may not need to be kept online. Nevertheless, the client must be able to find the server online when needed, otherwise, communication wouldn’t take place.
2. They have at least one reliable IP address with which they can be reached.

### Clients [#](https://www.educative.io/courses/grokking-computer-networking/R8MNDLDkALY#clients)

Client processes use the Internet to consume content and use the services. Client processes almost always initiate connections to servers, while server processes wait for requests from clients.

### An Example [#](https://www.educative.io/courses/grokking-computer-networking/R8MNDLDkALY#an-example)

A good example of the client-server architecture is the **web**.

Take **Google** for instance. Google has several servers that control access to videos. So when a [google.com](http://google.com) is accessed, a client process (a browser) requests Google’s homepage from one of Google’s servers. That server was presumably online, got the request, and granted access to the page by sending it.

## Peer-to-Peer Architecture (P2P) [#](https://www.educative.io/courses/grokking-computer-networking/R8MNDLDkALY#peer-to-peer-architecture-p2p)

In this architecture, applications on end-systems called ‘peers’ communicate with each other. No dedicated server or large data center is involved. Peers mostly reside on PCs like laptops and desktops in homes, offices, and universities.

The key advantage of the P2P architecture is that it can scale rapidly – without the need of spending large amounts of money, time or effort.

### An Example [#](https://www.educative.io/courses/grokking-computer-networking/R8MNDLDkALY#an-example-2)

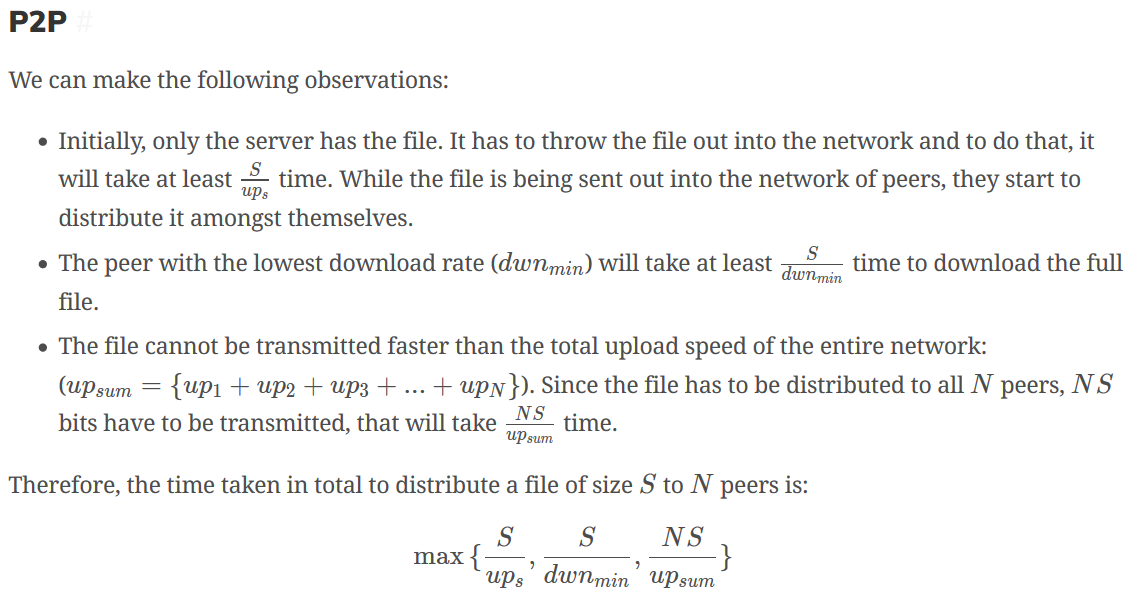
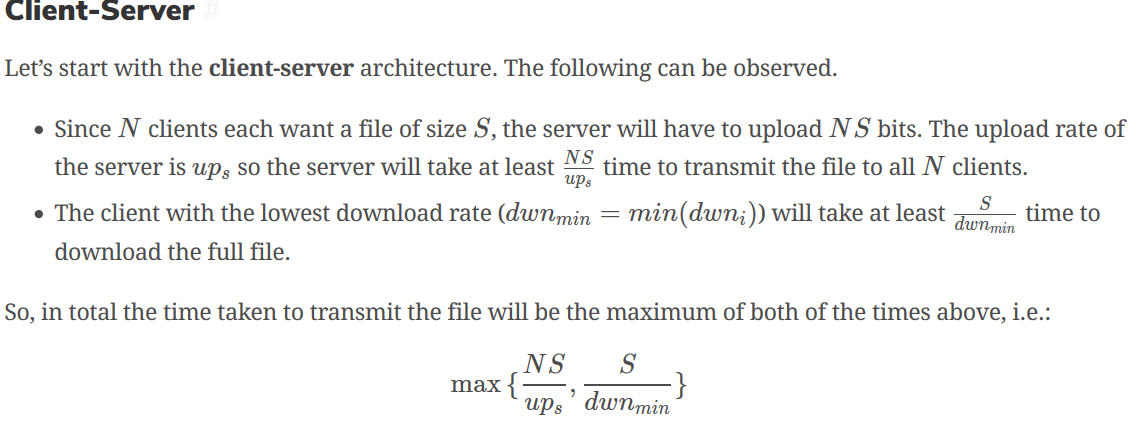
A lot of popular applications today, like **BitTorrent**, are based on P2P architectures.

