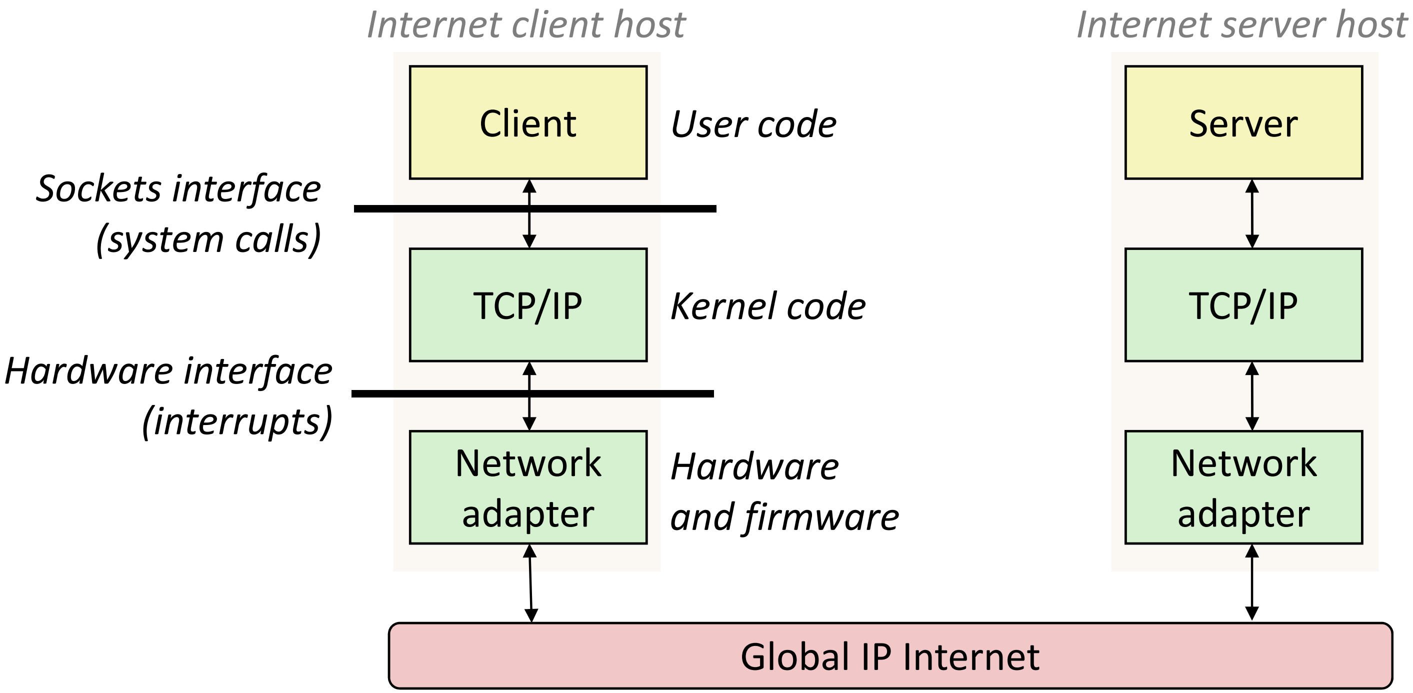
INTERNET

L4: Internet

Last Lecture Re-cap: IP Internet

- Based on the TCP/IP protocol family
  - IP (Internet protocol) :
    - Provides *basic naming scheme (DNS)* and unreliable *delivery capability* of packets (datagrams) from host-to-host
  - UDP (Unreliable Datagram Protocol)
    - Uses IP to provide unreliable datagram delivery from *process-to-process*
  - TCP (Transmission Control Protocol)
    - Uses IP to provide *reliable* byte streams from process-to-process over connections
- Accessed via a mix of Unix file I/O and functions from the *sockets interface*

Hardware and Software Organization of an Internet Application

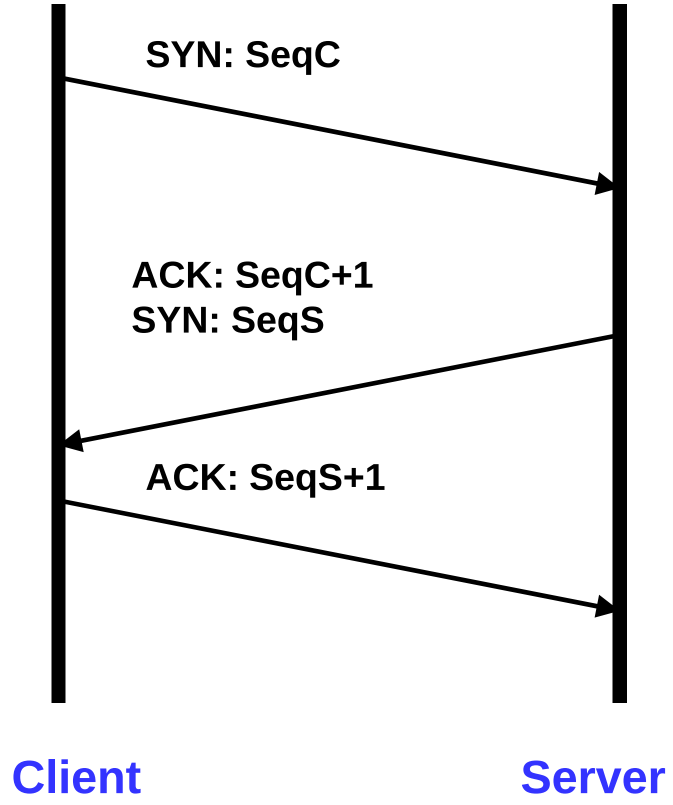


Today's Lecture

- TCP
  - Connection establishment, flow control, reliability, congestion control
- I/O
  - Unix I/O, (custom) RIO, standard I/O

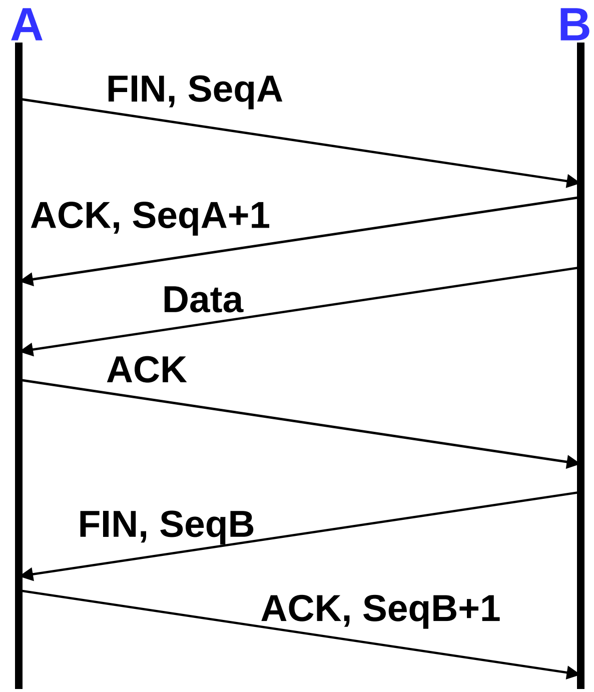
Establishing Connection: Three-Way handshake

- Each side notifies other of starting sequence number it will use for sending
  - Why not simply chose 0?
    - Must avoid overlap with earlier incarnation
    - Security issues
- Each side acknowledges other's sequence nu mber
  - SYN-ACK: Acknowledge sequence number + 1
- Can combine second SYN with first ACK

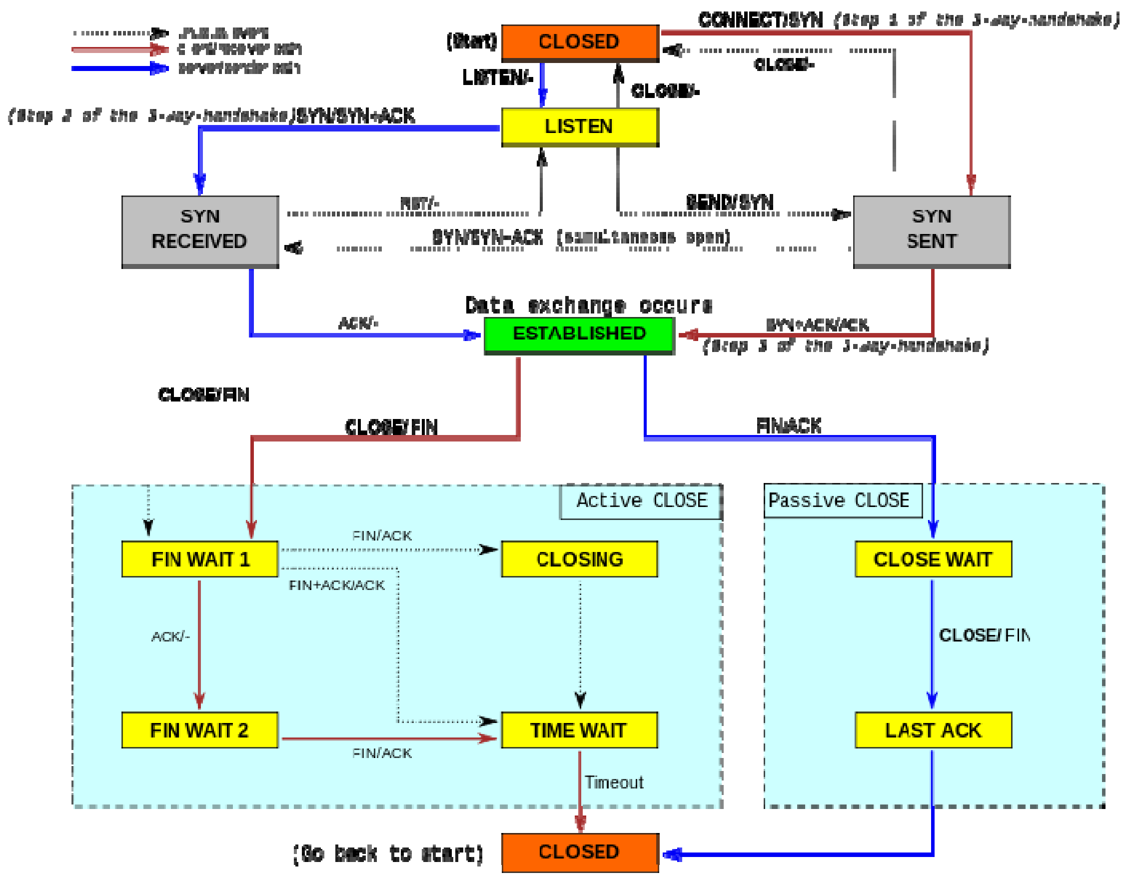


Tearing Down Connection

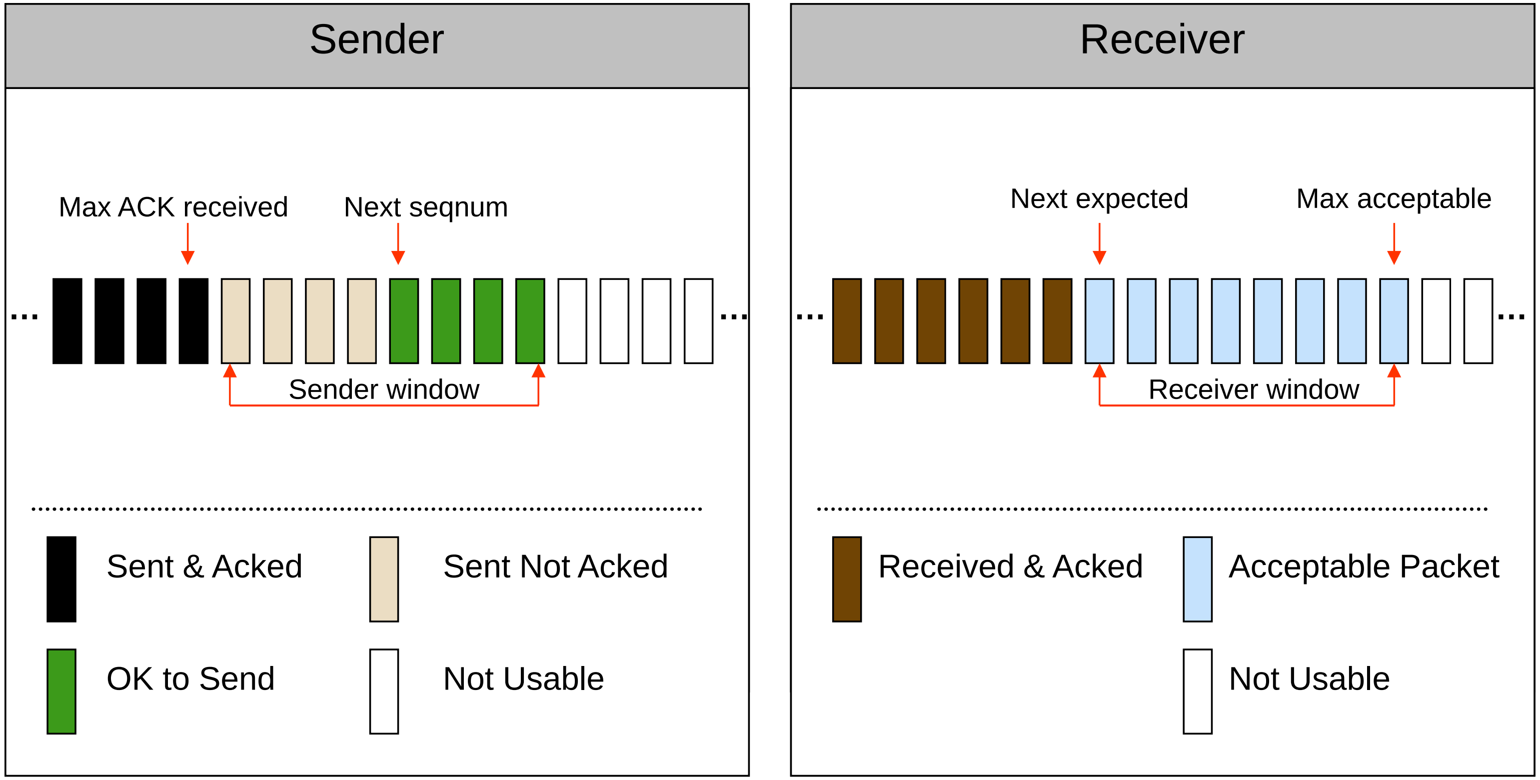
- Either side can initiate tear down
  - Send FIN signal
  - "I'm not going to send any more data"
- Other side can continue sending data
  - Half open connection
  - Must continue to acknowledge
- Acknowledging FIN
  - Acknowledge last sequence number + 1



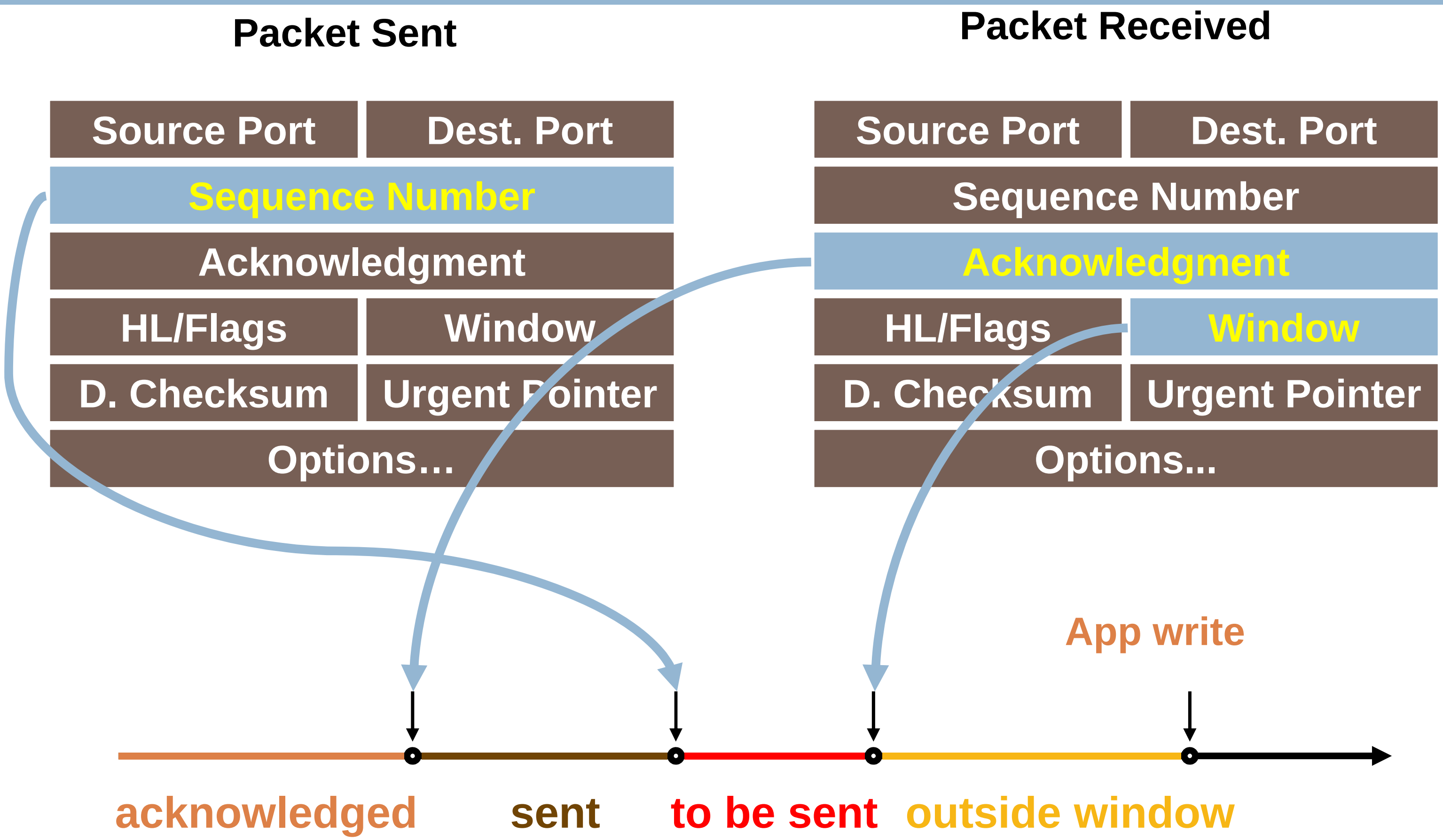
TCP state transition (connection state)



Sender/Receiver State



Window Flow Control: Send Side

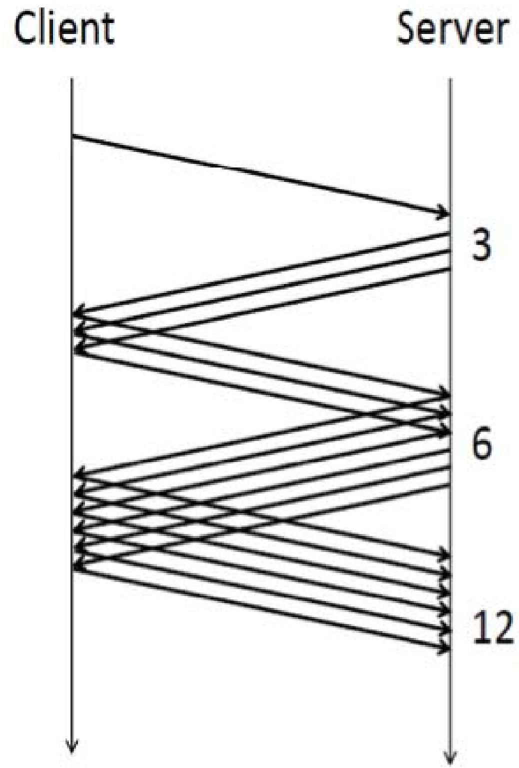


TCP congestion control

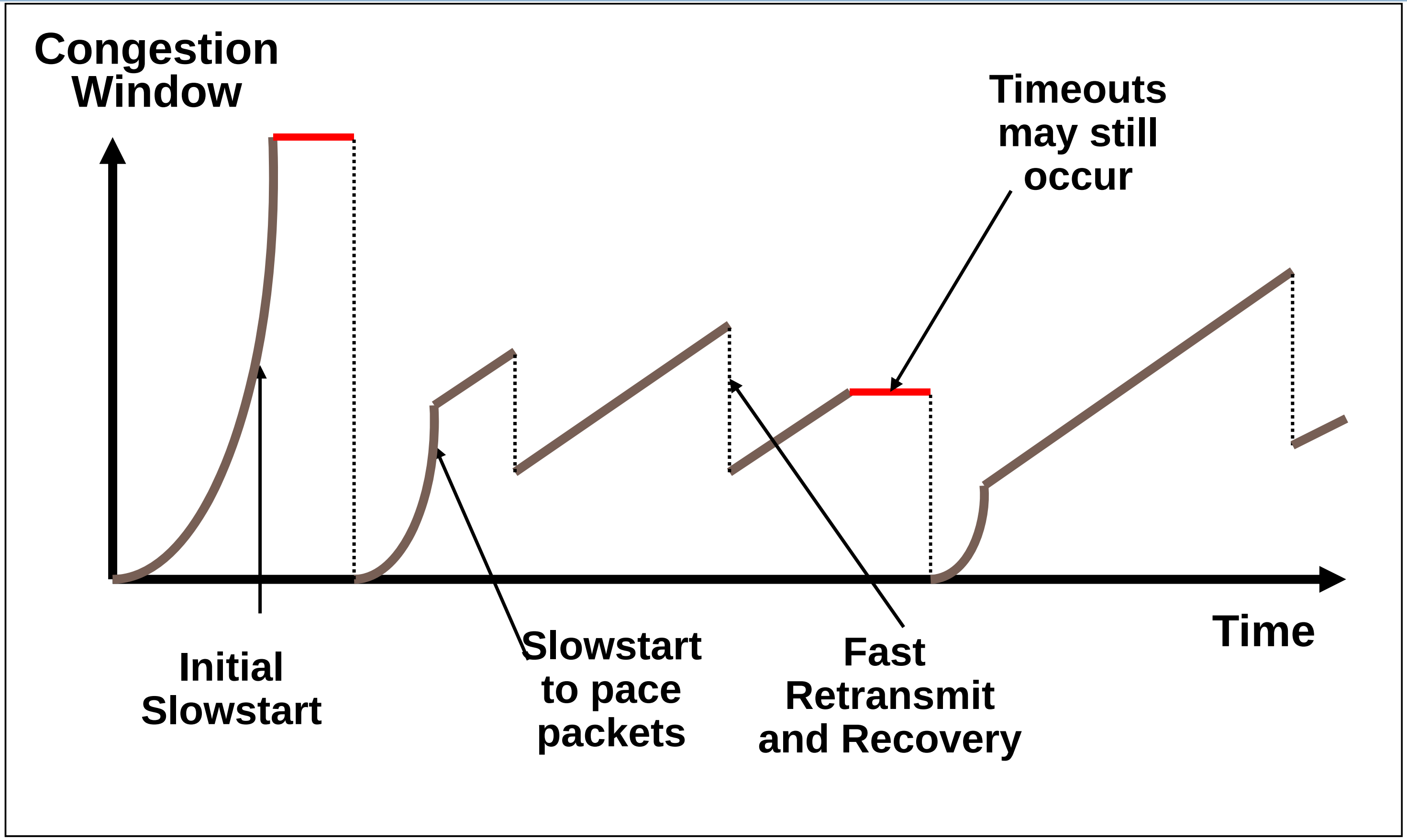
- Need to share network resources.
- But neither the sender or the receiver knows how much b/w is available.
- How much should we send?

TCP congestion control

- Slow start
  - Increase the congestion window +1 for every ack.
- Fast recovery
  - On detection of dropped packet (dup ack) reduce the congestion window by half.
    - Called multiplicative decrease
- Congestion avoidance
  - Increase 1 every RTT
    - Called additive increase
- Details in computer networks course



TCP Saw Tooth Behavior



Important Lessons

- TCP state diagram → setup/teardown
  - Making sure both sides end up in same state
- TCP congestion control
  - Need to share some resources without knowing their current state
  - Good example of adapting to network performance
- Sliding window flow control
  - Addresses buffering issues and keeps link utilized
  - Need to ensure that distributed resources that are known about aren't overloaded

Today

- TCP
- I/O
  - Unix I/O
  - RIO (robust I/O) package
  - Standard I/O

Unix Files

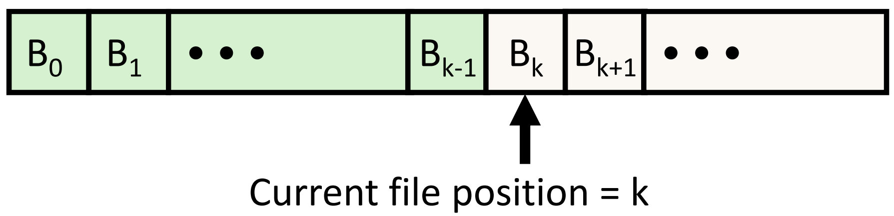
- A Unix *file* is a sequence of *m* bytes:
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$
- All I/O devices are represented as files:
  - **`/dev/sda2`** (**`/usr`** disk partition)
  - **`/dev/tty2`** (terminal)
- Even the kernel is represented as a file:
  - **`/dev/kmem`** (kernel memory image)
  - **`/proc`** (kernel data structures)

Unix File Types

- Regular file
  - File containing user/app data (binary, text, whatever)
  - OS does not know anything about the format
    - other than “sequence of bytes”, akin to main memory
- Directory file
  - A file that contains the names and locations of other files
- Character special and block special files
  - Terminals (character special) and disks (block special)
- FIFO (named pipe)
  - A file type used for inter-process communication
- Socket
  - A file type used for network communication between processes

Unix I/O

- Key Features
  - Elegant mapping of files to devices allows kernel to export simple interface called Unix I/O
  - Important idea: All input and output is handled in a consistent and uniform way
- Basic Unix I/O operations (system calls):
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the ***current file position*** (seek)
    - indicates next offset into file to read or write
    - **`lseek()`**



Opening Files

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

- Returns a small identifying integer *file descriptor*
  - **`fd == -1`** indicates that an error occurred
- Each process created by a Unix shell begins life with three open files associate d with a terminal:
  - 0: standard input
  - 1: standard output
  - 2: standard error

Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;     /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
   perror("close");
   exit(1);
}
```

- Closing an already closed file is a recipe for disaster in threaded programs (more on this later)
- Moral: Always check return codes, even for seemingly benign functions such as `close()`

Reading Files

- Reading a file copies bytes from the current file position to memory, and then up dates file position

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

- Returns number of bytes read from file `fd` into `buf`
  - Return type **`ssize_t`** is signed integer
  - **`nbytes < 0`** indicates that an error occurred
  - ***Short counts*** (**`nbytes < sizeof(buf)`**) are possible and are not errors!

Writing Files

- Writing a file copies bytes from memory to the current file position, and then u pdates current file position

```
char buf[512];
int fd;       /* file descriptor */
int nbytes;   /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

- Returns number of bytes written from `buf` to file `fd`
  - **`nbytes < 0`** indicates that an error occurred
  - As with reads, short counts are possible and are not errors!

Simple Unix I/O example

- Copying standard in to standard out, one byte at a time

```
#include "csapp.h"

int main(void)
{
    char c;

    while(Read(STDIN_FILENO, &c, 1) != 0)
        Write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

cpstdin.c

Note the use of error handling wrappers for read and write (Appendix A).

Dealing with Short Counts

- Short counts can occur in these situations:
  - Encountering (end-of-file) EOF on reads
  - Reading text lines from a terminal
  - Reading and writing network sockets or Unix pipes

- Short counts never occur in these situations:
  - Reading from disk files (except for EOF)
  - Writing to disk files

- One way to deal with short counts in your code:
  - Use the RIO (Robust I/O) package from your textbook's **`csapp.c`** file (Appendix B)

Today

- TCP
- I/O
  - Unix I/O
  - RIO (robust I/O) package
  - Standard I/O

The RIO Package

- RIO is a set of wrappers that provide efficient and robust I/O in apps, such as network programs that are subject to short counts

- RIO provides two different kinds of functions
  - Unbuffered input and output of binary data
    - **`rio_readn`** and **`rio_writen`**
  - Buffered input of binary data and text lines
    - **`rio_readlineb`** and **`rio_readnb`**
    - Buffered RIO routines are thread-safe and can be interleaved arbitrarily on the same descriptor

- Download from http://csapp.cs.cmu.edu/public/code.html
  → **`src/csapp.c`** and **`include/csapp.h`**

Unbuffered RIO Input and Output

- Same interface as Unix `read` and `write`
- Especially useful for transferring data on network sockets

```
#include "csapp.h"

ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

Return: num. bytes transferred if OK, 0 on EOF (`rio_readn` only), -1 on error

- **`rio_readn`** returns short count only if it encounters EOF
  - Only use it when you know how many bytes to read
- **`rio_writen`** never returns a short count
- Calls to **`rio_readn`** and **`rio_writen`** can be interleaved ar bitrarily on the same descriptor

Implementation of `rio_readn`

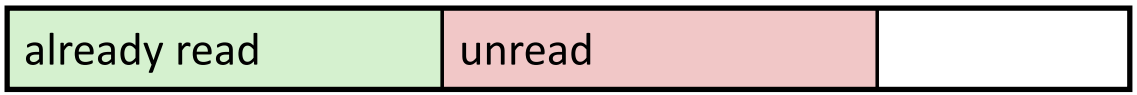
```
/*
 * rio_readn - robustly read n bytes (unbuffered)
 */
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nread = read(fd, bufp, nleft)) < 0) {
            if (errno == EINTR) /* interrupted by sig handler return */
                nread = 0;      /* and call read() again */
            else
                return -1;      /* errno set by read() */
        }
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);         /* return >= 0 */
}
```

csapp.c

Buffered I/O: Motivation

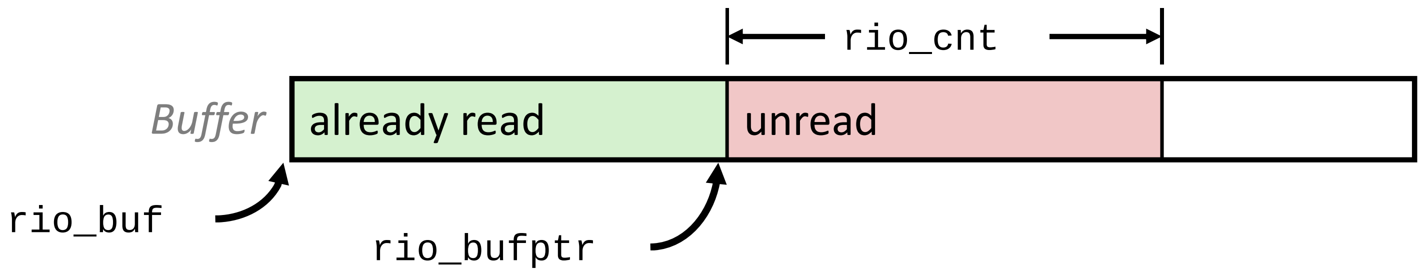
- Applications often read/write one character at a time
  - `getc, putc, ungetc`
  - `gets, fgets`
    - Read line of text on character at a time, stopping at newline
- Implementing as Unix I/O calls expensive
  - `read` and `write` require Unix kernel calls
    - > 10,000 clock cycles
- Solution: Buffered read
  - Use Unix `read` to grab block of bytes
  - User input functions take one byte at a time from buffer
    - Refill buffer when empty

Buffer

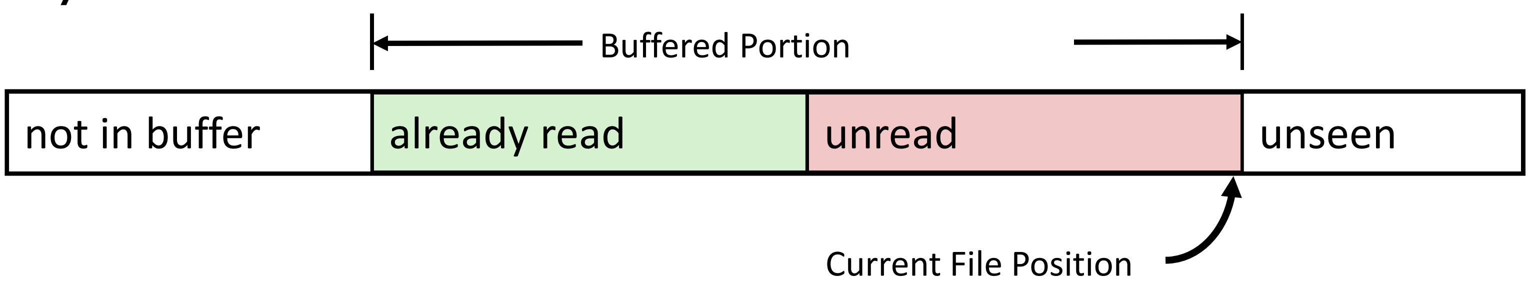


Buffered I/O: Implementation

- For reading from file
- File has associated buffer to hold bytes that have been r ead from file but not yet read by user code

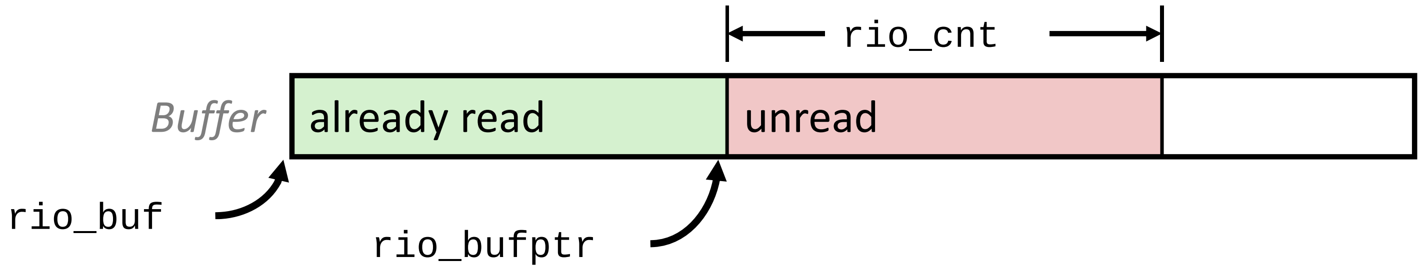


- Layered on Unix file:



Buffered I/O: Declaration

- All information contained in `struct`




```
typedef struct {
    int rio_fd;                /* descriptor for this internal buf */
    int rio_cnt;               /* unread bytes in internal buf */
    char *rio_bufptr;          /* next unread byte in internal buf */
    char rio_buf[RIO_BUFSIZE]; /* internal buffer */
} rio_t;
```

Buffered RIO Input Functions

- Efficiently read text lines and binary data from a file partially cached in an internal memory buffer

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);

                        Return: num. bytes read if OK, 0 on EOF, -1 on error
```

- **`rio_readlineb`** reads a text line of up to **`maxlen`** bytes from file **`fd`** and stores the line in **`usrbuf`**
  - Especially useful for reading text lines from network sockets
- Stopping conditions
  - **`maxlen`** bytes read
  - EOF encountered
  - Newline (‘**`\n`**’) encountered

Buffered RIO Input Functions (cont)

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
```

Return: num. bytes read if OK, 0 on EOF, -1 on error

- **`rio_readnb`** reads up to **`n`** bytes from file **`fd`**
- Stopping conditions
  - **`maxlen`** bytes read
  - EOF encountered
- Calls to **`rio_readlineb`** and **`rio_readnb`** can be interleaved arbitrarily on the same descriptor
  - Warning: Don't interleave with calls to **`rio_readn`**

RIO Example

- Copying the lines of a text file from standard input to standard output

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int n;
    rio_t rio;
    char buf[MAXLINE];

    Rio_readinitb(&rio, STDIN_FILENO);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
        Rio_writen(STDOUT_FILENO, buf, n);
    exit(0);
}                                                   cpfile.c
```

Standard I/O Functions

- The C standard library (`libc.so`) contains a collection of higher-level *standard I/O* functions
  - Documented in Appendix B of K&R.

- Examples of standard I/O functions:
  - Opening and closing files (**`fopen`** and **`fclose`**)
  - Reading and writing bytes (**`fread`** and **`fwrite`**)
  - Reading and writing text lines (**`fgets`** and **`fputs`**)
  - Formatted reading and writing (**`fscanf`** and **`fprintf`**)

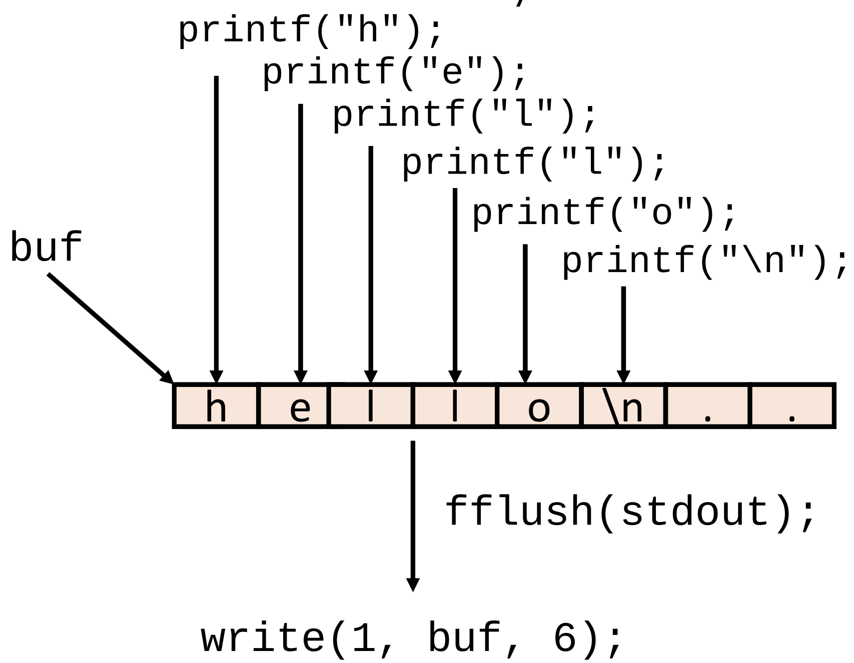
Standard I/O Streams

- Standard I/O models open files as *streams*
  - Abstraction for a file descriptor and a buffer in memory.
  - Similar to buffered RIO
- C programs begin life with three open streams (defined in `stdio.h`)
  - **`stdin`** (standard input)
  - **`stdout`** (standard output)
  - **`stderr`** (standard error)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

Buffering in Standard I/O

- Standard I/O functions use buffered I/O



- Buffer flushed to output fd on "\n" or `fflush()` call

Standard I/O Buffering in Action

- You can see this buffering in action for yourself, using the always fascinating Unix `strace` program:

```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

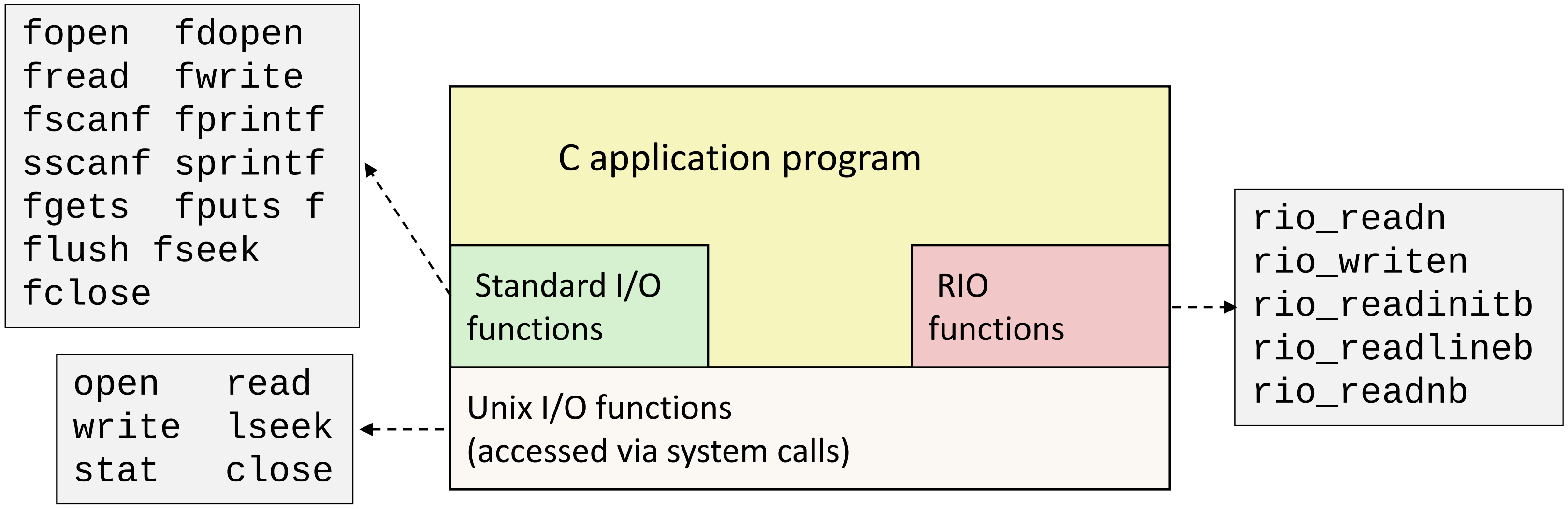
```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6)                = 6
...
exit_group(0)                         = ?
```

Today

- TCP
- Unix I/O
- RIO (robust I/O) package
- Standard I/O
- Conclusions

Unix I/O vs. Standard I/O vs. RIO

- Standard I/O and RIO are implemented using low-level Unix I/O



- Which ones should you use in your programs?

Choosing I/O Functions

- General rule: use the highest-level I/O functions you can
  - Many C programmers are able to do all of their work using th e standard I/O functions

- When to use standard I/O
  - When working with disk or terminal files
- When to use raw Unix I/O
  - Inside signal handlers, because Unix I/O is async-signal-safe.
  - In rare cases when you need absolute highest performance.
- When to use RIO
  - When you are reading and writing network sockets.
  - Avoid using standard I/O on sockets.

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 Interne t *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 anot her 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 fro m 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 (“l” = 32 bits, “s” = 16 bits)

`htonl`: convert uint32_t from host to network byte order
`htons`: convert uint16_t from host to network byte order
`ntohl`: convert uint32_t from network to host byte order
`ntohs`: convert uint16_t from network to host byte order

Dotted Decimal Notation

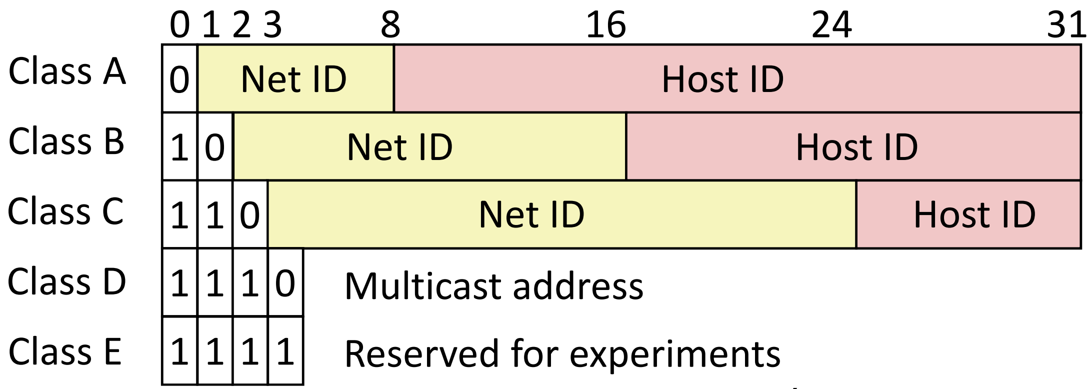
- By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period
  - IP address: **`0x8002C2F2 = 128.2.194.242`**

- 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

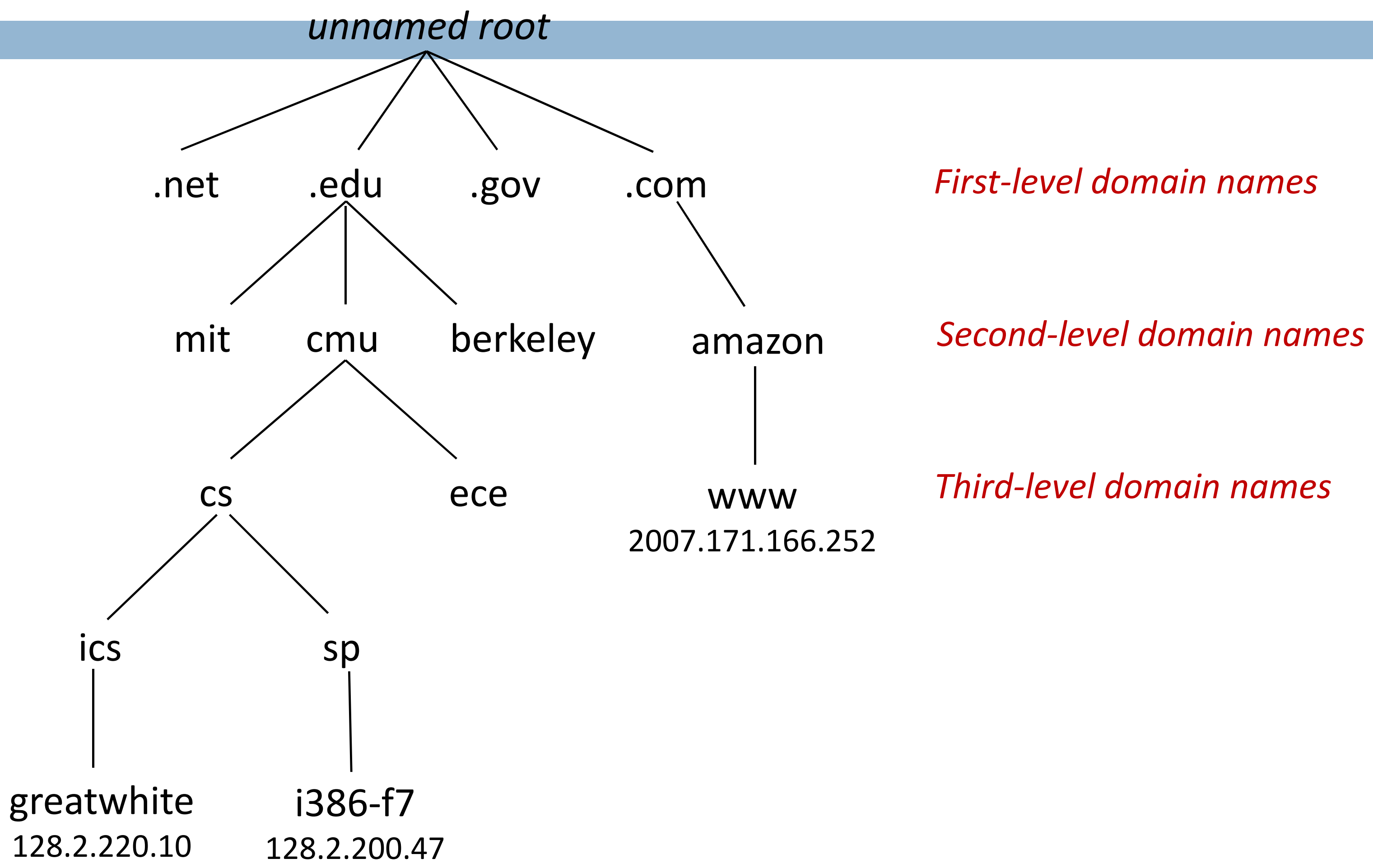
- IP (V4) Address space divided into classes:



- Network ID Written in form w.x.y.z/n
  - n = number of bits in host address
  - E.g., CMU written as 128.2.0.0/16
    - Class B address
- Unrouted (private) IP addresses:

  10.0.0.0/8   172.16.0.0/12   192.168.0.0/16

Internet Domain Names



Domain Naming System (DNS)

- The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called *DNS*
  - 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 an
d
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:
    - **`greatwhile.ics.cs.cmu.edu`** maps to **`128.2.220.10`**
  - 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:
    - **`google.com`** maps to multiple IP addresses
  - Some valid domain names don't map to any IP address:
    - for example: **`ics.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));
    }
}
```

Using DNS Program

```
linux> ./dns greatwhite.ics.cs.cmu.edu
official hostname: greatwhite.ics.cs.cmu.edu
address 128.2.220.10

linux> ./dns 128.2.220.11
official hostname: ANGELSHARK.ICS.CS.CMU.EDU
address: 128.2.220.11

linux> ./dns www.google.com
official hostname: www.l.google.com
alias: www.google.com
address: 72.14.204.99
address: 72.14.204.103
address: 72.14.204.104
address: 72.14.204.147
linux> dig +short -x 72.14.204.103
iad04s01-in-f103.1e100.net.
```

Querying DIG

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

```
linux> dig +short greatwhite.ics.cs.cmu.edu
128.2.220.10
linux> dig +short -x 128.2.220.11
ANGELSHARK.ICS.CS.CMU.EDU.
linux> dig +short google.com
72.14.204.104
72.14.204.147
72.14.204.99
72.14.204.103
linux> dig +short -x 72.14.204.103
iad04s01-in-f103.1e100.net.
```

More Exotic Features of DIG

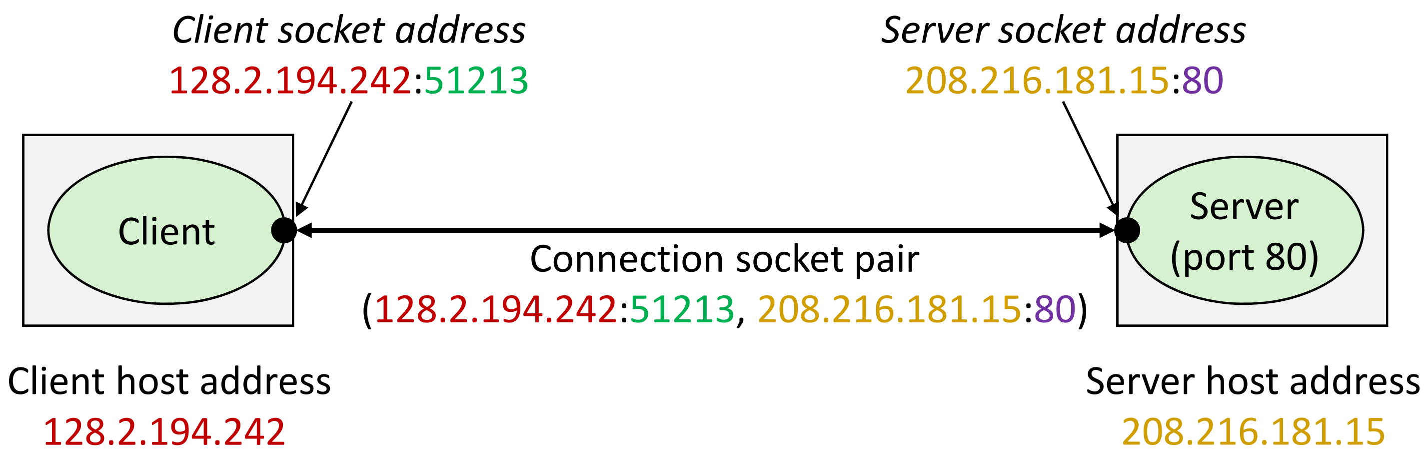
- Provides more information than you would ever want about DNS

```
linux> dig www.phys.msu.ru a +trace
128.2.220.10

linux> dig www.google.com a +trace
```

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



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 p ort 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 referre d 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 co
mprehensive list of the port ma
ppings 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

Sockets Interface

- Created in the early 80's as part of the original B
erkeley distribution of Unix that contained an earl
y version of the Internet protocols

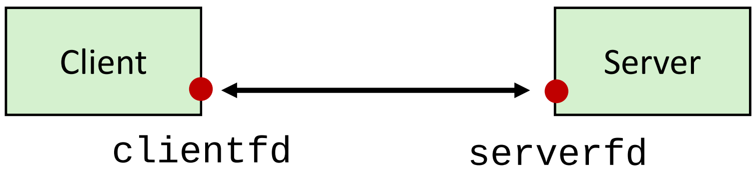
- Provides a user-level interface to the network

- 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 appl ication read/write from/to the network
    - ***Remember:*** All Unix I/O devices, including networks, are modeled a s files
- Clients and servers communicate with each other by readi ng 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

Example: Echo Client and Server

On Client | On Server

```
greatwhite> ./echoserveri 15213
```

```
linux> echoclient greatwhite.ics.cs.cmu.edu 15213
```

```
server connected to BRYANT-TP4.VLSI.CS.CMU.EDU (128.2.
213.29), port 64690
```

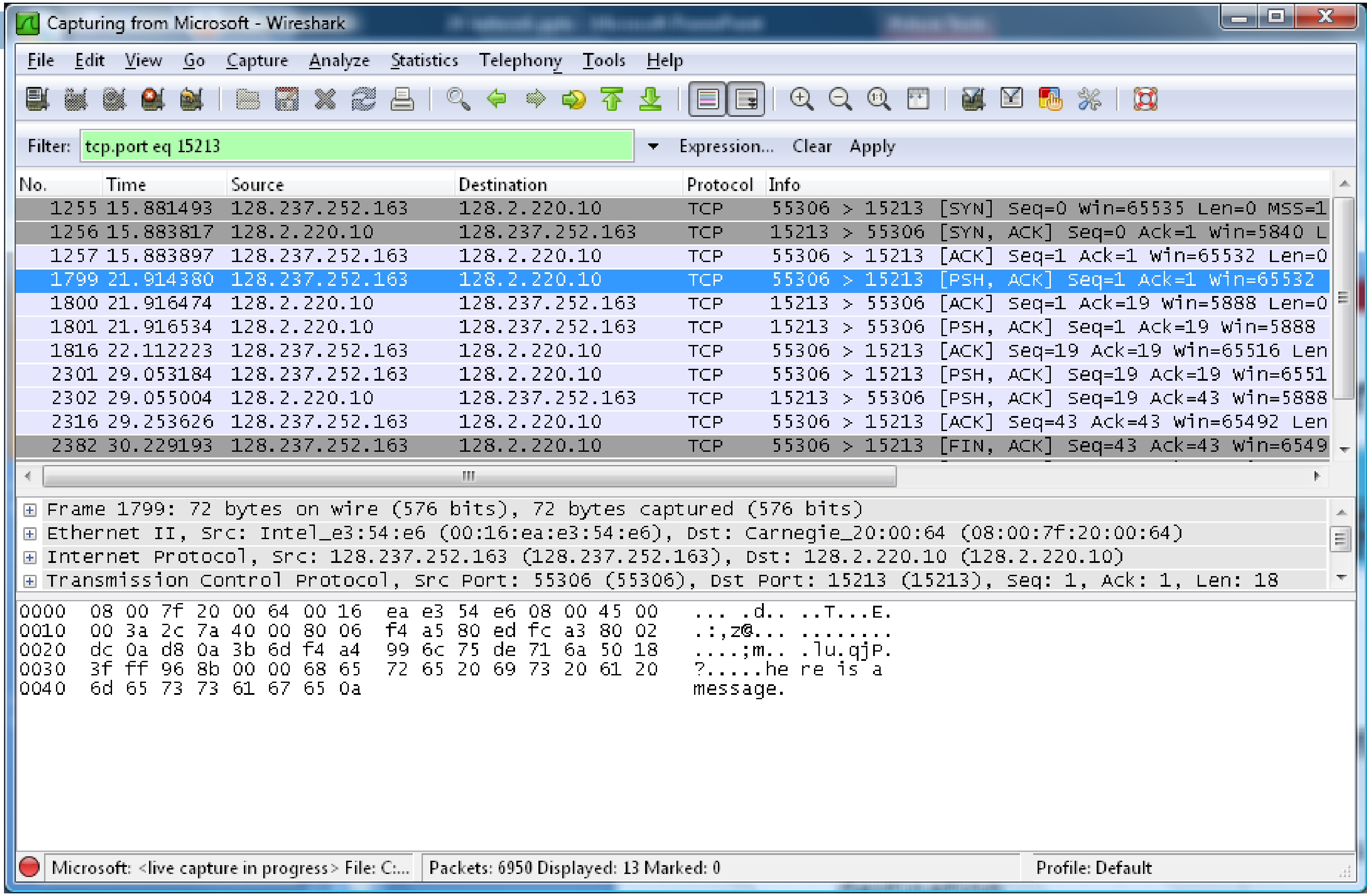
```
type: hello there
```

```
server received 12 bytes
```

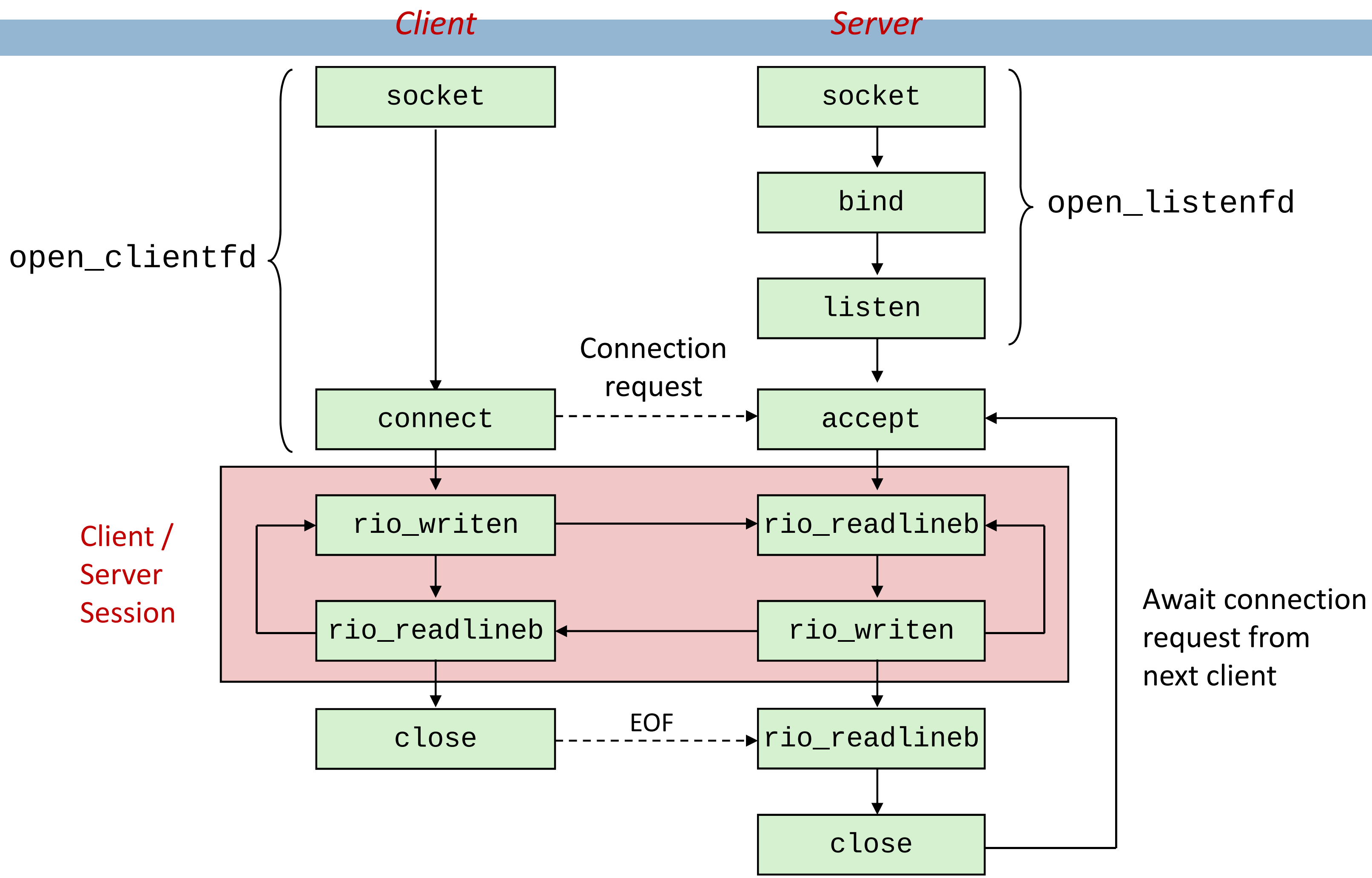
```
echo: HELLO THERE
type: ^D
```

```
Connection closed
```

Watching Echo Client / Server



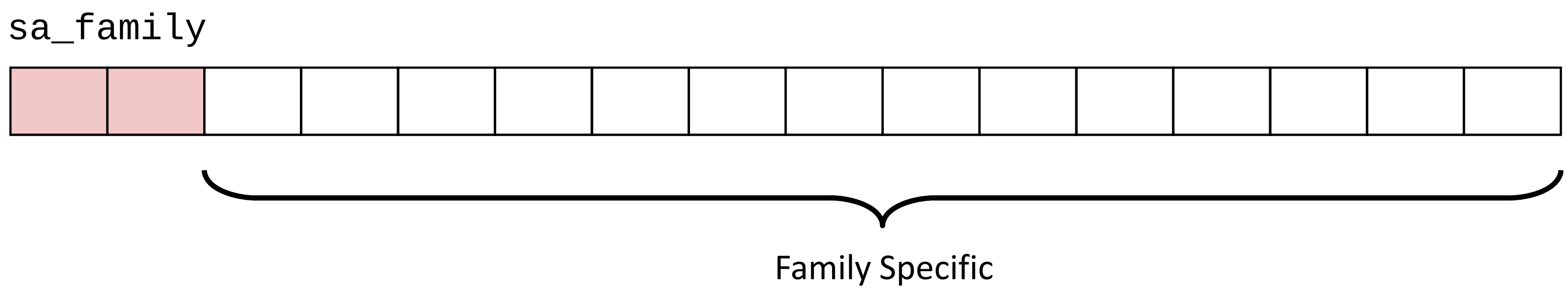
Overview of the Sockets Interface



Socket Address Structures

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

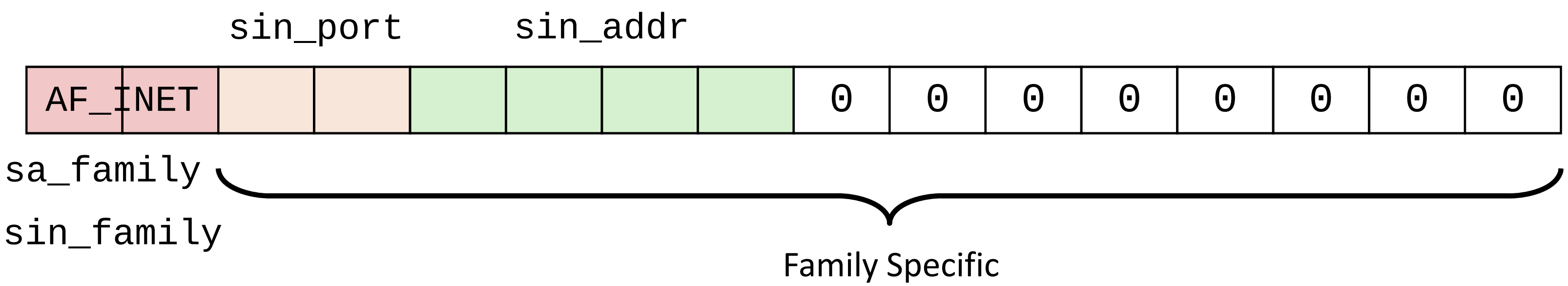
```c
struct sockaddr {
  unsigned short  sa_family;    /* protocol family */
  char            sa_data[14];  /* address data.   */
};
```



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) */
};
```



Echo Client Main Routine

```
#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("type:"); 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("type:"); fflush(stdout);
    }
    Close(clientfd);
    exit(0);
}
```

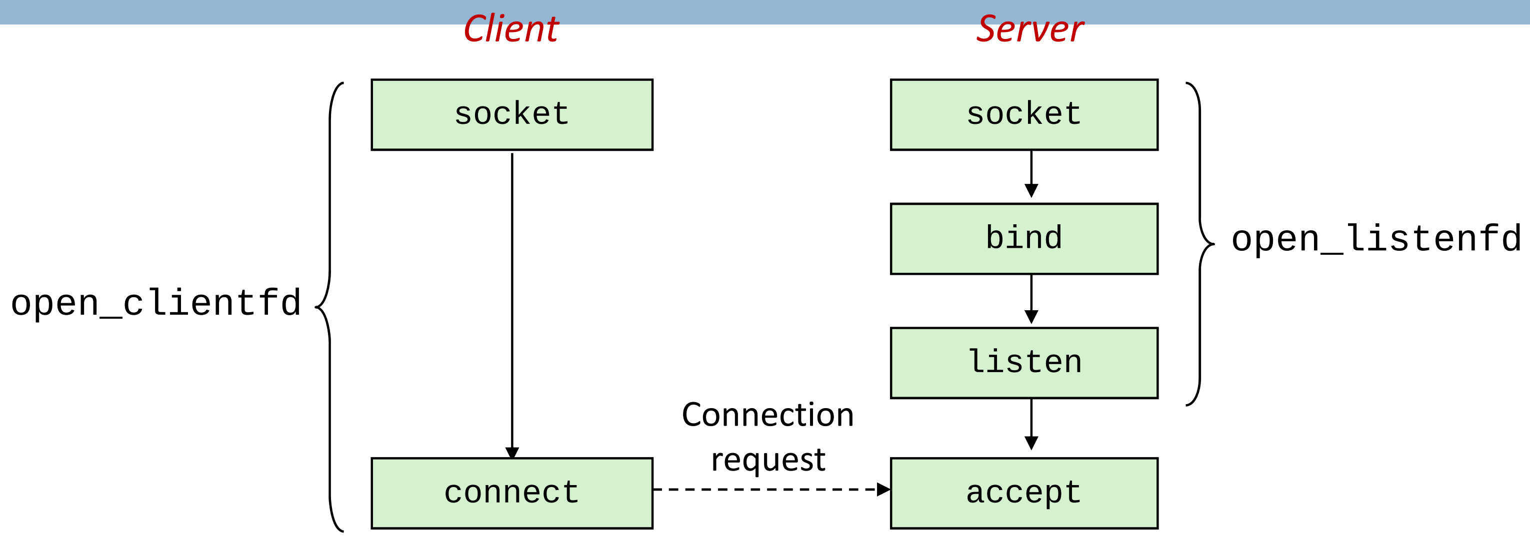
Read input line

Send line to ser ver

Receive line fro m server

Print server response

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);
```

Check this out!

A Careful Look at bcopy Arguments

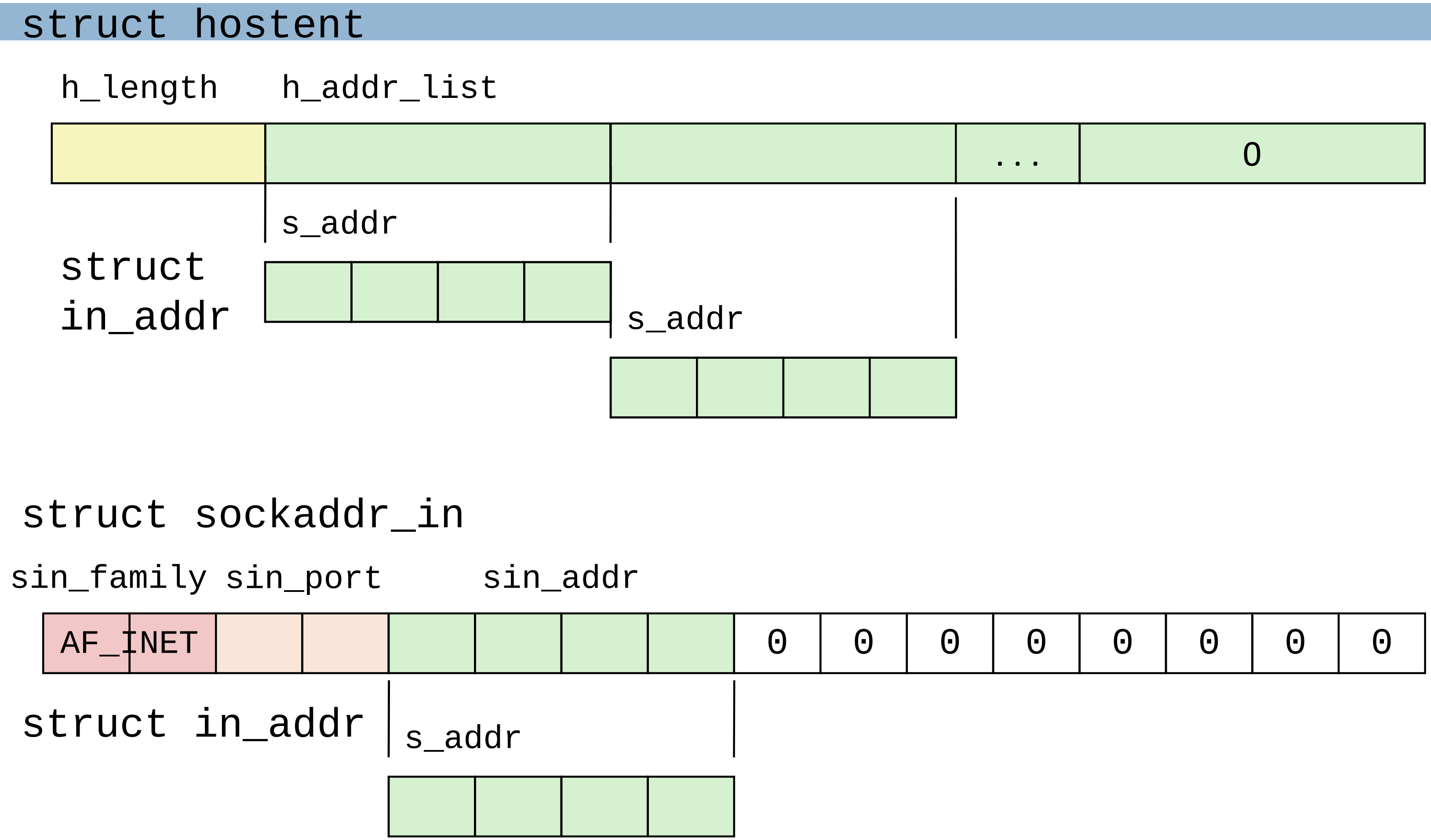
```
/* DNS host entry structure */
struct hostent {
   . . .
   int    h_length;      /* length of an address, in bytes */
   char   **h_addr_list; /* null-terminated array of in_addr structs */
};
```

```
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);
```

Bcopy Argument Data Structures



Echo Client: `open_clientfd` (`connect`)

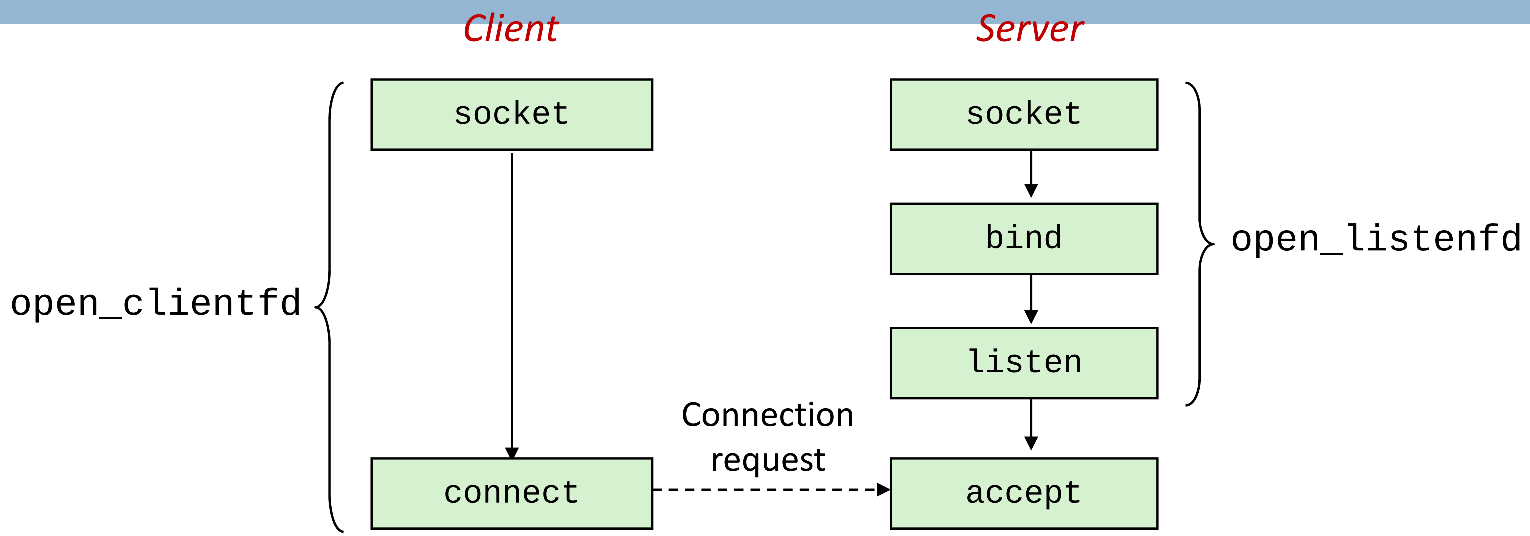
- Finally the client creates a connection with the server
  - Client process suspends (blocks) until the connection is created
  - After resuming, the client is ready to begin exchanging messages with the server via Unix I/O calls on descriptor **`clientfd`**

```
  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;
    unsigned short client_port;

    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);
        client_port = ntohs(clientaddr.sin_port);
        printf("server connected to %s (%s), port %u\n",
                hp->h_name, haddrp, client_port);
        echo(connfd);
        Close(connfd);
    }
}
```

Overview of the Sockets Interface



- Office Telephone Analogy for Server
  - Socket: Buy a phone
  - Bind: Tell the local administrator what number you want to use
  - Listen: Plug the phone in
  - Accept: Answer the phone when it rings

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 pr otocols
  - **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);
```

		sin_port		sin_addr											
AF_INET				INADDR_ANY				0	0	0	0	0	0	0	0

sa_family

sin_family

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 connec tion (`connect`) requests from clients
- `LISTENQ` is constant indicating how many pending r equests allowed

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

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

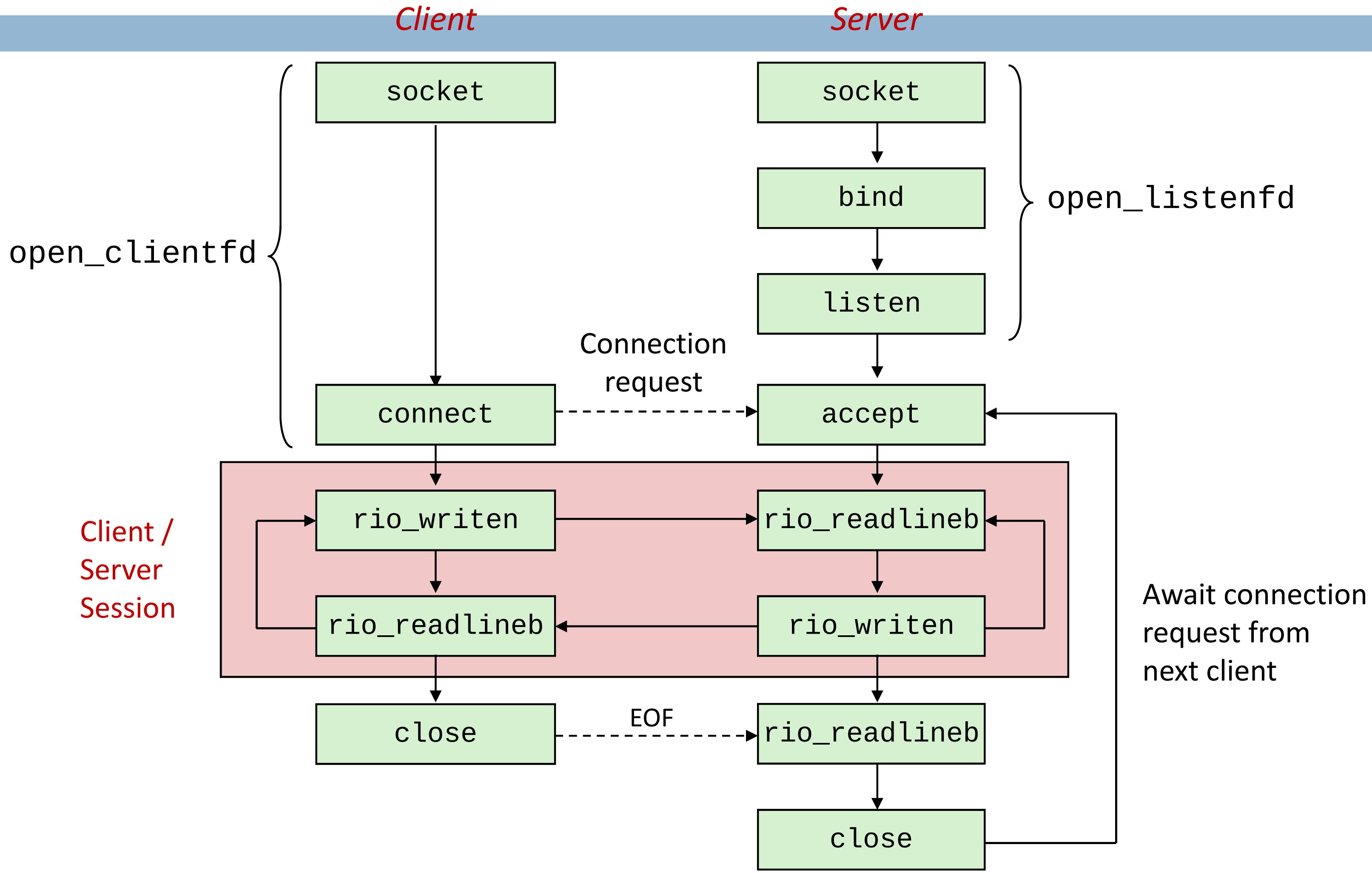
- We're finally ready to enter the main server loop th at accepts and processes client connection requests.

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



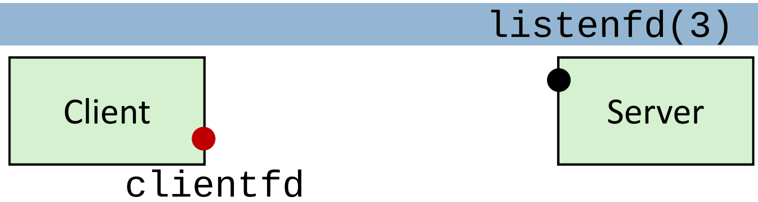
Echo Server: `accept`

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

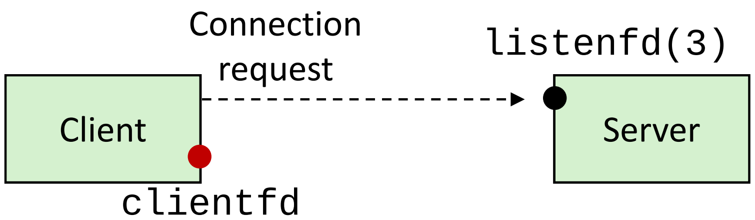
```
int listenfd; /* listening descriptor */
int connfd;   /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen;

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

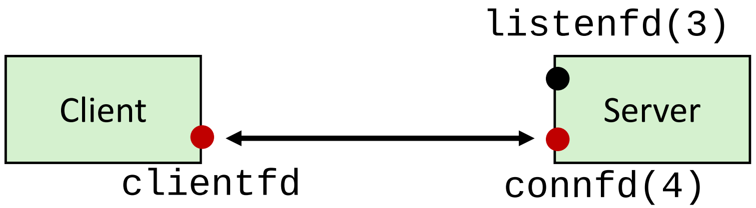
Echo Server: `accept` Illustrated



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 m any client connections simultaneously
    - E.g., Each time we receive a new request, we fork a child to han dle the request

Echo Server: Identifying the Client

- The server can determine the domain name, IP ad dress, and port of the client

```
struct hostent *hp;  /* pointer to DNS host entry */
char *haddrp;        /* pointer to dotted decimal string */
unsigned short client_port;
hp = Gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                        sizeof(clientaddr.sin_addr.s_addr), AF_INET);
haddrp = inet_ntoa(clientaddr.sin_addr);
client_port = ntohs(clientaddr.sin_port);
printf("server connected to %s (%s), port %u\n",
       hp->h_name, haddrp, client_port);
```

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)`**

```
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 A SCII 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 por t ***`<portnumber>`***

Testing the Echo Server With `telnet`

```
greatwhite> echoserver 15213

linux> telnet greatwhite.ics.cs.cmu.edu 15213
Trying 128.2.220.10...
Connected to greatwhite.ics.cs.cmu.edu.
Escape character is '^]'.
hi there
HI THERE
```

For More Information

- W. Richard Stevens, “Unix Network Programming: Networking APIs: Sockets and XTI”, Volume 1, Sec ond Edition, Prentice Hall, 1998
  - THE network programming bible
- Unix Man Pages
  - Good for detailed information about specific function s
- Complete versions of the echo client and server ar e developed in the text
  - Updated versions linked to course website
  - Feel free to use this code in your assignments