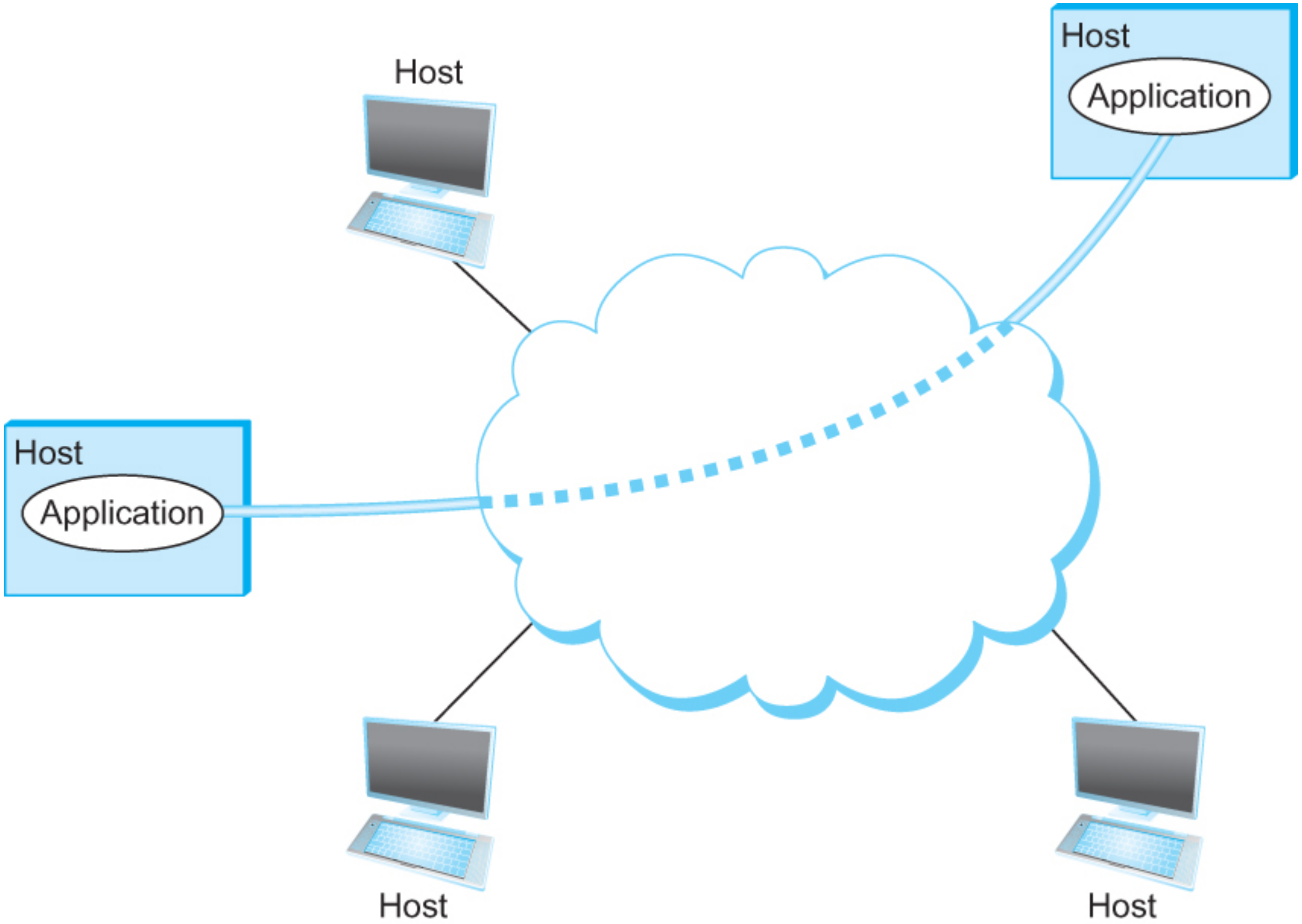
Review of Computer Networks

Yao Liu

How do applications talk to each other?



Layered protocols

- Networks are organized as a series of layers, in order to reduce complexity
  - Each layer builds upon the one below it and offers services to the one above it.
  - Between each layer is an interface

Physical and data link layers

- **Physical Layer:** Transmit and receive bits on physical media
  - analog and digital transmission
  - a definition of the 0 and 1 bits
  - bit rate (bandwidth)
- **Data Link Layer:** Provide error-free bit streams across physical media
  - Error detection/correction
  - reliability
  - flow control

Network/Internet layer

Controls the operations of the network

Provides host–to–host connections

- **Internetworking** of both homogeneous and heterogeneous networks
- **Routing:** determining the path from the source of a message to its destination
- **Congestion Control:** handling traffic jams

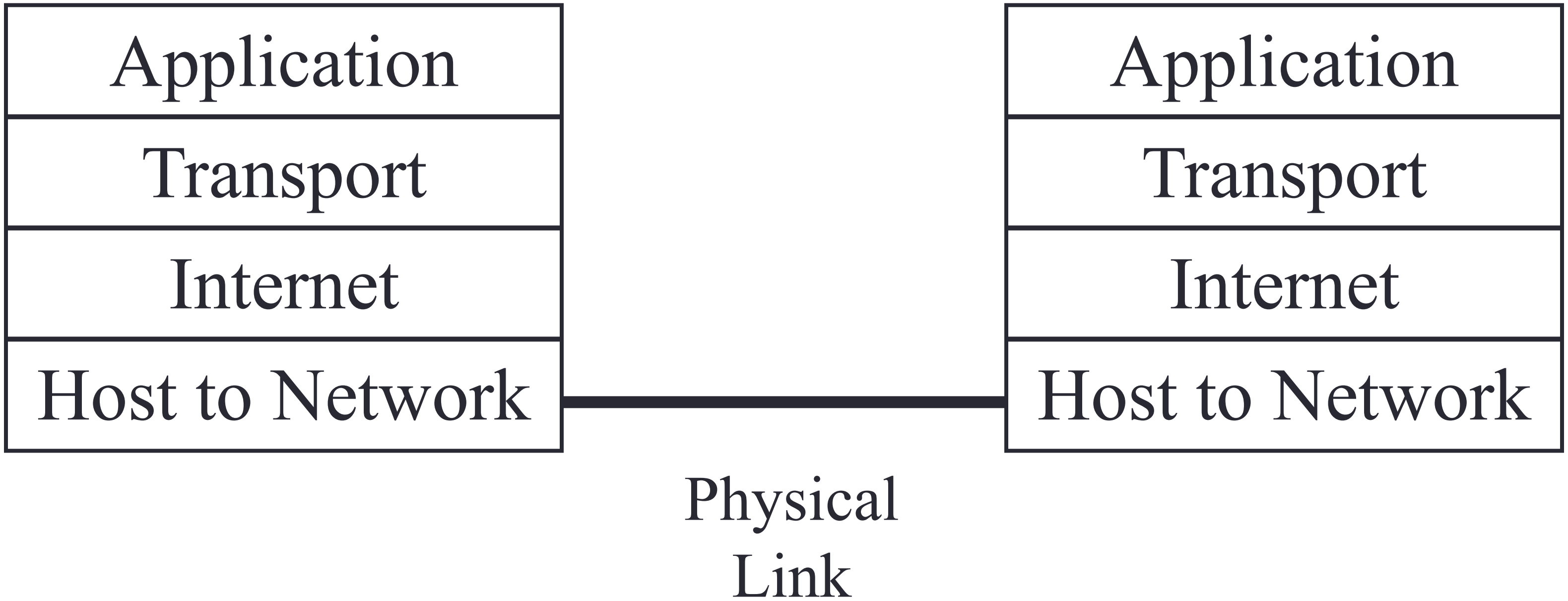
Transport layer

- Provides end–to–end connections
- **Packetization**: cut the messages into smaller chunks (packets)
- An ensuing issue is **ordering**: the receiving end must make sure that the user receives the packets in the right order
- End-to-end **flow control**

Application layer

- file transfer, email, WWW, multimedia, and so on

Internet Protocol Suite Reference Model



Network / Internet layer

- Provides logical communication between hosts

- Internet Protocol (IP)
  - Key tool used today to build scalable, heterogeneous internetworks
  - It runs on all the nodes in a collection of networks
  - Hosts are assigned **IP addresses** based on the networks they locate in

IPv4 addresses

- Fixed-length, 32-bit address
- Each network has its network ID
- Every interface in that network has an IP address comprising the network ID and a host ID
- *IP addresses identify interfaces*
  - There is no 1-to-1 correspondence between IP addresses and nodes (hosts or routers)

