Introduction to Unix Sockets

- A socket is a type of descriptor that defines a bi-directional endpoint of communication

- Sockets are used as a basic building block for interprocess communication

- Associated with a socket is a data structure that includes a send buffer and a receive buffer

- A socket is created using the *socket()* system call, which takes as parameters:
  - □ protocol family
  - □ type of communication
  - □ specific protocol (often implicit)

- A socket is created without a name (address); a name is later *bound* to the socket

Ports

- A “port” is an abstraction, provided by both UDP and TCP, that defines a destination endpoints. Think of a port as a mailbox number. Some ports are reserved for specific services and are “published.” (see /etc/services)
  - □ a socket (an OS-specific entity) can be mapped to a port (a transport protocol entity)
  - □ packets arriving for a particular port are queued (in the OS) until a process extracts them
  - □ processes waiting at a port are typically blocked until packets arrive
- To communicate with a remote process, the sender must know the internet address of the destination machine as well as the port number that the process is waiting on.

Socket Naming

- Every socket must be *bound* with a name (or address) before it can be referenced

- Different protocols associated with a socket may have different naming structures

- The name space defined by a protocol family is called a *domain*

- The most commonly used families are:
  - □ Unix Domain (AF_UNIX)
    - ○ UNIX system internal protocols
    - ○ interprocess communication within same host and file system
    - ○ socket name is a file pathname
  - □ Internet Domain (AF_INET)
    - ○ interprocess communication among different hosts
    - ○ socket name includes internet address and port
    - ○ some port numbers are reserved for system use (see /etc/services)
    - ○ the name of a socket may be obtained using the system call *getsockname()*

Primary Socket Types

- The socket has the indicated *type*, which specifies the semantics of communication. Currently defined types are:

- datagram socket (SOCK_DGRAM)

  □ unreliable datagram communication

  □ record boundary (fixed maximum) is preserved

  □ implemented using the UDP protocol

- stream socket (SOCK_STREAM)

  □ reliable virtual circuit communication

  □ "record" boundary is *not* preserved

  □ implemented using the TCP protocol

- raw socket (SOCK_RAW)

  □ direct interface to the IP protocol

  □ used in new protocol development

  □ super-user only

Using Sockets with UDP

- Specify use of datagrams upon socket creation.
- If only one-way communication, the sending process need not bind a name to its socket.
- The receiving process *must* bind a name to its socket. The sending process references the name of the receiver's socket.
- Procedures:
  - □ socket creation: *socket(domain, type, protocol)*
  - □ name (address) binding: *bind(socket, name)*
  - □ send or receive through the socket: (several primitives available; see man pages)
  - □ close the socket: *close(socket)*
- Sending and receiving datagrams
  - □ *sendto()*: used for unconnected sockets
  - □ *read()*: primitive routine for sockets and other entities
  - □ *recvfrom()*: also retrieves name of sending socket

Datagram Socket Example - Receiver

```
// recv-dgram.cc -- datagram receive code
// After creating a port and binding a name to it, this program prints the port
// number, which is to be used by the sending side. In a loop, the program waits
// for a line of text then sends reply, rot13 encoded.

<include files omitted>

#define BUFLEN 356

int rot13 ( char *inbuf, char *outbuf ) ;

void main ( )
{
    int     sk ;            // socket descriptor
    sockaddr_in remote ;        // socket address for remote
    sockaddr_in local ;         // socket address for us
    char        buf[BUFLEN] ;       // buffer from remote
    char        retbuf[BUFLEN] ;    // buffer to remote
    int     rlen = sizeof(remote) ; // length of remote address
    int     len = sizeof(local) ;   // length of local address
    int     moredata = 1 ;      // keep processing or quit
    int     mesglen ;       // actual length of message

    // create the socket
    sk = socket(AF_INET,SOCK_DGRAM,0) ;

    // set up the socket
    local.sin_family = AF_INET ;         // internet family
    local.sin_addr.s_addr = INADDR_ANY ; // wild card machine address
    local.sin_port = 0 ;                 // let system choose the port

    // bind the name (address) to a port
    bind(sk,(struct sockaddr *)&local,sizeof(local)) ;

    // get the port name and print it out
    getsockname(sk,(struct sockaddr *)&local,&len) ;
    cout << "socket has port " << local.sin_port << "\n" ;
    // cout << "socket has addr " << local.sin_addr.s_addr << "\n" ;
```

