CSE 3231
Computer Networks

Chapter 7
Applications
part 1

William Allen, PhD

Spring 2022

Application Layer

The application layer interfaces with the transport layer to isolate applications from the details of packet delivery

- Depending on its goals, the application uses either UDP or TCP to manage network traffic
  - As we know, if the application needs a reliable connection between endpoints, it can use TCP
  - Otherwise, it may use UDP
  - Some applications use UDP, but add features like acknowledgements on their own to get more reliable delivery without TCP's overhead

Application Protocols

- Applications that support interaction or transfer data usually have a *protocol* for communication between nodes
  - One endpoint may act as a *server*, collecting and delivering data or messages
  - Other endpoints may act as *clients*, requesting or providing data or messages
- This application protocol describes how the endpoints will *interact* to accomplish the designer's requirements

Application Protocols

- Application protocols may be open and available through Internet documents or public websites or may be proprietary
  - The *Internet RFC* (Request For Comment) documents also describe application protocols for standardization across vendors
    - https://www.rfc-editor.org/rfc-index.html
  - Examples include:
    - Simple Mail Transfer Protocol (SMTP) *RFC 5321*
    - Hypertext Transfer Protocol (HTTP) *RFC 2616/7540*
    - File Transfer Protocol (FTP) *RFC 959*

Application Protocols

- Based on a clear application protocol, developers can create servers and/or clients that will interact in *established* and *predictable* ways
  - However, some inconsistencies occur when developers decide to add features that are not part of the protocol
    - Sometimes they are hoping to encourage their addition or it's simply for marketing purposes
    - Also, some developers lag behind in updating their applications to new protocol versions

Application Protocols

- Generally, servers *listen on well-known ports* for connections from clients
  - Some servers will also act as a client to exchange data with other servers
  - Clients connect to the server through these known port numbers but their source port does not have to be publicly known, it is provided to the server during connection
  - After that, the client and server follow the protocol to exchange messages and data

Well-Known Port Numbers

- A range of UDP and TCP ports are assigned to specific protocols, others are not reserved
  - Ports numbered 0-1023 are reserved by the Internet Assigned Numbers Authority (IANA)
  - Servers listen on specific ports to provide services
    - SSH - port 22
    - HTTP - port 80
    - NTP (Network Time Protocol) - port 123
    - IMAP (email) - port 143 or port 220
    - LDAP (authentication protocol) - port 289
    - HTTPS (using TLS/SSL) - port 443

Connection to a Well-Known Port

This example shows a normal TCP connection starting with a 3-way handshake

Time	Source	Destination	Protocol	Length	Info
0.000000	128.208.2.151	74.125.127.104	TCP	66	56816 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
0.008026	74.125.127.104	128.208.2.151	TCP	66	80 → 56816 [SYN, ACK] Seq=0 Ack=1 Win=5720 Len=0 MSS=1430
0.008058	128.208.2.151	74.125.127.104	TCP	54	56816 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0

- The client application requests a connection with the application server that is bound to port 80 (HTTP), the client is using port 56816 for its end of the connection
- The server uses port 56816 as the destination port when it *replies* to the request and that combination of ports may be used for the remainder of this connection

Suggestion for Debugging

- When writing and testing network programs, debugging can be more difficult because you are sending packets to another computer and then waiting for replies to those packets
- If you use tcpdump or WireShark to monitor the out-going and in-coming traffic, you can see if your packets were correctly sent and if the reply arrived at the network interface, but your program didn't correctly process it
  - you can look at the packet's details to see if there are errors in the IP address, port, data, etc.

