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?

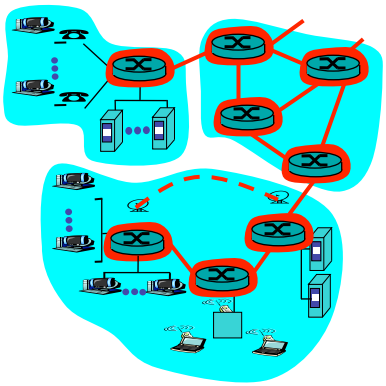
1

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"



2

Network Core: Circuit Switching

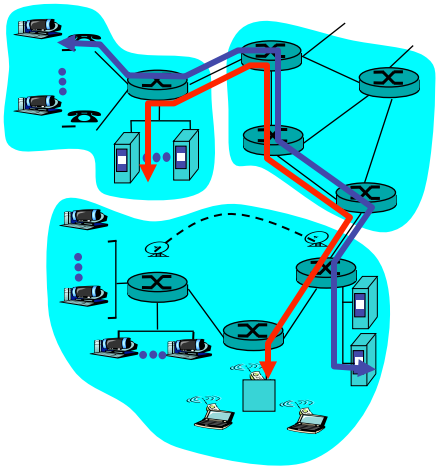
End-end resources reserved for "call"

link bandwidth, switch capacity

dedicated resources: no sharing

circuit-like (guaranteed) performance

call setup required



3

Network Core: Circuit Switching

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

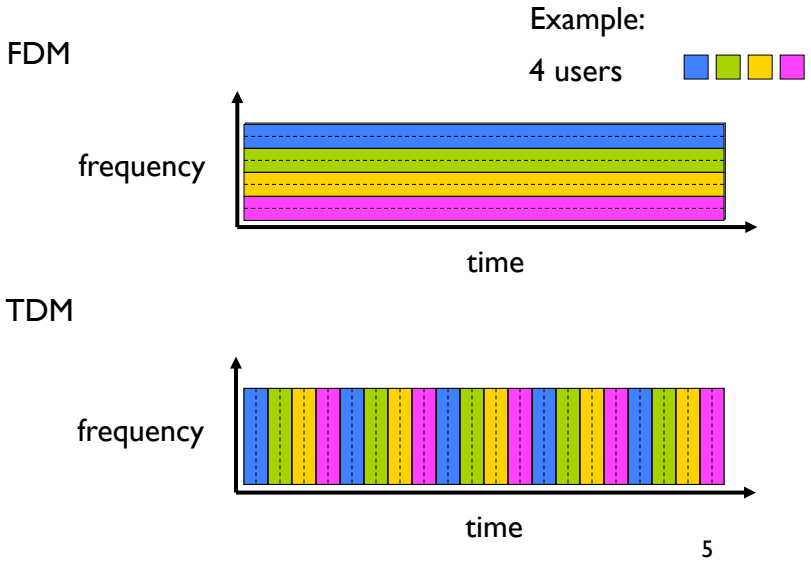
pieces allocated to calls

resource piece *idle* if not used by owning call *(no sharing)*

- dividing link bandwidth into "pieces"
  - frequency division
  - time division

4

Circuit Switching: FDM and TDM



5

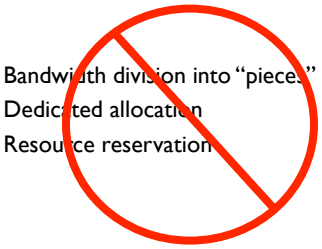
Network Core: Packet Switching

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*

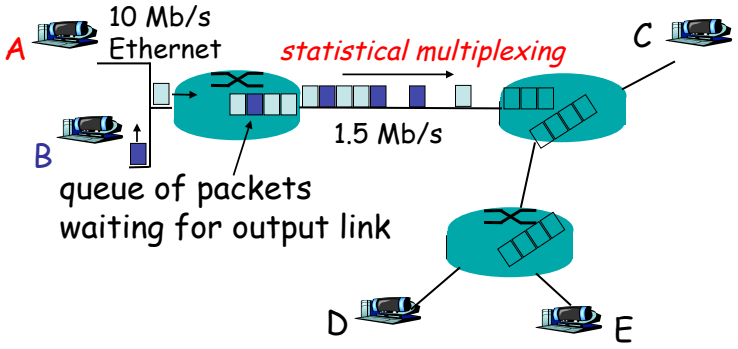


resource contention:

- 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

6

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.

7

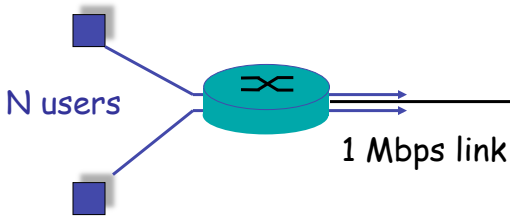
Packet switching versus circuit switching

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



8

Packet switching versus circuit switching

Is packet switching a "slam dunk winner?"