Datagram Socket Example (cont.)

```
    while( moredata ) {
        // wait for a message and print it
        mesglen = recvfrom(sk,buf,BUFLEN,0,(struct sockaddr *)&remote, &rlen) ;
        buf[mesglen] = '\0' ;
        cout << buf << "\n" ;
        moredata = rot13(buf,retbuf) ;

        if( moredata ) {
        // send a reply, using the address given in remote
        sendto(sk,retbuf,strlen(retbuf),0,(struct sockaddr *)&remote, sizeof(remote));
        }
    }

    /* close the socket */
    close(sk);
}

/*
 * Encode message using rot13 scheme.
 *
 */

int rot13 ( char *inbuf, char *outbuf ) {
    int idx ;

    if( inbuf[0]=='.' ) return 0 ;

    idx=0 ;

    while( inbuf[idx]!='\0' ) {
        if( isalpha(inbuf[idx]) ) {
        if( (inbuf[idx]&31)<=13 )
            outbuf[idx] = inbuf[idx]+13 ;
        else
            outbuf[idx] = inbuf[idx]-13 ;
        } else
        outbuf[idx] = inbuf[idx] ;
        idx++ ;
    }
    outbuf[idx] = '\0';
    return 1 ;
}
```

Datagram Socket Example - Sender

```
// send-dgram.cc -- datagram sending code
// This program sends a message in a datagram, waits for, and prints a reply.  The
// destination machine name and destination port number are command line arguments.

<include files deleted>

#define MSG1 "Have you heard about the new corduroy pillows?\nThey are making headlines!"
#define MSG2 "."
#define BUFLEN 356

void main ( int argc, char *argv[] )
{
    int        sk ;        // socket descriptor
    sockaddr_in    remote ;    // socket address for remote side
    char        buf[BUFLEN] ;    // buffer for response from remote
    hostent        *hp ;        // address of remote host
    int        mesglen ;    // actual length of the message

    // create the socket
    sk = socket(AF_INET,SOCK_DGRAM,0) ;
    // designate the addressing family
    remote.sin_family = AF_INET ;
    // get the address of the remote host and store
    hp = gethostbyname(argv[1]) ;
    memcpy(&remote.sin_addr,hp->h_addr,hp->h_length) ;
    remote.sin_port = atoi(argv[2]) ;

    // send the message to the other side
    sendto(sk,MSG1,strlen(MSG1),0,(struct sockaddr *)&remote, sizeof(remote)) ;

    // wait for a response and print it
    mesglen = read(sk,buf,BUFLEN) ;
    buf[mesglen] = ’\0’;
    cout << buf << "\n" ;

    // send message telling it to shut down
    sendto(sk,MSG2,strlen(MSG2),0,(struct sockaddr *)&remote, sizeof(remote)) ;

    // close the socket and exit
    close(sk);
}
```

Structures Defined in System Header Files

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


// IPv4 AF_INET sockets:

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()
};

struct hostent
{
  char *h_name; /* Official name of host.  */
  char **h_aliases; /* Alias list.  */
  int h_addrtype; /* Host address type.  */
  int h_length; /* Length of address.  */
  char **h_addr_list; /* List of addresses from name server.  */
  #define h_addr h_addr_list[0] /* Address, for backward compatibility.  */
};
```

Unix Stream Sockets

- Also created with *socket()* system call
- Socket type: SOCK_STREAM
  - □ reliable virtual circuit communication
  - □ "record" boundary is *not* preserved
  - □ When used in AF_INET domain, implemented using the TCP protocol
- *socketpair()* - create a pair of connected sockets
  - □ generalization of *pipes*
  - □ only supports the AF_UNIX domain
  - □ same machine
- Timeouts and broken connections: If data is not transmitted within a reasonable length of time, then the connection is broken and subsequent calls will fail with ETIMEDOUT.
- Use *bind()* system call for both datagram and stream sockets
  - □ socket descriptor
  - □ name (family, host addr, port)
  - □ length of name
  - □ often let the OS choose the port

Stream Socket – Passive Side

- Await (listen for) connections
- Use *listen()* system call to prepare operating system for connection requests
- Parameters:
  - □ socket descriptor
  - □ backlog: defines the maximum length the queue of pending connections (usually 5)

Passive Side (cont.)

- Use the *accept()* system call
- Parameters
  - □ socket descriptor
  - □ name (family, host addr, port)
  - □ length of name
- Extracts the first connection on the queue of pending connections, creates a new socket, and allocates a new file descriptor for the socket.
- Normally blocks, but this can be turned off
- Returns descriptor of new socket.
- Why does accept() work this way?

Stream Socket - Active Side

- Initiate a connection on a socket
- Use the *connect()* system call
- Parameters:
  - □ socket descriptor
  - □ name (family, host addr, port)
  - □ length of name
- For stream sockets, attempts to make a connection to the socket named in the call.
- Errors
  - □ ETIMEOUT
  - □ ECONNREFUSED
  - □ ENETDOWN or EHOSTDOWN
  - □ ENETUNREACH or EHOSTREACH

Transferring Data

- Transmitting
  - □ *write()* as for files
  - □ *send()* uses flags to specify such requests as out-of-band transmission (MSG_OOB)
- Receiving
  - □ *read()* as for files
  - □ *recv()* uses flags to specify such requests as examining data without reading it (MSG_PEEK)

Stream Example - Passive Side

```
// recv-stream.cc -- passive side stream socket example
void main ( )
{
    int        sk, sk2 ;            // socket descriptors
    sockaddr_in    remote ;        // socket address for remote
    sockaddr_in    local ;            // socket address for us
    char        buf[BUFLEN] ;        // buffer from remote
    char        retbuf[BUFLEN] ;    // buffer to remote
    int        rlen = sizeof(remote) ;    // length of remote address
    int        len = sizeof(local) ;    // length of local address
    int        moredata = 1 ;        // keep processing or quit
    int        mesglen ;        // actual length of message

    // create the socket
    sk = socket(AF_INET,SOCK_STREAM,0) ;
    // set up the socket
    local.sin_family = AF_INET ;         // internet family
    local.sin_addr.s_addr = INADDR_ANY ; // wild card machine address
    local.sin_port = 0 ;                 // let system choose the port
    // bind the name (address) to a port
    bind(sk,(struct sockaddr *)&local,sizeof(local)) ;
    // get the port name and print it out
    getsockname(sk,(struct sockaddr *)&local,&len) ;
    cout << "socket has port " << local.sin_port << "\n" ;
    // tell OS to queue (up to 1) connection requests
    listen(sk, 1);

    // wait for connection request, then close old socket
    sk2 = accept(sk, (struct sockaddr *)0, (int *)0) ;
    close(sk);

    if(sk2 == -1)
      cout << "accept failed!\n" ;
    else { while( moredata ) {
             // wait for a message and print it
             mesglen = read(sk2,buf,BUFLEN);
             buf[mesglen] = '\0' ;
             cout << buf << "\n" ;
             moredata = rot13(buf,retbuf) ;
             if( moredata ) {
             // send a reply
             write(sk2,retbuf,strlen(retbuf));
       } } }
    close(sk2);
    exit(0);
}
```

Stream Example - Active Side

