**4/21/21**

1. **HTTP**

HTTP is what we call an **application layer protocol** because it allows two applications to communicate over the network. Quite often **one of the applications is a web browser**, and **the other application is a web server** like IIS or Apache. We saw how HTTP messages allow the browser to request resources from the server. But, the HTTP specifications don't say anything about how the messages actually cross the network and reach the server – that's the job of lower layer protocols. A message from a web browser has to travel down a series of layers, and when it arrives at the web server it travels up through a series of layers to reach the web service process.

The layer underneath HTTP is a **transport layer protocol**. Most HTTP traffic travels over **TCP** (short for Transmission Control Protocol) in this layer, although TCP isn't required by HTTP. When a user types a URL into the browser, the browser opens a **TCP socket** by specifying the server address and port, then starts writing data into the socket. All the browser needs to worry about is writing the proper HTTP message into the socket. The TCP layer accepts the data and ensures the data gets delivered to the server without getting lost or duplicated. TCP will automatically resend any information that might get lost in transit. The application doesn't have to worry about lost data, and this is why TCP is known as a reliable protocol. In addition to error detection, TCP also provides flow control. Flow control ensures the sender does not send data too fast for the receiver or the network to process the data. Flow control is important in this world of varied networks and devices.

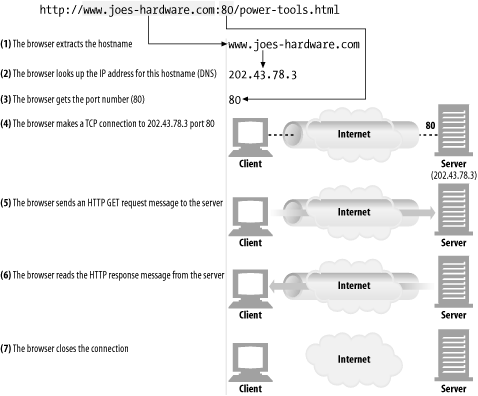
In short, TCP provides many vital services for the successful delivery of HTTP messages, but it does so in a transparent way. Most applications don't need to worry about TCP. And, TCP is just the first layer beneath HTTP. After TCP at the transport layer comes IP as a **network layer protocol**.

IP is short for **Internet Protocol**. While TCP is responsible for error detection, flow control, and overall reliability, **IP** is responsible for taking pieces of information and moving them through the various switches, routers, gateways, repeaters, and other devices that move information from one network to the next and all around the world. IP tries hard to deliver the data at the destination (but **it doesn't guarantee delivery** – that's TCP's job). To deliver data IP requires computers to have an address (the famous IP address, an example being 208.192.32.40). IP is also responsible for breaking data into packets (often called datagrams), and sometimes fragmenting and reassembling these packets so they are optimized for a particular network segment.

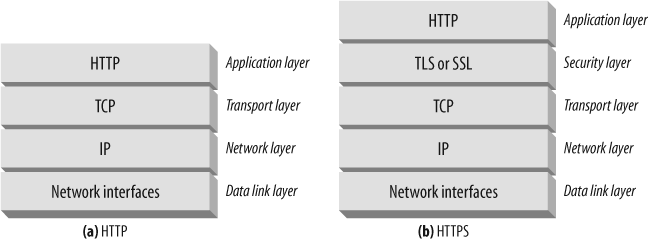
1. **HTTP connection over TCP/IP**

|  |
| --- |
| [http://www.joes-hardware.com:80/power-tools.html](http://www.joes-hardware.com/power-tools.html) |

When given this URL, your browser performs the steps shown in [Figure 4-1](https://www.oreilly.com/library/view/http-the-definitive/1565925092/ch04s01.html#httptdg-CHP-4-FIG-1). In Steps 1-3, the IP address and port number of the server are pulled from the URL. A TCP connection is made to the web server in Step 4, and a request message is sent across the connection in Step 5. The response is read in Step 6, and the connection is closed in Step 7.



