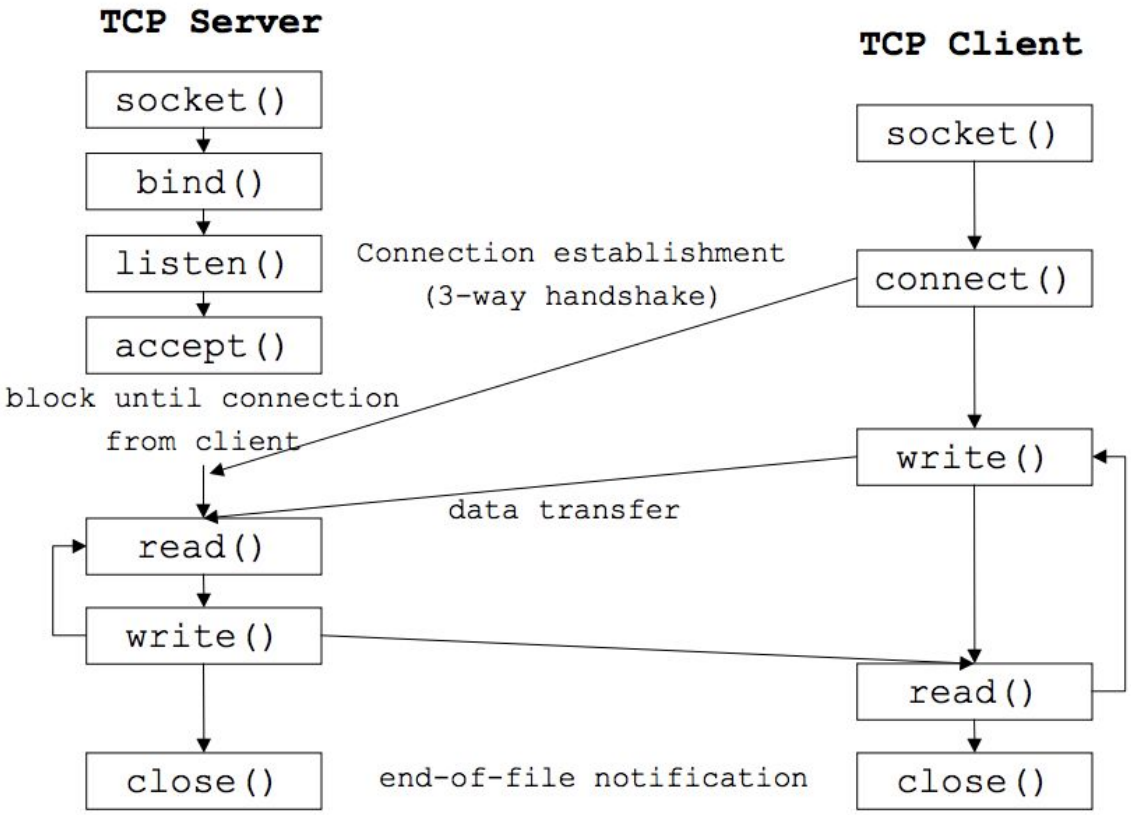
CS214 Recitation Sec.7

Dec. 12, 2017

Topics

1. Review: Sockets
2. Blocking & Non-blocking
3. Example: Group chat program

Review: TCP Server-Client Model



Creating a “Server Socket”

- **socket()**: *Creates a new socket* for a specific protocol (eg: TCP)
- **bind()**: Binds the socket to a *specific port* (eg: 80)
- **listen()**: Moves the socket into a state of *listening for incoming connections*.
- **accept()**: Accepts an incoming connection.

Creating a “Client Socket”

- **socket()**: *Creates a new socket* for a specific protocol (eg: TCP)
- **connect()**: Makes a network connection to *a specified IP address and port*.

socket()

```
int socket (int family, int type, int
   protocol);
```

- Create a socket.
  - Returns file descriptor or -1. Also sets `errno` on failure.
  - `family`: address family (namespace)
    - `AF_INET` for IPv4
    - other possibilities: `AF_INET6` (IPv6), `AF_UNIX` or `AF_LOCAL` (Unix socket), `AF_ROUTE` (routing)
  - `type`: style of communication
    - `SOCK_STREAM` for TCP (with `AF_INET`)
    - `SOCK_DGRAM` for UDP (with `AF_INET`)
  - `protocol`: protocol within family
    - typically 0

socket()

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */

tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
raw_socket = socket(AF_INET, SOCK_RAW, protocol);
```

bind()

```
int bind (int sockfd, struct sockaddr*
   myaddr, int addrlen);
```

- Bind a socket to a local IP address and port number
  - Returns 0 on success, -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `myaddr`: includes IP address and port number
    - IP address: set by kernel if value passed is `INADDR_ANY`, else set by caller
    - port number: set by kernel if value passed is 0, else set by caller
  - `addrlen`: length of address structure
    - `= sizeof (struct sockaddr_in)`

listen()

```
int listen (int sockfd, int backlog);
```

- Put socket into passive state (wait for connections rather than initiate a connection)
  - Returns 0 on success, -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `backlog`: bound on length of unaccepted connection queue (connection backlog); kernel will cap, thus better to set high
  - Example:

```
if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
}
```

Establishing a Connection

connect() for client side,
accept() for server side.

- Include file `<sys/socket.h>`

```
int connect (int sockfd, struct
   sockaddr* servaddr, int addrlen);
```

  - Connect to another socket.

```
int accept (int sockfd, struct sockaddr*
   cliaddr, int* addrlen);
```

  - Accept a new connection. Returns file descriptor or -1.

connect()

```
int connect (int sockfd, struct
   sockaddr* servaddr, int addrlen);
```

- Connect to another socket.
  - Returns 0 on success, -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `servaddr`: IP address and port number of server
  - `addrlen`: length of address structure
    - `= sizeof (struct sockaddr_in)`
- Can use with UDP to restrict incoming datagrams and to obtain asynchronous errors

accept()

```
int accept (int sockfd, struct sockaddr* cliaddr,
    int* addrlen);
```

- Block waiting for a new connection
  - Returns file descriptor or -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `cliaddr`: IP address and port number of client (returned from call)
  - `addrlen`: length of address structure = pointer to `int` set to `sizeof (struct sockaddr_in)`

- `addrlen` is a **value-result** argument
  - the caller passes the size of the address structure, the kernel returns the size of the client's address (the number of bytes written)

Sending and Receiving Data

```
int send(int sockfd, const void * buf,
   size_t nbytes, int flags);
```

- Write data to a stream (TCP) or “connected” datagram (UDP) socket.
  - Returns number of bytes written or -1.

```
int recv(int sockfd, void *buf, size_t
   nbytes, int flags);
```

- Read data from a stream (TCP) or “connected” datagram (UDP) socket.
  - Returns number of bytes read or -1.

send()

```
int send(int sockfd, const void * buf, size_t
   nbytes, int flags);
```

- Send data un a stream (TCP) or “connected” datagram (UDP) socket
  - Returns number of bytes written or -1 and sets `errno` on failure
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `buf`: data buffer
  - `nbytes`: number of bytes to try to write
  - `flags`: control flags
    - MSG_PEEK: get data from the beginning of the receive queue without removing that data from the queue

recv()

```
int recv(int sockfd, void *buf, size_t nbytes,
    int flags);
```

- Read data from a stream (TCP) or "connected" datagram (UDP) socket
  - Returns number of bytes read or -1, sets `errno` on failure
  - Returns 0 if socket closed
  - `sockfd`: socket file descriptor (returned from `socket`)
  - `buf`: data buffer
  - `nbytes`: number of bytes to try to read
  - `flags`: see man page for details; typically use 0

addrinfo

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

int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

```
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;
};
```

Building a simple TCP Client

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int s;
    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

    struct addrinfo hints, *result;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET; /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* TCP */

    s = getaddrinfo("www.cs.rutgers.edu", "80", &hints, &result);
    if (s != 0) {
            fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
            exit(1);
    }

    if(connect(sock_fd, result->ai_addr, result->ai_addrlen) == -1){
                perror("connect");
                exit(2);
        }

    char *buffer = "GET / HTTP/1.0\r\n\r\n";
    printf("SENDING: %s", buffer);
    printf("===\n");
```

```c
    write(sock_fd, buffer, strlen(buffer));


    char resp[1000];
    int len = read(sock_fd, resp, 999);
    resp[len] = '\0';
    printf("%s\n", resp);

    return 0;
}
```

This program writes the command “GET / HTTP/1.0\r\n\r\n” to “www.cs.rutgers.edu:80”. And then reads value back from this address.

Building a simple TCP Client

```
~/2017F/CS 214/recitation_11_28 » ./tcp_client
SENDING: GET / HTTP/1.0

===
HTTP/1.1 301 Moved Permanently
Date: Wed, 29 Nov 2017 17:13:04 GMT
Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.1e-fips mod_auth_gssapi/1.3.1 mod_auth
_kerb/5.4 mod_fcgid/2.3.9 mod_nss/2.4.6 NSS/3.19.1 Basic ECC PHP/5.4.16 SVN/1.7.
14 mod_wsgi/3.4 Python/2.7.5
Location: http://www.cs.rutgers.edu/
Content-Length: 234
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://www.cs.rutgers.edu/">here</a>.</p>
</body></html>
```

Building a simple TCP Server

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
    int s;
    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

    struct addrinfo hints, *result;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    s = getaddrinfo(NULL, "1234", &hints, &result);
    if (s != 0) {
            fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
            exit(1);
    }

    if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
        perror("bind()");
        exit(1);
    }

    if (listen(sock_fd, 10) != 0) {
        perror("listen()");
        exit(1);
    }

    struct sockaddr_in *result_addr = (struct sockaddr_in *) result->ai_addr;
    printf("Listening on file descriptor %d, port %d\n", sock_fd, ntohs(result_addr->sin_port));

    printf("Waiting for connection...\n");
    int client_fd = accept(sock_fd, NULL, NULL);
    printf("Connection made: client_fd=%d\n", client_fd);

    char buffer[1000];
    int len = read(client_fd, buffer, sizeof(buffer) - 1);
    buffer[len] = '\0';

    printf("Read %d chars\n", len);
    printf("===\n");
    printf("%s\n", buffer);

    return 0;
}
```

This program wait for ***connection from clients***. If client sends message to the server, the program will read that message and print it. If no message is sent, the server will wait for new message.

sockaddr & sockaddr_in

```
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};

struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};

struct in_addr {
    unsigned long s_addr;          // load with inet_pton()
};
```

sin_zero is padding blank bytes which are used for keep ***sockaddr*** and ***sockaddr_in*** the same sized. The pointer that points to ***sockaddr_in*** can also points to ***sockaddr***. Generally, we use ***sockaddr_in*** to substitute ***sockaddr*** in our program.

Notes on simple TCP Server

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
    int s;
    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

    struct addrinfo hints, *result;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    s = getaddrinfo(NULL, "1234", &hints, &result);
    if (s != 0) {
            fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
            exit(1);
    }

    if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
        perror("bind()");
        exit(1);
    }
```

```c
    if (listen(sock_fd, 10) != 0) {
        perror("listen()");
        exit(1);
    }

    struct sockaddr_in *result_addr = (struct sockaddr_in *) result->ai_addr;
    printf("Listening on file descriptor %d, port %d\n", sock_fd, ntohs(result_addr->sin_port));

    printf("Waiting for connection...\n");
    int client_fd = accept(sock_fd, NULL, NULL);
    printf("Connection made: client_fd=%d\n", client_fd);

    char buffer[1000];
    int len = read(client_fd, buffer, sizeof(buffer) - 1);
    buffer[len] = '\0';

    printf("Read %d chars\n", len);
    printf("===\n");
    printf("%s\n", buffer);

    return 0;
}
```

```c
struct sockaddr *ai_addr;
```

result -> ai_addr is a ***sockaddr** ** struct, here we force it to a ***sockaddr_in** ** struct

Building a simple TCP Server

```
~/2017F/CS 214/recitation_11_28 » ./tcp_server
Listening on file descriptor 3, port 1234
Waiting for connection...
```

Blocking & Non-blocking

- If using **blocking** mode, when a process executes `read()`, if the data is not available yet, the process is blocked and it will wait until the data is ready before the function returns.
- If using **non-blocking** mode, when a process executes `read()`, if the data is not available yet, the process will return immediately with a different value and continues executing.
- Non-blocking mode is more **efficient** than blocking mode. But may use more CPU resources.

Blocking & Non-blocking

- The default mode is **block** mode. If there are more than one sockets, when we work on one of these sockets, we cannot handle other sockets at the same time.
- Design a **concurrent** program to solve the above problem, and make sure multiple sockets can work together.

Non-blocking

- Three ways to set nonblocking
- 1. To set a file descriptor to be nonblocking

```
// fd is my file descriptor
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

- 2. For a socket, create it in nonblocking mode by adding SOCK_NONBLOCK to the second argument

```
fd = socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, 0);
```

Non-blocking

- The previous two methods continue looking up sockets, and *use many CPU resources*
- 3. Multiplexing with select(), it will wait for any of those file descriptors to become 'ready'.
- select() returns the **total number** of file descriptors that are ready. If none of them become ready during the time defined by timeout, it will return 0.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
        fd_set *exceptfds, struct timeval *timeout);
```

Non-blocking

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
        fd_set *exceptfds, struct timeval *timeout);
```

```
#ifndef FD_SETSIZE
#define FD_SETSIZE      64
#endif /* FD_SETSIZE */

typedef struct fd_set {
        u_int fd_count;        // how many are SET?
        SOCKET  fd_array[FD_SETSIZE]; // an array of SOCKETs
} fd_set;
```

FD_SET(int fd, fd_set *set); *add fd* to *set*

FD_CLR(int fd, fd_set *set); *remove fd* from *set*

FD_ISSET(int fd, fd_set *set); *If fd is in set*, return true

FD_ZERO(fd_set *set); Set the whole *set* to zero

```
struct timeval {
  int tv_sec; // second
  int tv_usec; // microseconds
};
```

Non-blocking

```
fd_set readfds, writefds;
FD_ZERO(&readfds);
FD_ZERO(&writefds);
for (int i=0; i < read_fd_count; i++)
  FD_SET(my_read_fds[i], &readfds);
for (int i=0; i < write_fd_count; i++)
  FD_SET(my_write_fds[i], &writefds);

struct timeval timeout;
timeout.tv_sec = 3;
timeout.tv_usec = 0;

int num_ready = select(FD_SETSIZE, &readfds, &writefds, NULL, &timeout);

if (num_ready < 0) {
  perror("error in select()");
} else if (num_ready == 0) {
  printf("timeout\n");
} else {
  for (int i=0; i < read_fd_count; i++)
    if (FD_ISSET(my_read_fds[i], &readfds))
      printf("fd %d is ready for reading\n", my_read_fds[i]);
  for (int i=0; i < write_fd_count; i++)
    if (FD_ISSET(my_write_fds[i], &writefds))
      printf("fd %d is ready for writing\n", my_write_fds[i]);
}
```

This code defines two file descriptor sets and sets up timeout to 3 seconds. Then the program calls select() to return the number of available data and print the data.

A chat program example

- client.c
- makefile
- readme.txt
- server.c

Run server first, then run several clients:

```
To compile source code:  make
For server side:  ./server servername
For client side:  ./client localhost clientname cs214
where cs214 is the passwd
```

Construct a chat program - Server (1)

```c
int main(int argc,char *argv[])
{
    struct sockaddr_in serverSockaddr,clientSockaddr;
    char sendBuf[MAX_DATA_SIZE],recvBuf[MAX_DATA_SIZE];
    int sendSize,recvSize;
    int sockfd,clientfd;
    fd_set servfd,recvfd;//use for select()
    int fd_A[BACKLOG+1];//save the socket file descriptor of clients
    char fd_C[BACKLOG+1][32];//save the username of clients
    int conn_amount;//count the number of clients
    int max_servfd,max_recvfd;
    int on=1;
    socklen_t sinSize=0;
    char username[32];
    char ch[64];
    int pid;
    int i,j;
    struct timeval timeout;
    struct user use;
    time_t now; struct tm *timenow;


    if(argc != 2)
    {
        printf("usage: ./server [username]\n");
        exit(1);
    }
    strcpy(username,argv[1]);
    printf("username:%s\n",username);

    /*establish a socket*/
    if((sockfd = socket(AF_INET,SOCK_STREAM,0))==-1)
    {
        perror("fail to establish a socket");
        exit(1);
    }
```

```c
strcpy(username,argv[1]);
printf("username:%s\n",username);

/*establish a socket*/
if((sockfd = socket(AF_INET,SOCK_STREAM,0))==-1)
{
    perror("fail to establish a socket");
    exit(1);
}
printf("Success to establish a socket...\n");

/*init sockaddr_in*/
serverSockaddr.sin_family=AF_INET;
serverSockaddr.sin_port=htons(SERVER_PORT);
serverSockaddr.sin_addr.s_addr=htonl(INADDR_ANY);
printf("%lu\n", sizeof(serverSockaddr));
bzero(&(serverSockaddr.sin_zero),8);

setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&on,sizeof(on));

/*bind socket*/
if(bind(sockfd,(struct sockaddr *)&serverSockaddr,sizeof(struct sockaddr))==-1)
{
    perror("fail to bind");
    exit(1);
}
printf("Success to bind the socket...\n");

/*listen on the socket*/
if(listen(sockfd,BACKLOG)==-1)
{
    perror("fail to listen");
    exit(1);
}

//time(&now);
timeout.tv_sec=2;//every 2 seconds
timeout.tv_usec=0;
sinSize=sizeof(clientSockaddr);//to get IP and port
```

Notes on chat program - Server (1)

include/netinet/in.h

```
/* Address to accept any incoming messages.  */
#define    INADDR_ANY        ((in_addr_t) 0x00000000)
```

0.0.0.0

Name

htonl, htons, ntohl, ntohs - convert values between host and network byte order

Synopsis

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

example: input 14, 32, 40, 00 *output* 00, 40, 32, 14

h --- host to n --- net l --- unsigned long

The **bzero()** function erases the data in the *n* bytes of the memory starting at the location pointed to by *s*, by writing zeroes (bytes containing '\0') to that area.

Construct a chat program - Server (2)

```
while(1)
{
    FD_ZERO(&servfd);//clear all fds of server
    FD_ZERO(&recvfd);//clear all fds of client
    FD_SET(sockfd,&servfd);
    //timeout.tv_sec=30;//reduce the check frequency
    switch(select(max_servfd+1,&servfd,NULL,NULL,&timeout))
    {
        case -1:
            perror("select error");
            break;
        case 0:
            break;
        default:
            //printf("has datas to offer accept\n");
            if(FD_ISSET(sockfd,&servfd))//sockfd if have data, means can be accepted
            {
                /*accept a client's request*/
                if((clientfd=accept(sockfd,(struct sockaddr  *)&clientSockaddr, &sinSize))==-1)
                {
                    perror("fail to accept");
                    exit(1);
                }
                printf("Success to accpet a connection request...\n");
                printf(">>>>>> %s:%d join in! ID(fd):%d \n",inet_ntoa(clientSockaddr.sin_addr),ntohs(cl
                    ientSockaddr.sin_port),clientfd);
```

The **inet_ntoa**() function converts the Internet host address *in*, given in network byte order, to a string in IPv4 dotted-decimal notation. The string is returned in a statically allocated buffer, which subsequent calls will overwrite.

```
$ ./a.out 0x7f.1                # First byte is in hex
127.0.0.1
```

Construct a chat program - Server (3)

```
        struct tm *info;
        time(&now);
        info = localtime(&now);
        printf("Join on:%s\n",asctime(info));

        if((recvSize=recv(clientfd,(char *)&use,sizeof(struct user),0))==-1 || recvSize==0)
        {
            perror("fail to receive datas");
        }
        printf("Username from client:%s,Passwd:%s\n",use.name,use.pwd);
        memset(recvBuf,0,sizeof(recvBuf));
        if(strcmp(use.pwd,"cs214")==0)
        {
            printf("Auth Success! \n");
            strcpy(sendBuf,"yes");
        }
        else
        {
            printf("Auth Failure! \n");
            strcpy(sendBuf,"no");
        }
        if((sendSize=send(clientfd,sendBuf,MAX_DATA_SIZE,0))==-1)
        {
            perror("fail to receive datas");
        }
        //Write fd_set when a new client joins
        fd_A[conn_amount]=clientfd;
        strcpy(fd_C[conn_amount],use.name);
        conn_amount++;
        max_recvfd=MAX(max_recvfd,clientfd);
    }
    break;
}
```

```
#define SERVER_PORT 12138
#define BACKLOG 20   // how many pending connections queue will hold
#define MAX_CON_NO 10
#define MAX_DATA_SIZE 4096

struct user
{
    char name[32];
    char pwd[32];
};

int MAX(int a,int b)
{
    if(a>b)
        return a;
    return b;
}
```

Construct a chat program - Server (4)

```
for(i=0;i<MAX_CON_NO;i++)
{
    if(fd_A[i]!=0)
    {
        FD_SET(fd_A[i],&recvfd);
    }
}

switch(select(max_recvfd+1,&recvfd,NULL,NULL,&timeout))
{
    case -1:
        //select error
        break;
    case 0:
        //timeout
        break;
    default:
        for(i=0;i<conn_amount;i++)
        {
            if(FD_ISSET(fd_A[i],&recvfd))
            {
                /*receive datas from client*/
                if((recvSize=recv(fd_A[i],recvBuf,MAX_DATA_SIZE,0))==-1 || recvSize==0)
                {
                    //perror("fail to receive datas");
                    //means the client is closed
                    printf("fd %d close\n",fd_A[i]);
                    FD_CLR(fd_A[i],&recvfd);
                    fd_A[i]=0;
                }
```

Construct a chat program - Server (5)

```
                else//forward data from one client to other clients
                {
                    /*send datas to client*/
                    strcpy(sendBuf,fd_C[i]);
                    strcat(sendBuf," ");
                    time(&now);
                    print_time(ch,&now);
                    //Add a time stamp
                    strcat(sendBuf,ch);
                    strcat(sendBuf,"\t\t");
                    strcat(sendBuf,recvBuf);
                    printf("Data is:%s\n",sendBuf);
                    for(j=0;j<MAX_CON_NO;j++)
                    {
                        if(fd_A[j]!=0&&i!=j)
                        {
                            printf("Data send to %d,",fd_A[j]);
                            if((sendSize=send(fd_A[j],sendBuf,strlen(sendBuf),0))!=strlen(sendBuf))
                            {
                                perror("fail");
                                exit(1);
                            }
                            else
                            {
                                printf("Success\n");
                            }
                        }
                    }
                    memset(recvBuf,0,MAX_DATA_SIZE);
                }
            }
        }
        break;
}
```

```
void print_time(char * ch,time_t *now)
{
    struct tm*stm;
    stm=localtime(now);//get the time right now
    sprintf(ch,"%02d:%02d:%02d\n",stm->tm_hour,stm->tm_min,stm->tm_sec);
}
```

Construct a chat program - Client (1)

```c
int main(int argc,char *argv[])
{
    int sockfd;//socket
    char sendBuf[MAX_BUF],recvBuf[MAX_BUF];
    int sendSize,recvSize;//get the time right now
    struct hostent * host;
    struct sockaddr_in servAddr;
    char username[32];
    char password[32];
    int pid;
    struct user use;

    if(argc != 4)
    {
        perror("use: ./client [hostname] [username] [password]");
        exit(-1);
    }
    strcpy(use.name,argv[2]);
    strcpy(use.pwd,argv[3]);
    printf("username:%s\n",use.name);
    printf("password:%s\n",use.pwd);
    host=gethostbyname(argv[1]);
    if(host==NULL)
    {
    perror("fail to get host by name.");
        exit(-1);
    }
    printf("Success to get host by name ...\n");

    //construct a socket
    if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
    {
    perror("fail to establish a socket");
    exit(1);
    }
    printf("Success to establish a socket...\n");

    /*init sockaddr_in*/
    servAddr.sin_family=AF_INET;
    servAddr.sin_port=htons(SERVER_PORT);
    servAddr.sin_addr=*((struct in_addr *)host->h_addr);
    //servAddr.sin_addr.s_addr=inet_addr("127.0.0.1");
    bzero(&(servAddr.sin_zero),8);

    /*connect the socket*/
    if(connect(sockfd,(struct sockaddr *)&servAddr,sizeof(struct sockaddr_in))==-1)
    {
    perror("fail to connect the socket");
    exit(1);
    }
    printf("Success to connect the socket...\n");

    //send username and passwd to server
    if(send(sockfd,(char *)&use,sizeof(struct user),0)==-1)
    {
        perror("fail to send datas.");
        exit(-1);
    }
    if((recvSize=recv(sockfd,recvBuf,MAX_BUF,0)==-1))
    {
        perror("fail to receive datas.");
        exit(-1);
    }
    //printf("Server:%s\n",recvBuf);
    if(strcmp(recvBuf,"no")==0)
    {
        perror("wrong passwd");
        exit(-1);
    }
```

Notes on chat program - Client (1)

gethostname() — Who am I?

Even easier than **getpeername()** is the function **gethostname()**. It returns the name of the computer that your program is running on. The name can then be used by **gethostbyname()**, below, to determine the IP address of your local machine.

What could be more fun? I could think of a few things, but they don't pertain to socket programming. Anyway, here's the breakdown:

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

The arguments are simple: *hostname* is a pointer to an array of chars that will contain the hostname upon the function's return, and *size* is the length in bytes of the *hostname* array.

The function returns 0 on successful completion, and -1 on error, setting *errno* as usual.

Construct a chat program - Client (2)

```
//send-recv
if((pid=fork())<0)
{
    perror("fork error\n");
}
else if(pid==0)/*child*/
{
    while(1)
    {
        fgets(sendBuf,MAX_BUF,stdin);
        printf("Me:%s\n", sendBuf);
        if(send(sockfd,sendBuf,strlen(sendBuf),0)==-1)
        {
            perror("fail to receive datas.");
        }
        memset(sendBuf,0,sizeof(sendBuf));
    }
}
else
{
    while(1)
    {
        if((recvSize=recv(sockfd,recvBuf,MAX_BUF,0)==-1))
        {
            printf("Server maybe shutdown!");
            break;
        }
        printf("%s\n",recvBuf);
        memset(recvBuf,0,sizeof(recvBuf));
    }
    kill(pid,SIGKILL);
}

close(sockfd);

return 0;
}
```

Running result of group chat

Server Side

```
~/2017F/CS 214/recitation_12_05/Chat program » ./server serv
username:serv
Success to establish a socket...
Success to bind the socket...
```

Server Side

```
~/2017F/CS 214/recitation_12_05/Chat program » ./server serv
username:serv
Success to establish a socket...
Success to bind the socket...
Success to accpet a connection request...
>>>>>> 127.0.0.1:62235 join in! ID(fd):4
Join on:Tue Dec 12 17:31:13 2017

Username from client:cli1,Passwd:cs214
Auth Success!
```

Client 1 Side

```
~/2017F/CS 214/recitation_12_05/Chat program » ./client localhost cli1 cs214
username:cli1
password:cs214
Success to get host by name ...
Success to establish a socket...
Success to connect the socket...
```

Running result of group chat (Cont.)

Server Side

```
~/2017F/CS 214/recitation_12_05/Chat program » ./server serv
username:serv
Success to establish a socket...
Success to bind the socket...
Success to accpet a connection request...
>>>>>> 127.0.0.1:62235 join in! ID(fd):4
Join on:Tue Dec 12 17:31:13 2017

Username from client:cli1,Passwd:cs214
Auth Success!
Success to accpet a connection request...
>>>>>> 127.0.0.1:62300 join in! ID(fd):5
Join on:Tue Dec 12 17:33:55 2017

Username from client:cli2,Passwd:cs214
Auth Success!
```

Client 2 Side

```
~/2017F/CS 214/recitation_12_05/Chat program » ./client localhost cli2 cs214
username:cli2
password:cs214
Success to get host by name ...
Success to establish a socket...
Success to connect the socket...
```

Running result of group chat (Cont.)

Client 2 Side

```
username:cli2
password:cs214
Success to get host by name ...
Success to establish a socket...
Success to connect the socket...
hello!
Me:hello!
```

Client 1 Side

```
username:cli1
password:cs214
Success to get host by name ...
Success to establish a socket...
Success to connect the socket...
cli2 17:38:52
                    hello!
```

Server Side

```
Data is:cli2 17:38:52
                    hello!

Data send to 4,Success
```

Running result of group chat (Cont.)

Client 3 Side

```
~/2017F/CS_214/recitation_12_05/Chat program » ./client localhost cli3 cs214
username:cli3
password:cs214
Success to get host by name ...
Success to establish a socket...
Success to connect the socket...
HELLO EVERYONE! WELCOME TO CS214 CLASS!
Me:HELLO EVERYONE! WELCOME TO CS214 CLASS!
```

Client 2 Side

```
hello!
Me:hello!

cli3 17:45:19
                    HELLO EVERYONE! WELCOME TO CS214 CLASS!
```

Server Side

```
>>>>>> 127.0.0.1:62996 join in! ID(fd):6
Join on:Tue Dec 12 17:44:41 2017

Username from client:cli3,Passwd:cs214
Auth Success!
Data is:cli3 17:45:19
                    HELLO EVERYONE! WELCOME TO CS214 CLASS!

Data send to 4,Success
Data send to 5,Success
```

Client 1 Side

```
cli2 17:38:52
                    hello!

cli3 17:45:19
                    HELLO EVERYONE! WELCOME TO CS214 CLASS!
```