CompSci 356: Computer Network Architectures
Lecture 4: Hardware and physical links
References: Chap 1.4, 1.5 of [PD]

Xiaowei Yang

xwy@cs.duke.edu

Overview

- Application Programming Interface (cont.)

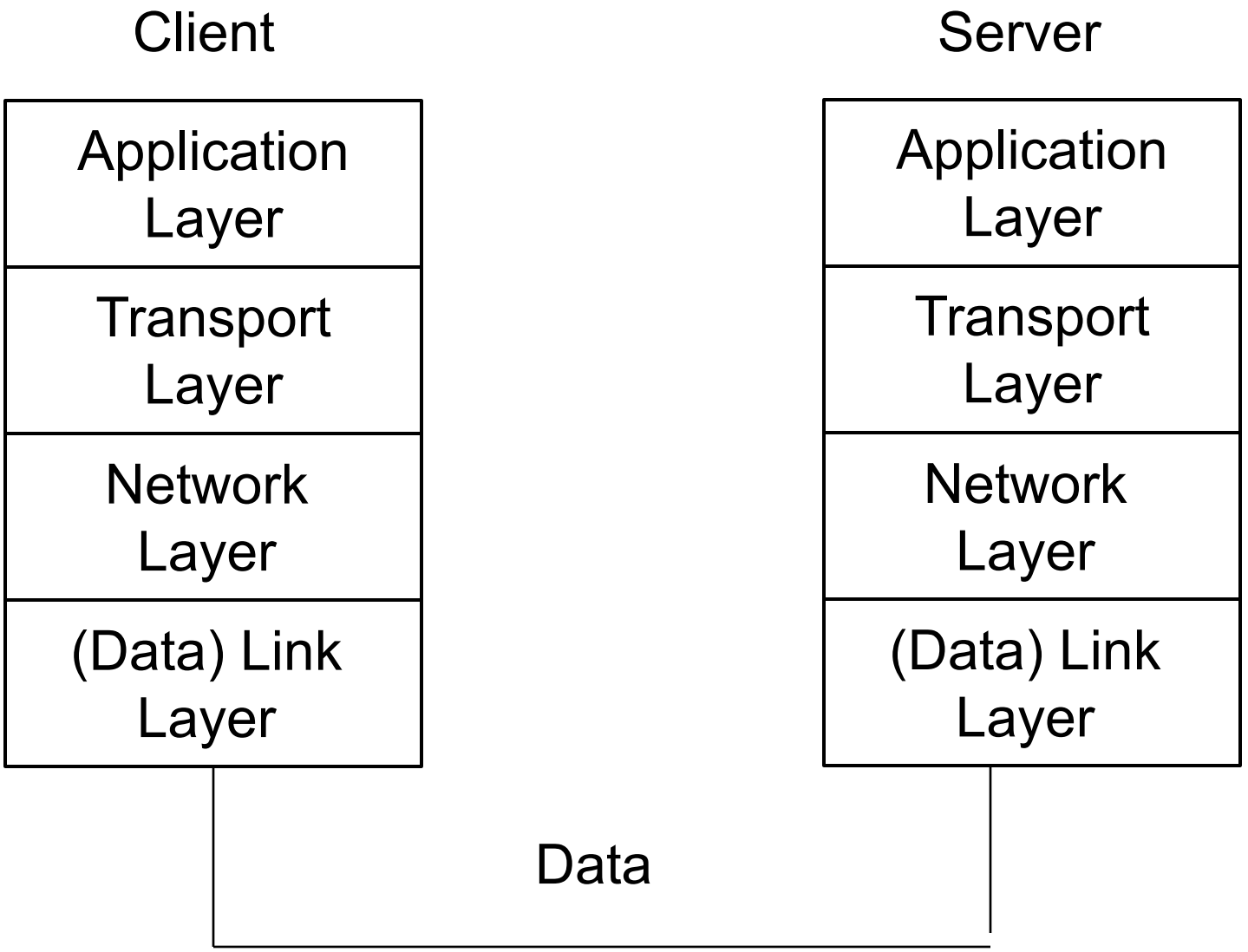
- Hardware and physical layer
  - Nuts and bolts of networking
  - Nodes
  - Links
    - Bandwidth, latency, throughput, delay-bandwidth product
    - Physical links

Application Programming Interface (Sockets)

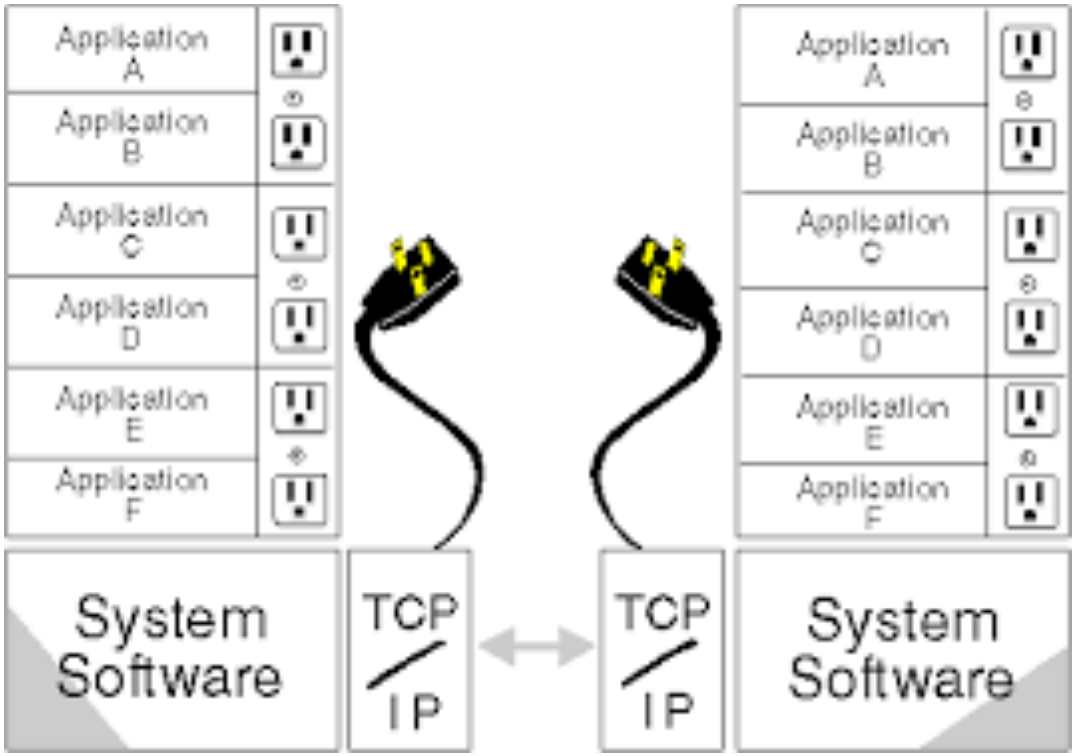
- Socket Interface was originally provided by the Berkeley distribution of Unix
  - Now supported in virtually all operating systems

- Each protocol provides a certain set of *services*, and the API provides a syntax by which those services can be invoked in this particular OS

A layered architecture



Socket



- What is a socket?
  - The point where a local application process attaches to the network
  - An interface between an application and the network
  - An application creates the socket

Socket Interface

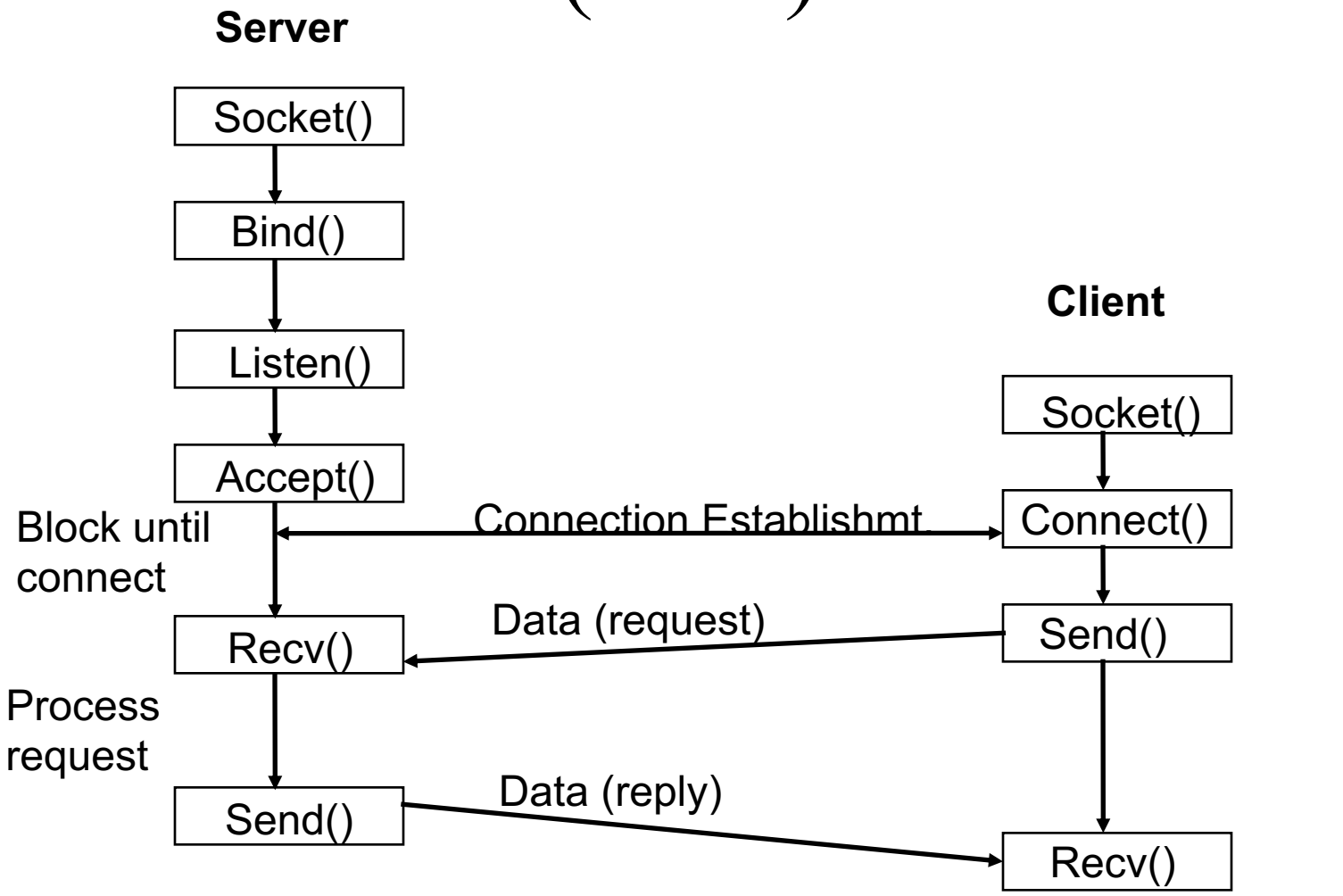
- The interface defines operations for
  - Creating a socket
  - Attaching a socket to the network
  - Sending and receiving messages through the socket
  - Closing the socket

