Protocol Programming

- Learn how to use UNIX socket interface to set up Inter-Process Communication (IPC) between processes in the network.
- Learn how to set up datagram and stream sockets.
- Learn how to fork child process for concurrent server processing.
- Learn how to set up time-out in UNIX and use select system call to listen to multiple inputs (from users, timer, and network).
- Learn how to implement a simple protocol -- alternating bit protocol.
- Introduction to Protocol Programming project:
  Design and implement a Multimedia Intelligent Network Service (MINS)

  which allows users without the knowledge of others' whereabouts to communicate in "real-time" with text and graphics!

References:

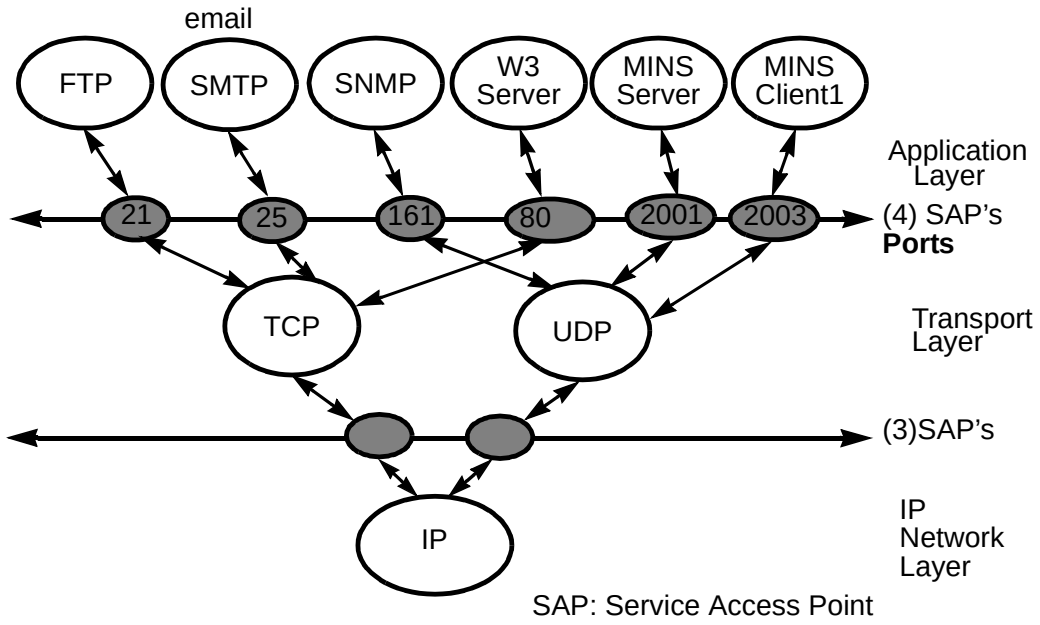
"UNIX Network Programming," by W. Richard Stevens, Prentice Hall, 1990 ISBN 0-13-949876-1.

"An Introductory 4.4BSD Interprocess Communication Tutorial" http://www-users.cs.umn.edu/~bentlema/unix/ipc/ipctut.html; Advanced IPC tutorial, http://www-users.cs.umn.edu/~bentlema/unix/advipc/ipc.html