When a file is downloaded via BitTorrent, the downloading party accesses **bits** of the file on several other users’ computers and puts them together on its end. No traditional ‘server’ is involved in this scenario.

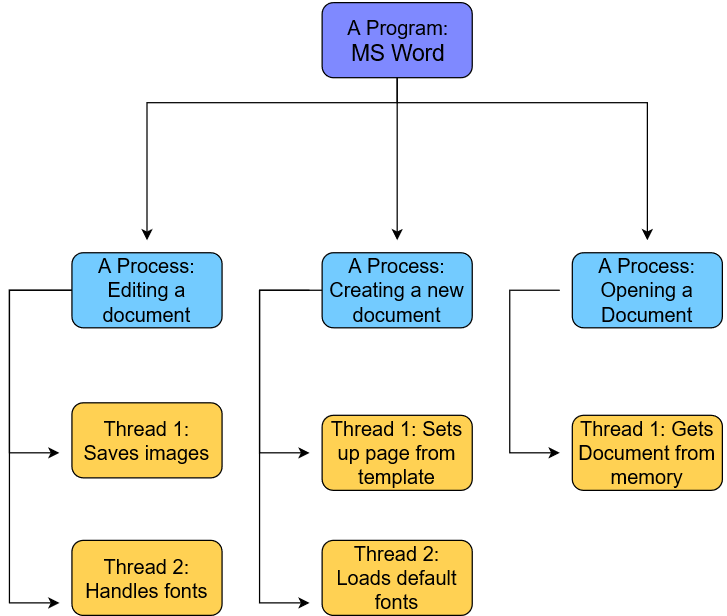
Also, file sharing is not the only application of P2P. Other examples include: streaming media, telephony, content distribution, routing, and volunteer computing.

## Hybrid [#](https://www.educative.io/courses/grokking-computer-networking/R8MNDLDkALY#hybrid)

The hybrid architecture involves server involvement to some degree. It’s essentially a combination of the P2P and client-server architectures.



## Program vs. Process vs. Thread [#](https://www.educative.io/courses/grokking-computer-networking/g2M6LyWPmRk#program-vs-process-vs-thread)

* A **program** is simply an executable file. An application such as MS Word is one example.
* A **process** is any currently running instance of a program. So one program can have several copies of it running at once. One MS Word program can have multiple open windows.
* A **thread** is a lightweight process. One process can have multiple running threads. The difference between threads and processes is that threads do lightweight singular jobs.
* 

Processes that exist on the same machine can and do regularly communicate with each other following the rules of the machine’s OS. However, we are more interested in how processes that run on different machines communicate.

## Sockets [#](https://www.educative.io/courses/grokking-computer-networking/g2M6LyWPmRk#sockets)

Processes on different machines send messages to each other through the computer network. The interface between a process and the computer network is called a **socket**. Note that sockets do not have anything to do with hardware – they are software interfaces.

Processes simply direct their messages to sockets and don’t worry about it after that.

## Addressing [#](https://www.educative.io/courses/grokking-computer-networking/g2M6LyWPmRk#addressing)

Messages have to be addressed to a certain application on a certain end system. How is it done with potentially millions of end systems and hundreds of applications on each of them?

Well, it’s done via addressing constructs like **IP addresses and ports**. While both were touched upon [previously](https://www.educative.io/collection/page/10370001/6105520698032128/4598191880142848/#ip-addresses), we would like to reintroduce ports a bit more in-depth.

### Ports

Since every end-system may have a number of applications running, **ports** are used to address the packet to specific applications. As stated previously, some ports are reserved such as port 80 for HTTP and port 443 for HTTPS.

#### An Analogy: Post

Continuing with our post analogy, you can think of an end-system like a large apartment complex. Each apartment in the complex is an application.

The mailing address of the complex is like the IP address of the end-system. All running applications share it, just like all apartments share the street address. Each application running on a host has a different port number, just like each apartment has a different apartment number.

**Ephemeral Ports**

The port that an application will use is usually predefined by its application developers. So an application can have port 3000 reserved for it. But what if several instances (processes) of an application are running at once? How will the system address those processes?

Well, the answer lies in Ephemeral Ports. Different port numbers are dynamically generated for each instance of an application. The port is freed once the application is done using it.

Furthermore, server processes need to have well defined and fixed port numbers so that clients can connect to them in a systematic and predictable way. However, clients don’t need to have reserved ports. They can use ephemeral ports. Servers can also use ephemeral ports in addition to the reserved ones. For instance, a client makes the initial connection to the server on a well-known port and the rest of the communication is carried out by connecting to an ephemeral port on the server.

### Do browsers use different ports to connect to different websites?

Yes, they do.

The remote ports are usually 80 (HTTP), 443 (HTTPS) or 8080 (HTTP Alternate).

Many other ports are used to host web servers. Search for HTTP on List of TCP and UDP port numbers for some of them. In general rendering a single web page uses multiple connections, not all of which will be to the same remote address. This is because web pages often include resources hosted elsewhere (javascript files, etc). Multiple connections to the same website (eg stackoverflow.com) also have different local ports (because they are separate connections in different tabs rendering different pages).

Each connection to a website uses a different socket with default destination TCP port 80 for plain HTTP and 443 for HTTPS. For the socket to be unique, the combination of the source IP address, source TCP port, destination IP address and destination TCP port must be different.

If you have multiple connections to the same website (assuming the website uses only 1 IP address) from the same computer, a different source TCP port must be used. This way, each connection is unique.