Great for bursty data
- resource sharing
- simpler, no call setup

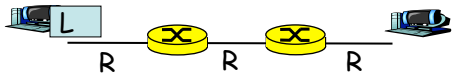
More resilient to failures

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

9

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

10

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 through call
- *routers maintain per-call state*

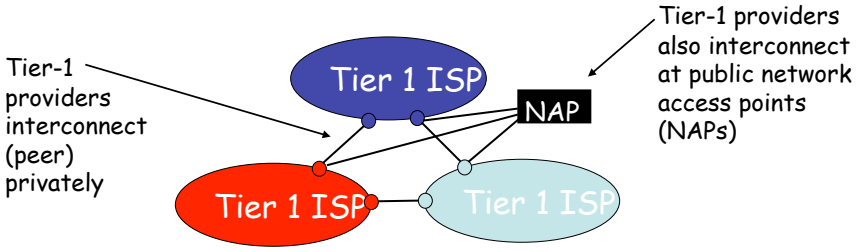
11

Internet structure: network of networks

roughly hierarchical

at center: "tier-1" ISPs (e.g., UUNet, BBN/Genuity, Sprint, AT&T), national/international coverage

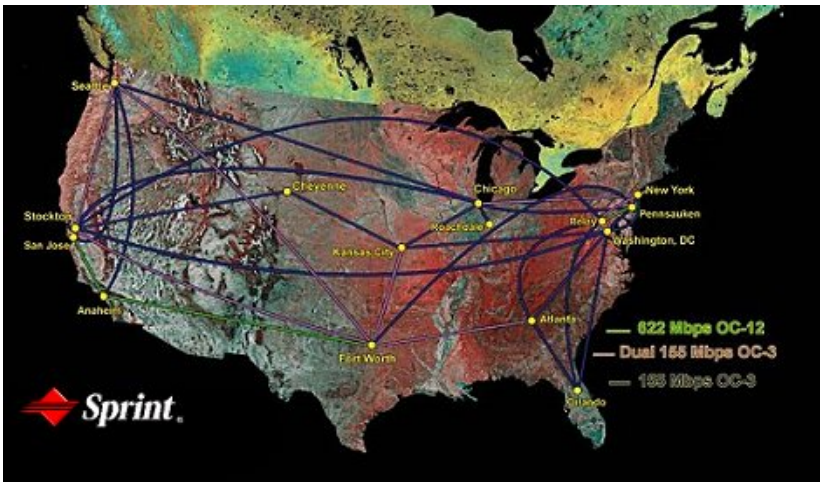
- treat each other as equals



12

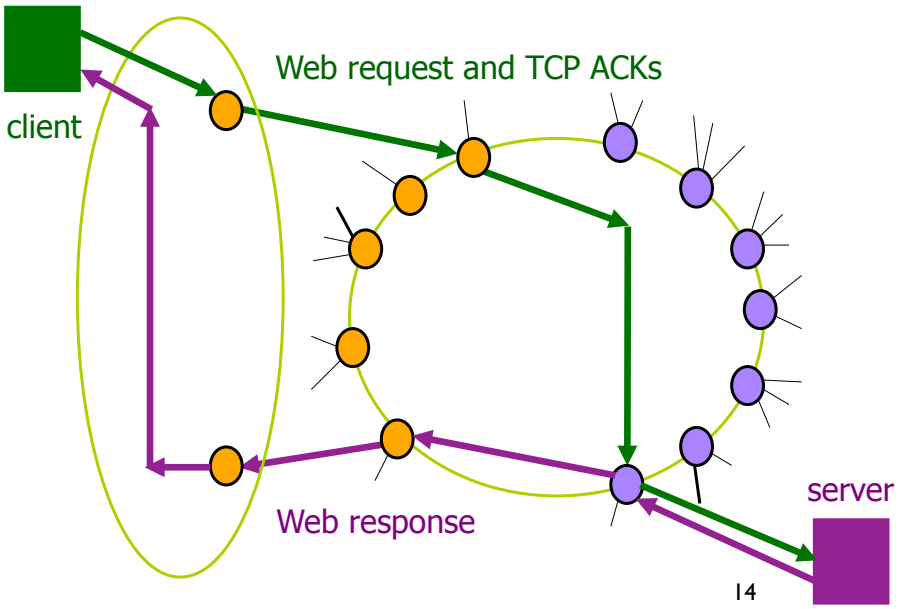
Tier-1 ISP: e.g., Sprint

Sprint US backbone network



13

Routing is Not Symmetric

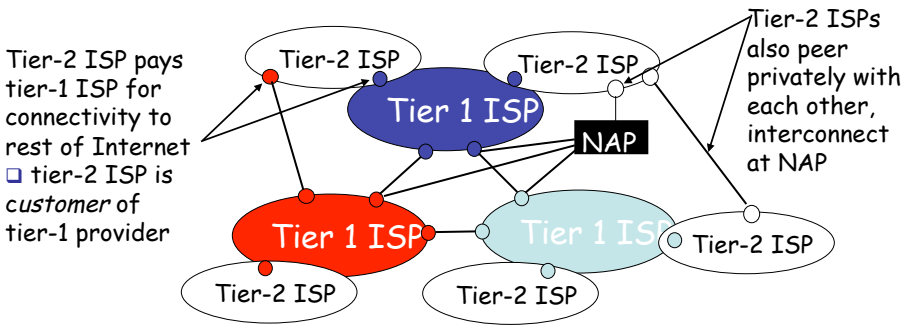


14

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

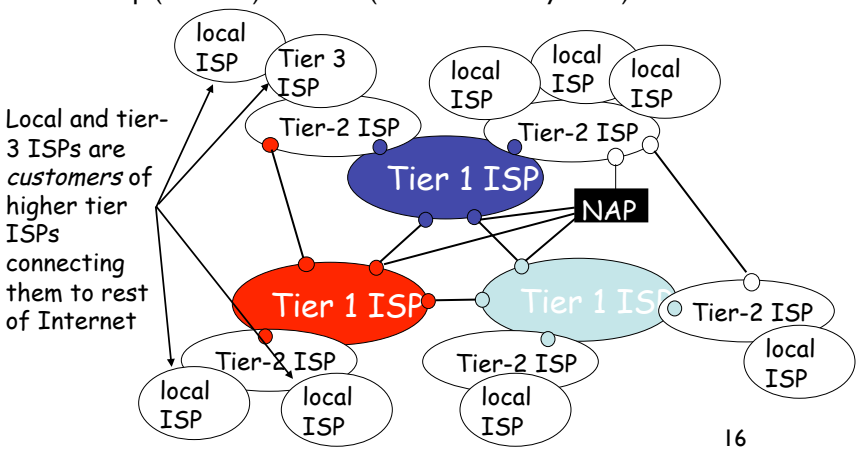


15

