Chapter 2: Application Layer

ATNLP: (layers)
Application
Transport
Network
Link
Physical

2.1 Principles of Network Applications:

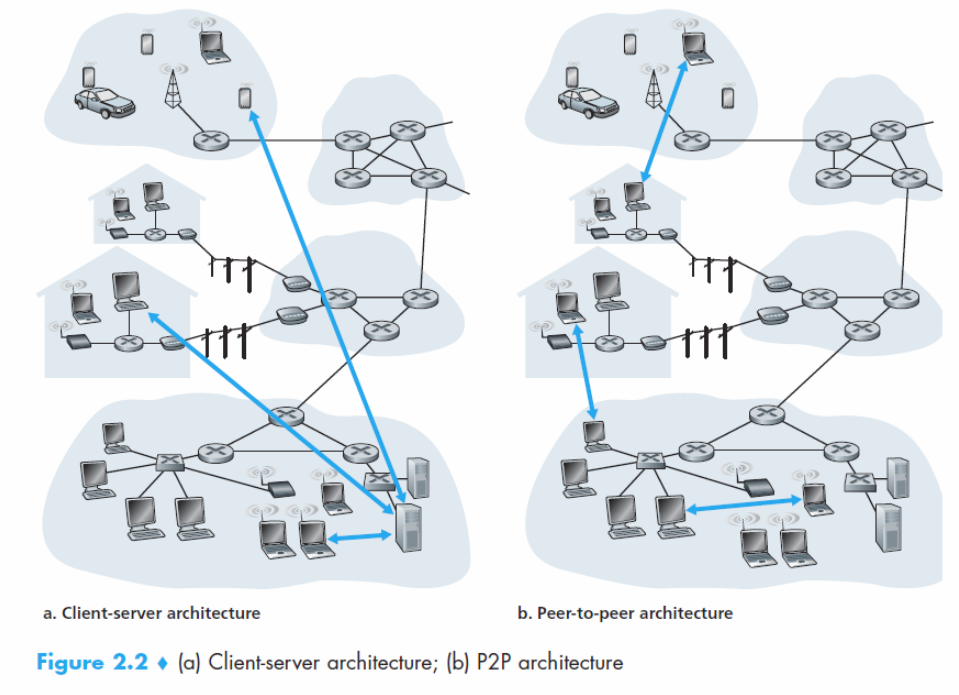
- Application architecture different from network architecture
- **Client-server architecture:**
  - Server = always-on host (reliance on some central server/datacentre always running)
  - Clients send requests to server
- **P2P architecture:**
  - little/no reliance on dedicated servers
  - Direct communication between connected peers/hosts
  - = self-scalable → peers both give and take
  - Challenges:
    - ISPs get pissed off (can't control P2P)
    - Security (no central system to secure - distributed nature)
    - Incentives (success depends on peers choosing to seed/upload instead of just take/leech from other peers)
  -
- 

Figure 2.2 ♦ (a) Client-server architecture; (b) P2P architecture

2.1.2 Process Communicating

- **Process** = program running within end system
- Processes on different end systems communicate by exchanging messages across network
- Process that initiates communication = **client**
- Process that waits to be contacted = **service**

- **Socket:** Processes send/receive messages through sockets
    - Process = house
    - Socket = door
    - Interface between application layer and transport-layer
- **Reliable Data Transfer** is provided by protocols that guarantee safe data delivery
    - If protocol provides this, sending process can simply pass data into socket and know with confidence that the data will arrive safely at destination
-

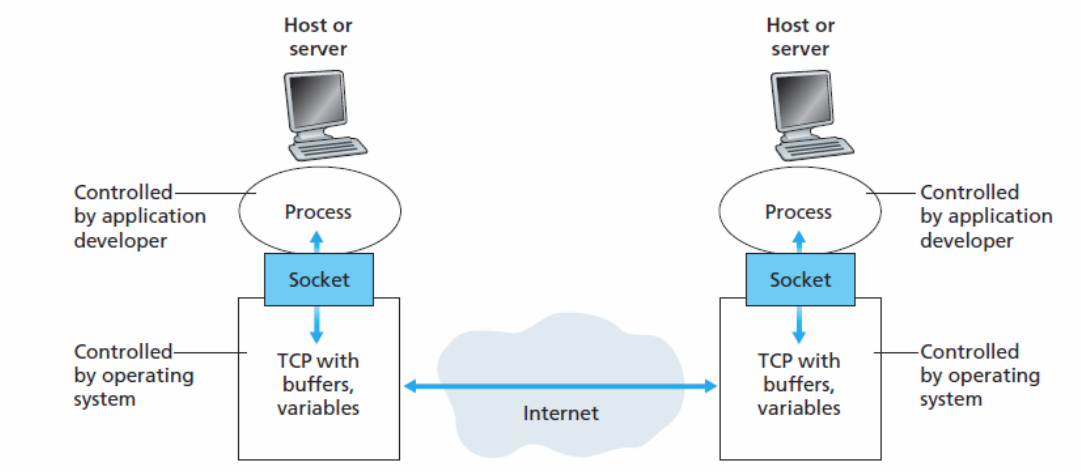


Figure 2.3 ♦ Application processes, sockets, and underlying transport protocol

-

Addressing protocols:

- Hosts are identified by 32-bit IP addresses
- Sockets (to which data are being sent to) are identified by **ports**

- If process on one host wants to send file to process on another host:
    - To identify the receiving process, two pieces of information need to be specified: (1) the address of the host and (2) an identifier that specifies the receiving process in the destination host.

Transport services available to applications

- **Reliable data transfer**
    - A bit of data loss only okay if dealing with loss-tolerant applications (ie: VoIP)
- **Throughput**
    - Applications with throughput requirements = **bandwidth-sensitive applications**

    - Applications without throughput requirements = **elastic applications**
- **Timing**
    - Lower delay obviously preferable to higher delay
- **Security**
    - Encrypt/decrypt transmitted data

Transport services provided by the internet:

- When creating network application, must decide whether to use **UDP** or **TCP**
- Choice depends on requirements of application
- **TCP:**
    - Connection-oriented service
        - Transport-layer **handshaking** procedure must occur between client and server before application-level messages can flow
        - Once connection established, two processes can send messages to one another over the connection.
        - When the client and server are finished sending messages, the connection must be terminated.
    - Reliable data transfer service
        - Can rely on TCP to deliver data intact and in proper order
    - Congestion control mechanism
        - Attempts to limit each TCP connection to its fair share of bandwidth
        -

SECURING TCP

Neither TCP nor UDP provide any encryption—the data that the sending process passes into its socket is the same data that travels over the network to the destination process. So, for example, if the sending process sends a password in cleartext (i.e., unencrypted) into its socket, the cleartext password will travel over all the links between sender and receiver, potentially getting sniffed and discovered at any of the intervening links. Because privacy and other security issues have become critical for many applications, the Internet community has developed an enhancement for TCP, called **Secure Sockets Layer (SSL)**. TCP-enhanced-with-SSL not only does everything that traditional TCP does but also provides critical process-to-process security services, including encryption, data integrity, and end-point authentication. We emphasize that SSL is not a third Internet transport protocol, on the same level as TCP and UDP, but instead is an enhancement of TCP, with the enhancements being implemented in the application layer. In particular, if an application wants to use the services of SSL, it needs to include SSL code (existing, highly optimized libraries and classes) in both the client and server sides of the application. SSL has its own socket API that is similar to the traditional TCP socket API. When an application uses SSL, the sending process passes cleartext data to the SSL socket; SSL in the sending host then encrypts the data and passes the encrypted data to the TCP socket. The encrypted data travels over the Internet to the TCP socket in the receiving process. The receiving socket passes the encrypted data to SSL, which decrypts the data. Finally, SSL passes the cleartext data through its SSL socket to the receiving process. We'll cover SSL in some detail in Chapter 8.