Socket

- Socket Family
  - PF_INET denotes the Internet family
  - PF_UNIX denotes the Unix pipe facility
  - PF_PACKET denotes direct access to the network interface (i.e., it bypasses the TCP/IP protocol stack)

- Socket Type
  - SOCK_STREAM is used to denote a byte stream
  - SOCK_DGRAM is an alternative that denotes a message oriented service, such as that provided by UDP

Connection-oriented example (TCP)



[Paul Barford]

Creating a Socket

```
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

Client-Serve Model with TCP

Server

- Passive open
- Prepares to accept connection, does not actually establish a connection

Server invokes

```
int bind (int socket, struct sockaddr *address,
                                    int addr_len)
int listen (int socket, int backlog)
int accept (int socket, struct sockaddr *address,
                                int *addr_len)
```

Client-Serve Model with TCP

Bind

- Binds the newly created socket to the specified address i.e. the network address of the local participant (the server)

- Address is a data structure which combines IP and port

Listen

- Defines how many connections can be pending on the specified socket

Client-Serve Model with TCP

Accept

- Carries out the passive open
- Blocking operation
  - Does not return until a remote participant has established a connection
  - When it does, it returns a new socket that corresponds to the new established connection and the address argument contains the remote participant's address

Client-Serve Model with TCP

Client

- Application performs active open
- It says who it wants to communicate with

Client invokes

```
int connect (int socket, struct sockaddr *address,
int addr_len)
```

Connect

- Does not return until TCP has successfully established a connection at which application is free to begin sending data
- Address contains remote machine's address

Client-Serve Model with TCP

In practice

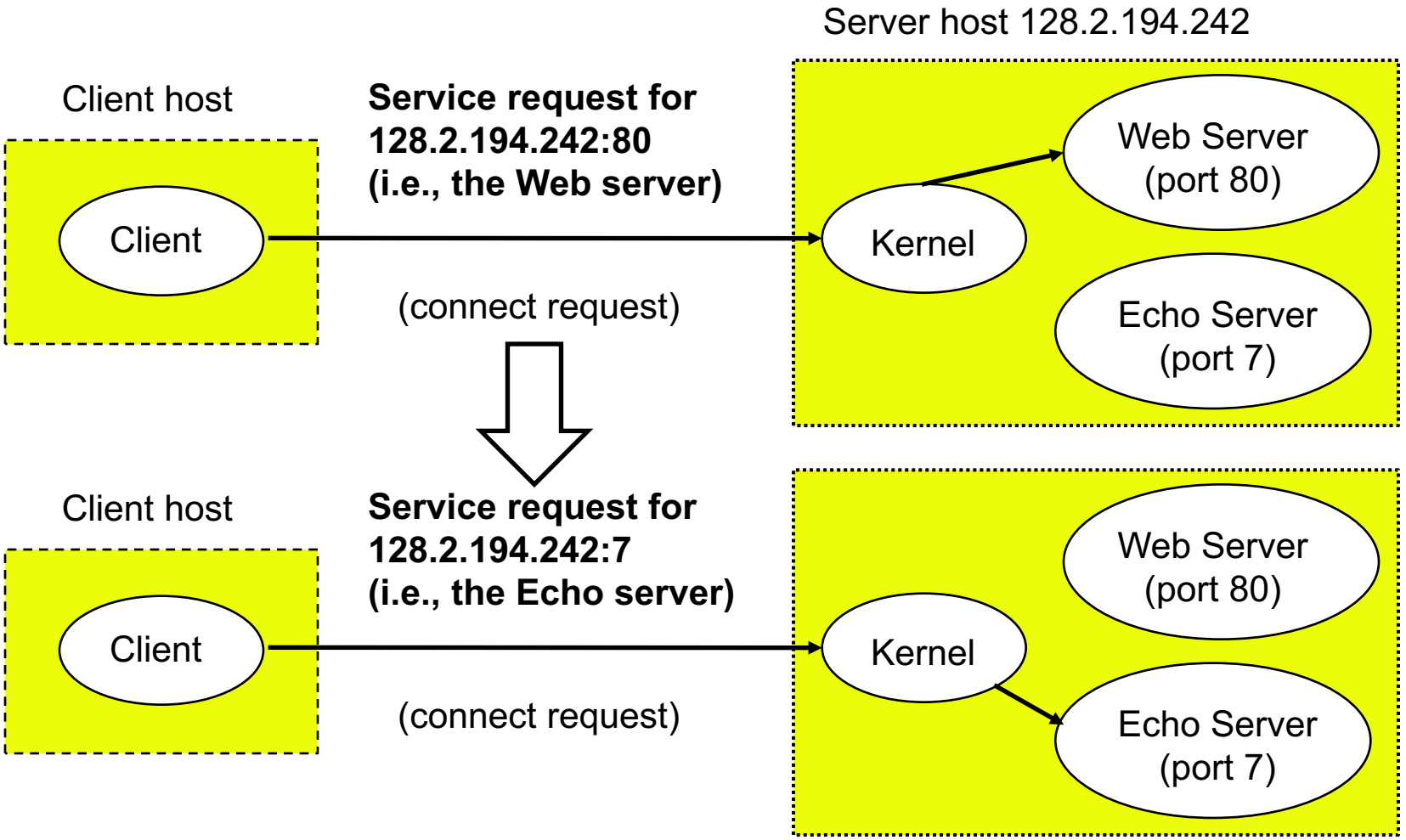
- The client usually specifies only remote participant's address and let's the system fill in the local information
- Whereas a server usually listens for messages on a well-known port
- A client does not care which port it uses for itself, the OS simply selects an unused one

Client-Serve Model with TCP

Once a connection is established, the application process invokes two operation

```
int send (int socket, char *msg, int msg_len,
                                        int flags)

int recv (int socket, char *buff, int buff_len,
                                          int flags)
```

Using Ports to Identify Services



[CMU 15-213]

Example Application: Client

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 5432
#define MAX_LINE 256

int main(int argc, char * argv[])
{
	FILE *fp;
	struct hostent *hp;
	struct sockaddr_in sin;
	char *host;
	char buf[MAX_LINE];
	int s;
	int len;
	if (argc==2) {
		host = argv[1];
	}
	else {
		fprintf(stderr, "usage: simplex-talk host\n");
	exit(1);
	}
```

Example Application: Client

```
        /* translate host name into peer's IP address */
        hp = gethostbyname(host);
        if (!hp) {
                fprintf(stderr, "simplex-talk: unknown host: %s\n", host);
                exit(1);
        }
        /* build address data structure */
        bzero((char *)&sin, sizeof(sin));
        sin.sin_family = AF_INET;
        bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
        sin.sin_port = htons(SERVER_PORT);
        /* active open */
        if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
                perror("simplex-talk: socket");
                exit(1);
        }
        if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
                perror("simplex-talk: connect");
                close(s);
                exit(1);
        }
        /* main loop: get and send lines of text */
        while (fgets(buf, sizeof(buf), stdin)) {
                buf[MAX_LINE-1] = '\0';
                len = strlen(buf) + 1;
                send(s, buf, len, 0);
        }
}
```

Example Application: Server

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#define SERVER_PORT 5432
#define MAX_PENDING 5
#define MAX_LINE 256

int main()
{
        struct sockaddr_in sin;
        char buf[MAX_LINE];
        int len;
        int s, new_s;
        /* build address data structure */
        bzero((char *)&sin, sizeof(sin));
        sin.sin_family = AF_INET;
        sin.sin_addr.s_addr = INADDR_ANY;
        sin.sin_port = htons(SERVER_PORT);