```
// send-stream.cc -- active side stream socket example

#define MSG1 "Have you heard about the new corduroy pillows?\nThey are making headlines!"
#define MSG2 "."
#define BUFLEN 356

void main ( int argc, char *argv[] )
{
    int        sk ;        // socket descriptor
    sockaddr_in    remote ;    // socket address for remote side
    char        buf[BUFLEN] ;    // buffer for response from remote
    hostent        *hp ;        // address of remote host
    int        mesglen ;    // actual length of the message

    // create the socket
    sk = socket(AF_INET,SOCK_STREAM,0) ;
    // designate the addressing family
    remote.sin_family = AF_INET ;
    // get the address of the remote host and store
    hp = gethostbyname(argv[1]) ;
    memcpy(&remote.sin_addr,hp->h_addr,hp->h_length) ;
    // get the port used on the remote side and store
    remote.sin_port = atoi(argv[2]) ;

    // connect to other side
    if(connect(sk, (struct sockaddr *)&remote, sizeof(remote)) < 0) {
        cout << "connection error!\n" ;
        close(sk);
        exit(1);
    }

    // send the message to the other side
    write(sk,MSG1,strlen(MSG1));
    // wait for a response and print it
    mesglen = read(sk,buf,BUFLEN) ;
    buf[mesglen] = '\0';
    cout << buf << "\n" ;
    // send message telling it to shut down
    write(sk,MSG2,strlen(MSG2));
    // close the socket and exit
    close(sk);
}
```

Why does accept() return a new socket descriptor?

Server Example - Passive Side

```
// recv-server.cc -- handles translation for multiple clients
void main ( )
{

    <variables as in earlier example>

    // create the socket
    sk = socket(AF_INET,SOCK_STREAM,0) ;
    // set up the socket
    local.sin_family = AF_INET ;         // internet family
    local.sin_addr.s_addr = INADDR_ANY ; // wild card machine address
    local.sin_port = 0 ;                 // let system choose the port

    // bind the name (address) to a port
    bind(sk,(struct sockaddr *)&local,sizeof(local)) ;
    // get the port name and print it out
    getsockname(sk,(struct sockaddr *)&local,&len) ;
    cout << "socket has port " << local.sin_port << "\n" ;
    // tell OS to queue (up to 5) connection requests
    listen(sk, 5);

    // we loop forever, taking connections and forking new servers
    for( ; ; ) {
        // wait for connection request
        sk2 = accept(sk, (struct sockaddr *)0, (int *)0) ;
        if(sk2 == -1)   cout << "accept failed!\n" ;
        else {
           if( fork()==0 ) {
                // this is the child process ...
                close(sk) ;            // sk is no longer needed
                while( moredata ) {
                    // wait for a message and print it
                    mesglen = read(sk2,buf,BUFLEN);
                    buf[mesglen] = '\0' ;
                    cout << buf << "\n" ;
                    moredata = rot13(buf,retbuf) ;
                    if( moredata ) {  // send a reply
                       write(sk2,retbuf,strlen(retbuf));
                  } }
                exit(0);
            }
            // this is the parent, so we no longer need sk2 ...
            close(sk2) ;
   } } }
```

Server Example - Active Side

```
// send-server.cc -- active side for server example

#define ENDMSG "."
#define BUFLEN 356
void main ( int argc, char *argv[] )
{
    int          sk ;        // socket descriptor
    sockaddr_in  remote ;    // socket address for remote side
    char         buf1[BUFLEN] ;    // buffer for sending to remote
    char         buf2[BUFLEN] ;    // buffer for response from remote
    hostent      *hp ;        // address of remote host
    int          mesglen ;    // actual length of the message

    // create the socket
    sk = socket(AF_INET,SOCK_STREAM,0) ;
    // designate the addressing family
    remote.sin_family = AF_INET ;
    // get the address of the remote host and store
    hp = gethostbyname(argv[1]) ;
    memcpy(&remote.sin_addr,hp->h_addr,hp->h_length) ;
    // get the port used on the remote side and store
    remote.sin_port = atoi(argv[2]) ;

