CompSci 356: Computer Network Architectures

Lecture 3: Network Architecture Examples and Lab 1

Xiaowei Yang
xwy@cs.duke.edu

Overview

- The Internet Architecture
- OSI Network Architecture

- Lab 1 Released
- Due: Jan 29, 11:59pm via Saikai

Network Architectures

- Many ways to build a network
- Use network architectures to characterize different ways of building a network
- The general blueprints that guide the design and implementation of networks are referred to as network architectures

Protocol standardization

- Standard bodies such as IETF govern procedures for introducing, validating, and approving protocols
  - The Internet protocol suite uses open standard

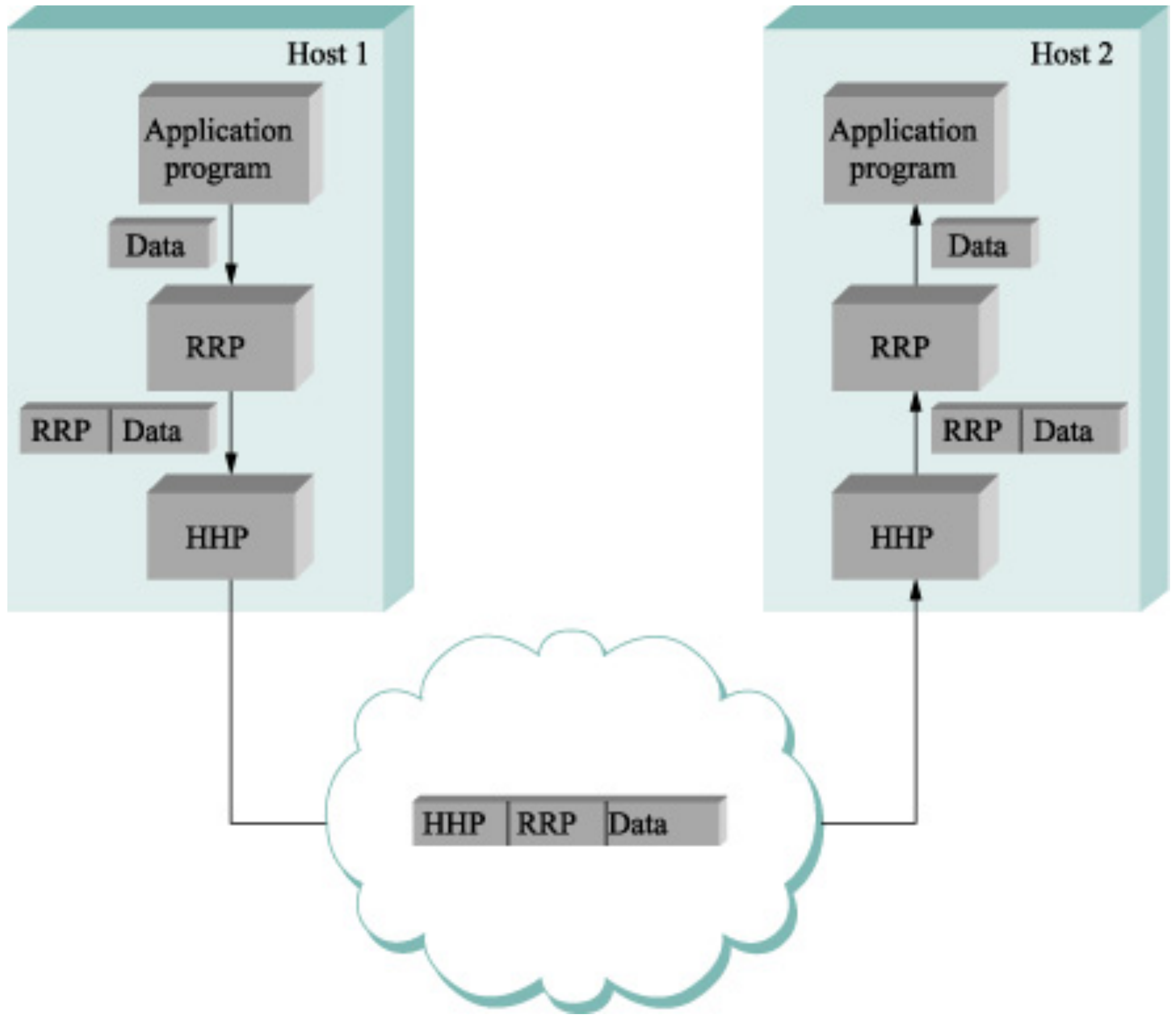
- Set of rules governing the form and content of a protocol graph are called a network architecture

We reject kings, presidents, and voting. We believe in rough consensus and running code

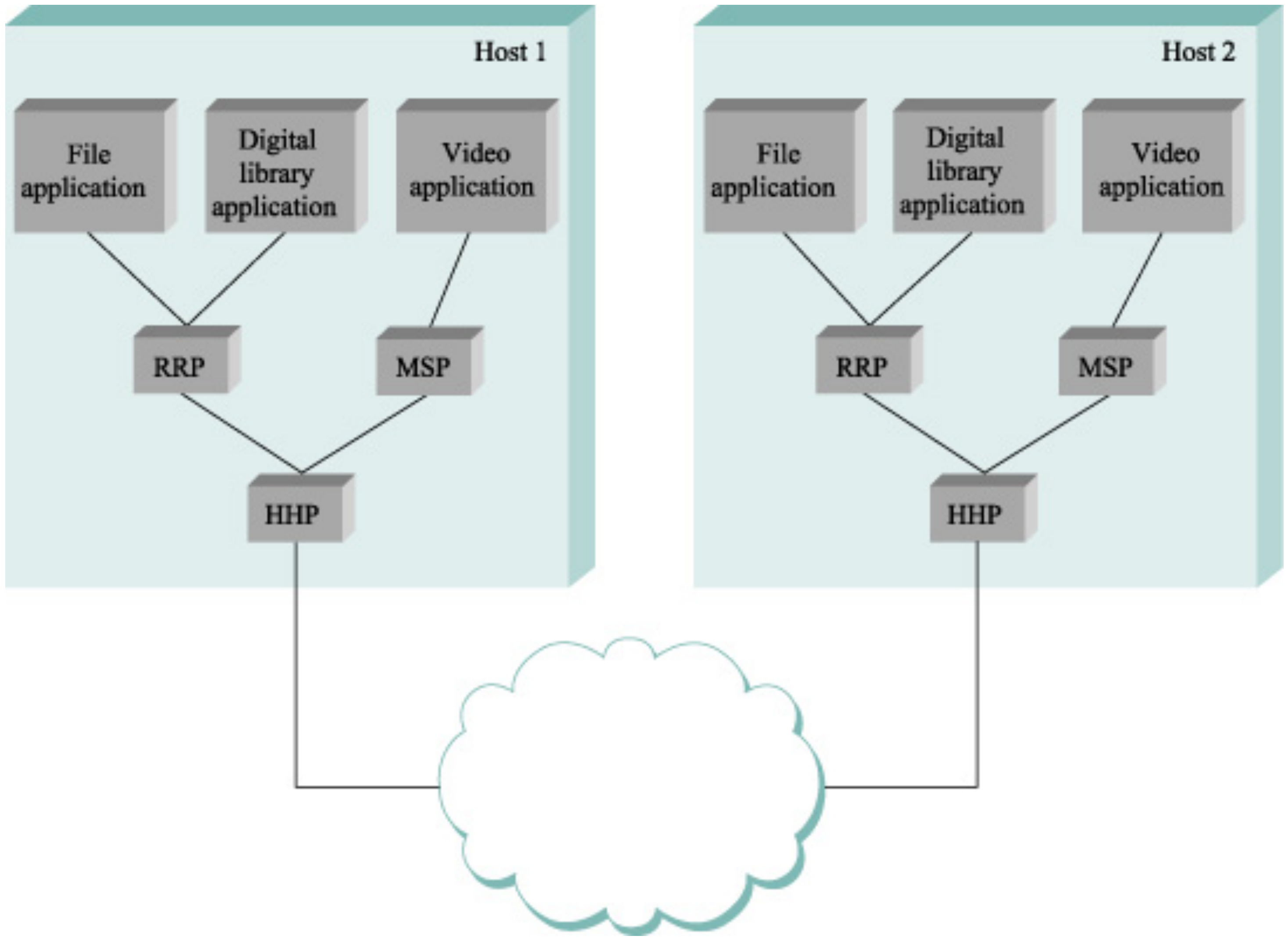
- David Clark

Encapsulation

- Upper layer sends a message using the service interface
- A header, a small data structure, to add information for peer-to-peer communication, is attached to the front message
  - Sometimes a trailer is added to the end
- Message is called payload or data
- This process is called encapsulation


