Class Notes

Hyper Text Transfer Protocol (HTTP)

(1) We saw many "mechanisms" in the last class to improve some aspects of HTTP. Those mechanisms included:
   a. Persistent vs non-persistent connections
   b. Parallel connections between the same client-server pair
   c. Pipelining multiple requests on a single TCP connection
   d. Multiplexing "frames" of different object in round-robin fashion to avoid head-of-line blocking

(2) It is easy to get lost in the details. It is always good to zoom out and to remind us:
   a. What problem are we trying to solve?
   b. Can some of you remind everyone what problem above mechanism are trying to solve at the top level and why should we care about it?

(3) What is the definition of Round Trip Time (RTT)?
   a. What kinds of time are included in RTT?
   b. Textbook definition: "we define the round-trip time (RTT), which is the time it takes for a small packet to travel from client to server and then back to the client. The RTT includes packet-propagation delays, packet-queuing delays in intermediate routers and switches, and packet-processing delays."
   c. Why textbook ignored transmission delays?
   d. In exam, clearly tell us did you include transmission times in RTT or not (other times a question explicitly tells you to include Tx time or exclude it. Therefore be vigilant.)

(4) Let's revise quickly what we learned last time about HTTP.

Problem:

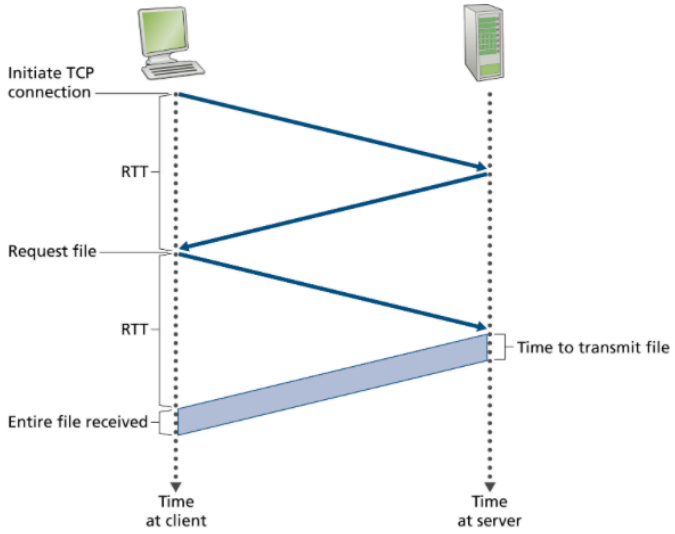
As per the service model of the layered network, a layer performs its functions by doing some work by itself and utilizing the services provided by the layer(s) below it. **HTTP uses TCP from the layer below (the transport layer).**

Each new TCP connection to a peer takes at least 2 round trips time (RTT) before the requested object start coming in at the client.

Also, **TCP slow-start phase** where TCP connection is learning how much capacity network has by being very cautious. (More details in next chapter.)

Client can piggyback HTTP request on the third message of the three-way TCP handshake. Still, it takes two RTTs before the requested object start coming at the client. A typical webpage can have tens of URLs to different objects (such as logos, pictures, videos etc.). Incurring 2 RTTs per object will add **substantial user-visible latency** in rendering the page in the browser.

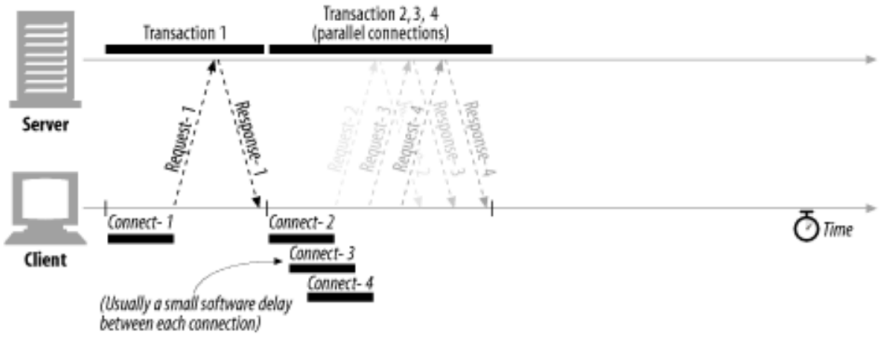
Figure 2.7: Back-of-the-envelope calculation for the time needed to request and receive an HTML file



Source: Computer Networks: A top down approach, 9th edition

Solution 1: Use multiple TCP connections between the client (browser) and the HTTP server (Serial to parallel):

The connection delays are overlapped in time, so it might feel as if page is loading faster.