IPv4 address notation

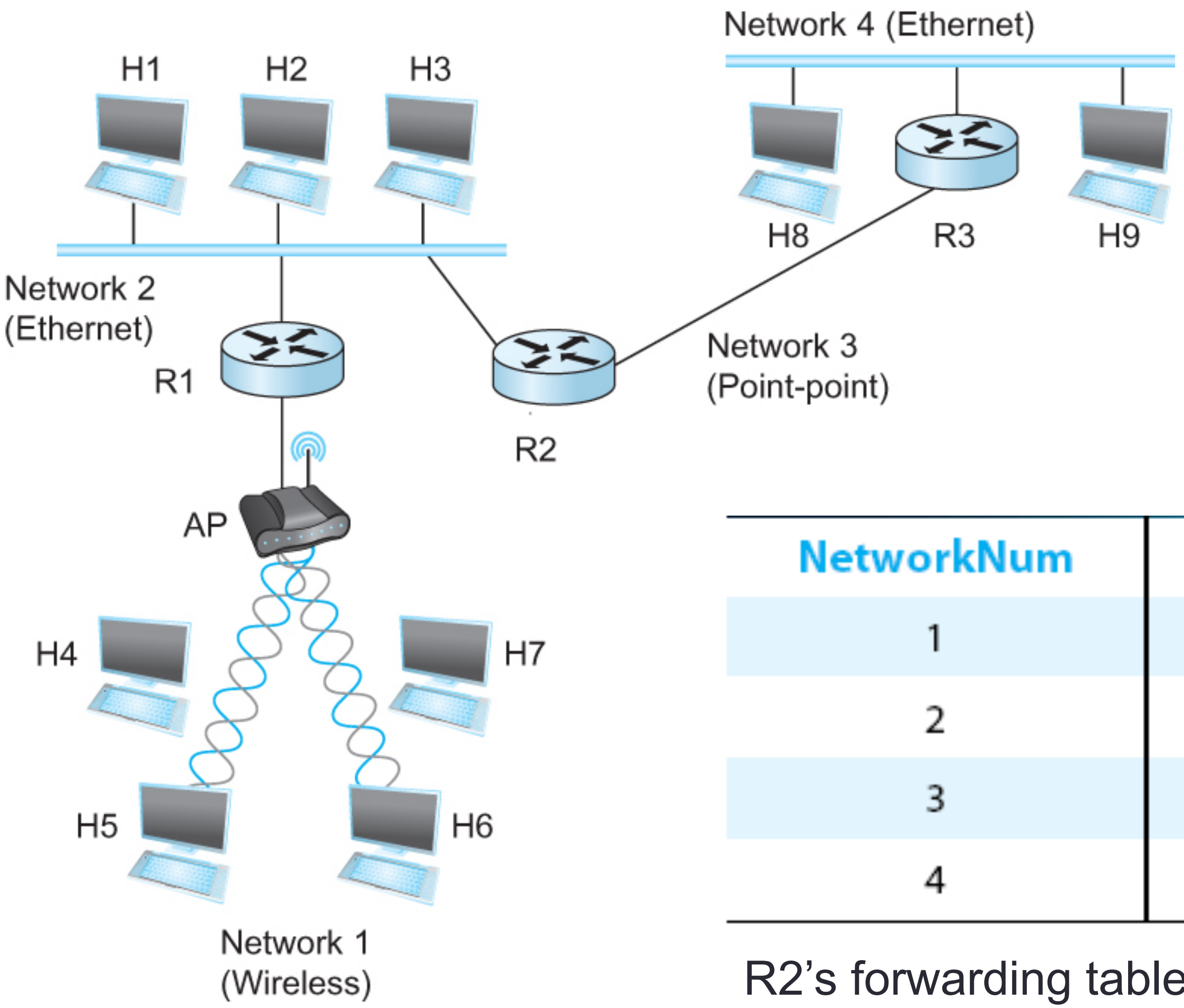
- A **dotted decimal** notation is used by human

- Binary: 11000000 01111111 11111101 00000001
  Decimal: 192. 127. 253. 1
- Examples:
  - cs.binghamton.edu = 128.226.118.12
  - remote02.cs.binghamton.edu = 128.226.114.202

Packet forwarding

- Each host has a default router
- Each router maintains a **forwarding table**
- Every packet contains destination’s address
- If directly connected to destination network, then forward to host
- If not directly connected to destination network, then forward to some router
- Forwarding table maps **network number** into next hop

Packet forwarding



NetworkNum	NextHop
1	R1
2	Interface 1
3	Interface 0
4	R3

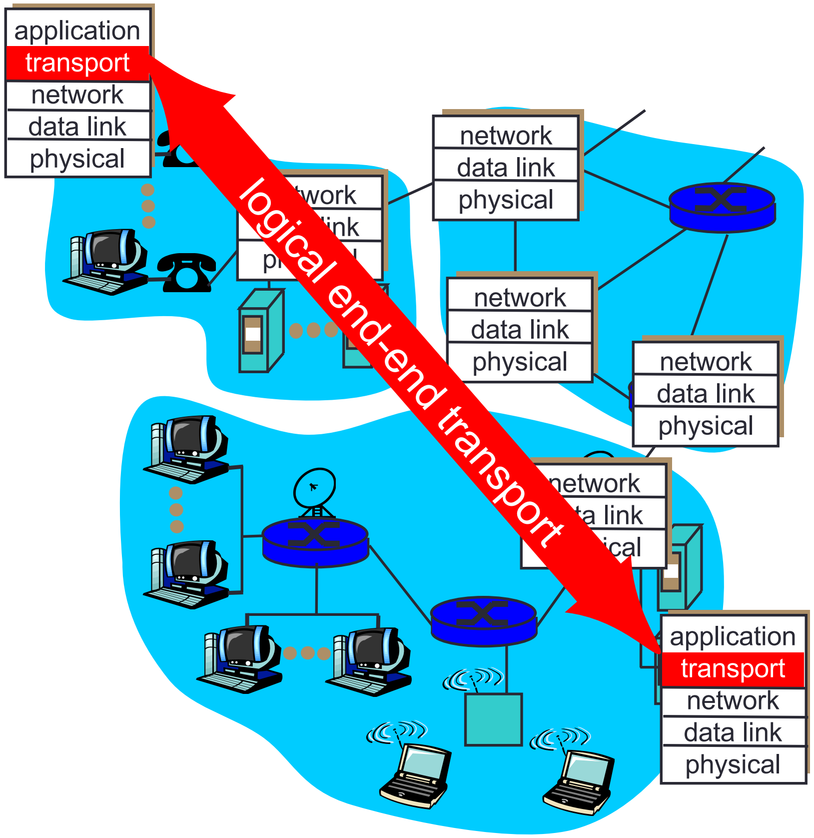
R2's forwarding table. Two are direct routes

IPv4 packet header

<table>
<tr><th>0</th><th>4</th><th>8</th><th>12</th><th>16</th><th>19</th><th>24 … 31</th></tr>
<tr><td>Vers</td><td>HLen</td><td colspan="2">Service Type</td><td colspan="3">Total Length</td></tr>
<tr><td colspan="4">Identification</td><td>Flags</td><td colspan="2">Fragment Offset</td></tr>
<tr><td colspan="2">Time to Live</td><td colspan="2">Protocol</td><td colspan="3">Header Checksum</td></tr>
<tr><td colspan="7">Source IP Address</td></tr>
<tr><td colspan="7">Destination IP Address</td></tr>
<tr><td colspan="6">IP Options</td><td>Padding</td></tr>
<tr><td colspan="7">Data</td></tr>
<tr><td colspan="7">• • • •</td></tr>
</table>

Transport services and protocols

- Provide *logical communication* between **application processes** running on different hosts
- Transport protocols run in end systems
  - sender side: breaks application messages into segments, passes to network layer
  - receiver side: reassembles segments into messages, passes to application layer
- More than one transport protocol available to applications
  - Internet: TCP and UDP