Host 1
Application program
Data
RRP
RRP | Data
HHP
Host 2
Application program
Data
RRP
RRP | Data
HHP
HHP | RRP | Data

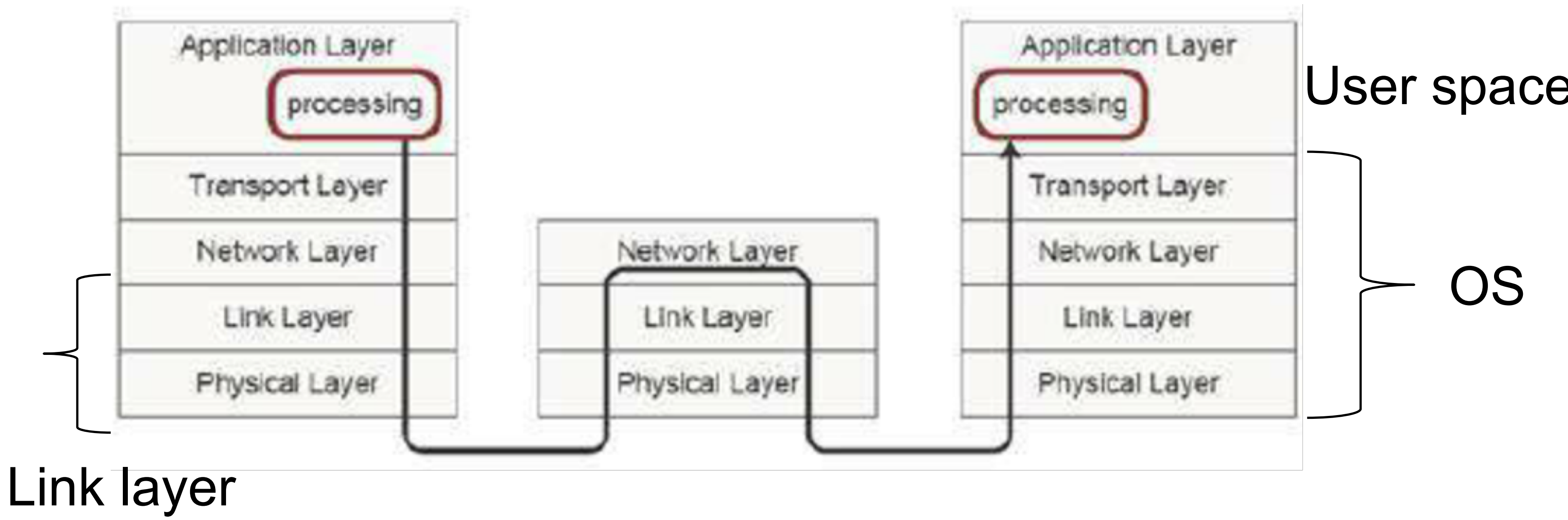
Multiplexing & Demultiplexing



- Same ideas apply up and down the protocol graph
- Header information is used for demultiplexing

Examples of Network Architectures

The Internet Protocol Suite



- The Internet architecture has four layers: Application, Transport, Network, and Data Link Layer (logical link layer, and physical link layer)
- Sending or receiving a packet from end systems (hosts) may involve actions of all four layers. Packet forwarding by (Routers) only involves the bottom three layers. By switches only involves the link layer.

Functions of the Layers

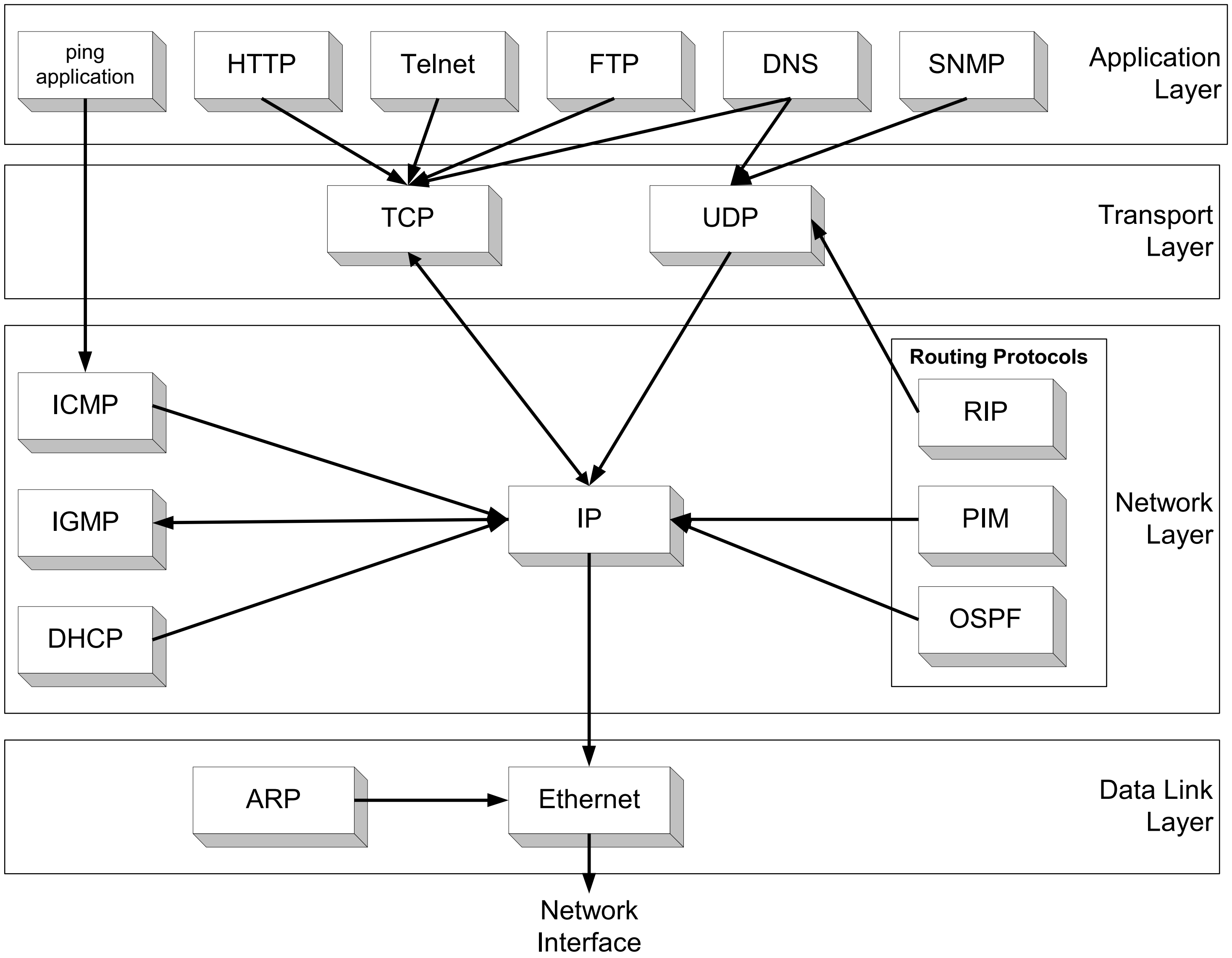
- Data Link Layer: (layer 2)
  - Service: Reliable transfer of frames over a link
    Media Access Control on a LAN
  - Functions: Framing, media access control, error checking

