CSE320 System Fundamentals II Unix Socket API

TONY MIONE

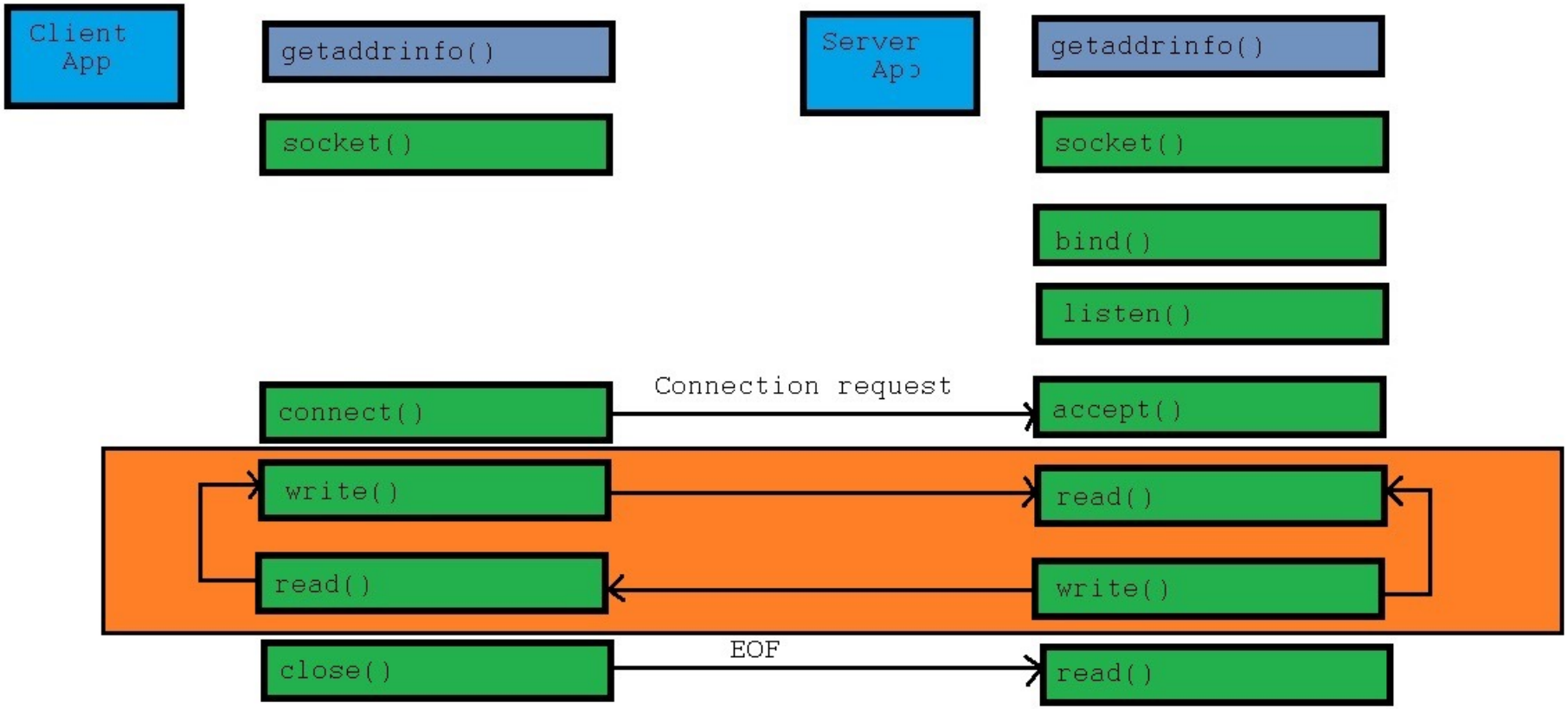
Topics

- Connections – Programmer's Perspective
- Socket API
  - Data Structures
  - API Calls
    - Informational calls
    - Connection calls
- Overview of Web servers
- Testing Servers

Connections

- **Active** – process (usually a ‘client’) tries to open a connection with a remote process (usually a ‘server’)
- **Passive** – A process (usually a ‘server’) waits or ‘listens’ for connections from other remote processes (usually ‘client’ programs)
- Different sets of API calls are needed for each type of connection