Request/Reply Protocols

- The most common client/server interaction is through a *request/reply protocol* where specific messages are transmitted by the client and server to manage the interaction and exchange data
- For example, a server may begin by notifying the client of the version number, language, data format and protocol options it supports
  - The client may then conform to the default set of specifications or may exchange messages with the server to *negotiate* which options it will use

Request/Reply Protocols

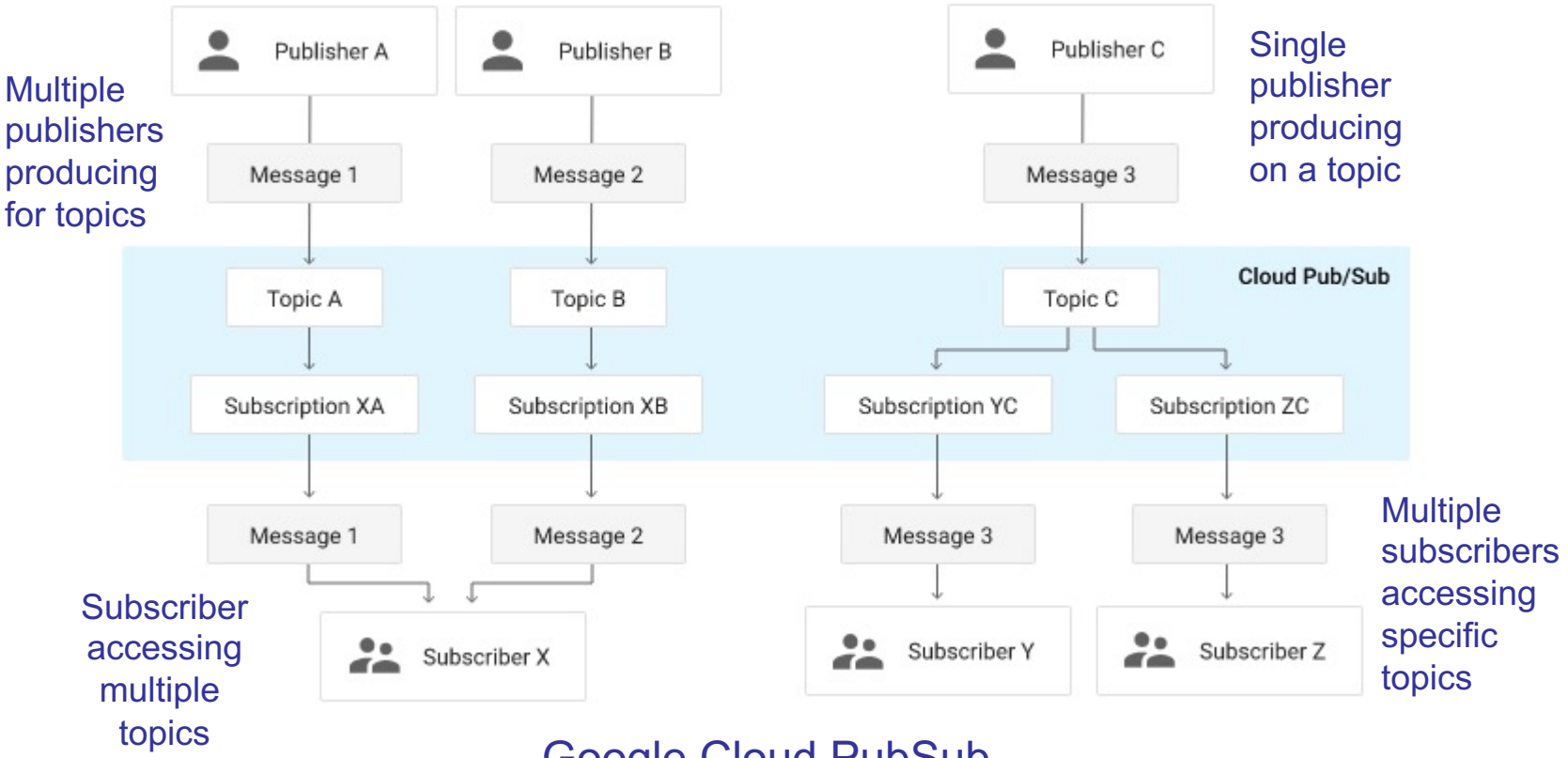
- Request and Reply messages often consist of text commands or pre-defined numbers or a combination of both types of command
- The protocol may require that the client and server both keep track of the current state of the exchange of messages
  - this is referred to as a "*stateful*" protocol
- Or, the server may not be required to keep track of the state of the exchange and may close the connection after each message
  - this is referred to as a "*stateless*" protocol

