Overview continued
and
sockets

EECS 489 Computer Networks

http://www.eecs.umich.edu/courses/eecs489/w07

Z. Morley Mao

Wednesday Jan 10, 2007

Acknowledgement: Some slides taken from Kurose&Ross and Katz&Stoica

Administrivia

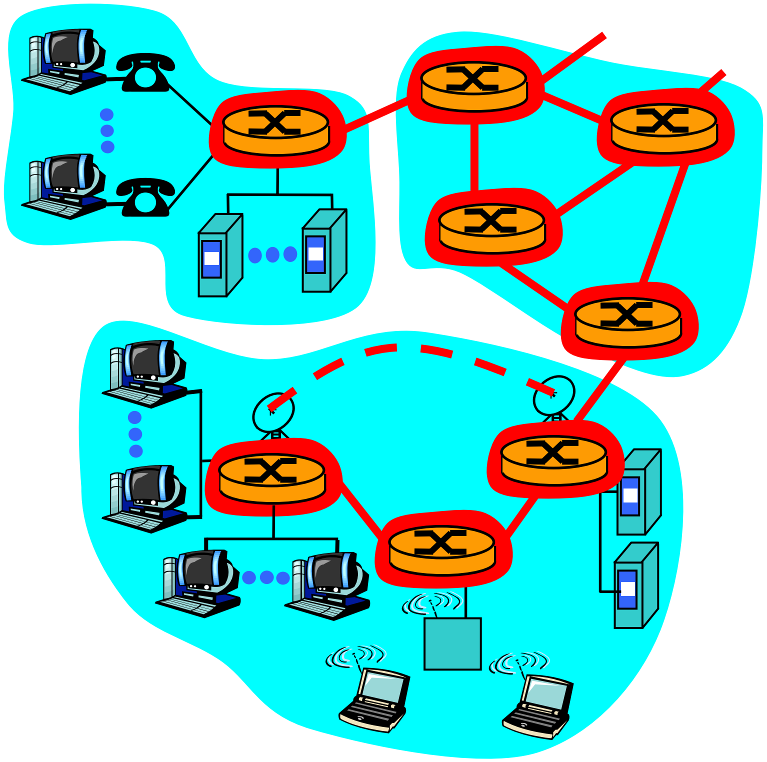
- Homework 1 is online later today
- Class Phorum: http://phorum.eecs.umich.edu
- Class mailing list: eecs489@eecs.umich.edu
- Please read chapter 1 of Kurose's book
- Any questions?

Small Review

- What is the difference between circuit switching and packet switching?
- What is the difference between connection-oriented and connectionless services?
- What is the difference between circuit switching and connection-oriented service?

The Network Core

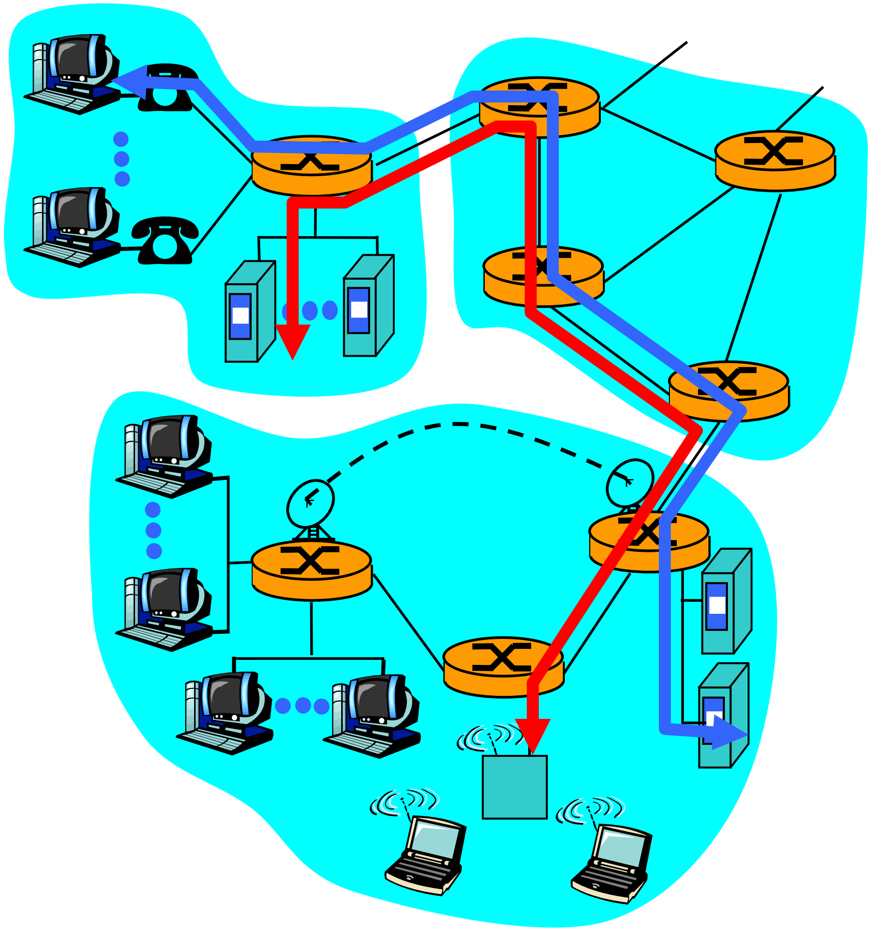
- mesh of interconnected routers
- the fundamental question: how is data transferred through net?
  - circuit switching: dedicated circuit per call: telephone net
  - packet-switching: data sent thru net in discrete "chunks"



Network Core: Circuit Switching

End-end resources reserved for “call”

- link bandwidth, switch capacity
- dedicated resources: no sharing
- circuit-like (guaranteed) performance
- call setup required



Network Core: Circuit Switching

network resources (e.g., bandwidth) divided into “pieces”

- pieces allocated to calls
- resource piece *idle* if not used by owning call *(no sharing)*

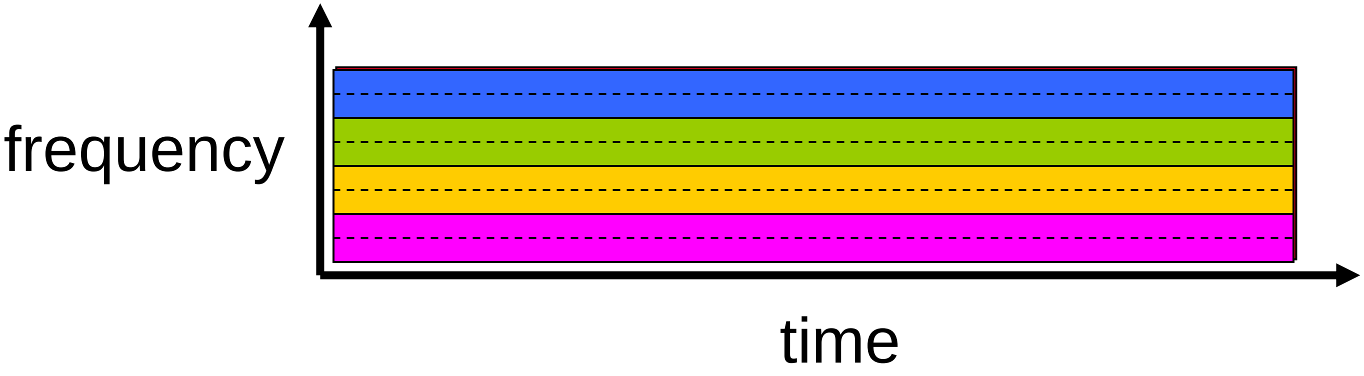
- dividing link bandwidth into “pieces”
  - frequency division
  - time division

Circuit Switching: FDM and TDM

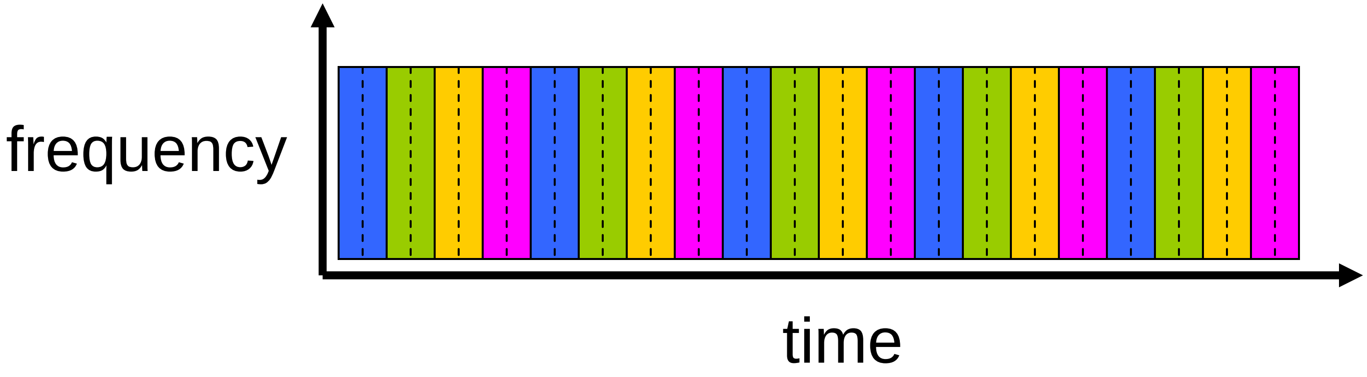
Example:

4 users 🟦 🟩 🟨 🟪

FDM



TDM



Network Core: Packet Switching

each end-end data stream divided into *packets*

- user A, B packets *share* network resources
- each packet uses full link bandwidth
- resources used *as needed*

Bandwidth division into “pieces”

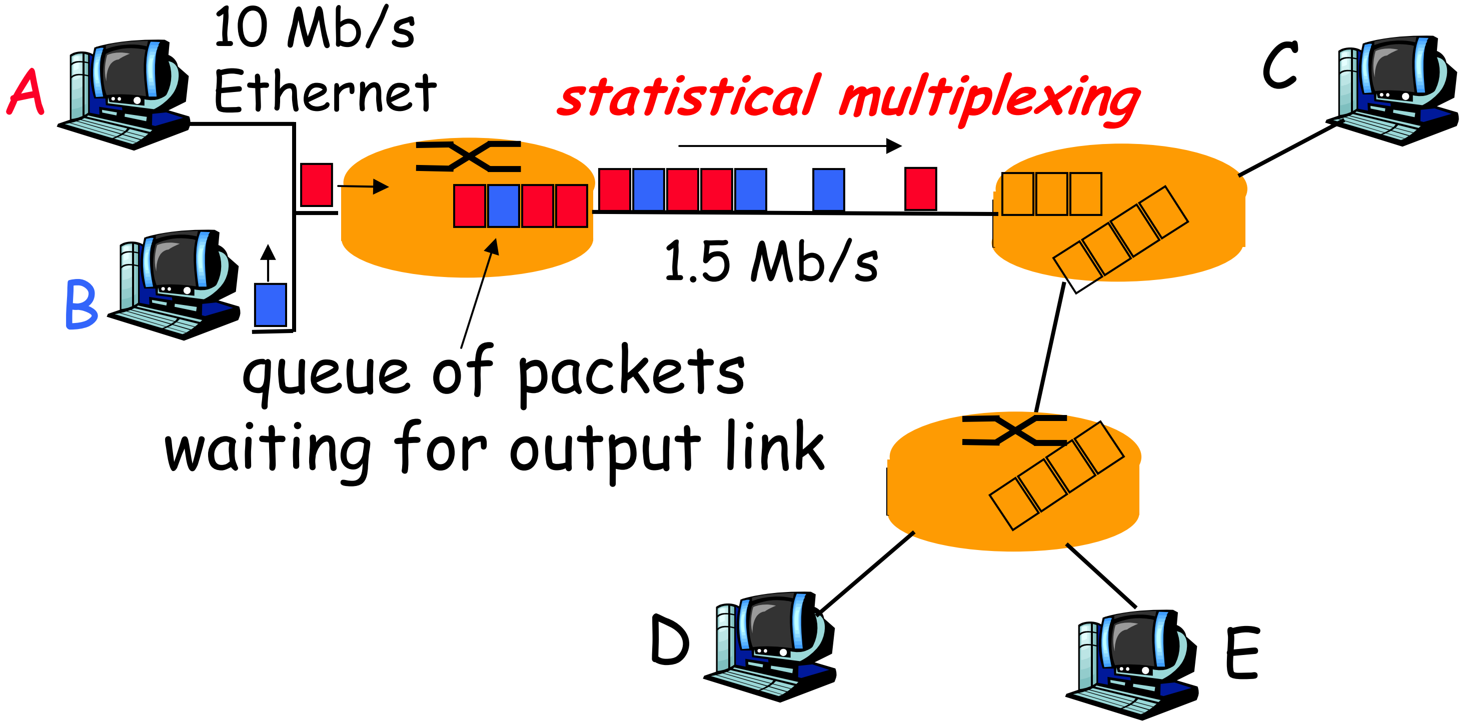
Dedicated allocation

Resource reservation

resource contention:

- aggregate resource demand can exceed amount available
- congestion: packets queue, wait for link use
- store and forward: packets move one hop at a time
  - Node receives complete packet before forwarding

Packet Switching: Statistical Multiplexing



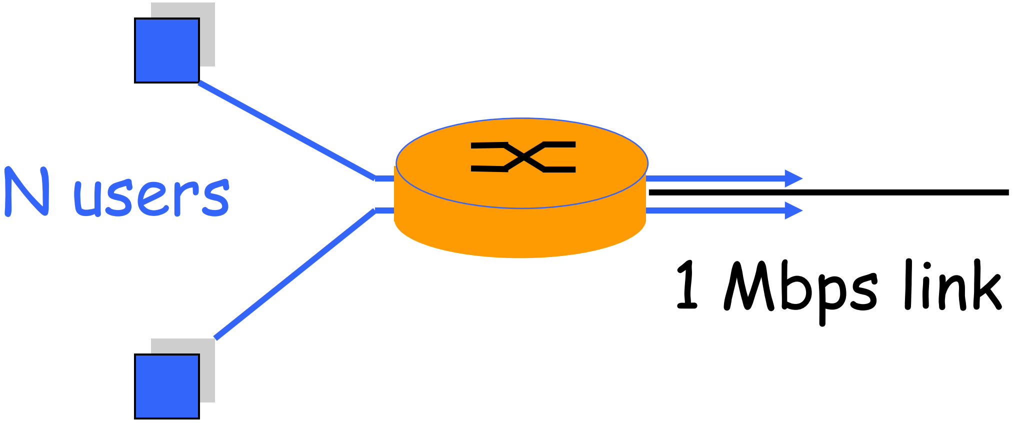
Sequence of A & B packets does not have fixed pattern ➔ ***statistical multiplexing***.