*Figure 4-1. Web browsers talk to web servers over TCP connections*

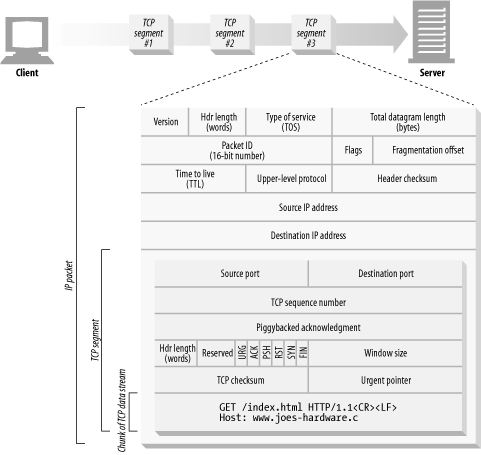


1. When HTTP wants to transmit a message, it streams the contents of the message data, in order, through an open TCP connection. **TCP** takes the stream of data, chops up the data stream into chunks called **segments**, and transports the segments across the Internet inside envelopes called **IP packets** (see [Figure 4-4](https://www.oreilly.com/library/view/http-the-definitive/1565925092/ch04s01.html#httptdg-CHP-4-FIG-4)). This is all handled by the TCP/IP software; the HTTP programmer sees none of it.

**Each TCP segment is carried by an IP packet from one IP address to another IP address**. Each of these **IP packets** contains:

* An IP packet header (usually 20 bytes)
* A TCP segment header (usually 20 bytes)
* A chunk of TCP data (0 or more bytes)

The **IP header** contains the source and destination IP addresses, the size, and other flags. The **TCP segment header** contains TCP port numbers, TCP control flags, and numeric values used for data ordering and integrity checking.



*Figure 4-4. IP packets carry TCP segments, which carry chunks of the TCP data stream*

1. Keeping TCP Connections Straight

A computer might have **several TCP connections** open at any one time. **TCP keeps all these connections** straight through ***port numbers*** .

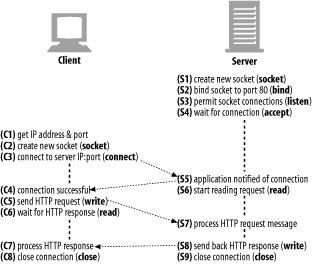
Port numbers are like employees’ phone extensions. Just as a company’s main phone number gets you to the front desk and the extension gets you to the right employee, **the IP address gets you to the right computer** and **the port number gets you to the right application**. **A TCP connection is distinguished by four values**:

<source-IP-address, source-port, destination-IP-address, destination-port>

Together, these four values uniquely define a connection. Two different TCP connections are not allowed to have the same values for all four address components (but different connections can have the same values for some of the components).

## Programming with TCP Sockets

Operating systems provide different facilities for manipulating their TCP connections. Let’s take a quick look at one TCP programming interface **socket**. This sockets API hides all the details of TCP and IP from the HTTP programmer.



Indeed, more specifically there is a special type of socket called a **"listening" socket**.

Normally a socket is associated with a combination of local IP, local port, remote IP and remote port.

**A listening socket is different. It is not associated with any specific remote IP and port. It is associated with a specific local port. It may or may not be associated with a specific local IP**.

Normally your web server will have a listening socket with a local port of 80

Then lets say a client A tries to connect to the server (make TCP/IP connection). During which a socket is created between these two.

A pair of sockets actually one on the client, one on the server.

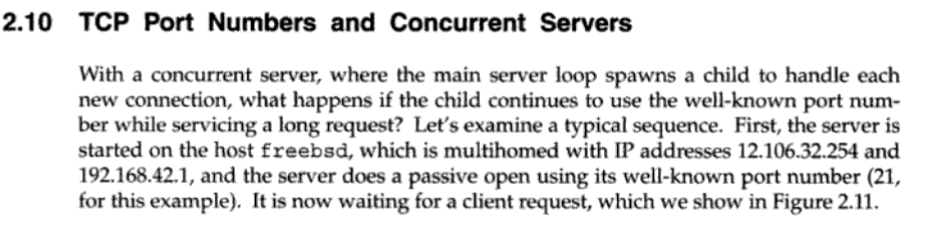
The client application creates a socket and asks the client OS to connect it to the server.