Source: https://learning.oreilly.com/library/view/http-the-definitive/1565925092/ch04s04.html

How many parallel connections to use depends on the browser. Usually, 4 to 6 is a typical number. Though by using multiple TCP connections, this specific user might be using "unfair" amount of available bandwidth. Remember Internet is a packet-switched network with statistical multiplexing (translation: sharing resource among parties needing the resource) and using multiple parallel connection might mean that you are trying to get more than your fair share of the "pie".

Solution 2: Request HTTP server to use persistent TCP connection:

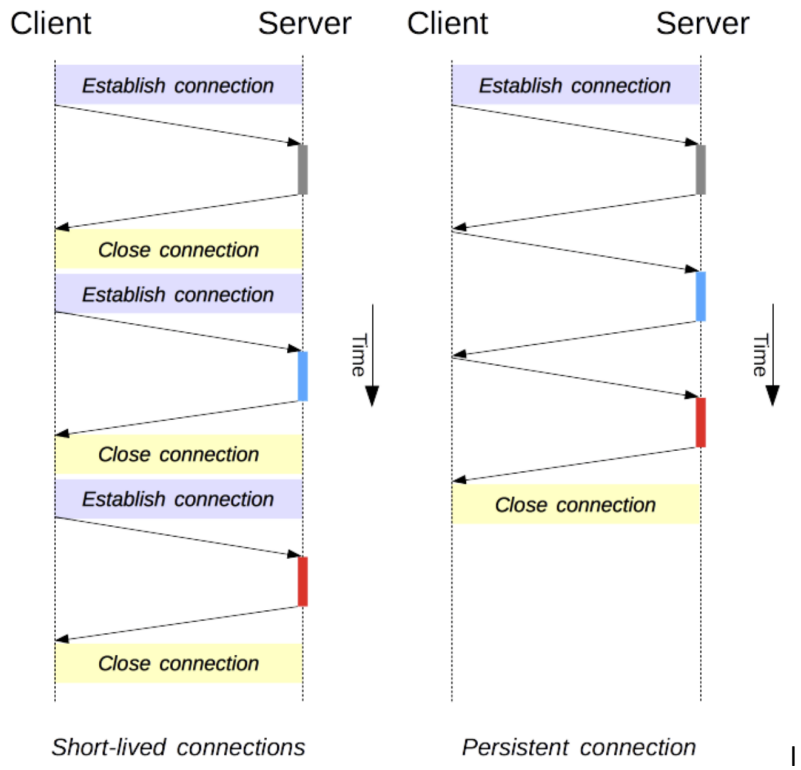
Simple Idea: Ask server to keep the TCP connection open.

Persistent connection's ability was added in HTTP/1.1

Using TCP headers:

```
Connection: keep-alive
```

```
Connection: close
```



Source: https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x

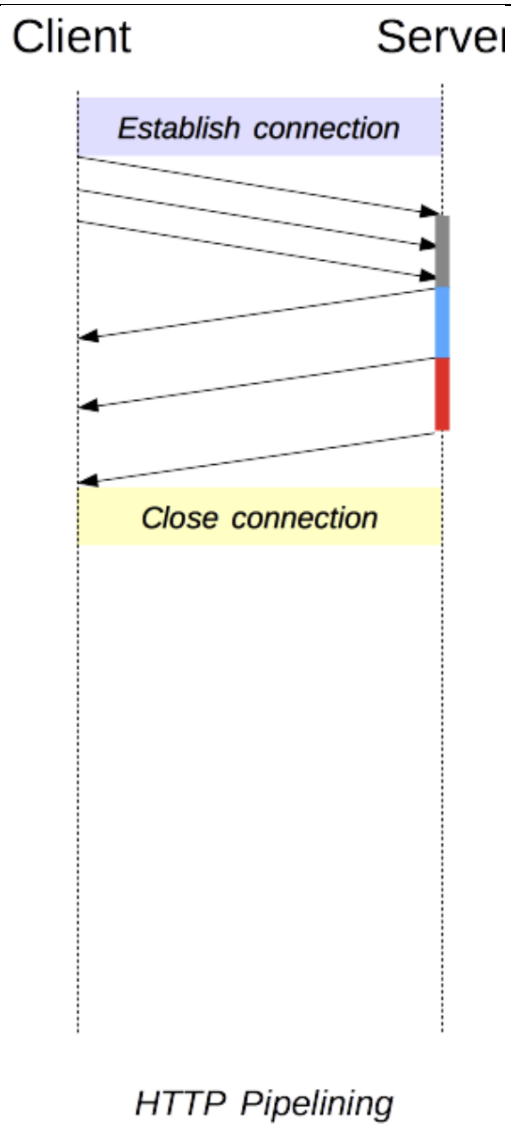
It depends on the server for how long it keeps a TCP connection open with a specific client. It is a configurable time in the server. A server might drop idle persistent connection in favor of serving new incoming requests.

Problem:

Persistent connection is an improvement. But we still incur an RTT to request the next object because we request and object, gets its reply, then request the next one and so on.