- **UDP:**
    - Connectionless

    - Light-weight
    - Minimal services
    - No handshaking
    - Unreliable data transfer service (no guarantee that message will arrive safely or in correct order)
    - No congestion control
- **TCP vs UDP comparison = NB**

Services not provided by Internet Transport Protocols:

- TCP and UDP (and all other internet transport protocols) make no throughput or timing guarantees
- Must design time/throughput sensitive applications with these lack of guarantees in mind

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP [RFC 5321]	TCP
Remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
File transfer	FTP [RFC 959]	TCP
Streaming multimedia	HTTP (e.g., YouTube)	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary (e.g., Skype)	UDP or TCP

Figure 2.5 ♦ Popular Internet applications, their application-layer protocols, and their underlying transport protocols

Figure 2.5 indicates the transport protocols used by some popular Internet applications. We see that e-mail, remote terminal access, the Web, and file transfer all use TCP. These applications have chosen TCP primarily because TCP provides reliable data transfer, guaranteeing that all data will eventually get to its destination. Because Internet telephony applications (such as Skype) can often tolerate some loss but require a minimal rate to be effective, developers of Internet telephony applications usually prefer to run their applications over UDP, thereby circumventing TCP's congestion control mechanism and packet overheads. But because many firewalls are configured to block (most types of) UDP traffic, Internet telephony applications often are designed to use TCP as a backup if UDP communication fails.

Application-Layer Protocols:

- = define how an application's processes - running on different end systems - pass messages to each other
    - **Types** of messages (request, response)
    - **Syntax** of messages (eg: fields)

    - **semantics**/meanings of fields
    - **Rules** (when/how to send/respond to messages)
- Some protocols open (HTTP), some closed/private (Skype)

2.2 The web and HTTP

- HyperText Transfer Protocol
- = Web's application-layer protocol
- Implemented in 2 programs: *client program* and *server program*
    - Client and server programs, being executed on different end systems, communicate using HTTP messages
    - HTTP defines structure of these messages
- **Webpage (document)** consists of **objects**
    - **Object** = file (HTML file, JPG file, etc.)
    - Most webpages have **base HTML file** and a few referenced objects
        - ie: webpage with 5 images and HTML text has 6 objects
        - **Base HTML file** references the other objects in the page with the **URLs** of the objects
        - An object's URL has 2 components:
            - Hostname of server housing the object
            - Path to the object on the server
            - `http://www.someSchool.edu/someDepartment/picture.gif`
        - Web browsers implement the client-side of HTTP, so browsers = clients
        - Web servers house web objects, each addressable by a URL (eg: Apache)
- When a user requests a webpage, browser sends HTTP request messages for the requested objects to the server, which sends back HTTP response messages containing the requested objects
- HTTP uses TCP as its underlying protocol
    - HTTP client initiates TCP connection with server
    - Once connection established, TCP messages can be sent from client socket to server socket
    - Client sends and receives HTTP messages through its socket
    - As soon as the client's HTTP request message passes through its socket, the message is 'in the hands of' TCP to be transported to the server socket
    - TCP gives HTTP a guarantee that its messages will travel/arrive safely
- HTTP is a **stateless protocol**
    - This means that an HTTP server stores no information about clients that request objects
    - ie: if a client requests the same object twice in 1 second, the server will respond to both requests as it 'forgets' who the client is after serving the first request

Non-persistent and Persistent connections

- **Persistent connection:** if client A sends a series of requests, each request is managed on the same TCP connection
- **Non-persistent connection:** if client A sends a series of requests, each request is managed on a separate TCP connection
- HTTP uses persistent by default, but can be configured to use non-persistent
- **HTTP with non-persistent connection:**
    - Suppose a page consists of a base HTML file and 10 images

`http://www.someSchool.edu/someDepartment/home.index`

Here is what happens:

1. The HTTP client process initiates a TCP connection to the server `www.someSchool.edu` on port number 80, which is the default port number for HTTP. Associated with the TCP connection, there will be a socket at the client and a socket at the server.
2. The HTTP client sends an HTTP request message to the server via its socket. The request message includes the path name `/someDepartment/home.index`. (We will discuss HTTP messages in some detail below.)
3. The HTTP server process receives the request message via its socket, retrieves the object `/someDepartment/home.index` from its storage (RAM or disk), encapsulates the object in an HTTP response message, and sends the response message to the client via its socket.
4. The HTTP server process tells TCP to close the TCP connection. (But TCP doesn't actually terminate the connection until it knows for sure that the client has received the response message intact.)
5. The HTTP client receives the response message. The TCP connection terminates. The message indicates that the encapsulated object is an HTML file. The client extracts the file from the response message, examines the HTML file, and finds references to the 10 JPEG objects.
6. The first four steps are then repeated for each of the referenced JPEG objects.

    - Since each TCP connection processes 1 request message and 1 response message (1 request/response msg pair), the above non-persistent example required 11 TCP connections (base HTML file + 10 images)
    - **TCP connections can be run in parallel** (most browsers open 5-10 in parallel)
- **Round-trip time (RTT) =** time taken for a small packet to travel from client, to server, and back to client
    - Takes into account delays
- TCP connection between client and server requires a '3 way handshake'

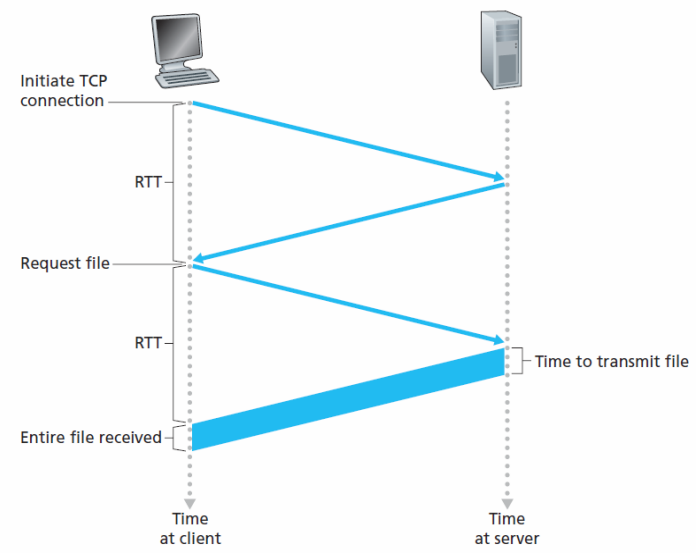


Figure 2.7 ♦ Back-of-the-envelope calculation for the time needed to request and receive an HTML file

- So, total response time to request and receive an HTML file = **2 RTTs + transmission time at server of HTML file**
  - One RTT to establish TCP connection, another RTT to request/receive object
- Cons of non-persistent connection:
  - Requires brand new TCP connection for each requested object
  - Object delivery is delayed by 2 RTTs

HTTP with Persistent Connections:

- Server leaves TCP connection open after sending a response
  - Subsequent request/response pairs can be sent of the same connection
- Multiple webpages residing on single server can be requested/sent over a single persistent TCP connection
- HTTP server closes persistent connection after lack of activity for some amount of time

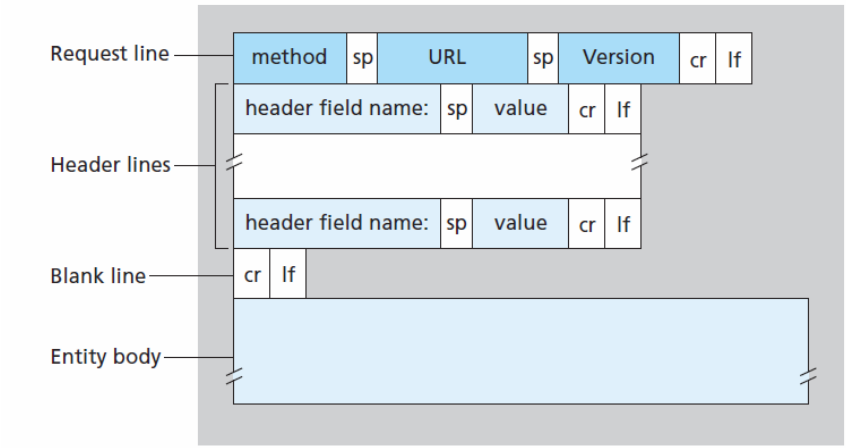
HTTP Message Format:

- Types of HTTP messages: Request, Response
- **HTTP Request:**

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu

Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

  - First line = **request line**
    - <u>Has 3 fields:</u>
      - **Method field** (GET, POST, HEAD, PUT)
        - GET requests an object
      - **URL field**
        - Requested object identified here

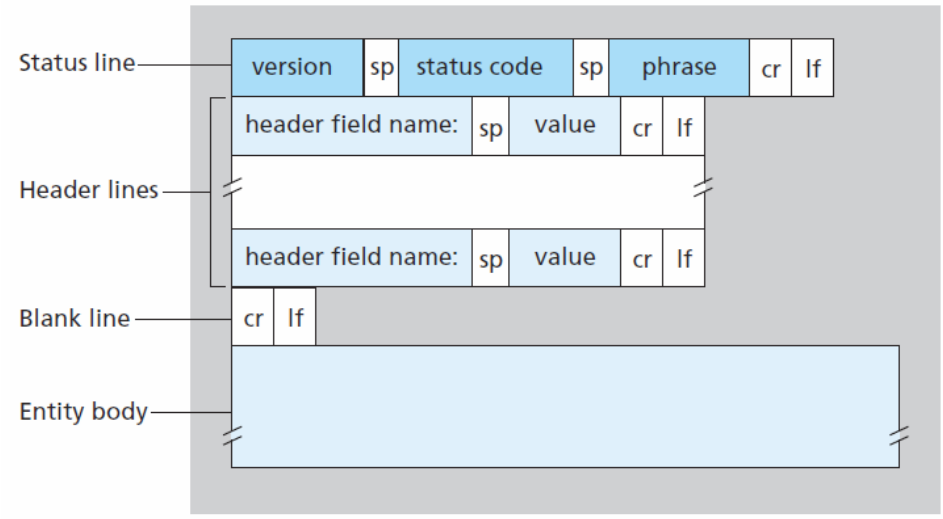
            - **HTTP version field**
                - self-explanatory
    - Subsequent lines = **header lines**
        - **Host** field specifies host/server on which object resides
            - Web proxy caches require this, even though TCP connection already specifies the host/server
        - **Connection** field:
            - 'Connection: close' means *use non-persistent connection*
            - 'Connection: open' means *use persistent connection* (ie: keep the connection open after the object is sent)
        - **User-agent:** specifies user-agent browser type making request (lets server know if, eg, a desktop or phone is making request, can serve appropriate content)
        - **Accept-language:** whether the English, French, etc. version of the object (ie webpage) is sent

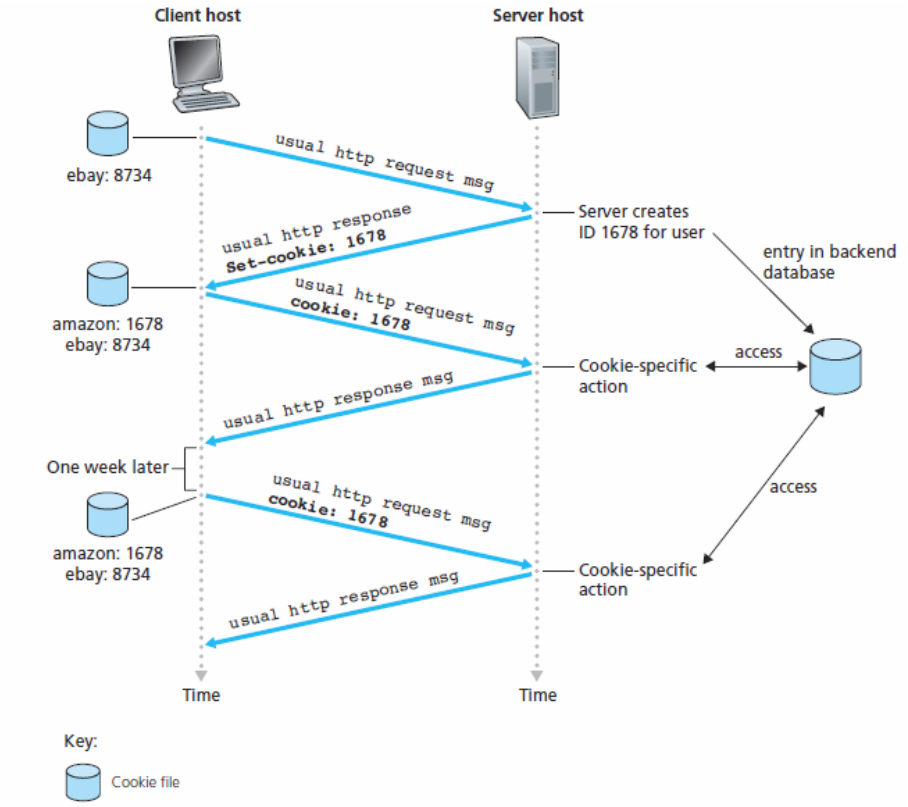
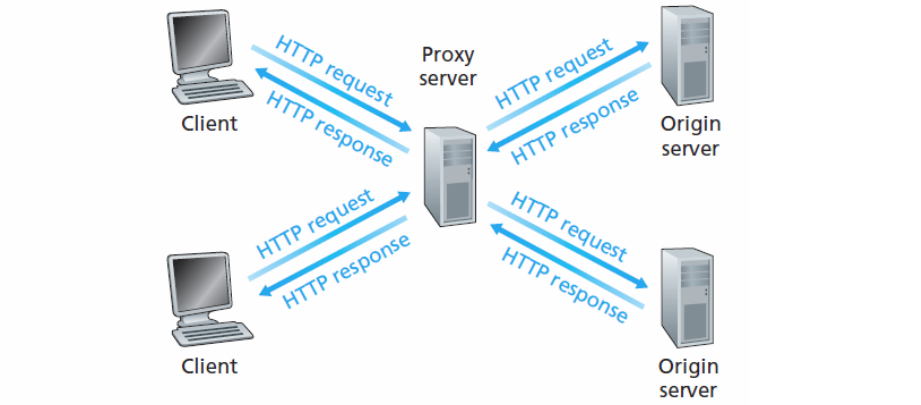
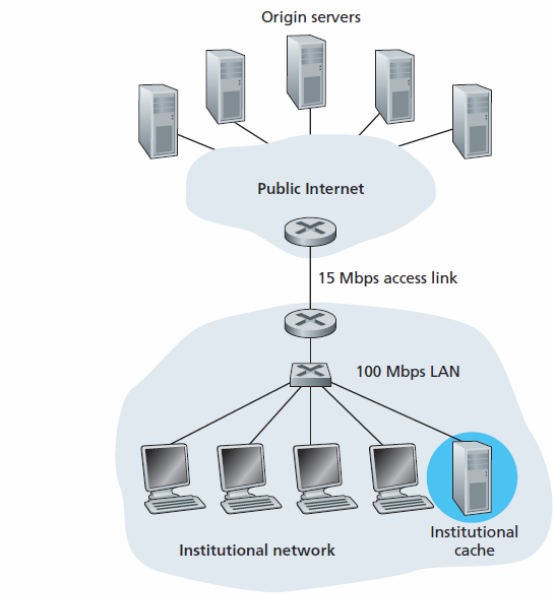
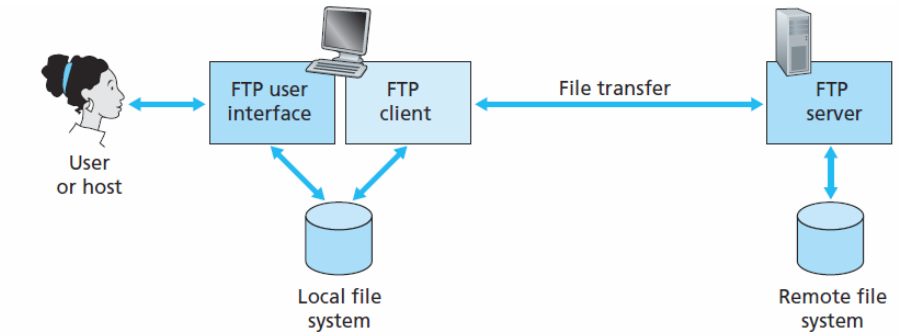


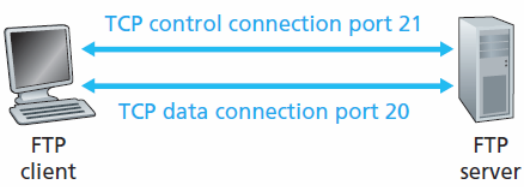
- **Figure 2.8** ♦ General format of an HTTP request message
- **Entity body** is often used with the **POST** method (but empty when using GET)
    - **POST** allows form data from webpage (ie: username entered in textbox) to be sent to server
    - If user types "how to dance" into Google search field, an HTTP POST message with "how to dance" in the Entity Body is sent to the server
    - **But,** form data can also be sent via GET (and included in the URL)
        - example, if a form uses the `GET` method, has two fields, and the inputs to the two fields are `monkeys` and `bananas`, then the URL will have the structure `www.somesite.com/animalsearch?monkeys&bananas`. In your day-to-
        - But more dangerous since data exposed in URL?