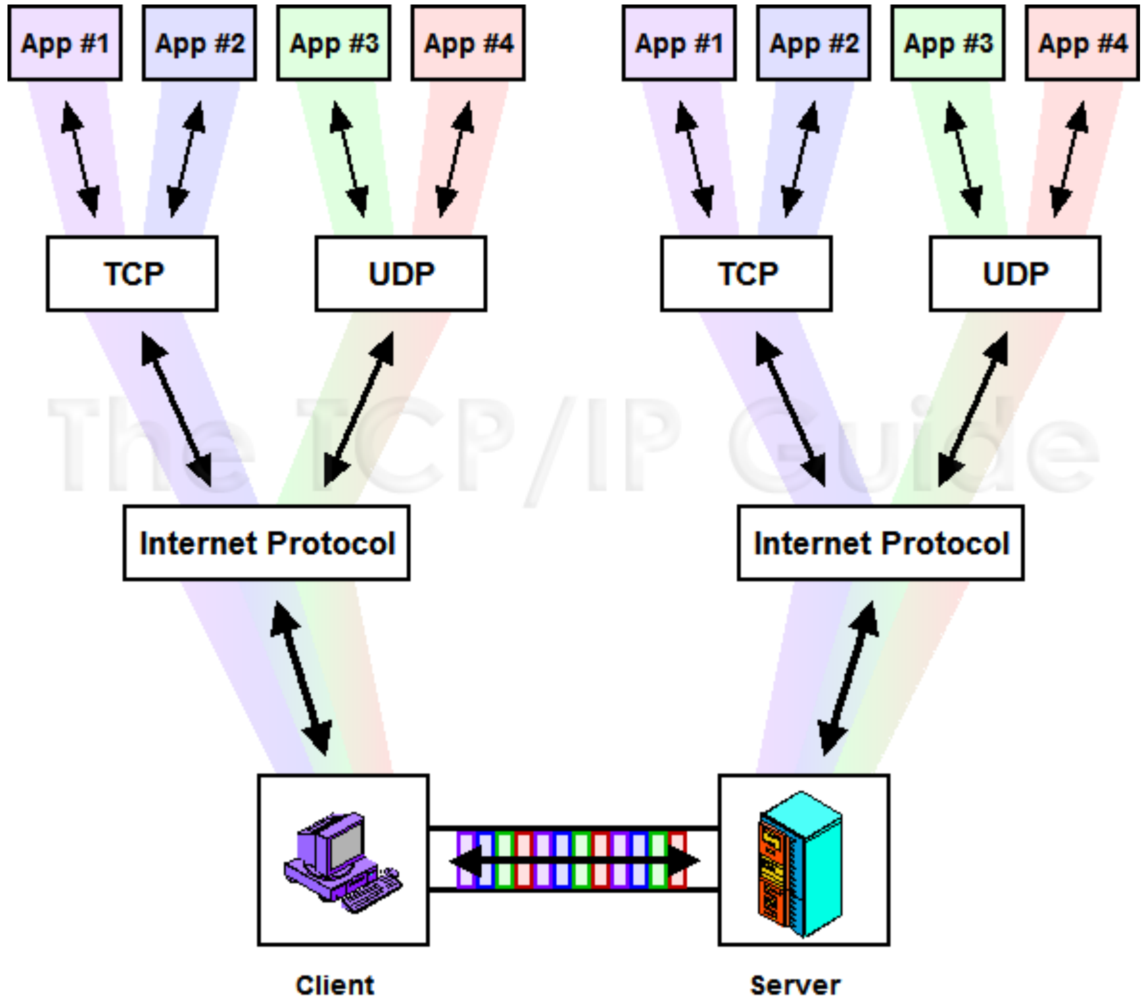


Transport layer

- Basic service: moving data from one application (or process) to a remote application
- Functions:
  - Packetization and reassembly of application data
  - Connection management
  - Error control
  - Reliability
  - Flow control
  - Congestion control (on the Internet)

Transport-layer protocols

- **Reliable, in-order** delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- **Unreliable, unordered** delivery (UDP)
  - none of above
- Services not available:
  - delay guarantees
  - bandwidth guarantees


App #1
App #2
App #3
App #4
App #1
App #2
App #3
App #4
TCP
UDP
TCP
UDP
The TCP/IP Guide
Internet Protocol
Internet Protocol
Client
Server

Socket

- The end-point where a local application process attaches to the network is called a socket

- End point determined by two things:
  - Host address: IP address is *Network Layer*
  - Port number: is *Transport Layer*

- Two end-points determine a connection: socket pair
  - e.g., 128.226.180.163,port 21 + 198.69.10.2,port 1500
  - e.g., 128.226.180.163,port 21 + 198.69.10.2,port 1499

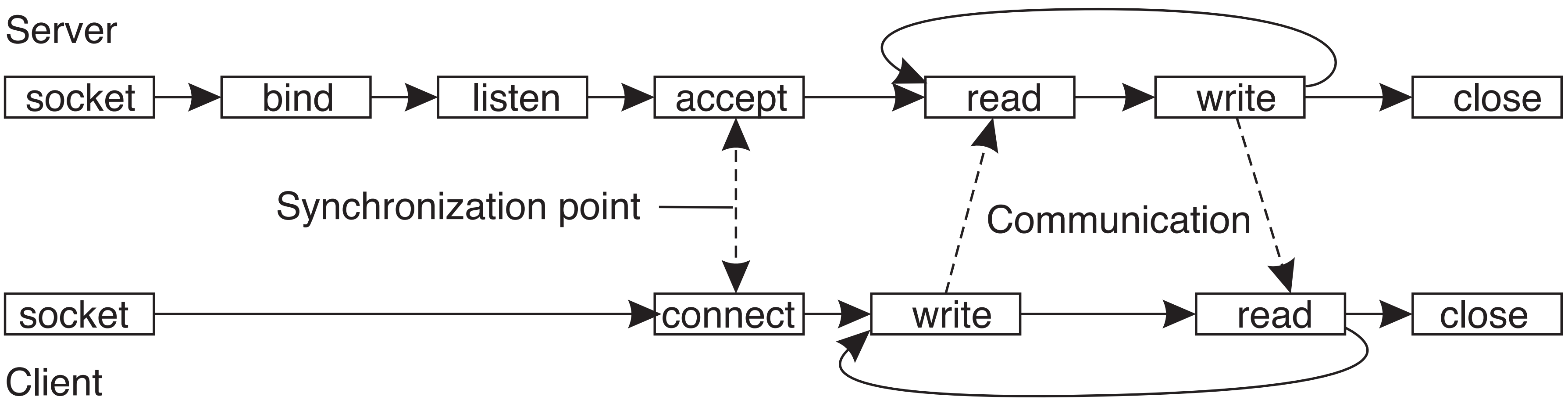
Port numbers

- Basic mechanism for multiplexing applications per host
  - 65,535 ports available
- Ports <1024 are reserved
  - Only privileged processes (e.g., superuser) may access
  - Well-known , reserved services (see /etc/services in Unix):
    - ftp 21/tcp, ssh 22/tcp, telnet 23/tcp, http 80/tcp, ntp 123/tcp , snmp 161/udp
- “I tried to open a port and got an error”
  - Port collision: only one application per port per host
  - Dangling sockets…

Socket API

- Socket is the interface between an application and the network
- The interface defines operations for:
  - Creating a socket
  - Attaching a socket to the network
  - Sending and receiving messages through the socket
  - Closing the socket

TCP socket API



Socket API

```
int sockfd = socket(address_family, type, protocol);
```

- The socket number returned is the socket descriptor for the newly created socket

- `int sockfd = socket (PF_INET, SOCK_STREAM, 0);`
- `int sockfd = socket (PF_INET, SOCK_DGRAM, 0);`

The combination of PF_INET and SOCK_STREAM implies TCP

The combination of PF_INET and SOCK_DGRAM implies UDP

Socket API

- Passive Open (on server)

  int bind(int socket, struct sockaddr *addr, int addr_len)

  int listen(int socket, int backlog)

  int accept(int socket, struct sockaddr *addr, int addr_len)

- Active Open (on client)

  int connect(int socket, struct sockaddr *addr, int addr_len)

- Sending/Receiving Messages

  int send(int socket, char *msg, int mlen, int flags)

  int recv(int socket, char *buf, int blen, int flags)

- The socket API offered by the Linux operating system is in C.
- But many programming languages also provide similar interfaces, providing accesses to the socket interface offered by the operating system.

Multiplexing/Demultiplexing

Demultiplexing at recv host:

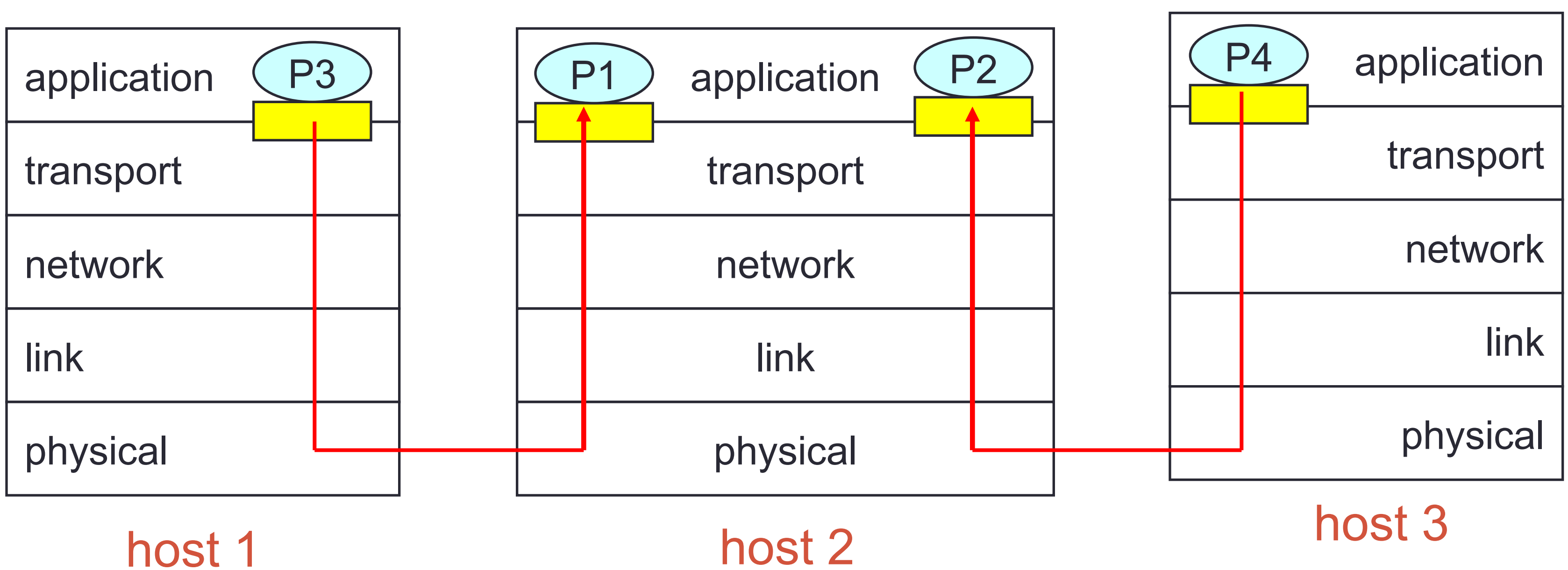
delivering received segments to correct socket

Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

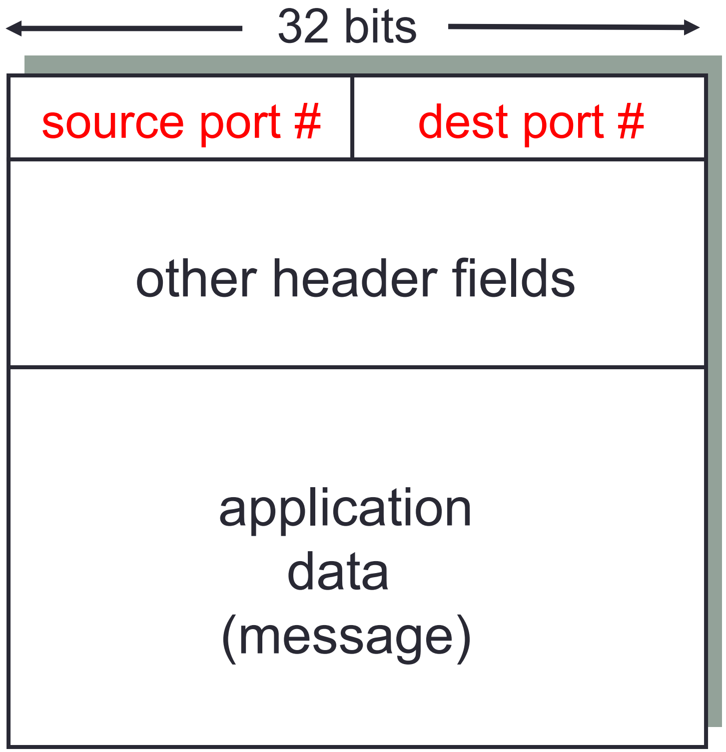
= socket

= process



How demultiplexing works

- Host receives IP packets
  - each *packet* has source IP address, destination IP address
  - each packet carries one transport-layer segment
  - each *segment* has source, destination **port numbers**
- Host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

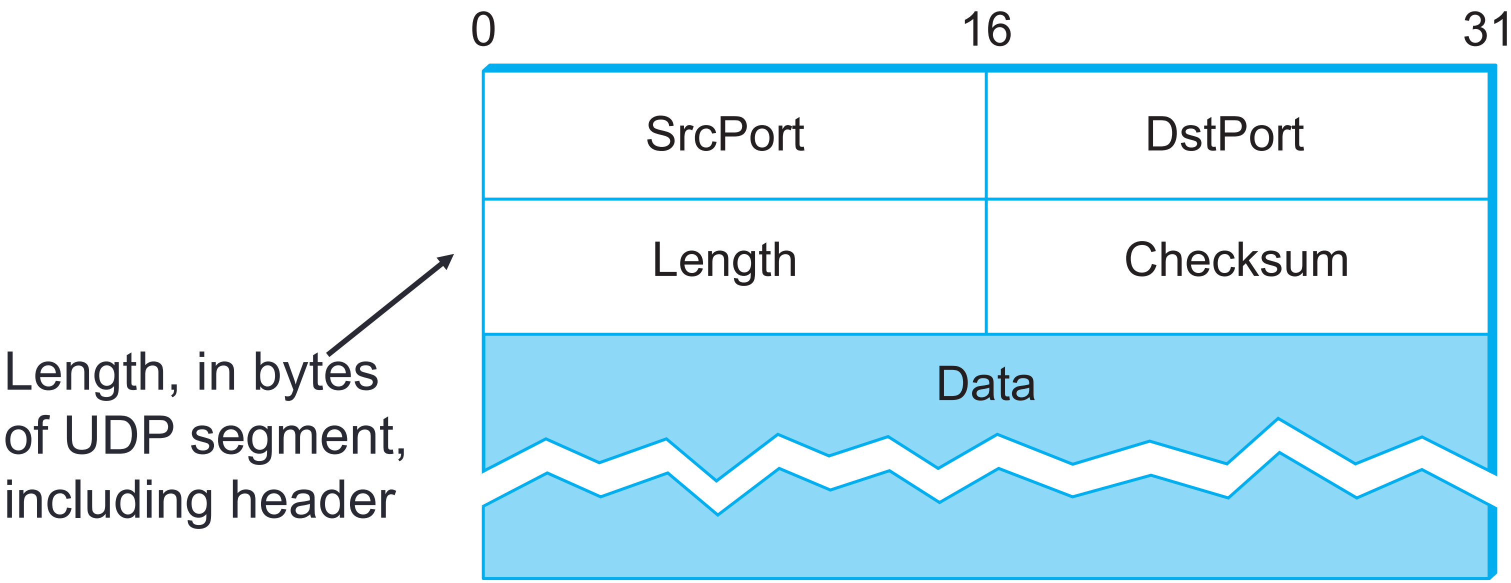
UDP

- “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

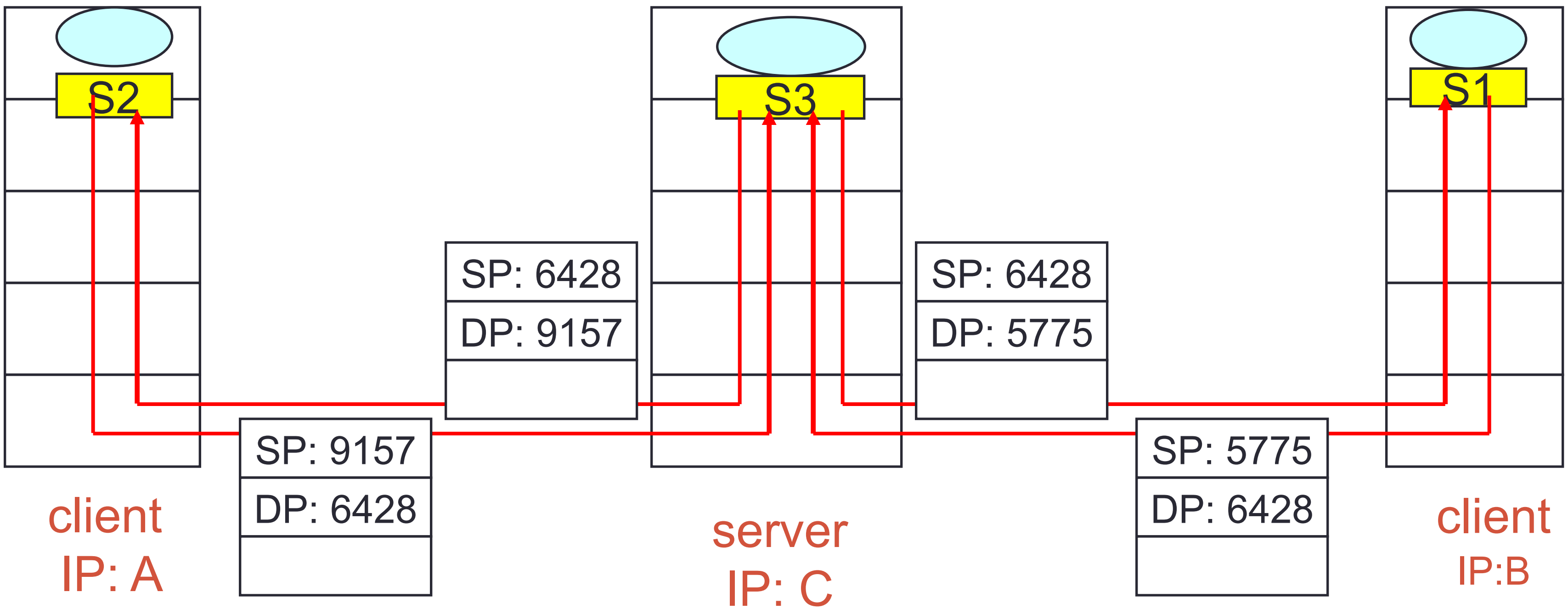
UDP format


0
16
31
SrcPort
DstPort
Length
Checksum
Data
Length, in bytes
of UDP segment,
including header

Connectionless demultiplexing

- **UDP** socket identified by
  - (dest IP address, dest port number)
- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with *different* source IP addresses and/or source port numbers directed to *same* socket

Connectionless demultiplexing



SP provides "return address"