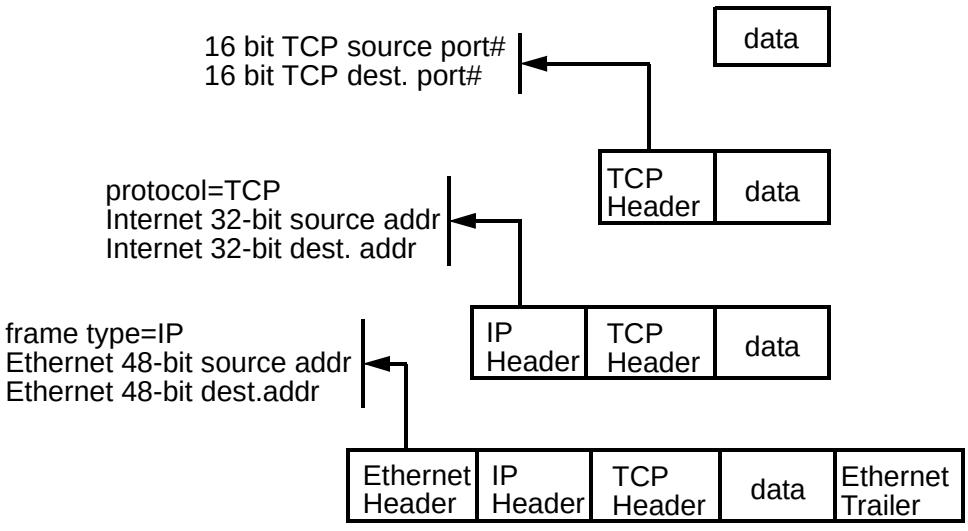
Protocol Implementation

- Stepwise refinement (transformation)
  Protocol spec. (such as CFSMs)→ code
- Automatic generation of program skeletons
- Implementation choices
  —modular structure
  procedures, processes, protocol entities, management functions,...
  —interface between protocol layers
  * internal
  * accessible to "user" or other layers
  —error handling of peer entity, user, others
  —buffer management (passing msgs between layers)
  —use of inter-task communications libraries (e.g., unix domain socket)
  e.g. Berkeley Socket, system V Transport Layer Interface (TLI) on UNIX.
  WinSock on Window, ...

UNIX Workstation Network Interface



Relation to Message Formats of TCP/UDP/IP



Berkeley Socket—An Application Program Interface

A set of library function calls forms an abstraction called **socket** to facilitate the programming involving inter-process communication (IPC).

Idea ⟶ The socket mimics the file I/O operations:
sending message—write(socket, msgbuf, strlen(msgbuf))
receiving message—read(socket, msgbuf, Maxmsglength)

However the creation of sockets is different, depending on the types of IPC:

- For process-to-process communication within a UNIX machine.
  a path name is used as an address to identify a socket,
  e.g. /tmp/cs522.chow.UA2Gui
  The space this type of socket addresses can be in is called **UNIX domain**.
  This type of socket is also called **UNIX domain socket.**
- For process-to-process communication between processes at two different machines using internet (TCP/UDP/IP) protocols.
  The 4-byte internet address of the host and a 2-byte port id is used to identify the socket, e.g., 128.198.162.62 and portId 21 identify the socket to which the FTP process on *sanluis* listen.
  See /etc/services for the designated port numbers.
  The space this type of socket addresses can be in is called **Internet domain**.
  This type of socket is also called **Internet domain socket.**

Socket Creation

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int family, int type, int protocol); // socket function call return fd
/* use man to see the description of each of the IPC routines and parameters.*/
```

Typical calls for Internet Domain sockets are

```
sockfd = socket(AF_INET, SOCK_STREAM, 0)
```

for byte stream connection-oriented service, default protocol is TCP.

```
sockfd=socket(AF_INET, SOCK_DGRAM, 0)
```

for datagram connectionless service, default protocol is UDP.

Typical calls for UNIX Domain sockets are

```
sockfd=socket(AF_UNIX, SOCK_STREAM, 0)
```

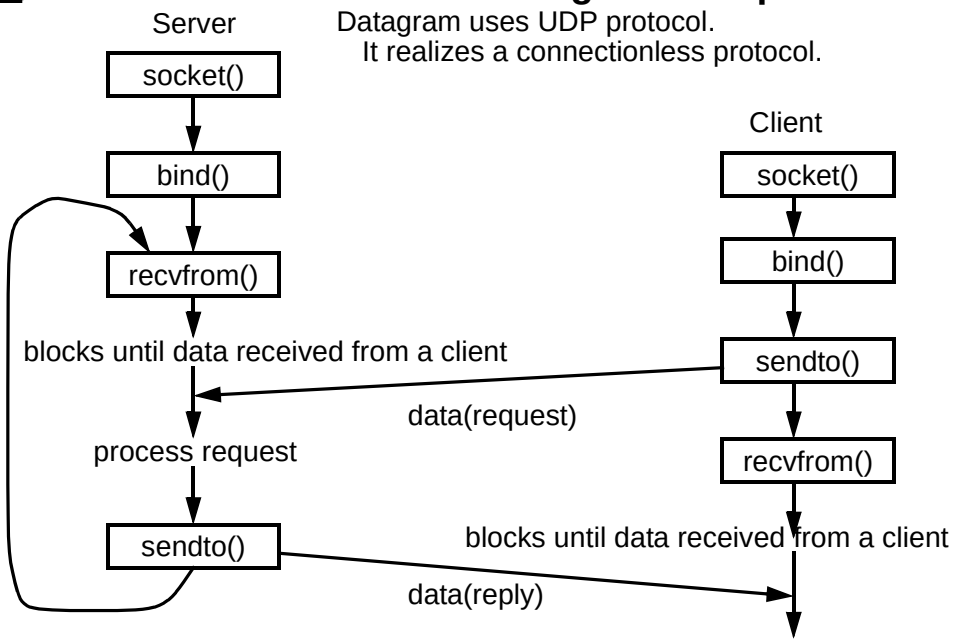
for byte stream connection-oriented service, default protocol is UNIX internal protocol.