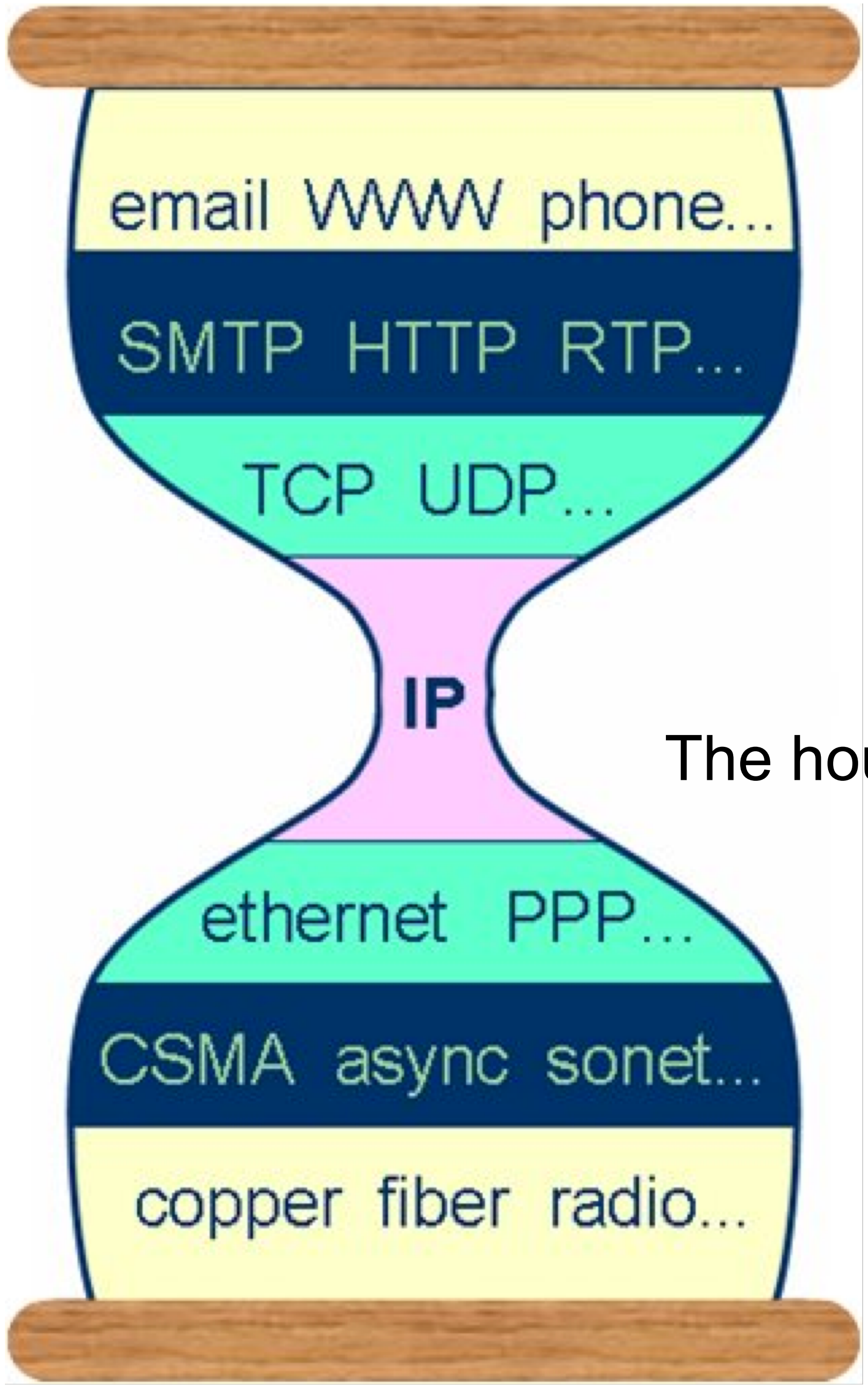
- Network Layer: (layer 3)
  - Service: Move packets from source host to destination host
  - Functions: Routing, addressing

- Transport Layer: (layer 4)
  - Service: Delivery of data between hosts
  - Functions: Connection establishment/termination, error control, flow control

- Application Layer:
  - Service: Application specific (delivery of email, retrieval of HTML documents, reliable transfer of file)
  - Functions: Application specific