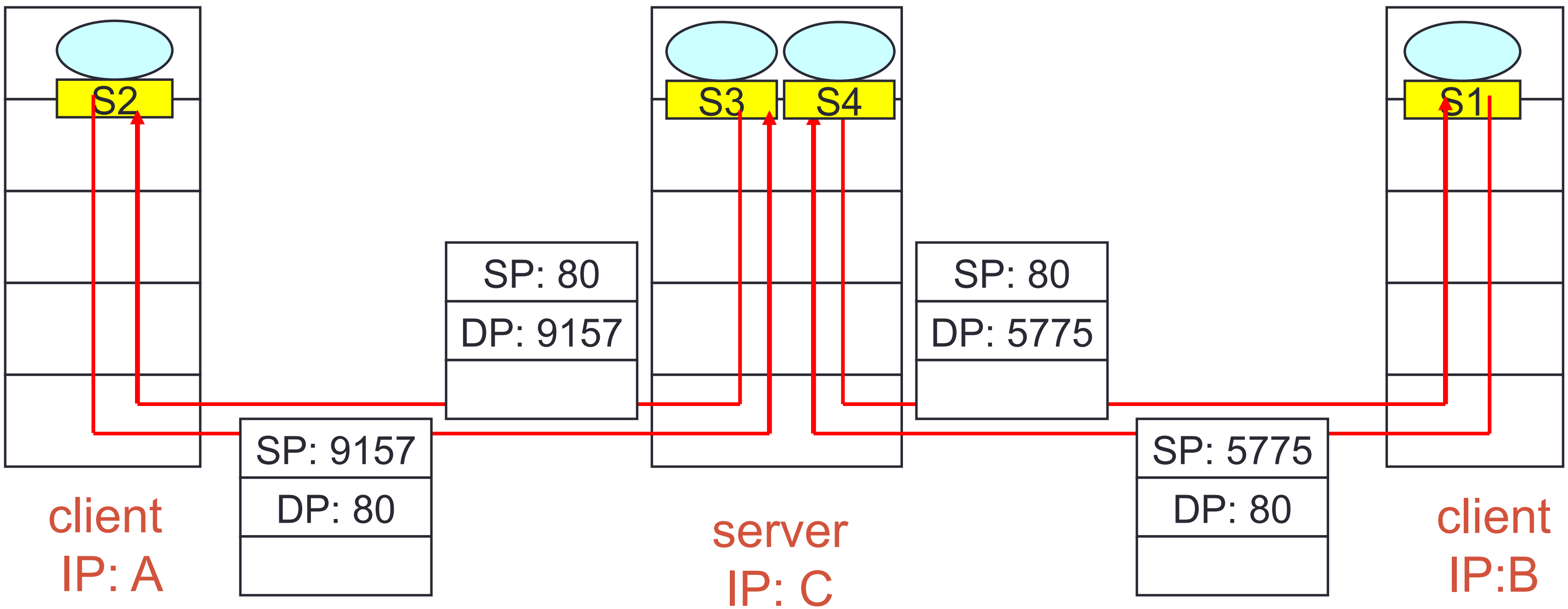
TCP

- TCP provides
  - reliable, stream delivery service
  - full-duplex (bidirectional) communication
  - flow control
  - congestion control
  - connection establishment and destruction

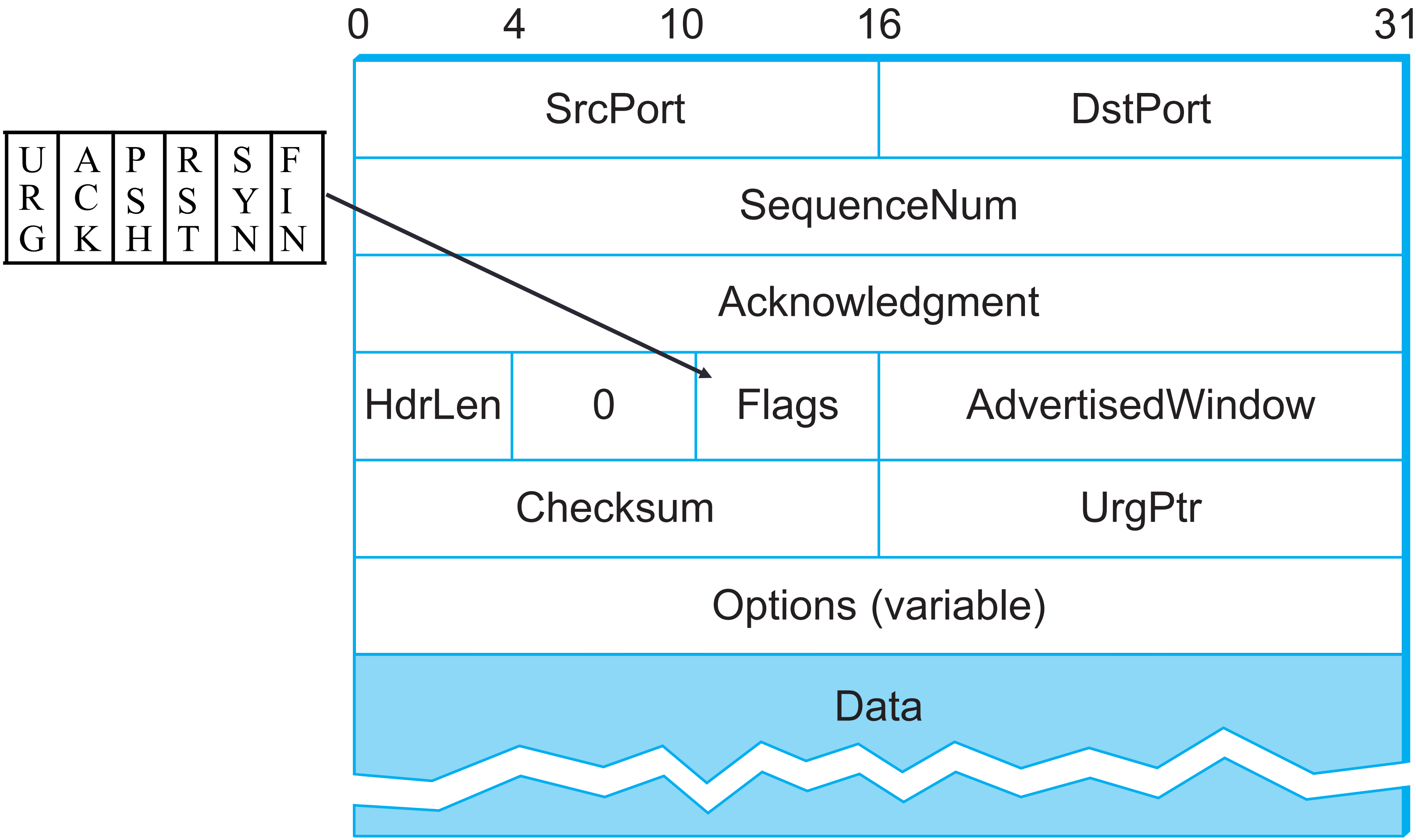
Connection-oriented demultiplexing

- **TCP** socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Receiving host uses all four values to direct segment to appropriate socket
- Server hosts may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client

Connection-oriented demultiplexing



TCP segment format



Stream delivery service

- Data from an application is treated as a stream of bytes
- The stream is divided into **segments** for delivery.
  - this division is up to TCP; application data boundaries are ignored
- Conceptually, **every byte** has a sequence #.
- Only the sequence # of the first byte of the segment is sent.

Three-way hand shaking

- Process A initiates a connection to process B.
- A sends a segment with ISN *x*, chosen according A's clock, and with SYN bit set to 1, this is the **connection request** message.

Three-way hand shaking

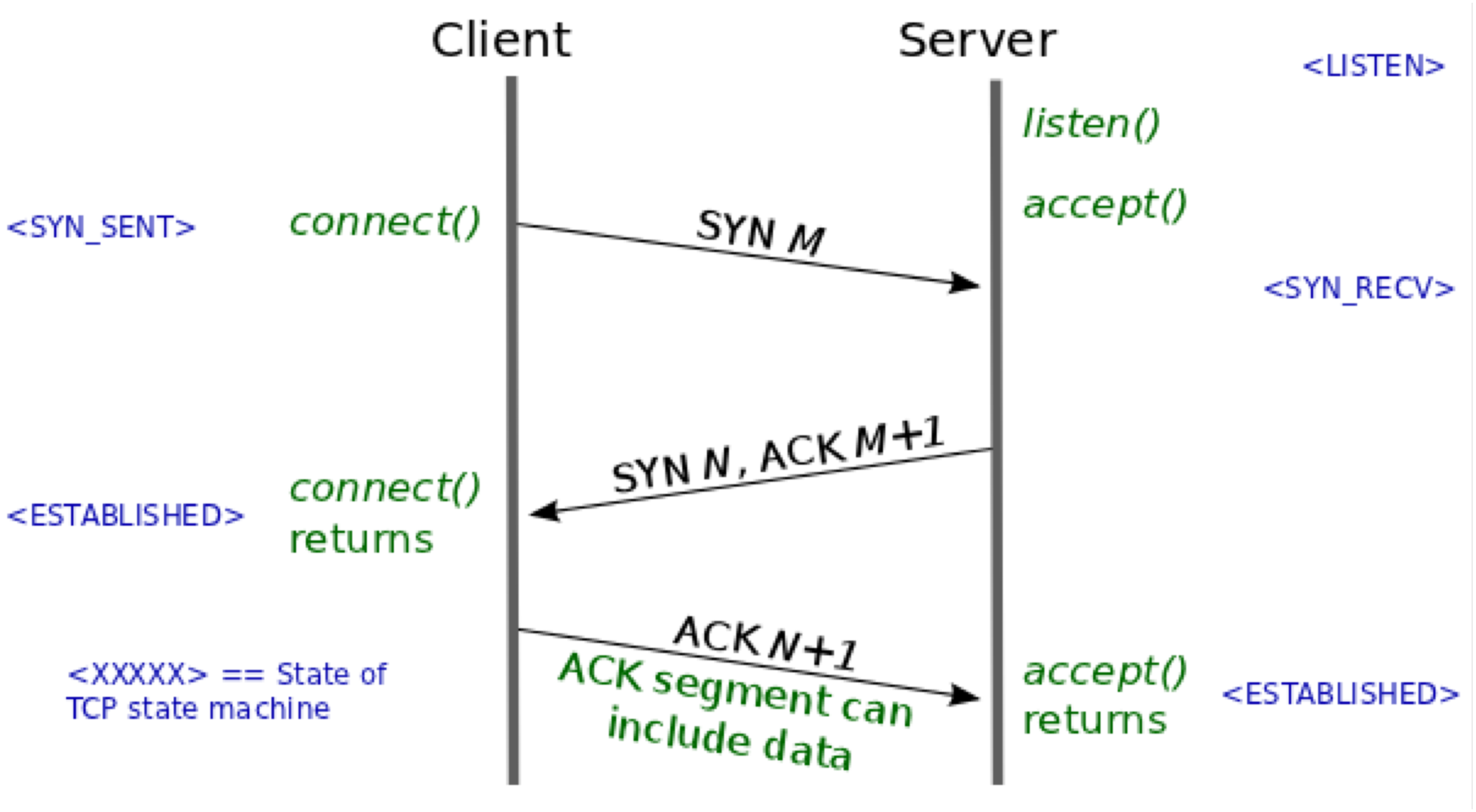
- B returns a segment whose ack=x+1, seq=y, the ISN of B chosen according to B's clock, and SYN=1.
  - this is the **connection acceptance** message.