However, it should be noted that as of HTTP 1.1, all connections are persistent for a given period of time (unless declared otherwise). This means that the same connection can be reused by your browser if multiple resources from the same website are requested (e.g. css/js files). This also applies if you have multiple instances of the same website in your browser.

If you are on Windows, the netstat -no -p TCP command will show you all active TCP sockets and their corresponding process ID, including those of your browser:

Have you ever thought about what happens when you surf the web? It’s not as simple as it seems:

1. You type an URL into address bar in your preferred browser.
2. The browser parses the URL to find the protocol, host, port, and path.
3. It forms a HTTP request (that was most likely the protocol)
4. To reach the host, it first needs to translate the human readable host into an IP number, and it does this by doing a DNS lookup on the host
5. **Then a socket needs to be opened from the user’s computer to that IP number, on the port specified (most often port 80)**
6. **When a connection is open, the HTTP request is sent to the host**
7. The host forwards the request to the server software (most often Apache) configured to listen on the specified port
8. The server inspects the request (most often only the path), and launches the server plugin needed to handle the request (corresponding to the server language you use, PHP, Java, .NET, Python?)
9. The plugin gets access to the full request, and starts to prepare a HTTP response.
10. To construct the response a database is (most likely) accessed. A database search is made, based on parameters in the path (or data) of the request
11. Data from the database, together with other information the plugin decides to add, is combined into a long string of text (probably HTML).
12. The plugin combines that data with some meta data (in the form of HTTP headers), and sends the HTTP response back to the browser.
13. **The browser receives the response, and parses the HTML (which with 95% probability is broken) in the response**
14. A DOM tree is built out of the broken HTML
15. **New requests are made to the server for each new resource that is found in the HTML source (typically images, style sheets, and JavaScript files). Go back to step 3 and repeat for each resource.**
16. Stylesheets are parsed, and the rendering information in each gets attached to the matching node in the DOM tree
17. Javascript is parsed and executed, and DOM nodes are moved and style information is updated accordingly
18. The browser renders the page on the screen according to the DOM tree and the style information for each node
19. You see the page on the screen
20. You get annoyed the whole process was too slow.

# HTTP: The Basics

People could watch what they wanted. **HTTP** or **HyperText Transfer Protocol** is the protocol at the core of the web.

## Objects [#](https://www.educative.io/courses/grokking-computer-networking/B8ZkJYkmR2W#objects)

* Web pages are objects that consist of other **objects**.
* An **object is simply a file** like an HTML file, PNG file, MP3 file, etc.
* Each object has a URL
* The **base object** of a web page **is often an HTML file** that has **references to other objects** by making requests for them via their URL.

**HTML** or HyperText Markup Language is the standard markup language to build webpages. A **URL**, or **Universal Resource Locator**, is used to locate files that exist on servers. URLs consist of the following parts:

* **Protocol** in use
* The **hostname** of the server
* The **location of the file**
* **Arguments** to the file



Let’s get back into **HTTP**. It’s a client-server protocol that specifies how Web clients request Web pages from Web servers and how Web servers send them.

Note that HTTP is a **stateless protocol**: servers do not store any information about clients by default. So if a client requests the same object multiple times in a row, the server would send it and would not know that the same client is requesting the same object repeatedly.

## HTTP Requires Lower Layer Reliability [#](https://www.educative.io/courses/grokking-computer-networking/B8ZkJYkmR2W#http-requires-lower-layer-reliability)

* Application layer protocols rely on underlying transport layer protocols called **UDP** (User Datagram Protocol) and **TCP** (Transmission Control Protocol).
* For now, all you need to know is that **TCP ensures that messages are always delivered**. Messages get delivered in the order that they are sent.
* **UDP does not ensure that messages get delivered**. This means that some messages may get dropped and so never be received.
* **HTTP uses TCP** as its underlying transport protocol so that messages are guaranteed to get delivered in order. This allows the application to function without having to build any extra reliability as it would’ve had to with UDP.
* **TCP is connection-oriented**, meaning a connection has to be initiated with servers using a series of starting messages.
* Once the connection has been made, the client exchanges messages with the server until the connection is officially closed by sending a few ending messages.

## Types of HTTP Connections [#](https://www.educative.io/courses/grokking-computer-networking/B8ZkJYkmR2W#types-of-http-connections)

There are two kinds of HTTP connections:

* **Non-persistent HTTP connections**
* **Persistent HTTP connections**

## Non-persistent HTTP [#](https://www.educative.io/courses/grokking-computer-networking/B8ZkJYkmR2W#non-persistent-http)

**Non-persistent HTTP** connections use **one TCP connection per request**. Assume a client requests the base HTML file of a web page. Here is what happens:

1. The client initiates a TCP connection with a server
2. The client sends an HTTP request to the server
3. The server retrieves the requested object from its storage and sends it
4. The client receives the object which in this case is an HTML file. If that file has references to more objects, steps 1-4 are repeated for each of those
5. The server closes the TCP connection

For each HTTP request, more requests tend to follow, as well to fetch images, javascript files, CSS files, and other objects.

## Persistent HTTP [#](https://www.educative.io/courses/grokking-computer-networking/B8ZkJYkmR2W#persistent-http)

An HTTP session typically involves multiple HTTP request-response pairs, for which separate TCP connections are established and then torn down between the same client and server. This is inefficient. Later on, **Persistent HTTP** was developed, which used a single client-server TCP connection for all the HTTP request-responses for a session.