Alternative Protocols

- A *publish-subscribe* protocol (PubSub) is another type of interaction for data sharing
  - *publishers* do not wait for a client to request data but make it available for *subscribers* who can register to receive specific types of message
  - publishers are *loosely coupled* to subscribers and produce content whether it is being used or not
  - since there is no requirement that subscribers request data, this approach has *better scalability* because publishers don't have to manage client connections or wait for requests to publish content
    - IP multicasting may be used for data sharing

Publish-Subscribe Protocols

Examples include Apache Kafka, Google Cloud Pub/Sub and the Data Distribution Service (DDS)



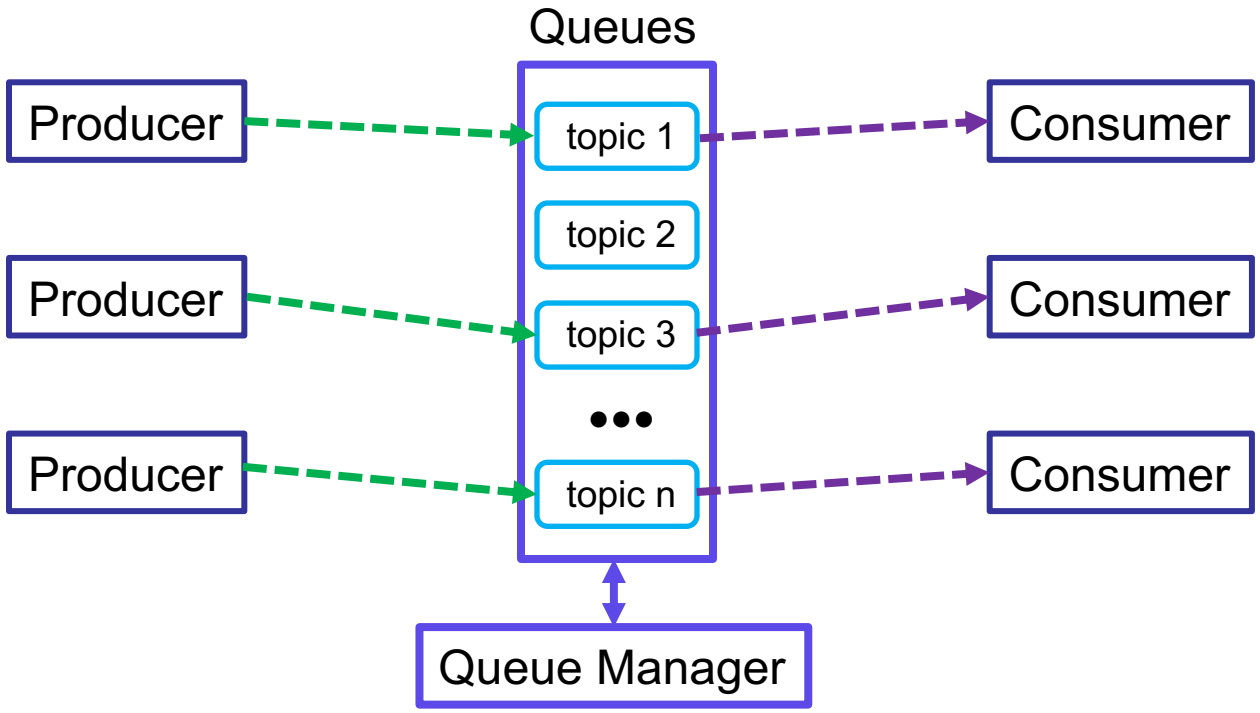
Google Cloud PubSub

Alternative Protocols

- *Message queueing* protocols are similar to PubSub protocols in that they do not require that servers wait until a client requests data
  - A *queue manager* is implemented and its location is announced to potential users
  - Applications can *register* to be notified when messages arrive in a specific queue and then download those messages when they are ready
  - Applications can also add messages to a queue
  - Queueing *decouples* senders and receivers so that senders do not have to wait for a request before they add new data to the queue

Message Queueing Protocols

Examples of Message Queueing protocols include Apache ActiveMQ, Microsoft Message Queueing, the Java Message Service and many others