Assignment of Protocols to Layers

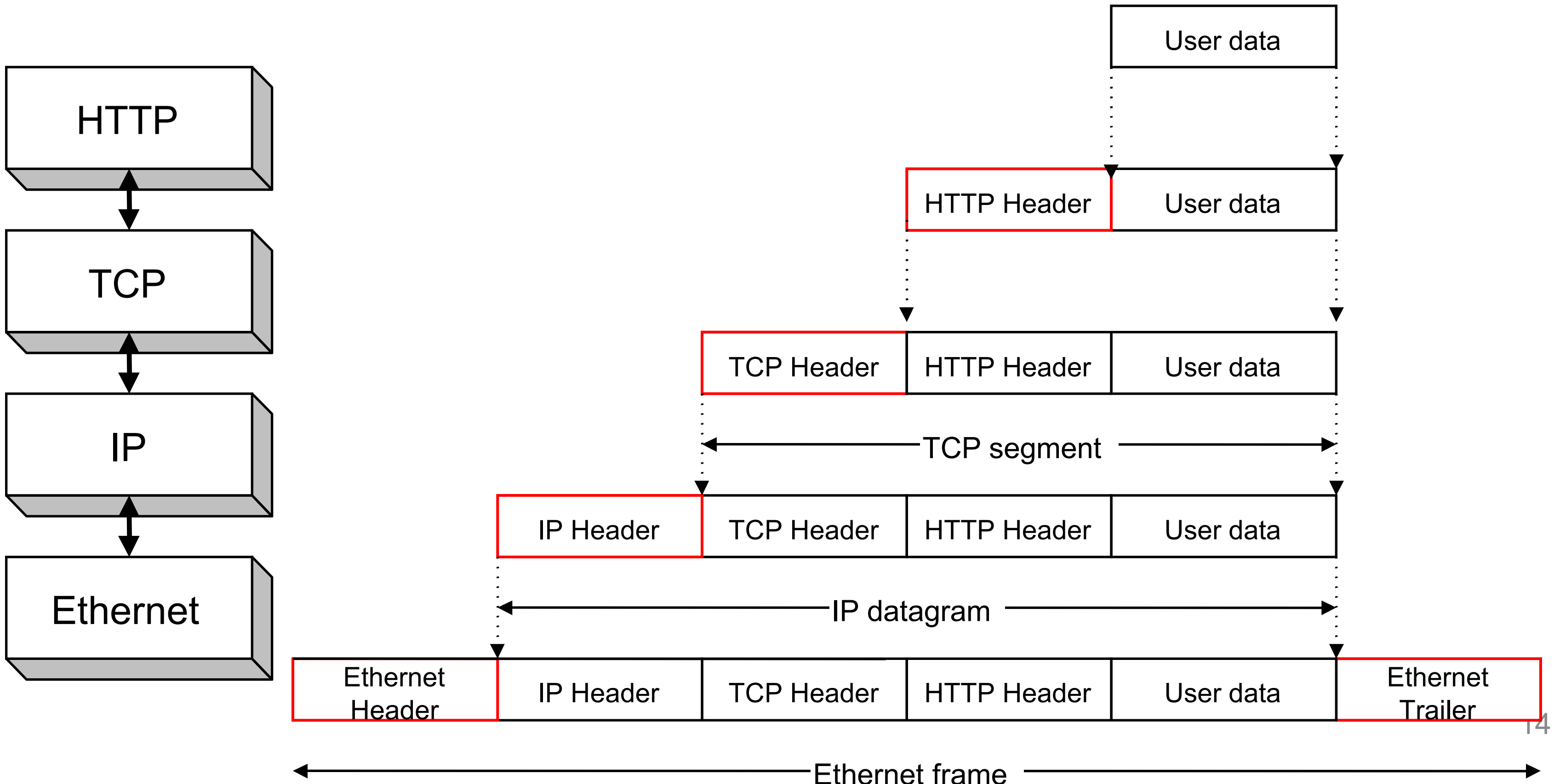




The hourglass model

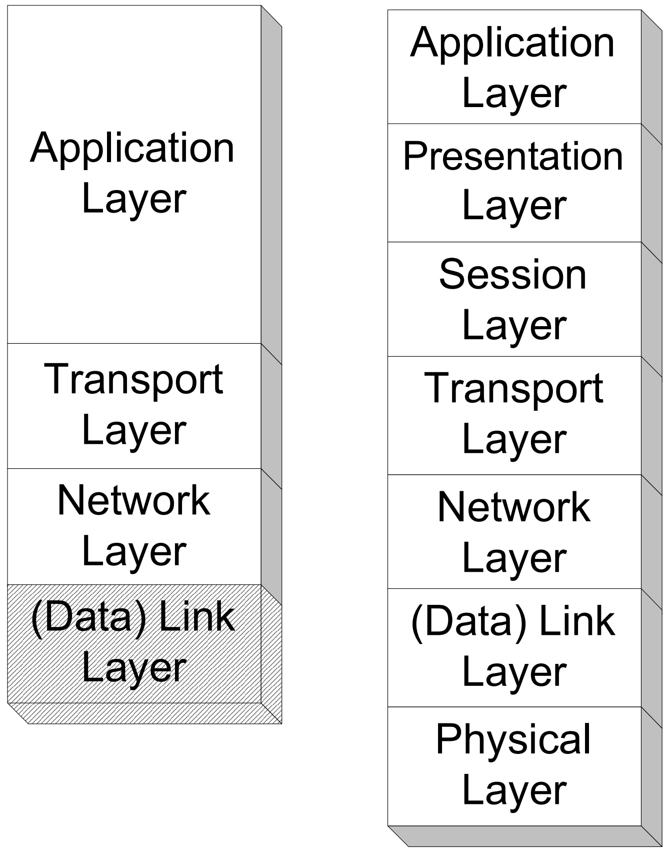
Use Encapsulation and Decapsulation to demultiplex

- Encapsulation: As data is moving down the protocol stack, each protocol is adding layer-specific control information.
- Decapsulation is the reverse process.



TCP/IP Suite vs OSI Reference Model

The TCP/IP protocol stack does not define the lower layers of a complete protocol stack



TCP/IP Suite

OSI Reference Model

- International Telecommunications Union (ITU) publishes protocol specs based on the OSI reference model
  - X dot series

- Physical layer: handles raw bits
- Data link layer: aggregate bits to frames. Network adaptors implement it
- Network layer: handles host-to-host packet delivery. Data units are called packets
- Transport: implements process channel. Data units are called messages
- Session layer: handles multiple transport streams belong to the same applications
- Presentation layer: data format, e.g., integer format, ASCII string or not
- Application layer: application specific protocols

Summary

- The design requirement of the Internet
- Network architectures that meet the design requirement
- New terms
  - Scalability, nodes, links, switches, routers, multiplexing/demultiplexing, circuit switching, packet switching, statistical multiplexing, layering, protocols, encapsulation/decapsulation, network archititetures

The History of the Internet

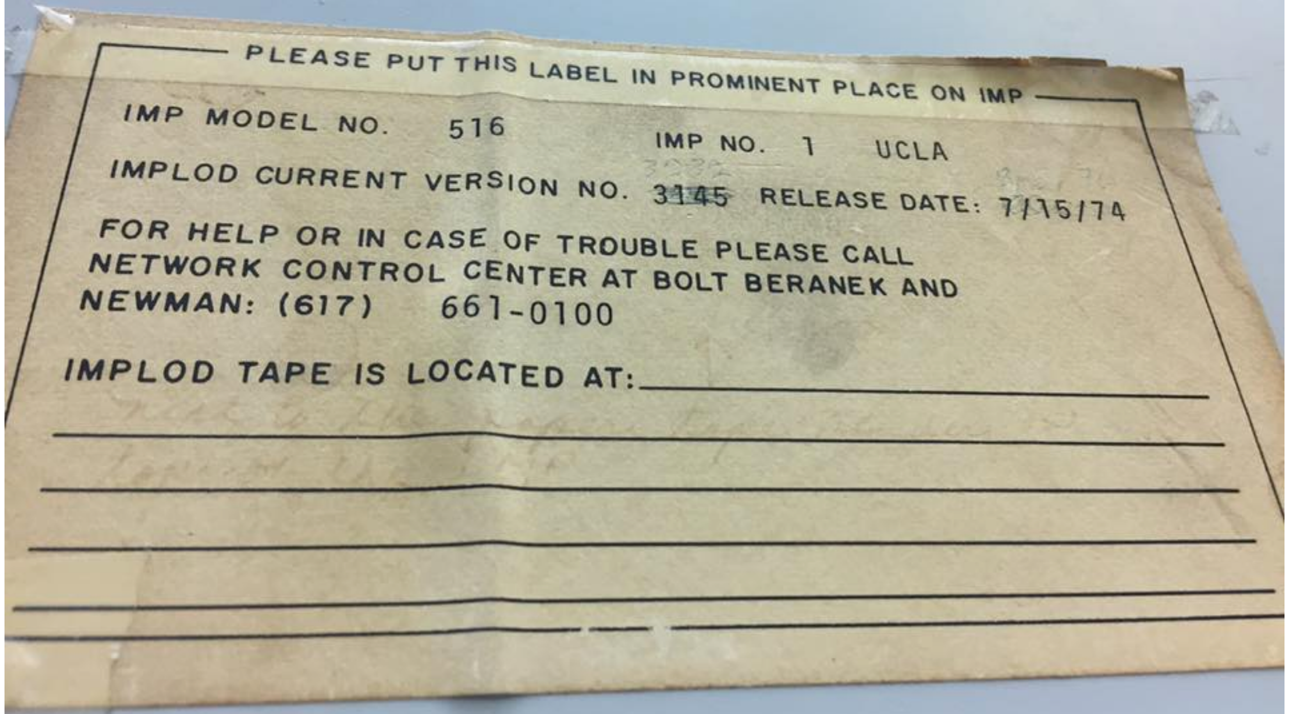
Internet History

1961-1972: Early packet-switching principles

- 1961: **Leonard** Kleinrock - queueing theory shows effectiveness of packet-switching
- 1964: Paul Baran - packet-switching in military nets
- 1967: ARPAnet conceived by Advanced Research Projects Agency
- 1969: first ARPAnet node operational
- 1972:
  - ARPAnet demonstrated publicly
  - NCP (Network Control Protocol) first host-host protocol
    - No TCP/IP yet
  - first e-mail program
  - ARPAnet has 15 nodes