In TDM each host gets same slot in revolving TDM frame.

Packet switching versus circuit switching

Packet switching allows more users to use network!

- 1 Mb/s link
- each user:
  - 100 kb/s when "active"
  - active 10% of time
- circuit-switching:
  - 10 users
- packet switching:
  - with 35 users, probability > 10 active less than .0004
  - 1-Sum of the probabilities that 1,2,…10 users are active

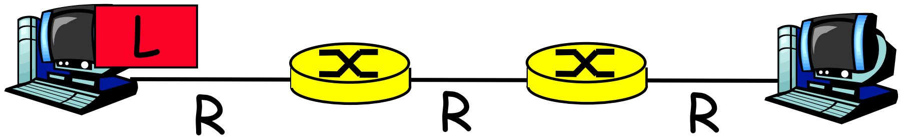


Packet switching versus circuit switching

Is packet switching a “slam dunk winner?”

- Great for bursty data
  - resource sharing
  - simpler, no call setup
- More resilient to failures
- Excessive congestion: packet delay and loss
  - protocols needed for reliable data transfer, congestion control
- Q: How to provide circuit-like behavior?
  - bandwidth guarantees needed for audio/video apps
  - still an unsolved problem
  - Overprovisioning often used

Packet-switching: store-and-forward



- Takes L/R seconds to transmit (push out) packet of L bits on to link or R bps
- Entire packet must arrive at router before it can be transmitted on next link:

  store and forward
- delay = 3L/R

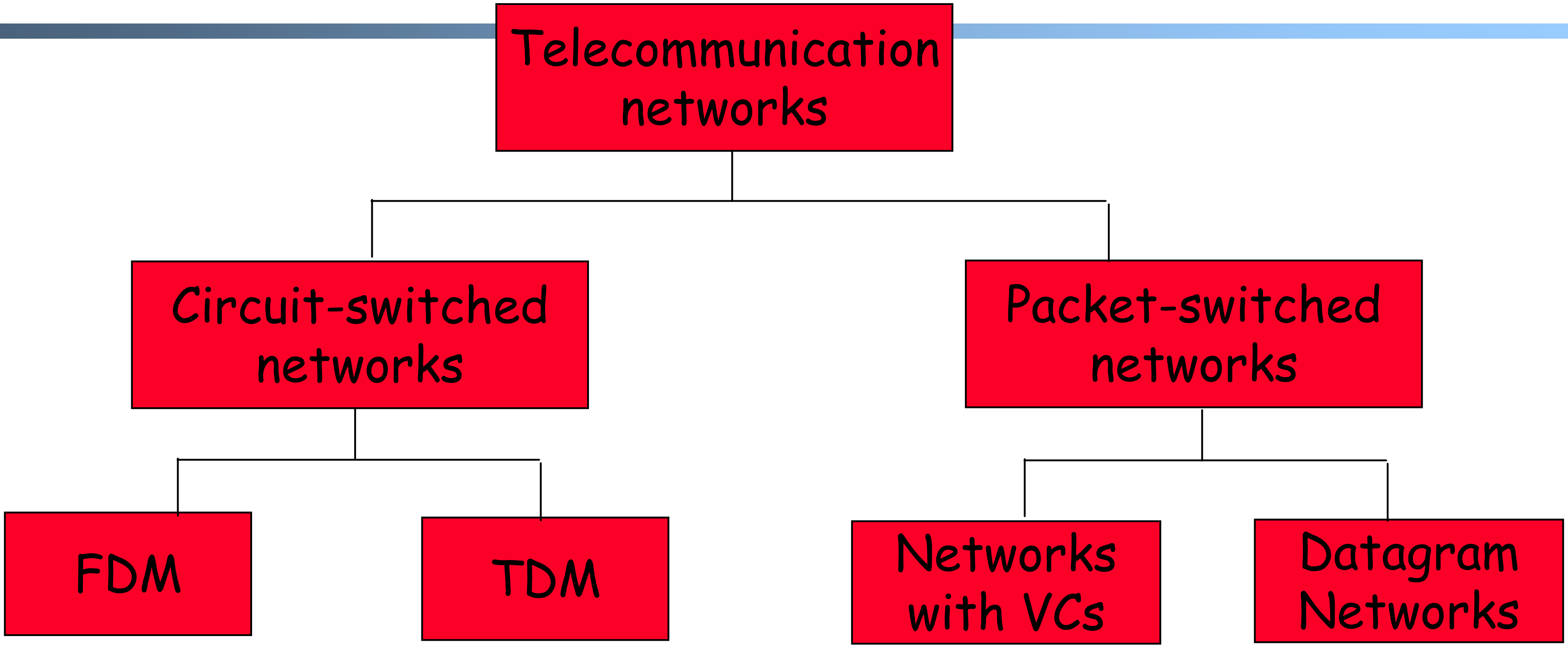
Example:

- L = 7.5 Mbits
- R = 1.5 Mbps
- delay = 15 sec

Packet-switched networks: forwarding

- *Goal:* move packets through routers from source to destination
  - we'll study several path selection (i.e. routing) algorithms
- datagram network:
  - *destination address* in packet determines next hop
  - routes may change during session
  - analogy: driving, asking directions
- virtual circuit network:
  - each packet carries tag (virtual circuit ID), tag determines next hop
  - fixed path determined at *call setup time*, remains fixed thru call
  - *routers maintain per-call state*

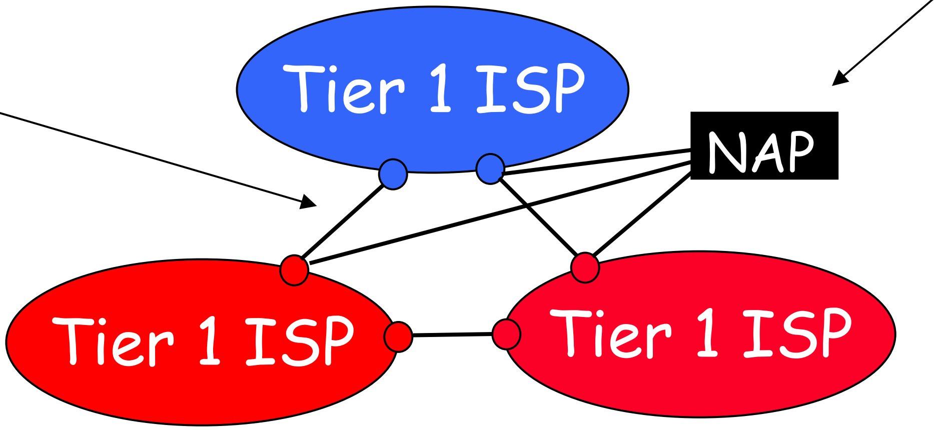
Network Taxonomy



- Datagram network is *neither* connection-oriented nor connectionless.
- Internet provides both connection-oriented (TCP) and connectionless services (UDP) to apps.

Internet structure: network of networks

- roughly hierarchical
- at center: “tier-1” ISPs (e.g., UUNet, BBN/Genuity, Sprint, AT&T), national/international coverage
  - treat each other as equals

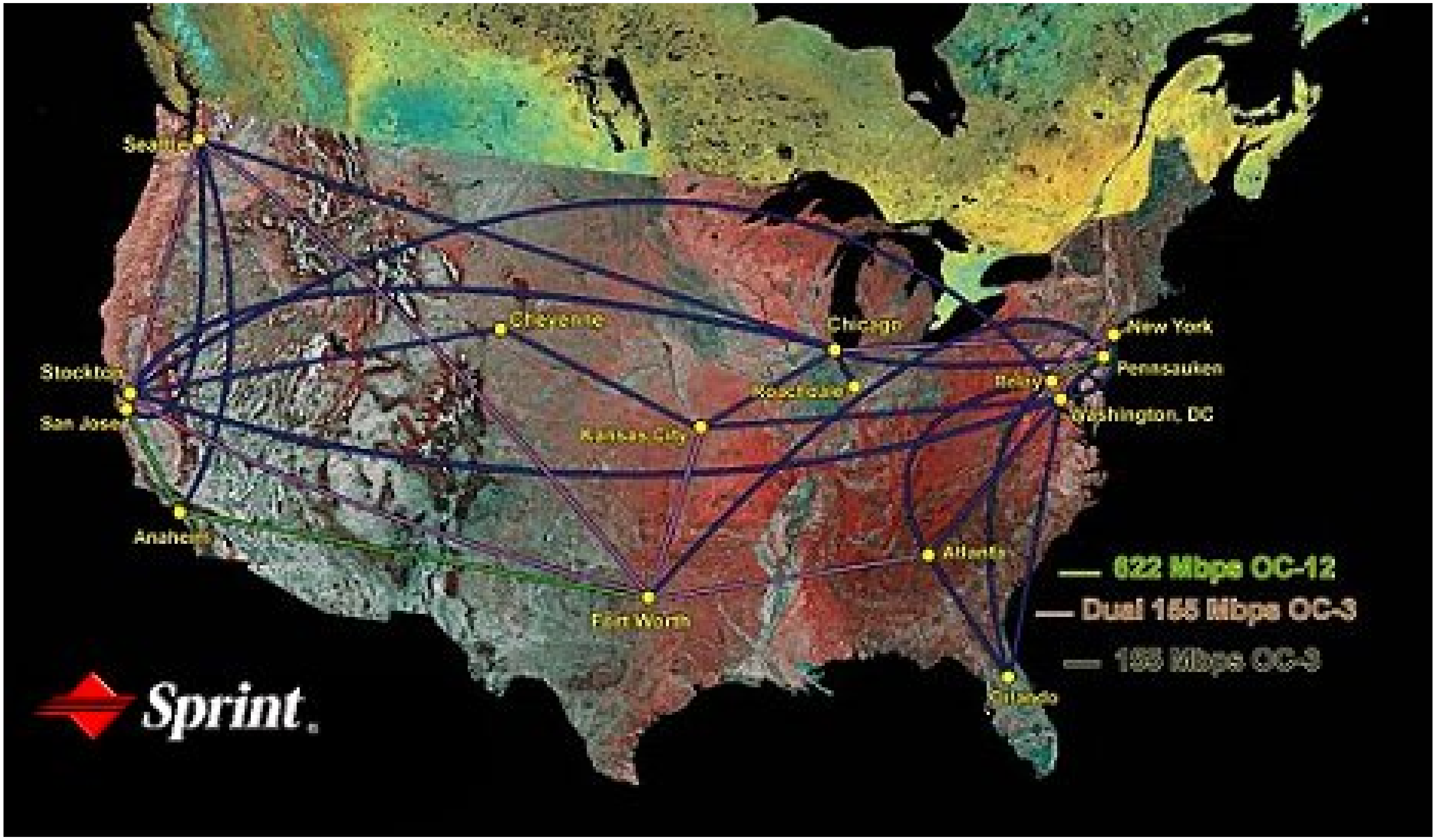


Tier-1 providers interconnect (peer) privately

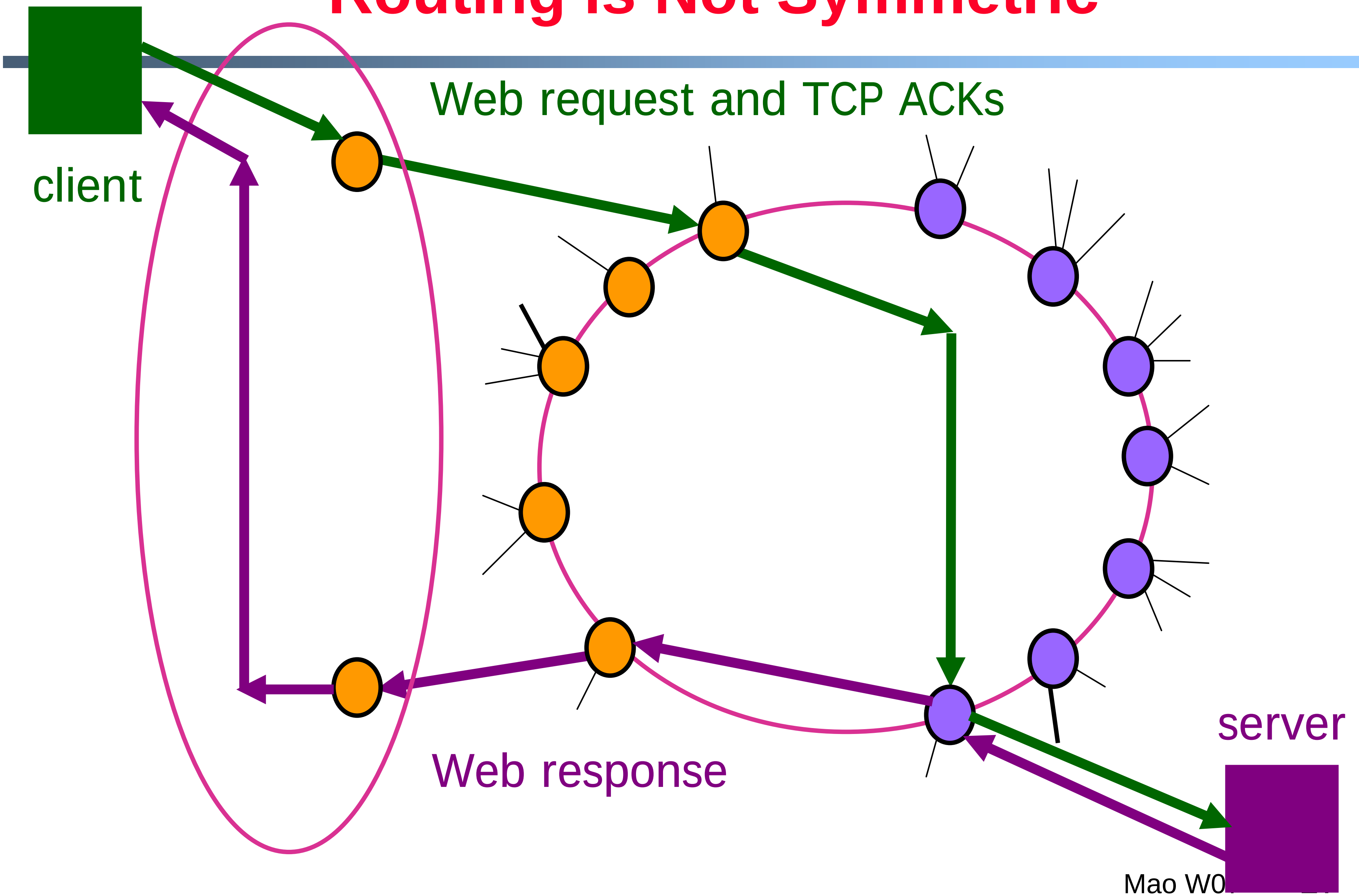
Tier-1 providers also interconnect at public network access points (NAPs)

