Introduction to Computer Systems

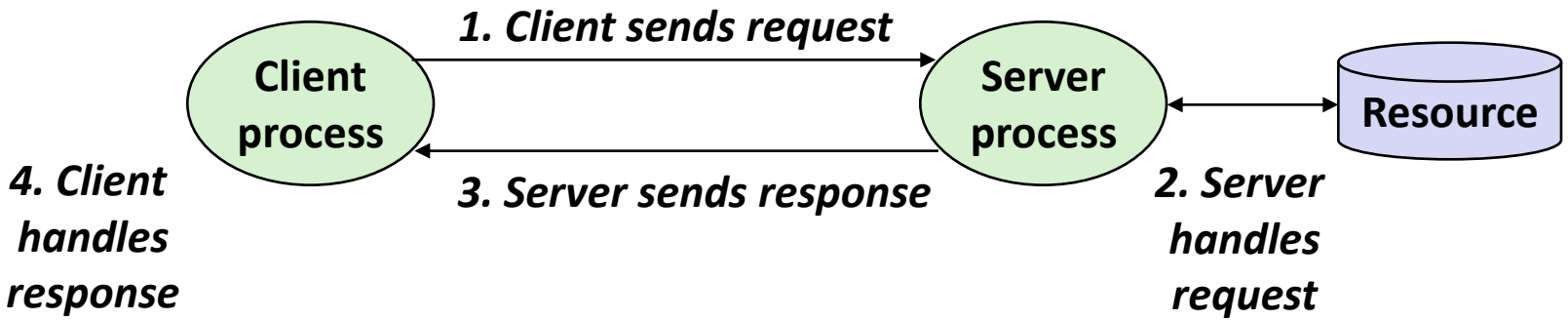
15-213/18-243, spring 2009

23rd Lecture, Apr. 14th

Instructors:

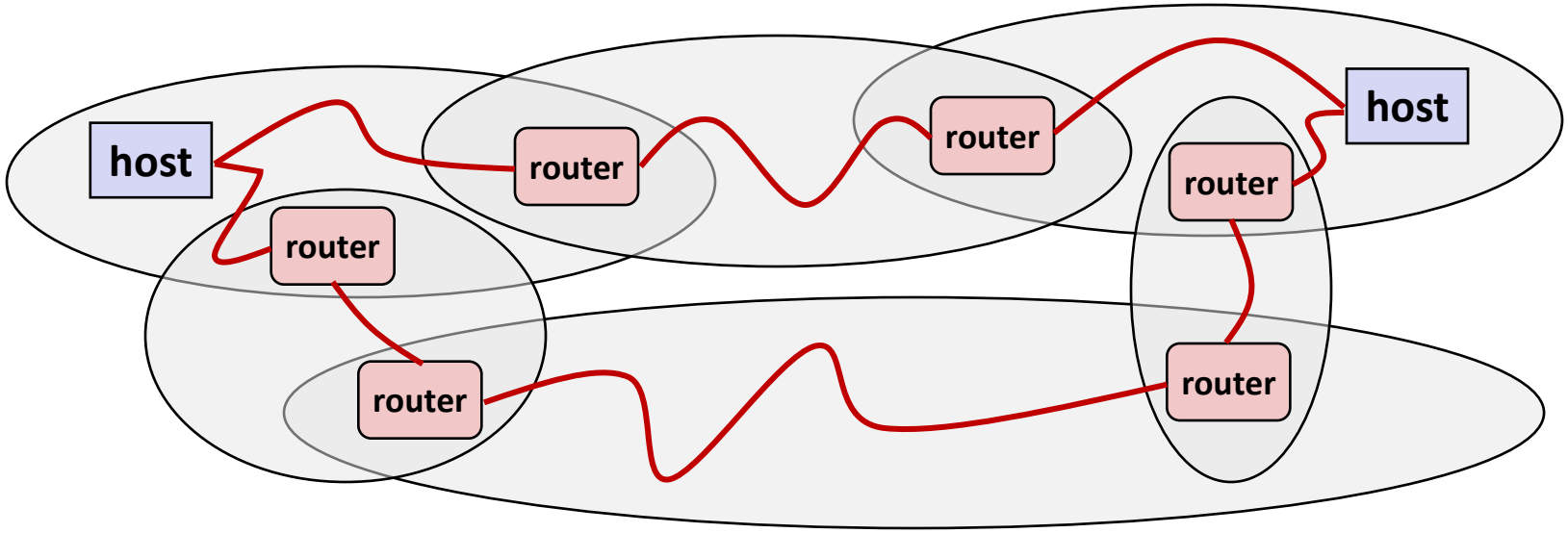
Gregory Kesden and Markus Püschel

Last Time: Client-Server Transaction



Note: clients and servers are processes running on hosts (can be the same or different hosts)

Last Time: Logical Structure of an internet

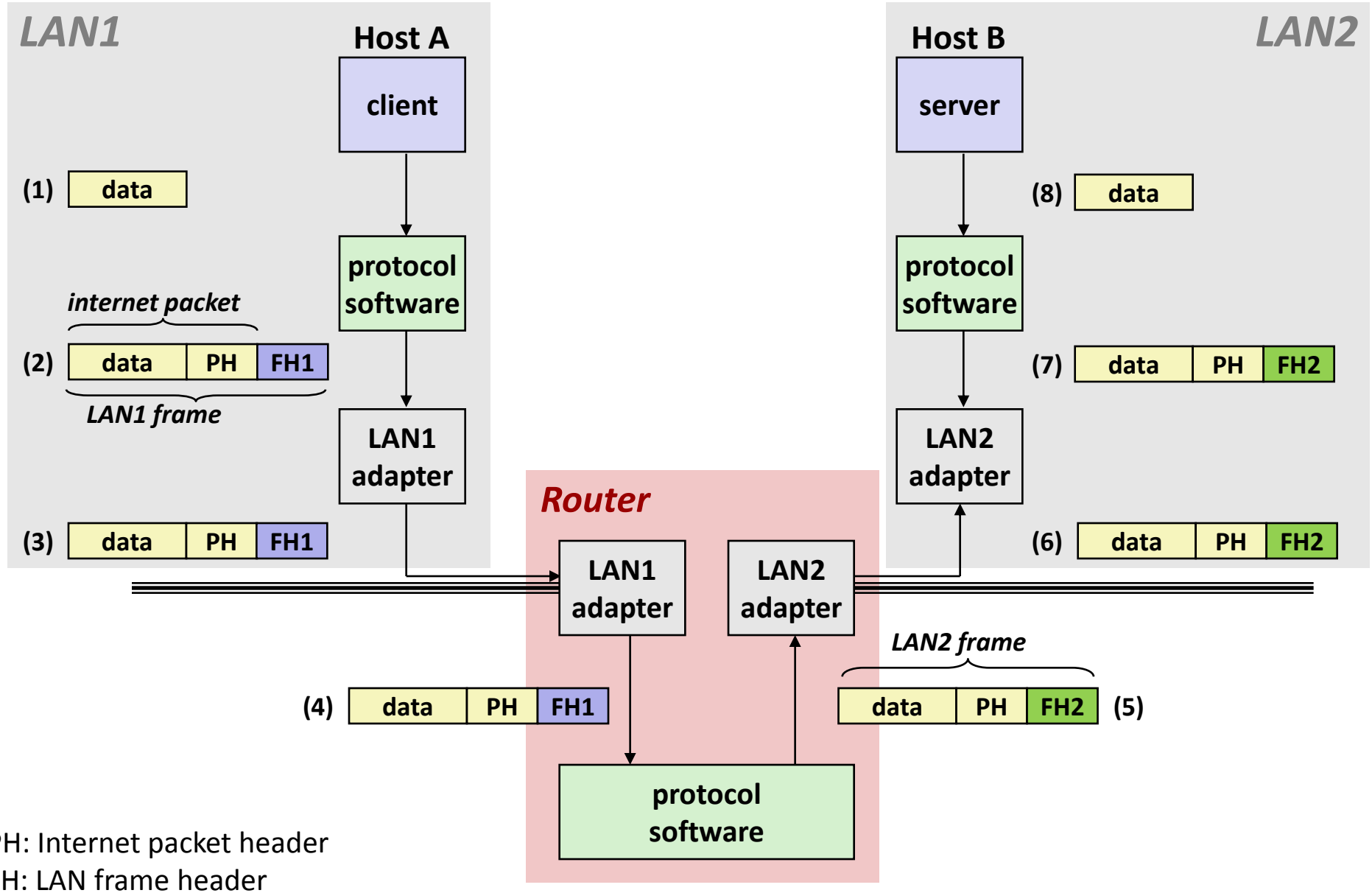


Last Time: internet Protocol

- **Provides a naming scheme**
  - An internet protocol defines a uniform format for ***host addresses***
  - Each host (and router) is assigned at least one of these internet addresses that uniquely identifies it