        /* setup passive open */
        if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
                perror("simplex-talk: socket");
                exit(1);
        }
```

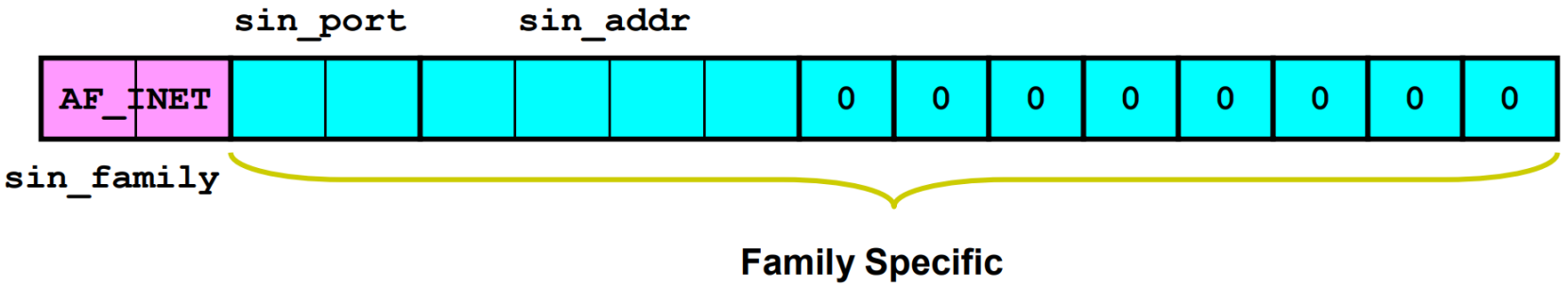
Example Application: Server

```
    if ((bind(s, (struct sockaddr *)&sin, sizeof(sin))) < 0) {
            perror("simplex-talk: bind");
            exit(1);
    }
    listen(s, MAX_PENDING);
    /* wait for connection, then receive and print text */
    while(1) {
            if ((new_s = accept(s, (struct sockaddr *)&sin, &len)) < 0) {
            perror("simplex-talk: accept");
            exit(1);
    }
    while (len = recv(new_s, buf, sizeof(buf), 0))
            fputs(buf, stdout);
            close(new_s);
    }
}
```

Socket Address Structs

- Internet-specific socket address

```
#include <netinit/in.h>

struct sockaddr_in {
    unsigned short sin_family; /* address family (always AF_INET)*/
    unsigned short sin_port; /* port num in network byte order */
    struct in_addr sin_addr  /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```



Big and Little Endian

- Describe the order in which a sequence of bytes is stored in memory
- Big Endian Byte Order
  - The **most significant** byte (the "big end") of the data is placed at the byte with the lowest address
  - IBM's 370 mainframes, most RISC-based computers, TCP/IP
  - **Network byte order in TCP/IP**

- Little Endian Byte Order
  - The **least significant** byte (the "little end") of the data is placed at the byte with the lowest address
  - Intel processors, DEC Alphas

Big and Little Endian

32-bit unsigned integer: 0x12345678		
Memory Address	Big-Endian Byte Order	Little-Endian Byte Order
1000	12	78
1001	34	56
1002	56	34
1003	78	12

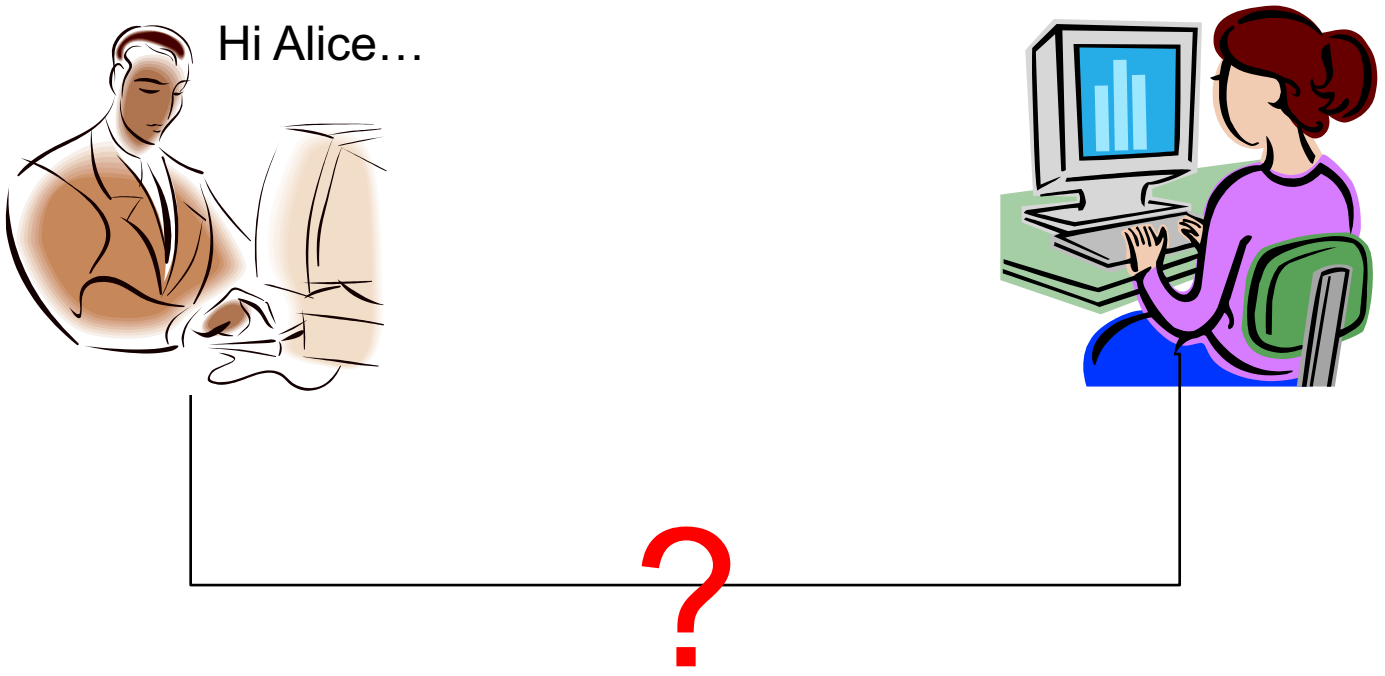
Big and Little Endian

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    short int a = 0x1234;
    char *p = (char *)&a;

    printf("p=%#hhx\n", *p);