Tier-1 ISP: e.g., Sprint

Sprint US backbone network



Routing is Not Symmetric

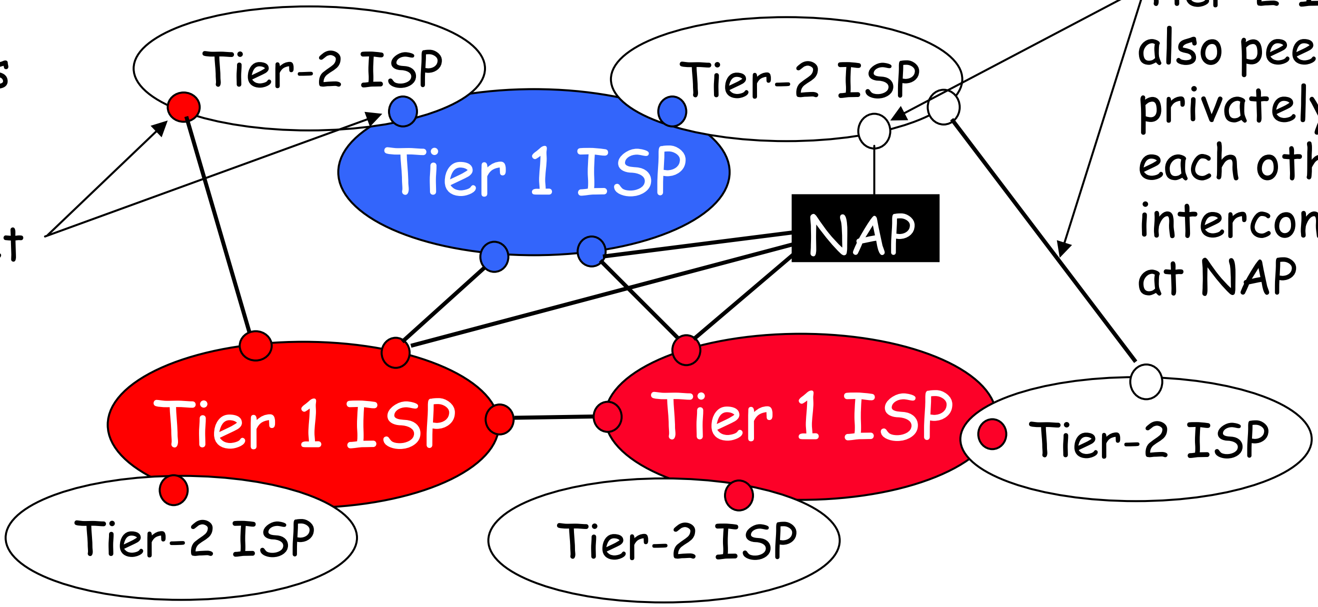


Internet structure: network of networks

- “Tier-2” ISPs: smaller (often regional) ISPs
  - Connect to one or more tier-1 ISPs, possibly other tier-2 ISPs

Tier-2 ISP pays tier-1 ISP for connectivity to rest of Internet

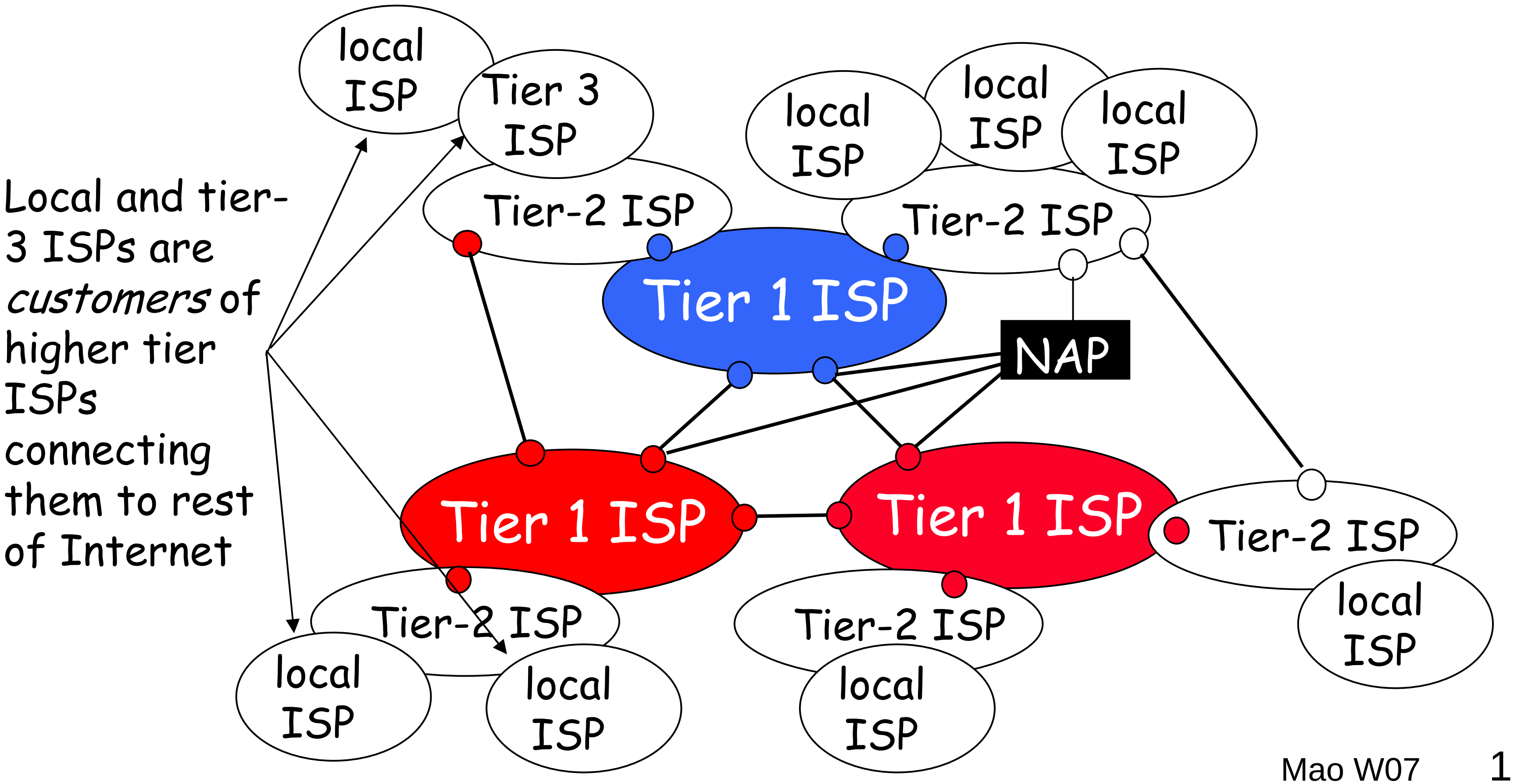
❑ tier-2 ISP is *customer* of tier-1 provider



Tier-2 ISPs also peer privately with each other, interconnect at NAP

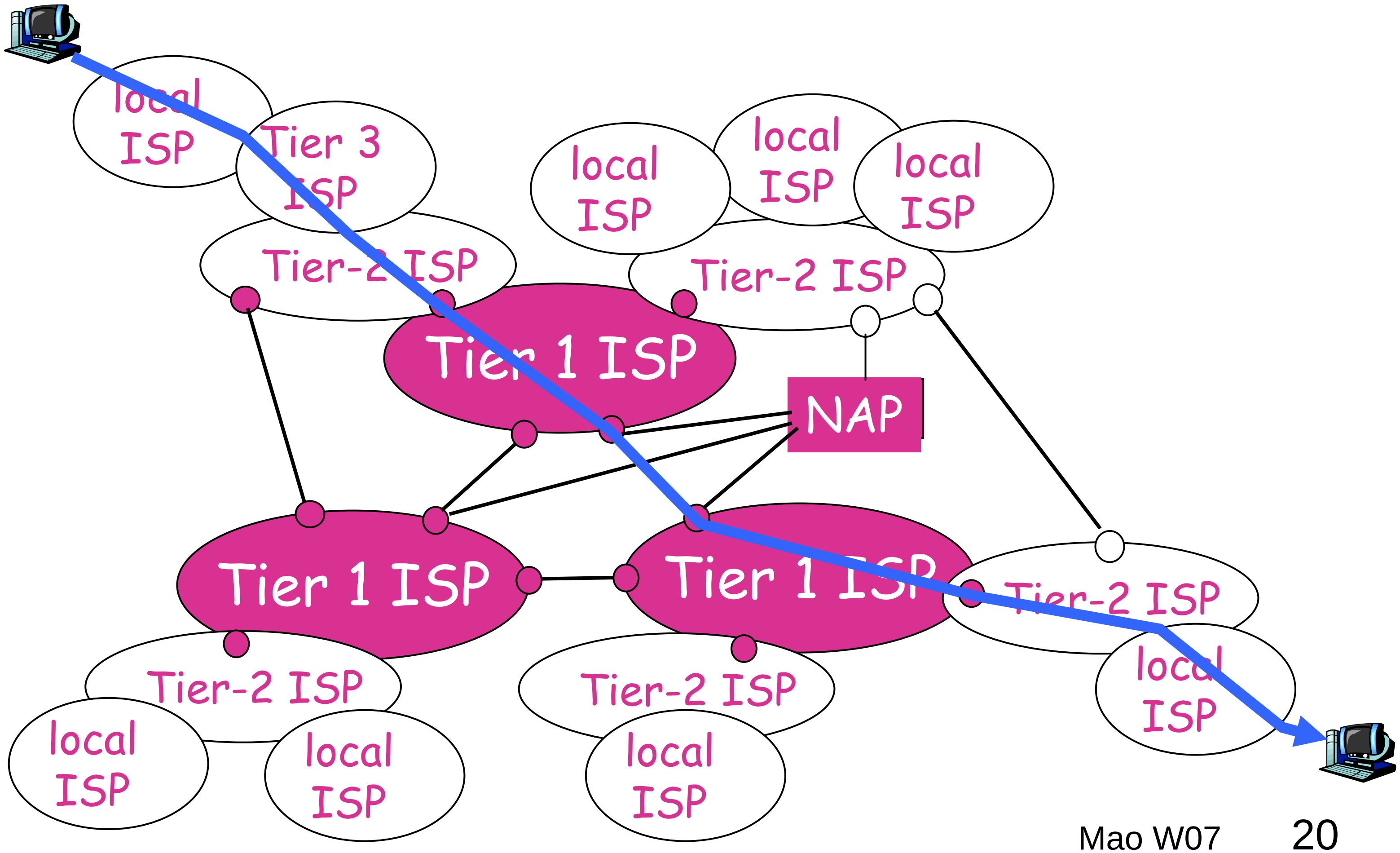
Internet structure: network of networks

- "Tier-3" ISPs and local ISPs
  - last hop ("access") network (closest to end systems)



Internet structure: network of networks

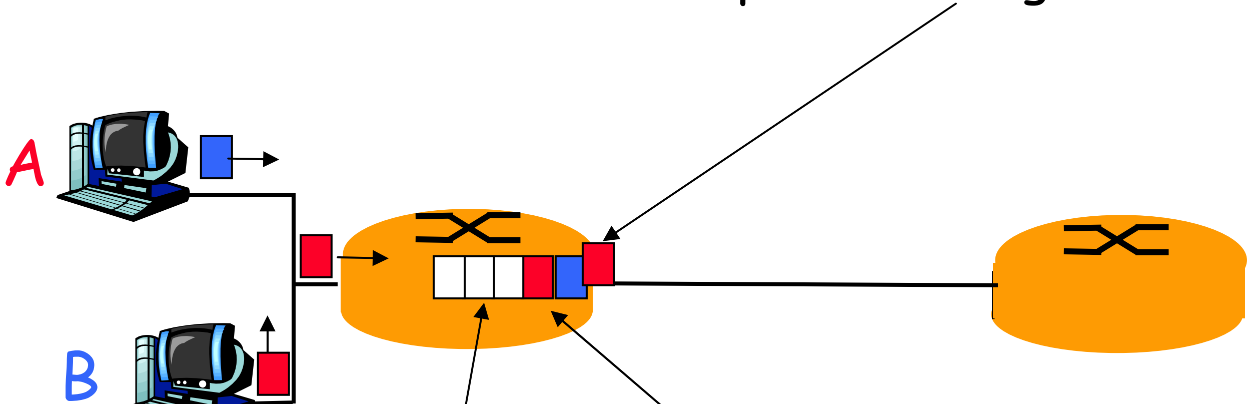
- a packet passes through many networks!