- **Provides a delivery mechanism**
  - An internet protocol defines a standard transfer unit (***packet***)
  - Packet consists of ***header*** and ***payload***
    - Header: contains info such as packet size, source and destination addresses
    - Payload: contains data bits sent from source host

Transferring Data Over an internet



PH: Internet packet header
FH: LAN frame header

Today

- Programmer's view of the internet
- Sockets interface

A Programmer's View of the Internet

- **Hosts are mapped to a set of 32-bit *IP addresses***
  - 128.2.203.179
- **The set of IP addresses is mapped to a set of identifiers called Internet *domain names***
  - 128.2.203.179 is mapped to www.cs.cmu.edu
- **A process on one Internet host can communicate with a process on another Internet host over a *connection***

IP Addresses

- **32-bit IP addresses are stored in an *IP address struct***
  - IP addresses are always stored in memory in network byte order (big-endian byte order)
  - True in general for any integer transferred in a packet header from one machine to another.
    - E.g., the port number used to identify an Internet connection.

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
};
```

Useful network byte-order conversion functions:

`htonl`: convert long int from host to network byte order
`htons`: convert short int from host to network byte order
`ntohl`: convert long int from network to host byte order
`ntohs`: convert short int from network to host byte order

Dotted Decimal Notation

- **By convention, each byte in a 32-bit IP address is represented by a string: decimal values for bytes, separated by a period**
  - IP address: `0x8002C2F2` = ***Blackboard?***

Dotted Decimal Notation

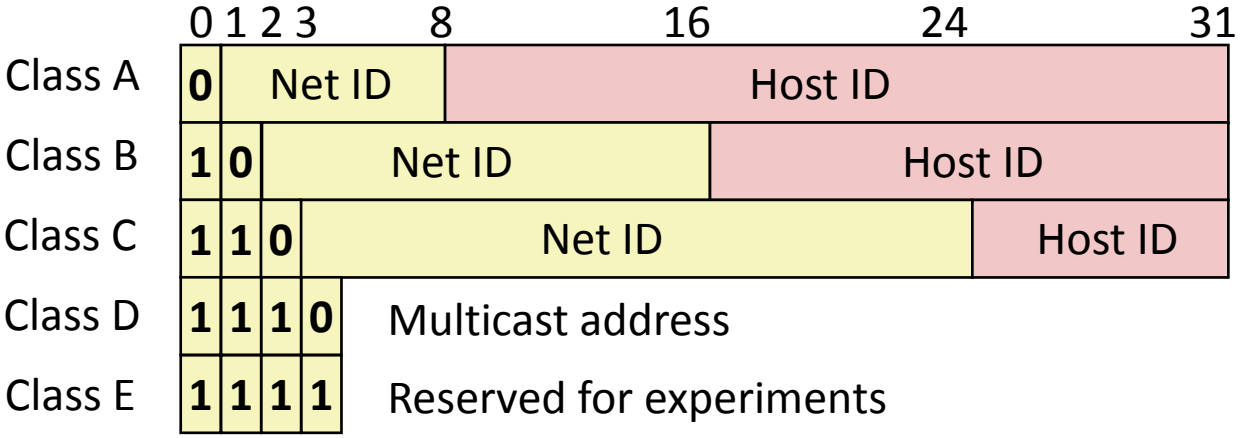
- **By convention, each byte in a 32-bit IP address is represented by a string: decimal values for bytes, separated by a period**
  - IP address: `0x8002C2F2 = 128.2.194.242`

- **Functions for converting between binary IP addresses and dotted decimal strings:**
  - **`inet_aton`:** dotted decimal string → IP address in network byte order
  - **`inet_ntoa`:** IP address in network byte order → dotted decimal string

  - "n" denotes network representation
  - "a" denotes application representation

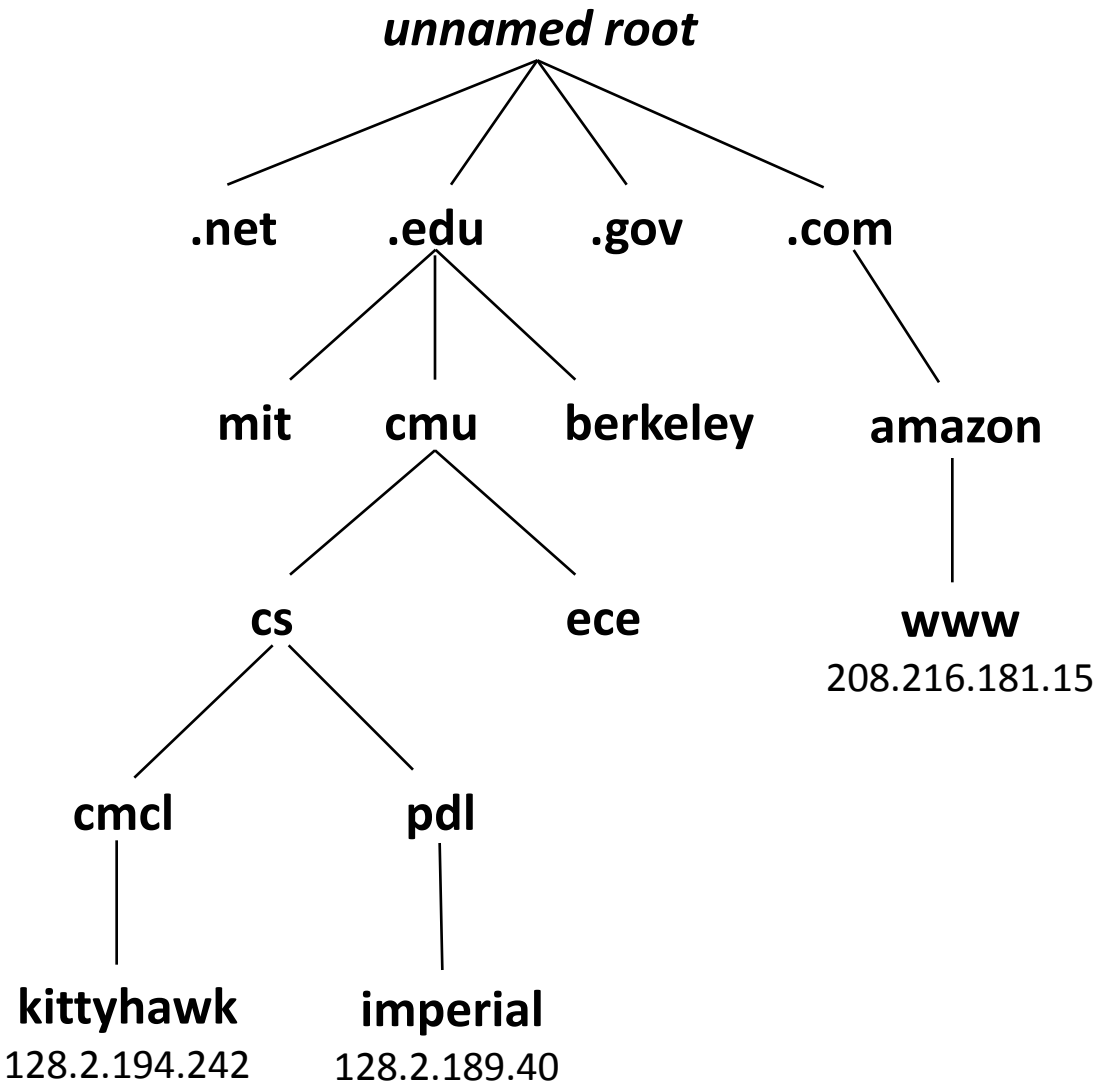
IP Address Structure

- **IP (V4) Address space divided into classes:**



- **Network ID written in form w.x.y.z/n**
  - n = number of bits in net id (yellow part above)
  - E.g., CMU written as 128.2.0.0/16
    - Which class is that?
- **Unrouted (private) IP addresses:**
  - 10.0.0.0/8 172.16.0.0/12 192.168.0.0/16
- **Nowadays: CIDR (Classless interdomain routing)**

Internet Domain Names



First-level domain names

Second-level domain names

Third-level domain names

Domain Naming System (DNS)

- **The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed DNS database**
  - Conceptually, programmers can view the DNS database as a collection of millions of *host entry structures*:

```
/* DNS host entry structure */
struct hostent {
   char    *h_name;        /* official domain name of host */
   char    **h_aliases;    /* null-terminated array of domain names */
   int     h_addrtype;     /* host address type (AF_INET) */
   int     h_length;       /* length of an address, in bytes */
   char    **h_addr_list;  /* null-terminated array of in_addr structs */
};
```

- **Functions for retrieving host entries from DNS:**
  - **`gethostbyname`**: query key is a DNS domain name
  - **`gethostbyaddr`**: query key is an IP address

Properties of DNS Host Entries

- **Each host entry is an equivalence class of domain names and IP addresses**
- **Each host has a locally defined domain name `localhost` which always maps to the *loopback address* `127.0.0.1`**
- **Different kinds of mappings are possible:**
  - Simple case: one-to-one mapping between domain name and IP address:
    - **`kittyhawk.cmcl.cs.cmu.edu`** maps to **`128.2.194.242`**
  - Multiple domain names mapped to the same IP address:
    - **`eecs.mit.edu`** and **`cs.mit.edu`** both map to **`18.62.1.6`**
  - Multiple domain names mapped to multiple IP addresses:
    - **`aol.com`** and **`www.aol.com`** map to multiple IP addresses
  - Some valid domain names don't map to any IP address:
    - for example: **`cmcl.cs.cmu.edu`**

A Program That Queries DNS

```
int main(int argc, char **argv) { /* argv[1] is a domain name */
    char **pp;                    /* or dotted decimal IP addr */
    struct in_addr addr;
    struct hostent *hostp;

    if (inet_aton(argv[1], &addr) != 0)
        hostp = Gethostbyaddr((const char *)&addr, sizeof(addr),
                  AF_INET);
    else
        hostp = Gethostbyname(argv[1]);
    printf("official hostname: %s\n", hostp->h_name);

    for (pp = hostp->h_aliases; *pp != NULL; pp++)
        printf("alias: %s\n", *pp);

    for (pp = hostp->h_addr_list; *pp != NULL; pp++) {
        addr.s_addr = ((struct in_addr *)*pp)->s_addr;
        printf("address: %s\n", inet_ntoa(addr));
    }
}
```

Querying DNS from the Command Line

- Domain Information Groper (`dig`) provides a scriptable command line interface to DNS

