Today's Agenda

1. Overview of Socket Programming
2. Project Description

Socket Programming

- Socket Programming Overview
- BSD Unix C Socket Programming API
- Handle asynchronous input/output from multiple socket descriptors

Additional Readings on BSD Socket on class website;
Online Project I Description

Soothing words

Don’t worry too much about the sense of information overload you may get from this lecture. There is a lot of detail about sockets that you don’t need to remember, because you can look it up. Pay attention to the patterns and outlines of what’s going on, and use reference material for the details as you program.

Refreshing

- Host (NIC card) identified by unique IP address
- Network application/process identified by port number
- Network connection identified by a 5-tuple (src ip, src port, dst ip, dst port, protocol)

- Two kinds of Internet transport services provided to applications
  - Connection-oriented TCP
  - Connectionless UDP

Huh?

- We haven’t yet covered the transport layer in class, and yet you’re being called on to use sockets to access transport services
- This is partly a matter of semester mechanics, and partly because everything has to start somewhere
- Treat this as your intro to the essentials of transport, with details to follow

Socket Programming API

- API: Application Programming Interface
- Socket analogous to door
  - sending process shoves message out door
  - sending process assumes transport infrastructure on other side of door that brings message to socket at receiving process
  - connection between sockets set-up/managed by OS

- Host or server controlled by app developer
- Host or server controlled by OS
- TCP
- UDP

Internet

TCP/IP protocols

buffers, variables
What APIs Needed?

Connection-Oriented TCP
• How to create socket (door)
• How to establish connection
  - Client connects to a server
  - Server accepts client req.
• How to send/recv data
• How to identify socket
• How to close socket (door)

Connectionless UDP
• How to create socket
• How to send/recv data
• How to identify socket
• How to close socket

Socket: Conceptual View

Another conceptual view

But why?
• Sockets provide a (mostly) uniform API to access many different network services
• Services may be implemented in the kernel or in user space
• Common API across multiple protocol stacks (UNIX domain, IPv4, IPv6, ISO/OsI, AppleTalk, SNA, Bluetooth, etc.)
• Common API across multiple layers

Creating a Socket

Format: int socket(family, type, protocol);
• domain, service and protocol parameters:
  - family: PF_INET/AF_INET, PF_UNIX/AF_UNIX
  - service:
    • SOCK_DGRAM: datagram service (i.e., UDP)
    • SOCK_STREAM: byte stream service (i.e., TCP)
  - protocol: usually 0 for default type
• return a socket descriptor, like a file descriptor in Unix

```
#include <sys/types.h>
#include <sys/socket.h>
if (sd = socket(AF_INET, SOCK_STREAM, 0) <0 ) {/*socket error*/};
```

Binding to a Local Address

Format: int bind(int sockid, struct sockaddr *addr, int addresslen);
• Servers need to call it
• optional for connection-oriented clients

```
int sd; struct sockaddr_in myaddr;
if (sd = socket(AF_INET, SOCK_DGRAM,0) )<0 {/*socket error*/};
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(5100); /* > 5000 */
myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
/* INADDR_ANY: allow OS to choose IP address for any interface */
if ( bind(sd, (struct sockaddr *) &myaddr, sizeof(myaddr)) < 0)
{ /* bind error */ }
```
Socket Address Structures

- Predefined data structures:

```c
struct sockaddr_in { /* INET socket address info */
    short sin_family; /* set me to AF_INET */
    u_short sin_port; /* 16 bit num, network byte order*/
    struct in_addr sin_addr; /* 32 bit host address */
    char sin_zero[8]; /* not used */
};
```

```c
struct in_addr {
    union {
        struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
        struct { u_short s_w1,s_w2; } S_un_w;
        u_long S_addr;
    } S_un;
    #define s_addr S_un.S_addr
};
```

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

---

find out own IP address

- Find out own host name

```c
#include <unistd.h>
int gethostname(char *name, int namelen);
```

- name : is character array way to store the name Of the machine with null terminated character.
- namelen : size of the character array

This function puts the name of the host in name. It returns 0 if ok, -1 if error.