Typically, if there have been no requests for a while, the server closes the connection. The duration of time before the server closes the connection is configurable.

# HTTP: Request Messages

There are two types of HTTP messages as discussed previously:

* HTTP **request messages**
* HTTP **response messages**
* GET /path/to/file/index.html HTTP/1.1
* Host: www.educative.io
* Connection: close
* User-agent: Mozilla/5.0
* Accept-language: fr
* Accept: text/html

This particular message has 6 lines, but HTTP messages can have **one or as many lines as needed**.

The first line is called the **request line** while the rest are called **header lines**.

### HTTP Methods [#](https://www.educative.io/courses/grokking-computer-networking/xoNPoDWnjpl#http-methods)

* **GET** is the most common and **requests data**.
* **POST** **puts an object on the server**.
  + This method is generally used when the client is not sure where the new data would reside. The server responds with the location of the object.
  + The data posted can be a message for a bulletin board, newsgroup, mailing list, a command, a web form, or an item to add to a database.
  + The POST method technically requests a page but that depends on what was entered.
* **HEAD** is similar to the GET method except that **the resource requested does not get sent in response. Only the HTTP headers are sent** instead.
  + This is useful for quickly retrieving meta-information written in response headers, without having to transport the entire content. In other words, it’s useful to check with minimal traffic if a certain object still exists. This includes its meta-data, like the last modified date. The latter can be useful for caching.
  + This is also useful for testing and debugging.
* **PUT** **uploads an enclosed entity under a supplied URI**. In other words, it **puts** data at a specific location. If the URI refers to an already existing resource, it’s replaced with the new one. If the URI does not point to an existing resource, then the server creates the resource with that URI.
* **DELETE** **deletes an object** at a given URL.
* Note that while most forms are sent from the POST method, the GET method is also used sometimes with the entries of the form appended to the URL, as in arguments like this:

C:\Users\do936e\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Capture.png

However, sending forms with a POST request is generally better because:

1. The amount of data that can be sent via a post request is unlimited.
2. The form’s fields are not shown in the URL.

📝 **Note: URIs & URLs**

* **Uniform Resource Locators (URLs)** URLs are used to identify an object over the web. [RFC 2396](https://tools.ietf.org/html/rfc2396). A URL has the following format: protocol://hostname:port/path-and-file-name
* **Uniform Resource Identifiers (URIs)** can be more specific than URLs in such a way that they can locate fragments within objects too [RFC 3986](https://tools.ietf.org/html/rfc3986). A URI has the following format: http://host:port/path?request-parameters#nameAnchor. For instance, https://www.educative.io/collection/page/10370001/6105520698032128/6460983855808512/#http-methods is a URI.

### The Anatomy of HTTP Header Lines [#](https://www.educative.io/courses/grokking-computer-networking/xoNPoDWnjpl#the-anatomy-of-http-header-lines)

The HTTP request line is followed by an HTTP header. A lot of HTTP headers exist! We’ll be covering the most important ones in this lesson. However, if you’re interested, you can [read further](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields) about all of them.

* The first header line specifies the Host that the request is for.
* The second one defines the type of HTTP Connection. It’s Non-persistent in the case of the following drawing as the connection is specified to be closed.
* The user-agent line specifies the client. This is useful when the server has different web pages that exist for different devices and browsers.
* The Accept-language header specifies the language that is preferred. The server checks if a web page in that language exists and sends it if it does, otherwise the server sends the default page.
* The Accept header defines the sort of response to accept. It can be anything like HTML files, images, and audio/video.

# HTTP: Response Messages

HTTP/1.1 200 OK

Connection: close

Date: Tue, 18 Aug 2015 15: 44 : 04 GMT

Server: Apache/2.2.3 (CentOS)

Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT

Content-Length: 6821

Content-Type: text/html

[The object that was requested]

## Status Line [#](https://www.educative.io/courses/grokking-computer-networking/xVDnOkp9yWJ#status-line)

* HTTP response status lines start with the **HTTP version**.

### Status Code [#](https://www.educative.io/courses/grokking-computer-networking/xVDnOkp9yWJ#status-code)

* The **status code** comes next which tells the client if the request succeeded or failed.
* There are a lot of status codes:
  + 1xx codes fall in the informational category
  + 2xx codes fall in the success category
  + 3xx codes are for redirection
  + 4xx is client error
  + 5xx is server error

Here is a list of some common status codes and their meanings:

* **200 OK**: the request was successful, and the result is appended with the response message.
* **404 File Not Found**: the requested object doesn’t exist on the server.
* **400 Bad Request**: generic error code that indicates that the request was in a format that the server could not comprehend.
* **500 HTTP Internal Server Error**: the request could not be completed because the server encountered some unexpected error.
* **505 HTTP Version Not Supported**: the requested HTTP version is not supported by the server.

## Header Lines [#](https://www.educative.io/courses/grokking-computer-networking/xVDnOkp9yWJ#header-lines)

Let’s study the header lines.

* **Connection type**. In this case, indicates that the server will close the TCP connection after it sends the response.
* **Date**. The date at which the response was generated.
* **Server**. Gives server software specification of the server that generated the message. Apache in this case.
* **Last-Modified.** The date on which the object being sent was last modified.
* **Content-Length.** The length of the object being sent in 8-bit bytes.
* **Content-Type.** The type of content. The type of the file is not determined by the file extension of the object, but by this header.

