Sockets Programming

EECS 489 Computer Networks

http://www.eecs.umich.edu/~zmao/eecs489

Z. Morley Mao

Thursday Sept 16, 2004

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

Outline

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

Administrivia

- Homework 1 is online
  - www.eecs.umich.edu/~zmao/eecs489/hw1
  - Code will shortly available by tomorrow

Quiz!

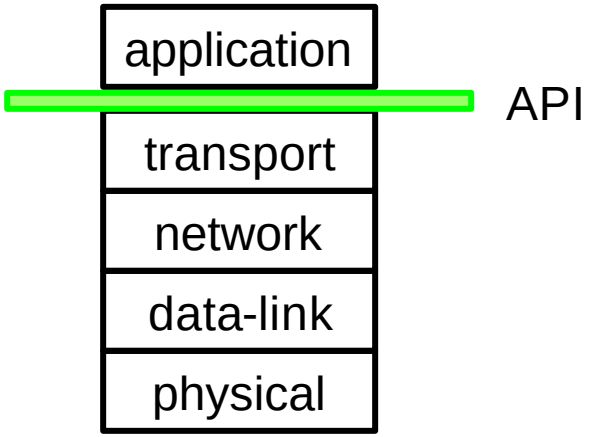
- What is wrong with the following code?

```
void alpha () {
   rtgptr ptr = rtg_headptr;
   while (ptr != NULL) {
        rtg_check (ptr);
        ptr = ptr->nextptr;
   }
}

void rtg_check (rtgptr ptr) {
   if (ptr->value == 0)
        free (ptr);
}
```

```
struct routeptr {
  int value;
  struct routeptr *nextptr;
}

typedef struct routeptr rtgptr;
```

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

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)

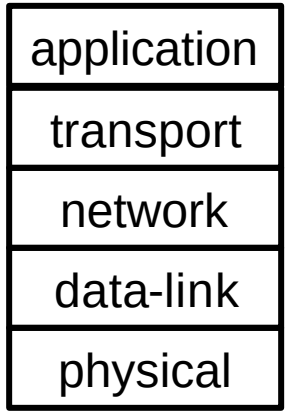
Types of Sockets (1)

- 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

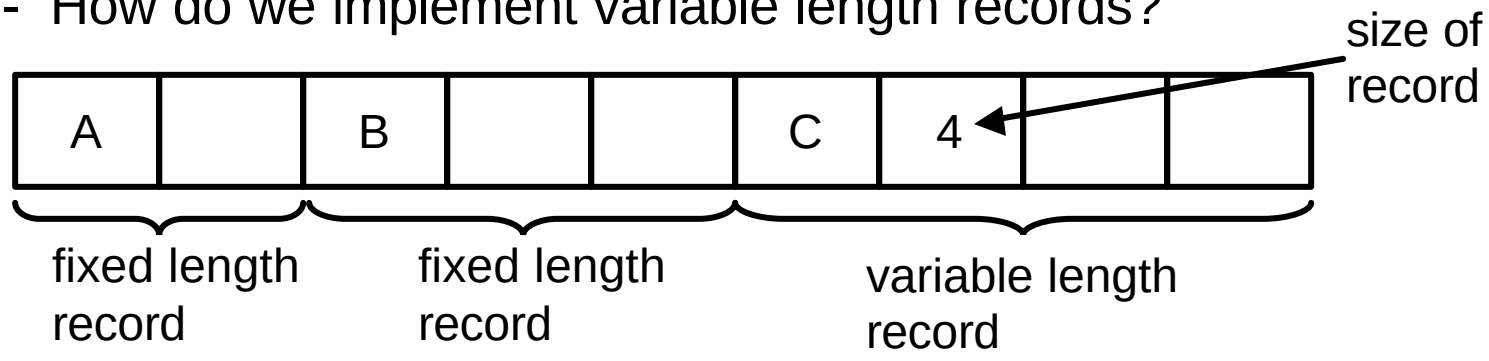
TCP sends messages joined together, ie. “Hi there!Hope you are well”



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?

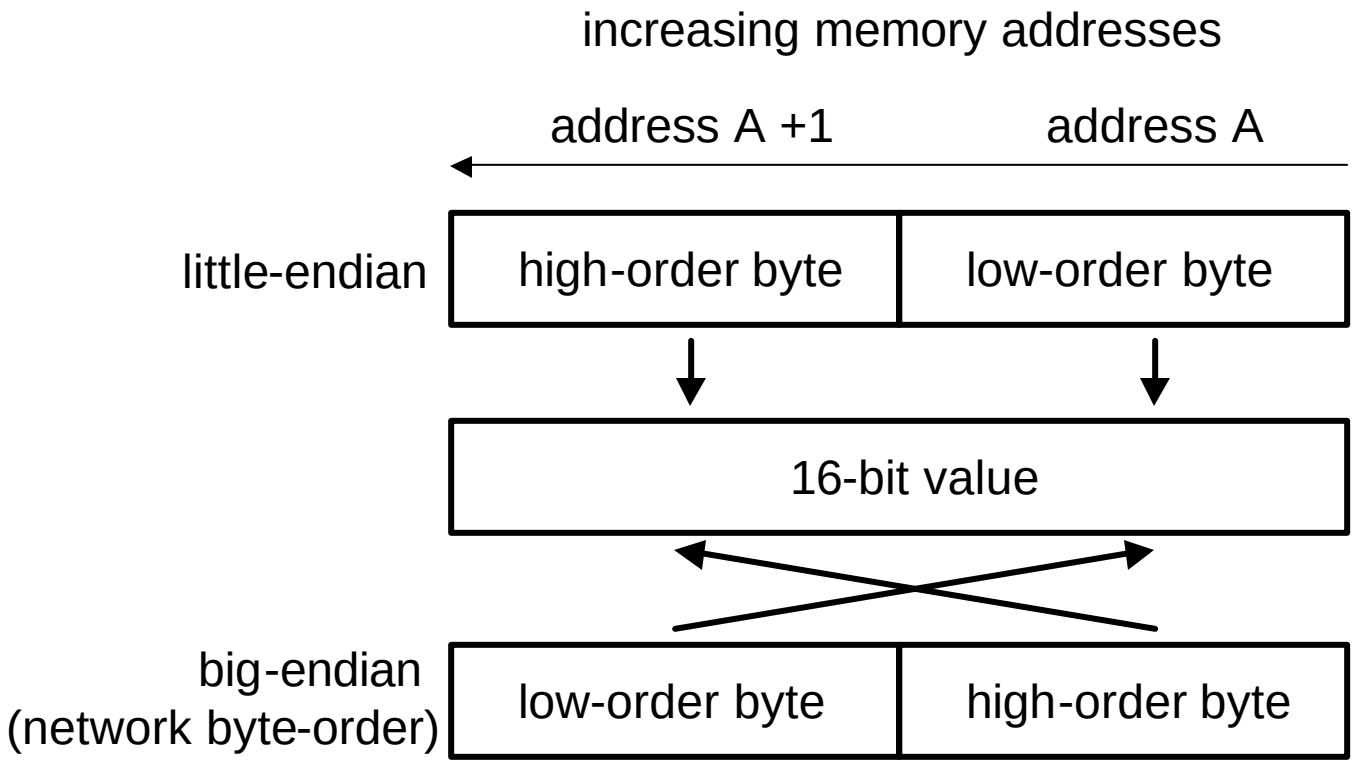
Types of Sockets (4)