- After receiving the acceptance, A considers the connection established.

Three-way hand shaking

- A then sends B a segment with ack=y+1 and SYN=0.
  - this is the **connection confirmation** message
  - all segments from this point on will have SYN=0
- After receiving the confirmation, B considers the connection established.

Visualize It



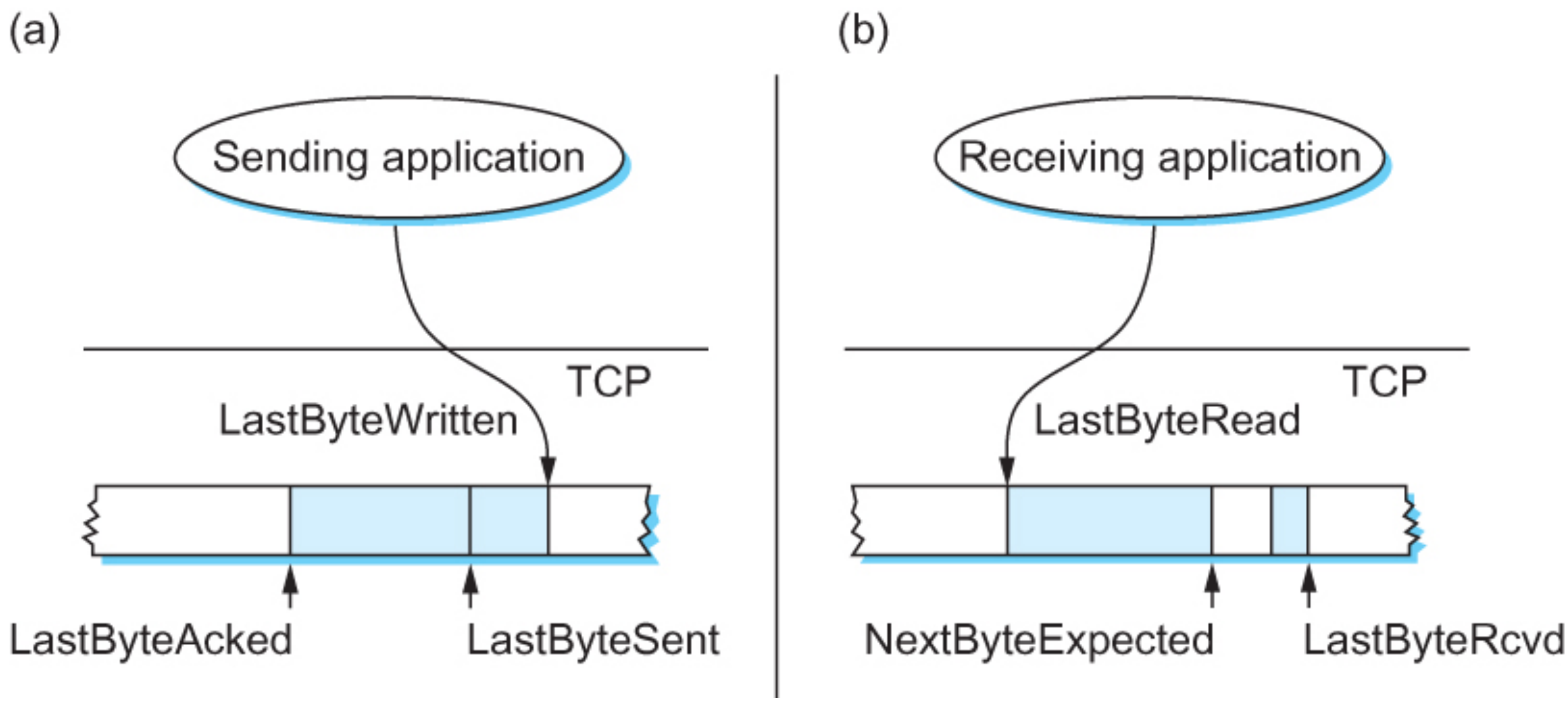
https://lwn.net/Articles/508865/

TCP acknowledgments

- TCP acknowledges the next byte expected (not the last one received).
- Ack=*x* indicates *all* bytes up to and including *x-1* has been successfully received.
- Also, a pure Ack is simply a segment with no data.

- What if a segment is received out of order? (next slide)

Sliding Window in TCP



Relationship between the TCP send buffer (a) and the receive buffer (b).

TCP flow control

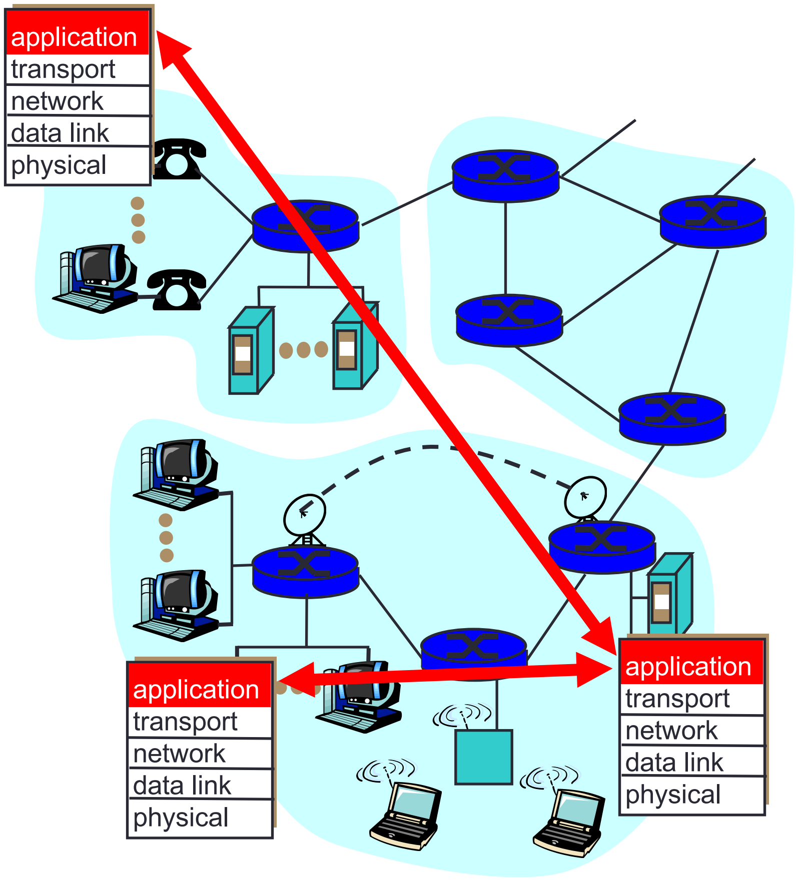
- Flow control aims to eliminate the possibility of overflowing the receiver's buffer.
- Receiver side:
  - LastByteRcvd − LastByteRead ≤ MaxRcvBuffer
  - **AdvertisedWindow** = MaxRcvBuffer − ((NextByteExpected − 1) − LastByteRead)
- Sender side:
  - **LastByteSent − LastByteAcked ≤ AdvertisedWindow**

TCP congestion control

- When load builds up in a network, congestion builds up.
- Senders limit the rate at which it sends traffic as a function of the perceived network congestion.
- If there is little to no perceived congestion,
  - Increase the sending rate
- If congestion is detected,
  - Decrease the sending rate
- Detect congestion via:
  - Lost packets

Applications and application-layer protocols

- Application: communicating, distributed processes
  - e.g., e-mail, Web, instant messaging
  - running in end systems (hosts)
  - exchange messages
- Application-layer protocols
  - one "piece" of an application
  - define messages exchanged by applications and actions taken
  - use communication services provided by lower layer protocols (TCP, UDP)



Application-layer protocol defines

- Types of messages exchanged, e.g., request & response messages
- Syntax of message types: what fields in messages & how fields are delineated
- Semantics of the fields, i.e., meaning of information in fields
- Rules for when and how processes send & respond to messages

Public-domain protocols:

- Defined in RFCs, allows for interoperability
- e.g., HTTP, FTP

Proprietary protocols:

- e.g., Skype

Client-server paradigm

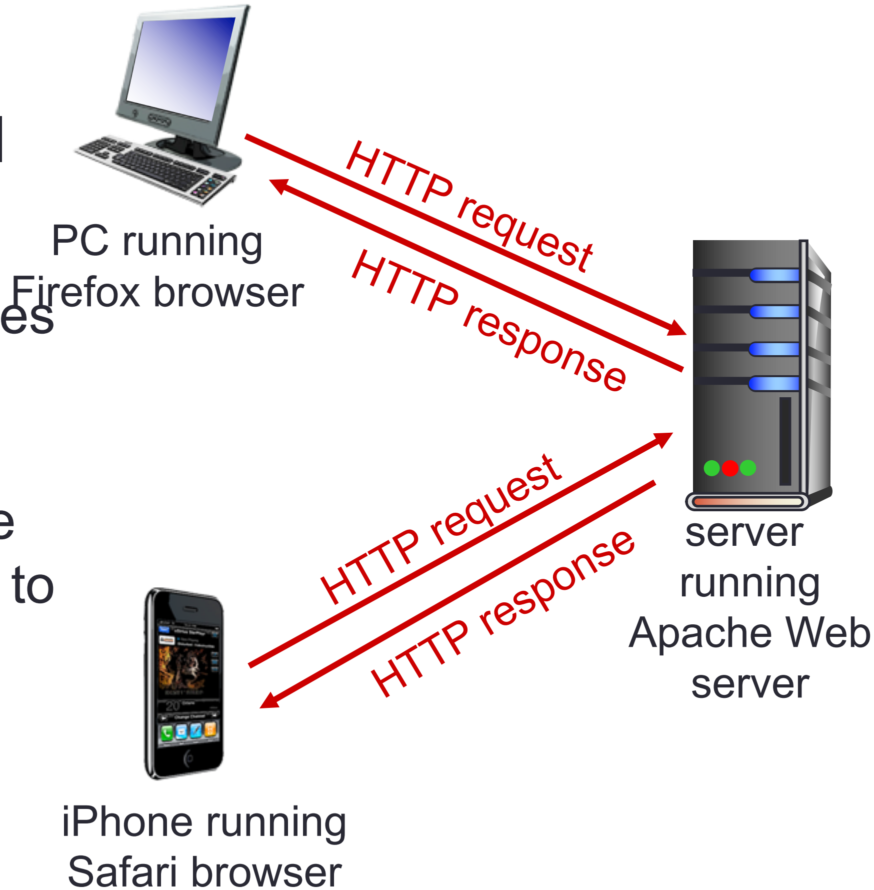
- Typical network app has two pieces: client and server
- Client:
  - initiates contact with server (“speaks first”)
  - typically requests service from server
  - e.g., web client implemented in browser
- Server:
  - provides requested service to client
  - e.g., web server sends requested web page

World Wide Web

- Server provides access to web pages.
- Browsers request web pages from server using Hypertext Transfer Protocol (HTTP).
- Objects on the web are identified by Uniform Resource Locators (URL).

HTTP

- A simple client-server protocol
- Web's application layer protocol
- Client/server model
  - client: browser that requests, receives (using the HTTP protocol), and displays Web objects
  - server: Web server sends (using the HTTP protocol) objects in response to requests
- HTTP/1.0: RFC 1945
- HTTP/1.1: RFC 2616
- HTTP/2: RFC 7540



HTTP

- Uses TCP:
  - client initiates TCP connection to server, typically 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
- 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

Web and HTTP

- Web page consists of objects
- Object can be HTML file, JPEG image, Javascript, audio file, video file, …
- Web page consists of base HTML-file which includes several referenced objects
- Each object is addressable by a URL
- Example URL:

$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

HTTP connections

- Non-persistent HTTP
  - At most one object is sent over a TCP connection.
  - HTTP/1.0 uses non-persistent HTTP

- Persistent HTTP
  - Multiple objects can be sent over single TCP connection between client and server.
  - HTTP/1.1 uses persistent connections in default mode

Non-persistent HTTP

Suppose user enters URL (contains text, references to 10 jpeg images)

`www.someSchool.edu/someDepartment/index.html`

1a. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on port 80

1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object /someDepartment/index.html

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

4. HTTP server closes TCP connection.

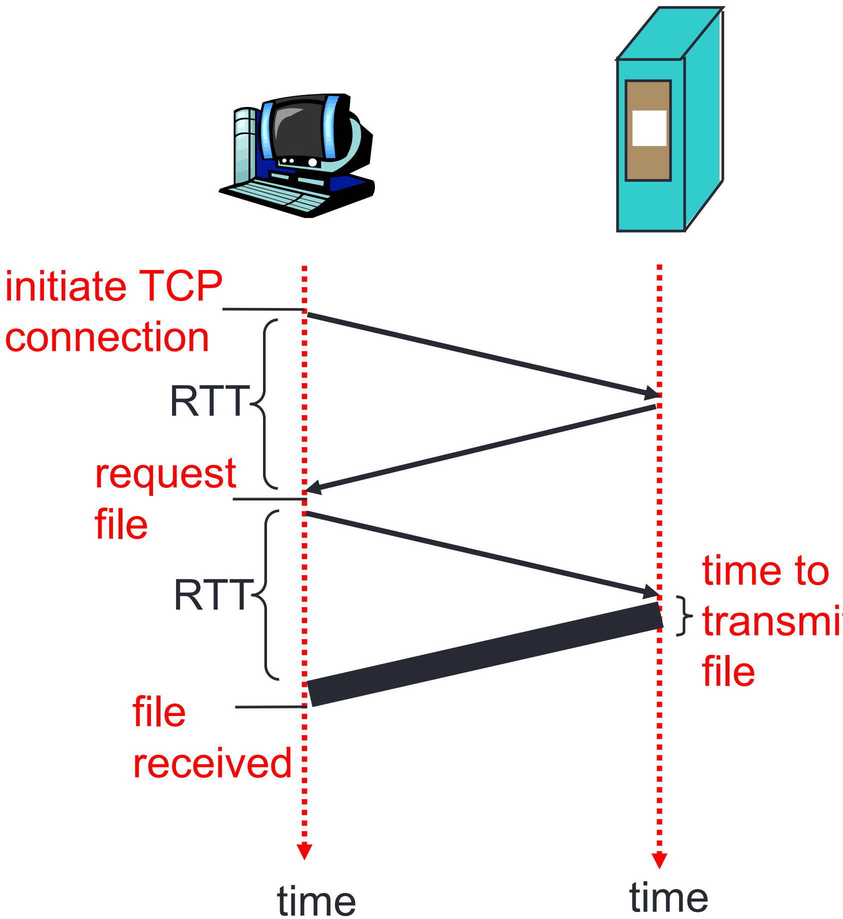
5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

time

6. Steps 1-5 repeated for each of 10 jpeg objects

Modeling response time

- Definition of RTT: time to send a small packet to travel from client to server and back.
- Response time:
  - one RTT to initiate TCP connection
  - one RTT for HTTP request and first few bytes of HTTP response to return
  - file transmission time
- total = 2RTT+transmit time