### How HTTP Headers Are Chosen [#](https://www.educative.io/courses/grokking-computer-networking/xVDnOkp9yWJ#how-http-headers-are-chosen)

Lastly, you must be wondering how browsers decide which HTTP headers to include in requests and how servers decide which headers to return in the response. That **depends on a complex mix of factors such as the browser, the user configurations and products**.

**cURL** (pronounced ‘curl’) is a command-line tool that transfers data to or from a server. The transfer can be based on a vast set of protocols, so we’ll be seeing cURL a lot. It’s perfect for our purposes because it doesn’t require live user interaction. cURL stands for “**Client URL**.” You can read more about cURL on its [manpage](https://curl.haxx.se/docs/manpage.html).

curl http://example.org --get -silent

[manpage](https://curl.haxx.se/docs/manpage.html) : More about cURL : <https://curl.haxx.se/docs/manpage.html>

# Cookies

HTTP is a stateless protocol, but we often see websites where session state is needed. For instance, imagine you are browsing for products on an e-commerce website. How does the server know if you are logged in or not, or if the protocol is stateless? How does the server know what’s in your shopping cart when checking out if the protocol is stateless? Cookies allow the server to keep track of this sort of information.

**How Cookies Work**

Cookies are **unique string identifiers** that can be stored on the client’s browser.

These identifiers are **set by the server through HTTP headers** when the client first navigates to the website.

After the cookie is set, it’s sent along with subsequent HTTP requests to the same server. This **allows the server to know who is contacting it** and hence serves content accordingly.

So the HTTP request, the HTTP response, the cookie file on the client’s browser, and a database of cookie-user values on the server’s end are all involved in the process of setting and using cookies.

## Set-cookie Header [#](https://www.educative.io/courses/grokking-computer-networking/YM5KVoj67GW#set-cookie-header)

Let’s look at how cookies work in a bit more detail. When a server wants to set a cookie on the client-side, it includes the header Set-cookie: value in the HTTP response. This value is then **appended to a special cookie file stored on your browser**. The cookie file contains:

* The website’s domain
* The string value of the cookie
* The date that the cookie expires (yes, much like actual cookies, they do expire)

### Third-party Cookies

Also, websites may not necessarily know personally identifiable information about you such as your name (by the way, websites that require you to sign-up do know your name), and they may only know the value of your cookie. But what if **websites can track what you do on other websites**? Well, they can. Welcome to the concept of third-party cookies.

While we can’t go into too much detail, it suffices to know that **third-party cookies are cookies set for domains that are not being visited**.

#### Example [#](https://www.educative.io/courses/grokking-computer-networking/YM5KVoj67GW#example)

1. A user visits [amazon.com](http://amazon.com).
2. A cookie for [**free-stats.com**](http://free-stats.com) is subsequently set on their browser because free-stats has placed an advertisement on Amazon. Notice that this is **a third-party cookie**!
3. Suppose, the user visits [ebay.com](http://ebay.com), and **eBay also has placed an advertisement for** [**free-stats.com**](http://free-stats.com).
4. The **same cookie set on the Amazon site will be reused** and sent to free-stats along in an HTTP request with the name of the host that the user is on.
5. Free-stats **can in this way track every website the user visits** that they are advertising on and create more targeted ads in order to generate greater revenue.

Also, the public has largely considered third-party cookies to be a breach of privacy and so rejected them. Most modern browsers come with the in-built option to block third-party cookies.

## Viewing Cookies For a Page [#](https://www.educative.io/courses/grokking-computer-networking/xlLoK6YjWnq#viewing-cookies-for-a-page)

For most browsers, while you can’t view your entire cookie file, you can view cookies for individual websites.

1. Open up any **website of your choice**.
2. Open up the **developer tools**. All browsers have some form of developer tools.
3. Click on the **storage tab** to view details of the cookies that the website has set.

# Email: SMTP

There are many protocols associated with email. One popular choice is a combination of POP3 and SMTP. One is used to send emails that are stored in a user’s inbox and the other is used to retrieve emails sent to you. However, the very core of electronic mail is the **Simple Mail Transfer Protocol (SMTP)**.

SMTP uses TCP, which means that transfers are **reliable**. The connection is established at port 25.

Also, for ease and consistency, we are defining **User Agents** as agents that allow users to compose, view, delete, reply to, and forward emails. Applications such as Apple Mail, Microsoft Outlook, and Gmail’s webmail are examples of user agents.

## How SMTP Works [#](https://www.educative.io/courses/grokking-computer-networking/B8PkWyQNyko#how-smtp-works)

Let’s look at how this ubiquitous protocol works.

1. When an email is sent, its sent to the sender’s **SMTP server** using the SMTP protocol.
   * The SMTP server is configured in your email client. The general format of the domain of the SMTP server is smtp.example.com where the main email address of the sender is user@example.com. But it’s not mandatory to adhere to this format. We could set up, say, zeus.example.com to serve as our SMTP server, if we wanted. From a security point of view, it is probably a good idea, since people are unlikely to guess it as easily.
2. The **email is now placed on a message queue in the sending SMTP server**.
3. Then, the SMTP server initiates a connection with the recipient server and will conduct an initial SMTP handshake.
   * If the recipient is on the same SMTP server as the sender (for instance [alice@gmail.com](mailto:alice@gmail.com) sending to [bob@gmail.com](mailto:bob@gmail.com)), then the SMTP server doesn’t need to connect to the recipient’s server.
4. The SMTP server will finally send the message to the recipient’s email server.
5. The **email is then downloaded from the recipient’s SMTP server** using other protocols when the recipient logs in **to their email account or 'user agent**.’ In other words, the recipient’s SMTP server copies the email to the recipient’s mail-box.