Internet structure: network of networks

"Tier-3" ISPs and local ISPs

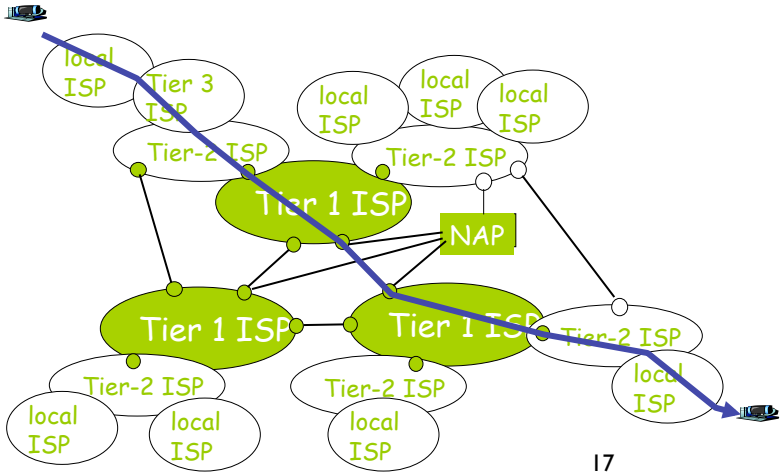
- last hop ("access") network (closest to end systems)



16

Internet structure: network of networks

a packet passes through many networks!



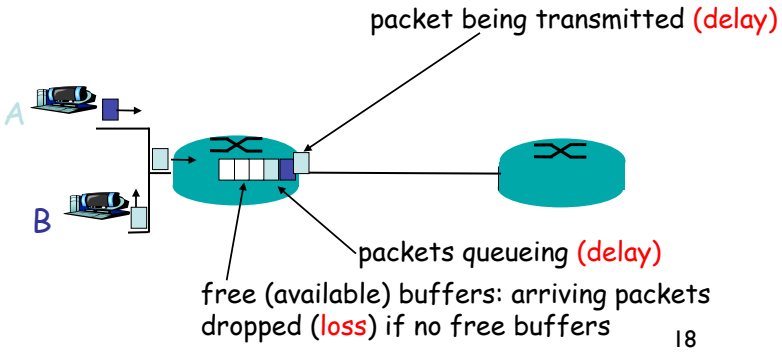
17

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

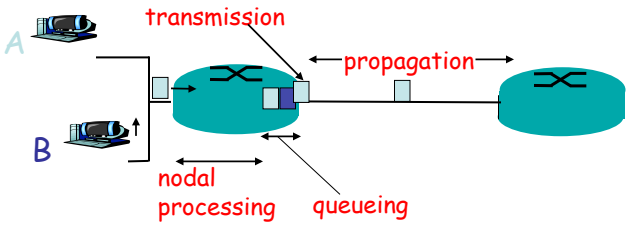


18

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



19

Delay in packet-switched networks

3. Transmission delay:

R=link bandwidth (bps)

L=packet length (bits)

time to send bits into link = L/R

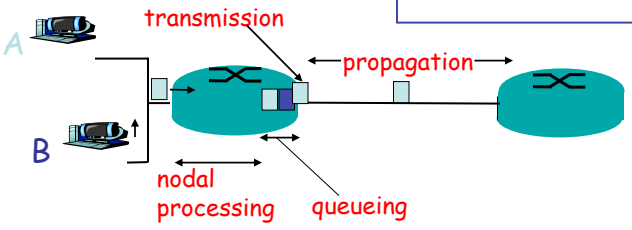
4. Propagation delay:

d = length of physical link

s = propagation speed in medium (~2×10^8 m/sec)

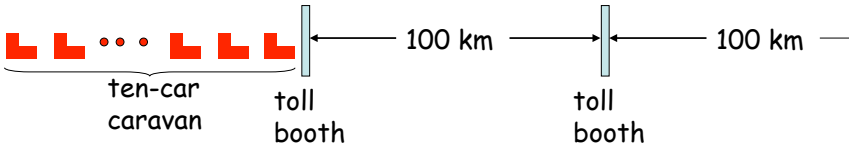
propagation delay = d/s

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



20

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?