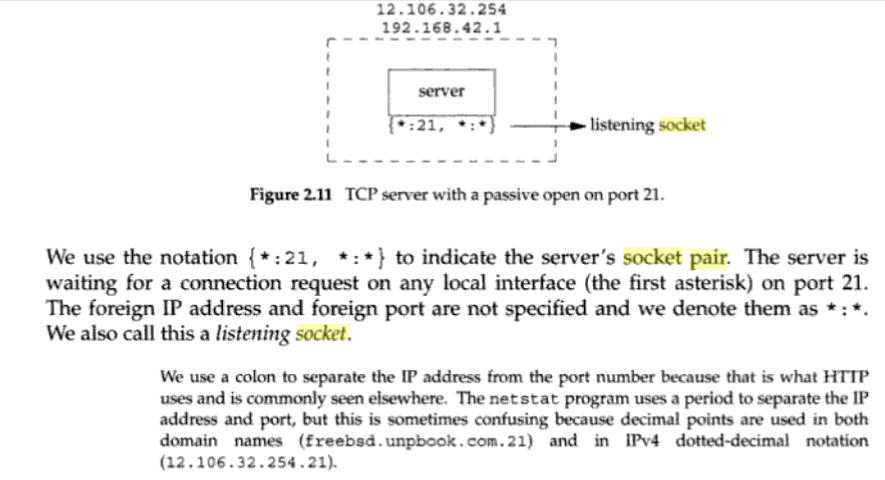
The client OS allocates a random local port, chooses a local IP (normally based on which interface the packet will be sent out on) and fills in the remote IP and port requested by the client application. Then it starts the process of connecting to the server.

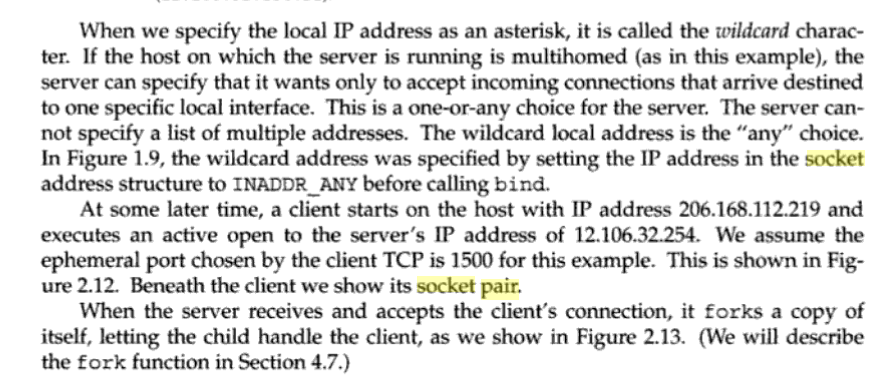
The **server OS** notifies the holder of the listening socket that **a new connection** is coming in. The server application accepts the connection and **a new socket is created to handle it**; the new socket has server’s **IP** and **80** as port number for webserver.