    if (*p == 0x34)
        printf("little endian\n");
    else if (*p == 0x12)
        printf("big endian\n");
    else
printf("unknown endian\n");
    return 0;
}
```

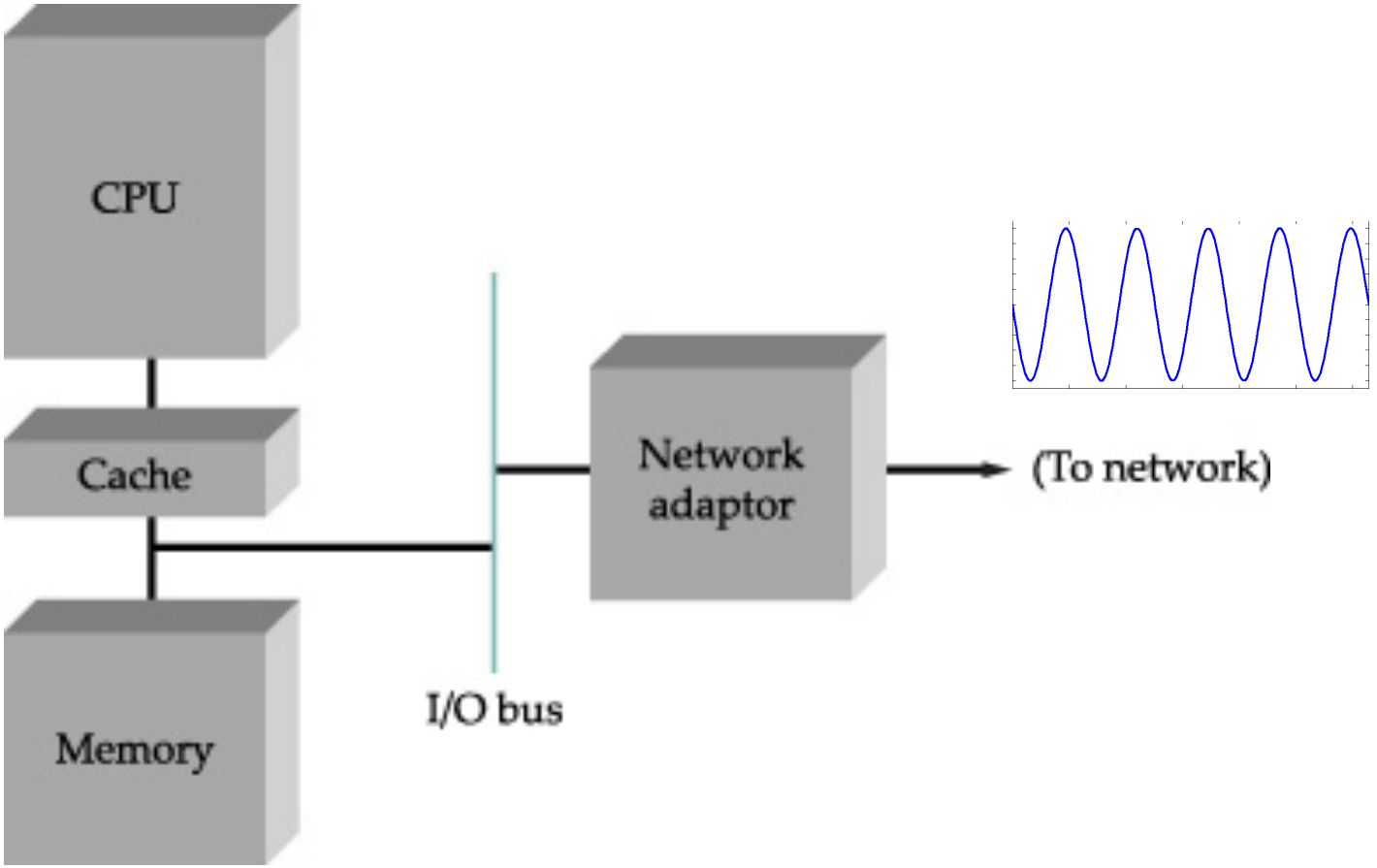
Overview

- Application Programming Interface
- Hardware and physical layer
  - Nuts and bolts of networking
  - Nodes
  - Links
    - Bandwidth, latency, throughput, delay-bandwidth product
    - Physical links

The simplest network is one link plus two nodes

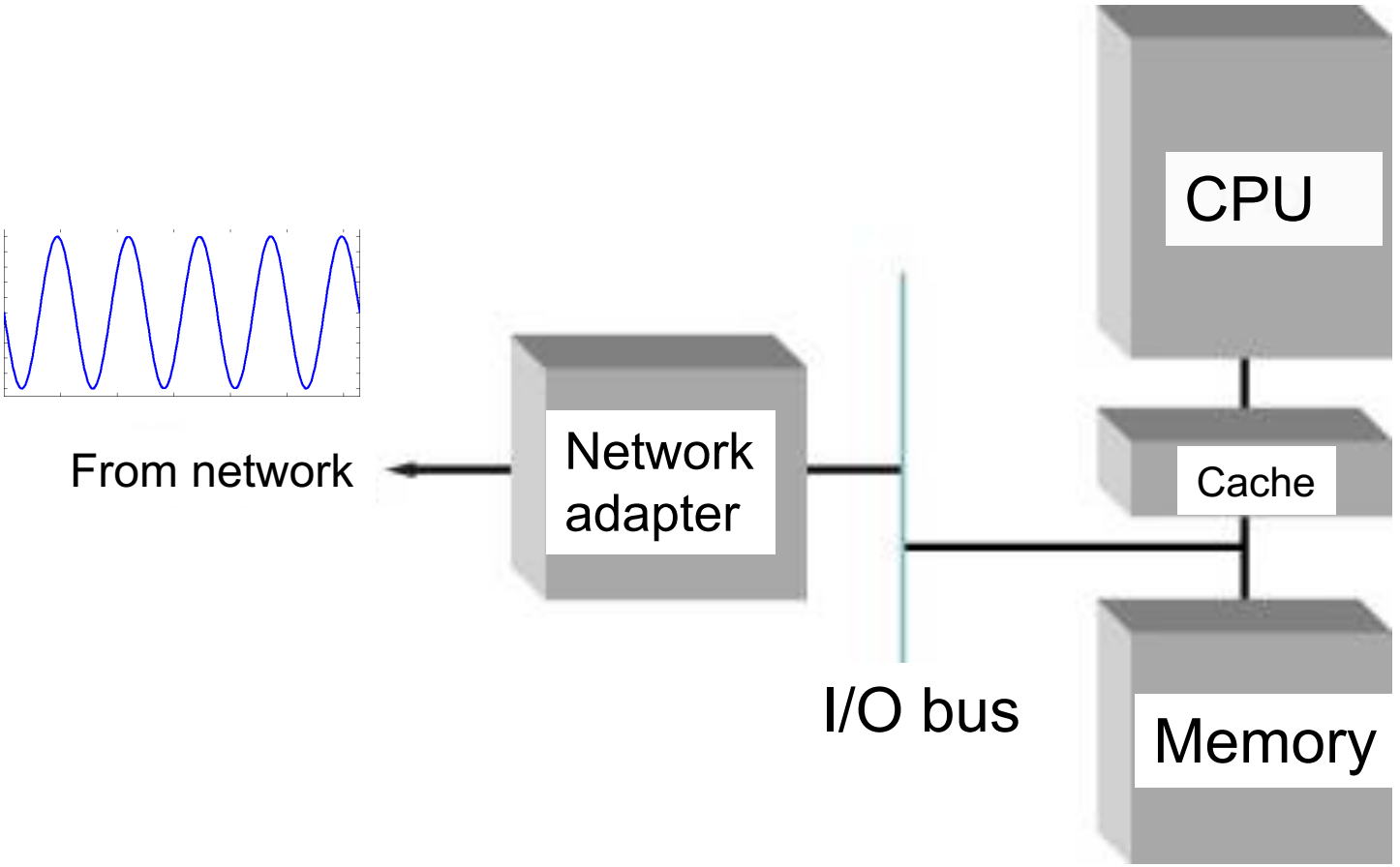


Sender side



Hi Alice

Receiver side



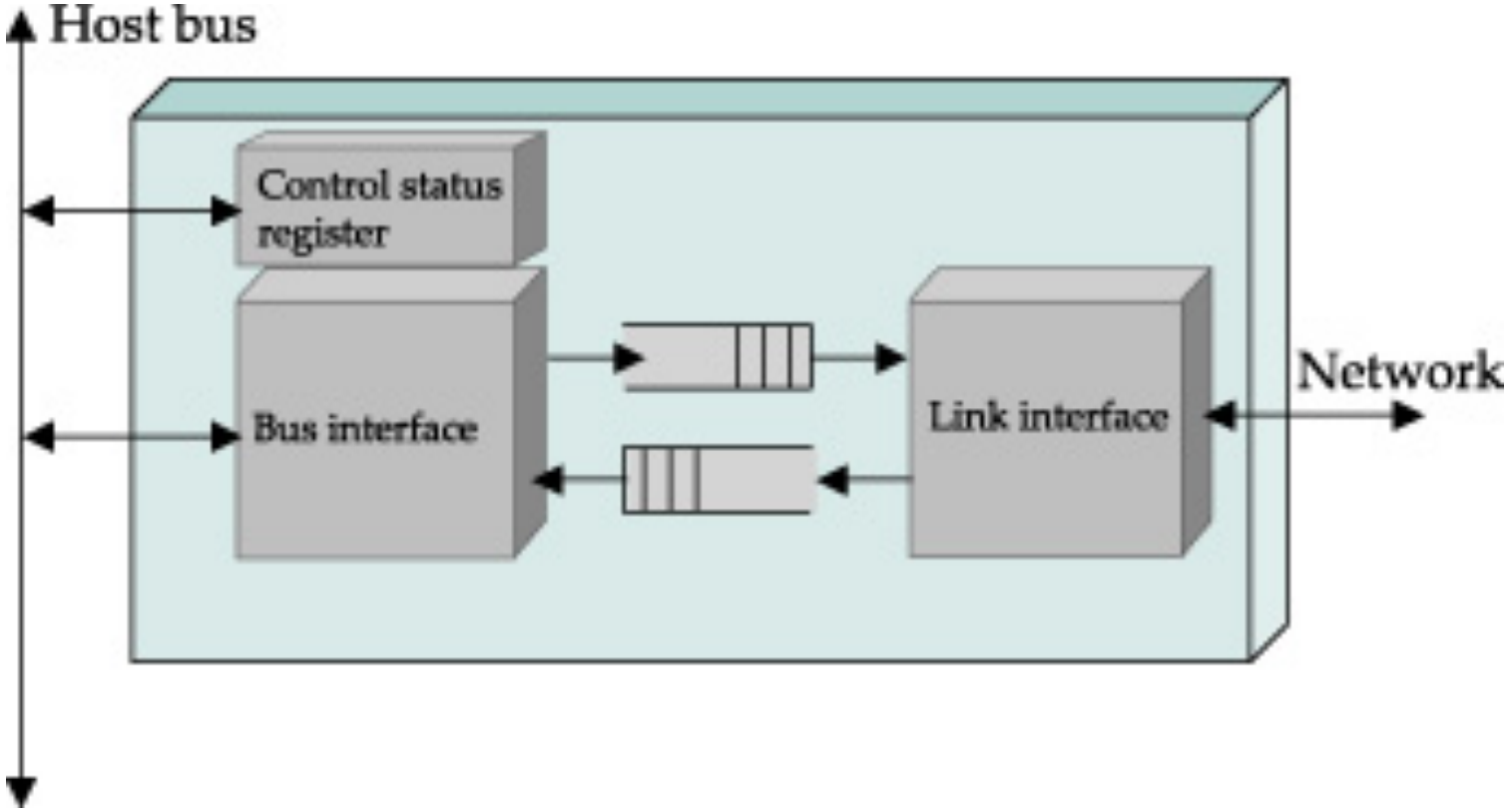
What actually happened

- On the sender side
  - Payload (“Hi Alice) is encapsulated into a packet
  - The packet is encapsulated into a frame (a block of data)
  - The frame is transmitted from main memory to the network adaptor
  - At the adaptor, the frame is encoded into a bit stream
  - The encoded bit stream is modulated into signals and put on the wire

The reverse process at the receiver

- On the receiver side
  - Signals demodulated into a bit stream
  - The bit stream decoded into a frame
  - The frame is delivered to a node's main memory
  - Payload is decapsulated from the frame

A typical adaptor

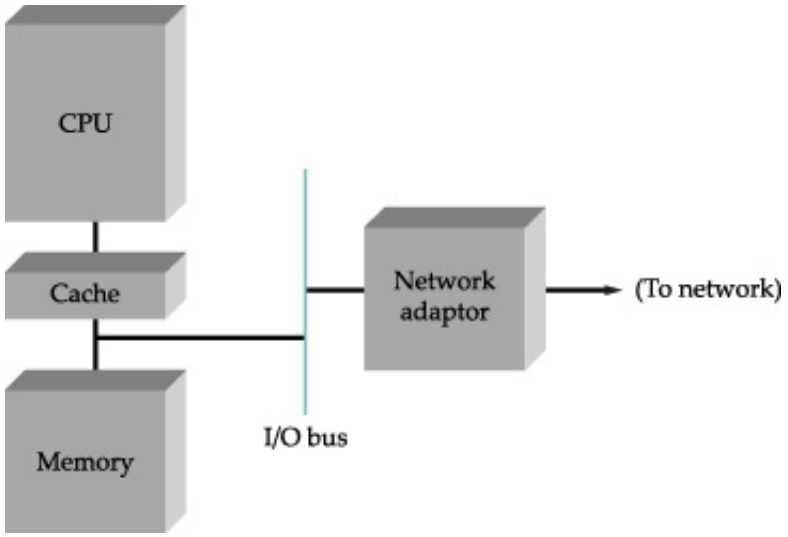


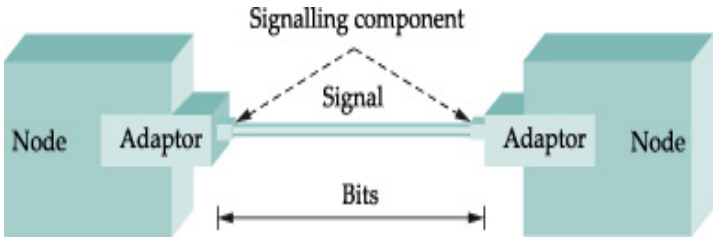
- A bus interface to talk to the host memory and CPU
- A link interface to talk to the network
- A CSR (Control Status Register) typically maps to a memory location
  - A device writes to CSR to send/receive data
  - Reads from CSR to learn the state
  - Adapter interrupts the host when receiving a frame

DMA and programmed I/O

- Direct Memory Access
  - Adaptor directly reads and writes the host memory without CPU involvement
- PIO
  - CPU moves data

Put bits on the wire

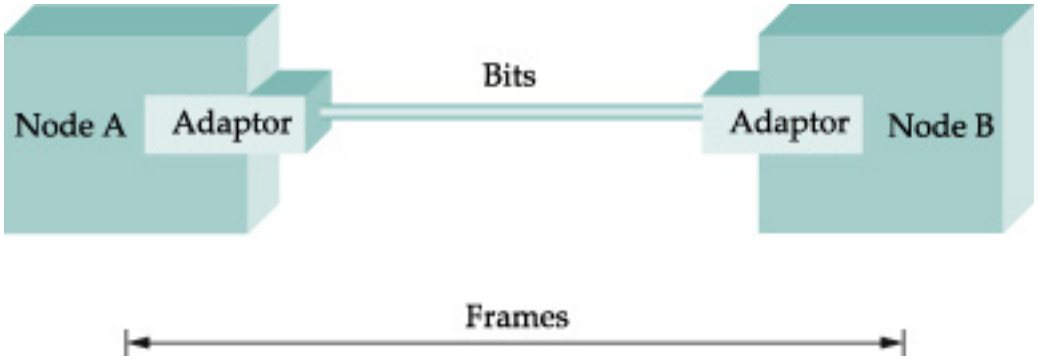