Solution 3: Pipeline the requests and responses

Client sends requests for all the objects back-to-back. The response for those requests come in order from the server (the same order as they were requested).



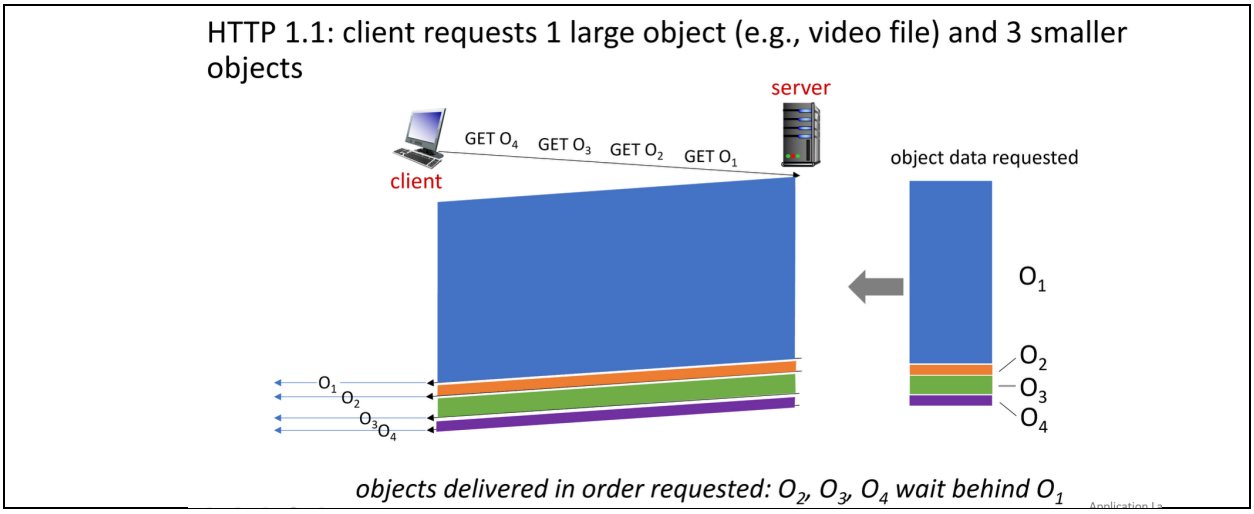
HTTP Pipelining

Source: https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x

Introduced in HTTP/1.1

Problem:

Potential head-of-line blocking where small-size objects are stuck behind a large object.



Example:

In **HTTP/1.1 pipelining**, the server must send responses **in the same order** as the requests arrived.

- Suppose the client pipelines 3 requests:
  - `GET /index.html` (heavy dynamic page, takes 500 ms to generate)
  - `GET /style.css` (cached, takes 5 ms)
  - `GET /logo.png` (small static file, takes 5 ms)
- Because responses must be sent **in order**:
  - The CSS and logo can't be sent until the HTML is ready.
  - So the browser is waiting **500 ms** for *all* three, even though two were instantly available.

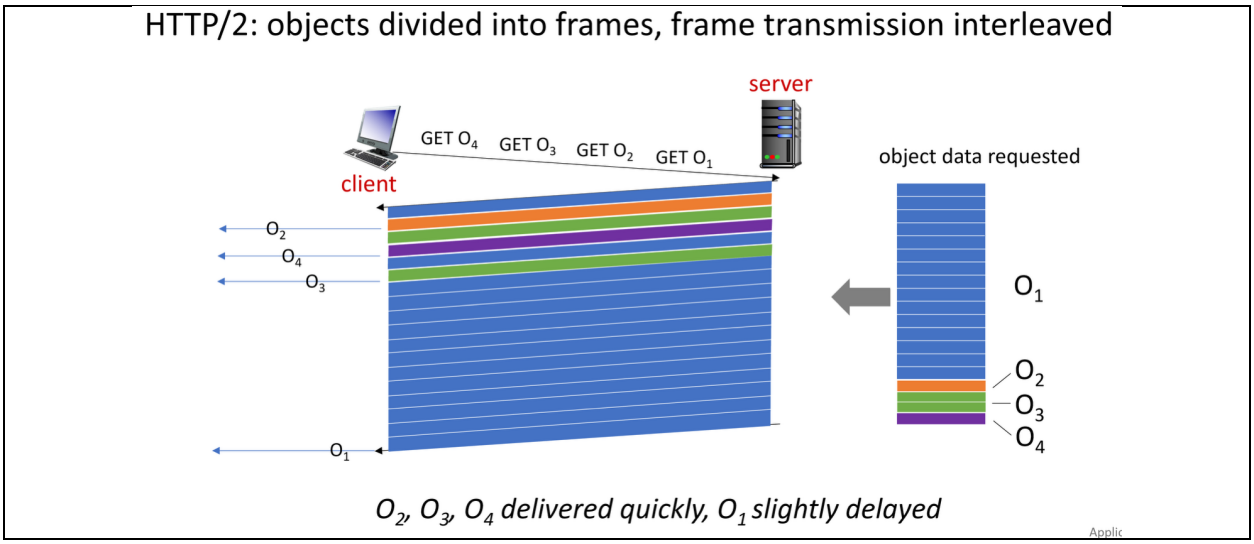
👉 This is **head-of-line blocking**: one slow item at the "head" of the queue blocks everything behind it.