**SMTP is a push protocol** because the email client sends the email out to the server when it needs to. Which means it only sends data to servers. Other protocols called **Mail Access Protocols** such as **POP** and **IMAP** are used for getting email from a server and are called pull protocols because the client asks their POP/IMAP server if they have any new messages whenever they feel like.

SMTP does not use any intermediate servers. So, even if an attempt to send an email fails because of any reason such as the receiving server being down, the email won’t be stored on an intermediary server. It will be stored on the sending SMTP server.

# Email: POP & IMAP

### POP Phases [#](https://www.educative.io/courses/grokking-computer-networking/B8W3ko6j7NW#pop-phases)

Emails are simply downloaded from the server in **4 phases: connect, authorize, transaction, update**.

1. **Connect:** The user agent first connects to the POP3 server on TCP using port 110.
2. **Authorize:** The user agent authenticates the user with a username and a password.
3. **Transaction:** The user can now retrieve emails and mark emails for deletion.
4. **Update:** After the user agent quits and closes the POP3 session, the server makes updates based on the user’s commands. So if the user marked an email for deletion, it will delete it. No copy of a deleted email is kept on the server.
   * Note that only what’s in the user’s inbox is downloaded. Other folders such as sent items, outbox, or drafts are not synced. So POP3 does not synchronize the folders.

POP works in two modes.

* **Download and delete**: Once emails are downloaded from the server to the user agent, they are all deleted from there.
* **Download and keep**: Emails are not deleted from the server once they are downloaded onto the user agent.

However, with the download and delete model, you can only use one client to check your emails. If you use multiple devices to check your email, this method is not appropriate because emails will not look the same across devices at different times. Also, users won’t be able to reread emails from different devices.

## IMAP [#](https://www.educative.io/courses/grokking-computer-networking/B8W3ko6j7NW#imap)

The **Internet Message Access Protocol (IMAP)**, like POP, is also a mail access protocol used for retrieving email. It is a bit more complex than POP and hence allows you to view your email from multiple devices. With IMAP, though:

* Emails are **kept** on the server and **not deleted**.
* Local copies of the emails are cached on each device.
* It **syncs** up all of the **user’s folders** including custom folders.
* The **inbox** would look exactly the **same on all clients**.
* If an email is deleted from one user agent, it will be **deleted off the server**.
* Deleted emails **won’t be visible** from other devices either.

# Email: Message Format

Email messages have a format the same way that HTTP request and response messages do. Let’s dive right into it.

### Header Lines [#](https://www.educative.io/courses/grokking-computer-networking/m2kGRXYG5NA#header-lines)

Email messages start with header lines, much akin to HTTP. The header lines contain important metadata about the email.

The header lines consist of keywords followed by a colon, followed by a value.

Every header line is separated by a new line with a carriage return (\r).

Every header must have the To: and From: header lines.

The rest of the headers, including the subject: header line, are optional.

### Message Body [#](https://www.educative.io/courses/grokking-computer-networking/m2kGRXYG5NA#message-body)

The message body of the email follows the header lines after a blank line.

Here is an example of what an email message looks like:

MIME-Version: 1.0

Date: Mon, 10 Feb 2020 21:14:57 +0530

Message-ID: <CAEapuu=7YONbZKHTp=s0uXVB6xVCD-fKpkSZ6SzhM\_5oEo6G4w@mail.gmail.com>

Subject: hi trail

From: Sai Ram <svnsr444@gmail.com>

To: Svn Sai Ram <svnsairam.sakhamuri@gmail.com>

Content-Type: multipart/alternative; boundary="000000000000106c0d059e3a9f81"

--000000000000106c0d059e3a9f81

Content-Type: text/plain; charset="UTF-8"

--000000000000106c0d059e3a9f81

Content-Type: text/html; charset="UTF-8"

<div dir="ltr"><br></div>

--000000000000106c0d059e3a9f81--

# DNS: Introduction

At the core, the Internet operates on IP addresses, but these are difficult to remember for humans. So, DNS names are preferably used at the application layer for which the DNS provides a mapping to IP addresses. For example, HTTP first translates the DNS hostname provided by the user in the URL to its IP address and then attempts to connect to the server. Furthermore, DNS is not just a protocol. It also consists of a distributed database of names that map to IP addresses. So essentially it’s a directory service.

One single database on one single server does not scale for reasons such as:

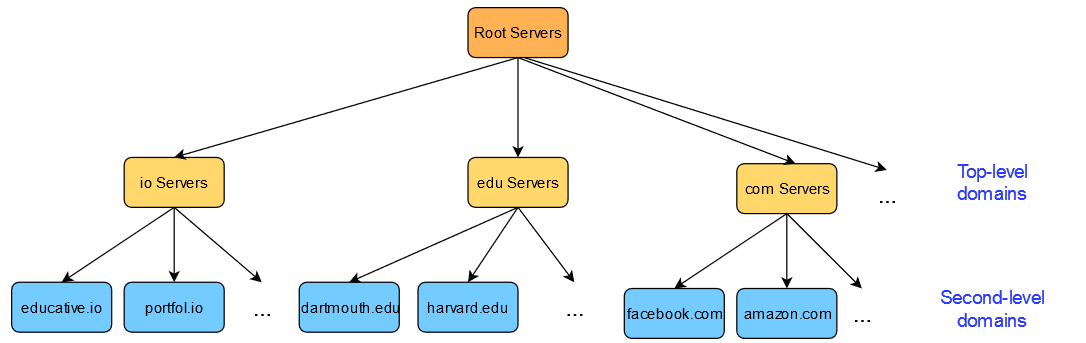
1. Single point of failure. If the server that has the database crashes, DNS would stop working entirely, which is too risky.