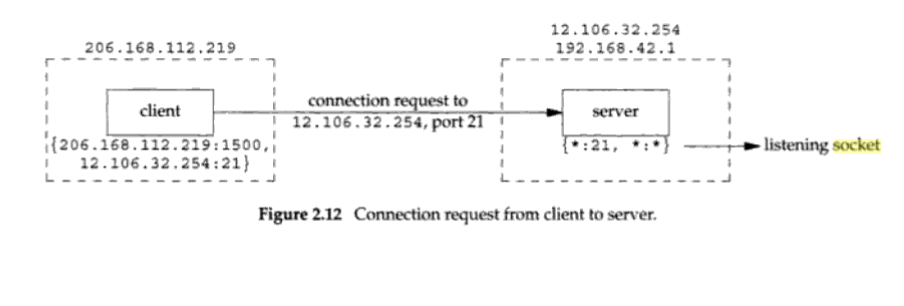
A **server** socket listens on a single port. All established client connections on that server are associated with that same listening port **on the server side** of the connection. An established connection is uniquely identified by the combination of client-side and server-side IP/Port pairs. Multiple connections on the same server can share the same **server-side** IP/Port pair as long as they are associated with different **client-side** IP/Port pairs, and the **server** would be able to handle as many clients as available system resources allow it to.

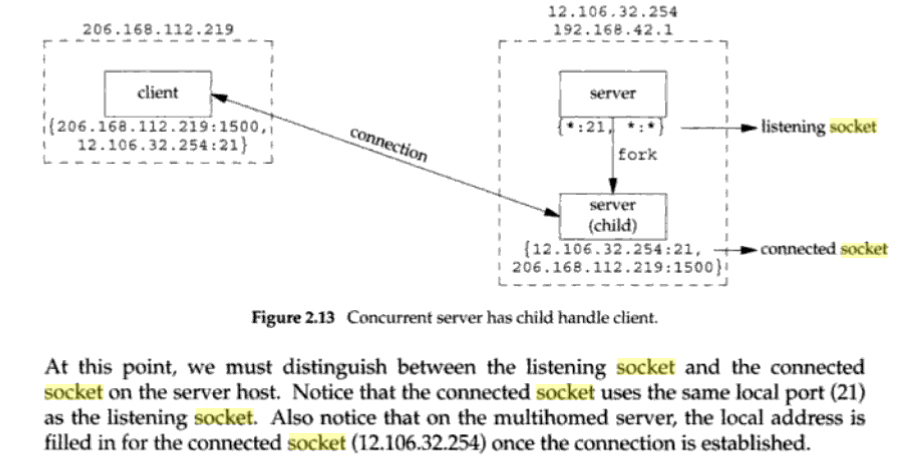
On the **client-side**, it is common practice for new outbound connections to use a random **client-side** port, in which case it is possible to run out of available ports if you make a lot of connections in a short amount of time

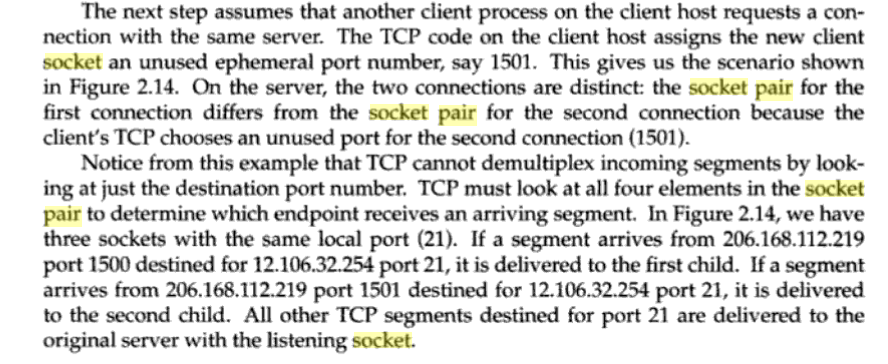












Recall that with TCP sockets, a server first creates a socket whose sole purpose is listening for and accepting incoming connections. It does so with the **listen** system call. This socket is said to be in the LISTEN state. It will never be used for data transfer. Its only purpose is to accept incoming connections.