How do loss and delay occur?

packets *queue* in router buffers

- packet arrival rate to link exceeds output link capacity
- packets queue, wait for turn



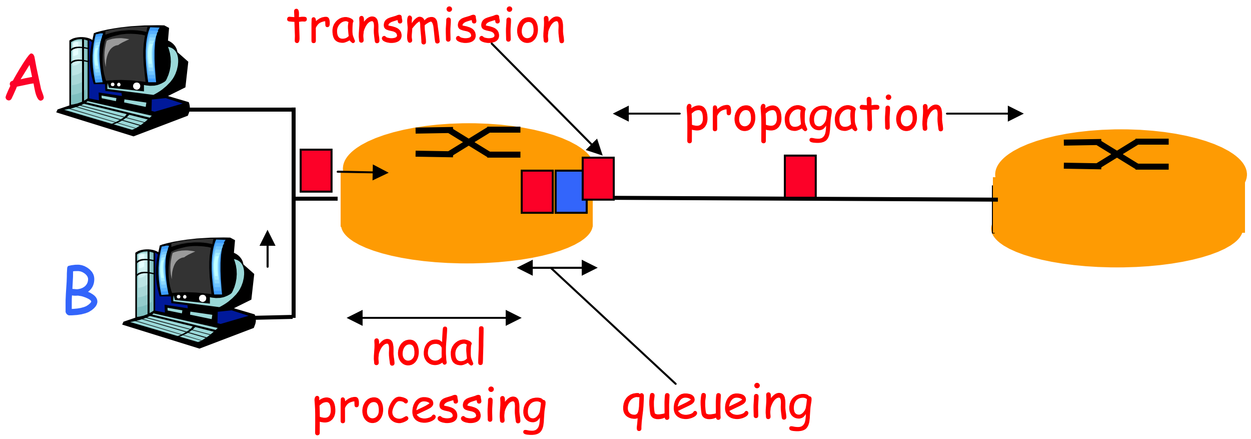
packet being transmitted (delay)

packets queueing (delay)

free (available) buffers: arriving packets dropped (loss) if no free buffers

Four sources of packet delay

- 1. nodal processing:
  - check bit errors
  - determine output link
- 2. queueing
  - time waiting at output link for transmission
  - depends on congestion level of router



Delay in packet-switched networks

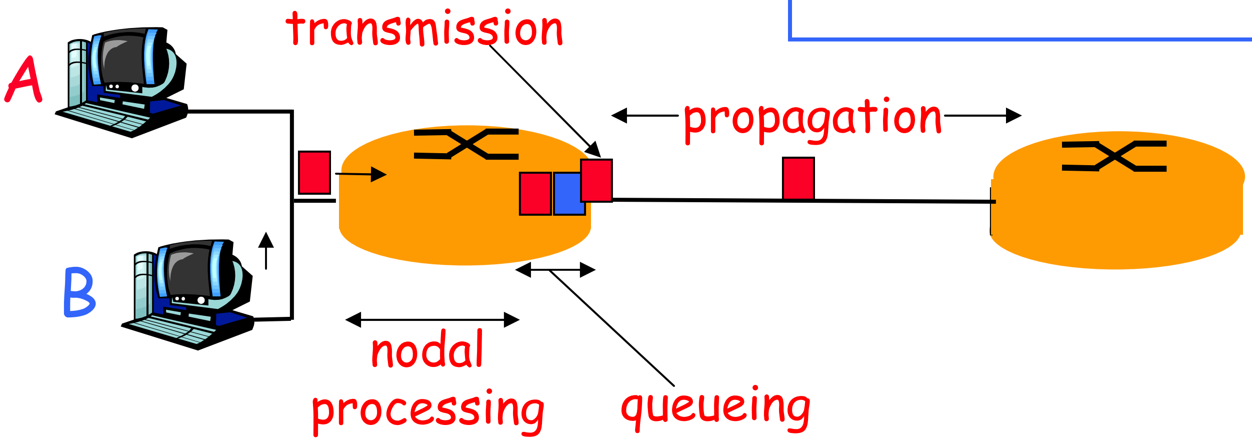
3. Transmission delay:

- R=link bandwidth (bps)
- L=packet length (bits)
- time to send bits into link = L/R

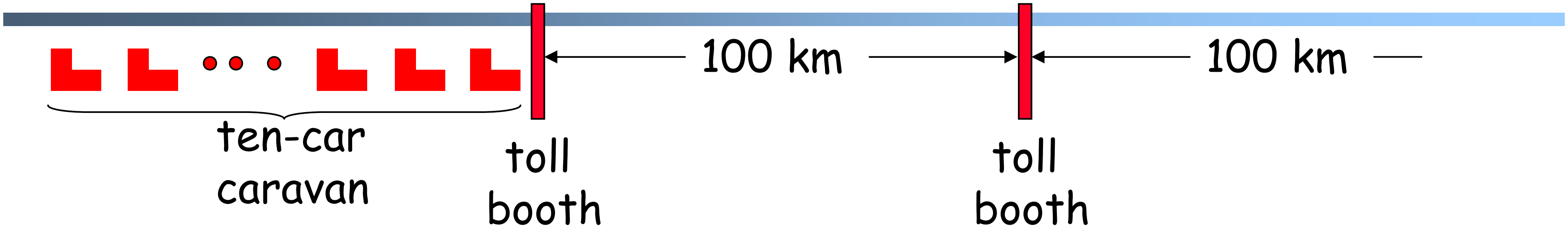
4. Propagation delay:

- d = length of physical link
- s = propagation speed in medium (~2x10^8 m/sec)
- propagation delay = d/s

Note: s and R are *very* different quantities!

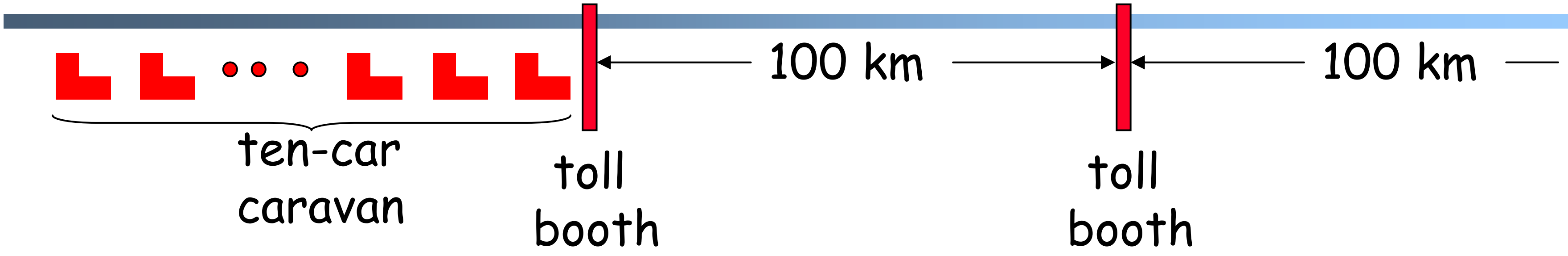


Caravan analogy



- Cars “propagate” at 100 km/hr
- Toll booth takes 12 sec to service a car (transmission time)
- car~bit; caravan ~ packet
- Q: How long until caravan is lined up before 2nd toll booth?
- Time to “push” entire caravan through toll booth onto highway = 12*10 = 120 sec
- Time for last car to propagate from 1st to 2nd toll both: 100km/(100km/hr)= 1 hr
- A: 62 minutes

Caravan analogy (more)



- Cars now “propagate” at 1000 km/hr
- Toll booth now takes 1 min to service a car
- Q: Will cars arrive to 2nd booth before all cars serviced at 1st booth?

- Yes! After 7 min, 1st car at 2nd booth and 3 cars still at 1st booth.
- 1st bit of packet can arrive at 2nd router before packet is fully transmitted at 1st router!

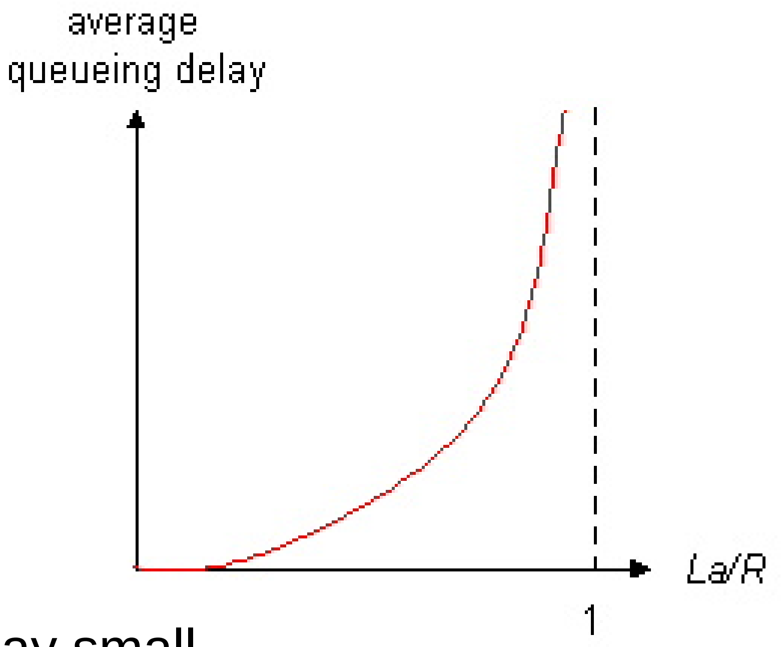
Nodal delay

$$\boxed{d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}}$$

- d_{proc} = processing delay
  - typically a few microsecs or less
- d_{queue} = queuing delay
  - depends on congestion
- d_{trans} = transmission delay
  - = L/R, significant for low-speed links
- d_{prop} = propagation delay
  - a few microsecs to hundreds of msecs

Queueing delay (revisited)

- R=link bandwidth (bps)
- L=packet length (bits)
- a=average packet arrival rate

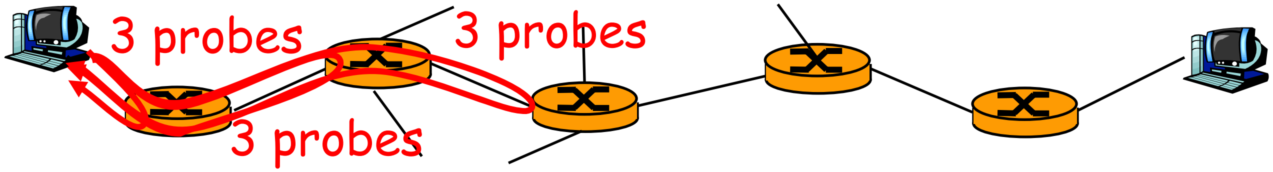


traffic intensity = La/R

- La/R ~ 0: average queueing delay small
- La/R -> 1: delays become large
- La/R > 1: more “work” arriving than can be serviced, average delay infinite!

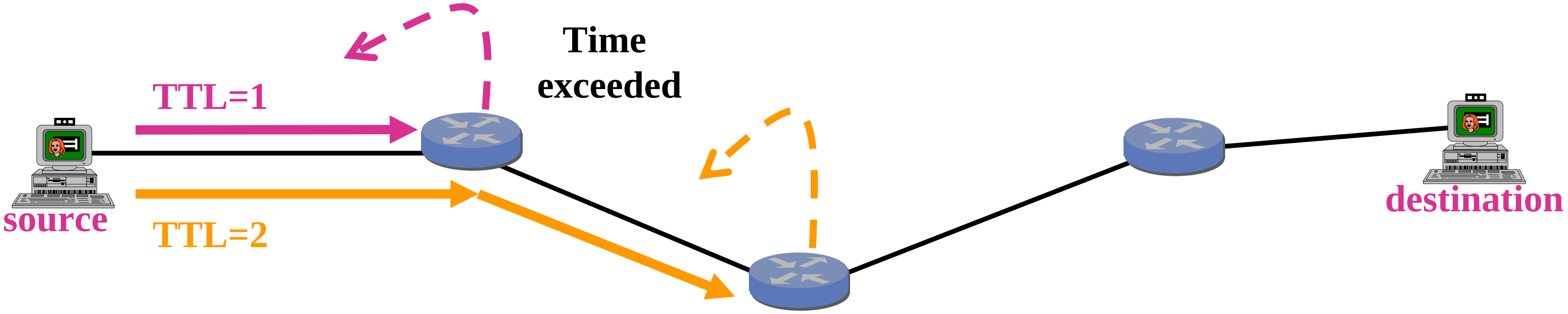
“Real” Internet delays and routes

- What do “real” Internet delay & loss look like?
- **Traceroute** program: provides delay measurement from source to router along end-end Internet path towards destination. For all *i:*
  - sends three packets that will reach router *i* on path towards destination
  - router *i* will return packets to sender
  - sender times interval between transmission and reply.



Traceroute: Measuring the Forwarding Path

- Time-To-Live field in IP packet header
    - Source sends a packet with a TTL of *n*
    - Each router along the path decrements the TTL
    - “TTL exceeded” sent when TTL reaches *0*
- Traceroute tool exploits this TTL behavior



Send packets with TTL=1, 2, 3, … and record source of “time exceeded” message

“Real” Internet delays and routes

traceroute: gaia.cs.umass.edu to www.eurecom.fr

Three delay measements from gaia.cs.umass.edu to cs-gw.cs.umass.edu

```
1  cs-gw (128.119.240.254)  1 ms  1 ms  2 ms
2  border1-rt-fa5-1-0.gw.umass.edu (128.119.3.145)  1 ms  1 ms  2 ms
3  cht-vbns.gw.umass.edu (128.119.3.130)  6 ms 5 ms 5 ms
4  jn1-at1-0-0-19.wor.vbns.net (204.147.132.129)  16 ms 11 ms 13 ms
5  jn1-so7-0-0-0.wae.vbns.net (204.147.136.136)  21 ms 18 ms 18 ms
6  abilene-vbns.abilene.ucaid.edu (198.32.11.9)  22 ms  18 ms  22 ms
7  nycm-wash.abilene.ucaid.edu (198.32.8.46)  22 ms  22 ms  22 ms
8  62.40.103.253 (62.40.103.253)  104 ms 109 ms 106 ms
9  de2-1.de1.de.geant.net (62.40.96.129)  109 ms 102 ms 104 ms
10  de.fr1.fr.geant.net (62.40.96.50)  113 ms 121 ms 114 ms
11  renater-gw.fr1.fr.geant.net (62.40.103.54)  112 ms  114 ms  112 ms
12  nio-n2.cssi.renater.fr (193.51.206.13)  111 ms  114 ms  116 ms
13  nice.cssi.renater.fr (195.220.98.102)  123 ms  125 ms  124 ms
14  r3t2-nice.cssi.renater.fr (195.220.98.110)  126 ms  126 ms  124 ms
15  eurecom-valbonne.r3t2.ft.net (193.48.50.54)  135 ms  128 ms  133 ms
16  194.214.211.25 (194.214.211.25)  126 ms  128 ms  126 ms
17  * * *
18  * * *
19  fantasia.eurecom.fr (193.55.113.142)  132 ms  128 ms  136 ms
```

trans-oceanic link

* means no reponse (probe lost, router not replying)

Packet loss

- queue (aka buffer) preceding link in buffer has finite capacity
- when packet arrives to full queue, packet is dropped (aka lost)
- lost packet may be retransmitted by previous node, by source end system, or not retransmitted at all

Protocol “Layers”

Networks are complex!