- 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



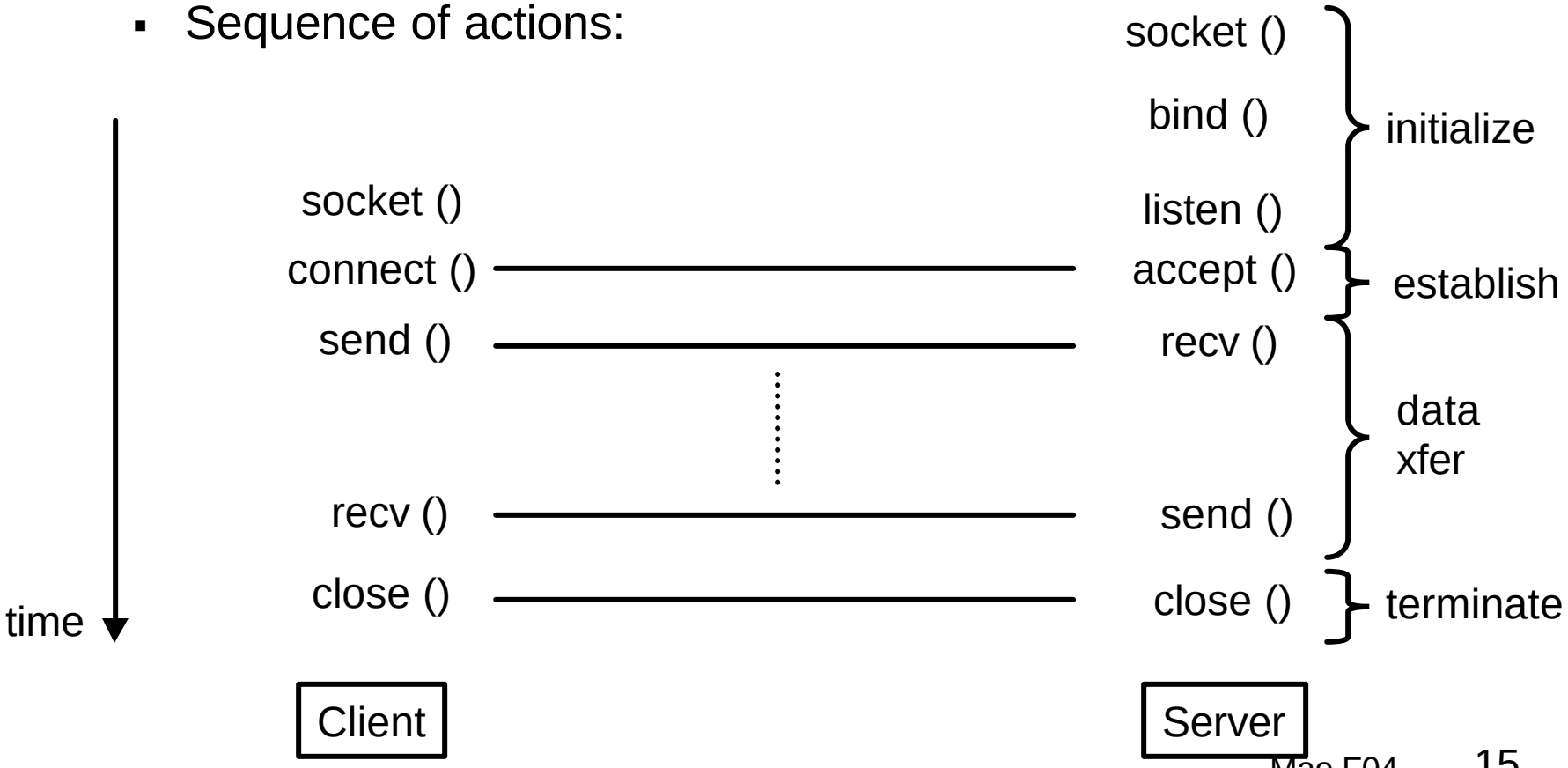
Byte Ordering Solution

```
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

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)

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 (sock, read_set);
    time_out.tv_usec = 100000; time_out.tv_sec = 0;
    select_retval = select(MAX(stdin, sock) + 1, &read_set, NULL,
                           NULL, &time_out);
    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

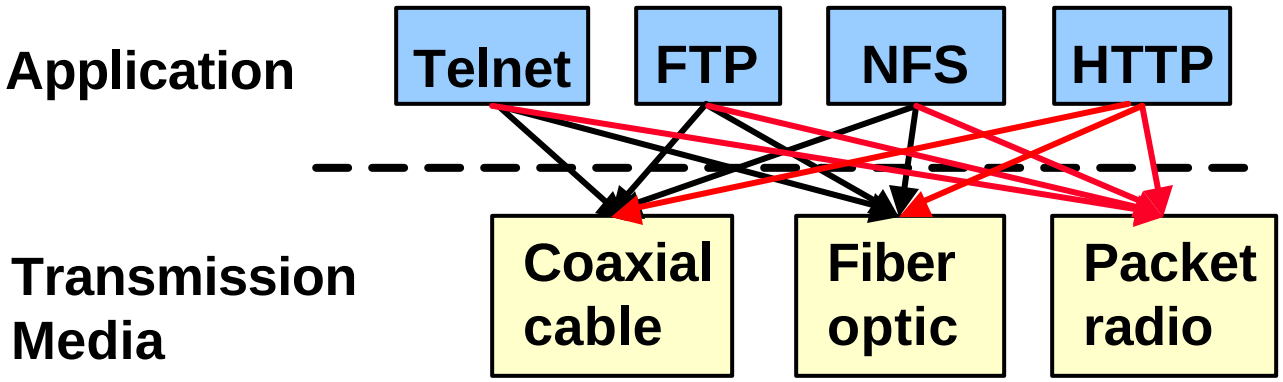
- Common mistakes:
  - C programming
    - Use gdb
    - Use printf for debugging, remember to do `fflush(stdout);`
  - Byte-ordering
  - Use of `select()`
  - Separating records in TCP stream
  - Not knowing what exactly gets transmitted on the wire
    - Use tcpdump / Ethereal
- Hints:
  - Use man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

Network Architecture

A Quick Review

- Many different network styles and technologies
  - circuit-switched vs packet-switched, etc.
  - wireless vs wired vs optical, etc.
- Many different applications
  - ftp, email, web, P2P, etc.
- How do we organize this mess?

The Problem



- Do we re-implement every application for every technology?
- Obviously not, but how does the Internet architecture avoid this?

Today's Lecture: Architecture

- Architecture is not the implementation itself
- Architecture is how to "organize" implementations
  - what interfaces are supported
  - where functionality is implemented
- Architecture is the modular design of the network

Software Modularity

Break system into modules:

- Well-defined interfaces gives flexibility
  - can change implementation of modules
  - can extend functionality of system by adding new modules
- Interfaces hide information
  - allows for flexibility
  - but can hurt performance

Network Modularity

Like software modularity, but with a twist:

- Implementation distributed across routers and hosts
- Must decide both:
  - how to break system into modules
  - where modules are implemented
- Lecture will address these questions in turn

Outline

- Layering
  - how to break network functionality into modules
- The End-to-End Argument
  - where to implement functionality

Layering

- Layering is a particular form of modularization
- System is broken into a vertical hierarchy of logically distinct entities (layers)
- Service provided by one layer is based solely on the service provided by layer below
- Rigid structure: easy reuse, performance suffers

ISO OSI Reference Model for Layers

- Application
- Presentation
- Session
- Transport
- Network
- Datalink
- Physical

Layering Solves Problem

- Application layer doesn’t know about anything below the presentation layer, etc.
- Information about network is hidden from higher layers
- Ensures that we only need to implement an application once!
- Caveat: not quite....

OSI Model Concepts

- Service: what a layer does
- Service interface: how to access the service
  - Interface for layer above
- Peer interface (protocol): how peers communicate
  - Set of rules and formats that govern the communication between two network boxes
  - Protocol does not govern the implementation on a single machine, but how the layer is implemented between machines

Physical Layer (1)