https://www2.cs.duke.edu/courses/fall18/compsci514/slides/IMG_1342.MOV

INTERFACE
MESSAGE
PROCESSOR



PLEASE PUT THIS LABEL IN PROMINENT PLACE ON IMP

IMP MODEL NO. 516 IMP NO. 1 UCLA

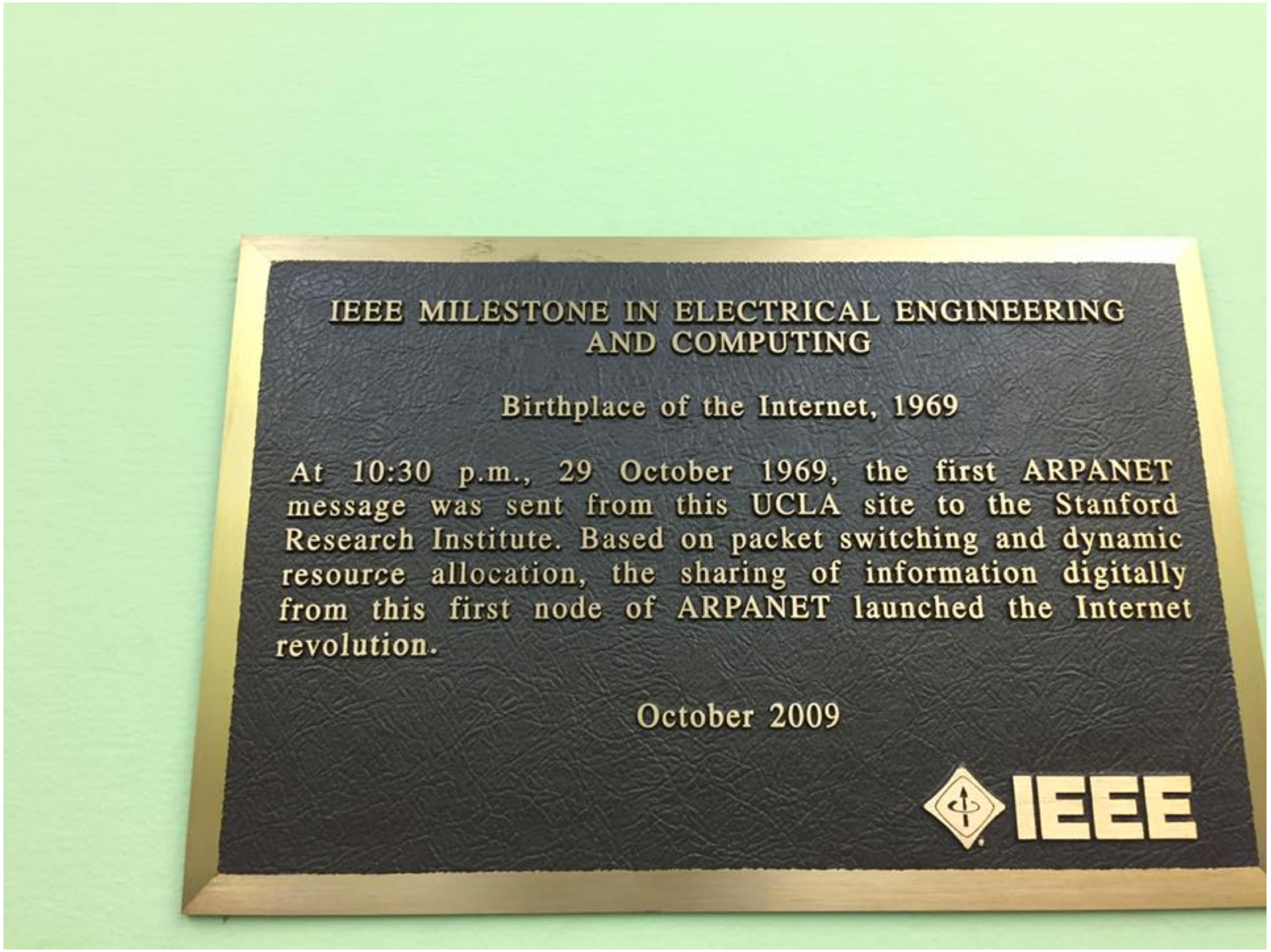
IMPLOD CURRENT VERSION NO. 3022 3145 RELEASE DATE: [illegible] 7/15/74

FOR HELP OR IN CASE OF TROUBLE PLEASE CALL NETWORK CONTROL CENTER AT BOLT BERANEK AND NEWMAN: (617) 661-0100

IMPLOD TAPE IS LOCATED AT:

Next to the paper tape reader [illegible]

Tape at [illegible]



IEEE MILESTONE IN ELECTRICAL ENGINEERING
AND COMPUTING

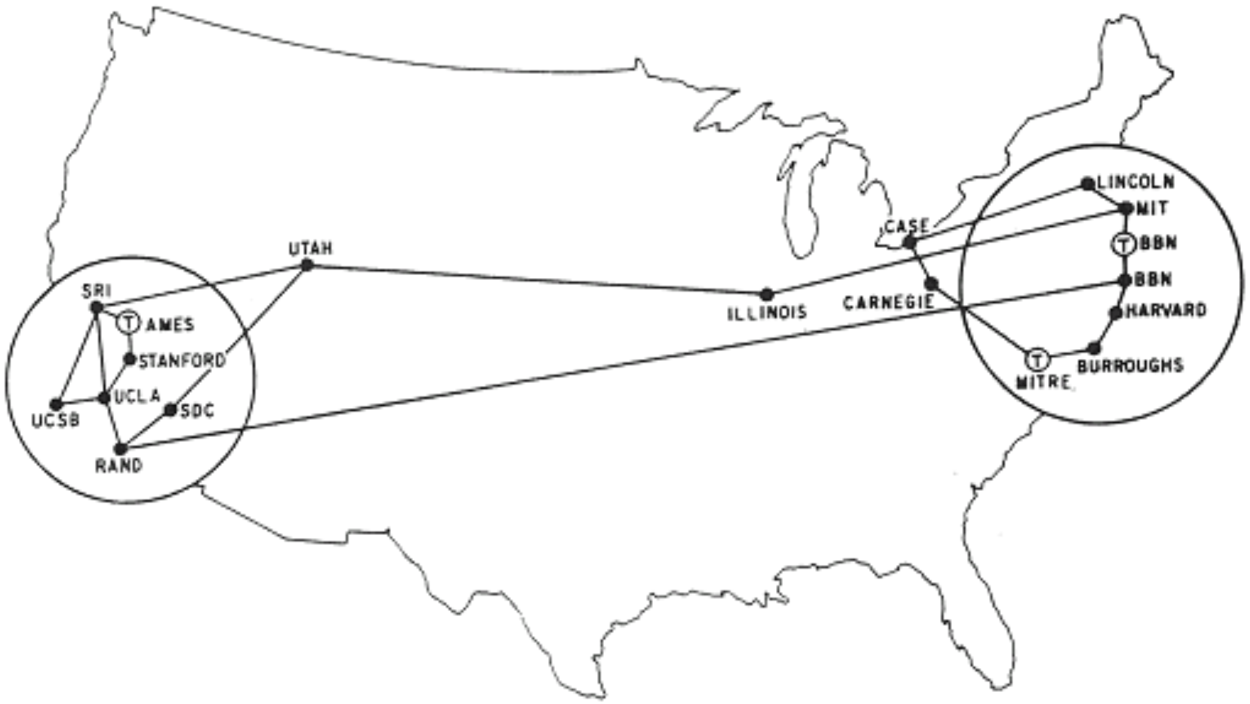
Birthplace of the Internet, 1969

At 10:30 p.m., 29 October 1969, the first ARPANET message was sent from this UCLA site to the Stanford Research Institute. Based on packet switching and dynamic resource allocation, the sharing of information digitally from this first node of ARPANET launched the Internet revolution.

October 2009

IEEE

Internet in 1971



MAP 4 September 1971

Internet History

1972-1980: Internetworking, new and proprietary nets

- 1970: ALOHAnet satellite network in Hawaii
- 1973: Metcalfe' s PhD thesis proposes Ethernet
- 1974: Cerf and Kahn - architecture for interconnecting networks (Turing award work)
- late70' s: proprietary architectures: DECnet, SNA, XNA
- late 70' s: switching fixed length packets (ATM precursor)
- 1979: ARPAnet has 200 nodes

Cerf and Kahn's internetworking principles:

- minimalism, autonomy - no internal changes required to interconnect networks
- best effort service model
- stateless routers
- decentralized control

define today' s Internet architecture

Internet History

1990, 2000's: commercialization, the Web, new apps

- Early 1990's: ARPAnet decommissioned
- 1991: NSF lifts restrictions on commercial use of NSFnet (decommissioned, 1995)
- early 1990s: Web
  - hypertext [Bush 1945, Nelson 1960's]
  - HTML, HTTP: Berners-Lee
  - 1994: Mosaic, later Netscape
  - late 1990's: commercialization of the Web

Late 1990's – 2000's:

- more killer apps: instant messaging, P2P file sharing
- network security to forefront
- est. 50 million host, 100 million+ users
- backbone links running at Gbps

Internet history

- 2000-now:
  - Cloud computing
  - Mobile computing
  - Social applications
  - Internet of Things
  - Smart everything
  - Virtual reality
  - ….