```
linux> dig +short kittyhawk.cmcl.cs.cmu.edu
128.2.194.242
linux> dig +short -x 128.2.194.242
KITTYHAWK.CMCL.CS.CMU.EDU.
linux> dig +short aol.com
205.188.145.215
205.188.160.121
64.12.149.24
64.12.187.25
linux> dig +short -x 64.12.187.25
aol-v5.websys.aol.com.
```

Internet Connections

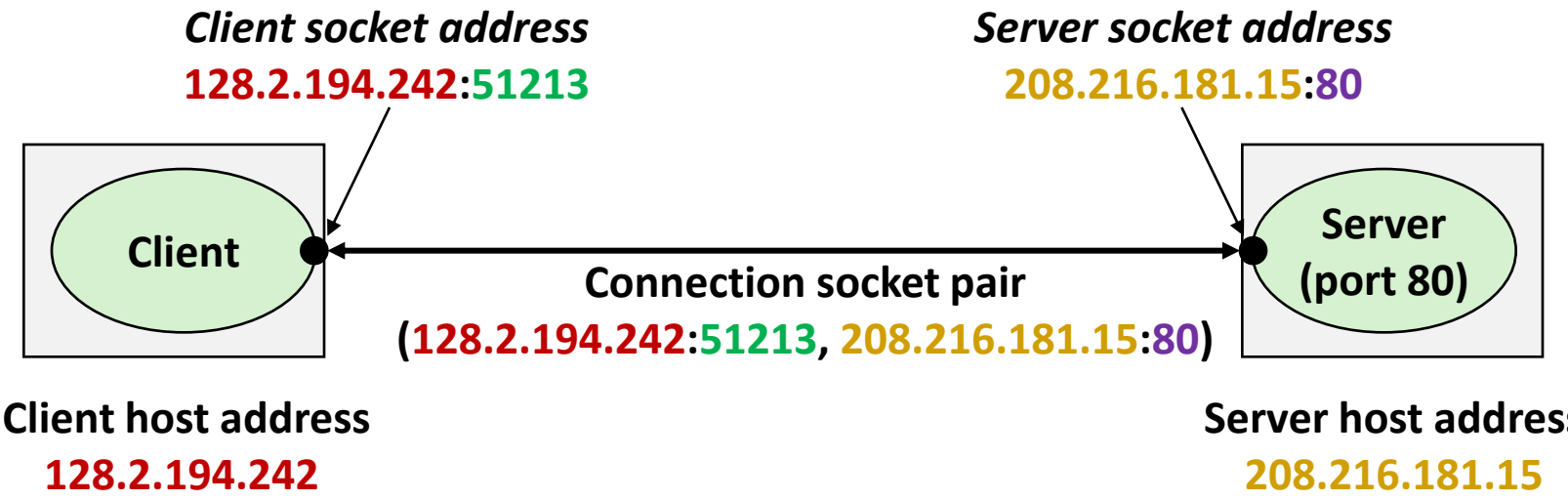
- **Clients and servers communicate by sending streams of bytes over *connections*:**
  - Point-to-point, full-duplex (2-way communication), and reliable.

- ***A socket* is an endpoint of a connection**
  - Socket address is an `IPaddress:port` pair

- **A *port* is a 16-bit integer that identifies a process:**
  - ***Ephemeral port*:** Assigned automatically on client when client makes a connection request
  - ***Well-known port:*** Associated with some service provided by a server (e.g., port 80 is associated with Web servers)

- **A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)**
  - `(cliaddr:cliport, servaddr:servport)`

Putting it all Together: Anatomy of an Internet Connection



51213 is an ephemeral port allocated by the kernel

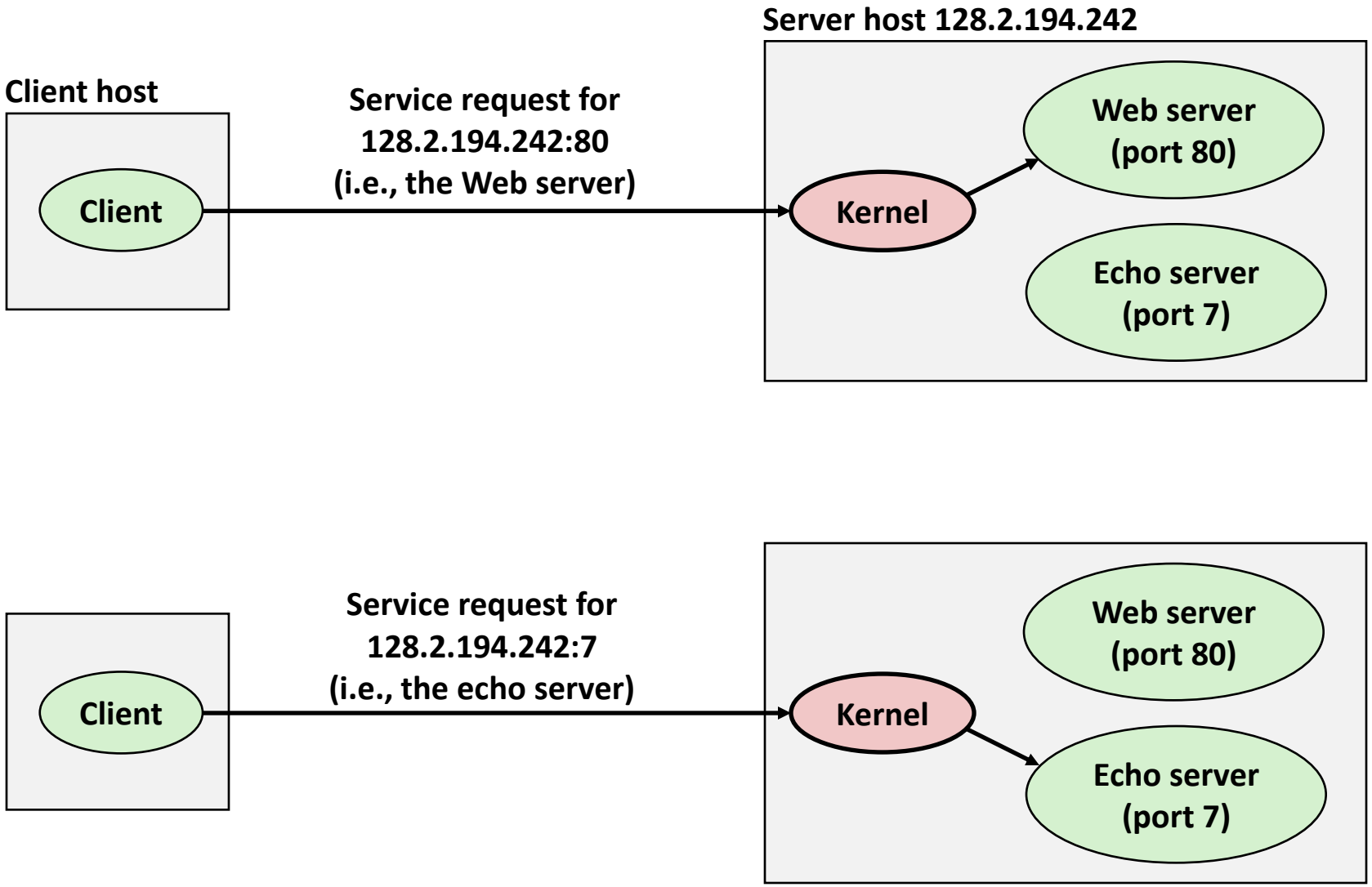
80 is a well-known port associated with Web servers

Clients

- **Examples of client programs**
  - Web browsers, **`ftp`**, **`telnet`**, **`ssh`**

- **How does a client find the server?**
  - The IP address in the server socket address identifies the host (more precisely, an adapter on the host)
  - The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.
  - Examples of well know ports
    - Port 7: Echo server
    - Port 23: Telnet server
    - Port 25: Mail server
    - Port 80: Web server

Using Ports to Identify Services



Servers

- **Servers are long-running processes (daemons)**
  - Created at boot-time (typically) by the init process (process 1)
  - Run continuously until the machine is turned off

- **Each server waits for requests to arrive on a well-known port associated with a particular service**
  - Port 7: echo server
  - Port 23: telnet server
  - Port 25: mail server
  - Port 80: HTTP server

- **A machine that runs a server process is also often referred to as a "server"**

Server Examples

- **Web server (port 80)**
  - Resource: files/compute cycles (CGI programs)
  - Service: retrieves files and runs CGI programs on behalf of the client

- **FTP server (20, 21)**
  - Resource: files
  - Service: stores and retrieve files

See `/etc/services` for a comprehensive list of the port mappings on a Linux machine

- **Telnet server (23)**
  - Resource: terminal
  - Service: proxies a terminal on the server machine

- **Mail server (25)**
  - Resource: email “spool” file
  - Service: stores mail messages in spool file

Today

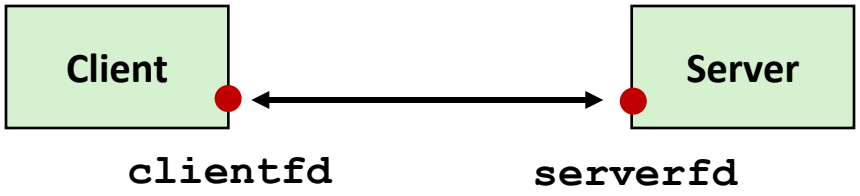
- Programmer's view of the internet
- **Sockets interface**

Sockets Interface

- Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols
- Provides a user-level interface to the network
- Underlying basis for all Internet applications
- Based on client/server programming model

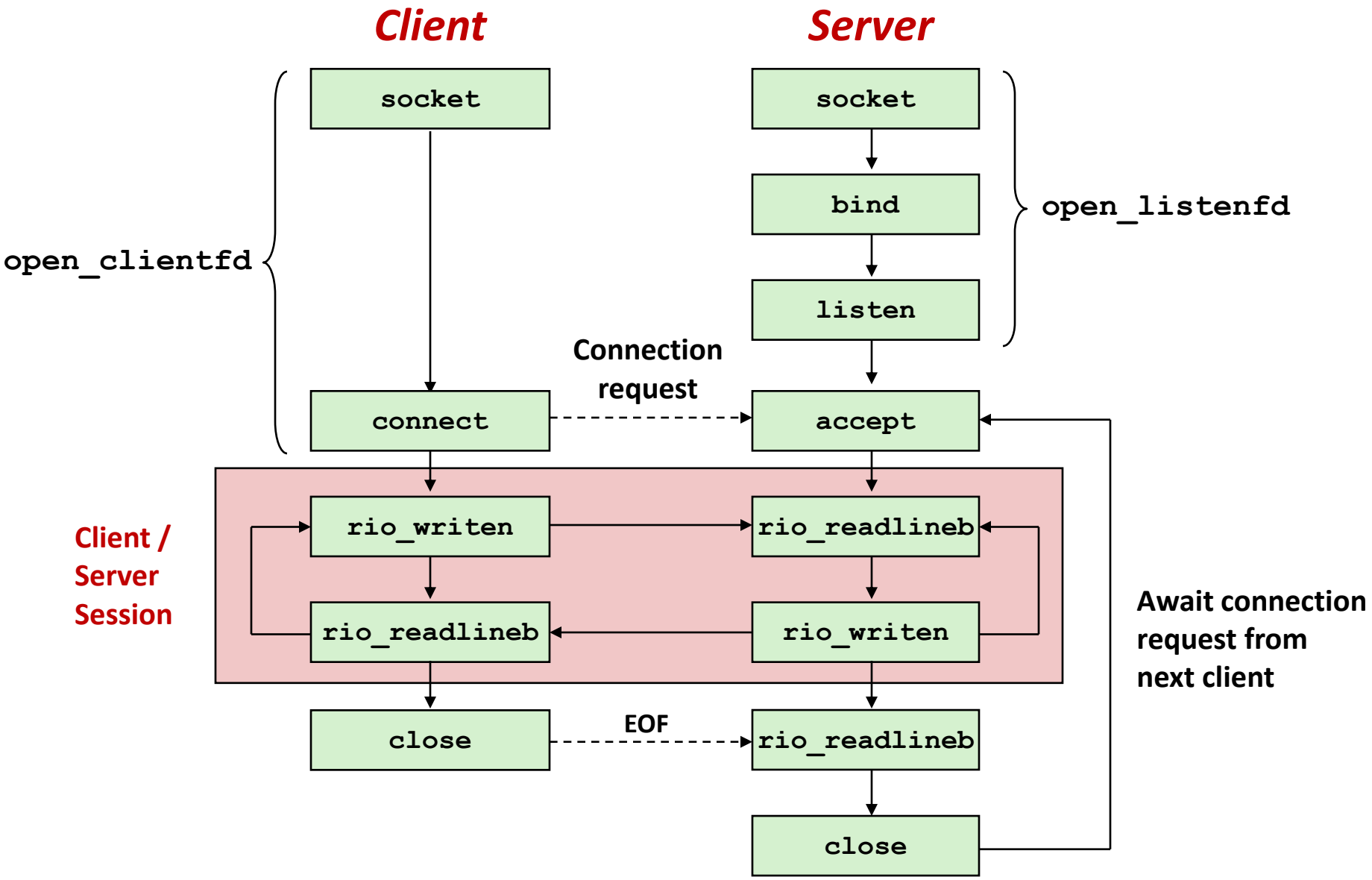
Sockets

- **What is a socket?**
  - To the kernel, a socket is an endpoint of communication
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
    - ***Remember:*** All Unix I/O devices, including networks, are modeled as files
- **Clients and servers communicate with each other by reading from and writing to socket descriptors**



- **The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors**

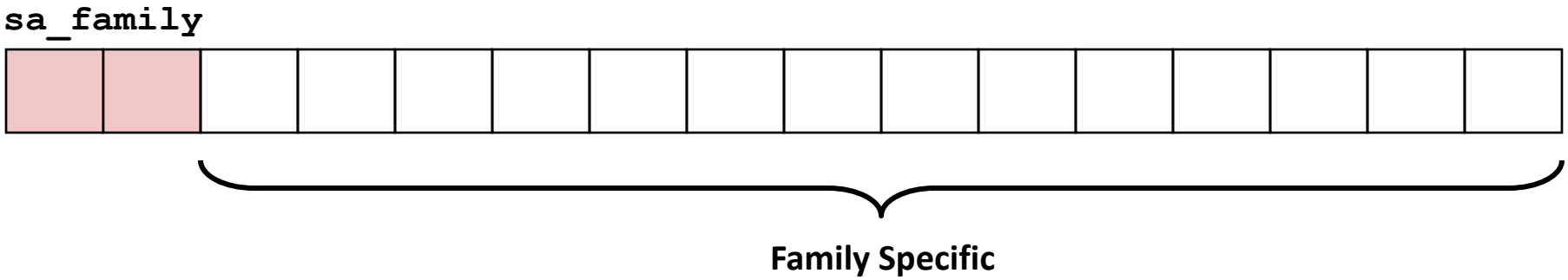
Overview of the Sockets Interface



Socket Address Structures

- **Generic socket address:**
  - For address arguments to **`connect`**, **`bind`**, and **`accept`**
  - Necessary only because C did not have generic (**`void *`**) pointers when the sockets interface was designed

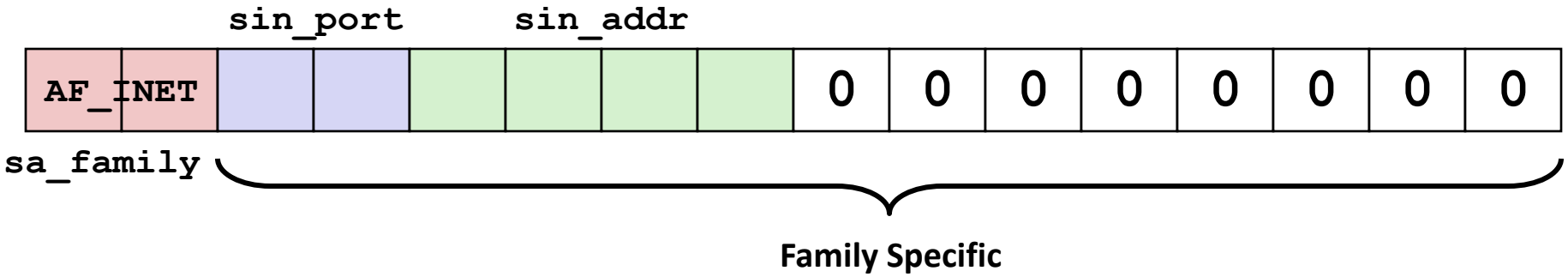
```
struct sockaddr {
  unsigned short  sa_family;    /* protocol family */
  char            sa_data[14];  /* address data.  */
};
```



Socket Address Structures

- **Internet-specific socket address:**
  - Must cast (**sockaddr_in ***) to (**sockaddr ***) for **connect**, **bind**, and **accept**