- **Service**: move information between two systems connected by a physical link
- **Interface**: specifies how to send a bit
- **Protocol**: coding scheme used to represent a bit, voltage levels, duration of a bit
- Examples: coaxial cable, optical fiber links; transmitters, receivers

Datalink Layer (2)

- **Service**:
  - Framing (attach frame separators)
  - Send data frames between peers
  - Others:
    - arbitrate the access to common physical media
    - per-hop reliable transmission
    - per-hop flow control

- **Interface**: send a data unit (packet) to a machine connected to the same physical media
- **Protocol**: layer addresses, implement Medium Access Control (MAC) (e.g., CSMA/CD)…

Network Layer (3)

- **Service**:
  - Deliver a packet to specified network destination
  - Perform segmentation/reassemble
  - Others:
    - packet scheduling
    - buffer management

- **Interface**: send a packet to a specified destination
- **Protocol**: define global unique addresses; construct routing tables

Transport Layer (4)

- **Service**:
  - Demultiplexing
  - Optional: error-free and flow-controlled delivery
- **Interface**: send message to specific destination
- **Protocol**: implements reliability and flow control
- Examples: TCP and UDP

Session Layer (5)

- **Service**:
  - Full-duplex
  - Access management (e.g., token control)
  - Synchronization (e.g., provide check points for long transfers)
- **Interface**: depends on service
- **Protocol**: token management; insert checkpoints, implement roll-back functions

Presentation Layer (6)

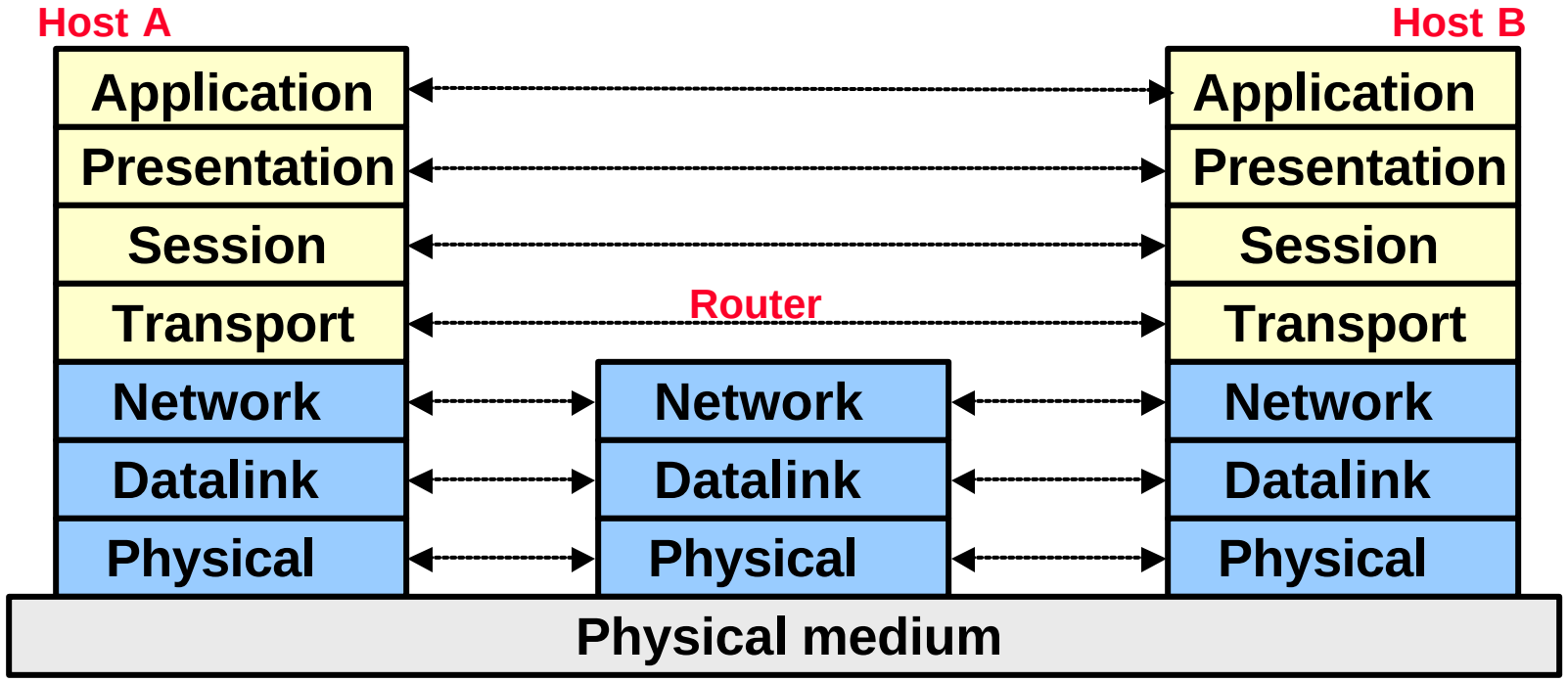
- **Service**: convert data between various representations