Programmer's Perspective



Socket API: Data Structures

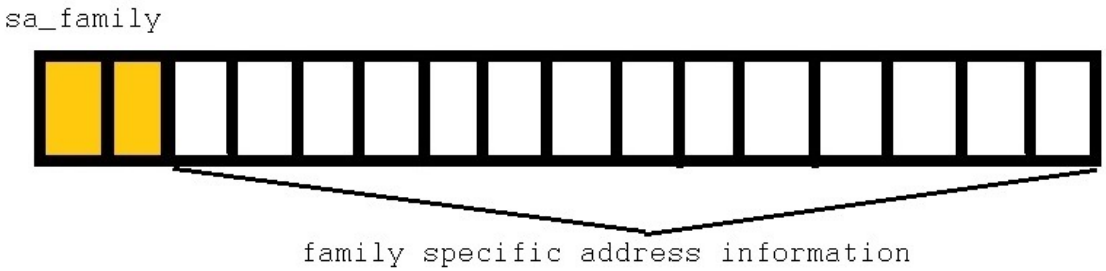
- sockaddr –
  - C struct
  - Carries information about an address tied to a socket
  - Generic form – Used in various socket API calls
  - Family specific forms – Used for different network types

Socket API : sockaddr

Generic struct

- Necessary as the C language did not have (void *) pointers when the socket API was first developed.

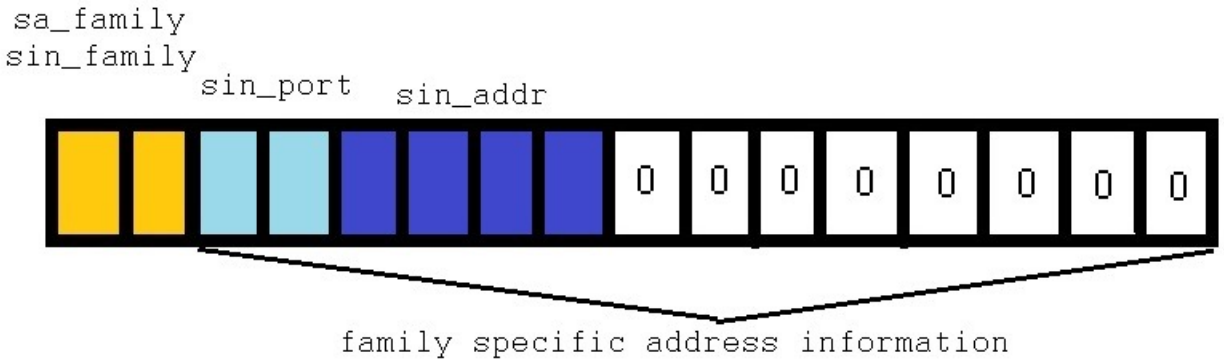
```
struct sockaddr {
    uint16_t  sa_family;    /* Protocol Family */
    char      sa_data[14];  /* Address data */
}
```



Socket API : Internet specific sockaddr

```
struct sockaddr_in {
    uint16_t  sin_family;          /* protocol family (AF_INET)  */
    uint16_t  sin_port;            /* port num (network byte order) */
    struct in_addr  sin_addr;      /* IP addr (network byte order) */
    unsigned char sin_zero[8];  /* Padding (to size of struct sockaddr) */
}
```