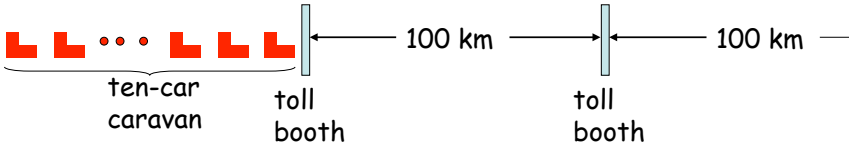
Time to "push" entire caravan through toll booth onto highway = 12*10 = 120 sec

Time for last car to propagate from 1st to 2nd toll both: 100km/(100km/hr)= 1 hr

A: 62 minutes

21

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!

22

Nodal delay

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

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

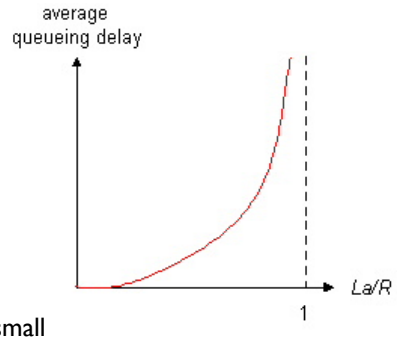
d_{prop} = propagation delay
- a few microsecs to hundreds of msecs

23

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!

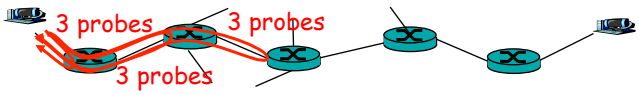
24

"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.



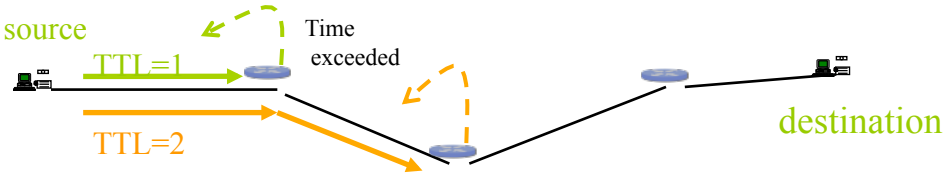
25

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

26

"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)

27

Packet loss

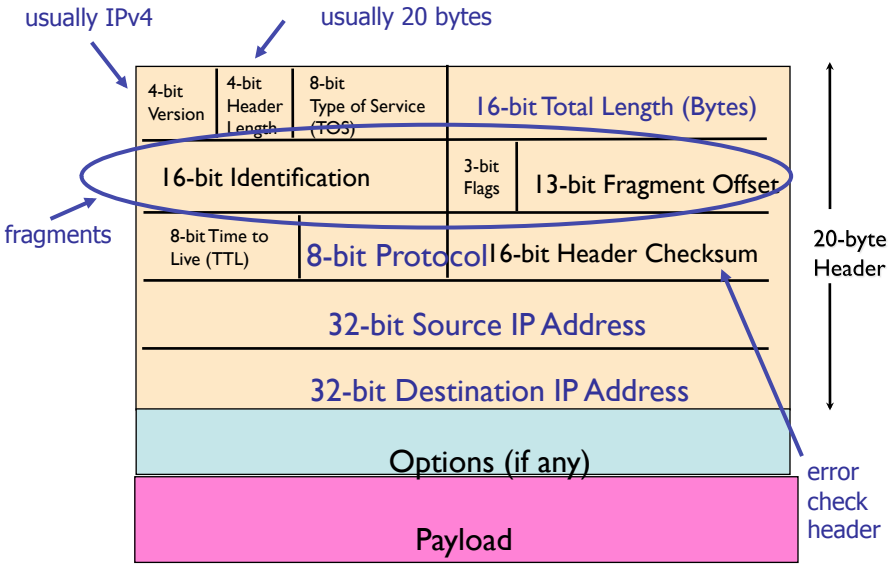
queue (aka buffer) preceding link in buffer has finite capacity

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

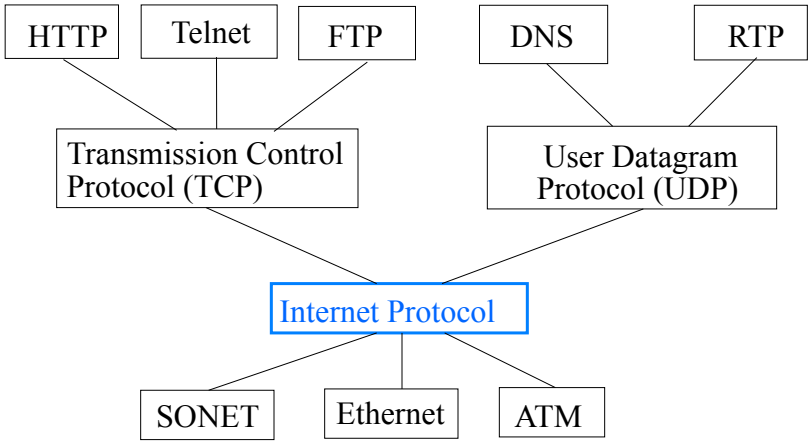
28

IP Packet Structure



29

Layering in the IP Protocols



30

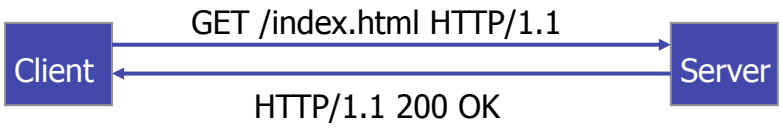
Application-Layer Protocols

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)