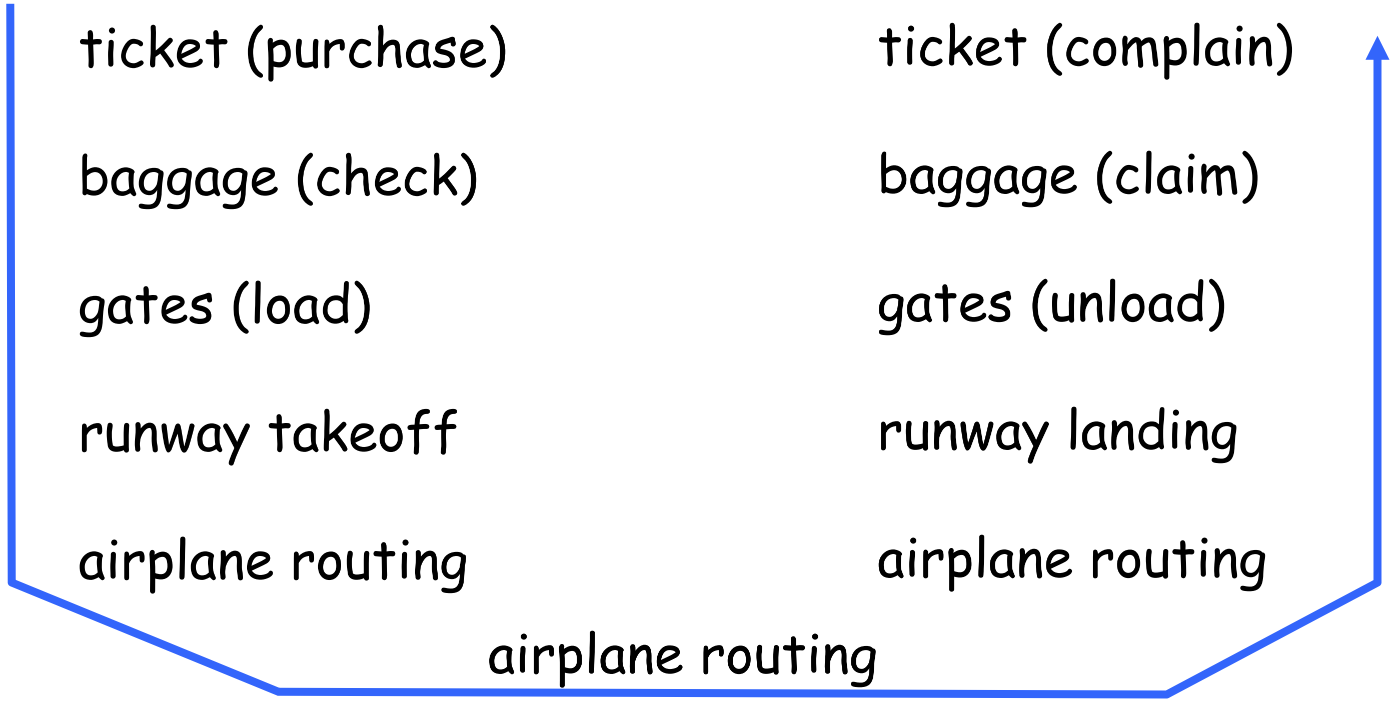
- many “pieces”:
  - hosts
  - routers
  - links of various media
  - applications
  - protocols
  - hardware, software

Question:

Is there any hope of *organizing* structure of network?

Or at least our discussion of networks?

Organization of air travel

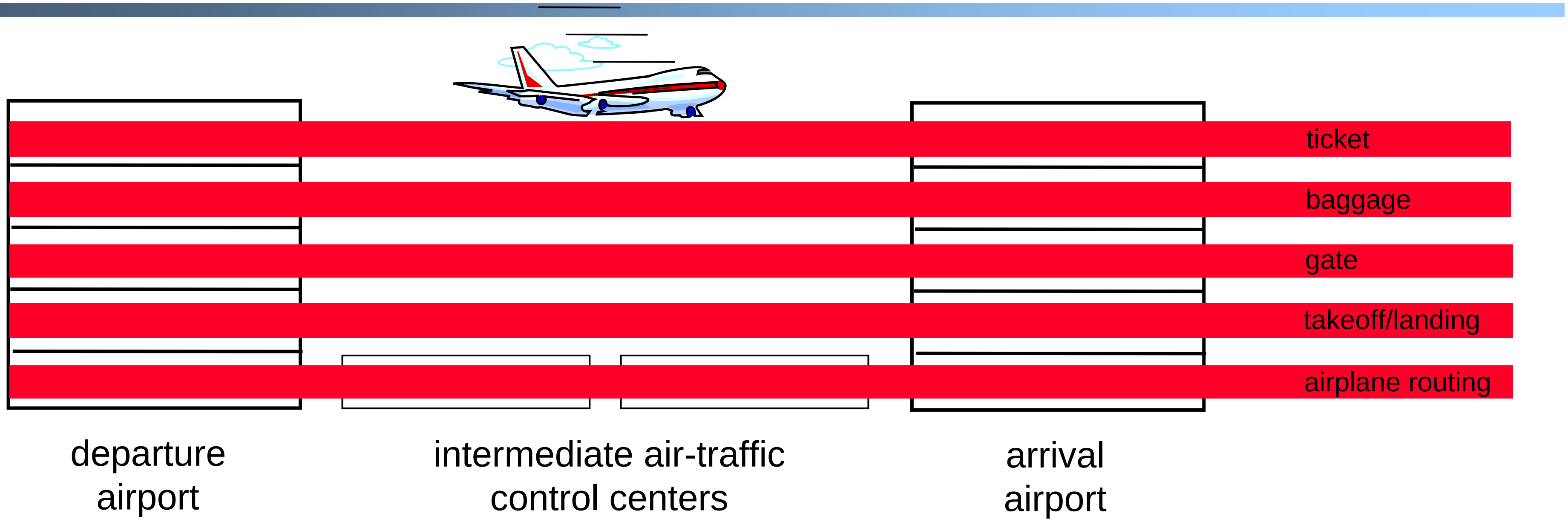


ticket (purchase)	ticket (complain)
baggage (check)	baggage (claim)
gates (load)	gates (unload)
runway takeoff	runway landing
airplane routing	airplane routing

airplane routing

- a series of steps

Layering of airline functionality



Layers: each layer implements a service

- via its own internal-layer actions
- relying on services provided by layer below

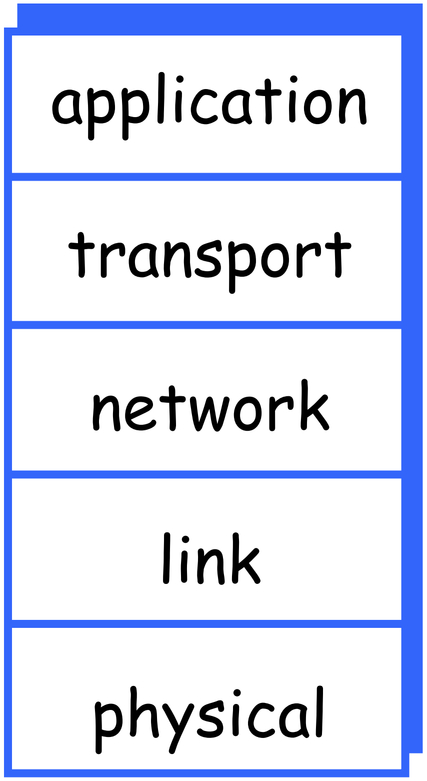
Why layering?

Dealing with complex systems:

- explicit structure allows identification, relationship of complex system's pieces
  - layered reference model for discussion
- modularization eases maintenance, updating of system
  - change of implementation of layer's service transparent to rest of system
  - e.g., change in gate procedure doesn't affect rest of system
- layering considered harmful?

Internet protocol stack

- application: supporting network applications
  - FTP, SMTP, STTP
- transport: host-host data transfer
  - TCP, UDP
- network: routing of datagrams from source to destination
  - IP, routing protocols
- link: data transfer between neighboring network elements
  - PPP, Ethernet
- physical: bits "on the wire"