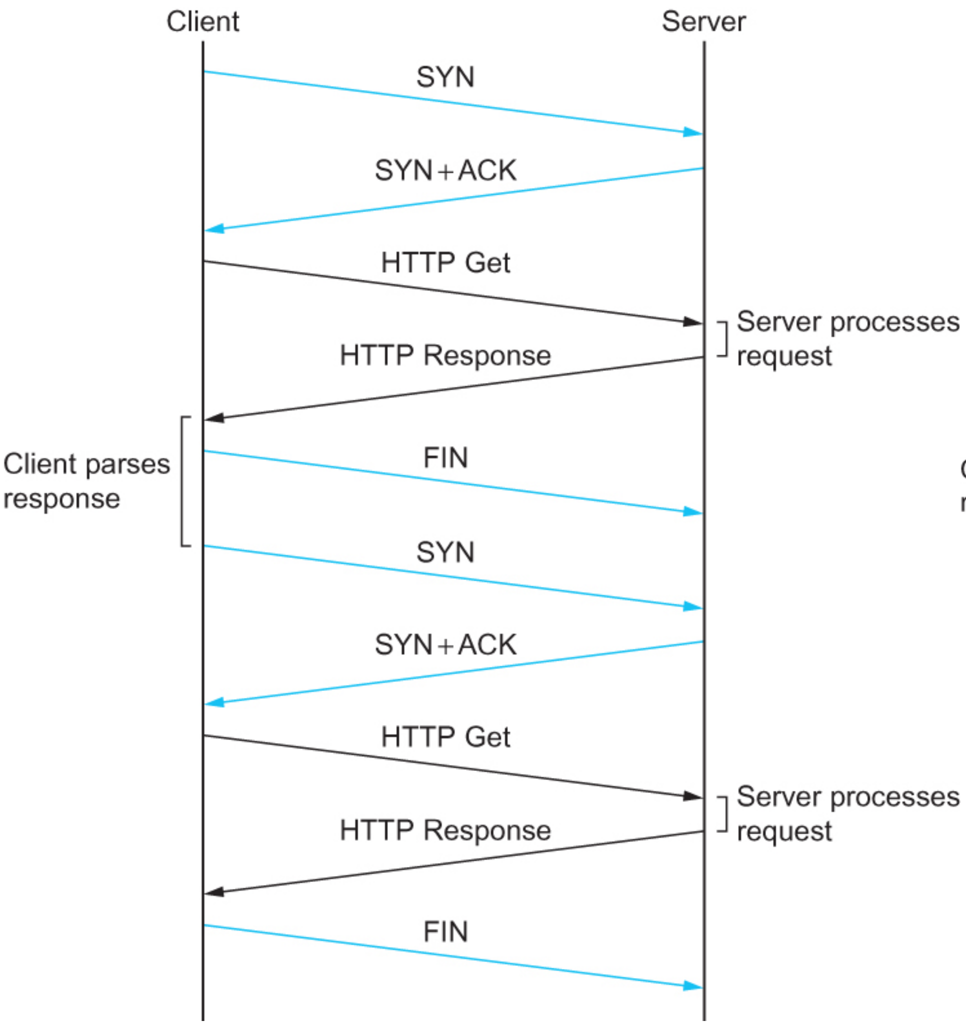


Persistent HTTP

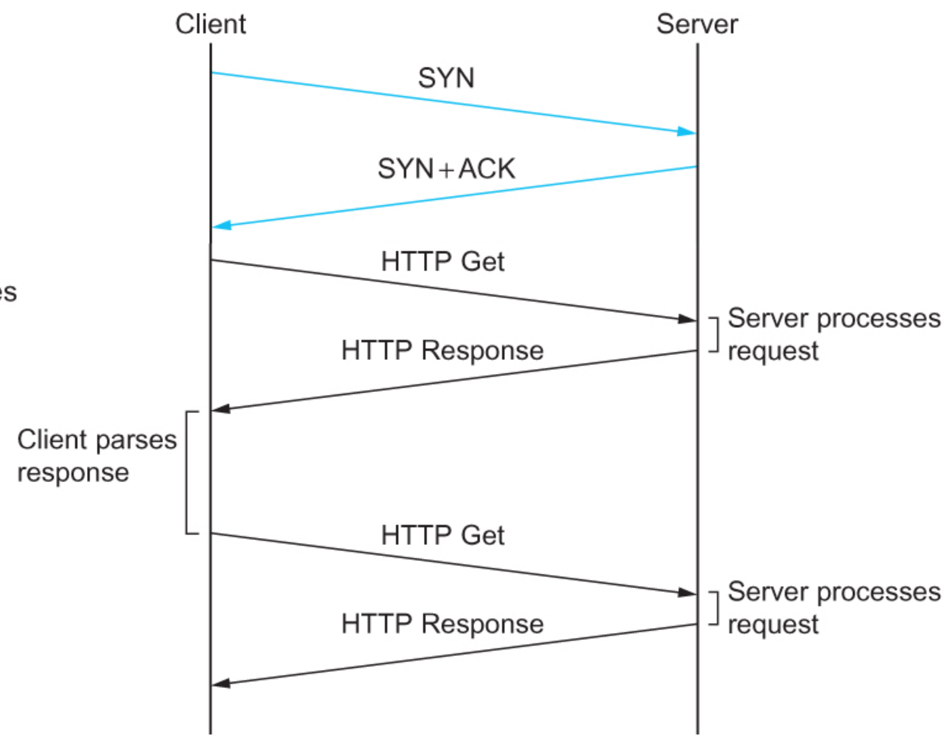
- Non-persistent HTTP issues:
  - requires at least 2 RTTs per object
  - OS must work and allocate host resources for each TCP connection

- Persistent HTTP
  - server leaves connection open after sending response
  - subsequent HTTP messages between same client/server are sent over connection

Non-persistent HTTP vs. persistent HTTP



Non-persistent HTTP behavior



Persistent HTTP behavior

Persistent HTTP

- Persistent without pipelining:
  - client issues new request only when previous response has been received
  - one RTT for each referenced object
- Persistent with pipelining:
  - default in HTTP/1.1
  - client sends requests as soon as it encounters a referenced object
  - as little as one RTT for all the referenced objects

HTTP message: request

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

carriage return character

line-feed character

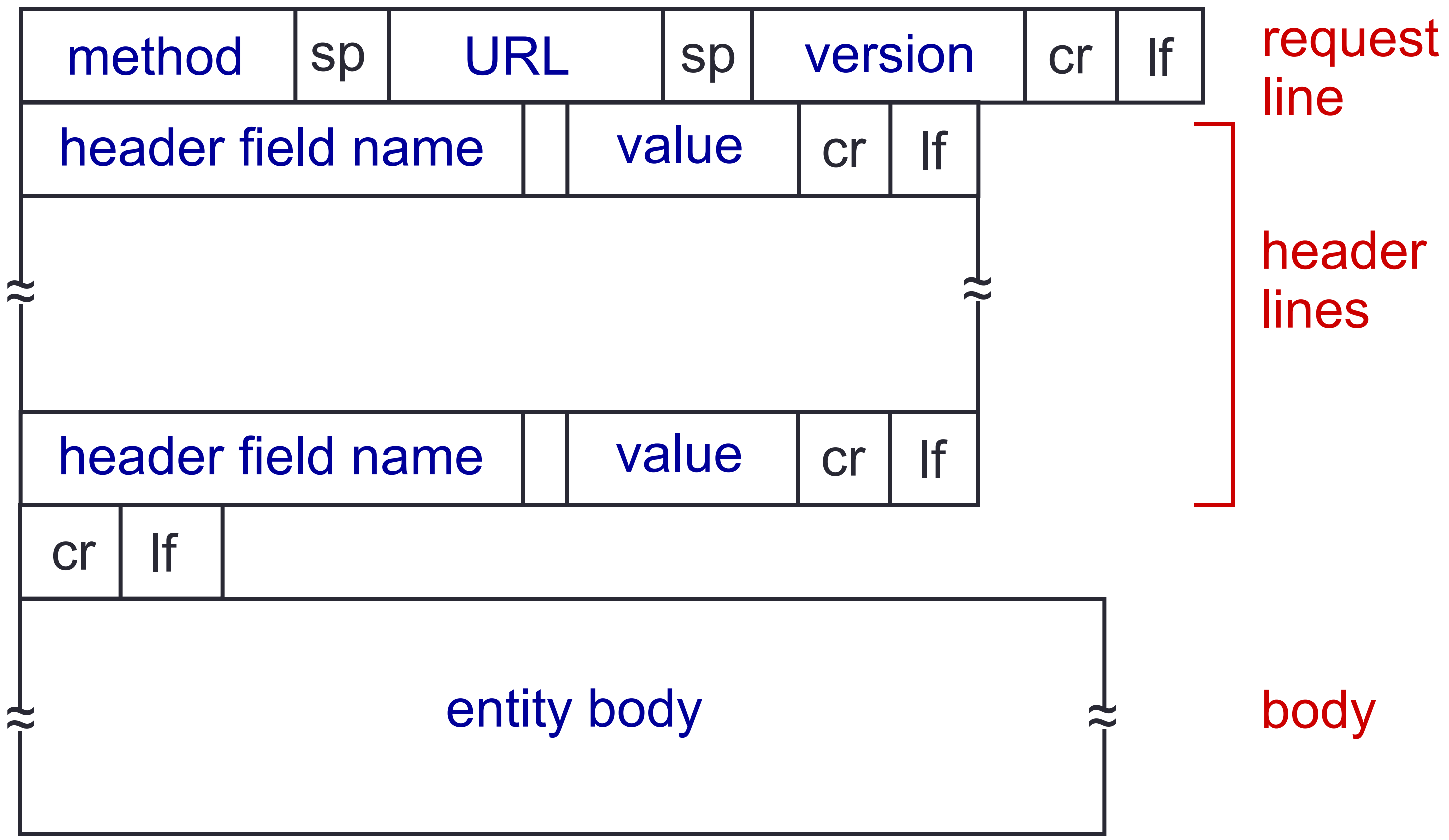
request line (GET, POST, HEAD commands)

header lines

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

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\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
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

HTTP request: general format



sp: space
cr: carriage return (\r)
lf: line feed (\n)

Uploading form input

- POST method:
  - Web page often includes form input
  - Input is uploaded to server in entity body
- URL method:
  - Uses GET method
  - Input is uploaded in URL field of request line:

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

Method types

- HTTP/1.0
  - GET
  - POST
  - HEAD
    - asks server to leave requested object out of response
- HTTP/1.1
  - GET, POST, HEAD
  - PUT
    - uploads file in entity body to path specified in URL field
  - DELETE
    - deletes file specified in the URL field

HTTP message: response

status line (protocol status code status phrase)

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

header lines

data, e.g., requested HTML file

HTTP response status codes

Status codes appear in the 1st line in server to client response message.

A few sample codes:

- 200 OK
  - request succeeded, requested object later in this message
- 301 Moved Permanently
  - requested object moved, new location specified later in this message (Location:)
- 400 Bad Request
  - request message not understood by server
- 404 Not Found
  - requested document not found on this server
- 505 HTTP Version Not Supported

Trying out HTTP

1. Telnet to a Web server:

```
telnet cs.binghamton.edu 80
```

Opens TCP connection to port 80 (default HTTP server port) at cs.binghamton.edu. Anything typed is sent to port 80 at cs.binghamton.edu

2. Type in an HTTP GET request:

```
GET /~yaoliu/courses/cs557/test.html HTTP/1.1
Host: cs.binghamton.edu
```

By typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

3. Look at response message sent by HTTP server!

(or use Wireshark to look at captured HTTP request/response)

Reading

- Textbooks on computer networks
  - Chapters on IP, TCP, and HTTP
- Socket tutorial by Paul Krzyzanowski@Rutgers
  - http://www.cs.rutgers.edu/~pxk/352/notes/sockets/index.html
- Beej's famous socket tutorial
  - http://www.beej.us/guide/bgnet/html/multi/index.html
- Socket programming documentation pages of your chosen programming language: Python (preferred), C/C++, Java.