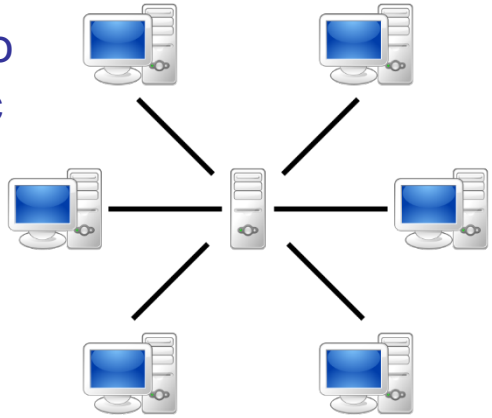
Alternative Protocols

- *Peer-to-peer protocols* allow each node to exchange messages with any other node
  - effectively, they can be both a client and a server
  - data storage is normally *decentralized* and data is passed between peers to update local storage
  - not all nodes may have the same capabilities, some peers may be able to perform specialized services that other peers need done for them
  - peer networks can have a *structured* organization that fits a specific topology to support efficiency and regular updates or they can be *unstructured* and allow rapid changes to adapt to conditions

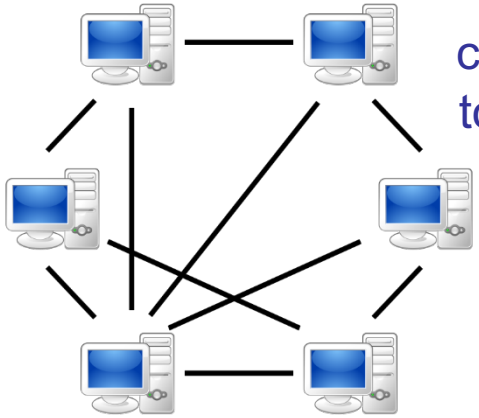
Peer-to-Peer Protocols

There are many peer-to-peer protocols, some of which are for specialized applications, such as Bitcoin, and others support distributed file sharing, such as BitTorrent and Gnutella

Nodes connect to a specific server



Classic Client/Server Connections



Any node can connect to any other node

Connections in a Peer-to-Peer Network

HyperText Transfer Protocol (HTTP)

HTTP (HyperText Transfer Protocol) is the protocol for connections between the client (browser) and servers in the World Wide Web

- the idea of linking information in documents was presented in an article by Vannevar Bush in 1945
- Ted Nelson coined the term *hypertext* in 1963 and helped create a simple system using hyperlinks
- Douglas Engelbart demonstrated a system in 1968 with a user interface that linked different tools and documents (it also used the *first mouse*)
- In 1990, *Tim Berners-Lee* created a hypertext data sharing system he called the *WorldWideWeb*