Source: Example generated by AI tool

Solution4: Client request in a "good" order

If a client can guess the size of the objects, client might put the request in a "good" order. For example, shorter sized objects first. But it is not always possible for the client to correctly guess object sizes because some objects are generated on-the-fly by the server whose size is unknown apriori.

Solution 5: Multiplex “frames” of requested objects in the response + let the client set the priority of the objects



Further tweaks in HTTP/2

(a) **[HTTP Server Push]** Server might send sub-objects in a page to the client without client explicitly sending in the request. Server assumes that client will request them anyway. Doing so, we can save one RTT when client will parse the main page and send in requests for the sub-objects. Helps for apps that need low-latency processing of events. Instead of client **polling**, server can tell when some event occurs. Of course, client-server need a persistent connection for that.

(b) **[Binary Format]** HTTP 2.0 shifted from ASCII text to Binary format. There are many reasons. shifting to a **binary format** allowed HTTP/2 to achieve much higher performance, improved reliability, and better support for modern web requirements, such as multiplexing and efficient resource usage.

(c) **[Security]** Traditional TCP sockets did not provide any security (confidentiality, integrity, peer authentication). Transport Layer Security (TLS) incorporated security using application-level libraries.

Problem:

Data loss can stall a TCP flow until it is trying to recover from the loss (acks + data re-send). Let’s assume data of the first blue “frame” is lost. It can stall next orange “frame” from transmission. Still a form of head-of-line blocking but coming due to TCP loss and recovery.

(There is fine print here. I made above scenario extra simple for understanding. TCP usually does not stall right away with a single loss. But let’s not worry about it now. We will study it in the next chapter.)

Solution 6: Do not use TCP

If you think carefully, many of application's challenges are due to some behavior (or side-effect) of TCP.

Idea: Application takes on more responsibility if it needs more controlled behavior.

Idea: Let's use UDP where application will use some of TCP's techniques to deal with:

(a) Flow control
(b) Congestion control
(c) Reliability

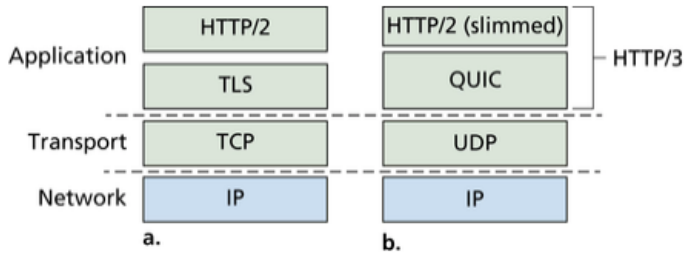
But without the negative side effects for the application.

Idea: Each request is a separate "stream". Any UDP packet loss in one stream should not impact other streams whose data is not lost.

First invented by Google (SPDY protocol) now a standard as QUIC (Quick UDP Internet Connection) and used by HTTP/3

Trick question: Is QUIC a transport or application layer protocol?

Figure 2.11: a. traditional secure HTTP protocol stack; b. secure QUIC-based HTTP/3 protocol stack