application

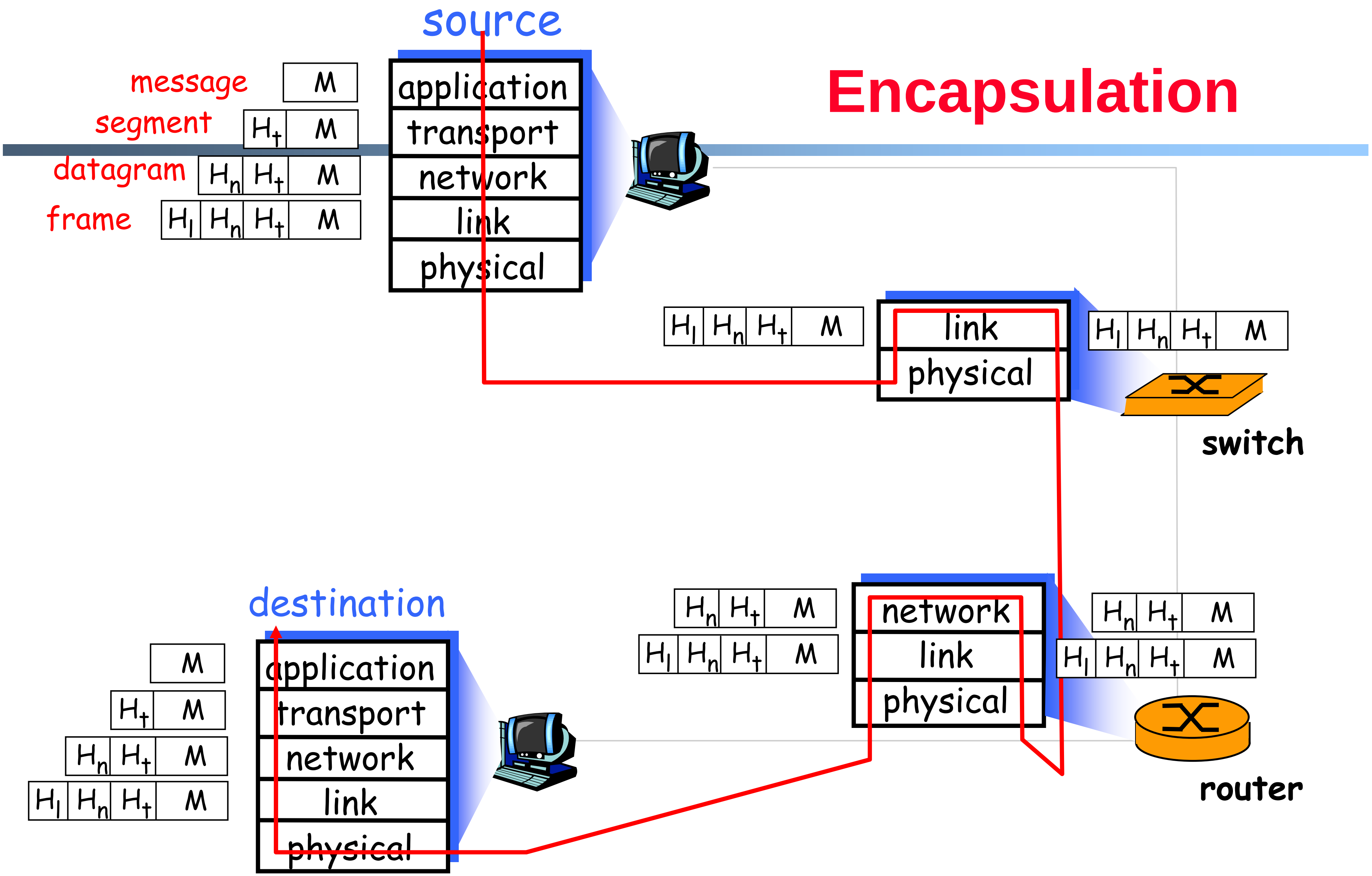
transport

network

link

physical

Encapsulation



source

message M

segment H_t M

datagram H_n H_t M

frame H_l H_n H_t M

application

transport

network

link

physical

H_l H_n H_t M

link

physical

H_l H_n H_t M

switch

H_n H_t M

H_l H_n H_t M

network

link

physical

H_n H_t M

H_l H_n H_t M

router

destination

M

H_t M

H_n H_t M

H_l H_n H_t M

application

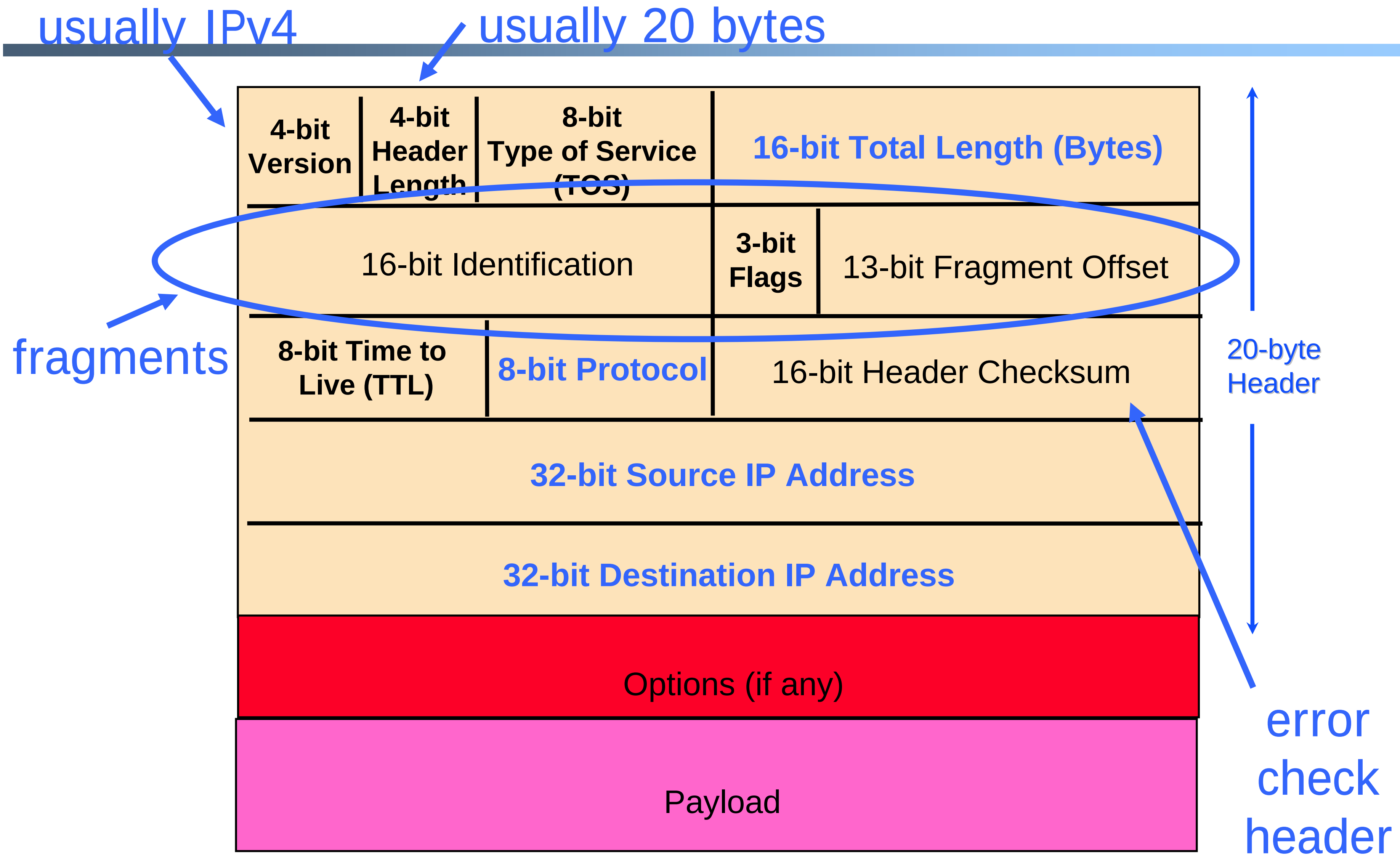
transport

network

link

physical

IP Packet Structure



4-bit Version

4-bit Header Length

8-bit Type of Service (TOS)

16-bit Total Length (Bytes)

16-bit Identification

3-bit Flags

13-bit Fragment Offset

8-bit Time to Live (TTL)

8-bit Protocol

16-bit Header Checksum

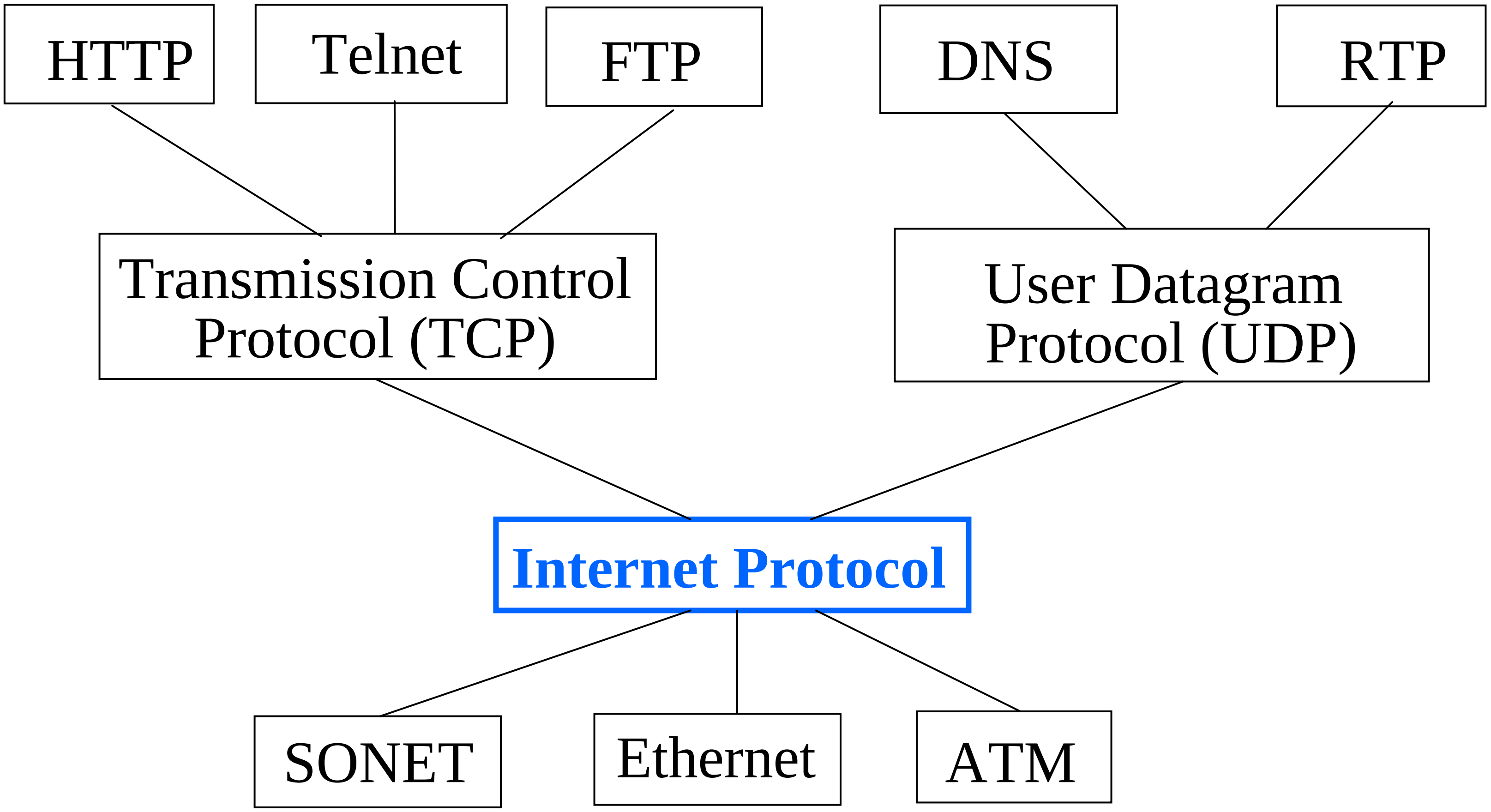
32-bit Source IP Address

32-bit Destination IP Address

Options (if any)

Payload

Layering in the IP Protocols



HTTP

Telnet

FTP

DNS

RTP

Transmission Control Protocol (TCP)

User Datagram Protocol (UDP)

Internet Protocol

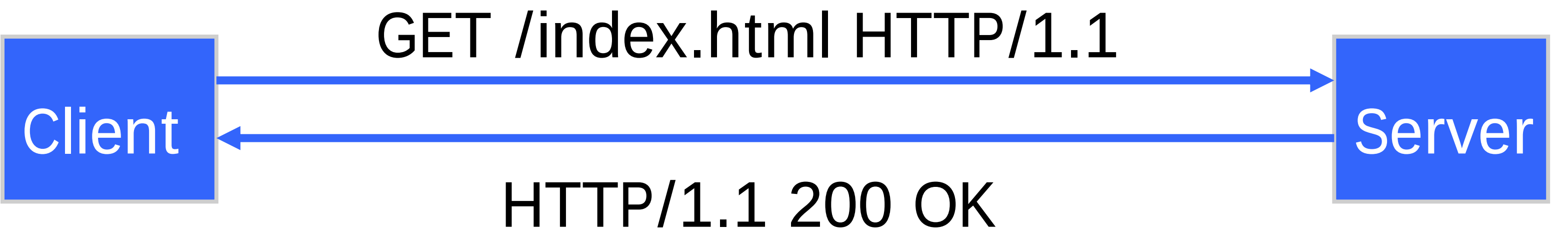
SONET

Ethernet

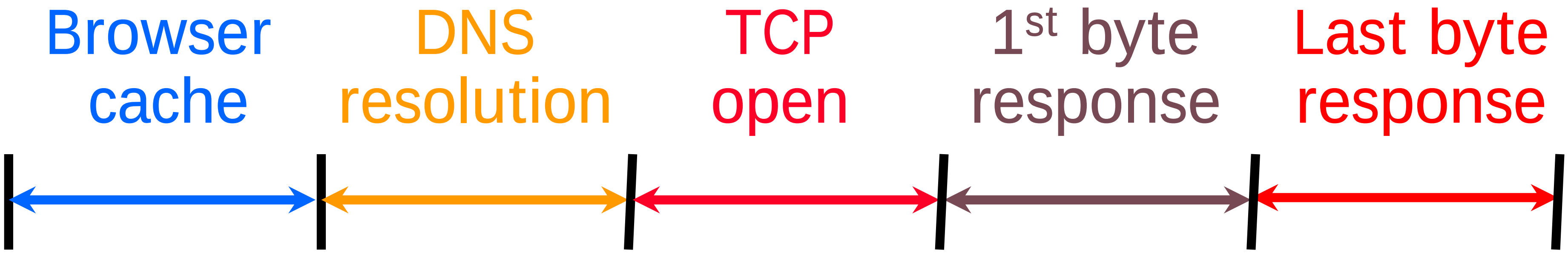
ATM

Application-Layer Protocols

- Messages exchanged between applications
  - Syntax and semantics of the messages between hosts
  - Tailored to the specific application (e.g., Web, e-mail)
  - Messages transferred over transport connection (e.g., TCP)
- Popular application-layer protocols
  - Telnet, FTP, SMTP, NNTP, HTTP, …



Example: Many Steps in Web Download



Browser cache

DNS resolution

TCP open

1st byte response

Last byte response

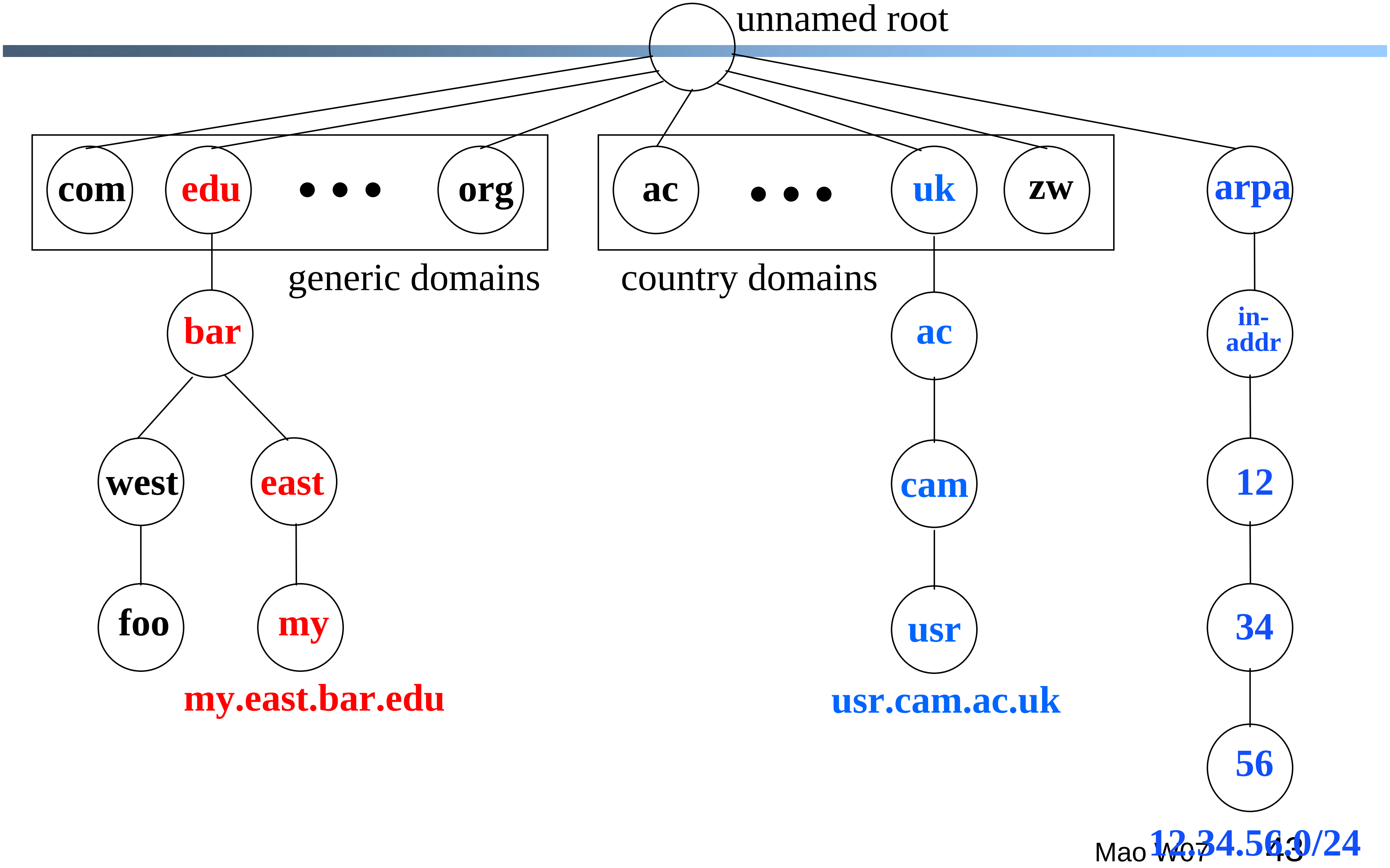
Sources of variability of delay

- Browser cache hit/miss, need for cache revalidation
- DNS cache hit/miss, multiple DNS servers, errors
- Packet loss, high RTT, server accept queue
- RTT, busy server, CPU overhead (e.g., CGI script)
- Response size, receive buffer size, congestion
- ... downloading embedded image(s) on the page