- **HEAD method:** Similar to GET method → except doesn't actually send the requested object
    - Used for debugging
- **PUT method:** used to upload objects to specific paths/directories on a webserver
- **DELETE method:** delete an object on a webserver

HTTP Response Message

- ```
  HTTP/1.1 200 OK
  Connection: close
  Date: Tue, 09 Aug 2011 15:44:04 GMT
  Server: Apache/2.2.3 (CentOS)
  Last-Modified: Tue, 09 Aug 2011 15:11:03 GMT
  Content-Length: 6821
  Content-Type: text/html

  (data data data data data ...)
  ```
- 3 sections: **status line,** 6 **header lines, entity body**
- **Entity body:** 'meat of the msg'
  - Contains requested object itself ('data data data data . . .)
- **Status line:**
  - Protocol version field
  - Status code
  - Corresponding status message
  - In above example, status line indicates that server is using HTTP/1.1 and that everything is 'OK' (the server found and is sending the object)
- **Header lines:**
  - 'Connection: close' means the TCP connection will be closed after the requested object has been sent
  - 'The Data:' → time and date when server retrieves object from its filesystem and inserts it into the response message to the client (NOT when object was created/last modified)
  - 'The Server' → webserver hosting the file (analogous to 'user-agent' field)
  - 'Last-Modified' → time/date when object was created / last modified
  - 'Content-type' → indicates type of object → ie HTML text
  - 'Content-length' → number of bytes in/comprising requested object
    - Object type is officially indicated by this header, and not the file extension of the object
- 

HTTP Status Codes:
  ```

- `200 OK:` Request succeeded and the information is returned in the response.
- `301 Moved Permanently:` Requested object has been permanently moved; the new URL is specified in `Location:` header of the response message. The client software will automatically retrieve the new URL.
- `400 Bad Request:` This is a generic error code indicating that the request could not be understood by the server.
- `404 Not Found:` The requested document does not exist on this server.
- `505 HTTP Version Not Supported:` The requested HTTP protocol version is not supported by the server.

Cookies:

- Remember that HTTP is stateless (doesn't remember users)
  - Simplifies design, allows for higher performance
  - But sometimes a website needs to identify users
    - Restrict access
    - Serve content based on user identity
    - Etc
  - Cookies allows sites to keep track of users
- **Cookie technology has 4 components:**
  - Header line in HTTP response msg
  - Header line in HTTP request msg
  - Cookie file kept on user's end system (and managed by browser)
  - Back-end database at website

How do cookies work?

user's browser, and (3) a back-end database at the Web site. Using Figure 2.10, let's walk through an example of how cookies work. Suppose Susan, who always accesses the Web using Internet Explorer from her home PC, contacts Amazon.com for the first time. Let us suppose that in the past she has already visited the eBay site. When the request comes into the Amazon Web server, the server creates a unique identification number and creates an entry in its back-end database that is indexed by the identification number. The Amazon Web server then responds to Susan's browser, including in the HTTP response a `Set-cookie:` header, which contains the identification number. For example, the header line might be:

```
Set-cookie: 1678
```

When Susan's browser receives the HTTP response message, it sees the `Set-cookie:` header. The browser then appends a line to the special cookie file that it manages. This line includes the hostname of the server and the identification number in the `Set-cookie:` header. Note that the cookie file already has an entry for eBay, since Susan has visited that site in the past. As Susan continues to browse the Amazon site, each time she requests a Web page, her browser consults her cookie file, extracts her identification number for this site, and puts a cookie header line that includes the identification number in the HTTP request. Specifically, each of her HTTP requests to the Amazon server includes the header line:

- `Cookie: 1678`

In this manner, the Amazon server is able to track Susan's activity at the Amazon site. Although the Amazon Web site does not necessarily know Susan's name, it knows exactly which pages user 1678 visited, in which order, and at what times! Amazon uses cookies to provide its shopping cart service—Amazon can maintain a list of all of Susan's intended purchases, so that she can pay for them collectively at the end of the session.

If Susan returns to Amazon's site, say, one week later, her browser will continue to put the header line `Cookie: 1678` in the request messages. Amazon also recommends products to Susan based on Web pages she has visited at Amazon in the past. If Susan also registers herself with Amazon—providing full name, e-mail address, postal address, and credit card information—Amazon can then include this information in its database, thereby associating Susan's name with her identification number (and all of the pages she has visited at the site in the past!). This is how Amazon and other e-commerce sites provide "one-click shopping"—when Susan chooses to purchase an item during a subsequent visit, she doesn't need to re-enter her name, credit card number, or address.

From this discussion we see that cookies can be used to identify a user. The first time a user visits a site, the user can provide a user identification (possibly his or her name). During the subsequent sessions, the browser passes a cookie header to the server, thereby identifying the user to the server. Cookies can thus be used to create a user session layer on top of stateless HTTP. For example, when a user logs in to a Web-based e-mail application (such as Hotmail), the browser sends cookie information to the server, permitting the server to identify the user throughout the user's session with the application.

- 

- **Figure 2.10** ♦ Keeping user state with cookies

Web Caching (Proxying):

- **Web cache** AKA **proxy server =** network entity that satisfies HTTP requests on behalf of an origin web server
- Web cache has its own disk storage
    - Keeps copies of recently requested objects in this storage
- Can configure browser such that all HTTP requests are first directed to web cache / proxy before the actual locations of the requested files

- **Once a browser is configured to direct traffic through a webcache/proxy, this happens:**
  - 1. The browser establishes a TCP connection to the Web cache and sends an HTTP request for the object to the Web cache.
  - 2. The Web cache checks to see if it has a copy of the object stored locally. If it does, the Web cache returns the object within an HTTP response message to the client browser.
  - 3. If the Web cache does not have the object, the Web cache opens a TCP connection to the origin server, that is, to `www.someschool.edu`. The Web cache then sends an HTTP request for the object into the cache-to-server TCP connection. After receiving this request, the origin server sends the object within an HTTP response to the Web cache.
  - 4. When the Web cache receives the object, it stores a copy in its local storage and sends a copy, within an HTTP response message, to the client browser (over the existing TCP connection between the client browser and the Web cache).
  - 

  - **Figure 2.11** ♦ Clients requesting objects through a Web cache
- Note: a cache is both a server and client at the same time
  - Middleman between user and server hosting requested file
  - = server to the user/browser, client to the actual server hosting the file
- Pros of webcache/proxy:
  - Reduce response time for client requests to servers
  - Reduce traffic on an institution's access link to internet (since a lot of requested content can be retrieved from the local webcache instead of downloading from remote server
    - = reduce bandwidth costs



■ **Figure 2.13** ♦ Adding a cache to the institutional network

- **Content Distribution Network (CDN)** companies install geographically distributed caches throughout the internet, localising much of the internet's traffic (ie: allowing users to download nearby cached content instead of expensive downloading of remote content)
    - **Shared CDNs**: Akamai, Limelight
    - **Dedicated CDNs**: Google, Microsoft

The Conditional GET:

- So caching can reduce response times, but a problem with caching is the possibility of **stale/old copies of objects residing in the cache**
    - ie: most up-to-date version of the object hasn't yet been put in the webcache
- **Conditional GET** allows a cache to verify that its objects are up to date
- A Conditional GET message:
    - Uses the GET method
    - Includes an '**If-Modified-Since:'** header line
- **How it works:**
    - The first time a user requests object X from server Y via cache/proxy Z, cache Z requests and stores object X from server Y, then…. (below)

The cache forwards the object to the requesting browser but also caches the object locally. Importantly, the cache also stores the last-modified date along with the object. Third, one week later, another browser requests the same object via the cache, and the object is still in the cache. Since this object may have been modified at the Web server in the past week, the cache performs an up-to-date check by issuing a conditional GET. Specifically, the cache sends:

```
GET /fruit/kiwi.gif HTTP/1.1
Host: www.exotiquecuisine.com
If-modified-since: Wed, 7 Sep 2011 09:23:24
```

Note that the value of the `If-modified-since:` header line is exactly equal to the value of the `Last-Modified:` header line that was sent by the server one week ago. This conditional GET is telling the server to send the object only if the object has been modified since the specified date. Suppose the object has not been modified since 7 Sep 2011 09:23:24. Then, fourth, the Web server sends a response message to the cache:

```
HTTP/1.1 304 Not Modified
Date: Sat, 15 Oct 2011 15:39:29
Server: Apache/1.3.0 (Unix)
```

- ```
  (empty entity body)
  ```

File Transfer Protocol (FTP)

- Allows client to transfer files to or from a remote host/server
- = application-layer protocol, like HTTP

1. User provides hostname of remote server/host to FTP useragent
2. TCP connection established between FTP process in client host and FTP process in remote server
3. User enters username + password → sent over TCP connection
4. Once authorised, user can copy files to remote server (or vice-versa)

- 

Figure 2.14 ♦ FTP moves files between local and remote file systems
  ```



- **Figure 2.15** ♦ Control and data connections

HTTP vs FTP:

FTP	HTTP
Uses 2 parallel TCP connections (**control connection, data connection**)	Uses 1 TCP connection for both control/commands and for data (actual sending of files)

- (FTP) **control connection:**
  - For sending control info (username, password, commands to change directory, commands to upload/download files)
- (FTP) **data connection:**
  - For actual sending of data/files
- Because FTP uses separate control connection, **FTP is said to send its control information out-of-band**
  - In contrast, since HTTP uses 1 TCP connection for control & data, we say that **HTTP is said to send its control information in-band**
  - SMTP also sends control information in-band
- When a user starts an FTP session with a remote host, the client side of FTP (user) first initiates a control TCP connection with the server side (remote host) on server port number 21. The client side of FTP sends the user identification and password over this control connection. The client side of FTP also sends, over the control connection, commands to change the remote directory. When the server side receives a command for a file transfer over the control connection (either to, or from, the remote host), the server side initiates a TCP data connection to the client side. FTP sends exactly one file over the data connection and then closes the data connection. If, during the same session, the user wants to transfer another file, FTP opens another data connection. Thus, with FTP, the control connection remains open throughout the duration of the user session, but a new data connection is created for each file transferred within a session (that is, the data connections are non-persistent).

  Throughout a session, the FTP server must maintain **state** about the user. In particular, the server must associate the control connection with a specific user account, and the server must keep track of the user's current directory as the user wanders about the remote directory tree. Keeping track of this state information for each ongoing user session significantly constrains the total number of sessions that FTP can maintain simultaneously. Recall that HTTP, on the other hand, is stateless—it does not have to keep track of any user state.

FTP commands and replies

- Sent across control connection (in 7-bit ASCII format)
    - Thus commands = readable to humans
    - Each command = 4 uppercase ASCII letters
- **Some FTP commands:**
    - `USER username:` Used to send the user identification to the server.
    - `PASS password:` Used to send the user password to the server.
    - `LIST:` Used to ask the server to send back a list of all the files in the current remote directory. The list of files is sent over a (new and non-persistent) data connection rather than the control TCP connection.
    - `RETR filename:` Used to retrieve (that is, get) a file from the current directory of the remote host. This command causes the remote host to initiate a data connection and to send the requested file over the data connection.
    - `STOR filename:` Used to store (that is, put) a file into the current directory of the remote host.
- **Some FTP replies:**
    - `331 Username OK, password required`
    - `125 Data connection already open; transfer starting`
    - `425 Can't open data connection`
    - `452 Error writing file`

Section 2.4 (Mail & SMTP)

- **This section is optional (not gonna be tested), so not writing notes on it**

DNS (Section 2.5)

- DNS is 'internet's directory service' → = internet's **Domain Name System (DNS)**
- Internet hosts identified in 2 ways:
    - **Hostname** (ie: www.google.com)
        - Human-friendly
        - Tells us a little about location of host (ie: ".fr" is French host) but not much else
        - Not very router-friendly, since hostnames can consist of variable-length alphanumeric characters that routers would have to decode
    - **IP address:** (ie: 216.58.223.46)
        - Each period separates a 0-255 length byte in decimal notation
        - Router-friendly
        - Structured hierarchically → as you read IP address from left to right = tells you more about location of corresponding host in the internet
- DNS translates hostnames <---> IP addresses

Services provided by DNS:

- Distributed database implemented in a hierarchy of **DNS servers**
- Application-layer protocol that allows hosts to query the distributed database

- DNS servers often = UNIX machines running **Berkeley Internet Name Domain (BIND)** software
- DNS protocol **runs over UDP,** not TCP
    a. Uses (socket) port 53
- DNS used by other application-layer protocols (like HTTP, SMTP, FTP) to translate hostnames to IP addresses
- **How DNS plays a role in a website request**
    a. HTTP client (webbrowser) requests google.com
    b. Browser extracts hostname ("google.com") from URL and passes to local DNS client
    c. Local DNS client sends URL to remote DNS server in a query
    d. Local DNS client receives corresponding IP address in a reply from server
    e. DNS client passes IP address to browser, which can now initiate TCP connection to google.com on port 80 (which is for HTTP requests)
- Requirement of local DNS client having to query IP address from remote DNS server = **delay**
    - BUT, desired IP address often cached in nearby DNS server
- **Additional services provided by DNS:**
    - **Host aliasing**
        - A ***canonical hostname*** like "relay1.west-coast.enter-prise.com" can have ***aliases*** like "enterprise.com" and "www.enterprise.com"
        - Alias hostnames = more mnemonic / human friendly than canonical hostnames
        - DNS can provide both canonical hostname and IP address for a given alias hostname
    - **Mail server aliasing**
        - Humans obviously want mnemonic email addresses
            - ie: "bob@company.com" (alias) instead of "bob@relay3.amakai.company.com" (canonical)
            - DNS again allows us humans to use the alias hostname, returning the canonical hostname and/or IP address if invoked to do so
            - ***MX Record:*** allows a company's mail server and webserver to have same hostname
                - ie: webserver is "company.com" and email address is "bob@company.com"
    - **Load distribution**
        - Busy sites like cnn.com are replicated over multiple servers (relay1.cnn..., relay2.cnn…., etc.)
            - Each server = different end system with different IP address
        - DNS allows us to associate a **set of IP addresses** (stored in DNS database) with one canonical hostname
        - When clients request a hostname that is mapped to a set of IP addresses, they typically request the first address in the set
            - So DNS rotates the order of IP sets to distribute traffic among the replicated servers

        - Similar rotation used for load-distribution of mail servers too

How DNS works:

(Namely, how hostname-to-IP translation works)

1. App needs to translate hostname to IP
2. App invokes client-side of DNS (local) with hostname
    a. “gethostbyname()” command on many UNIX machines
3. DNS client-side sends query to DNS network
    a. All DNS queries/replies sent in **datagrams**
4. *delay* (milliseconds to seconds)
5. DNS client-side receives reply from DNS server containing desired hostname-IP mapping
6. Mapping (which contains the desired IP address) is passed to app that sent the query