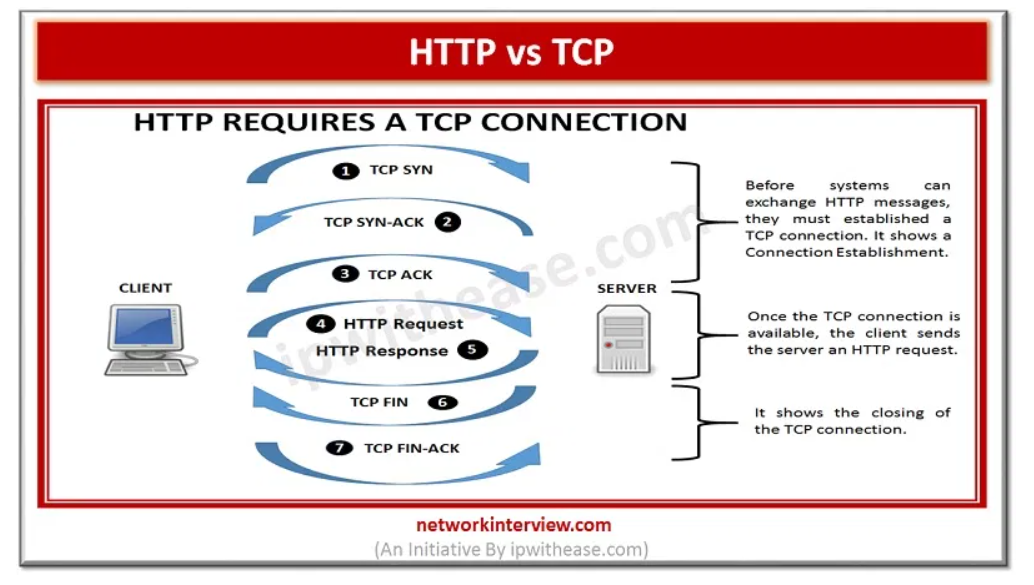
When an incoming TCP connection request arrives at the host, the kernel searches for a socket in the LISTEN state where the packet’s destination address and port match those of the socket (the address can be “any” - a wildcard). The kernel then creates a new socket. The remote address and port are copied from the TCP segment header onto the new socket structure. Once the connection is set up, this new socket is in the ESTABLISHED state, indicating to the kernel that it has a connection to another socket. Any incoming TCP data segment will go to the socket that is associated with the source and destination addresses and ports in the segment header.

An HTTP session is a sequence of network request-response transactions. An HTTP client initiates a request by establishing a Transmission Control Protocol (TCP) connection to a particular port on a server.

So a HTTP session is created by establishing a TCP connection. It follows that the session ends when the TCP connection is terminated.

#### Connection Termination

Finally the connection is torn down (or terminated). Each end will send a **FIN** flag to the other end to tell it that it’s done sending. This **FIN** is then acknowledged. When both ends have sent their **FIN** flags and they have been acknowledged, the connection is fully closed:



1. There are **application layer load balancers** like HAProxy, where the full HTTP request and response is passed through the proxy. There are two separate TCP connections here: one between client and load balancer and another between load balancer and upstream server. The choice of the upstream server can be done based on the contents of the HTTP request, for example the Host header and/or path but also session cookies - to make sure that the same session is always handled by the same upstream server. If the decision for the upstream server is done on the HTTP request then the connection to the upstream server can only be established after the HTTP request was read, since it is not known before. But the request does not need to be inside a single packet.

There are **network or transport layer load balancers** which do not act on the packet payload at all. Instead the choice of upstream server is usually done based on the client IP, so that the same client ends up on the same upstream server. In this case the decision which upstream to use is already done on the first packet (i.e. the SYN starting the TCP handshake) and the client essentially establishes the connection directly with the upstream proxy - the load balancer only forwards the packets like a router does. The size of the HTTP request does not matter here either since the TCP payload is not even inspected to make the routing decision.

With a network or transport layer load balancer there can be asymmetric routing, i.e. the response might go a different way and not pass through the load balancer. With application layer load balancing the response goes back through the load balancer instead.