```
sockfd=socket(AF_UNIX, SOCK_DGRAM, 0)
```

for datagram connectionless service, default protocol is UNIX internal protocol.

The return value is a file descriptor for later reference.
If socket() fails, return value is negative.

Flow Chart of Datagram Setup



Bind Internet Domain Socket to Address

int bind(int *sockfd*, struct sockaddr **myaddr*, int *addrlen*);
//Before the bind() call, specify the proper socket address in struct sockaddr:
//For a receiving Internet domain socket,

```
struct sockaddr_in from;
int length;

from.sin_family = AF_INET;
from.sin_addr.s_addr = INADDR_ANY; /*ask system to accept all nic ip addr.*/
from.sin_port = 0; /* Let system choose the port number */
/* 0< portno.< 1024 are reserved for privilege process */
/* portno. > 50000 reserved for non-privilege server */
/* to find the assigned port no., call getsockname()*/

bind(sockfd, &from, sizeof (from));
length = sizeof(from);
getsockname(sockfd, &from, &length)

printf("socket assigned port=%d\n", ntohs(from.sin_port)
```

Receiving Internet Datagram Messages

```
int sockfd;
struct sockaddr_in from;
char frombuf[2048], toBuf[2048];

/* after bind the socket to address */
while (flag) {
        n = recvfrom(sockfd, fromBuf, sizeof(fromBuf), 0, &from, &length);
        fromBuf[n] = 0; /* null terminate */
        printf("received msg=%s\n", fromBuf);
        /* do some processing according to the request */
        /* use the returned sender's socket address in the from structure to */
        /* send the reply message */
        sprintf(toBuf, "received message: %s\n", fromBuf);
        if (sendto(sockfd, toBuf, strlen(toBuf), 0, &from, sizeof(from)) < 0)
                perror("sending datagram message");
}
close(sockfd);
```

~cs522/project/socket contains sample programs, idgr.c (receiver), idgs.c (sender) for I386, Alpha, SPARC machines.

Sending Internet Datagram Messages

```
int sockfd;
struct sockaddr_in to;
struct hostent *hp, *gethostbyname();
char msgbuf[1024];

sockfd=socket(AF_INET, SOCK_DGRAM, 0)
toaddr.sin_family = AF_INET;

/* hostname and portno of the receiver may be entered from command line */
hp = gethostbyname(receiver_hostname); /* host name of receiver process */

/* bcopy(hp->h_addr, &to.sin_addr, hp->h_length); bcopy() is depreciated*/
memcpy(&to.sin_addr, hp->h_addr, hp->h_length);

to.sin_port = htons(toPort_no); /* port number of the receiver process */
strcpy(msgbuf, "how are you?");
if (sendto(sockfd, msgbuf, strlen(msgbuf), 0, &to, sizeof(to)) < 0)
                        perror("sending datagram message");
```

Running idgr and idgs

telnet zeppo.uccs.edu using cs522p1 with same password of cs522 on europa
zeppo> cd ~cs522p1/project/socket/inetdomain/SPARC
zeppo> idgr -d
socket has port #55401

login to elvis with cs522
elvis>cd project/socket/inetdomain/ALPHA
elvis> idgs -d zeppo 55401
socket has port #2455
sending message: packet 0!

The no. of bytes received=10
received msg=packet 0!
rcvd from sockaddr_in: Domain=2, Hostname=elvis.uccs.edu, Port=2455,
Address=128.198.1.117,
number of bytes received=29
received ack msg=received message: packet 0!

Receiver: Bind UNIX Domain Socket to Address