- **Problems with centralised DNS server design:**
    - (ie: one simple server containing all mappings needed by internet)
    - 1. **If DNS server crashes, internet crashes (**single point of failure)
    - 2. **Traffic volume** → one server handling all queries from 100s of millions of hosts
    - 3. **Distant centralised database:** huge delays when DNS server on one side of planet, queries coming from other side → can’t use caching to help since only one centralised server
    - 4. **Maintenance**: would have to be updated unfeasibly frequently

- **Therefore, DNS is distributed by design:**
    - DNS uses large number of servers organises hierarchically and distributed all around world
        - No single DNS server has all IP<-->Hostname mappings
- **Three classes of DNS servers (and how they work together):**
    - **Root DNS servers**
        - Contacted first by DNS client-side querying for a hostname’s IP
        - Sends client a list of TLD servers
    - **Top-level Domain (TLD) DNS servers**
        - Client now has list of TLD servers: one for *“.com”*, one for *“.org”*, one for *“.edu”*, etc.
        - Queries the appropriate TLD server, which sends back the IP of the authoritative server for the desired hostname
    - **Authoritative DNS servers**
        - Client queries the authoritative server, which sends back the IP address for the hostname
    - (process illustrated below)

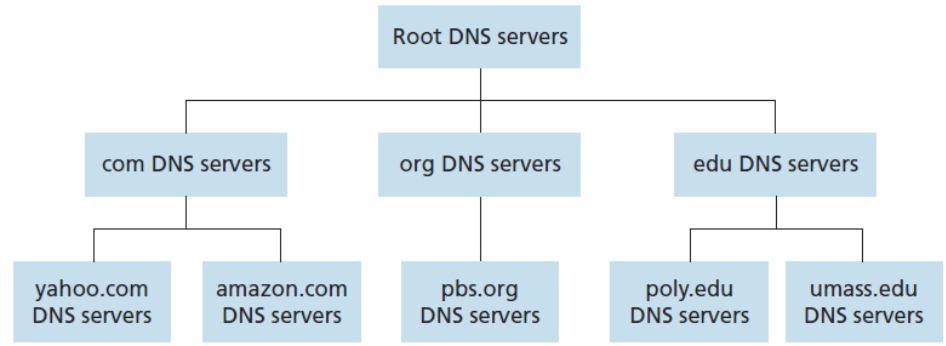


Figure 2.19 ♦ Portion of the hierarchy of DNS servers

Root DNS servers:

- 13 in the internet
- Each of the 13 'servers' is a network of replicated servers → 247 altogether

TLD servers:

- Responsible for top-level domains like com, org, net, edu, gov, etc.
- ".com" maintained by **Verisign Global Registry Servers**
- ".edu" maintained by **Educause**

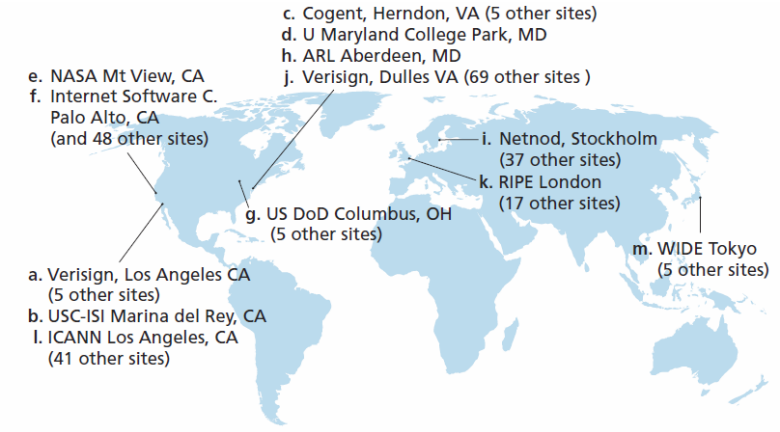


Figure 2.20 ♦ DNS root servers in 2012 (name, organization, location)

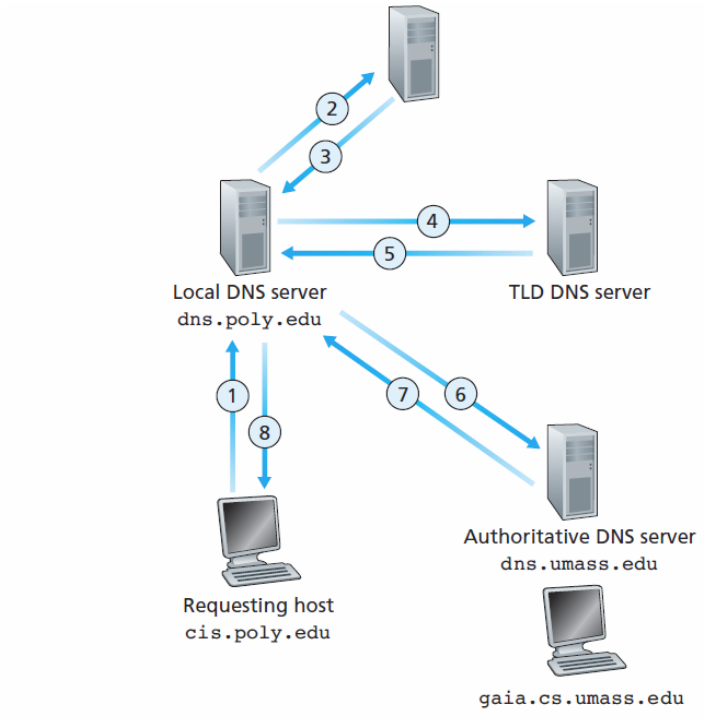
Authoritative DNS servers:

- Every organisation with public hosts (webservers, mailservers) must provide public DNS records mapping their hosts to IP addresses
- This mapping is stored in an organisation's authoritative DNS server
- Can use own authoritative DNS server, or pay to use a 3rd party one

Local DNS Server:

- Not strictly part of the 3-tier hierarchy discussed above (root, TLD, authoritative)
- Every ISP (university, company, residential ISP, etc.) has a **Local DNS Server (AKA default name server)**
- When a host connects to an ISP, ISP provides IP address(es) of its local DNS server(s)
    - (through DHCP...discussed in later chapters)
- Local DNS Server = close to the host
- When a host makes a DNS query, query is sent to the Local DNS Server (which acts as a proxy) which forwards it to the 3-tier DNS server hierarchy
- For one IP<-->hostname translation, takes 8 DNS queries (4 requests, 4 responses)
    - Browser → Local
    - Local → Root
    - Root → Local
    - Local → TLD

- TLD → Local
- Local → Authoritative
- Authoritative → Local
- Local → Browser
    - DNS caching helps out with this 8-step fuckup



- **Figure 2.21** ♦ Interaction of the various DNS servers

- Sometimes the TLD server doesn't know the Authoritative server for a hostname
    - In such cases, TLD server has to query an intermediate server which does know
    - Increases number of required DNS queries
- **Recursive queries:**
    - When a client sends queries to an intermediate DNS server which sends on the queries to subsequent servers
    - ie: when a 'middleman' DNS server is used
- **Iterative queries:**
    - When all DNS queries/responses are direct

*** In fig 2.21 (bottom of previous pg), only steps 1 and 8 were recursive**
*** In fig 2.22 (below), all queries are recursive**