2. Massive amounts of traffic. Everyone would be querying that one server. It will not be able to handle that amount of load.

3. Maintenance. Maintaining the server would become critical to the operation of DNS.

4. Location. Where would the server be located?

This is why DNS employs several servers, each with part of the database. Also, the servers exist in a hierarchy. To understand this hierarchy better, you need to understand how URLs are broken down into their hierarchies. Have a look at the following diagram.



Root DNS servers are the first point of contact for a DNS query (after the client’s local cache of names and IP addresses). They exist at the top of the hierarchy and point to the appropriate TLD server in reply to the query. So a query for educative.io would return the IP address of a server for the top-level domain, io.

As of the writing of this course, there are 1017 instances of root servers operated by 12 different organizations. To get a full list and an interactive map, have a look at [root-servers.org](http://root-servers.org).

Today, the set of top-level domain-names(.com, .edu, .net, .io, .gov …. etc) is managed by the [Internet Corporation for Assigned Names and Numbers](https://www.icann.org/) (ICANN).

### Authoritative Servers [#](https://www.educative.io/courses/grokking-computer-networking/39LZ8PvPLnR#authoritative-servers)

Every organization with a public website or email server provides DNS records. These records have hostname to IP address mappings stored for that organization. These records can either be stored on a dedicated DNS server for that organization or they can pay for a service provider to store the records on their server.

This is the next link in the chain. If this server has the answer that we are looking for, the IP address that it has is finally returned to the client. However, this server may not have the sought after answer if the domain has a **sub-domain**. In that case, this server may point to a server that has records of the subdomain.

For instance, if the DNS record for [cs.stanford.edu](http://cs.stanford.edu) is being looked for, a DNS server separate from ‘[stanford.edu](http://stanford.edu)’ may hold records for the sub-domain ‘cs.’

### Local DNS Cache [#](https://www.educative.io/courses/grokking-computer-networking/39LZ8PvPLnR#local-dns-cache)

DNS mappings are often also cached locally on the client end-system to avoid repetitive lookups and saves time for often visited websites.

This is done via an entity called the **local resolver library**, which is part of the OS. The application makes an API call to this library. This library manages the local DNS cache.

If the local resolver library does not have a cached answer for the client, it will contact the organization’s local DNS server.

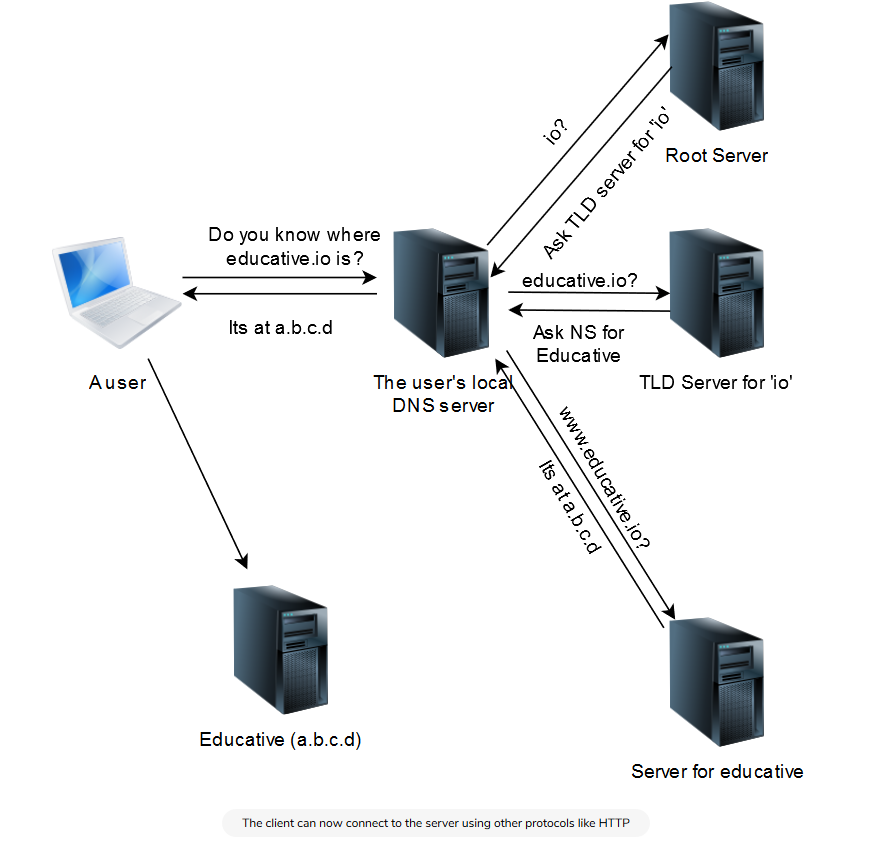
This local DNS server is typically configured on the client machine either using a protocol called **DHCP** or can be configured statically.