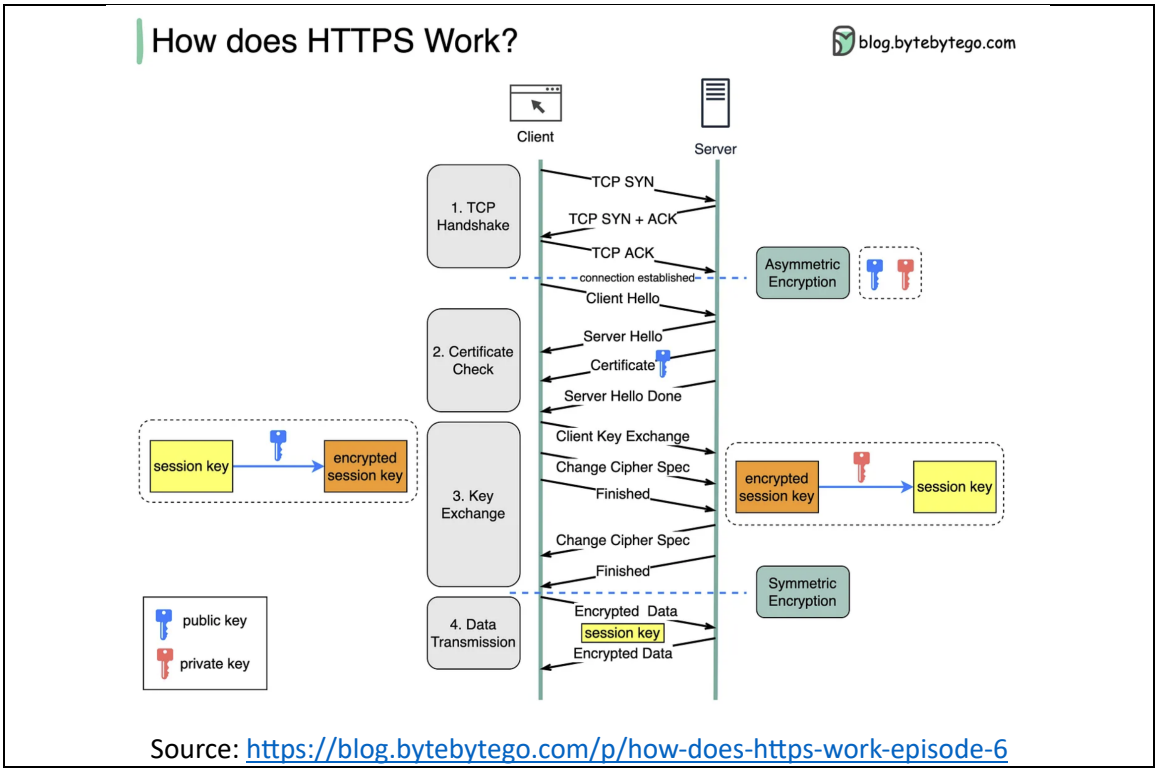


Source: CN Textbook

Remember: HTTP/3 did not obsolete HTTP/2

Problem:

A typical HTTP connection using TLS still takes quite a lot of RTTS before HTTP application peers could securely talk. Can we further reduce this delay?



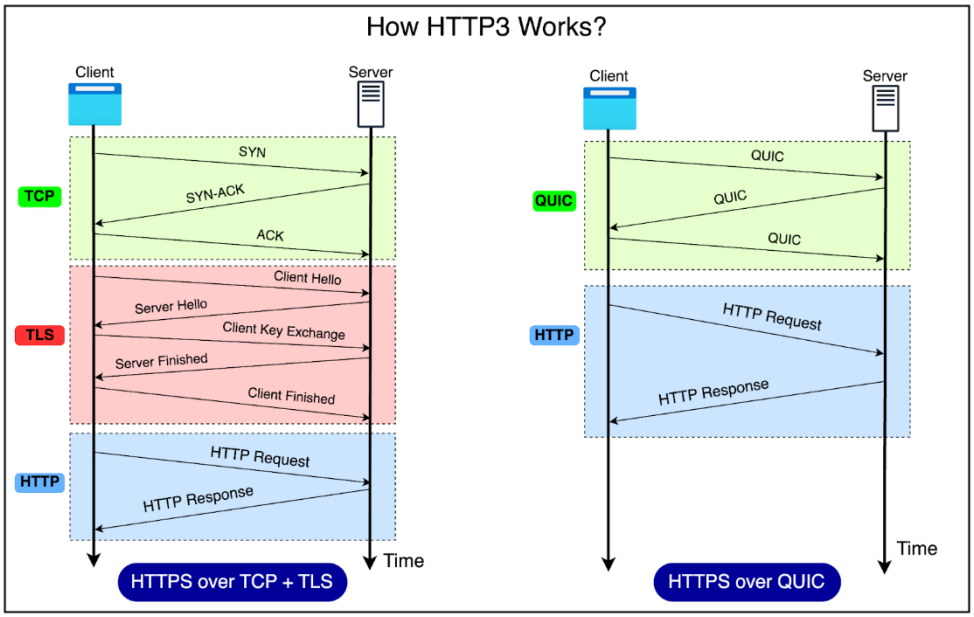
Source: https://blog.bytebytego.com/p/how-does-https-work-episode-6

Solution 7: Merge TLS security stuff with QUIC

QUIC uses TLS 1.3 by default (client do not have choice not to use it. Security is that important in 2025!)

Idea: Again, as an app if I need finer control over things, I can't leave it to "others".

Philosophical point: The storyline of UDP/HTTP 3.0 is a counter example against network layering and abstractions.