- Must cast to (struct sockaddr *) for use with Socket API calls



Socket API : Informational Calls

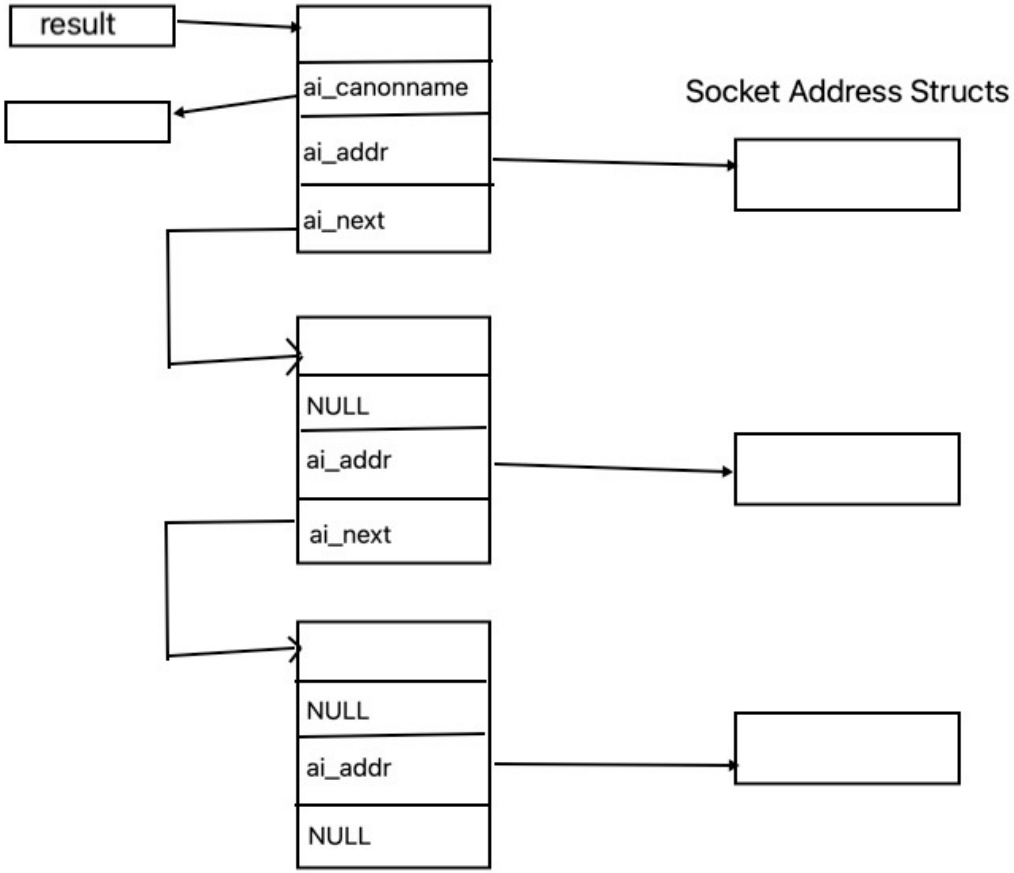
- **getaddrinfo()** - Return address information about a host and service
  - This call replaces older gethostbyname() and getservbyname()
  - Code is reentrant
  - Returns a linked list of addrinfo structures which match the request
  - *hints* used to narrow search (i.e. look only for IPv4 addresses)

```
int getaddrinfo(char *host, char *service, const struct addrinfo *hints, struct addrinfo **result)
```

Example:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
…
  struct addrinfo *ainfo:
  int res;

  res = getaddrinfo(“www3.cs.stonybrook.edu”, “http”, NULL, &ainfo);
…
```

getaddrinfo return struct



Socket API : Informational Calls

struct addrinfo

- contains returned information
- Space allocated by getaddrinfo()
- Free the space after done with **freeaddrinfo(struct addrinfo *ainfo);**

```
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    socklen_t ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;
    struct addrinfo *ai_next;
}
```

Socket API : struct sockaddr_in

Recall struct sockaddr_in

- sin_port – 16 bit port – network byte order
- sin_addr – 32 bit IPv4 address – network byte order

'Endianness' of host may or may not be the same as network byte order

Socket API provides utilities to convert these

- **htonl()** – Converts long (32 bits) from host to network byte order
- **ntohl()** – Convert long (32 bits) from network to host byte order
- **htons()** – Converts short (16 bits) from host to network byte order
- **ntohs()** – Conerts short (16 bits) from network to host byte order

Use htonl() on IP address before inserting it in sockaddr_in.sin_addr

Use htons() on Port before inserting it into sockaddr_in.sin_port

These only switch byte order if host and network byte order differ. Otherwise they are 'no-ops'

Socket API : Sockets and connections

- Network connections use ‘sockets’
- The *socket* is a small integer that relates to information about the connection (small integer like a file descriptor)
- Initially, socket is not related to a specific connection
- *int socket(int domain, int type, int protocol);*
  - Provide domain, connection type, and protocol
    - Domain – AF_INET, AF_UNIX, AF_LOCAL
    - Type – SOCK_STREAM, SOCK_DGRAM
    - Protocol – IPPROTO_TCP, IPPROTO_UDP
  - Returned integer is socket (-1 = error)

Socket API : Sockets and connections

- Address Families
  - AF_INET – Internet address
  - AF_UNIX – Unix file
- Socket types:
  - SOCK_STREAM
    - Usually associated with a TCP connection (Transmission Control Protocol)
    - Provides reliable transfer with in-order delivery and retransmission
  - SOCK_DGRAM
    - Associated with a socket that uses UDP. (User Datagram Protocol [NOT Unreliable Datagram Protocol])
    - Provides a *best-effort* delivery service
    - Note: This is a **connectionless** protocol

Socket API : Example: socket creation