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

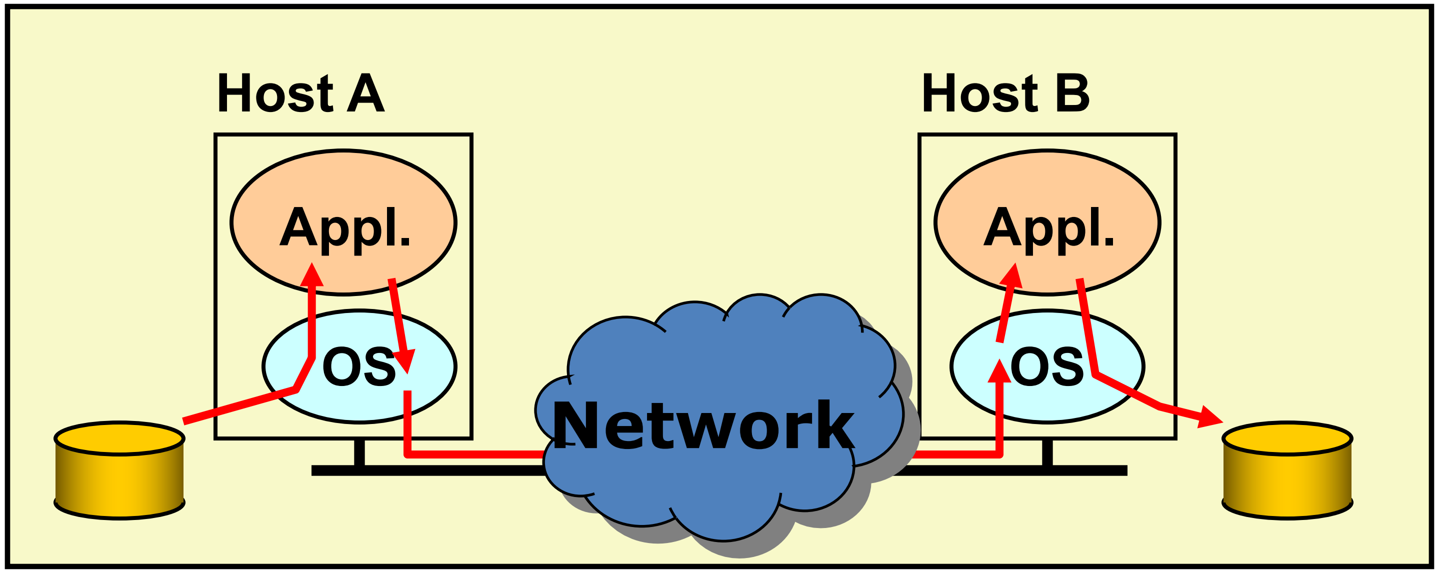
- Modularity argument:
  - It is tempting to implement functions at lower layers so that higher level applications can reuse them

- 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

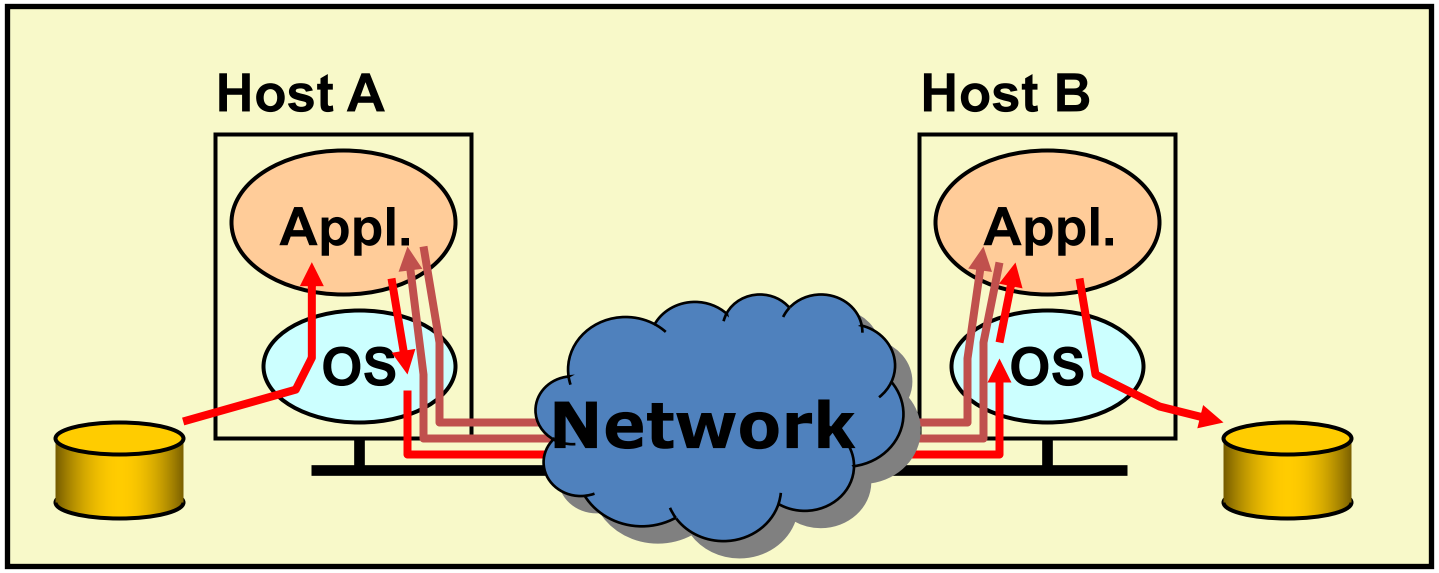
- The authors made their argument by analyzing examples
  - Reliable file transfer
  - Delivery guarantees
  - Secure data transmission
  - Duplicate message suppression
  - FIFO
  - Transaction management
  - *Can you think of more examples to argue for or against the end-to-end argument*?
- Can be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

Example: Reliable File Transfer



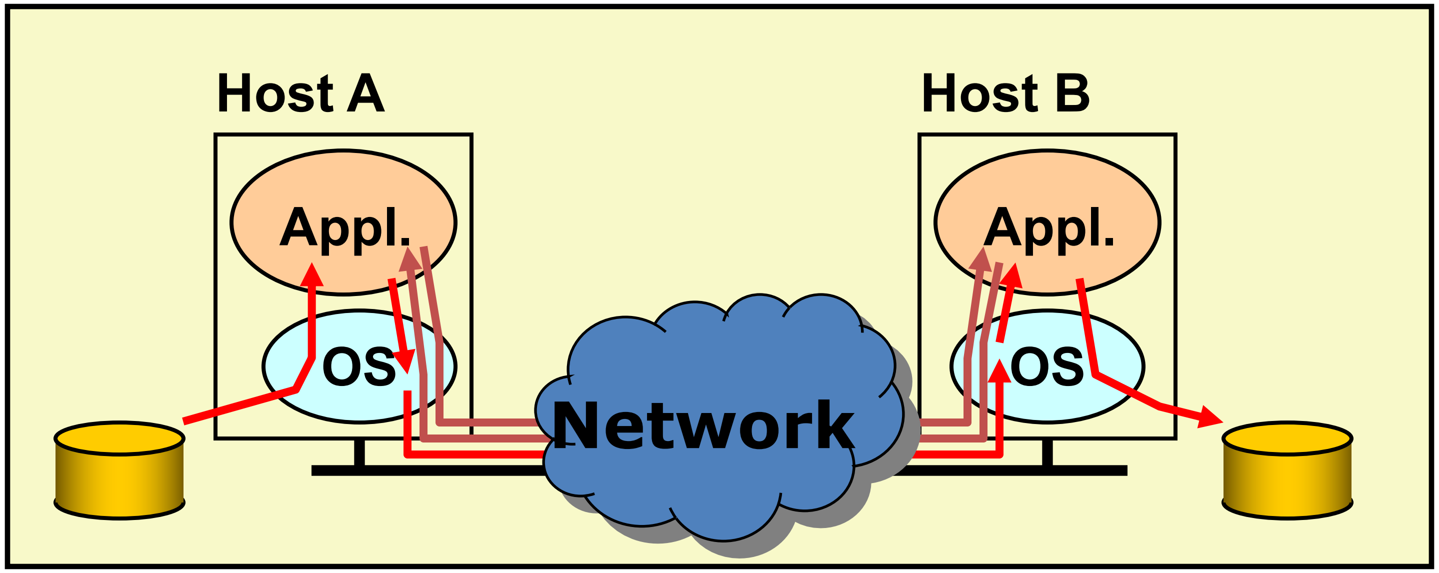
- Solution 1: make each step reliable, and then concatenate them
  - Uneconomical if each step has small error probability

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: $(1-1/100)^{1000}$ ~ 4.3e-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

CompSci 356: Computer Network Architectures
Lecture 3: Introduction to labs

Xiaowei Yang
xwy@cs.duke.edu

Lab overview

- Three labs
  - An echo server
  - A simple router
  - Dynamic routing
- C/C++

Set up the lab environment

1. Download and install VirtualBox
2. Install the provided virtual machine image
   - Wireshark
   - Mininet
3. Write your code in your favorite editor
4. Compile, debug
   - printf is your best friend