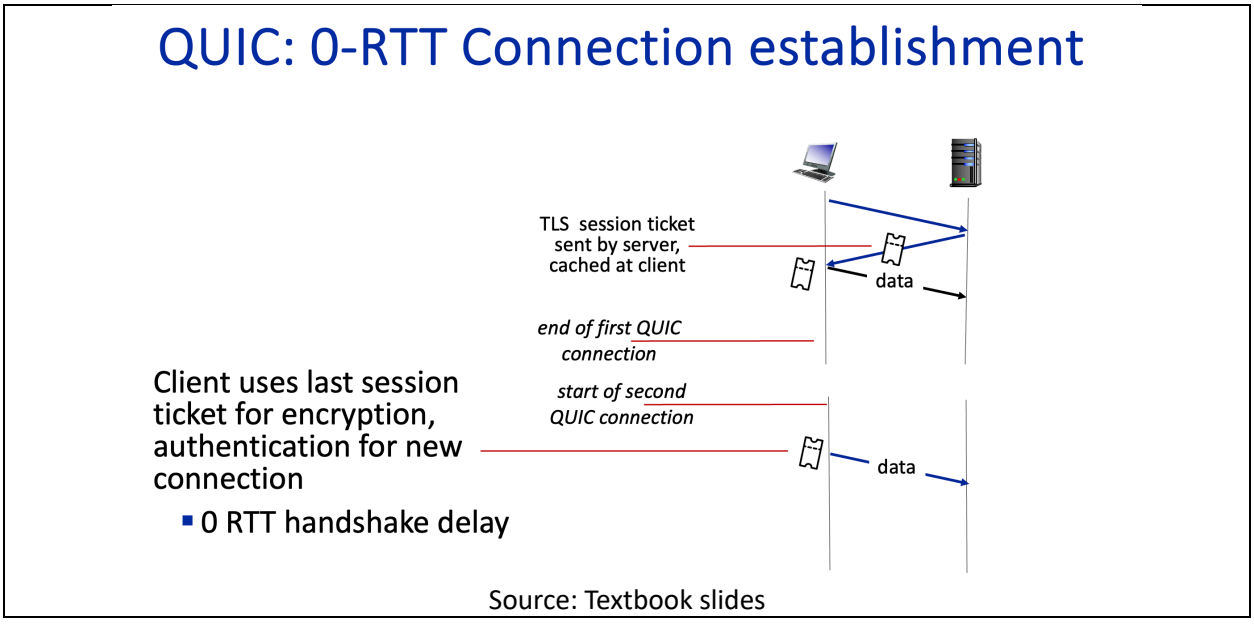
Source: https://blog.bytebytego.com/p/http1-vs-http2-vs-http3-a-deep-dive

Problem:

Can we achieve 0-RTT (meaning client able to send request right away)?

Solution 8: Use longer-term TLS sesson tickets

Quick UDP Internet Connection (QUIC) does the necessary handshakes in fewer RTTs. It can do so if a client-server pair have communicated before and there is valid "TLS session ticket".



Source: Textbook slides

Which HTTP version introduced what?

Please read the book!

We expect you to know which HTTP version introduced which feature.

Feature	HTTP/0.9	HTTP/1.0	HTTP/1.1	HTTP/2.0	HTTP/3.0
Year Introduced	1991	1996	1997	2015	2020
Primary Use Case	Simple, text-based requests	Full-featured protocol	Persistent connections	Multiplexing, performance	QUIC-based, faster connections
Request Methods	Only GET	GET, POST, HEAD	GET, POST, HEAD, PUT, DELETE	Same as 1.1	Same as 1.1
Response Format	Only HTML	HTML, images, files	Supports multiple content types	Binary framing	Binary framing over QUIC
Status Codes	No status codes	Basic status codes (200, 404, etc.)	Full set of status codes	Full set of status codes	Full set of status codes
Persistent Connections	No	No	Yes (default)	Yes	Yes
Connection Handling	Single connection per request	Single connection per request	Keep-Alive (multiple requests per connection)	Multiplexing (multiple streams over a single connection)	Multiplexing over QUIC
Compression	None	None	Optional (Gzip)	Header compression (HPACK)	Header compression (QPACK)
Multiplexing	No	No	No	Yes	Yes
Protocol Layer	Application Layer (TCP-based)	Application Layer (TCP-based)	Application Layer (TCP-based)	Application Layer (TCP-based)	Application Layer over QUIC (UDP-based)
Security (Encryption)	No	Optional (via SSL/TLS)	Optional (via SSL/TLS)	Mandatory TLS	Mandatory TLS 1.3
Header Structure	No headers	Plain text, no compression	Plain text, no compression	Binary, compressed (HPACK)	Binary, compressed (QPACK)
Server Push	No	No	No	Yes	Yes
Transport Protocol	TCP	TCP	TCP	TCP	QUIC (UDP-based)

Trivia

- **HPACK** = Header compression scheme for HTTP/2.
- **QPACK** = Header compression scheme for HTTP/3 (QUIC).