- **Interface**: depends on service
- **Protocol**: define data formats, and rules to convert from one format to another

Application Layer (7)

- **Service**: any service provided to the end user
- **Interface**: depends on the application
- **Protocol**: depends on the application
- Examples: FTP, Telnet, WWW browser

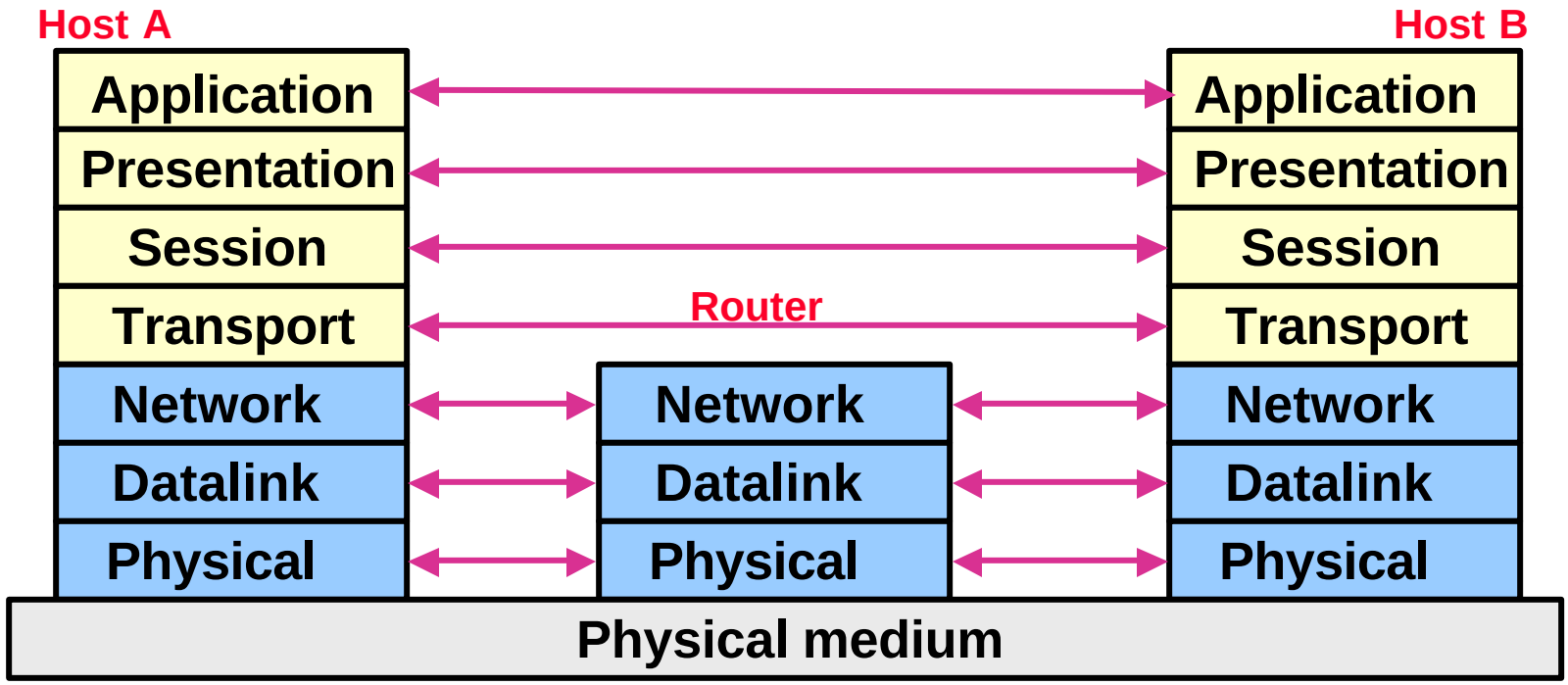
Who Does What?

- Seven layers
  - Lower three layers are implemented everywhere
  - Next four layers are implemented only at hosts



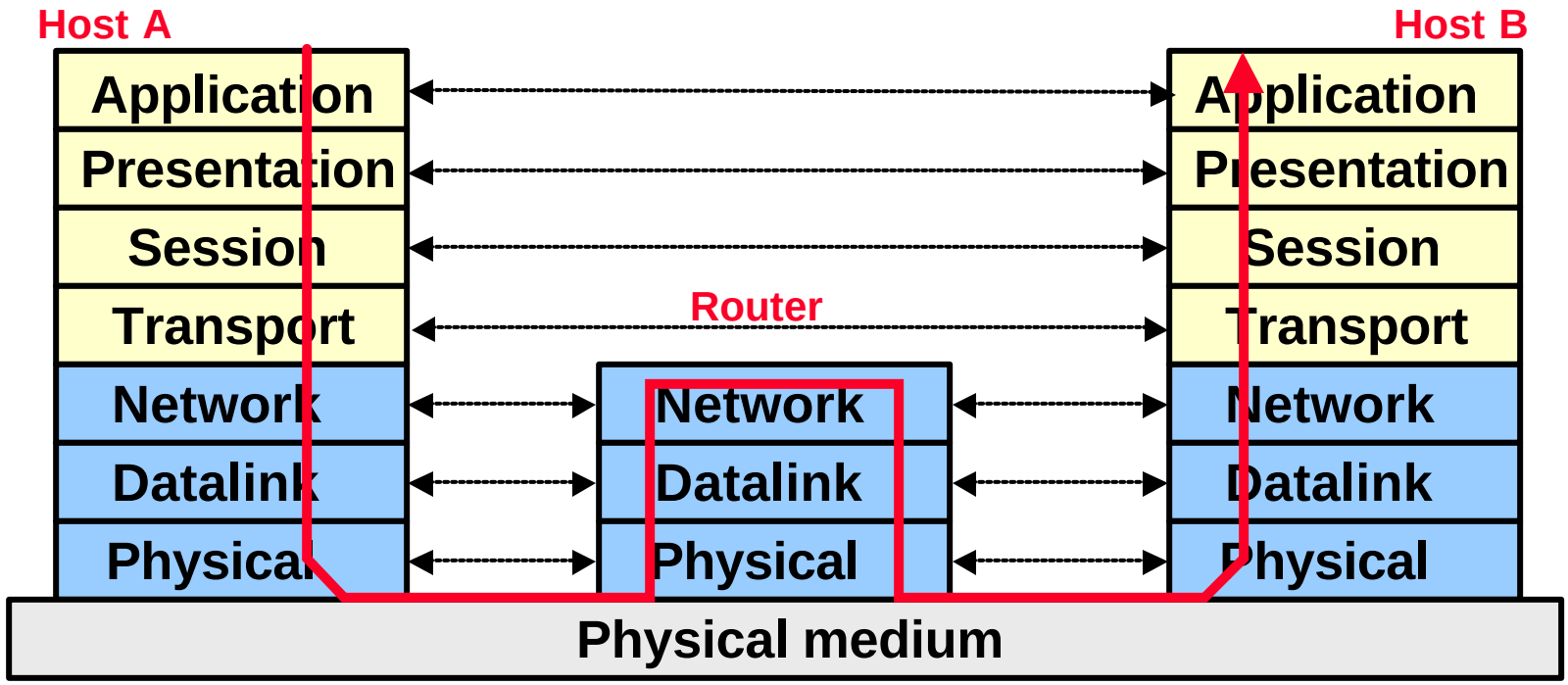
Logical Communication

- Layers interacts with corresponding layer on peer



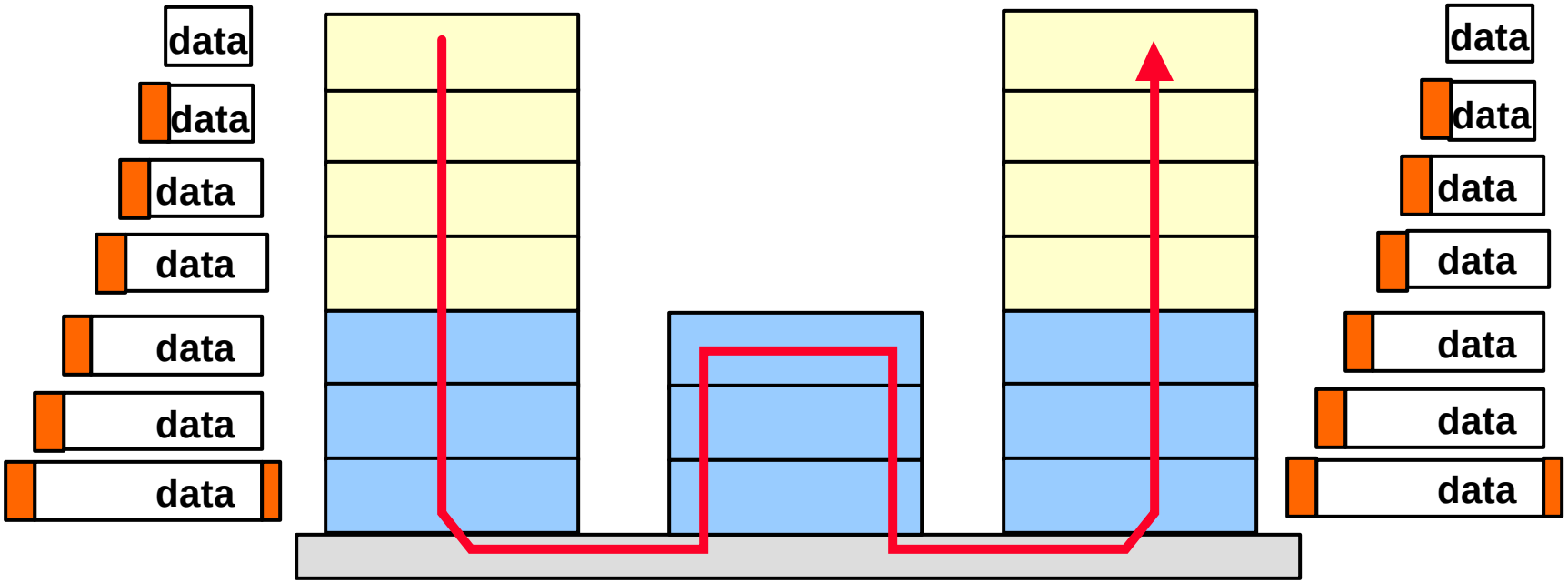
Physical Communication

- Communication goes down to physical network, then to peer, then up to relevant layer



Encapsulation

- A layer can use only the service provided by the layer immediate below it
- Each layer may change and add a header to data packet



Example: Postal System

Standard process (historical):

- Write letter
- Drop an addressed letter off in your local mailbox
- Postal service delivers to address
- Addressee reads letter (and perhaps responds)

Postal Service as Layered System

Layers:

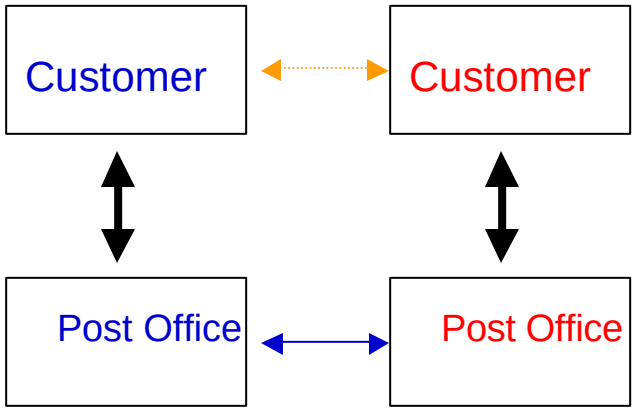
- Letter writing/reading
- Delivery

Information Hiding:

- Network need not know letter contents
- Customer need not know how the postal network works

Encapsulation:

- Envelope



Questions?

Standards Bodies

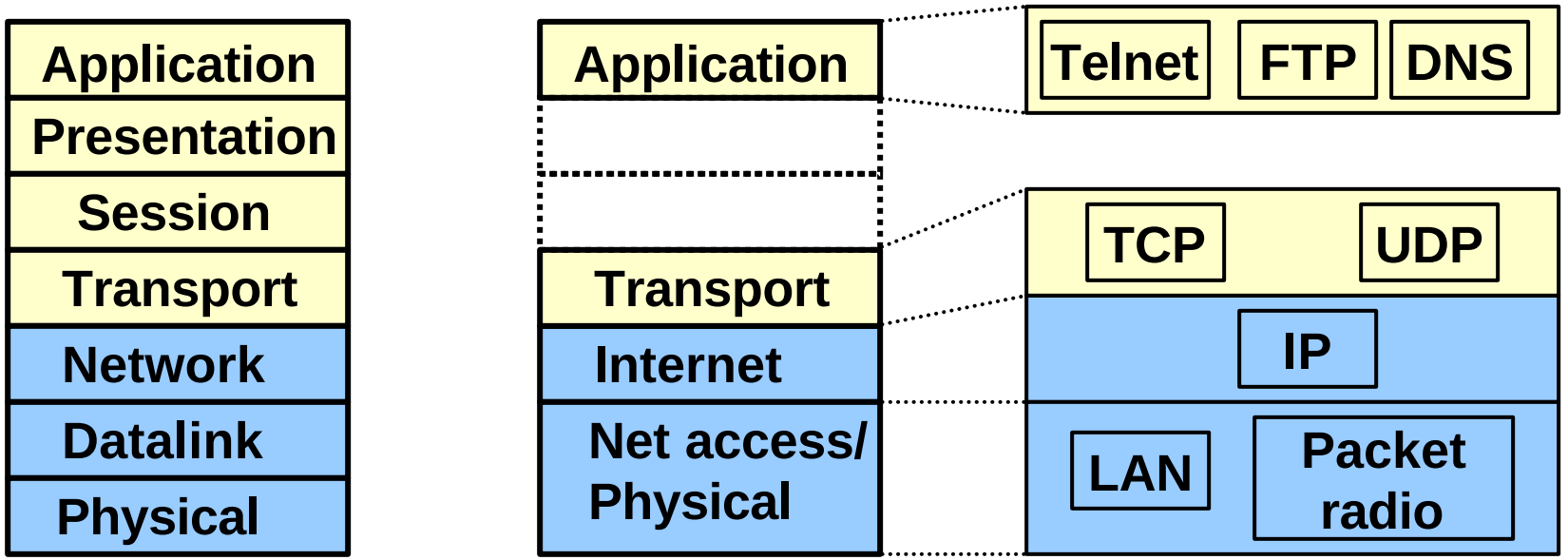
- ISO: International Standards Organization
  - professional bureaucrats writing standards
  - produced OSI layering model

- IETF: Internet Engineering Task Force