So, if it’s configured manually, any local DNS server of the client’s choice can be chosen. A few open DNS servers are incredibly popular, such as the ones by [Google](https://en.wikipedia.org/wiki/Google_Public_DNS).

### Local DNS Servers [#](https://www.educative.io/courses/grokking-computer-networking/39LZ8PvPLnR#local-dns-servers)

There is one type of server that we ignored — the **local DNS Server**. Local DNS servers are usually the first point of contact after a client checks its local cache. These servers are generally hosted at the ISP and contain some mappings based on what websites users visit.

**Security Warning**: ISPs have a record of which IP address they assigned to which customers. Furthermore, their DNS server has the IP addresses of who contacts it and what hostname they were trying to resolve. So your **ISP technically has a record of all of the websites you visit. Yikes!** P.S. if this makes you uncomfortable, you can change your DNS server to any open public DNS server.



## Checking What Your Local DNS Server Is [#](https://www.educative.io/courses/grokking-computer-networking/RMLO138QMEY#checking-what-your-local-dns-server-is)

To check the IP address of your local DNS server, run the following command on UNIX based machines. If you’re on a mobile machine, try [www.whatsmydnsserver.com](http://www.whatsmydnsserver.com). There are a lot of instructions available for Windows machines online.

# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)  
#     DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN  
nameserver 169.254.169.254  
search c.educative-exec-env.internal google.internal

You can safely ignore the first two lines since they are comments. On the third line, nameserver shows the IP address of the local DNS server. On the last line, search represents the default search domain that is used to resolve a query for a domain with no suffix (for example, www.facebook).

📝 **Note**: Educative’s code widgets run on a remote server. The code runs there, the output is then sent back to your machine and displayed. So, the DNS server shown here is local to the server that runs Educative’s code.

Now we can use AWS EC2 instances or Google cloud server it deploy our application without wasting money on building and maintaining our own server infrastructure.

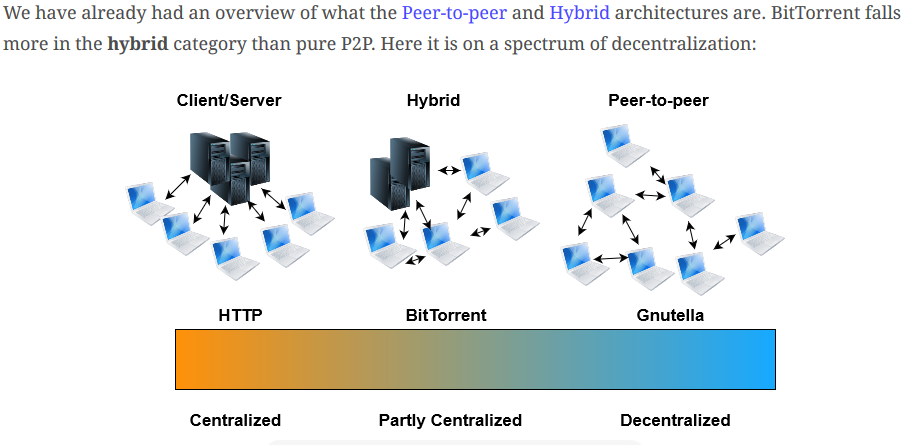
# DNS: Records and Messages

Let's now get into what DNS records and messages look like.

<https://www.educative.io/courses/grokking-computer-networking/RMxW1oROP7O>

<https://www.educative.io/courses/grokking-computer-networking/N8B0xgk73KK>

# BitTorrent



## Overview of BitTorrent [#](https://www.educative.io/courses/grokking-computer-networking/R1Axlvo5Ymq#overview-of-bittorrent)

BitTorrent is a protocol for peer-to-peer file sharing. A **BitTorrent Client** is an application that uses this protocol.

Since BitTorrent is based on a hybrid architecture, it retains some centralized components.

* For example, a **central controller** that maintains a list of participating nodes is involved.
* But the centralized component is not involved in resource-intensive operations. So there will never be too much load on it.
* Data is instead **downloaded or uploaded directly to and by peers**.
* The file is first supplied to a peer in pieces called chunks, and then they also distribute the file to other peers.
* This is sometimes called a **peer-assisted** system.

### Trackers and torrent files [#](https://www.educative.io/courses/grokking-computer-networking/R1Axlvo5Ymq#trackers-and-torrent-files)

How do clients find peers to connect to? Well, clients connect to a special tracker node first. The tracker responds with the IP and the port of a few other peers who are downloading the same file.

📝 **Note**: Modern BitTorrent clients are trackerless and use a Distributed Hash Table instead, but that’s beyond the scope of this course.

So clients can find peers through trackers. But how do clients find the tracker in the first place? Clients begin by downloading a ‘torrent file’ from a web server which has the URL of the tracker. The torrent file also contains a SHA1 hash of each file chunk. Can you guess why?

### A Simplified BitTorrent Session [#](https://www.educative.io/courses/grokking-computer-networking/R1Axlvo5Ymq#a-simplified-bittorrent-session)

1. Download the ‘torrent file’ from a web server.
2. Connect to the tracker and get a list of peers.
3. Connect to the peers - initially as a ‘leecher.’
4. While the file is not yet fully downloaded:
   * Advertise to peers which blocks are available locally.
   * Request blocks from peers.
   * Compare hash of downloaded blocks to the hash in the torrent file. Can you guess why?
5. Turn into a ‘seeder,’ i.e., continue uploading to peers without downloading.

# What Is the Transport Layer?