Lab 1

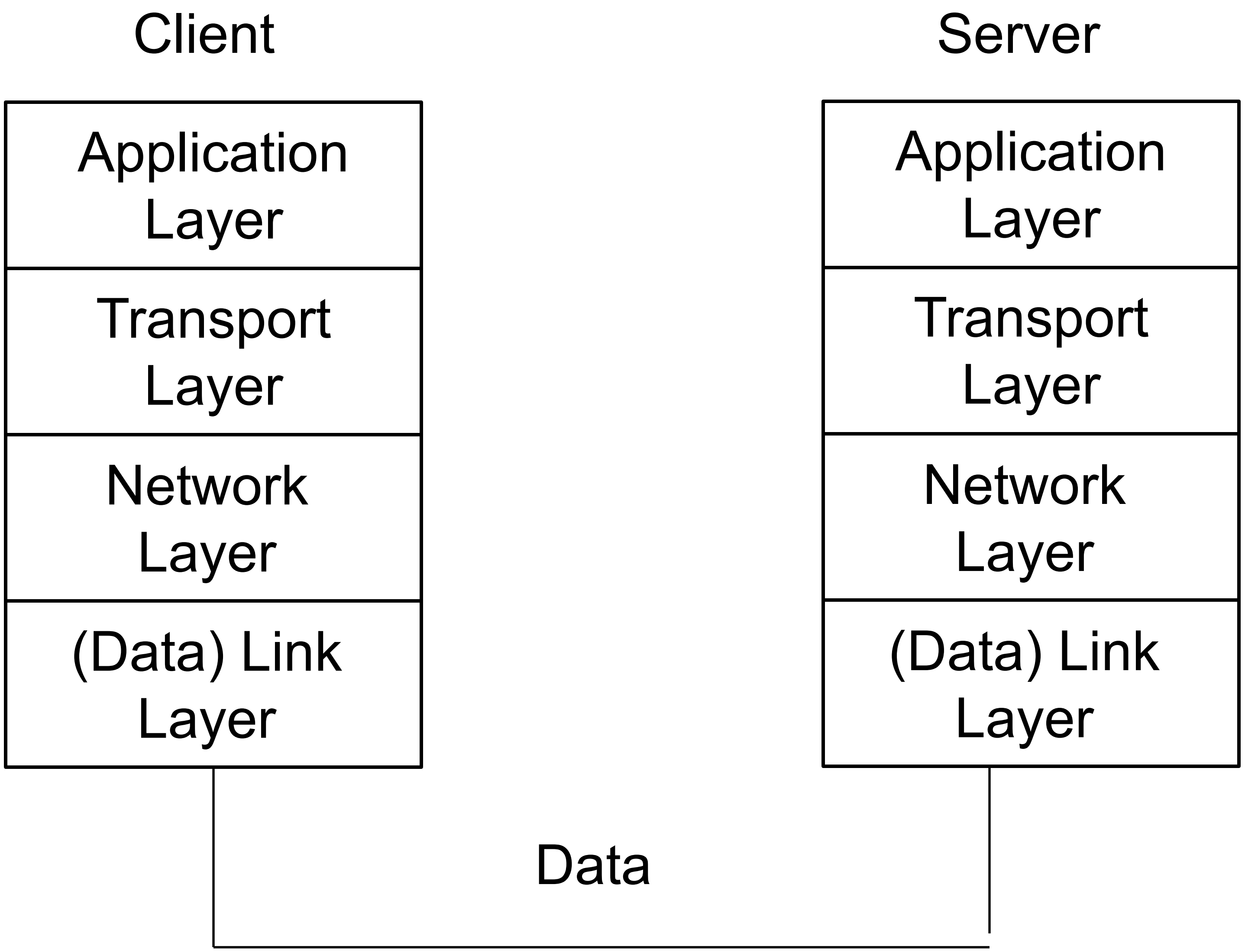
- Write an echo server using TCP sockets

Application Programming Interface (Sockets)

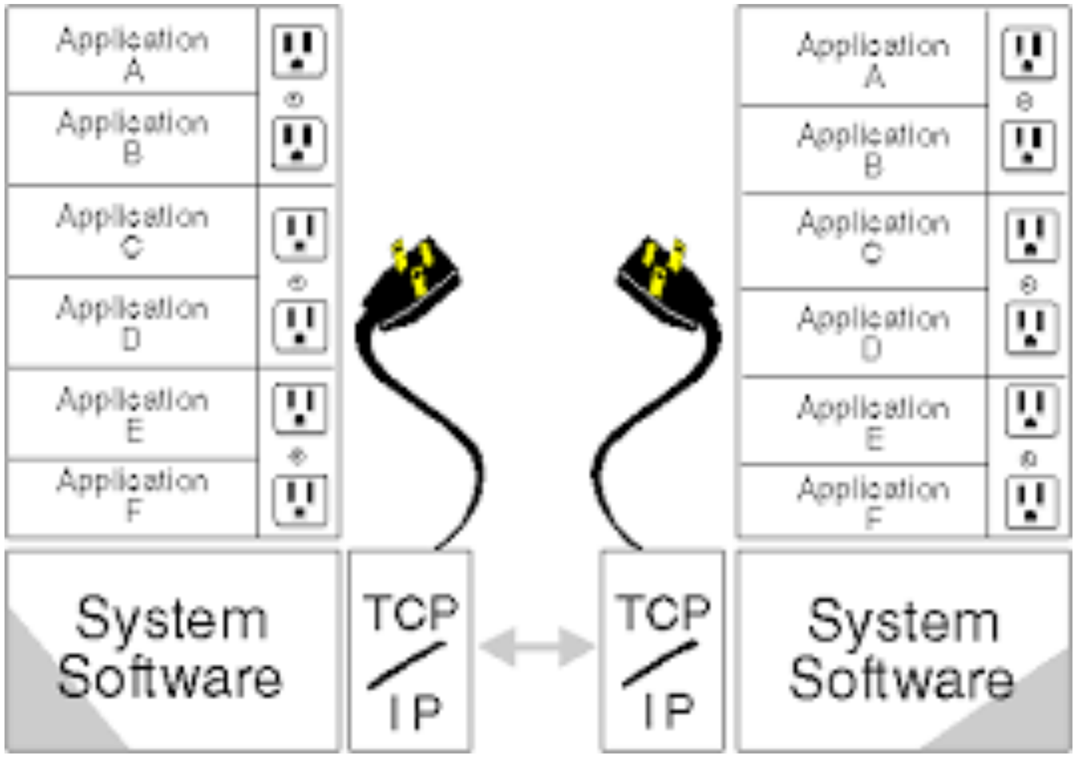
- Socket Interface was originally provided by the Berkeley distribution of Unix
  - Now supported in virtually all operating systems