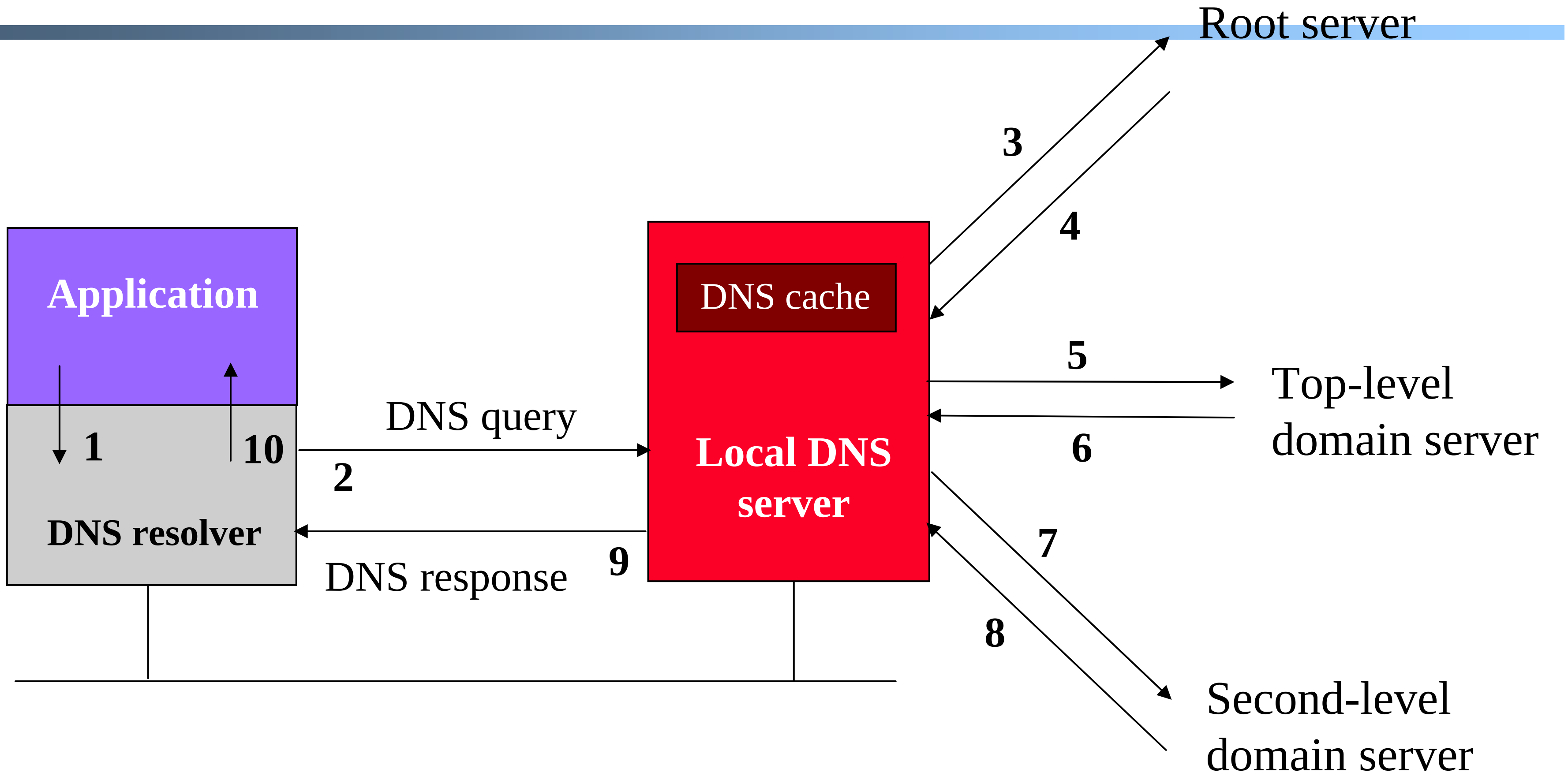
Domain Name System (DNS)

- Properties of DNS
  - Hierarchical name space divided into zones
  - Translation of names to/from IP addresses
  - Distributed over a collection of DNS servers
- Client application
  - Extract server name (e.g., from the URL)
  - Invoke system call to trigger DNS resolver code
  - E.g., *gethostbyname()* on “www.foo.com”
- Server application
  - Extract client IP address from socket
  - Optionally invoke system call to translate into name
  - E.g., *gethostbyaddr()* on “12.34.158.5”

Domain Name System



DNS Resolver and Local DNS Server



Caching based on a time-to-live (TTL) assigned by the DNS server responsible for the host name to reduce latency in DNS translation.

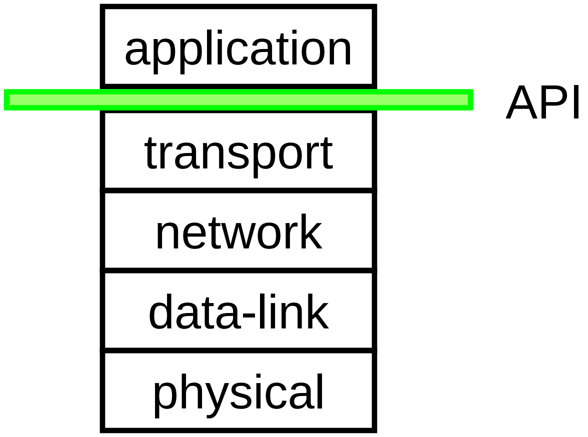
Sockets Programming

Outline

- Socket API motivation, background
- Names, addresses, presentation
- API functions
- I/O multiplexing

Motivation

- Applications need Application Programming Interface (API) to use the network

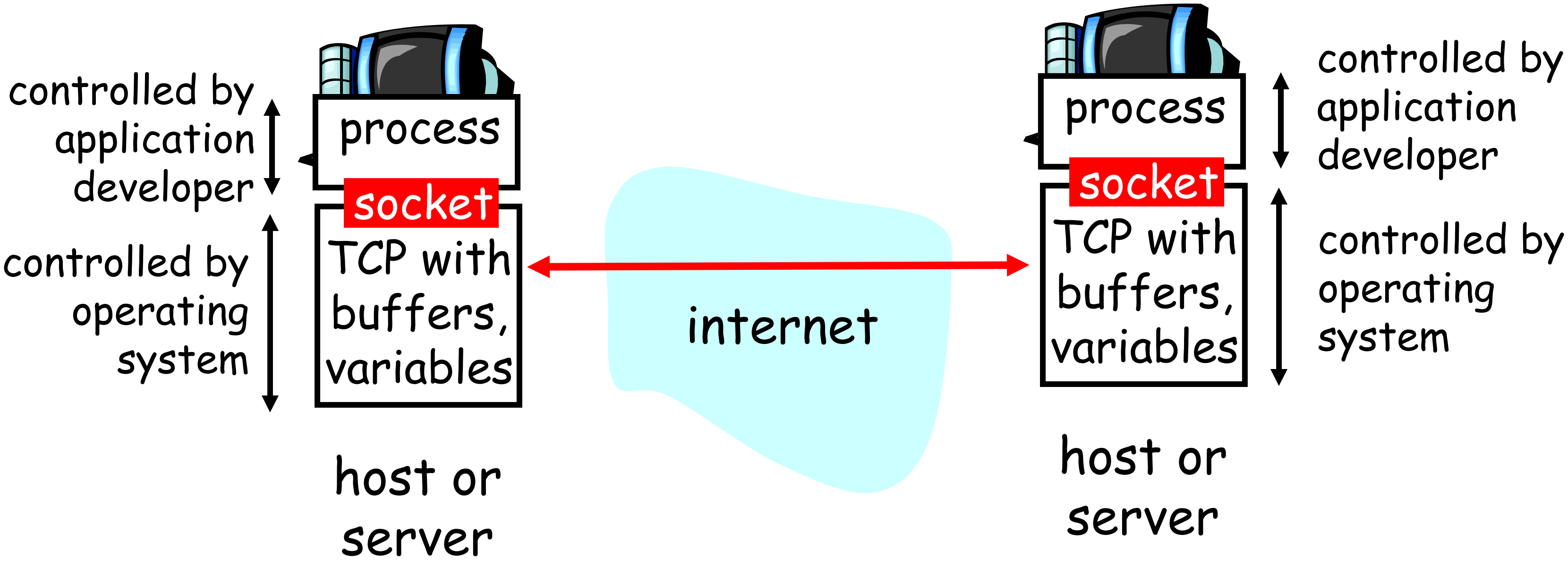


- API: set of function types, data structures and constants
  - Allows programmer to learn once, write anywhere
  - Greatly simplifies job of application programmer

Socket-programming using TCP

Socket: a door between application process and end-end-transport protocol (UCP or TCP)

TCP service: reliable transfer of **bytes** from one process to another


controlled by application developer
controlled by operating system
process
socket
TCP with buffers, variables
host or server
internet
process
socket
TCP with buffers, variables
host or server
controlled by application developer
controlled by operating system

Socket programming *with TCP*

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

- creating client-local TCP socket
- specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP

- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Sockets (1)

- Useful sample code available at
  - http://www.kohala.com/start/unpv22e/unpv22e.html
- What exactly are sockets?
  - An endpoint of a connection
  - A socket is associated with each end-point (end-host) of a connection
- Identified by IP address and port number
- Berkeley sockets is the most popular network API
  - Runs on Linux, FreeBSD, OS X, Windows
  - Fed/fed off popularity of TCP/IP

Sockets (2)

- Similar to UNIX file I/O API (provides file descriptor)

- Based on C, single threaded model
  - Does not require multiple threads

- Can build higher-level interfaces on top of sockets
  - e.g., Remote Procedure Call (RPC)

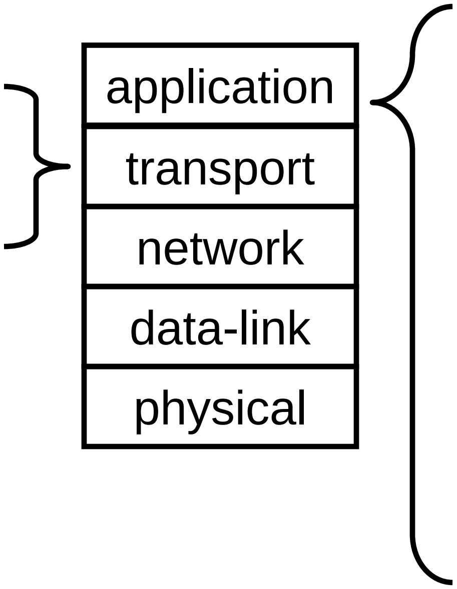
Types of Sockets (1)

- Different types of sockets implement different service models
  - Stream v.s. datagram
- Stream socket (aka TCP)
  - Connection-oriented (includes establishment + termination)
  - Reliable, in order delivery
  - At-most-once delivery, no duplicates
  - E.g., ssh, http
- Datagram socket (aka UDP)
  - Connectionless (just data-transfer)
  - “Best-effort” delivery, possibly lower variance in delay
  - E.g., IP Telephony, streaming audio

Types of Sockets (2)

- How does application programming differ between stream and datagram sockets?
- Stream sockets
  - No need to packetize data
  - Data arrives in the form of a byte-stream
  - Receiver needs to separate messages in stream

TCP sends messages joined together, ie. “Hi there!Hope you are well”



application

transport

network

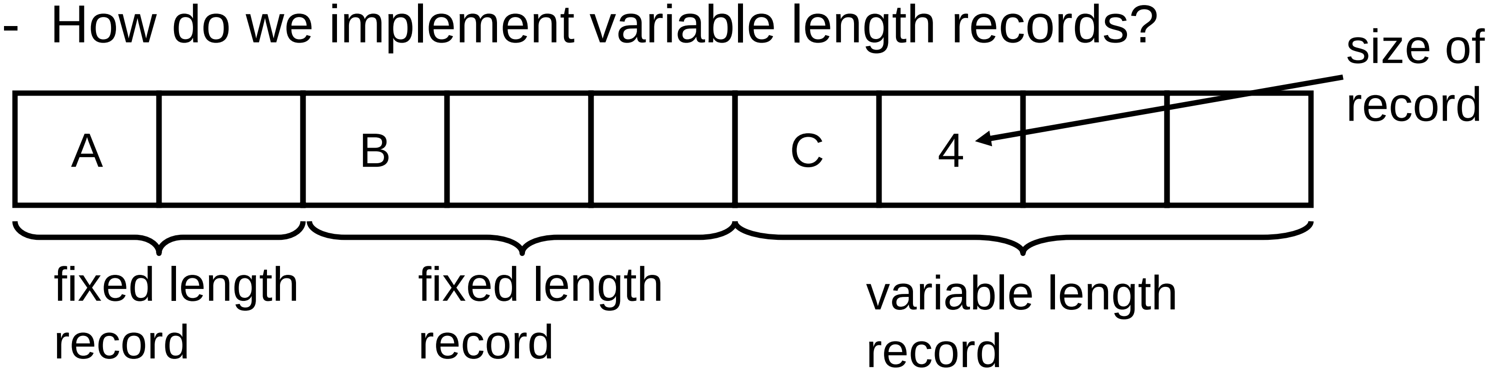
data-link

physical

User application sends messages “Hi there!” and “Hope you are well” separately

Types of Sockets (3)

- Stream socket data separation:
  - Use records (data structures) to partition data stream
  - How do we implement variable length records?



  - What if field containing record size gets corrupted?
    - Not possible! Why?