  - started with early Internet hackers
  - more technical than bureaucratic

"We reject kings, presidents, and voting. We believe in rough consensus and running code" (David Clark)

OSI vs. Internet

- OSI: conceptually define services, interfaces, protocols
- Internet: provide a successful implementation

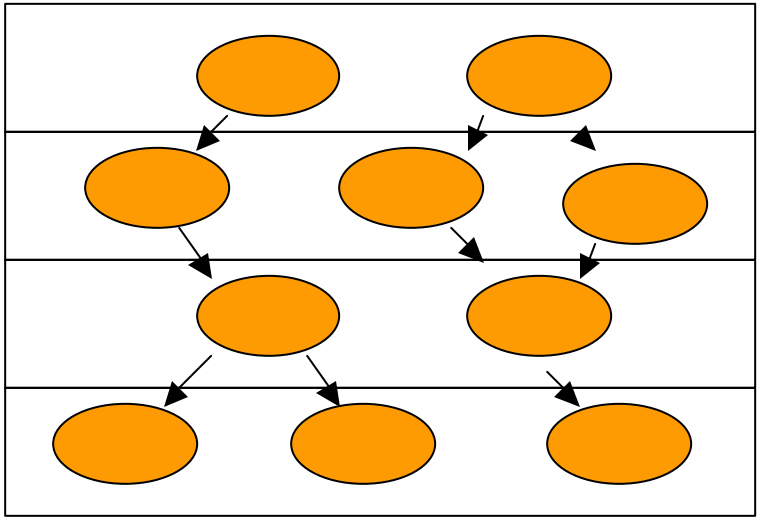


OSI (formal) **Internet (informal)**

Multiple Instantiations

- Can have several instantiations for each layer
  - many applications
  - many network technologies
  - transport can be reliable (TCP) or not (UDP)
- Applications dictate transport
  - In general, higher layers can dictate lower layer
- But this is a disaster!
  - applications that can only run certain networks

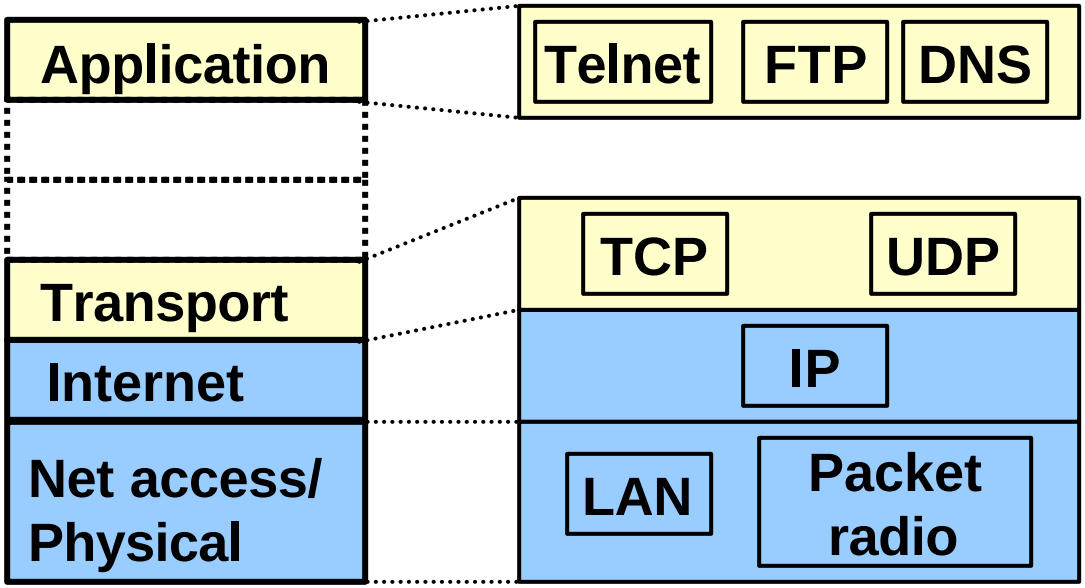
Multiple Instantiations of Layers



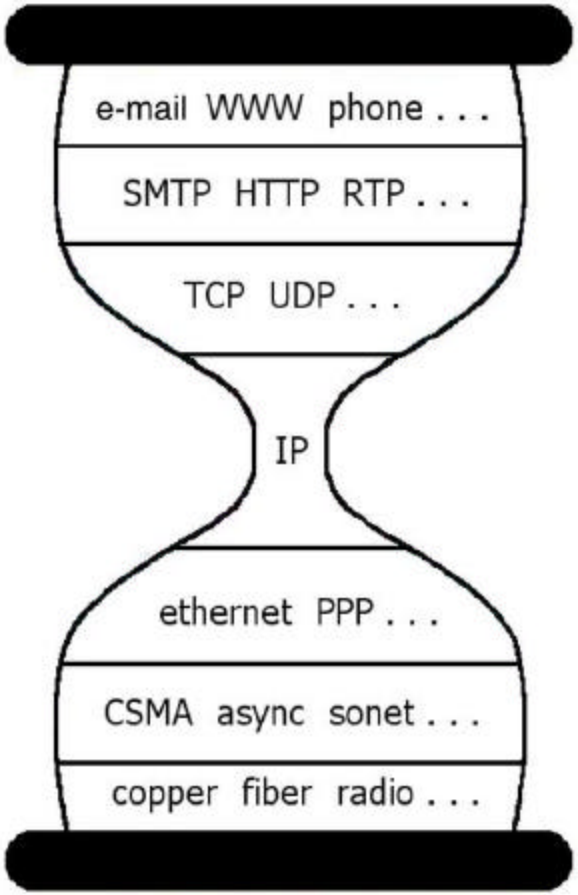
Solution

A universal Internet layer:

- Internet has only IP at the Internet layer
- Many options for modules above IP
- Many options for modules below IP



Hourglass


e-mail WWW phone . . .
SMTP HTTP RTP . . .
TCP UDP . . .
IP
ethernet PPP . . .
CSMA async sonet . . .
copper fiber radio . . .

Implications of Hourglass

A single Internet layer module:

- Allows all networks to interoperate
  - all networks technologies that support IP can exchange packets (my rant last lecture)
- Allows all applications to function on all networks
  - all applications that can run on IP can use any network
- Simultaneous developments above and below IP

Network Modularity

Two crucial decisions

- Layers, not just modules
  - alternatives?
- Single internetworking layer, not multiple
  - alternatives?

Back to Reality