# **When Does HTTP comes into Play?**

Internet is made of millions of computers called nodes, whenever any kind of data transfer or say request for data is made from one computer to another (**client** and **server**), This request is first formatted into valid HTTP format and then sent over the network channel.

When you google HTTP, You might come across certain terms like:  
connection less protocol, stateless protocol etc.

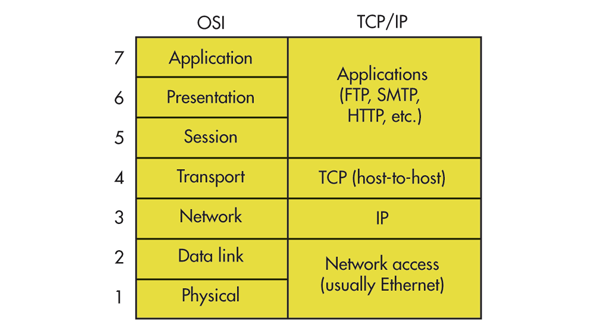
**Connection Less:** By this it means that when a client sends request to the server it is connected to the sever but after the request is sent no connection exists. **Why this connection less property?** It is because if both client and the server are connected even after sending the request (by client) until the response is received by the client, this might result is loss of computation power and storage, Moreover server will not be able to listen to other clients until it has processed one request, Thus applications using HTTP have to be synchronous with no choice left.

**Stateless:**By stateless it means that every request from client to the server is independent of previous request i.e. no data is stored or cached by default by HTTP. **why this stateless property?**It is because if HTTP was state full it would have been very difficult to develop applications as applications would have been developed keeping in mind the structure HTTP unlike now. State fullness might also result in unnecessary loss of storage.

# **Architecture Of Network Layers**

Networking is a Layered concept, i.e. sending and receiving data involves several layers before it reaches it’s destination.





There are two networking layer models:

**OSI —**  
Open System Interconnection Model Developed ISO.  
This model has 7 Layers with application being the layer in contact with the user.

**TCP/IP model —**

This model combines 5–7 and 1– 2 layers of OSI model reducing 3 layers in the model.  
**HTTP resides in application layer,**as it is concerned with the format of request or response data being transferred, and application layer is in direct contact with the user, This is where your applications work (websites, android apps etc.)

# **In what format HTTP transfers data?**

To achieve connection less and stateless property every requests has some data known as headers which is enough to make every request independent of each other.  
There are 3 types of headers —

1. General Headers
2. Response Headers
3. Request Headers

**General Headers** are included in every kind of data transfer (request or response)

**eg:  
1. cache-control headers:** To control the storage of data by cache.  
**2. connection:**Defines the state of the connection  
**3. Date:**Defines the time-stamp on which the transfer took place.

**Request Headers** are included when client requests data from server.

**eg:  
1. Accept:**Defines the type of data client wants.  
**2. Authorization:**Sets the authorization header required by the server to authenticate the client.  
**3. Host:**Specifies the host which will serve the request.  
**4. If-Modified-Since:**This header is used when clients wants content only if it is modified since the specified date.

**Response Headers** are included when server sends data to the client.

**eg:  
1. Set-Cookie:** To store a cookie for the given URL.  
**2. Server:**Contains the information about the software used by the origin server.

# **Structure Of Request Body:**

* Request Line (eg: Get /images/first.jpg HTTP/1.1 )
* Request Header Fields
* An Empty Line
* Body Message (Mandatory in POST method)

**Note:** Request line and every header field should end with Carriage Return & Line Feed characters and empty line should not consist of any white spaces. All header fields except Host is optional.

# **Structure Of Response Body:**

* Status Line (Code and Message) (eg: HTTP/1.1 200 OK)
* Response Header Fields
* An Empty Line
* Response Body

**Note:**Status line and every header fields should end with Carriage Return & Line Feed characters and empty line should not consist of any white spaces. In Response Body Carriage Return & Line Feed characters are not mandatory because of the maybe presence of Line Breaks in the message body.