World Wide Web

The original WWW server - programmed by Tim Berners-Lee, 1990

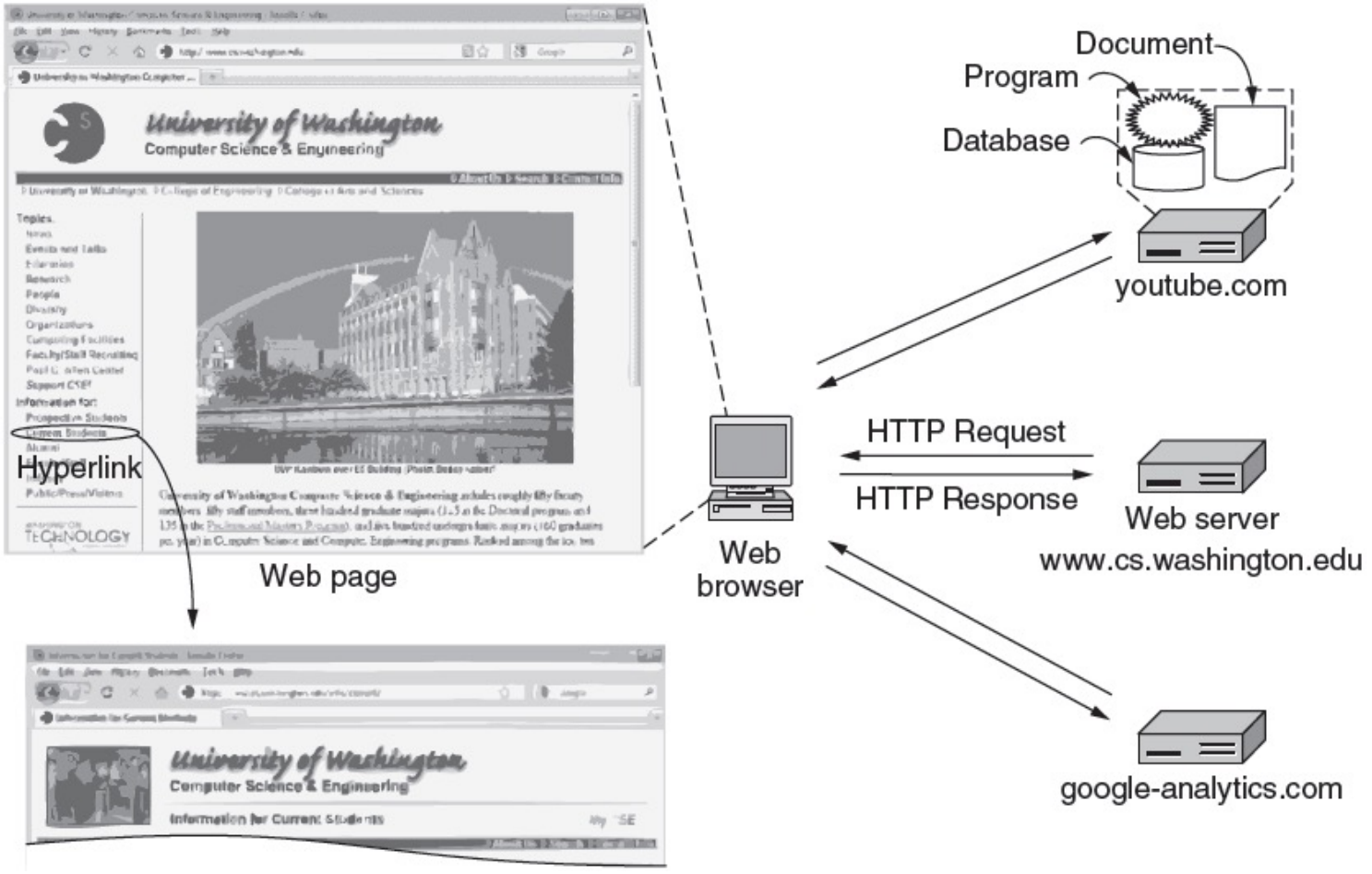
HTTP

HTTP provides a *request/response* interaction between a server and a number of clients

- The web browser acts as a *User Agent* and communications are based on TCP
- HTTP is a *stateless* protocol that originally consisted of individual request/response interactions followed by a disconnection
  - a *keep-alive* feature was added in version 1.1
  - the problem of retaining state information was solved by either passing variables in messages or storing data in "*web cookies*" on the client's host

HTTP

HTTP transfers pages from servers to browsers



Cookies Retain State Information

Web cookies can be used to support *stateful* client/server interactions

- Server sends cookies (state) with its response
- Client stores the cookies locally
- When the client makes a new request to the same server, it sends the cookies with the request
- Server extracts state information from the cookie

Domain	Path	Content	Expires	Secure
toms-casino.com	/	CustomerID=297793521	15-10-10 17:00	Yes
jills-store.com	/	Cart=1-00501;1-07031;2-13721	11-1-11 14:22	No
aportal.com	/	Prefs=Stk:CSCO+ORCL;Spt:Jets	31-12-20 23:59	No
sneaky.com	/	UserID=4627239101	31-12-19 23:59	No

Examples of cookies

HTTP

Identification of servers and hyperlinks is based on *Uniform Resource Locators* (URLs) which contain information needed to access a target server and a specific document on that server

- HTTP headers are text-based and use a number of standard header fields to manage connections and exchange data
- HTTP connections are not encrypted but an update, called *HTTPS*, creates an encrypted connection before data is exchanged
  - this protects data but does not authenticate users

Uniform Resources Locators (URL)