- Each node (e.g. a PC) connects to a network via a network adaptor.
- The adaptor delivers data between a node's memory and the network.
- A device driver is the program running inside the node that manages the above task.

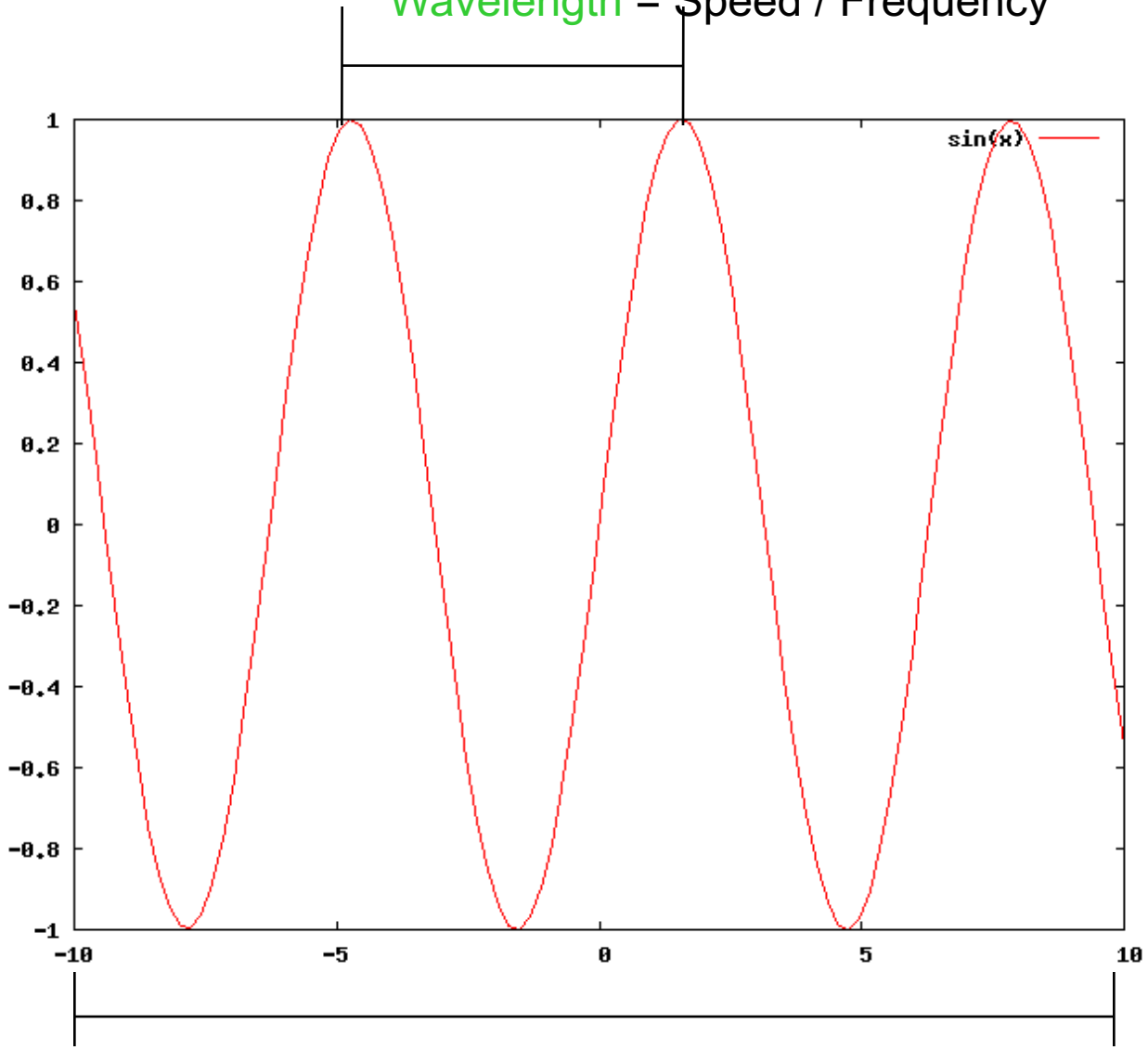
- At one end, a network adaptor encodes and modulates a bit into signals on a physical link.
- At the other end, a network adaptor reads the signals on a physical link and converts it back to a bit.

Framing



- Signals always present on a link: how to determine the start/end of a transmission?
  - Data are embedded into blocks of data called frames
  - Framing determines where the frame begins and ends is the central task of a network adaptor

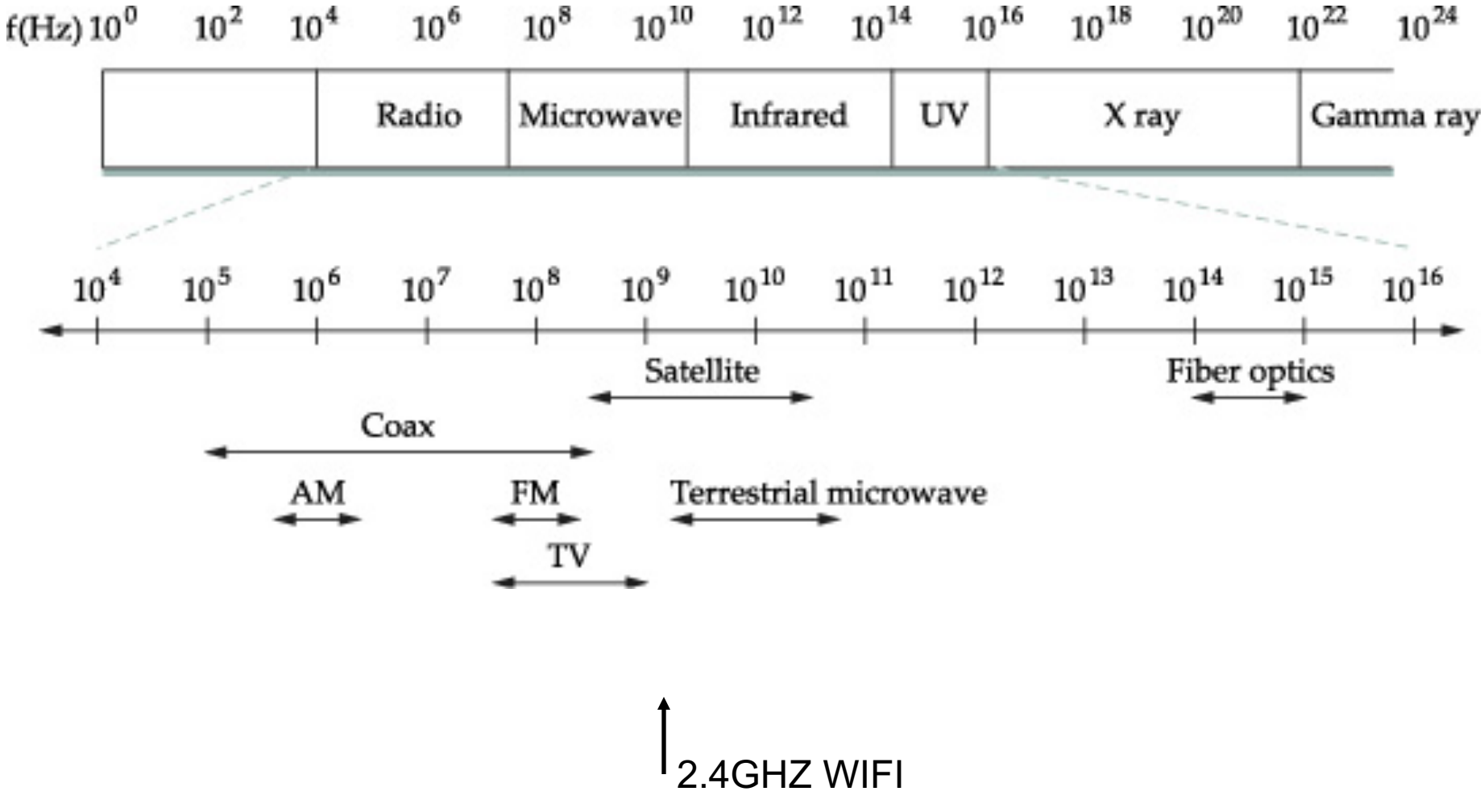
Wavelength = Speed / Frequency



Speed = how fast it travels in unit time

Frequency = how many cycles it goes through in unit time

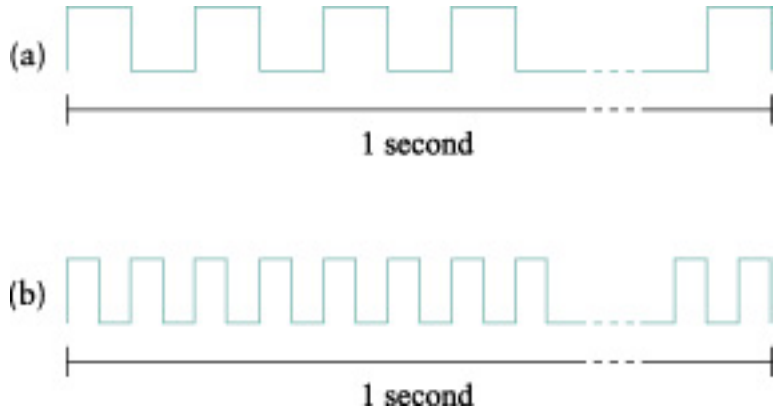
Electromagnetic spectrum



Full-duplex and half-duplex

- How many bit streams can be encoded on it
- One: then nodes connected to the link must share access to the link
  - Computer bus
- Full-duplex: one in each direction on a point-to-point link
- Half-duplex: two end points take turns to use it

Bandwidth



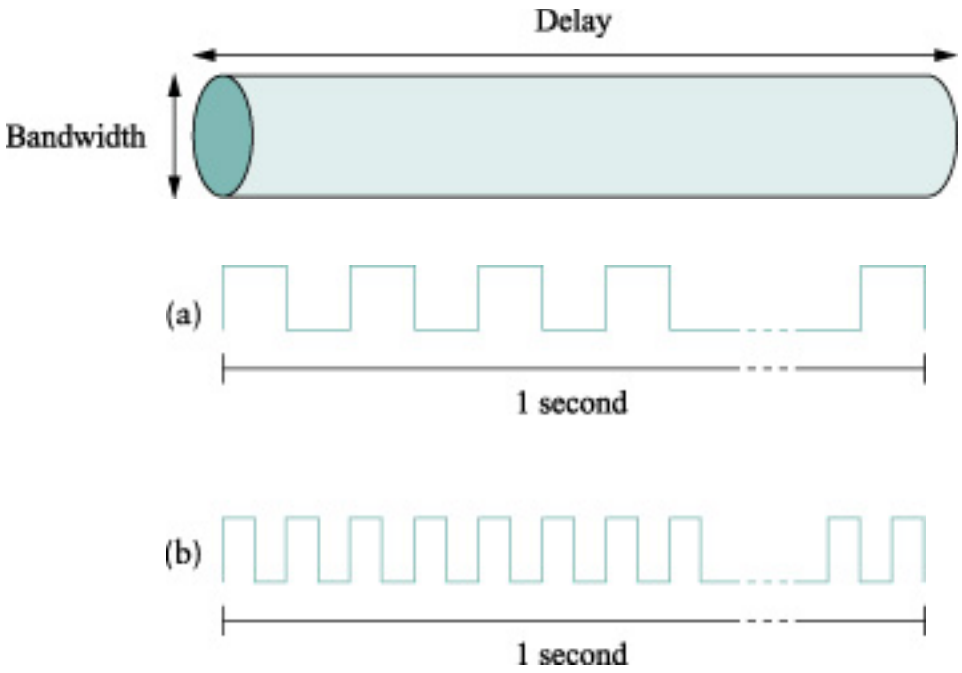
- Bandwidth is a measure of the width of a frequency band. E.g., a telephone line supports a frequency band 300-3300hz has a bandwidth of 3000 hz

- Bandwidth of a link normally refers to the number of bits it can transmit in a unit time
  - A second of time as distance
  - Each bit as a pulse of width

Propagation delay

- How long does it take for one bit to travel from one end of link to the other?
- Length Of Link / Speed Of WaveInMedium
- 2500m of copper: $2500/(2/3 * 3*10^8) = 12.5\mu S$

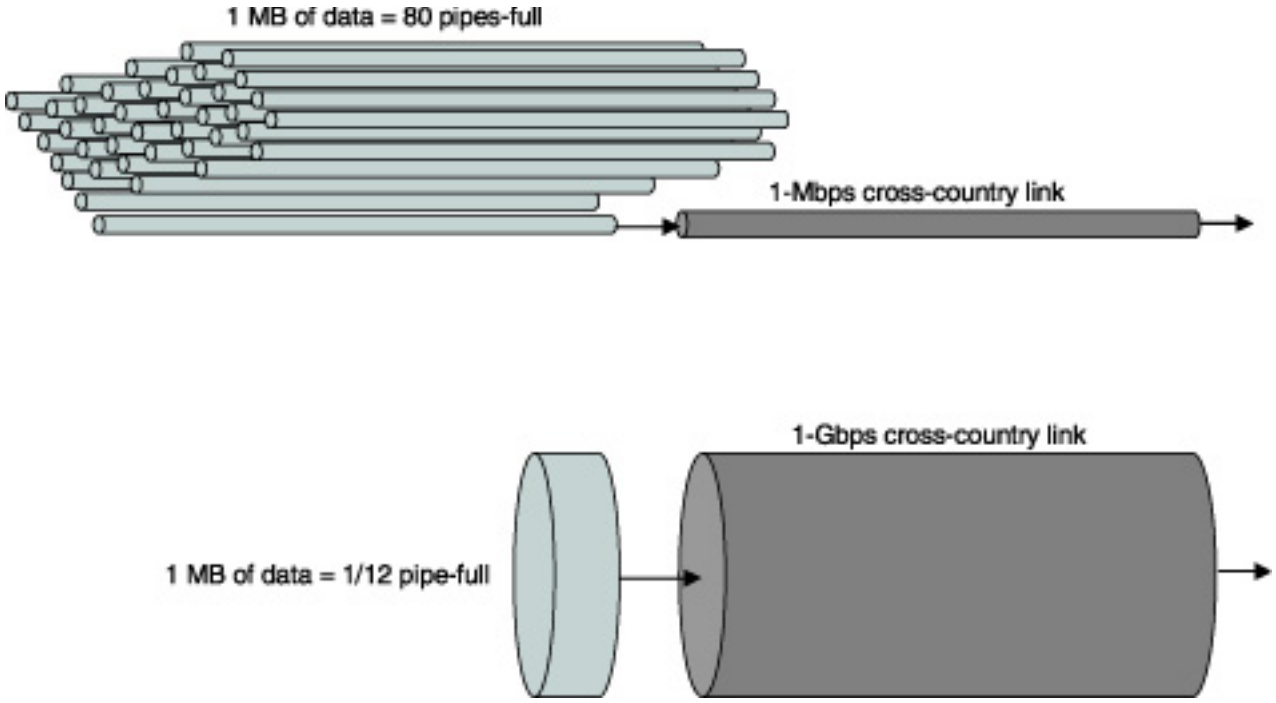
Delay x bandwidth product



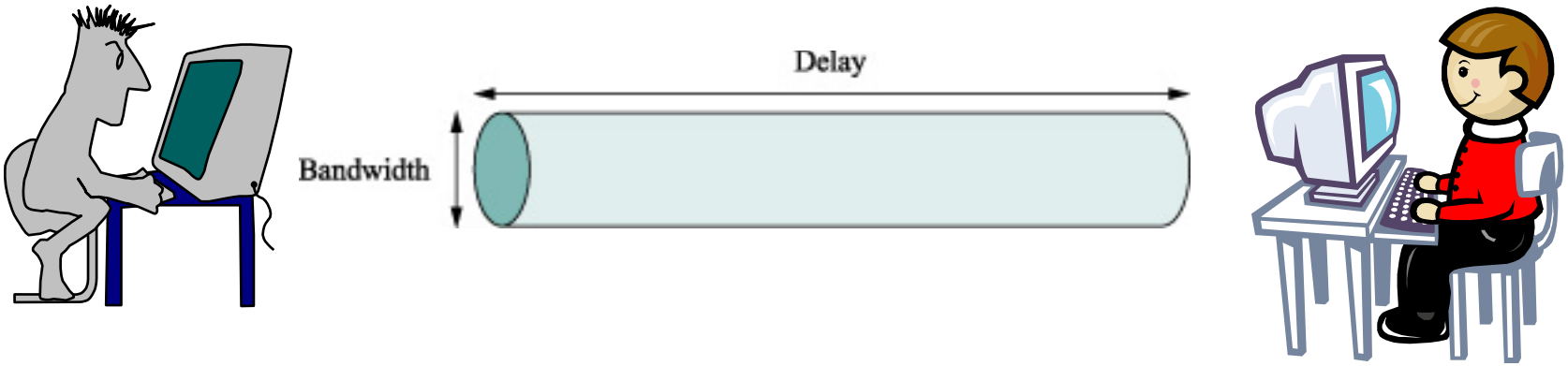