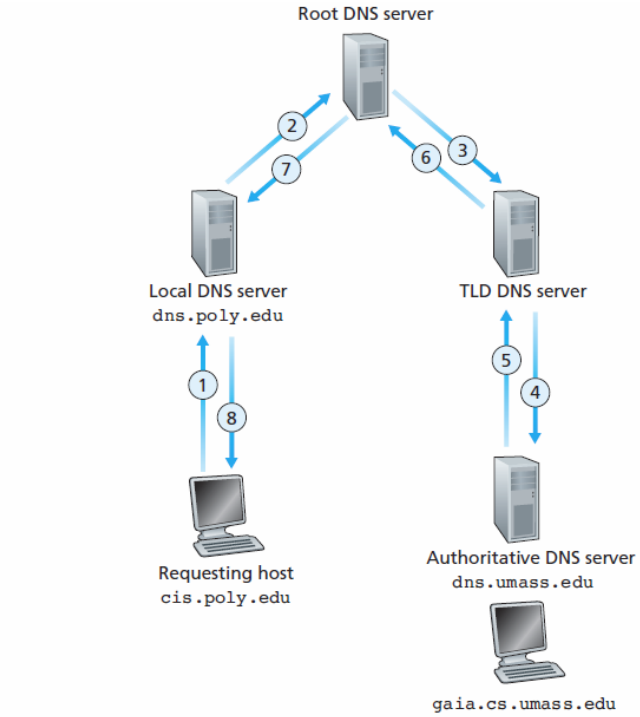


Figure 2.22 ♦ Recursive queries in DNS

- *In practice, the queries typically follow the pattern in Figure 2.21: The query from the requesting host to the local DNS server is recursive, and the remaining queries are iterative.*

DNS Caching:

- In a query chain, when a DNS server receives an IP<->Host mapping, it can store the mapping in a local cache
- Subsequent queries for one of the hostnames in this mapping can be catered for with minimal delay as the requested mapping will be 'nearby' (stored locally in memory).
    - ie: local DNS server can cache mappings between hostnames and IP addresses from authoritative DNS servers, so cached hostnames can be pulled from the cache immediately after the requesting client's first query without having to go through the hierarchy
    - Can also cache IPs from TLD servers, so even if the hierarchy has to be followed, it can skip the 'root server query' step
- Caches are wiped after certain amounts of time

2.5.3 DNS Records and Messages

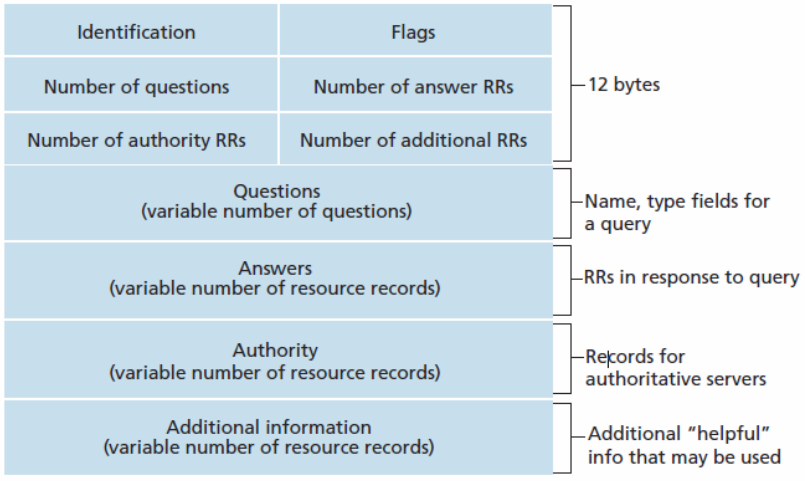
- **Resource Records:**
    - Contained in DNS messages
    - 4-tuple containing the fields (Name, Value, Type, TTL)
        - TTL = time to live → specifies when the resource record should be wiped from the cache
    - Meaning of *Name* and *Value* depend on *Type*

- If `Type=A`, then `Name` is a hostname and `Value` is the IP address for the hostname. Thus, a Type A record provides the standard hostname-to-IP address mapping. As an example, `(relay1.bar.foo.com, 145.37.93.126, A)` is a Type A record.
- If `Type=NS`, then `Name` is a domain (such as `foo.com`) and `Value` is the hostname of an authoritative DNS server that knows how to obtain the IP addresses for hosts in the domain. This record is used to route DNS queries further along in the query chain. As an example, `(foo.com, dns.foo.com, NS)` is a Type NS record.
- If `Type=CNAME`, then `Value` is a canonical hostname for the alias hostname `Name`. This record can provide querying hosts the canonical name for a hostname. As an example, `(foo.com, relay1.bar.foo.com, CNAME)` is a CNAME record.
- If `Type=MX`, then `Value` is the canonical name of a mail server that has an alias hostname `Name`. As an example, `(foo.com, mail.bar.foo.com, MX)` is an MX record. MX records allow the hostnames of mail servers to have simple aliases. Note that by using the MX record, a company can have the same aliased name for its mail server and for one of its other servers (such as its Web server). To obtain the canonical name for the mail server, a DNS client would query for an MX record; to obtain the canonical name for the other server, the DNS client would query for the CNAME record.

- If a DNS server is authoritative for hostname 'some_hostname', it'll contain a Type-A record for that hostname where *Name* = some_hostname and *Value* = IP_of_some_hostname
- If a DNS server is not authoritative for a hostname 'some_hostname':
  - it'll contain a Type-NS record where ***Name*** = 'some_hostname' and ***Value*** = some_authoritative_hostname (which knows IP_of_some_hostname)
  - And it'll contain a Type-A record where ***Name*** = some_authoritative_hostname and *Value* = IP_of_some_authoritative_hostname

DNS messages:

- Only 2 types = *query* & *reply*
  - Have same format



- **Figure 2.23** ♦ DNS message format
- **Header section:** first 12 bytes
  - *Identification*: 16-bit number identifying query

        - Allows responses to be matched with replies
    - *Flags:*
        - 1-bit query flag indicates whether message is query (0) or reply (1)
        - 1-bit authoritative flag is set in a reply msg when DNS server is an authoritative server for a queried hostname
        - 1-bit recursion-desired flag is set when a recursive query ('middleman') is required to fetch a record
        - 1-bit recursion-available field set in a reply when DNS server supports recursion
- **Number-of fields:**
    - Indicate number of occurrences of last 4 data sections (Questions, Answers, Authority, Additional information)
- **Questions:** info about query being made
    - Field containing queried hostname
    - Field indicating type of query (ie: map alias to canonical or map alias to IP?)
- **Answers:** (sent in a reply)
    - Contains Resource Records for queried hostname (ie: Type-A, Type-NS, etc.)
- **Authority:** records of other authoritative servers
- **Additional:** additional helpful records (ie: in MX query reply, contains IP address of canonical hostname even though just the alias <-->canonical mapping was required

(**nslookup:** program to send DNS queries to DNS servers from any PC)

Inserting records into DNS database

- **Registrar:** verifies uniqueness of your domain name and enters the domain name into the DNS database
    - ie: GoDaddy,
    - ICANN (Internet Corporation for Assigned Names and Numbers) accredits/authorises registrars
- More recently, an UPDATE option has been added to the DNS protocol to allow data to be dynamically added or deleted from the database via DNS messages.

Registering domain names with registrars (yellow highlighted text below):

When you register the domain name networkutopia.com with some registrar, you also need to provide the registrar with the names and IP addresses of your primary and secondary authoritative DNS servers. Suppose the names and IP addresses are dns1.networkutopia.com, dns2.networkutopia.com, 212.212.212.1, and 212.212.212.2. For each of these two authoritative DNS servers, the registrar would then make sure that a Type NS and a Type A record are entered into the TLD com servers. Specifically, for the primary authoritative server for networkutopia.com, the registrar would insert the following two resource records into the DNS system:

```
(networkutopia.com, dns1.networkutopia.com, NS)

(dns1.networkutopia.com, 212.212.212.1, A)
```

Example summarising DNS section (blue highlighted text below):

what we have learned about DNS. Suppose Alice in Australia wants to view the Web page www.networkutopia.com. As discussed earlier, her host will first send a DNS query to her local DNS server. The local DNS server will then contact a TLD com server. (The local DNS server will also have to contact a root DNS server if the address of a TLD com server is not cached.) This TLD server contains the Type NS and Type A resource records listed above, because the registrar had these resource records inserted into all of the TLD com servers. The TLD com server sends a reply to Alice's local DNS server, with the reply containing the two resource records. The local DNS server then sends a DNS query to 212.212.212.1, asking for the Type A record corresponding to www.networkutopia.com. This record provides the IP address of the desired Web server, say, 212.212.71.4, which the local DNS server passes back to Alice's host. Alice's browser can now initiate a TCP connection to the host 212.212.71.4 and send an HTTP request over the connection. Whew! There's a lot more going on than what meets the eye when one surfs the Web!