- 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

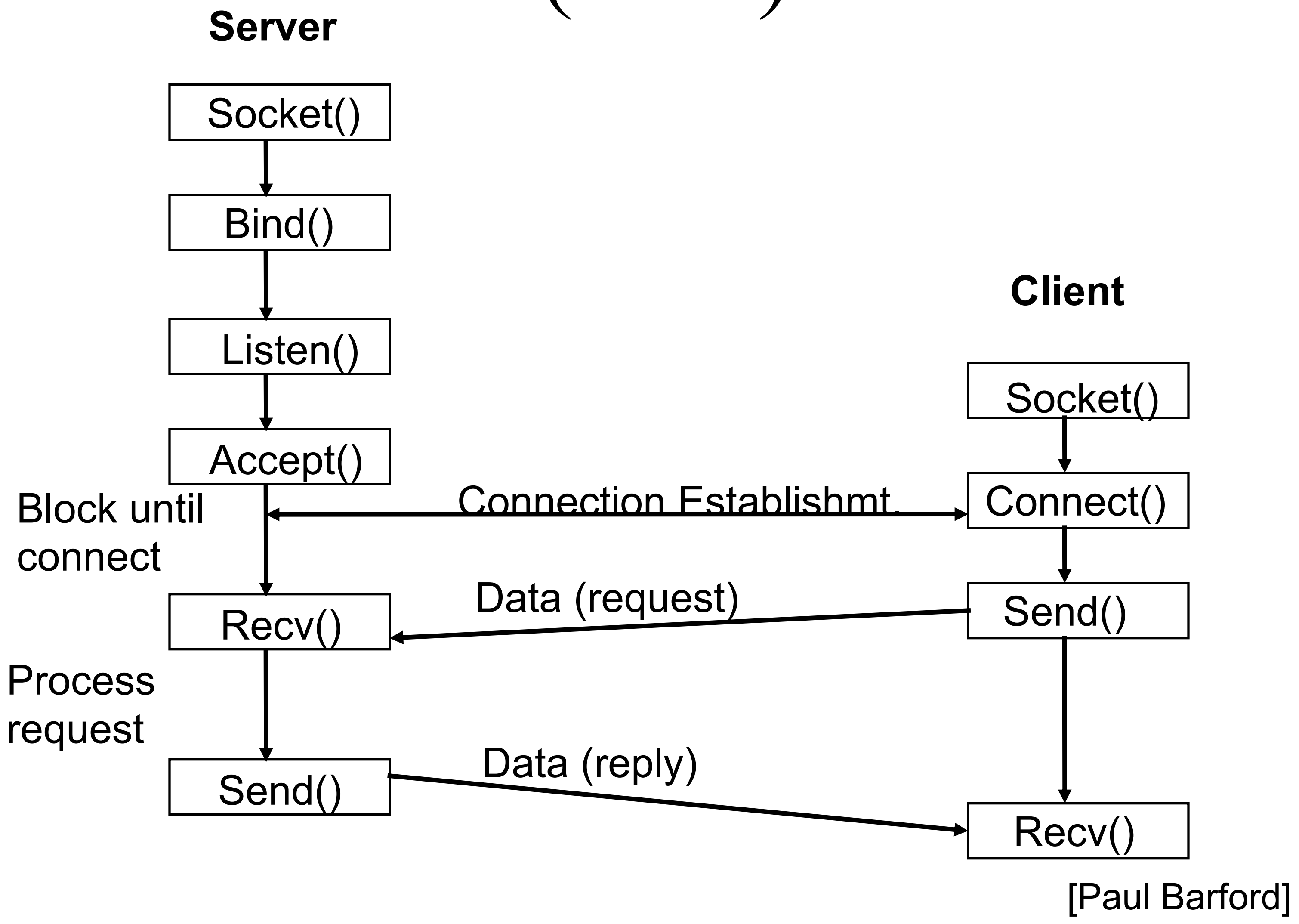
- The interface defines operations for
  - Creating a socket
  - Attaching a socket to the network
  - Sending and receiving messages through the socket
  - Closing the socket

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

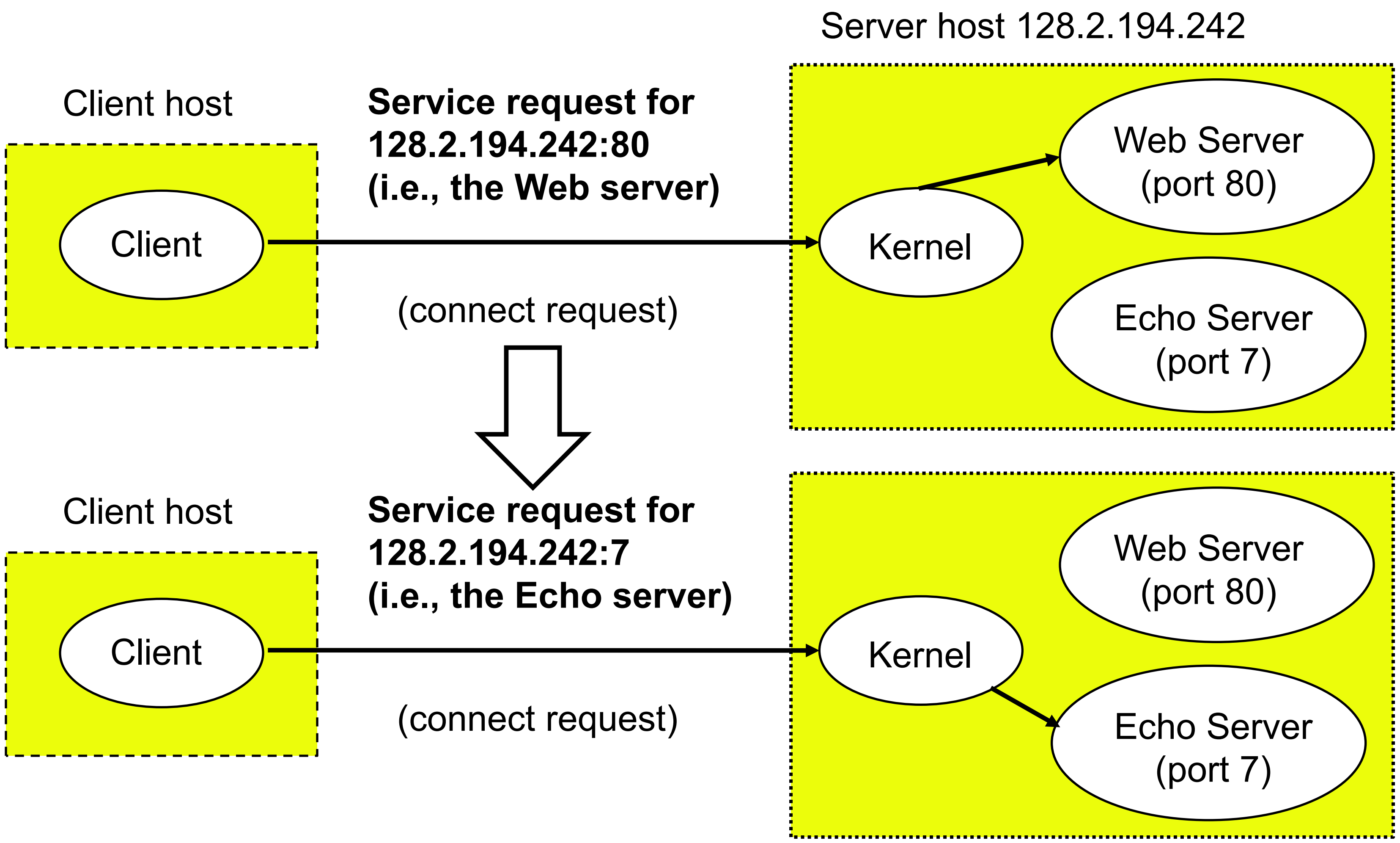
Client-Serve Model with TCP

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

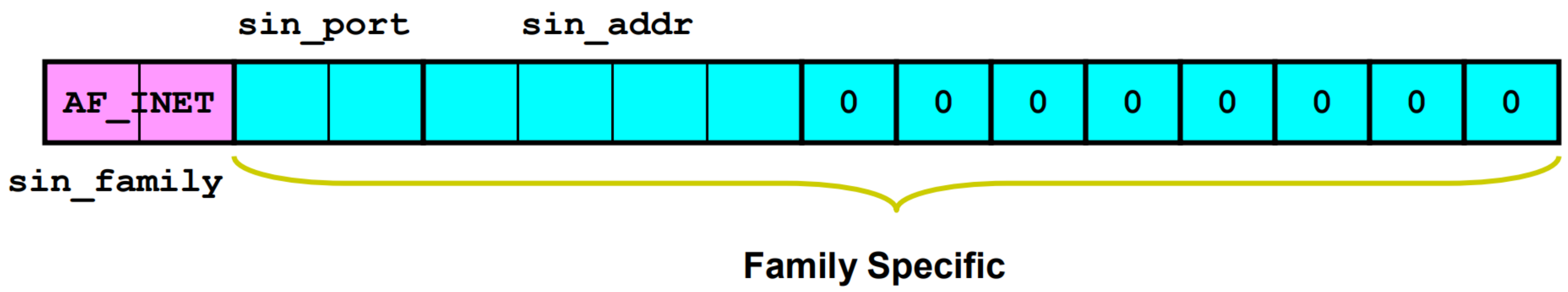
```
    if ((bind(s, (struct sockaddr *)&sin, sizeof(sin))) < 0) {
            perror("simplex-talk: bind");
            exit(1);
    }
    listen(s, MAX_PENDING);
    /* wait for connection, then receive and print text */
    while(1) {
            if ((new_s = accept(s, (struct sockaddr *)&sin, &len))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



[CMU 15-213]

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;
}
```