Which has higher bandwidth?

- Measure the volume of a "pipe": how many bits can the sender sends before the receiver receives the first bit
- An important concept when constructing high-speed networks
- When a "pipe" is full, no more bits can be pumped into it

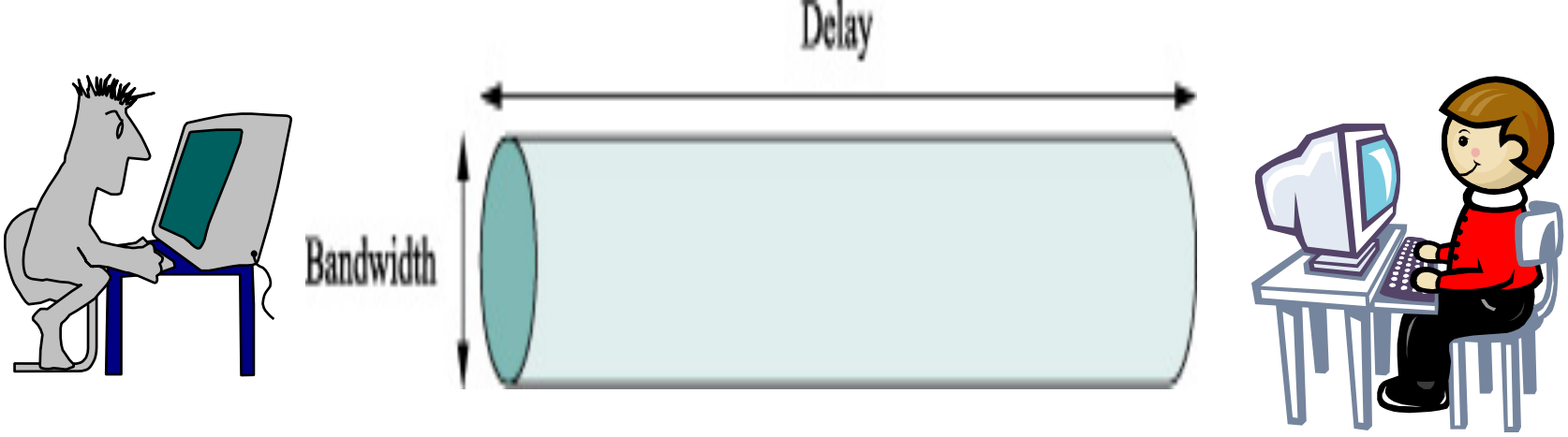
High speed versus low speed links



- A high speed link can send more bits in a unit time than a low speed link
- 1MB of data, 100ms one-way delay
- How long will it take to send over different speed of links?



- 1Mbps, 100ms, 1MB data
- Delay * Bandwidth = 100Kb
- 1MB/100Kb = 80 pipes of data
- 80 * 100ms + 100ms = 8.1s
- Transfer time = propagation time + transmission time (serialization) + queuing time



- 1Gbps, 100ms, 1MB data
- Delay * Bandwidth = 100Mb
- 1MB/100Mb = 0.08 pipe of data
- TransferTime = 0.08 * 100ms + 100ms = 108ms
- Throughput = TransferSize/TransferTime = 1MB/108ms = 74.1Mbps
  - Why is it less than 1Gbps?

Commonly Used Physical Links

- Different links have different transmission ranges
  - Signal attenuation
- Cables
  - Connect computers in the same building
- Leased lines
  - Lease a dedicated line to connect far-away nodes from telephone companies

Cables

- CAT-5: twisted pair
- Coaxial: thick and thin
- Fiber

Fiber Cable Ethernet 40GbE

10BASE2 cable, thin-net

200m

10Base4, thick-net
500m

CAT-5

Leased lines

- Tx series speed: multiple of 64Kpbs
  - Copper-based transmission
    - DS-1 (T1): 1,544, 24*64kpbs
    - DS-2 (T2): 6,312, 96*64kps
    - DS-3 (T3): 44,736, 672*64kps
- OC-N series speed: multiple of OC-1
  - Optical fiber based transmission
    - OC-1: 51.840 Mbps
    - OC-3: 155.250 Mbps
    - OC-12: 622.080 Mbps

Last mile links

- Wired links
  - POTS: 28.8-56Kbps (Plain old telephone service)
  - ISDN: 64-128Kbps (Integrated Services Digital Network)