Uniform Resource Locators (URLs) can identify a number of locations and delivery protocols

Example: `http://www.phdcomics.com/comics.php`

Protocol **Server** **Page on server**

- A URL begins with the *protocol* it will connection to
- It can specify both a *domain name* (server) and a path to a *specific file* on that server
- URLs can also specify email accounts, local files, links to FTP servers and other sources of data
  - standard protocols and data types are described in RFCs

HTTP Messages

HTTP is text-based *request/response* protocol where every message has the general form:

```
START_LINE <CRLF>
MESSAGE_HEADER <CRLF>
<CRLF>
MESSAGE_BODY <CRLF>
```

- The first line (`START_LINE`) indicates whether this is a request message or a response message
  - Each section ends with a CRLF

Client Requests

Steps a browser takes to follow a URL:

- Determine the protocol (e.g., HTTP, HTTPS)
- Ask DNS for the IP address of the server
- Make a TCP connection to the server
- Send a request for the desired page
  - wait for the server to send it back
- Fetch other URLs as needed to display all of the components in the page
  - using DNS again, as needed
- Close any idle TCP connections

Request Messages

- The first line of an HTTP request message specifies three things: the operation to be performed, the web page it should be performed on, the HTTP version being used
- HTTP defines a wide assortment of request operations—including “write” operations that allow a Web page to be posted on a server
- Two common operations are
  - GET (fetch the specified web page)
  - POST (send data to a web server)

HTTP

Many headers carry key information:

Function	**Example Headers**
Browser capabilities (client → server)	User-Agent, Accept, Accept-Charset, Accept-Encoding, Accept-Language
Caching related (mixed directions)	If-Modified-Since, If-None-Match, Date, Last-Modified, Expires, Cache-Control, ETag
Browser context (client → server)	Cookie, Referrer, Authorization, Host
Content delivery (server → client)	Content-Encoding, Content-Length, Content-Type, Content-Language, Content-Range, Set-Cookie

HTTP

HTTP has several different request methods

Operation	Description
OPTIONS	Request information about available options
GET	Retrieve document identified in URL
HEAD	Retrieve metainformation about document identified in URL
POST	Give information (e.g., annotation) to server
PUT	Store document under specified URL
DELETE	Delete specified URL
TRACE	Loopback request message
CONNECT	For use by proxies

HTTP request operations

HTTP Request

```
GET /download.html HTTP/1.1
Host: www.ethereal.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1;
en-US; rv:1.6) Gecko/20040113
Accept:
text/xml,application/xml,application/xhtml+xml,text/htm
l;q=0.9,text/plain;q=0.8,image/png,image/jpeg,image/gif
;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.ethereal.com/development.html
```

The Server's Response

Steps a server takes to serve pages:

- Accept a TCP connection from a client
- Receive the page request and map it to the requested resource to create the response
  - This resource may be a static page or may require executing a program which creates the reply
- Send the reply to the client
  - The reply may include links to other resources that the client's browser will have to access on its own
- Release any idle TCP connections
  - HTTP is stateless, connections do not stay open

HTTP Response

- Like request messages, response messages begin with a START LINE
- In this case, the line specifies
  - the version of HTTP being used
  - a three-digit code indicating whether or not the request was successful
  - a text string giving the reason for the response

HTTP

Response codes tell the client how the request fared:

Code	Meaning	Examples
1xx	Information	100 = server agrees to handle client's request
2xx	Success	200 = request succeeded; 204 = no content present
3xx	Redirection	301 = page moved; 304 = cached page still valid
4xx	Client error	403 = forbidden page; 404 = page not found
5xx	Server error	500 = internal server error; 503 = try again later