```
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) */
};
```



Example: Echo Client and Server

On Server

```
bass> echoserver 5000
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 4 bytes: 123
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 7 bytes: 456789
...
```

On Client

```
kittyhawk> echoclient bass 5000
Enter message: 123
Echo: 123
Enter message: ^D
kittyhawk> echoclient bass 5000
Enter message: 456789
Echo: 456789
Enter message: ^D
kittyhawk>
```

Echo Client Main Routine

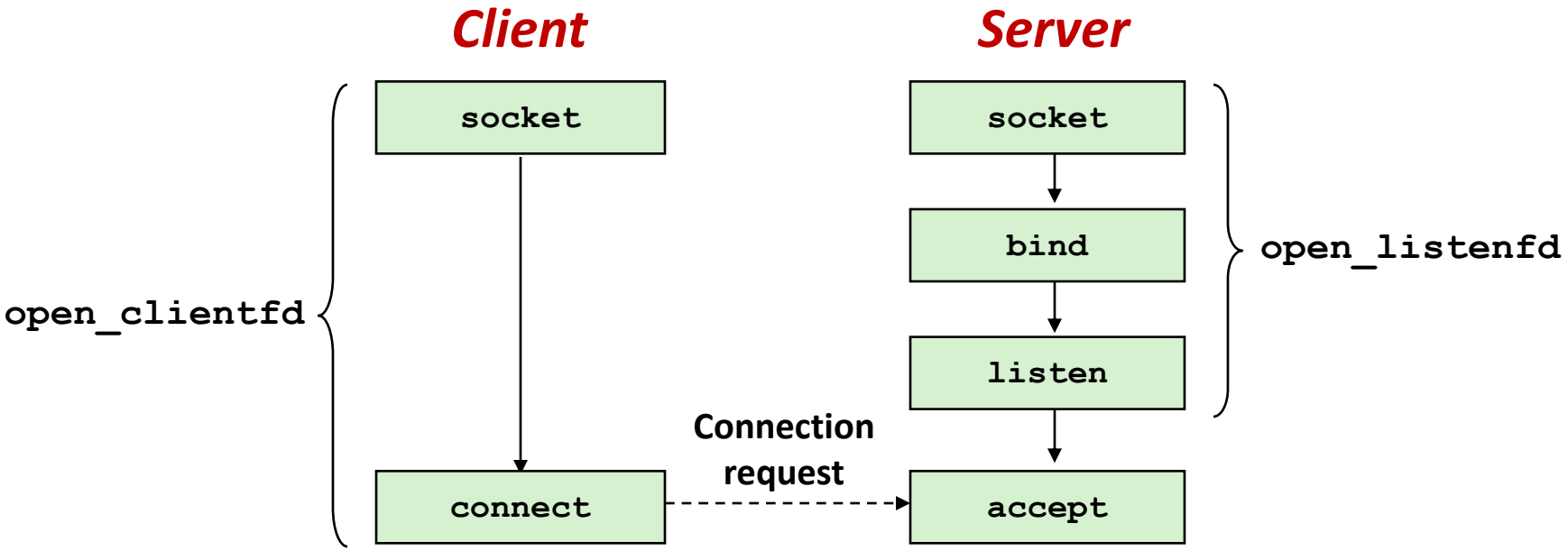
Send line to server

Receive line from server