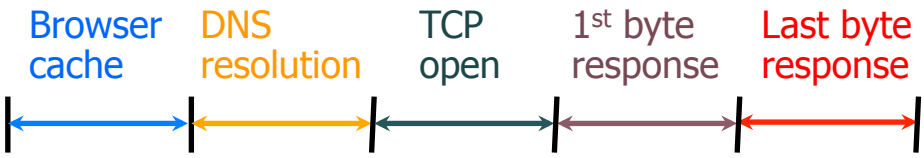
Popular application-layer protocols

- Telnet, FTP, SMTP, NNTP, HTTP, …



31

Example: Many Steps in Web Download



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

32

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

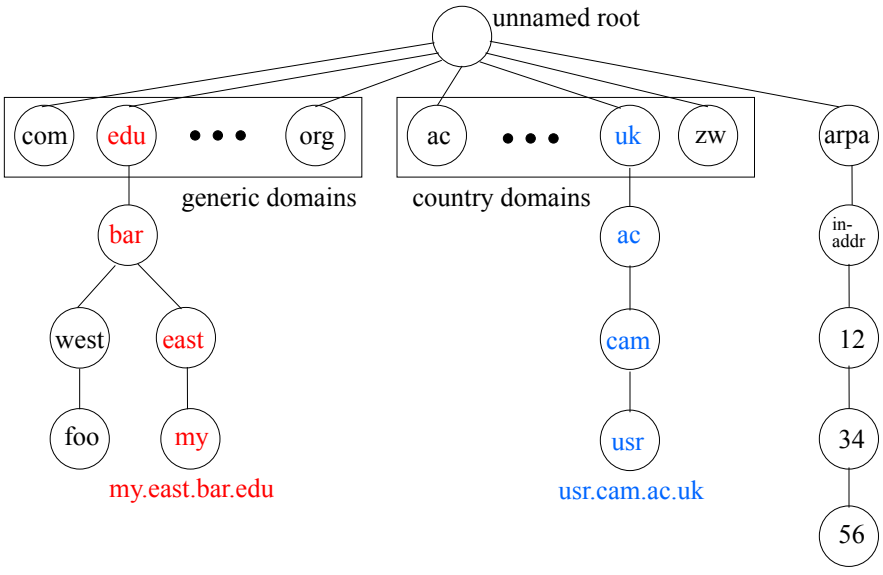
- Extract server name (e.g., from the URL)
- Invoke system call to trigger DNS resolver code
- E.g., *gethostbyname()* on "www.foo.com"

Server application

- Extract client IP address from socket
- Optionally invoke system call to translate into name
- E.g., *gethostbyaddr()* on "12.34.158.5"

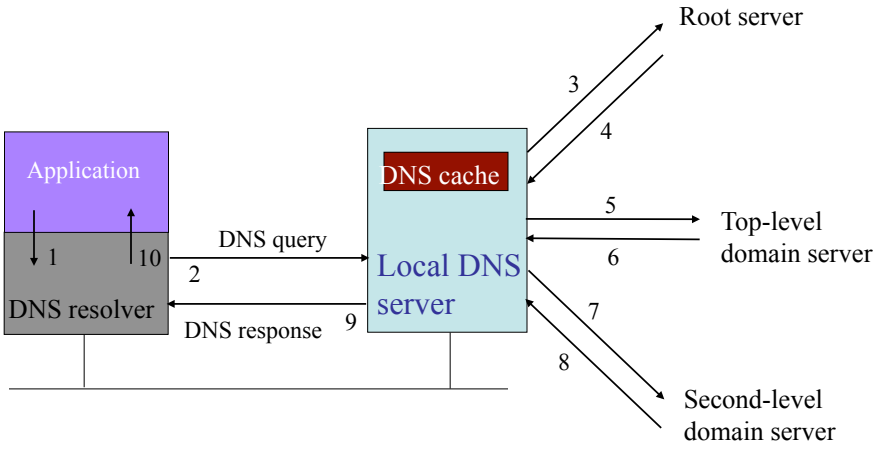
33

Domain Name System



34 12.34.56.0/24

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.

35

Sockets

What exactly are sockets?