```
#include <sys/socket.h>

int sock;

sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

If (sock == -1) {
    printf("socket failure!\n");
   exit(1);
}
```

Arguments describe the type of connection for which socket will be used.

Returns a small integer like a file descriptor. However, this is not ready to use as there is no connection (for TCP).

Do not forget to verify the returned socket is good!

Socket API : Creating connections from sockets

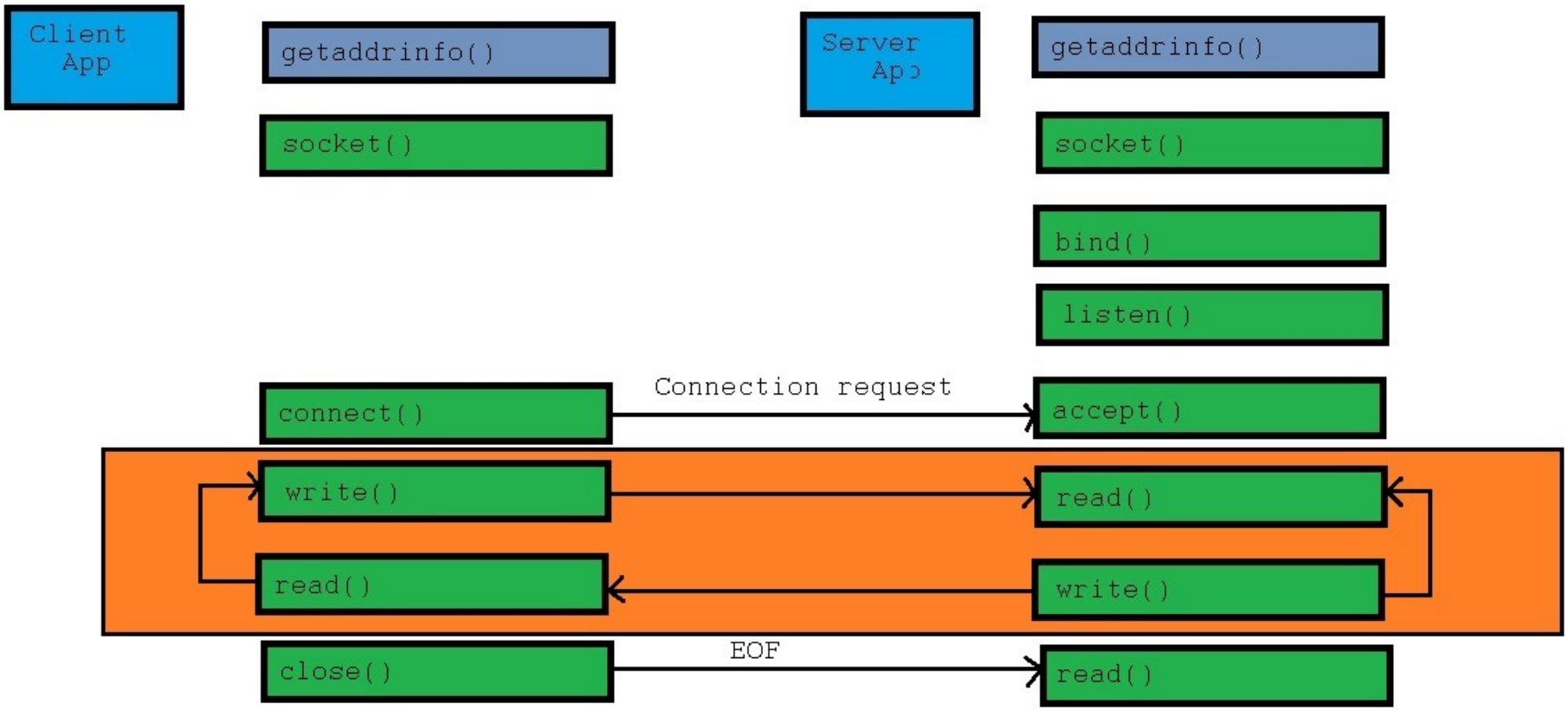
Active – This is the side that is initiating the connection

- Create socket (**socket**())
- Issue connect() request (**connect**())
- Send/receive data

Passive – This is the side that is waiting for a connection request

- Create socket (**socket**())
- Bind to an address/port (**bind**())
- Listen for a connection request [TCP only] (**listen**())
- When a request arrives, accept the connection [TCP only] (**accept**())
- Send/receive data

Programmer's Perspective



Socket API : Active connection setup

- **Active** refers to the side that is **requesting** the connection

int connect(int sock, const struct sockaddr *address, socklen_t length);

- Provide:
  - socket number
  - peer information (in sockaddr_in) – This is the host and service to which we want to connect
  - the length of the sockaddr_in struct
- Returns -1 on error (0 on success)
- On success, *sock* (the returned integer) can be used for communication
- Usually used only for STREAM connections (TCP).
- Can be used for UDP – No connection created but all communication occurs to/from host/port in connect()

Socket API : Example: Active connection

```
#include <sys/socket.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv) {
  int sock = 0;
  struct addrinfo *ainfo;
  struct sockaddr_in serverAddr;
  int result;
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (sock > 0) {
    result = 0;
    result = getaddrinfo("www3.cs.stonybrook.edu", "http", NULL, &ainfo);
    if (result == 0) {
      memset(&serverAddr, 0, sizeof(serverAddr));
      serverAddr.sin_family      = AF_INET;
      serverAddr.sin_addr.s_addr = (((struct sockaddr_in *)(ainfo->ai_addr))->sin_addr).s_addr; // don't need htonl()!
      serverAddr.sin_port        = ((struct sockaddr_in *)(ainfo->ai_addr))->sin_port;      // don't need htons()!
… // continued on next slide
```

Socket API : Example: Active connection (2)

```
result = connect(sock, (struct sockaddr *) &serverAddr, sizeof(serverAddr));
     if (result == 0) {
              // sock is ready for communication
              printf("Worked\n");
              close(sock);
     }
    }
  } else {
    printf("Socket call failed\n");
  }

}
```

Socket API : Passive connection

- **Passive** connection refers to the process that is listening and accepts an incoming active connection.
- UDP or TCP:

  int bind(int socket, const struct sockaddr *addr, socklen_t length);

  - Provide
    - socket number
    - sockaddr struct with IP address/port info (for **this** application!)
    - the length of the sockaddr structure
  - Returns -1 on error
  - This associates the socket with an address/port on the host

Socket API : Passive connection

TCP only:

int listen(int socket, int backlog);

- Provide
  - Socket
  - number of 'pending' requests to hold
- Returns -1 on error (or no connect waiting)
- Returns 0 if a remote peer is waiting for a connect response

Socket API : Passive connection

TCP only:

int accept(int socket, struct sockaddr *addr, socklen_t *length);

- Provide
  - the socket
  - a pointer to a sockaddr_in structure
  - pointer to a socklen_t (this will be length of sockaddr_in struct
- Returns -1 on error
- Returns connected socket number (> 0) on success. This also **fills in the peer address/port information** and the length of the returned sockaddr_in struct
- Note: returned socket number is different from the 'socket' parameter used to listen for connections!
  - Allows servers to connect to multiple clients at one time

Socket API : Example: Passive connection (1) Socket creation and bind

```
int desc;
int listreturn;
struct sockaddr_in servAddr;
int slen = sizeof (servAddr);
pid_t chldPid;
int sock;
int res;
const int max_line_len = 256;
ssize_t len;
int recv_line_len = max_line_len;
char recv_line[recv_line_len];

memset((void *) &servAddr, 0, sizeof(servAddr));
servAddr.sin_port = htons(25000);            // bind to port 25000
servAddr.sin_addr.s_addr = htonl(0x7F000001);  // on host 127.0.0.1
servAddr.sin_family = AF_INET;

// do socket(), bind(), and listen()
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

res = bind(sock, (struct sockaddr *)&servAddr, slen);
if (res < 0) { printf("bind failed\n"); return 1; }
```

Socket API : Example: Passive connection (2) Listen on port

```
if ((listreturn = listen(sock, 5)) < 0) {
    printf("Listen failed: %d\n", listreturn);
  }
```

Socket API : Example: Passive connection (3) Accepting connection request

```
while (1) {
   desc = accept(sock, (struct sockaddr *) &servAddr, &slen);
   if (desc < 0) {
     printf("accept failed\n");
     sleep(1);    // wait a bit and try again.
   } else {      // fork and let child continue
     if ((chldPid = fork()) != 0) {
        continue;  // Parent: go back to top of loop and accept another connection
                } else {
                  printf("Connected, creating child...\n");
                  send(desc, "Hi from server!\n", 16, 0);
                  close(desc);
     }
   }
 }
```

Socket API : Sending data

- **ssize_t send(int sock, const void *buf, size_t nbytes, int flags);**
  - Provide
    - socket number
    - Pointer to buffer with data
    - size of data
    - some flags
      - MSG_OOB – Out of band data. Delivered ahead of other queued data packets.
  - Returns number of characters sent or -1 for error

Socket API : Example: Send to TCP peer

```
#include <sys/socket.h>

 int count;
 … /* Perform socket(), connect() */

 count = send(sock, "procs sshd", 10, 0);
 if (count < 0) {
   printf (“Error sending!\n”);
 }
```

Socket API : Receiving data

ssize_t recv(int sock, void *buf, size_t nbytes, int flags);

- Provide
  - Socket
  - a pointer to a buffer
  - the size of the buffer
  - optional flags
- Returns number of characters read from socket or -1 on error

- NB: If implementing a request/response type protocol, after sending data, it is a good idea to pause for a short while before trying to retrieve a response [sleep(1)]

Socket API : Example: Receive from TCP peer

```
#include <sys/socket.h>

 const int max_line_len = 256;
 ssize_t len;
 int recv_line_len = max_line_len;
 char recv_line[recv_line_len];

 … /* Perform socket(), connect() */
 len = recv(sock, &recv_line[0], recv_line_len, 0);
 if (len < 0) {
   printf("Error on recv!\n");
 } else {
    printf ("Recv: %s\n", recv_line);
 }
```

Echo Server Example

```
//util.h
#ifndef __UTIL__
#define __UTIL__

#define ERRQUIT(msg)          (err_quit((msg), __FILE__, __func__, __LINE__))
#define CHKBOOLQUIT(exp, msg)((exp) || ERRQUIT(msg))
extern int err_quit(char *msg, const char *file, const char *func,
                    const int line);
extern int readn(int fd, char *ptr, int nbytes);
extern int writen(int fd, char *ptr, int nbytes);
extern int readline(int fd, char *ptr, int maxlen);
#endif
```

Echo Server Example (cont)

```
//util.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int err_quit(char *msg, const char *file, const char *func, const
int line) {
    printf("Error in function %s (%s, line %d): %s\n", func, file,
line, msg);
    exit(1);
    return 0;
}
```

Echo Server Example (cont)

```
int readn(int fd, char *ptr, int nbytes) {
    int nleft, nread;
    for(nleft = nbytes; nleft > 0; ) {
        nread = read(fd, ptr, nleft);
        if(nread < 0)       //error
            return nread;
        else if(nread == 0) //EOF
            break;
        nleft -= nread;
        ptr   += nread;
    }
    return nbytes - nleft;
}
```

Echo Server Example (cont)

```
int writen(int fd, char *ptr, int nbytes) {
    int nleft, nwritten;
    for(nleft = nbytes; nleft > 0; ) {
        nwritten = write(fd, ptr, nleft);
        if(nwritten < 0)        //error
            return nwritten;

        nleft -= nwritten;
        ptr   += nwritten;
    }
    return nbytes - nleft;
}
```

Echo Server Example (cont)

```
int readline(int fd, char *ptr, int maxlen) {
    int n, rc;
    char c;
    for(n = 1; n < maxlen; n++) { //read up to n-1 characters
        rc = read(fd, &c, 1);
        if(rc == 1) {
            *ptr++ = c;
            if(c == '\n')
                break;
        }
        else if(rc == 0) {
            if(n == 1)
                return 0;   //EOF, no data read
            else
                break;      //EOF, some data was read
        }
        else
            return -1;      //error
    }

    *ptr = 0; //add the null terminator
    return n;
}
```

Echo Server Example (cont)

```
//echo_server.c
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include "util.h"

#define SERV_PORT   6000
#define MAXLINE     512
```

Echo Server Example (cont)

```
void echo(int fd) {
    while(1) {
        char line[MAXLINE];
        int n = readline(fd, line, MAXLINE);
        CHKBOOLQUIT(n >= 0, "readline error");

        if(n == 0)  //connection terminated
            return;
        else
            CHKBOOLQUIT(writen(fd, line, n) == n, "writen error");
    }
}
```

Echo Server Example (cont)

```
int main(int argc, char **argv) {
    int sfd;
    struct sockaddr_in saddr;

    CHKBOOLQUIT( (sfd = socket(AF_INET, SOCK_STREAM, 0)) >= 0, "socket failed" );
    memset(&saddr, 0, sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = htonl(INADDR_ANY);
    saddr.sin_port = htons(SERV_PORT);
    CHKBOOLQUIT( bind(sfd, (struct sockaddr*)&saddr, sizeof(saddr)) >= 0, "bind failed");
    CHKBOOLQUIT( listen(sfd, 1024) >= 0, "listen failed" );
```

Echo Server Example (cont)

```
while(1) {
        struct sockaddr_in caddr;
        int cfd, clen = sizeof(caddr);
        CHKBOOLQUIT( (cfd = accept(sfd, (struct sockaddr*) &caddr, &clen)) >= 0,
                                                                "accept failed");
        if(fork() == 0) {   //child
            close(sfd);
            printf("pid: %d, client: %s:%d\n", getpid(),
                                        inet_ntoa(caddr.sin_addr), caddr.sin_port);
            echo(cfd);
            printf("pid: %d done\n", getpid());
            exit(0);
        }
        else
            close(cfd);
    }
    return 0;
}
```

Echo Server Example [Client]

```
//echo_client.c
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include "util.h"

#define SERV_ADDR   "127.0.0.1"
#define SERV_PORT   6000
#define MAXLINE     512
```

Echo Server Example [Client] (cont)

```
void copy(int sfd) {
    char sline[MAXLINE], rline[MAXLINE];
    while(fgets(sline, MAXLINE, stdin) != NULL) {
        int n = strlen(sline);
        CHKBOOLQUIT( writen(sfd, sline, n) == n, "writen error" );

        CHKBOOLQUIT( (n = readline(sfd, rline, MAXLINE)) >= 0,
"readline error" );
        rline[n] = 0;
        fputs(rline, stdout);
    }
    CHKBOOLQUIT( ferror(stdin) == 0, "cannot read file" );
}
```

Echo Server Example [Client] (cont)

```
int main(int argc, char **argv) {
    int sfd;
    struct sockaddr_in saddr;

    CHKBOOLQUIT( (sfd = socket(AF_INET, SOCK_STREAM, 0)) >= 0,
                 "socket failed" );

    memset(&saddr, 0, sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = inet_addr(SERV_ADDR);
    saddr.sin_port = htons(SERV_PORT);
```

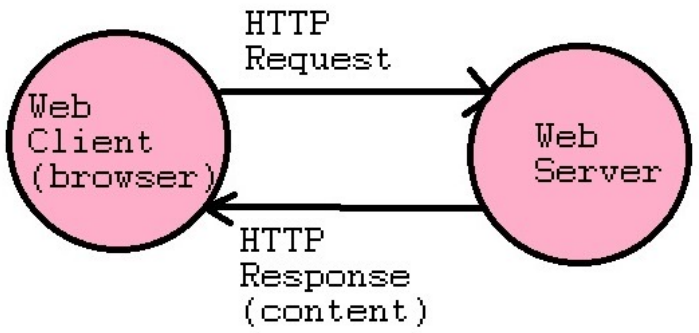
Echo Server Example [Client] (cont)

```
    CHKBOOLQUIT( connect(sfd, (struct sockaddr*)&saddr, sizeof(saddr)) >= 0,
                 "connectfailed" );
    printf("server: %s:%d\n", inet_ntoa(saddr.sin_addr), saddr.sin_port);

    copy(sfd);


    close(sfd);
    return 0;
}
```

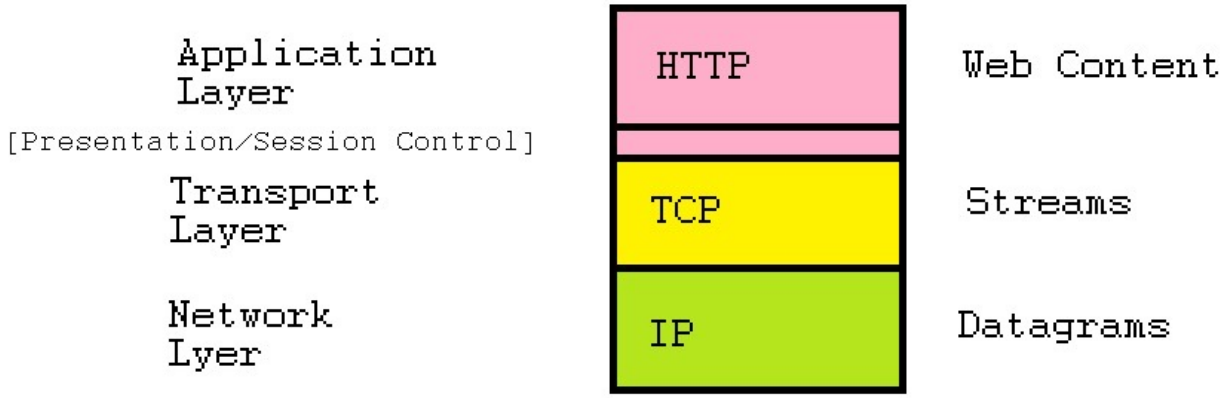
A look at Web Servers

Clients/Servers communicate using Hypertext Transfer Protocol (HTTP)

- Client and server establish connection
- Client sends a request
- Server sends a response with content
- Client and server close connection



A look at Web Servers



Recall 7-layer ISO model

- HTTP is an application protocol (but handles Presentation and Session control layers as well
- TCP handles Transport (client host to server host)
- IP handles Network issues (packetizing data from the stream into datagrams)
- Data Link and Physical layers are dependent on underlying technology (Ethernet, 802.11, etc.

Web Content

- Web servers return *content* to clients
  - *content:* a sequence of bytes with an associated MIME (Multipurpose Internet Mail Extensions) type

- Example MIME Types:
  - text/plain – unformatted text
  - text/html – HTML content
  - Image/jpeg – JPEG Image

List of MIME types:

`http://www.iana.org/assignments/media-types/media-types.xhtml`

Static and Dynamic Content

- The content returned in HTTP responses can be either *static* or *dynamic*
  - ***Static content***: content stored in files and retrieved in response to an HTTP request
    - Examples: HTML files, images, audio clips
    - Request identifies which content file
  - ***Dynamic content***: content produced on-the-fly in response to an HTTP request
    - Example: content produced by a program executed by the server on behalf of the client
    - Request identifies file containing executable code

URLs

- Unique name for a file across the entire web: URL (***Uniform Resource Locator*** – NOT **Universal Resource Locator** as the text indicates)
- Example URL: `http://www.google.com:80/index.html`
- Clients use *prefix* (`http://www.google.com:80`) to indicate:
  - What application protocol to use (HTTP)
  - Where the server is (`www.google.com`)
  - What port it is listening on (80)
- Servers use the *suffix* (`/index.html`) to:
  - Determine if request is for static or dynamic content.
    - There are numerous file extensions some of which map to static content (html, pdf, jpg, etc) and many that refer to dynamic content (.php, jsp, etc)
    - One convention: executables reside in `cgi-bin` directory
  - Find file on file system
    - Initial "`/`" in suffix denotes home directory for requested content. This root maps to a web server specific root given in its configuration files.
    - Minimal suffix is "`/`", which server expands to configured default filename (usually, `index.html`)

HTTP Requests

- HTTP request is a *request line*, followed by zero or more *request headers*
- Request line: `<method> <uri> <version>`
  - `<method>` is one of `GET`, `POST`, `OPTIONS`, `HEAD`, `PUT`, `DELETE`, or `TRACE`
  - `<uri>` is typically URL for proxies, URL suffix for servers
    - A URL is a type of URI (Uniform Resource Identifier)
      - ~~See https://www.rfc-editor.org/info/rfc2396~~
      - **See https://www.rfc-editor.org/info/rfc3986**
    - `<version>` is HTTP version of request (`HTTP/1.0` or **`HTTP/1.1`**)

HTTP Responses

- HTTP response is
  - a *response line*
  - zero or more *response headers*
  - [Possibly] *content*, with blank line ("`\r\n`") separating headers from content.

HTTP Responses

- Response headers: `<header name>: <header data>`
  - Provide additional information about response
  - `Content-Type:` MIME type of content in response body
  - `Content-Length:` Length of content in response body

HTTP Responses

- Response line:
- `<version> <status code> <status msg>`
  - <version> is HTTP version of the response
  - <status code> is numeric status
  - <status msg> is corresponding English text
    - 200 OK — Request was handled without error
    - 301 Moved — Provide alternate URL
    - 404 Not found — Server couldn't find the file
  - Status codes are 3 digits and have categories:
    - 1xx – Informational
    - 2xx – Success
    - 3xx – Redirect
    - 4xx – Client error
    - 5xx – Server error

Testing Servers

- Can test servers with *telnet.* Old terminal communication application
  - Transmit ascii strings over internet connections
  - Can specify host and port for remote server
- Usage: telnet <host> <port>
  - Creates a connection to whatever server is running on <port> on the target <host>
  - Examples:
    - Connect to port 80 to talk directly to a web server
    - Use port 25 to talk with mail exchange server

Questions?