```
#include "csapp.h"

/* usage: ./echoclient host port */
int main(int argc, char **argv)
{
    int clientfd, port;
    char *host, buf[MAXLINE];
    rio_t rio;
    host = argv[1];  port = atoi(argv[2]);
    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);
    printf("Enter message:"); fflush(stdout);
    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        printf("Echo:");
        Fputs(buf, stdout);
        printf("Enter message:"); fflush(stdout);
    }
    Close(clientfd);
    exit(0);
}
```

Overview of the Sockets Interface



Echo Client: open_clientfd

```
int open_clientfd(char *hostname, int port) {
  int clientfd;
  struct hostent *hp;
  struct sockaddr_in serveraddr;

  if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1; /* check errno for cause of error */

  /* Fill in the server's IP address and port */
  if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
  bzero((char *) &serveraddr, sizeof(serveraddr));
  serveraddr.sin_family = AF_INET;
  bcopy((char *)hp->h_addr_list[0],
        (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
  serveraddr.sin_port = htons(port);

  /* Establish a connection with the server */
  if (connect(clientfd, (SA *) &serveraddr,
      sizeof(serveraddr)) < 0)
    return -1;
  return clientfd;
}
```

This function opens a connection from the client to the server at `hostname:port`

Create socket

Create address

Establish connection

Echo Client: `open_clientfd` (`socket`)

- **`socket` creates a socket descriptor on the client**
  - Just allocates & initializes some internal data structures
  - **`AF_INET`**: indicates that the socket is associated with Internet protocols
  - **`SOCK_STREAM`**: selects a reliable byte stream connection
    - provided by TCP

```
int clientfd;  /* socket descriptor */

if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1; /* check errno for cause of error */

... <more>
```

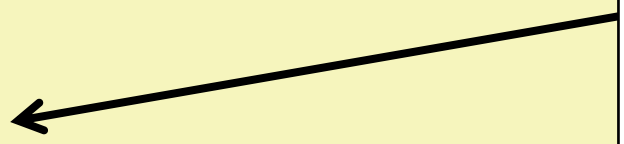
Echo Client: `open_clientfd` (`gethostbyname`)

- The client then builds the server's Internet address

```
int clientfd;                    /* socket descriptor */
struct hostent *hp;              /* DNS host entry */
struct sockaddr_in serveraddr;   /* server's IP address */

...

/* fill in the server's IP address and port */
if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(port);
bcopy((char *)hp->h_addr_list[0],
      (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
```



A Careful Look at bcopy Arguments

```
/* DNS host entry structure */
struct hostent {
    . . .
    int     h_length;       /* length of an address, in bytes */
    char    **h_addr_list; /* null-terminated array of in_addr structs */
};
```

```
struct sockaddr_in  {
   . . .
   struct in_addr  sin_addr;     /* IP addr in network byte order */
   . . .
};
```

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
};
```

```
struct hostent *hp;              /* DNS host entry */
struct sockaddr_in serveraddr; /* server's IP address */
...
bcopy((char *)hp->h_addr_list[0], /* src, dest */
      (char *)&serveraddr.sin_addr.s_addr, hp->h_length);
```

Echo Client: `open_clientfd` (`connect`)

- **Finally the client creates a connection with the server**
  - Client process suspends (blocks) until the connection is created
  - After resuming, the client is ready to begin exchanging messages with the server via Unix I/O calls on descriptor **`clientfd`**

```
  int clientfd;                    /* socket descriptor */
  struct sockaddr_in serveraddr;   /* server address */
  typedef struct sockaddr SA;      /* generic sockaddr */
...
  /* Establish a connection with the server */
  if (connect(clientfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
    return -1;
  return clientfd;
}
```

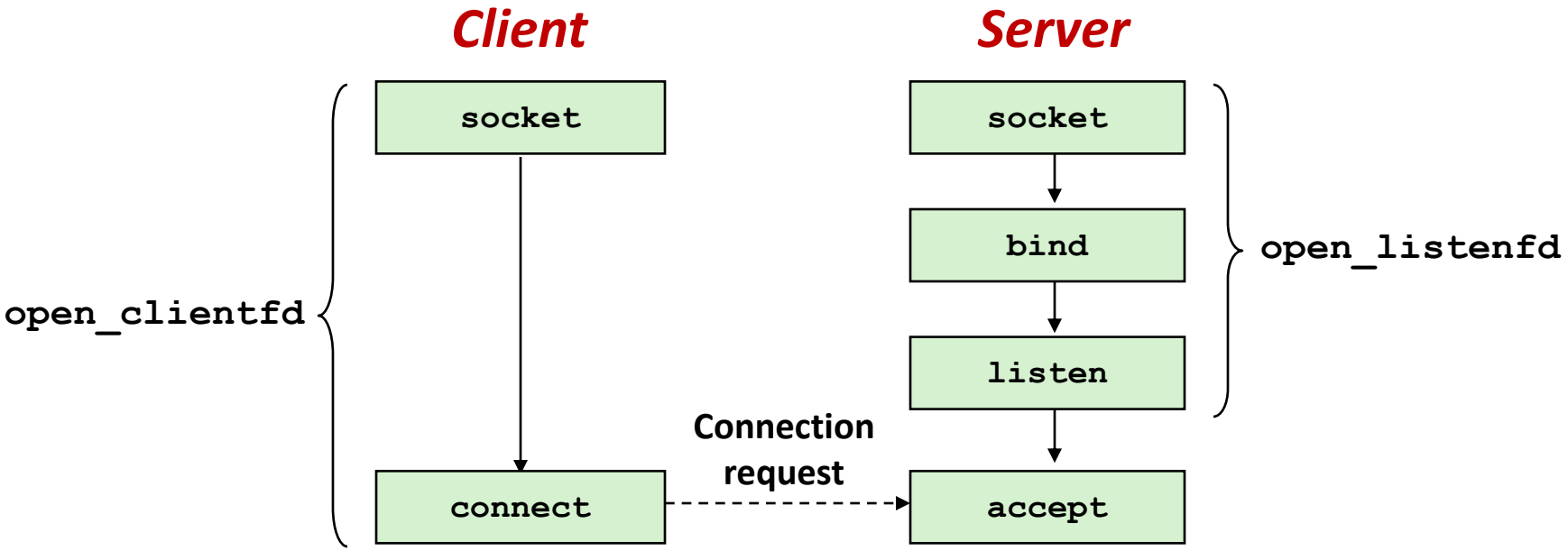
Echo Server: Main Routine

```
int main(int argc, char **argv) {
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;
    struct hostent *hp;
    char *haddrp;

    port = atoi(argv[1]); /* the server listens on a port passed
                             on the command line */
    listenfd = open_listenfd(port);

    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                           sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        haddrp = inet_ntoa(clientaddr.sin_addr);
        printf("server connected to %s (%s)\n", hp->h_name, haddrp);
        echo(connfd);
        Close(connfd);
    }
}
```

Overview of the Sockets Interface



Echo Server: open_listenfd

```
int open_listenfd(int port)
{
    int listenfd, optval=1;
    struct sockaddr_in serveraddr;

    /* Create a socket descriptor */
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    /* Eliminates "Address already in use" error from bind. */
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int)) < 0)
        return -1;

... <more>
```

Echo Server: open_listenfd (cont.)

```
...

  /* Listenfd will be an endpoint for all requests to port
       on any IP address for this host */
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons((unsigned short)port);
    if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
        return -1;

    /* Make it a listening socket ready to accept
       connection requests */
    if (listen(listenfd, LISTENQ) < 0)
        return -1;

   return listenfd;
}
```

Echo Server: `open_listenfd` (`socket`)

- **`socket` creates a socket descriptor on the server**
  - **`AF_INET`**: indicates that the socket is associated with Internet protocols
  - **`SOCK_STREAM`**: selects a reliable byte stream connection (TCP)

```
int listenfd; /* listening socket descriptor */

/* Create a socket descriptor */
if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1;
```

Echo Server: `open_listenfd` (`setsockopt`)

- **The socket can be given some attributes**

```
...
/* Eliminates "Address already in use" error from bind(). */
if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
               (const void *)&optval , sizeof(int)) < 0)
    return -1;
```

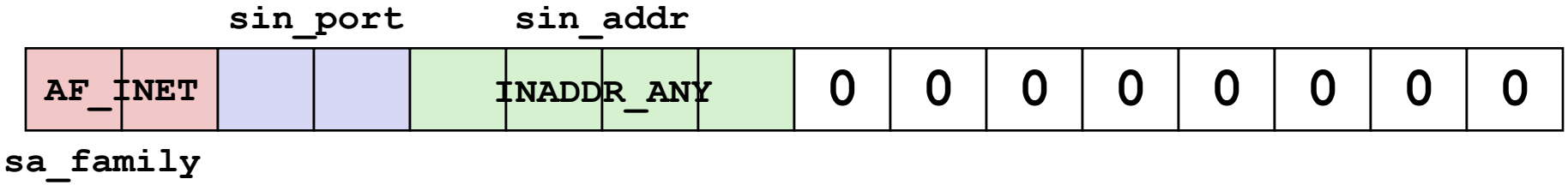
- **Handy trick that allows us to rerun the server immediately after we kill it**
  - Otherwise we would have to wait about 15 seconds
  - Eliminates “Address already in use” error from **`bind()`**
- **Strongly suggest you do this for all your servers to simplify debugging**

Echo Server: `open_listenfd`
(initialize socket address)

- Initialize socket with server port number
- Accept connection from any IP address

```
  struct sockaddr_in serveraddr; /* server's socket addr */
...
  /* listenfd will be an endpoint for all requests to port
     on any IP address for this host */
  bzero((char *) &serveraddr, sizeof(serveraddr));
  serveraddr.sin_family = AF_INET;
  serveraddr.sin_port = htons((unsigned short)port);
  serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

- IP addr and port stored in network (big-endian) byte order



Echo Server: `open_listenfd` (`bind`)

- `bind` associates the socket with the socket address we just created

```
int listenfd;                    /* listening socket */
struct sockaddr_in serveraddr; /* server's socket addr */

...
  /* listenfd will be an endpoint for all requests to port
     on any IP address for this host */
  if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
      return -1;
```

Echo Server: `open_listenfd` (`listen`)

- `listen` indicates that this socket will accept connection (`connect`) requests from clients
- `LISTENQ` is constant indicating how many pending requests allowed

```
int listenfd; /* listening socket */

...
 /* Make it a listening socket ready to accept connection requests */
    if (listen(listenfd, LISTENQ) < 0)
        return -1;
    return listenfd;
}
```

- We're finally ready to enter the main server loop that accepts and processes client connection requests.

Echo Server: Main Loop

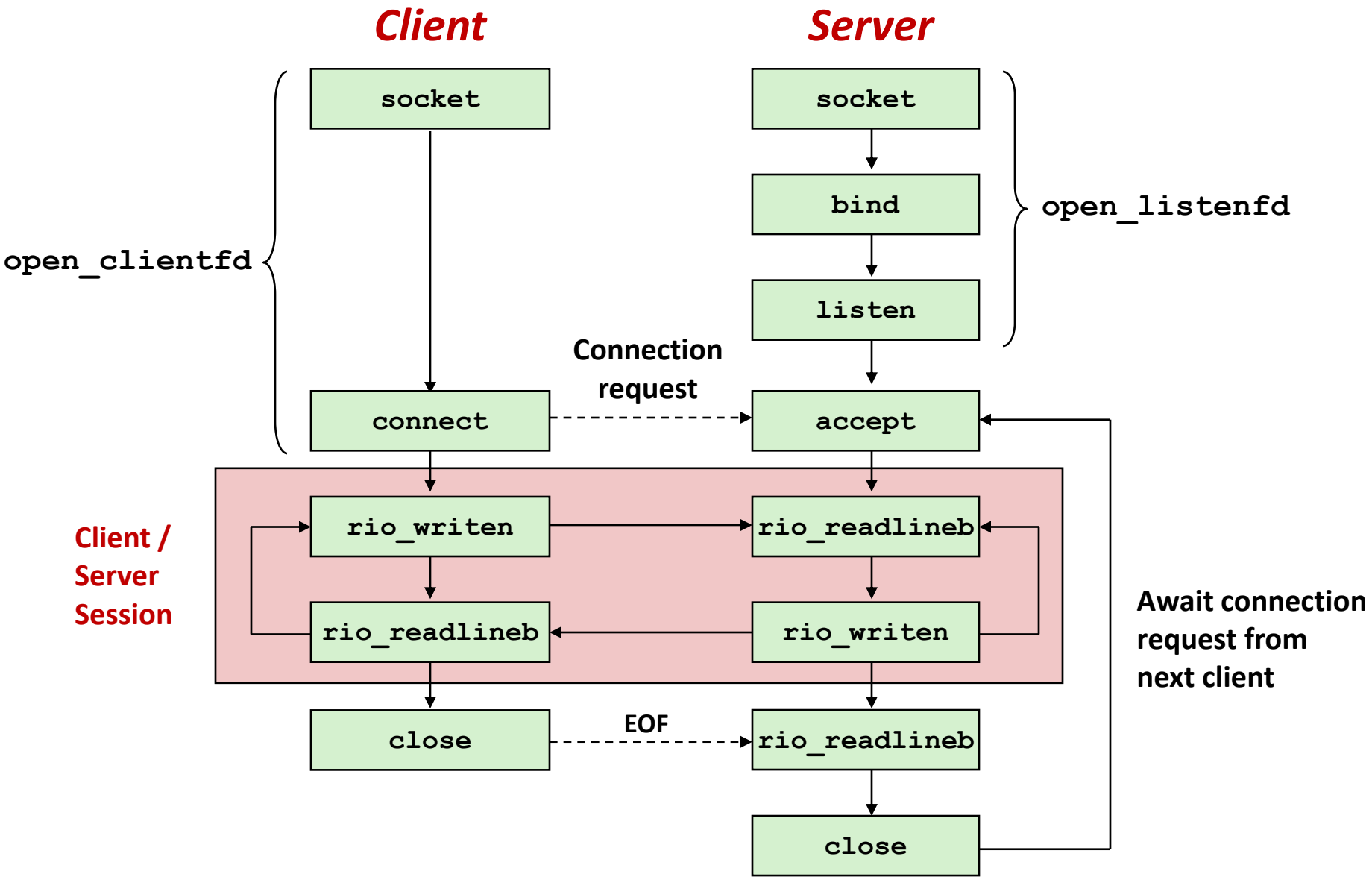
- The server loops endlessly, waiting for connection requests, then reading input from the client, and echoing the input back to the client.

```
main() {

   /* create and configure the listening socket */

   while(1) {
      /* Accept(): wait for a connection request */
      /* echo(): read and echo input lines from client til EOF */
      /* Close(): close the connection */
   }
}
```

Overview of the Sockets Interface



Echo Server: `accept`

- **`accept()` blocks waiting for a connection request**

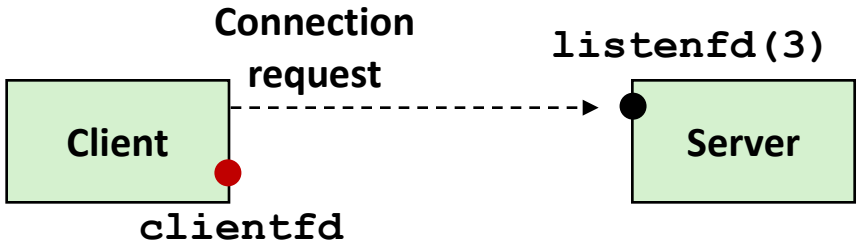
```
int listenfd; /* listening descriptor */
int connfd;   /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

clientlen = sizeof(clientaddr);
connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
```

- **`accept` returns a *connected descriptor* (`connfd`) with the same properties as the *listening descriptor* (`listenfd`)**
  - Returns when the connection between client and server is created and ready for I/O transfers
  - All I/O with the client will be done via the connected socket
- **`accept` also fills in client's IP address**

Echo Server: `accept` Illustrated

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`



2. Client makes connection request by calling and blocking in `connect`

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

Connected vs. Listening Descriptors

- **Listening descriptor**
  - End point for client connection requests
  - Created once and exists for lifetime of the server

- **Connected descriptor**
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client

- **Why the distinction?**
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to handle the request

Echo Server: Identifying the Client

- The server can determine the domain name and IP address of the client

```
struct hostent *hp;  /* pointer to DNS host entry */
char *haddrp;        /* pointer to dotted decimal string */

hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                   sizeof(clientaddr.sin_addr.s_addr), AF_INET);
haddrp = inet_ntoa(clientaddr.sin_addr);
printf("server connected to %s (%s)\n", hp->h_name, haddrp);
```

Echo Server: `echo`

- **The server uses RIO to read and echo text lines until EOF (end-of-file) is encountered.**
  - EOF notification caused by client calling **`close(clientfd)`**
  - IMPORTANT: EOF is a condition, not a particular data byte

```
void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        upper_case(buf);
        Rio_writen(connfd, buf, n);
        printf("server received %d bytes\n", n);
    }
}
```

Testing Servers Using `telnet`

- **The `telnet` program is invaluable for testing servers that transmit ASCII strings over Internet connections**
  - Our simple echo server
  - Web servers
  - Mail servers

- **Usage:**
  - **`unix> telnet <host> <portnumber>`**
  - Creates a connection with a server running on ***`<host>`*** and listening on port ***`<portnumber>`***

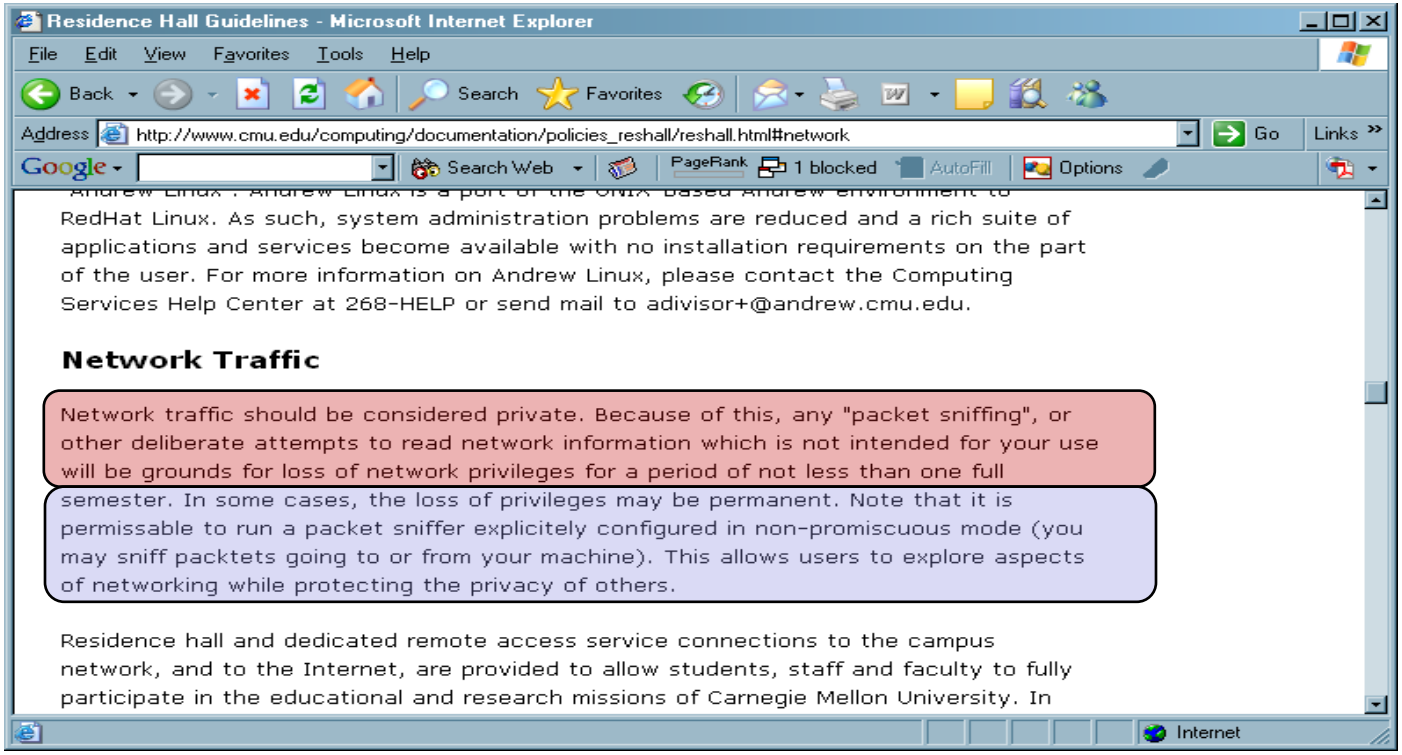
Testing the Echo Server With `telnet`

```
bass> echoserver 5000
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 5 bytes: 123
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 8 bytes: 456789

kittyhawk> telnet bass 5000
Trying 128.2.222.85...
Connected to BASS.CMCL.CS.CMU.EDU.
Escape character is '^]'.
123
123
Connection closed by foreign host.
kittyhawk> telnet bass 5000
Trying 128.2.222.85...
Connected to BASS.CMCL.CS.CMU.EDU.
Escape character is '^]'.
456789
456789
Connection closed by foreign host.
kittyhawk>
```

Packet Sniffing

- **Program That Records Network Traffic Visible at Node**
  - Promiscuous Mode
    - Record traffic that does not have this host as source or destination



For More Information

- **W. Richard Stevens, "Unix Network Programming: Networking APIs: Sockets and XTI", Volume 1, Second Edition, Prentice Hall, 1998**
  - THE network programming bible
- **Unix Man Pages**
  - Good for detailed information about specific functions
- **Complete versions of the echo client and server are developed in the text**
  - Available from **`csapp.cs.cmu.edu`**
  - You should compile and run them for yourselves to see how they work
  - Feel free to borrow any of this code