Next Problem: Repeatedly fetching objects that are not changed at the server

Every time we fetch a webpage, few objects do not change. For example, the logo of FAST when we visit www.nu.edu.pk and so on. So why waste bandwidth fetching such objects repeatedly?

Solution 9: Conditional GET

Before we see the solution, let's see how a typical HTTP request and response looks like.

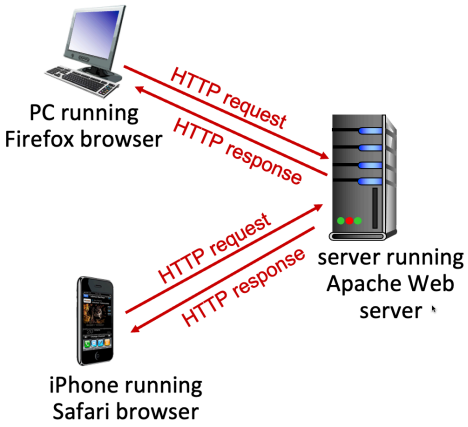
- web page consists of *objects,* each of which can be stored on different Web servers
- object can be HTML file, JPEG image, Java applet, audio file,…
- web page consists of *base HTML-file* which includes *several referenced objects, each* addressable by a *URL,* e.g.,

`www.someschool.edu/someDept/pic.gif`

host name | path name

HTTP: hypertext transfer protocol

- Web's application-layer protocol
- client/server model:
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests



HTTP uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"

- server maintains *no* information about past client requests

aside

protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

HTTP request message

- two types of HTTP messages: *request*, *response*
- HTTP request message:
  - ASCII (human-readable format)

carriage return character
line-feed character

request line (GET, POST, HEAD commands)

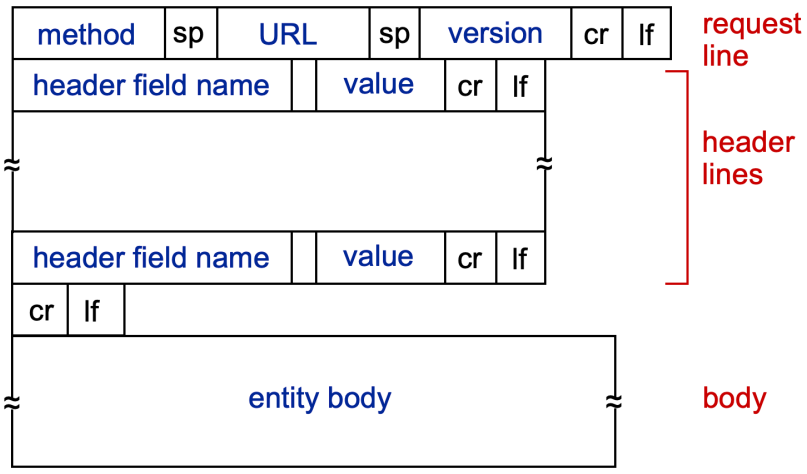
header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
   10.15; rv:80.0) Gecko/20100101 Firefox/80.0 \r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Connection: keep-alive\r\n
\r\n
```

carriage return, line feed at start of line indicates end of header lines

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

HTTP request message: general format



Other HTTP request messages

POST method:

- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

GET method (for sending data to server):

- include user data in URL field of HTTP GET request message (following a '?'):

```
www.somesite.com/animalsearch?monkeys&banana
```

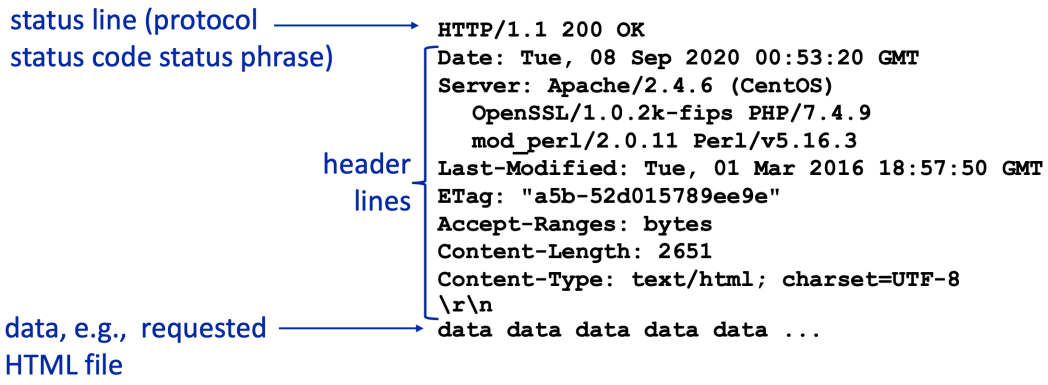
HEAD method:

- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

PUT method:

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

HTTP response message



* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Request:

```
GET /fruit/kiwi.gif HTTP/1.1
Host: www.exotiquecuisine.com
```

Response:

```
HTTP/1.1 200 OK
Date: Sat, 3 Oct 2015 15:39:29
Server: Apache/1.3.0 (Unix)
Last-Modified: Wed, 9 Sep 2015 09:23:24
Content-Type: image/gif

(data data data data data ...)
```

Conditional GET:

```
GET /fruit/kiwi.gif HTTP/1.1
Host: www.exotiquecuisine.com
If-modified-since: Wed, 9 Sep 2015 09:23:24
```

Response:

```
HTTP/1.1 304 Not Modified
Date: Sat, 10 Oct 2015 15:39:29
Server: Apache/1.3.0 (Unix)

(empty entity body)
```

Typical HTTP status code groups:

Table 27. Five Types of HTTP Result Codes.

Code	Type	Example Reasons
1xx	Informational	request received, continuing process
2xx	Success	action successfully received, understood, and accepted
3xx	Redirection	further action must be taken to complete the request
4xx	Client Error	request contains bad syntax or cannot be fulfilled
5xx	Server Error	server failed to fulfill an apparently valid request

Virtual hosting:

Question: Why is there host line in the HTTP request? Why not merge it with URL?

URI and URL:

- **URI (Uniform Resource Identifier):** A general string that identifies a resource by name, location, or both.
  Example: mailto:someone@example.com (identifies resource via email scheme).
- **URL (Uniform Resource Locator):** A type of URI that specifies the *location* of a resource and how to access it.
  Example: https://example.com/index.html.

Answer:

Before HTTP/1.1 one IP was hosting one webserver. Host header was introduced in HTTP/1.1 so that multiple websites could be hosted by a single webserver (possibly on a single IP). It is called shared hosting. Server reads this header to return appropriate website.

Virtual hosting is the technique of running **multiple websites (domains) on a single web server and IP address**.

Example: example.com and example.org both point to the same server IP, but the server serves different content depending on which hostname the client requested.
This is extremely common in shared hosting environments.

The Problem with HTTPS and Virtual Hosting
With **HTTP** + virtual hosting: the Host header tells the server which site you want.
But in **HTTPS**, the HTTP headers (including Host) are encrypted inside the TLS session.
The server needs to know **which site's certificate** to present **before** decryption begins — chicken-and-egg problem.

Solution: SNI
SNI (Server Name Indication) is an extension to TLS.
During the **TLS handshake**, the client includes the **hostname** it wants to connect to.
This lets the server select the correct TLS certificate (and therefore the right virtual host) before the encrypted HTTP request arrives.

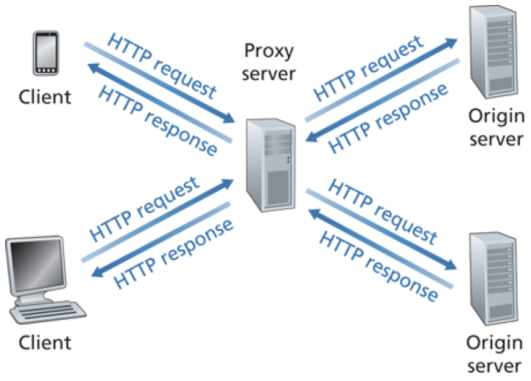
Example flow:
Client says: *"Hello, I want to talk to* www.example.com*"* → (SNI field in ClientHello).
Server picks the correct certificate for www.example.com.
Encrypted HTTP session begins.
Inside the HTTP request, client also sends Host: www.example.com (for HTTP-layer virtual hosting).

Problem: But what if multiple parties are asking for the same object again and again?

In an organization like FAST, what if many students are fetching www.nu.eud.pk multiple times in a day. FAST's logo will still be fetched once and cached locally in the browser cache. But can we do better? Can we have a cache for the organization?

Solution 10: Web Caching vi a proxy server (at times called forward proxy)

Figure 2.11 **Clients requesting objects through a Web cache**



An aside:

When HTTPS was not widely used, web caches were common.
Now, when HTTPS is a norm, parties who have session key can cache only.
Example: Client's browser cache
Front-end server close to the customer where a TCP connection terminate

Let's switch to second slide deck (already posted on GCR), at slide no 42 for caches discussion.

Problem: How to do stateful work on stateless HTTP?

Remember HTTP is a stateless request-response protocol. But many applications need state across multiple requests.

Examples: Buying books online, logged-in customer of bank etc.

Solution 11: Use cookies to create application-level state

Let's switch to second slide deck for edition 9 of the textbook (already posted on GCR), at slide no 32 for state discussion.

That completes our overview of the HTTP protocol. See the wiki page of HTTP or read the relevant HTTP RFCs for more details.