- Layering is a convenient way to think about networks
- But layering is often violated
  - Firewalls
  - Transparent caches
  - NAT boxes
  - .....
- More on this later....on to part two of this lecture

- Questions?

Placing Functionality

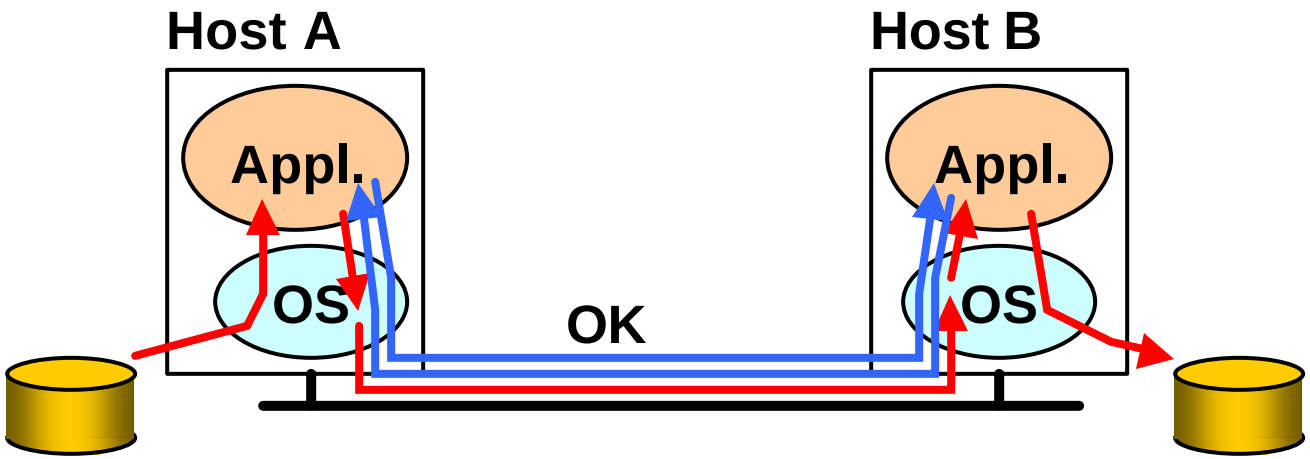
- The most influential paper about placing functionality is “End-to-End Arguments in System Design” by Saltzer, Reed, and Clark

- The “Sacred Text” of the Internet
  - Endless disputes about what it means
  - Everyone cites it as supporting their position

Basic Observation

- Some applications have end-to-end performance requirements
  - Reliability, security, etc.
- Implementing these in the network is very hard:
  - Every step along the way must be fail-proof
- The hosts:
  - Can satisfy the requirement without the network
  - Can’t depend on the network

Example: Reliable File Transfer



- Solution 1: make each step reliable, and then concatenate them
- Solution 2: end-to-end check and retry

Example (cont'd)

- Solution 1 not complete
  - What happens if any network element misbehaves?
  - The receiver has to do the check anyway!
- Solution 2 is complete
  - Full functionality can be entirely implemented at application layer with no need for reliability from lower layers
- Is there any need to implement reliability at lower layers?

Conclusion

Implementing this functionality in the network:

- Doesn't reduce host implementation complexity
- Does increase network complexity
- Probably imposes delay and overhead on all applications, even if they don't need functionality

- However, implementing in network can enhance performance in some cases