- **READ PG 143 FOR 'DNS SECURITY' SUBSECTION**

Peer-to-Peer (P2P) Applications

- P2P architecture → minimal/no reliance on *always-on* servers
    - Instead, intermittently (irregularly) connected hosts/peers (desktops, laptops, etc.) communicate directly with each other
- eg: BitTorrent, Skype
- In P2P file distribution, each peer can redistribute any portion of the file it has received to any other peers
- **Distribution time:** time taken to get copy of file to all peers
    - With client-server architecture, distribution time increases linearly with number of peers

    - With P2P server, distribution time is always less than client-server architecture (and less than 1 hour for any # of peers N)
- P2P applications can be self-scaling → since peers are both consumers and redistributors of bits of data

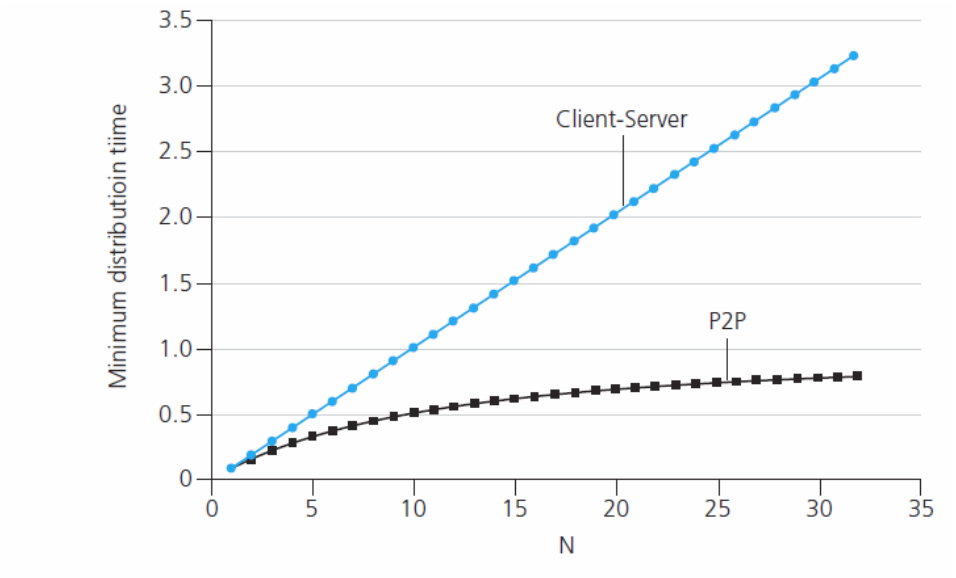


Figure 2.25 ♦ Distribution time for P2P and client-server architectures

Socket Programming: Creating network applications

- Typical network app: consists of *client program* and *network program*
    - Reside on different operating systems
    - *Client process* and *server process* communicate with each other through sockets
    - Code must be written for client and server programs
- **2 types of network app:**
    - **Open network apps**
        - Underlying operation/rules specified in public protocol standard (ie: RFC)

        - Client and server program must conform to these rules
        - ie: FTP client → FTP 'rules' are defined in RFC
        - Usually one developer works on client side, another works on server side
            - ie: firefox developer works on client-side, Apache developer works on server-side → Firefox browser can communicate with an Apache webserver
            - ie: BitTorrent client communicating with Tracker
    - **Proprietary network apps**
        - 'rules'/operations are not publicly available (ie: in RFC)
        - One developer/team works on both client and server side → complete control
            - But other developers can' write programs that hook into / extend the app
- Must decide whether to use TCP (connection-oriented, safe) or UDP (connectionless, unsafe)

2.7.1 Socket Programming with UDP

- **This section = <u>optional</u> but go back to pg 157 and read through it some time**

2.7.2 Socket Programming with TCP

- Before client and server can communicate, must first *handshake* to establish TCP connection
    - One end of TCP connection attached to client socket (IP + port num), other end attached to server socket (IP + port num)
    - If one side wants to send data to other side, it just drops the data into its socket to be sent over the TCP connection to the other side
- Client must initiate contact with server
    - TCP server must be running as a process prior to this
    - Server has to be 'ready' → must have special socket that welcomes arbitrary 'initial contact' messages
        - "Knocking on the welcome door[/socket]"
- After initial contact, client process initiates TCP connection to server by creating a TCP socket which knows the IP & port num of the server and its socket
- After client has created this socket, it initiates a 3-way handshake and establishes TCP connection with server
    - TCP connection (takes place within transport-layer) = invisible to client & server processes
- During 3-way handshake, client process 'knocks on door/socket' of server process
    - When server 'hears knocking', it creates a new socket/door dedicated to that client process
- *Server's initial welcoming door* **= serverSocket**
    - Initial point of contact for all clients wanting to communicate with server → server only has one of these
- *Server's socket dedicated to particular client* **= connectionSocket**

- Server creates one of these for each client it's communicating with
- As portrayed below (fig 2.29), there is a direct 'pipe' connecting the client socket to the server socket

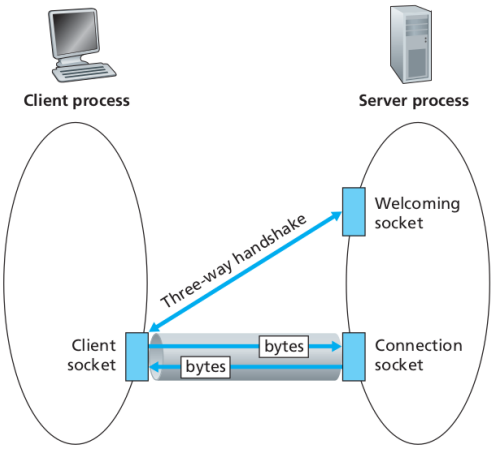


Figure 2.29 ♦ The `TCPServer` process has two sockets

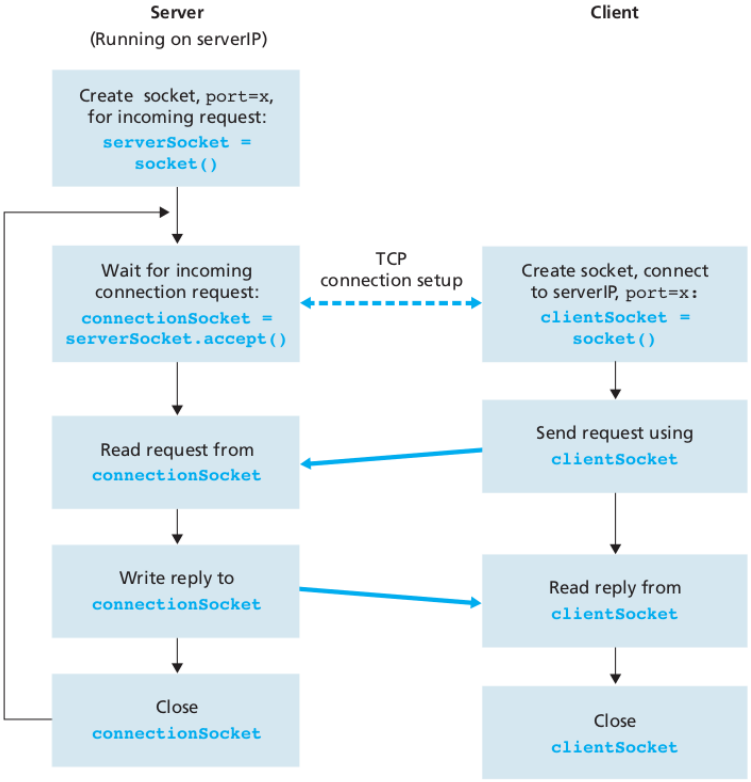


Figure 2.30 ♦ The client-server application using TCP

Application layer

BGP · DHCP · DNS · FTP · HTTP · IMAP · LDAP · MGCP · NNTP · NTP · POP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP ·