# 21 Lecture: Introduction to Networks

**Outline:**

- Announcements
- Asgn 5 and Socket Programming
- Basics
- How it works
  - Sockets
- UDP
  - A note on addresses
  - The functions
  - Example: Passing messages with UDP
- TCP
  - Details
  - Example: Passing messages with TCP
- A real chat system
  - Choosing between I/O streams: select(2) and poll(2)
  - The Client
  - The Server
- Add content to this and to asgn5 writeup
- handin directories
- libraries to CSL

**Outline:**

- Announcements
- Asgn 5 and Socket Programming
- Basics
- How it works
  - Sockets
- UDP
  - A note on addresses
  - The functions
  - Example: Passing messages with UDP
- TCP
  - Details
  - Example: Passing messages with TCP
- A real chat system
  - Choosing between I/O streams: select(2) and poll(2)
  - The Client
  - The Server

## 21.1 Announcements

- Coming attractions:

| **Event** | **Subject** | **Due Date** | | | **Notes** |
|---|---|---|---|---|---|
| asgn4 | mytar | Mon | Nov 27 | 23:59 | |
| asgn5 | mytalk | Fri | Dec 1 | 23:59 | |
| lab07 | forkit | Mon | Dec 4 | 23:59 | |
| asgn6 | shell | Fri | Dec 8 | 23:59 | |

Use your own discretion with respect to timing/due dates.

- Be working on Asgn4 (now's the chance to make it good.)
- `/bin/pwd` under linux calls `getcwd(3)` which reads the `/proc` filesystem

## 21.2 Asgn 5 and Socket Programming

This is APUE Chapter 16. (This is not cpe464.)

So we have a network, how does one connect to it?

## 21.3 Basics

- A network connects two computers
- Communications are governed by protocols
- To talk to another machine you need to be able to name it (address)
- To talk to a particular program on another machine you need to be able to name it (port)
- These functions look a little hairy, but they're not as bad as they look. Keep your man pages handy for structures.
- We will be using IPV4
- The unix$N$ servers can talk to each other, but not off campus
- This just scratches the surface (take 464)

## 21.4 How it works

### 21.4.1 Sockets

- Invented by BSD as a network endpoint
- Created with `socket(2)`

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

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

You will want `AF_INET` and either

| | |
|---|---|
| `SOCK_STREAM` | TCP (Transmssion Control Protocol), stream oriented |
| `SOCK_DGRAM` | UDP (User Datagram Protocol), datagram oriented |

`protocol` is a sub-protocol, and in this case you'll want it to be 0.
Return value is a file descriptor.

## 21.5 UDP

| Client | Server |
|---|---|
| `socket(2)` | `socket(2)` |
|  | `bind(2)` |
| `recv_from(2)` | `send_to(2)` |
| `send_to(2)` | `recv_from(2)` |
| `close(2)` | `close(2)` |

- UDP is **connectionless**
- UDP is **unreliable**. You'll never know if it got there at all.
- Create a socket, and send a message.
- An un-bound socket is issued a binding on the way out so replies are possible.

### 21.5.1 A note on addresses

To talk to something you have to be able to name it.

- Both of these protocols use an **IP address** and a **port** to identify an endpoint.
- Many of these functions take a `struct sockaddr *` as a parameter.
- `struct sockaddr` is really a family. The Internet version is:

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```

- man `ip(7)` for IPv4 structures
- What if you don't know the address: `gethostbyname(3)` or `getaddrinfo(3)`
- What if you have an address and want to know who it belongs to? `gethostbyaddr(3)` or `getnameinfo(3)` .
- What if you don't care? `INADDR_ANY`

### Address Translation

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

void freeaddrinfo(struct addrinfo *res);

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

```
uint32_t getaddress(const char *hostname) {
  /* return the IPv4 address for the given host, or 0
   * Address is in network order   */
  struct addrinfo *ai,hints;
  uint32_t res=0;
  int rvalue;

  memset(&hints,0,sizeof(hints));
  hints.ai_family = AF_INET;
  if ( 0 == (rvalue=getaddrinfo(hostname,NULL,&hints,&ai))) {
    if ( ai )
      res = ((struct sockaddr_in*)ai->ai_addr)->sin_addr.s_addr;
    freeaddrinfo(ai);
  } else {
    fprintf(stderr,"%s:%s\n", hostname, gai_strerror(rvalue));
  }
  return res;
}
```

### 21.5.2 The functions

### socket(2): create a socket

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

You will want `AF_INET` and either
`SOCK_STREAM` TCP (Transmission Control Protocol), stream oriented
`SOCK_DGRAM` UDP (User Datagram Protocol), datagram oriented
`protocol` is a sub-protocol, and in this case you'll want it to be 0.
Return value is a file descriptor, or -1 on error.

**bind(2): Attach an address to a socket**

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

**getsockname(2)**

```
int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

getsockname()  returns  the  current  address  to which the socket
sockfd is bound
```

**inet_ntop(3)**

```
const char *inet_ntop(int af, const void *src,char *dst, socklen_t size);

This function converts the network address structure src in the af
address  family  into a character string.  The resulting string is
copied to the buffer pointed to by dst, which must be  a  non-NULL
pointer.   The  caller  specifies the number of bytes available in
this buffer in the argument size.
```

(`inet_pton(3)` does it backwards)

**recvfrom(2)**

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Returns bytes received.
`INADDR_ANY` is good to know about.

**sendto(2)**

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

Returns bytes received.
`send(2)`, `recv(2)`, `sendto(2)` and `recvfrom(2)` are just like `read(2)`, and `write(2)` except for the `flags` field. (e.g.`MSG_DONTWAIT`)

### 21.5.3 Example: Passing messages with UDP

Figures 89 and 90 show the two halves of a UDP conversation.

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

#define MESG "Hello, Client\n"
#define MAXLEN 1000

int main(int argc, char *argv[]) {
  int sockfd;
  struct sockaddr_in sa;
  socklen_t len;
  char localaddr[INET_ADDRSTRLEN];
  char buff[MAXLEN+1];
  int mlen,port = 10000;
  socklen_t slen;

  /* Create the socket */
  sockfd = socket(AF_INET, SOCK_DGRAM, 0);

  sa.sin_family      = AF_INET;
  sa.sin_port        = htons(port);     /* use our port  */
  sa.sin_addr.s_addr = htonl(INADDR_ANY);   /* all local interfaces */

  /* bind */
  bind(sockfd, (struct sockaddr *)&sa, sizeof(sa));

  /* receive */
  slen = sizeof(sa);
  mlen = recvfrom(sockfd, buff, sizeof(buff),0,(struct sockaddr *)&sa,&slen);

  /* who is this guy? */
  inet_ntop(AF_INET, &sa.sin_addr.s_addr,localaddr, sizeof(localaddr));
  printf("Message from <%s:%d>:  ",localaddr,ntohs(sa.sin_port));
  fflush(stdout);

  write(STDOUT_FILENO,buff,mlen);

  mlen = strlen(MESG);
  len=sendto(sockfd, MESG, mlen, 0, (struct sockaddr *)&sa, sizeof(sa));

  close(sockfd);  /* all done, clean up */
  return 0;
}
```

Figure 89: UDP Server

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

#define MESG "Hello, Server\n"
#define MAXLEN 1000
int main(int argc, char *argv[]) {
  int sockfd;                /* socket descriptors */
  struct sockaddr_in sa;
  struct hostent  *hostent;
  char buff[MAXLEN+1];
  int len,mlen;
  socklen_t slen;

  int port = 10000;
  char *hostname = "localhost";

  /* figure out who we're talking to */
  hostent = gethostbyname(hostname);

  /* Create the socket */
  sockfd = socket(AF_INET, SOCK_DGRAM, 0);

  /* connect it (the AP address is already network ordered) */
  sa.sin_family      = AF_INET;
  sa.sin_port        = htons(port); /* use our port  */
  sa.sin_addr.s_addr = *(uint32_t*)hostent->h_addr_list[0];

  mlen = strlen(MESG);
  len=sendto(sockfd, MESG, mlen, 0, (struct sockaddr *)&sa, sizeof(sa));

  sa.sin_family      = AF_INET;
  sa.sin_port        = htons(port);     /* use our port  */
  sa.sin_addr.s_addr = htonl(INADDR_ANY);   /* all local interfaces */

  slen = sizeof(sa);
  len = recvfrom(sockfd, buff, sizeof(buff),0,(struct sockaddr *)&sa,&slen);
  write(STDOUT_FILENO,buff,len);

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

Figure 90: UDP Client

## 21.6 TCP

TCP provides a reliable transport mechanism between two processes

- it is connection-oriented. That is, a connection has to be established before it can be used to communicate.
- It is reliable
- It **does not** preserve boundaries between messages.
- It's almost exactly like a pipe, except typically one uses `send(2)` and `recv(2)` rather than `read(2)` and `write(2)`

| **Client** | **Server** |
|---|---|
| `socket(2)` | `socket(2)` |
| | `bind(2)` |
| | `listen(2)` |
| `connect(2)` | `accept(2)` |
| | `getsockname(2)` |
| `send(2)` | `recv(2)` |
| `recv(2)` | `send(2)` |
| `close(2)` | `close(2)` |

### 21.6.1 Details

Also uses:

- `socket(2)` to create a socket
- `bind(2)` to attach an address to a socket

and may use `getsockname(2)`, `getpeerinfo(2)`, and `inet_ntop(2)` to find and report info about the other end.

**listen(2): wait for someone to `connect()`**

This is the server-side call. Wait for something to happen.

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

backlog is how many connections to allow to be pending before "Connection Refused"

**connect(2): try to talk to a `listen(2)`ing socket**

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

**accept(2): complete the connection**

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

Important: this returns a new socket. The addr is info on our peer. The original listening socket is unaffected.

**getsockname(2)**

```
int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

getsockname()  returns  the  current  address  to which the socket
sockfd is bound
```

**send(2)**

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

**recv(2)**

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

### 21.6.2 Example: Passing messages with TCP

```
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#define DEFAULT_BACKLOG 100
#define MESG "Hello, TCP Client.\n"
#define MAXLEN 1000

int main(int argc, char *argv[]) {
  int mlen,sockfd, newsock,port = 10000;
  struct sockaddr_in sa, newsockinfo, peerinfo;
  socklen_t len;
  char localaddr[INET_ADDRSTRLEN], peeraddr[INET_ADDRSTRLEN], buff[MAXLEN+1];

  sockfd = socket(AF_INET, SOCK_STREAM, 0);  /* Create the socket */

  sa.sin_family         = AF_INET;
  sa.sin_port           = htons(port);        /* use our port   */
  sa.sin_addr.s_addr    = htonl(INADDR_ANY); /* all local interfaces */
  bind(sockfd, (struct sockaddr *)&sa, sizeof(sa));

  listen(sockfd,DEFAULT_BACKLOG);  /* listen */

  len = sizeof(newsockinfo);    /* accept */
  newsock = accept(sockfd, (struct sockaddr *)&peerinfo, &len);

  /* now we're in business, I suppose */
  len = sizeof(newsockinfo);
  getsockname(newsock, (struct sockaddr *) &newsockinfo, &len);

  /* find out about the new socket and the other end */
  inet_ntop(AF_INET, &newsockinfo.sin_addr.s_addr,localaddr, sizeof(localaddr));
  inet_ntop(AF_INET, &peerinfo.sin_addr.s_addr,peeraddr, sizeof(peeraddr));

  printf("New Connection:  %s:%d->%s:%d\n",peeraddr,ntohs(peerinfo.sin_port),
                                   localaddr,ntohs(newsockinfo.sin_port));

  /* pass some data receive, then send*/
  mlen = recv(newsock, buff, sizeof(buff), 0);
  write(STDOUT_FILENO,buff,mlen);
  mlen = send(newsock, MESG, strlen(MESG), 0);

  close(sockfd);                /* all done, clean up */
  close(newsock);
  return 0;
}
```

Figure 91: TCP Server

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

#define MESG "Hello, TCP Server.\n"
#define MAXLEN 1000

int main(int argc, char *argv[]) {
  int port,len;
  int sockfd;                   /* socket descriptors */
  struct sockaddr_in sa;
  struct hostent  *hostent;
  const char *hostname;
  char buff[MAXLEN+1];

  port = 10000;
  hostname = "localhost";

  /* figure out who we're talking to */
  hostent = gethostbyname(hostname);

  /* Create the socket */
  sockfd = socket(AF_INET, SOCK_STREAM, 0);

  /* connect it */
  sa.sin_family        = AF_INET;
  sa.sin_port          = htons(port);               /* use our port  */
  sa.sin_addr.s_addr   = *(uint32_t*)hostent->h_addr_list[0]; /* net order */

  connect(sockfd, (struct sockaddr *)&sa, sizeof(sa));

  len = send(sockfd, MESG, strlen(MESG), 0);

  len = recv(sockfd, buff, sizeof(buff), 0);
  write(STDOUT_FILENO,buff,len);

  close(sockfd);                /* clean up and go home */

  return 0;
}
```

Figure 92: TCP Client

## 21.7 A real chat system

These programs represent a bare-bones TCP-based chat system. These have no error-checking, which is terrifying, but they'll fit on a page this way.

### 21.7.1 Choosing between I/O streams: select(2) and poll(2)

What if you have multiple potential sources of input or output and don't know which one to service next. You have two options:

- Polling using nonblocking IO
- `Select(2)` or `poll(2)`
-

`Select(2)` or `poll(2)` is a better approach.

`select(2)`

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

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

`poll(2)`

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

    struct pollfd {
        int   fd;         /* file descriptor */
        short events;     /* requested events */
        short revents;    /* returned events */
    };

POLLIN There is data to read.
```

### 21.7.2 The Client

It's easier to talk about the client first, so we will. It establishes a connection with the waiting server and then relays messages from the console to the server or the server to the console until the message is "bye".

Two parts:

- `main()` (figure 93) sets up the connection
- `chat()` (figure 94) handles the actual conversation.

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

#define MAX_CONNECTIONS 100
#define MAXLINE 1024
void chatclient(int sockfd);

int main(int argc, char *argv[]) {
  int sockfd;                   /* socket descriptors */
  struct sockaddr_in sa;
  struct hostent  *hostent;
  int port;
  char *hostname = "localhost";

  if ( argc==2 )
    port = atoi(argv[1]);
  else
    port = 10000;

  /* figure out who we're talking to */
  hostent = gethostbyname(hostname);

  /* Create the socket */
  sockfd = socket(AF_INET, SOCK_STREAM, 0);

  /* connect it */
  sa.sin_family = AF_INET;
  sa.sin_port    = htons(port);                 /* use our port  */
  sa.sin_addr.s_addr    = *(uint32_t*)hostent->h_addr_list[0]; /* net order */

  connect(sockfd, (struct sockaddr *)&sa, sizeof(sa));

  chatclient(sockfd);

  /* send some data */
  close(sockfd);
  return 0;
}
```

Figure 93: MiniChat client, `main()`

```
void chatclient(int sockfd) {
  int done, len, mlen;
  fd_set readfds;
  char buff[MAXLINE];
  socklen_t slen;

  done = 0;
  do {
    FD_ZERO(&readfds);
    FD_SET(STDIN_FILENO,&readfds);
    FD_SET(sockfd,&readfds);
    select(sockfd+1,&readfds,NULL,NULL,NULL);

    if (FD_ISSET(STDIN_FILENO,&readfds)) {
      /* read a line and send it to remote */
      len = read(STDIN_FILENO,buff,MAXLINE);
      mlen=send(sockfd, buff, len, 0);
    }
    if ( FD_ISSET(sockfd,&readfds)) {
      /* receive a line and print it */
      slen = sizeof(struct sockaddr_in);
      mlen = recv(sockfd, buff, sizeof(buff),0);

      write(STDOUT_FILENO,buff,mlen);
    }
    if ( !strncmp(buff,"bye",3) )
      done = 1;
  } while (!done);
}
```

Figure 94: MiniChat client, `chat()`

```
#include "minichat.h"
#include <poll.h>

#define LOCAL  0
#define REMOTE (LOCAL + 1)

void chatclient(int sockfd) {
  int done, len, mlen;
  char buff[MAXLINE];
  struct pollfd fds[REMOTE+1];

  fds[LOCAL].fd = STDIN_FILENO;
  fds[LOCAL].events = POLLIN;
  fds[LOCAL].revents = 0;
  fds[REMOTE]=fds[LOCAL];
  fds[REMOTE].fd = sockfd;

  done = 0;
  do {

    poll(fds,sizeof(fds)/sizeof(struct pollfd),-1); /* negative->wait forever */
    if (fds[LOCAL].revents & POLLIN ) {
      /* read a line and send it to remote */
      len = read(STDIN_FILENO,buff,MAXLINE);
      mlen=send(sockfd, buff, len, 0);
    }
    if (fds[REMOTE].revents & POLLIN ) {
      /* receive a line and print it */
      mlen = recv(sockfd, buff, sizeof(buff),0);

      write(STDOUT_FILENO,buff,mlen);
    }
    if ( !strncmp(buff,"bye",3) )
      done = 1;
  } while (!done);
}
```

Figure 95: MiniChat client, `chat()`, `poll(2)` version

```
#include "minichat.h"
#include <poll.h>

#define LISTENER 0

void chatserver(int listener) {
    int i, j, done, mlen, newsock, num = 0;
    struct sockaddr_in peerinfo;
    socklen_t slen;
    char buff[MAXLINE], addr[INET_ADDRSTRLEN];
    struct pollfd fds[MAX_CONNECTIONS + 1];
    done = 0;

    buff[0] = '\0';  /* Make sure this is a string for strncmp below */
    fds[LISTENER].fd = listener;
    fds[LISTENER].events = POLLIN;
    fds[LISTENER].revents = 0;
    num = 1;
    do {
        poll(fds, num, -1);
        /* check for connections */
        if ((fds[LISTENER].revents & POLLIN) && (num <= MAX_CONNECTIONS)) {
            /* accept a new connection and add the client to the list */
            slen = sizeof(peerinfo);
            newsock = accept(listener, (struct sockaddr *)&peerinfo, &slen);
            inet_ntop(AF_INET, &peerinfo.sin_addr.s_addr, addr, sizeof(addr));
            printf("New connection from:  %s:%d\n", addr, htons(peerinfo.sin_port));
            fds[num].fd = newsock;
            fds[num].events = POLLIN;
            fds[num].revents = 0;
            num++;
        }
        /* check for data */
        for (i = 1; i < num; i++) {
            if (fds[i].revents & POLLIN) {
                /* read from this client and broadcast to all */
                slen = sizeof(struct sockaddr_in);
                mlen = recv(fds[i].fd, buff, sizeof(buff), 0);
                for (j = 1; j < num; j++) /* broadcast to everyone else*/
                    if (i != j)          /* not self */
                        send(fds[j].fd, buff, mlen, 0);
            }
        }
        if (!strncmp(buff, "bye", 3))
            done = 1;
    } while (!done);
}
```

Figure 96: MiniChat client, `chat()`, `poll(2)` version

### 21.7.3 The Server

The server is a little more complicated.

- `main()` (figure 97) sets up the connection a listening socket then waits.
- `chat()` (figure 98)
  - accepts connections on the listener
  - accepts input from any client and forwards it on to all other clients.

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

#define MAX_CONNECTIONS 100
#define DEFAULT_BACKLOG MAX_CONNECTIONS
#define MAX_ADDR 100
#define MAXLINE 1024
void chatserver(int sockfd);

int main(int argc, char *argv[]) {
  int sockfd;            /* socket descriptors */
  struct sockaddr_in sa;
  int port;

  if ( argc==2 )
    port = atoi(argv[1]);
  else
    port = 10000;

  /* Create the socket */
  sockfd = socket(AF_INET, SOCK_STREAM, 0);

  /* bind it to our address*/
  sa.sin_family = AF_INET;
  sa.sin_port   = htons(port);     /* use our port  */
  sa.sin_addr.s_addr   = htonl(INADDR_ANY);   /* all local interfaces */

  bind(sockfd, (struct sockaddr *)&sa, sizeof(sa));

  listen(sockfd,DEFAULT_BACKLOG);

  chatserver(sockfd);  /* accept connections and chat */

  /* all done, clean up */
  close(sockfd);
  return 0;
}
```

Figure 97: MiniChat server, `main()`

```
void chatserver(int listener) {
  int clients[MAX_CONNECTIONS];
  int max,i,j, done, mlen, newsock, num_clients = 0;
  fd_set readfds;
  struct sockaddr_in peerinfo;
  socklen_t slen;
  char buff[MAXLINE],addr[INET_ADDRSTRLEN];

  done = 0;
  do {
    FD_ZERO(&readfds);
    FD_SET(listener,&readfds);
    max = listener;
    for(i=0;i<num_clients; i++) {
      FD_SET(clients[i],&readfds);
      if ( clients[i] > max )
        max = clients[i];
    }
    select(max+1,&readfds,NULL,NULL,NULL); /* wait for it... */

    if (FD_ISSET(listener,&readfds) && ( num_clients < MAX_CONNECTIONS − 1 )) {
      /* accept a new connection and add the client to the list */
      slen = sizeof(peerinfo);
      newsock = accept(listener, (struct sockaddr *)&peerinfo, &slen);

      inet_ntop(AF_INET, &peerinfo.sin_addr.s_addr,addr, sizeof(addr));
      printf("New connection from:  %s:%d\n", addr, htons(peerinfo.sin_port));
      clients[num_clients++] = newsock;
    } else {
      for(i=0;i<num_clients;i++) {
        if ( FD_ISSET(clients[i],&readfds) ) {
          /* read from this client and broadcast to all */
          slen = sizeof(struct sockaddr_in);
          mlen = recv(clients[i], buff, sizeof(buff),0);

          for(j=0;j<num_clients;j++)          /* broadcast to everyone else*/
            if ( i != j )      /* not self */
              send(clients[j], buff, mlen, 0);
        }
      }
    }
    if ( !strncmp(buff,"bye",3) )
      done = 1;
  } while (!done);
}
```

Figure 98: MiniChat server, `chat()`