  - very lossy link

Conservative Interpretation

- “Don’t implement a function at the lower levels of the system unless it can be completely implemented at this level” (Peterson and Davie)

- Unless you can relieve the burden from hosts, then don’t bother

Radical Interpretation

- Don’t implement anything in the network that can be implemented correctly by the hosts
  - E.g., multicast
- Make network layer absolutely minimal
  - Ignore performance issues

Moderate Interpretation

- Think twice before implementing functionality in the network
- If hosts can implement functionality correctly, implement it a lower layer only as a performance enhancement
- But do so only if it does not impose burden on applications that do not require that functionality

Extended Version of E2E Argument

- Don’t put application semantics in network
  - Leads to loss of flexibility
  - Cannot change old applications easily
  - Cannot introduce new applications easily
- Normal E2E argument: performance issue
  - Introducing more functionality imposes more overhead
  - Subtle issue, many tough calls (e.g., multicast)
- Extended version:
  - Short-term performance vs long-term flexibility

Back to Reality (again)

- Layering and E2E Principle regularly violated:
  - Firewalls
  - Transparent caches
  - Other middleboxes
- Battle between architectural purity and commercial pressures
  - Extremely important
  - Imagine a world where new apps couldn't emerge

Summary

- Layering is a good way to organize networks
- Unified Internet layer decouples apps from networks
- E2E argument encourages us to keep IP simple
- Commercial realities threaten to undo all of this...