```
int sockfd;
struct sockaddr_un addr, to;
char frombuf[2048], toBuf[2048];

addr.sun_family = AF_UNIX;
sprintf(addr.sun_path, "/tmp/cs522.%s.server", getlogin());
unlink(addr.sun_path); /* if previous incarnation of socket still exists, kill it*/
if (bind(sockfd, &addr, sizeof(struct sockaddr_un))) {
        perror("binding name to datagram socket");
        exit(1);
}
 /* ls -F /tmp you will find a file with the same name there */
/* the socket file has srwxr-xr-x as access right and 0 length */
```

Why we put the login name in the socket pathname?
In what situation this will not uniquely identify the socket?

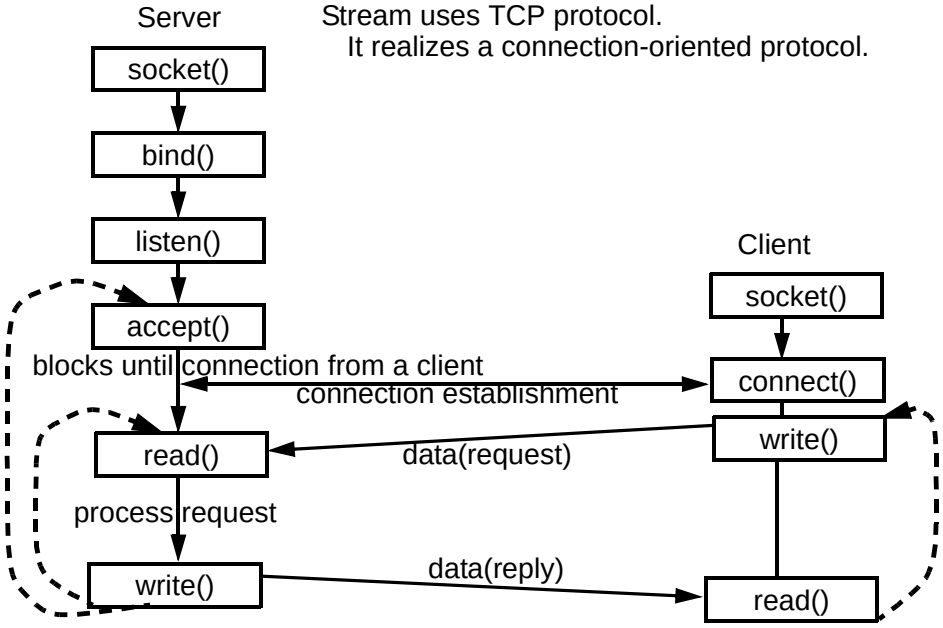
Receiver: Waiting/Reply UNIX Domain Datagram

```
    to.sun_family = AF_UNIX;
    sprintf(to.sun_path, "/tmp/cs522.%s.client", getlogin());
    printf("unix domain datagram client uses sun_path=%s\n", to.sun_path);
/* receive message */
 while (flag) {
    n = recv(sock, fromBuf, sizeof(fromBuf), 0);
    if (n < 0) perror("receiving datagram message");
    fromBuf[n] = 0; /* null terminate */
    printf("The no. of bytes received=%d\n", n);
    printf("received msg=%s", fromBuf);
    /* formulate the response */
    sprintf(toBuf, "received message: %s\n", fromBuf);
    if (sendto(sock, toBuf, strlen(toBuf), 0, &to, sizeof(to)) < 0)
            perror("sending datagram message");
    }
    unlink(addr.sun_path); /* clean up by removing the socket file descriptor */
    close(sock);
```


Sender: Sending UNIX Domain Datagram

```
/* create name structure with wildcard using INADDR_ANY */
    addr.sun_family = AF_UNIX;
    sprintf(addr.sun_path, "/tmp/cs522.%s.client", getlogin());
    unlink(addr.sun_path);
    if (bind(sock, &addr, sizeof(struct sockaddr_un))) {
            perror("binding name to datagram socket");
            exit(1);
    }
/*send message */
    sprintf(toBuf, "This is packet one!\n");
    if (sendto(sock, toBuf, strlen(toBuf), 0, &to, sizeof(to)) < 0) {
            perror("sending datagram message"); exit(1);l}
    n = recv(sock, fromBuf, 1024, 0);
    if (n < 0) perror("receiving datagram message");
    fromBuf[n] = 0; /* null terminate */
    printf("received ack msg=%s\n", fromBuf);
    unlink(addr.sun_path);
    close(sock);
```

Flow Chart of Stream Connection Setup



Stream Connections