    // connect to other side
    if(connect(sk, (struct sockaddr *)&remote, sizeof(remote)) < 0) {
        cout << "connection error!\n" ;
        close(sk);
        exit(1);
    }
    // loop, reading input and sending to other side, until a single ‘.’ is typed
    cin.getline (buf1, sizeof(buf1));
    while (buf1[0] != ’.’) {
        // send the message to the other side
        write(sk,buf1,strlen(buf1));
        // wait for a response and print it
        mesglen = read(sk,buf2,BUFLEN) ;
        buf2[mesglen] = ’\0’;
        cout << buf2 << "\n" ;
        // get next line of input
        cin.getline (buf1, sizeof(buf1));
    }
    // send (last) message telling it to shut down
    write(sk,buf1,strlen(buf1));
    // close the socket and exit
    close(sk);
}
```

Select System Call

- Parameters:
  - □ number of file descriptors
  - □ &readmask
  - □ &writemask
  - □ &exceptmask
  - □ timeout
- Operation
  - □ examines the I/O descriptor sets whose addresses are passed as mask parameter
  - □ replaces passed mask with map of those ready
  - □ passes back the number that are ready
- How can we use this?

Important Functions

Find how to use the following functions:

- socket(); accept(); bind(); close(); connect(); fork(); listen();
- getaddrinfo(); gethostbyaddr(); gethostbyname(); gethostname(); getsockname();
- htonl(); htons(); ntohl(); ntohs(); inet_aton(); inet_ntoa(); inet_ntop(); inet_pton();
- sendto(); recvfrom(); send(); select();

Resources:

- 'man' on Unix/Linux machines
- online socket programming guides, e.g., http://beej.us/guide/bgnet/
- Unix Network Programming (Volume 1) by W. Richard Stevens)

Internet Address:
data structures & function

- Address: 35.8.10.140
- Name: www.msu.edu
- Representation preferred by programs?

- struct addrinfo {
-     int   ai_flags;   // AI_PASSIVE, AI_CANONNAME, etc.
-     int   ai_family;   // AF_INET, AF_INET6, AF_UNSPEC
-     int   ai_socktype; // SOCK_STREAM, SOCK_DGRAM
-     int   ai_protocol; // use 0 for "any"
-     size_t   ai_addrlen;   // size of ai_addr in bytes
-     struct sockaddr *ai_addr;   // struct sockaddr_in or _in6
-     char   *ai_canonname; // full canonical hostname
-     struct addrinfo *ai_next;   // linked list, next node
- };
- Function getaddrinfo( )

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

struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};

// (IPv4 only--see struct in6_addr for IPv6)
// Internet address (a structure for historical reasons)
struct in_addr {  uint32_t s_addr;  // that's a 32-bit int (4 bytes) };
```

struct hostent {

- char *h_name; //This is the "official" name of the host.
- char **h_aliases; //These are alternative names for the host, represented as a null-terminated vector of strings.
- int h_addrtype; //This is the host address type; in practice, its value is always either AF_INET or AF_INET6
- int h_length; //This is the length, in bytes, of each address.
- char **h_addr_list; //This is the vector of addresses for the host.
- char *h_addr; //This is a synonym for h_addr_list[0];

}

Important Function.

Find how to use the following functions:

- socket(); accept(); bind(); close(); connect(); fork(); listen();
- getaddrinfo(); gethostbyaddr(); gethostbyname(); gethostname(); getsockname();
- htonl(); htons(); ntohl(); ntohs(); inet_aton(); inet_ntoa(); inet_ntop(); inet_pton();
- sendto(); recvfrom(); send(); recv(); select();

Resources:

- 'man' on Unix/Linux machines
- online socket programming guides, e.g., http://beej.us/guide/bgnet/
- Unix Network Programming (Volume 1) by W. Richard Stevens), online at: http://proquestcombo.safaribooksonline.com.proxy2.cl.msu.edu/0-13-141155-1