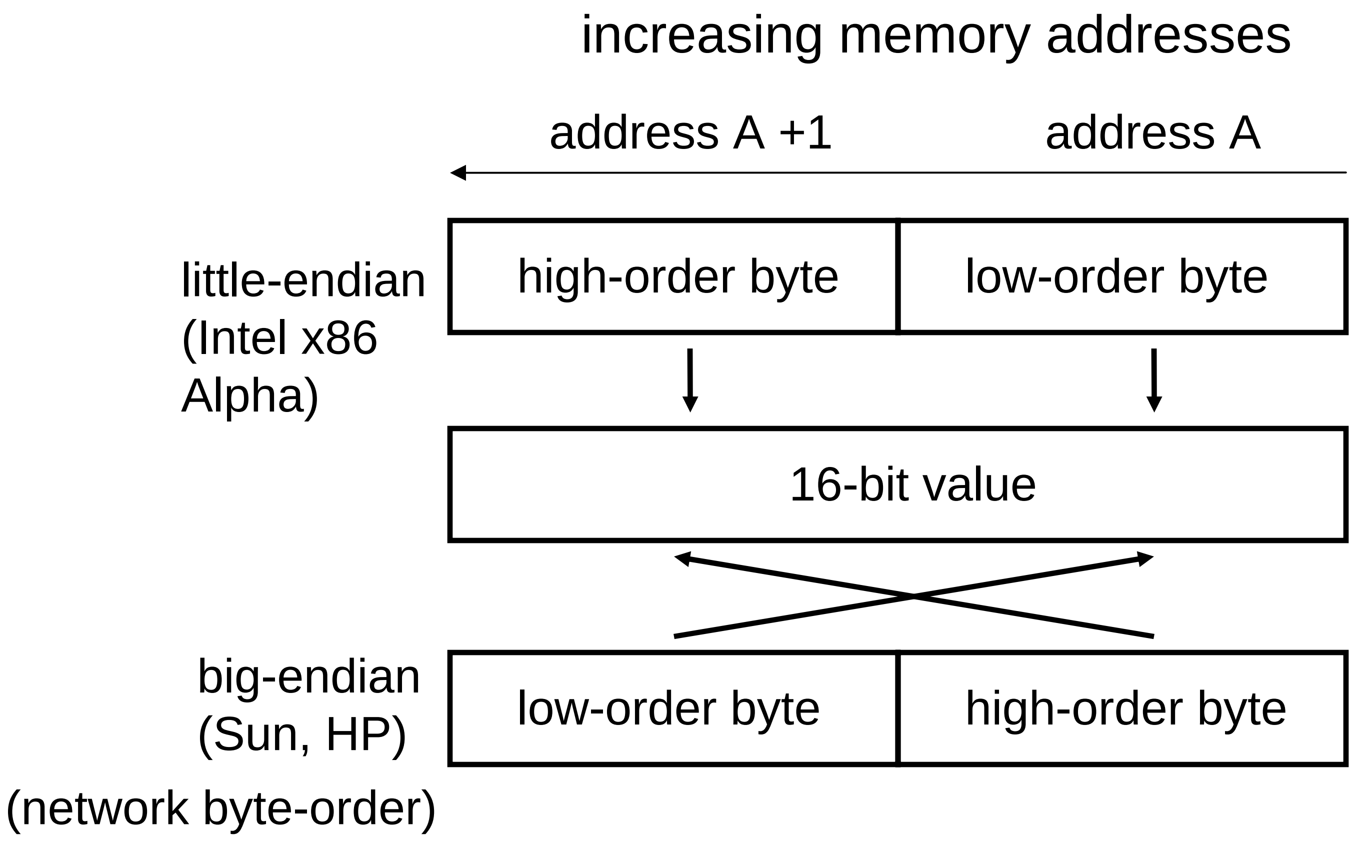
Types of Sockets (4)

- Datagram sockets
  - User packetizes data before sending
  - Maximum size of 64Kbytes
  - Further packetization at sender end and depacketization at receiver end handled by transport layer
  - Using previous example, “Hi there!” and “Hope you are well” will definitely be sent in separate packets at network layer

Naming and Addressing

- IP version 4 address
  - Identifies a single host
  - 32 bits
  - Written as dotted octets
    - e.g., 0x0a000001 is 10.0.0.1
- Host name
  - Identifies a single host
  - Variable length string
  - Maps to one or more IP address
    - e.g., www.cnn.com
  - Gethostbyname translates name to IP address
- Port number
  - Identifies an application on a host
  - 16 bit unsigned number

Presentation



increasing memory addresses

address A +1

address A

little-endian (Intel x86 Alpha)

high-order byte

low-order byte

16-bit value

big-endian (Sun, HP)

low-order byte

high-order byte

(network byte-order)

Always translate short, long, int to/from “network byte order” before/after transmission: htons(), htonl(), ntohs(), ntohl()

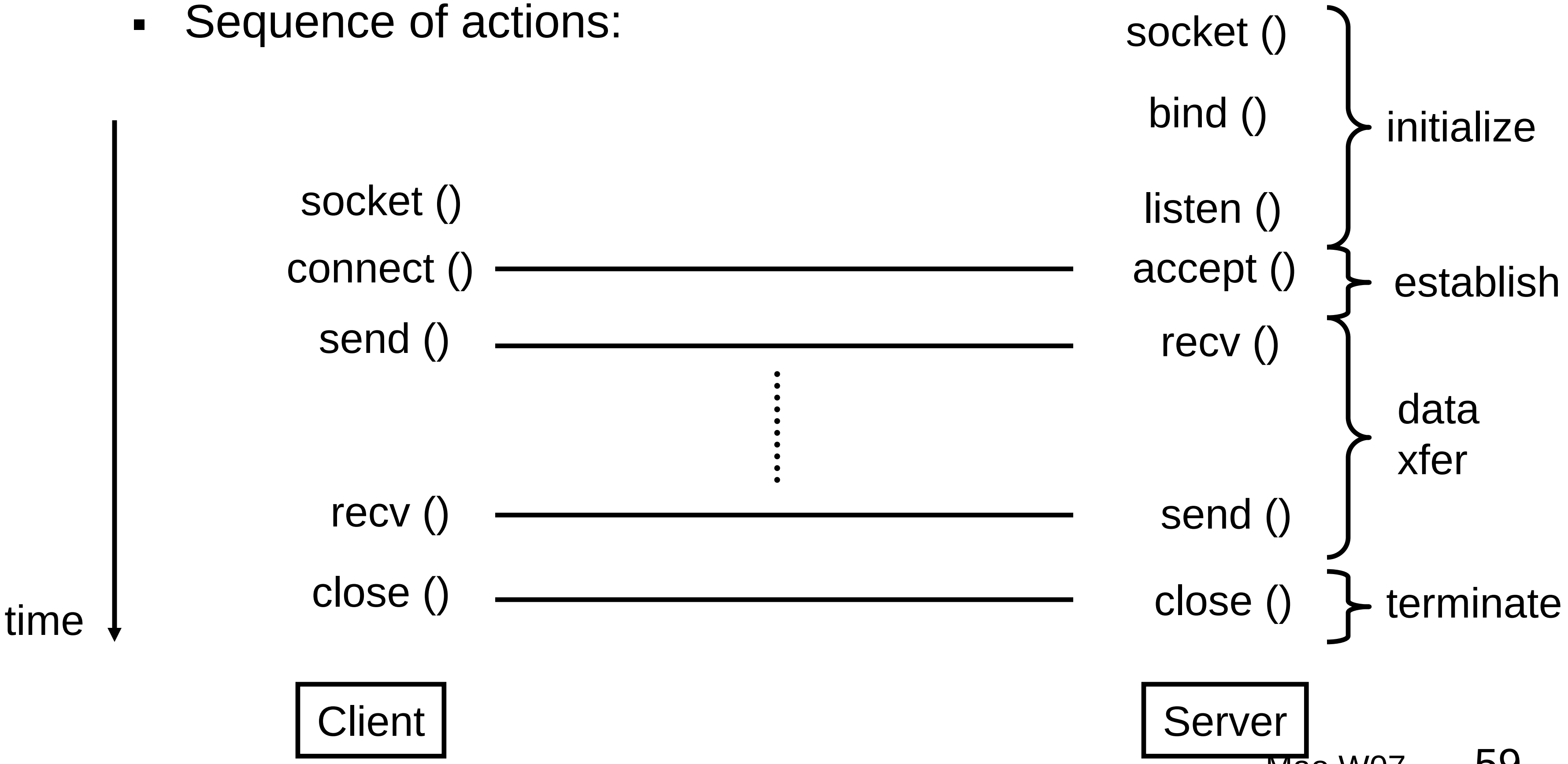
Byte Ordering Solution

```
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);
```

- Use for all numbers (int, short) to be sent across network
  - Including port numbers, but not IP addresses

Stream Sockets

- Implements Transmission Control Protocol (TCP)
- Does NOT set up virtual-circuit!
- Sequence of actions:



Initialize (Client + Server)

```
int sock;
if ((sock = socket(AF_INET, SOCK_STREAM,
                         IPPROTO_TCP)) < 0) {
    perror("socket");
    printf("Failed to create socket\n");
    abort ();
}
```

- Handling errors that occur rarely usually consumes most of systems code
  - Exceptions (e.g., in java) helps this somewhat

Initialize (Server reuse addr)

- After TCP connection closes, waits for 2MSL, which is twice maximum segment lifetime (from 1 to 4 mins)
- Segment refers to maximum size of a packet
- Port number cannot be reused before 2MSL
- But server port numbers are fixed ⇒ must be reused
- Solution:

```
int optval = 1;
if ((sock = socket (AF_INET, SOCK_STREAM, 0)) < 0)
   {
     perror ("opening TCP socket");
     abort ();
   }
 if (setsockopt (sock, SOL_SOCKET, SO_REUSEADDR,
 &optval,
        sizeof (optval)) <0)
   {
     perror (“reuse address");
     abort ();
   }
```

Initialize (Server bind addr)

- Want port at server end to use a particular number

```
struct sockaddr_in sin;

memset (&sin, 0, sizeof (sin));

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = IN_ADDR;
sin.sin_port = htons (server_port);

if (bind(sock, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
    perror(“bind");
    printf("Cannot bind socket to address\n");
    abort();
}
```

Initialize (Server listen)

- Wait for incoming connection
- Parameter BACKLOG specifies max number of established connections waiting to be accepted (using `accept()`)

```
if (listen (sock, BACKLOG) < 0)
   {
     perror (“error listening");
     abort ();
   }
```

Establish (Client)

```
struct sockaddr_in sin;

struct hostent *host = gethostbyname (argv[1]);
unsigned int server_addr = *(unsigned long *) host->h_addr_list[0];
unsigned short server_port = atoi (argv[2]);

memset (&sin, 0, sizeof (sin));

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = server_addr;
sin.sin_port = htons (server_port);

if (connect(sock, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
    perror("connect");
    printf("Cannot connect to server\n");
    abort();
}
```

Establish (Server)

- Accept incoming connection

```
int addr_len = sizeof (addr);
int sock;

sock = accept (tcp_sock, (struct sockaddr *)
               &addr, &addr_len);

if (sock < 0)
  {
    perror ("error accepting connection");
    abort ();
  }
```

Sending Data Stream

```
int send_packets (char *buffer, int buffer_len)
{
  sent_bytes = send (sock, buffer, buffer_len, 0);

  if (send_bytes < 0)
       perror (“send”);

  return 0;
}
```

Receiving Data Stream

```
int receive_packets(char *buffer, int buffer_len, int *bytes_read){
    int left = buffer_len - *bytes_read;
    received = recv(sock, buffer + *bytes_read, left, 0);
    if (received < 0) {
        perror ("Read in read_client");
        printf("recv in %s\n", __FUNCTION__);
    }
    if (received == 0) { /* occurs when other side closes
  connection */
        return close_connection();
    }
    *bytes_read += received;
    while (*bytes_read > RECORD_LEN) {
        process_packet(buffer, RECORD_LEN);
        *bytes_read -= RECORD_LEN;
        memmove(buffer, buffer + RECORD_LEN, *bytes_read);
    }
    return 0;
}
```

Datagram Sockets

- Similar to stream sockets, except:
  - Sockets created using SOCK_DGRAM instead of SOCK_STREAM
  - No need for connection establishment and termination
  - Uses `recvfrom()` and `sendto()` in place of `recv()` and `send()` respectively
  - Data sent in packets, not byte-stream oriented

Socket programming *with UDP*

UDP: no “connection” between client and server

- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

UDP: transmitted data may be received out of order, or lost

application viewpoint

UDP provides unreliable transfer of groups of bytes (“datagrams”) between client and server

How to handle multiple connections?

- Where do we get incoming data?
  - Stdin (typically keyboard input)
  - All stream, datagram sockets
  - Asynchronous arrival, program doesn’t know when data will arrive
- Solution: I/O multiplexing using select ()
  - Coming up soon
- Solution: I/O multiplexing using polling
  - Very inefficient
- Solution: multithreading
  - More complex, requires mutex, semaphores, etc.
  - Not covered

I/O Multiplexing: Polling

```
int opts = fcntl (sock, F_GETFL);
if (opts < 0) {
    perror ("fcntl(F_GETFL)");
    abort ();
}
opts = (opts | O_NONBLOCK);
if (fcntl (sock, F_SETFL, opts) < 0) {
    perror ("fcntl(F_SETFL)");
    abort ();
}
while (1) {
    if (receive_packets(buffer, buffer_len, &bytes_read) != 0) {
        break;
    }
    if (read_user(user_buffer, user_buffer_len,
                  &user_bytes_read) != 0) {
        break;
    }
}
```

first get current socket option settings

then adjust settings

finally store settings back

get data from socket

get user input

I/O Multiplexing: Select (1)

- Select()
  - Wait on multiple file descriptors/sockets and timeout
  - Application does not consume CPU cycles while waiting
  - Return when file descriptors/sockets are ready to be read or written or they have an error, or timeout exceeded
- Advantages
  - Simple
  - More efficient than polling
- Disadvantages
  - Does not scale to large number of file descriptors/sockets
  - More awkward to use than it needs to be

I/O Multiplexing: Select (2)

```
fd_set read_set;
struct timeval time_out;
while (1) {
```

set up parameters for select()

```
    FD_ZERO (read_set);
    FD_SET (stdin, read_set); /* stdin is typically 0 */
    FD_SET (sock, read_set);
    time_out.tv_usec = 100000; time_out.tv_sec = 0;
```

run select()

```
    select_retval = select(MAX(stdin, sock) + 1, &read_set, NULL,
                           NULL, &time_out);
```

interpret result

```
    if (select_retval < 0) {
        perror ("select");
        abort ();
    }
    if (select_retval > 0) {
        if (FD_ISSET(sock, read_set)) {
            if (receive_packets(buffer, buffer_len, &bytes_read) != 0) {
                break;
            }
        if (FD_ISSET(stdin, read_set)) {
            if (read_user(user_buffer, user_buffer_len,
                          &user_bytes_read) != 0) {
                break;
            }
        }
    }
}
```

Common Mistakes + Hints

- Common mistakes:
  - C programming
    - Use gdb
    - Use printf for debugging, remember to do `fflush(stdout);`
  - Byte-ordering
  - Use of `select()`
  - Separating records in TCP stream
  - Not knowing what exactly gets transmitted on the wire
    - Use tcpdump / Ethereal
- Hints:
  - Use man pages (available on the web too)
  - Check out WWW, programming books