  - xDSL: 128Kbps-100Mbps (over telephone lines)
    - Digital Subscriber Line
  - CATV: 1-40Mpbs (shared, over TV cables)
    - Newer standards increase to 1Gbps
  - FTTH (fibre to the home): 50Mbps-1Gbps

- Wireless links
  - Wifi, WiMax, Bluetooth, ZigBee, 4G, 5G…
  - Data rates: 4G (20Mbps), 5G (10Gbps)

Wireless links

- Wireless links transmit electromagnetic signals through space
  - Used also by cellular networks, TV networks, satellite networks etc.
- Shared media
  - Divided by frequency and space
- FCC determines who can use a spectrum in a geographic area, ie, “licensing”
  - Auction is used to determine the allocation
  - Expensive to become a cellular carrier
- Unlicensed spectrum
  - WiFi, Bluetooth, Infrared

Summary

- Application Programming Interface
  - sockets, sockets operations
  - ports
- Hardware and physical layer
  - Links
    - Bandwidth, latency, throughput, delay-bandwidth product
    - Types of Physical links

END-TO-END ARGUMENTS IN SYSTEM DESIGN

By J.H. Saltzer, D.P. Reed and D.D. Clark

End-to-End Argument

- Extremely influential
- “…functions placed at the lower levels may be *redundant* or of *little value* when compared to the cost of providing them at the lower level…”
- “…sometimes an *incomplete* version of the function provided by the communication system (lower levels) may be useful as a *performance enhancement*…”

The counter argument

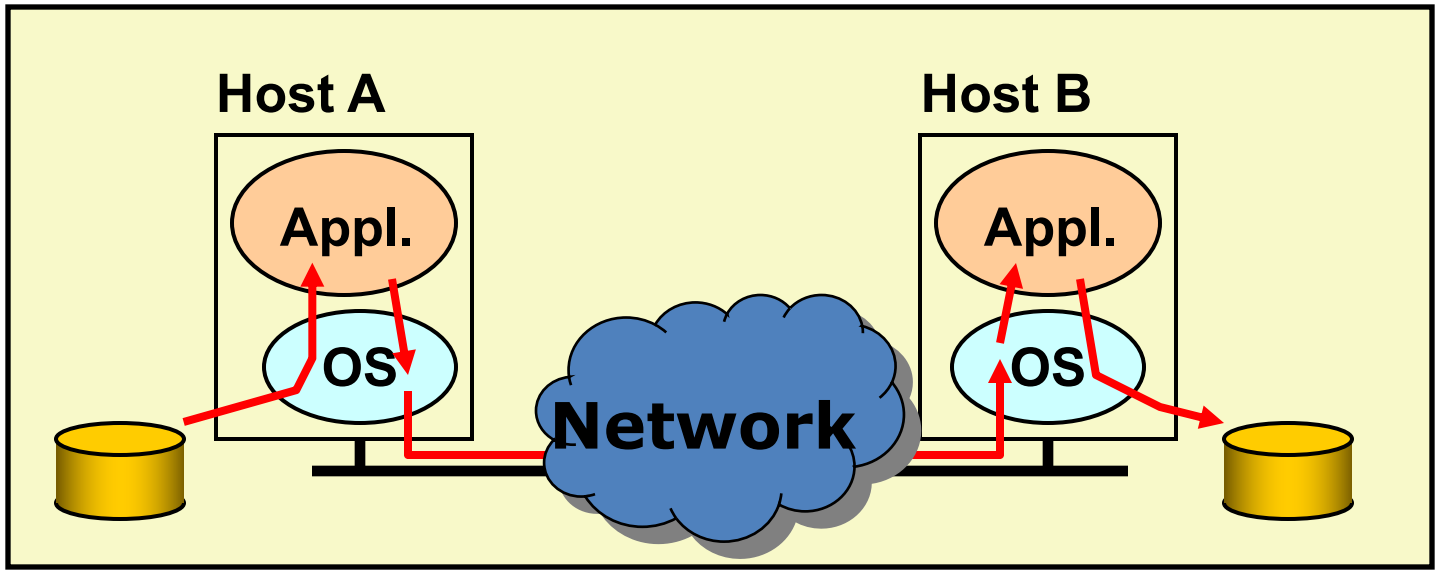
- Modularity argument:
  - It is tempting to implement functions at lower layers so that higher level applications can reuse them

- The end-to-end argument:
  - “The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of communication.”
  - “Centrally-provided versions of each of those functions will be incomplete for some applications, and those applications will find it easier to build their own version of the functions starting with datagrams.”

Techniques used by the authors

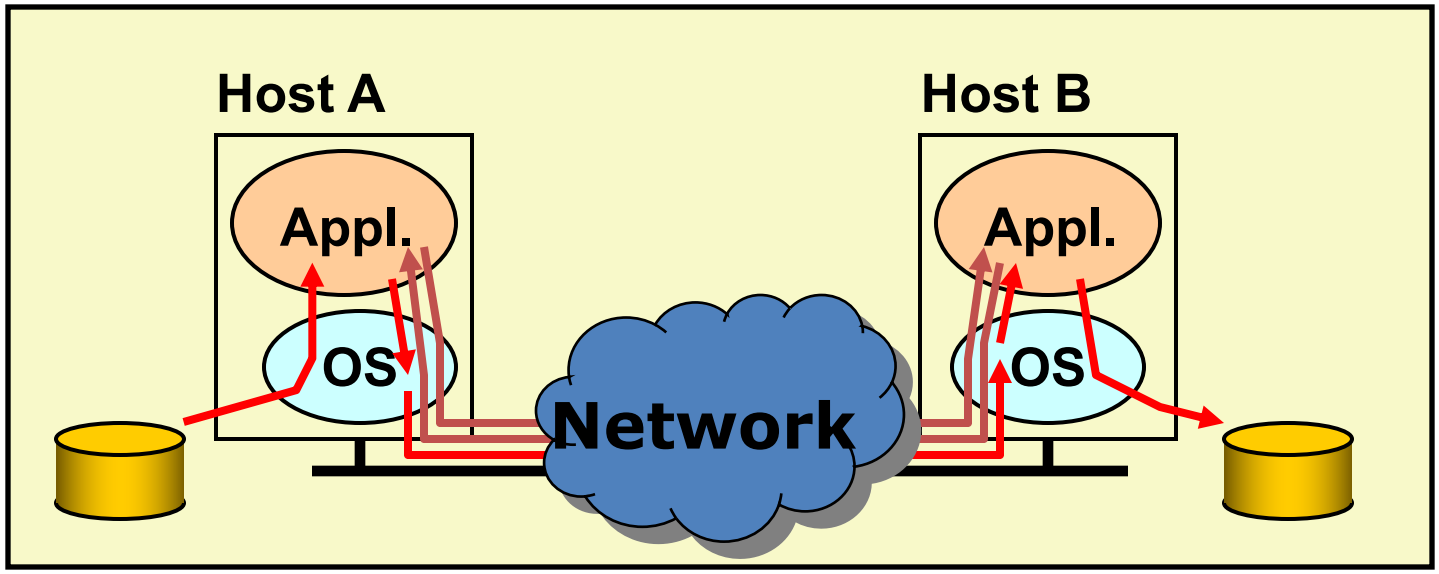
- The authors made their argument by analyzing examples
  - Reliable file transfer
  - Delivery guarantees
  - Secure data transmission
  - Duplicate message suppression
  - FIFO
  - Transaction management
  - *Can you think of more examples to argue for or against the end-to-end argument*?
- Can be applied generally to system design

Example: Reliable File Transfer



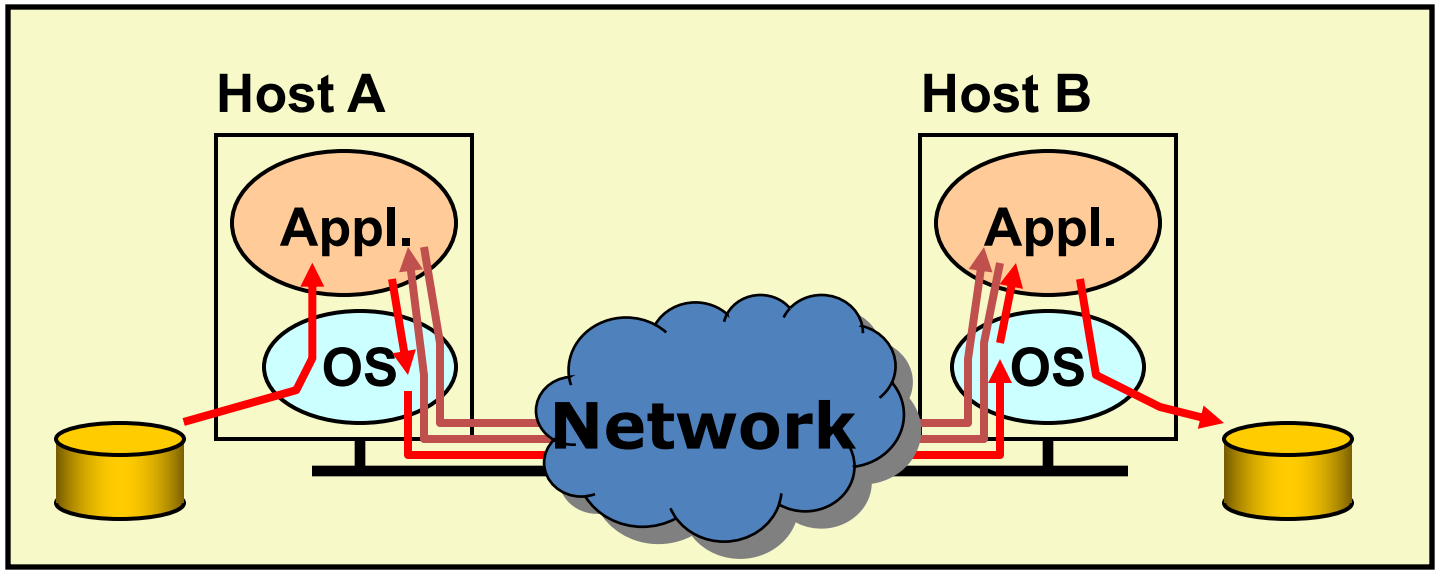
- Solution 1: make each step reliable, and then concatenate them
  - Uneconomical if each step has small error probability

Example: Reliable File Transfer



- Solution 2: end-to-end check and retry
  - Correct and complete

Example: Reliable File Transfer



- An intermediate solution: the communication system provides internally, a guarantee of reliable data transmission, e.g., a hop-by-hop reliable protocol
  - Only reducing end-to-end retries
  - No effect on correctness

Question: should lower layer play a part in obtaining reliability?

- Answer: it depends
  - Example: extremely lossy link
    - One in a hundred packets will be corrupted
    - 1K packet size, 1M file size
    - Prob of no end-to-end retry: $(1\text{-}1/100)^{1000} \sim 4.3e\text{-}5$

Performance enhancement

- “put into reliability measures within the data communication system is seen to be an engineering tradeoff based on performance, rather than a requirement for correctness.”

Summary: End-to-End Arguments

- If the application can do it, don't do it at a lower layer -- anyway the application knows the best what it needs
  - add functionality in lower layers iff it is (1) used and improves performances of a large number of applications, and (2) does not hurt other applications
- Success story: Internet
  - a minimalist design