- an endpoint of a connection
- similar to UNIX file I/O API (provides a file descriptor)
- associated with each end-point (end-host) of a connection
  - identified by the IP address and port number of *both* the sender and receiver

Berkeley sockets is the most popular network API

- runs on Linux, FreeBSD, OS X, Windows
- fed off the popularity of TCP/IP
- can build higher-level interfaces on top of sockets
  - e.g., Remote Procedure Call (RPC)

Based on C, single threaded model

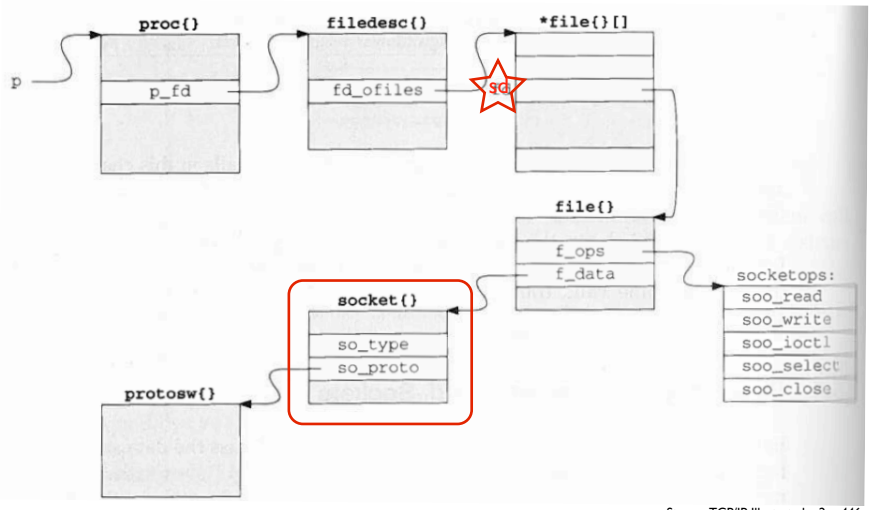
- does not require multiple threads

Useful sample code available at

- http://www.kohala.com/start/unpv12e.html

36

Process File Table and Socket Descriptor



Stevens TCP/IP Illustrated v. 2 p. 446

37

Types of Sockets

Different types of sockets implement different service models

- Stream v.s. datagram

Stream socket (aka TCP)

- connection-oriented
- reliable, in order delivery
- at-most-once delivery, no duplicates
- used by e.g., ssh, http

Datagram socket (aka UDP)

- connectionless (just data-transfer)
- "best-effort" delivery, possibly lower variance in delay
- used by e.g., IP telephony, streaming audio, streaming video, Internet gaming, etc.

38

Simplified E-mail Delivery

You want to send email to friend@cs.usc.edu

At your end, your mailer

- translates cs.usc.edu to its IP address (128.125.1.45)
- decides to use TCP as the transport protocol (Why?)
- creates a socket
- `connect`s to 128.125.1.45 at the well-known SMTP port # (25)
- parcels out your email into packets
- `send`s the packets out

On the Internet, your packets got:

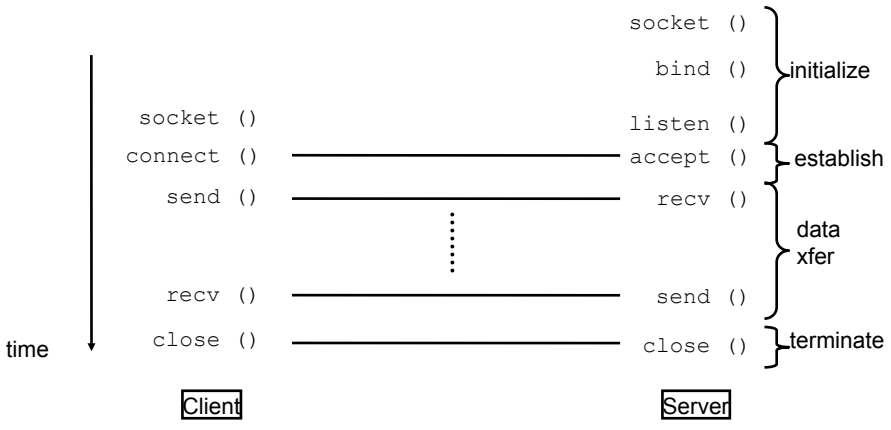
- transmitted
- routed
- buffered
- forwarded, or
- dropped

At the receiver, smtpd

- must make a "receiver" ahead of time:
- creates a socket
- decides on TCP
- `bind`s the socket to smtp's well-known port #
- `listen`s on the socket
- `accept`s your smtp connection requests
- `recv`es your email packets

39

Stream/TCP Sockets



40

Stream/TCP Socket

Server:

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

Client:

- creates client-local TCP socket
- specifies IP address, port number of server process
- When client contacts server: 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

41

Initialize (Client)

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

`socket()` creates a socket data structure and attaches it to the process's file descriptor table

Handling errors that occur rarely usually consumes most of systems code

42

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

`connect()` initiates connection (for TCP)

43

Sending Data Stream (Client)

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

   if (send_bytes < 0)
        perror("send");

   return 0;
}
```

- returns how many bytes are actually sent
- must loop to make sure that all is sent
  (except for blocking I/O, see UNP Section 6.2)

What is blocking and non-blocking I/O?
Why do you want to use non-blocking I/O?

44

Initialize (Server)

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

`SO_REUSEADDR` allows server to restart or multiple servers to bind to the same port # with different IP addresses

45

Initialize (Server `bind` addr)

```
struct sockaddr_in sin;

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

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

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

`bind()` used only by server, to "label" a socket with an IP address and/or port#

- Why do we need to label a socket with a port#?
- Must each service have a well-known port?
- Why do we need to label a socket with IP address?
- What if we want to receive packets from all network interfaces of the server machine?
- Why not always receive from all interfaces?
- What defines a connection?

46

Initialize (Server `listen`)

```
if (listen(sd, qlen) < 0) {
 perror("error listening");
 abort();
}
```

- specifies max number of pending TCP connections waiting to be accepted (using `accept()`)
- only useful for connection oriented services, but may be used by UDP also
- TCP SYN denial of service attack

API design question: why not merge `bind()` and `listen()`?

47

Establish (Server `accept`)

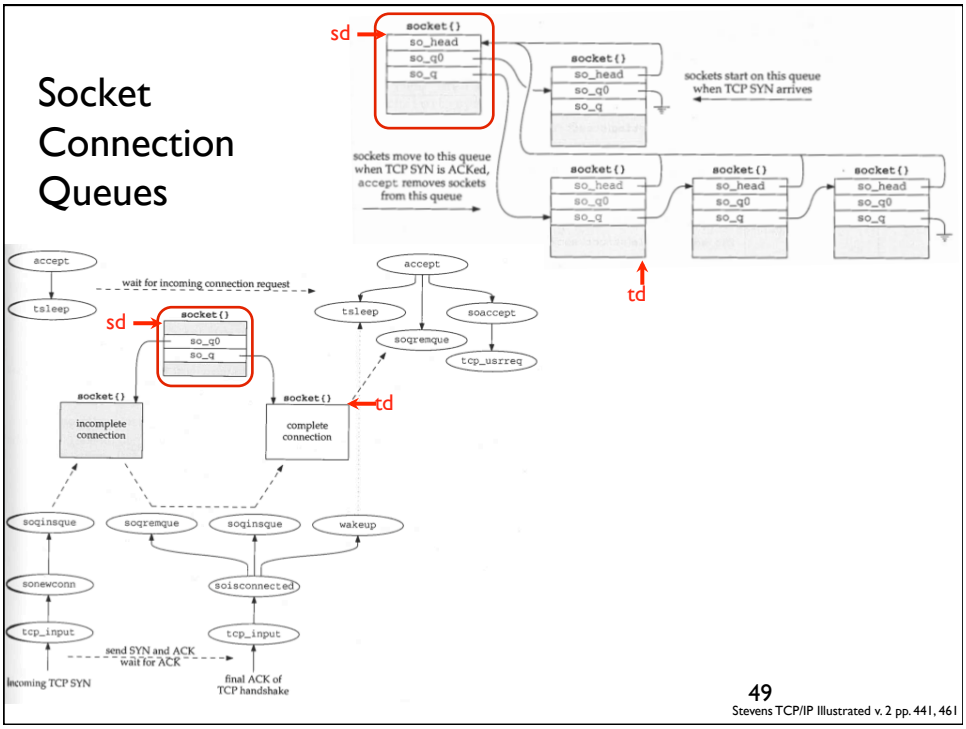
```
int addr_len = sizeof(addr);
int td;

td = accept(sd, (struct sockaddr *) &addr, &addr_len);

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

- waits for incoming client connection
- returns a connected socket (different from the `listen`ed to socket)

API design question: why not merge `listen()` and `accept()`?

48

Socket Connection Queues



49

Stevens TCP/IP Illustrated v. 2 pp. 441, 461

Receiving Data Stream (Server)

```
int
receive_packets(char *buffer, int buffer_len, int
  *bytes_read)
{
    int left = buffer_len - *bytes_read;
    received = recv(td, buffer + *bytes_read, left, 0);
    . . . .
    return 0;
}
```

- returns the number of bytes actually received
- 0 if connection is closed, -1 on error
- if non-blocking: -1 if no data, with `errno` set to `EWOULDBLOCK`
- must loop to ensure all data is received
- Why doesn't recv return all of the data at once?
- How do you know you have received everything sent?

50

Connection `close` (Client and Server)

- `close()` marks socket unusable
- actual tear down depends on TCP
  `bind: Address already in use`
- socket option `SO_LINGER` can be used to specify whether `close()` should return immediately or abort connection or wait for termination
- The APIs `getsockopt()` and `setsockopt()` are used to query and set socket options (see UNP Ch. 7)
- Other useful options:
  - `SO_RCVBUF` and `SO_SNDBUF` used to set buffer sizes
  - `SO_KEEPALIVE` tells server to ping client periodically

51

How to Handle Multiple I/O Streams?

Where do we get incoming data?

- `stdin` (typically keyboard/mouse input)
- sockets

Asynchronous arrival, program doesn't know when data will arrive

Alternatives:

multithreading: each thread handles one I/O stream (482)

I/O multiplexing: a single thread handles multiple I/O streams

Flavors:

a. blocking I/O (default):
   - put process to sleep until I/O is ready
   - blocking for: device availability and I/O completion
   - by polling or use of `select()`

b. non-blocking I/O:
   - only checks for device availability
   - by polling or signal driven (not covered)

c. asynchronous I/O:
   - process is notified when I/O is completed (not covered)

52

Non-Blocking I/O: Polling

```
int opts = fcntl(sock, F_GETFL);
if (opts < 0) {
    perror("fcntl(F_GETFL)");
    abort();
}

if (fcntl(sock, F_SETFL, opts | O_NONBLOCK) < 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;
    }
}
```

get current socket option settings

set non-blocking I/O socket option

get data from socket

get user input

53

Blocking I/O: `select()`

`select(maxfd, readset, writeset, exceptset, timeout)`

- waits on multiple file descriptors/sockets and timeout
- application does not consume CPU cycles while waiting
- `maxfd` is the maximum file descriptor number + 1
  - if you have only one descriptor, number 5, `maxfd` is 6
- descriptor sets provided as bit mask
  - use `FD_ZERO`, `FD_SET`, `FD_ISSET`, and `FD_CLR` to work with the descriptor sets
- returns as soon as *one* of the specified sockets are ready to be read or written, or they have an error, or timeout exceeded
  - returns # of ready sockets, -1 on error, 0 if timed out and no device is ready (what for?)

54

Blocking I/O: `select()`

set up parameters for `select()`

run `select()`

interpret result

```
fd_set read_set;
struct timeval time_out;
while (1) {
    FD_ZERO(read_set);
    FD_SET(stdin, read_set); /* stdin is typically 0 */
    FD_SET(sd, read_set);
    time_out.tv_usec = 100000; time_out.tv_sec = 0;

    err = select(MAX(stdin, sd) + 1, &read_set, NULL, NULL, &time_out);

    if (err < 0) {
        perror ("select");
        abort ();
    }
    if (err > 0) {
        if (FD_ISSET(sd, 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;
    }
    else { . . . /* timed out */ }
}
```

55

Blocking I/O: polling

Which of the following would you use? Why?

```
loop {
  select(. . . , timeout);

  recv();
} till done;
```

or:

```
loop {
  sleep(seconds)

  recv();
} till done;
```

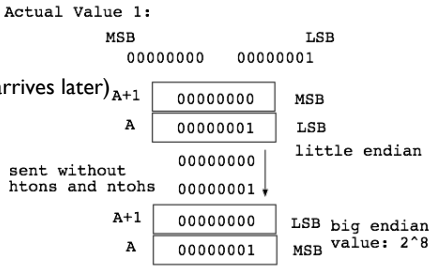
56

Byte Ordering

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

Little-endian:
Most Significant Byte (MSB) in high address (sent/arrives later)
(Intel x86 and Alpha)

Big-endian: MSB in low address (sent/arrives first)
(PowerPC, Sun Sparc, HP-PA)



Bi-endian: switchable endians (ARM, PowerPC after G5, Alpha, SPARC V9)

57

Byte Ordering Solution

To ensure interoperability, **ALWAYS** translate `short`, `long`, `int`, `uint16_t`, `uint32_t`, to/from "network byte order" before/after transmission

Use these macros:

`htons()`: host to network short
`htonl()`: host to network long
`ntohs()`: network to host short
`ntohl()`: network to host long

Do we have to be concerned about byte ordering for `char` type?
How about `float` and `double`?

58

Establish (Client)

```
struct sockaddr_in sin;
struct hostent *host = gethostbyname(argv[1]); // argv[1] contains host name
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(sd, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
  perror("connect");
  printf("Cannot connect to server\n");
  abort();
}
```

host name, e.g., www.eecs.umich.edu

- identifies a single host
- variable length string
- maps to one or more IP address
- `gethostbyname()` translates host name to IP address

59

Naming and Addressing

Example DNS name in ASCII string: www.eecs.umich.edu

Its IP address in dotted-decimal (dd) ASCII string: 141.212.113.110

Its IP address in 32-bit binary representation:
10001101 11010100 01110001 01101110

Why do we need names instead of using the addresses directly?

Why do we need addresses in addition to names?

60

Name and Address Manipulation

Syscalls to map name to/from address:

- dns to binary: `gethostbyname()`
- binary to dns: `gethostbyaddress()`

and to change representation:

- dd to binary: `inet_aton()`
- binary to dd: `inet_ntoa()`

dns to dd: `gethostbyname()` plus `inet_ntoa()`

`gethostbyname()` and `gethostbyaddr()` both return `struct hostent` that contains both binary & dd (See Fig. 11.2 of UNP)

Other useful syscalls:

- `gethostname()`: returns DNS name of current host
- `getsockname()`: returns IP address bound to socket (in binary)

Used when address and/or port is not specified (`INADDR_ANY`), to find out the actual address and/or port used

- `getpeername()`: returns IP address of peer (in binary)

61

Flat vs. Hierarchical Space

Example of flat name space:

- file system that doesn't support folders/sub-directories

Examples of hierarchical name space:

- Duncan McLeod, William Wallace

Examples of hierarchical address space:

- 5 Wilberforce Rd., Cambridge, Cambridgeshire, England, UK
- Japan, Tokyo-to, Minato-ku, Shirokanedai 4-chome 6-41
- +1 734 763 1583

Why form hierarchy?

- John Doe
- John Smith
- John Keynes
- John Woo

Advantage of hierarchical space: allows for decentralized management

62

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

63