HTTP Response

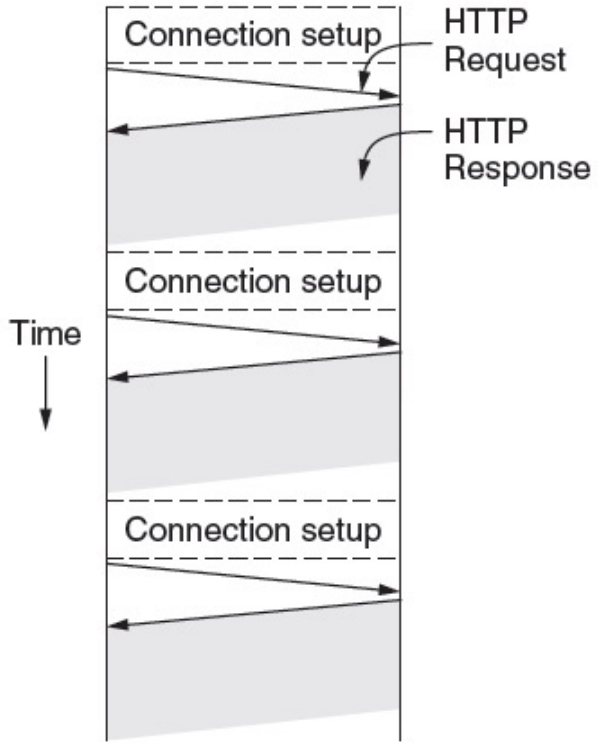
```
HTTP/1.1 200 OK
Date: Thu, 13 May 2004 10:17:12 GMT
Server: Apache
Last-Modified: Tue, 20 Apr 2004 13:17:00 GMT
ETag: "9a01a-4696-7e354b00"
Accept-Ranges: bytes
Content-Length: 18070
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=ISO-8859-1
```

HTTP

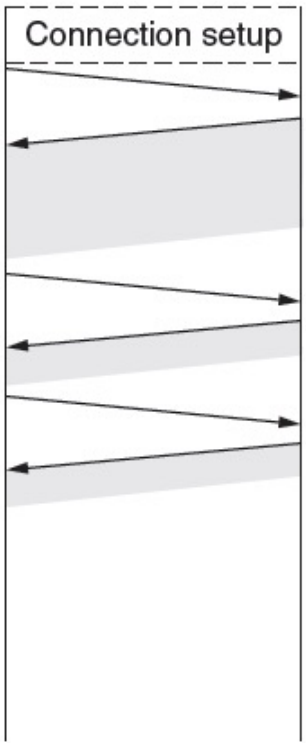
- HTTP servers normally listen on *port 80* and they support common MIME types to transfer non-text data between server and client
- While HTTP originally supported short-term connection for each individual request, it has been revised to allow *persistent* connections
  - this reduces the overhead from TCP connections
  - *sequential requests* exchange a series of requests without closing the connection until the link has been idle for a preset length of time
  - *pipelining of requests* allows overlapping a series of requests instead of doing them sequentially

HTTP

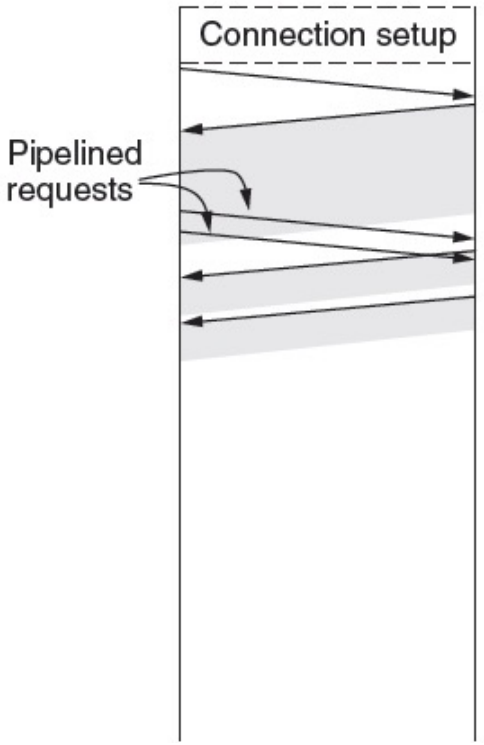
Persistent connections can improve performance



One connection for each request



Sequential requests over one connection



Pipelined requests on one connection

Managing HTTP Requests

To scale performance, Web servers can use *caching*, *multi-threading*, and a *front end* process that coordinates all of the requests

- caching reduces disk access for common requests
- concurrent requests are assigned to separate threads