```
int sockfd, newsockfd;
if ((sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket creation error"); exit(1); }
if (bind(sockfd,...) <0) { perror("binding error"); exit(1); }
if (listen(sockfd, 5) < 0) { perror("listen error"); exit(1); }/* allow 5 connection
    requests in queue, for Linux2.2 it is 5 established sockets*/
for (;;) {
    newsockfd = accept(sockfd, peer, addrlen); // from peer, know who request it
    if (newsockfd < 0) perror("accept error");
    doit(newsockfd);/* process the request, first use read()*/
}
close(newsockfd);
void doit(int newsockfd) {
    read(newsockfd, request, 1024); /* request can be struct or char * */
    /* analyze request; formulate the response */
    write(newsockfd, response, strlen(response);
}
```

If doit() is long, says involve DB query, all other requests need to wait.

Concurrent Processing With Fork()

```
int sockfd, newsockfd;
if ((sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket creation error"); exit(1);}
if (bind(sockfd,...) <0) { perror("binding error"); exit(1); }
if (listen(sockfd, 5) < 0) { perror("listen error"); exit(1); }/* allow 5 connection
    requests in queue, for Linux2.2 it is 5 established sockets*/
for (;;) {
    newsockfd = accept(sockfd, peer, addrlen); // from peer, know who request it
    if (newsockfd < 0) perror("accept error");
    if (fork() == 0) {                    /* child process */
        close(sockfd);
        doit(newsockfd);                  /* process the request, first use read()*/
        exit(0);
    }
    close(newsockfd);
    }
```

The decision to use fork() depends on whether the complexity of the task worths the overhead of forking process.

Running ist2r and two ist2s without fork processing

in the same SPARC directory containing idgr, zeppo> ist2r
before getsockname socket has port #0
socket has port #43652
in the same ALPHA directory containing idgs elvis> ist2s zeppo 43652
msg for receiver("$" to exit):dns_query elvis
europa> ist2s zeppo 43652
rcvd msg-->dns_query elvis msg for receiver("$" to exit):dns europa
What is your reply ("$" to break connection)?128.198.1.117
reply msg=128.198.1.117
msg for receiver("$" to exit):dns_query elbert
rcvd msg-->dns_query elbert
What is your reply ("$" to break connection)?128.198.162.68
reply msg=128.198.162.68
msg for receiver("$" to exit):$
Ending connection.../* accept next request */
rcvd msg-->dns europa
What is your reply ("$" to break connection)?128.198.1.247
reply msg=128.198.1.247

Running ist2r and two ist2s with fork processing

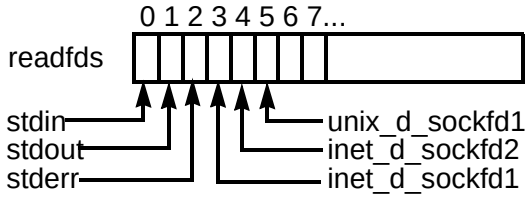
```
zeppo> ist2r -f
turn on fork processing mode. before getsockname socket has port #0
socket has port #43662
                    elvis> ist2s zeppo 43662
                    msg for receiver("$" to exit):dns_query elvis
                                        europa> ist2s zeppo 43662
                                        msg for receiver("$" to exit):dns europa
rcvd msg-->dns_query sanluis
What is your reply ("$" to break connection)?128.198.1.117
                    reply msg=128.198.1.117
                    msg for receiver("$" to exit):dns_query elbert
rcvd msg-->dns europa
What is your reply ("$" to break connection)?128.198.162.64
                                        reply msg=128.198.162.64
                                        msg for receiver("$" to exit):dns_query cs
```

Here europa's request will be accpeted and processed earlier by a different child process.

Handle multiple inputs and time-out

There is a special bit-vector type, fd_set, allowing you to indicate which file descriptors you would like to pay attention to.

```
#include <sys/types.h>
#include <sys/time.h>
FD_ZERO(fd_set *fdset); /* clear all bits in fdset */
FD_SET(int fd, fd_set *fdset); /*turn the bit for fd on in fdset */
FD_CLR(int fd, fd_set *fdset); /* turn the bit for fd off in fdset */
FD_ISSET(int fd, fd_set *fdset); /* test the bit for fd in fdset */
```




```
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microsecond */
```

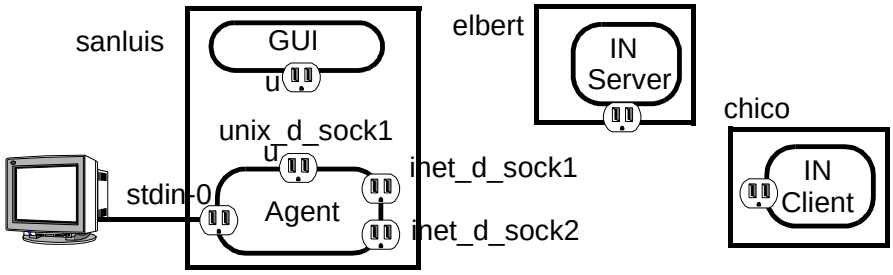
int select(int *maxfdpl*, fd_set **readfds*, fd_set **writefds*, fd_set **exceptfds*,
struct timeval * *timeout*);
maxfdpl: the number of file descriptors to be checked.
readfds: the input channels (file descriptors) to receive incoming msgs.

writedfds: the output channels (file descriptors) to send msgs.
exceptfds: the execptional signal channels (file descriptors).
timeout: a pointer to data that specify the time-out value.

If (timeout == NULL) select() will wait indefinitely until some of the channels have "actions"
If (timeout != NULL) select() will return when

- some of channels indicates "actions", returns value of select() will be the number of file descriptors that have "actions". The fd_set* will be overwritten with the bits set on for those with actions and bits set off for those without.

- time expires and time-out event occurs, return value is 0.

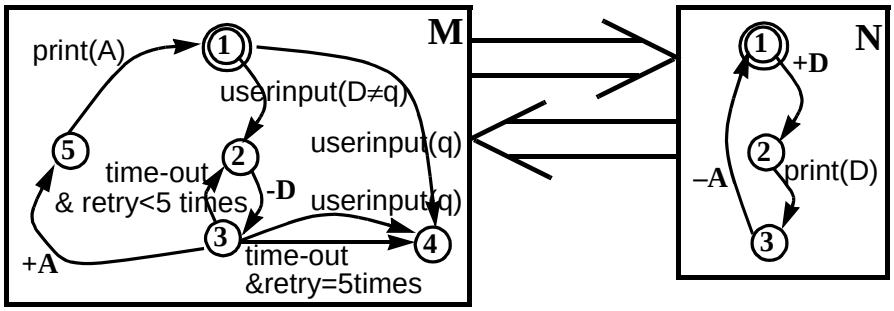


I/O multiplexing using select()

```
fd_set readfds;
int unix_d_sockfd, maxfdpl;
int inet_d_sockfd;
struct timeval timeout;
    /* assume you have create the above sockets properly */
    maxfdpl=(unix_d_sockfd>inet_d_sockfd)?unix_d_sockfd+1: inet_d_sockfd+1;
while (quitflag) {
    FD_ZERO(&readfds); FD_SET(0, &readfds); /* listen to stdin */
    FD_SET(unix_d_sockfd, &readfds); /* fd_set need to be reset, why? */
    FD_SET(inet_d_sockfd, &readfds); /* because select() overwrites the value*/
    timeout.tv_sec = 3; /* need to reset timeout in every select() call*/
    timeout.tv_usec= atol("500000"); /* 500000μseconds, use atol() to convert str*/
    if ((i=select(maxfdpl, &readfds, 0, 0, &timeout)) < 0) {
          perror("select error"); exit(1); }
    if (i==0) { printf("time-out"); /* return value of select() is 0*/ }
    if (FD_ISSET(0, &readfds)) { scanf("%s\n", buf); /* user input */ }
    if (FD_ISSET(unix_d_sockfd, &readfds)) { /* unix_d_sockfd receives msg */}
    if (FD_ISSET(inet_d_sockfd, &readfds)) { /* inet_d_sockfd receives msg */}
```


Mapping CFSM to C-code

How to implement the following simple protocol using datagram sockets.
Goal: Learn how to use select and time-out.



D represents any string user type in except 'q'.
When time-out happens, print out "time out n time" where n the number of time-outs during the sending of a message.
cs522/project/socket/mapping contain ex2r.c and ex2s.c
zeppo>SPARC/ex2r -t // this sets the receiver to throw away two out of 3 msgs
wetterhorn>I386/ex2s 5 0 zeppo 54333 // this sets sender timeout to 5sec.

Sample of C-code implements Machine M

```
int state = 1;
int quitflag = 1;
int retry = 1;
while (quitflag) do {
        switch (state) {
        case 1: /* state 1 */
                fflush(stdin); scanf("%s", buf);
                if (strcmp(buf, "q") ==0) {
                        /* userinput(q) */
                        state=4;
                } else {/* userinput(D), D¦q */
                        state=2;}
                break;
        case 3: /* state 3 */
                /* set time-out value */
                /* set fdest */
                if ((i=select(sinfd+1, fdset, 0, 0, timeout)) < 0) {
                        perror("select"); exit(1);}
                if (i==0) /* timeout */
                        if (retry == 5) state=4;
                        else {retry++; state=2;}
                if (FD_ISSET(0, fdset)) {
                        /* handle userinput */}
                if (FD_ISSET(sinfd, fdset)) {
                        /* handle inet msg */}
                break;
        case 4:
                quitflag = 0;
                break;
        ...
        } /* end of switch(state) */
} /* end of while(quitflag) */
```

The above while loop implementation of CFSM is straightforward but not efficient.

Don't forget to reset retry after receiving the acknowledgment.

Sending/Receiving Different Message Types

```
struct PenRecord {
    unsigned char msgType;
    unsigned char size;
    short X;
    short Y;};
struct ConnRecord {
    unsigned char msgType;
    unsigned char size;
    char name[20];
    char originator[20];};
union Record {                  /* see ~cs522/project/mins/msg.h for detail */
    PenRecord           pr;
    ConnRecord          nr;
    ACKRecord           ackr;
    ...} r;
r.pr.msgType = PEN; r.pr.X=320, r.pr.Y=600; r.pr.size=4;
r.nr.msgType=CONN; strcpy(r.nr.originator, getlogin()); strcpy(r.nr.name, dst_usr)
```

How to know the type of received msgs?

```
struct sockaddr_un toGui;
struct sockaddr_un fromGui;

n = recv(sockinet, &(r.pr.msgType), sizeof(r), 0);
switch(r.pr.msgType) {
case PEN:
    printf("receiving pen msg, X=%d, Y=%d\n", r.pr.X, r.pr.Y);
    if (sendto(socktoGui, &(r.pr.msgType), sizeof(r), 0, &toGui, sizeof(toGui)) < 0)
        perror("sending datagram msg"); /* relay the msg to GUI process */
    break;
case CONN:
    printf("receiving connect msg, originator =%s, dst user=%s\n",
        r.nr.originator, r.nr.name);
    /* update table, and send conn msg to Gui */
    break;}
```


Run penr and pen to understand Byte Ordering

```
zeppo> cd ~cs522p1/project/socket/byteorder/SPARC
zeppo> penr
socket has port #55468

    55468

 /* receive pen msg */
sizeof(pen)=24
The no. of bytes received=24
receive pen message:
    pen.header.sender = cs522
    pen.header.sessionID = 16777216
    pen.msg_type = 3
    pen.size = 8
    pen.x = 512
    pen.y = 8192 1,0,0,0
```

```
elvis> cd
elvis> cd project/socket/byteorder/ALPHA
elvis> pens -x 3 -s od -i 32 -t 5 -z 1 zeppo

socket has port #2472
sizeof(pen)=24
send pen message:
    pen.header.sender = cs522
    pen.header.sessionID = 1
    pen.msg_type = 3
    pen.size = 8
    pen.x = 3
    pen.y = 32
```

Byte Ordering Observation

zeppo is big endian machine. elvis is little endian machine.
sender fields is character array. Their value is not affected.
sessionID is integer. It is affected. value 1 becomes 1*2^24=16777216. since the least significant byte is interpreted by zeppo as most significant byte.
msg_type and size are character type (one byte), value not affected.
x and y are short integer (two bytes), value are affected.
pen.x originally is 2 after swap with other byte, it has weight of 256, and value becomes 2*256=512
pen.y originally is 32, after swap with other byte, it has weight of 256, and value becomes 32*256=8192.

You can modify the pens to send pen.x with value 1024. zeppo will print out pen.x as pen.x=4. Why is that?

I modified pens to allow changes of these pen fields.
elan> pens -s chow -y 1 -z 5 -x 512 -i 256 zeppo 55468
sender set to chow; The y is set to 1; The size is set to 5
The x is set to 512; The sessionID is set to 256