---

find out own IP address (cont)

- Address stored in h_addr is in struct in_addr form.
- To obtain "." separated ip address we could use

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```c
char *inet_ntoa(const struct in_addr in);
```

- The routine inet_ntoa() returns a pointer to a string in the base 256 notation d.d.d.
CO Server: listen() and accept()

listen():
  int listen (int sd, int backlog);
  - notify OS/network ready to accept requests
  - backlog: max. listen queue size while waiting for accept()
  - does not block/wait for requests

accept():
  int accept (int sd, struct sockaddr *fromaddr, int addrlen);
  - use socket (sd) for accepting incoming connection requests
  - create new socket (return value) for data exchange w/ client
  - block until client connects, cannot selectively accept

Connecting to Server: connect()

Format:
  int connect (int sd, struct sockaddr *toaddr, int addrlen);

Client issues connect() to:
  - establish remote address, port (connection-oriented, connectionless)
  - establish connection with server (connection-oriented only)
  - CO: block until accepted or error
    - fails: server/host not listening to port, timeout

Sending and Receiving Data

- Connection-oriented
  - send() and recv(); or read() and write()
  - Format
    - int send(int sockfd, char *buff, int nbytes, int flags)
    - int recv(int sockfd, char *buff, int nbytes, int flags)

- Connectionless
  - sendto() and recvfrom()
  - Format
    - int sendto(int sockfd, char *buff, int nbytes, int flags, struct sockaddr *to, int addrlen)
    - int recvfrom(int sockfd, char *buff, int nbytes, int flags, struct sockaddr *from, int addrlen)

Identifying socket

- Getting own address and port associated to a connected socket:
  #include <sys/types.h>
  #include <sys/socket.h>
  int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
  - returns the current name for socket s. The namelen parameter should be initialized to indicate the amount of space pointed to by name. On return it contains the actual size in bytes of the name returned.

Identifying socket (cont)

#include <sys/types.h>
#include <sys/socket.h>
int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
returns the name of the peer connected to socket s.

Closing a Socket

- Format - int close(int fd)
  - Call this function to close a created socket.
  - System takes care of buffered data.
A summary of BSD Socket

<table>
<thead>
<tr>
<th>protocol</th>
<th>localAddr</th>
<th>localPort</th>
<th>remoteAddr</th>
<th>remotePort</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn-oriented server</td>
<td>socket()</td>
<td>bind()</td>
<td>listen(), accept()</td>
<td></td>
</tr>
<tr>
<td>conn-oriented client</td>
<td>socket()</td>
<td>bind()</td>
<td>connect()</td>
<td></td>
</tr>
<tr>
<td>connectionless server</td>
<td>socket()</td>
<td>bind()</td>
<td>recvfrom()</td>
<td></td>
</tr>
<tr>
<td>connectionless client</td>
<td>socket()</td>
<td>bind()</td>
<td>sendto()</td>
<td></td>
</tr>
</tbody>
</table>

BSD Socket Programming Flows

- Server
  - bind()
  - listen()
  - accept()
  - receive
  - send

- Client
  - connect()

Data Conversion & Name/Address Translation

- Integers:
  - Little endian: least significant byte first (e.g., DEC, Intel)
  - Big endian: most significant byte first (e.g., Sun, HP, SGI)
- Network byte order: big endian
  - htonl(): network-to-host byte order conversion, long(32 bits)
  - htons(): host-to-network byte order conversion, long(32 bits)
  - ntohs(), htons(): for short integer (16 bits)
- Host name to IP address (numeric): gethostbyname()
- Host IP address to name: gethostbyaddr()
- IP address conversion:
  - From numeric to dotted quad: inet_ntoa()
  - You may need this for printing the IP address of a host
  - From dotted quad to numeric: inet_addr()

Example of Stream Server: echo

```c
/* stream server: echo what is received from client */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int s, t, sinlen;
    struct sockaddr_in sin;
    char msg[80];

    if (argc < 2) {
        printf("%s port\n", argv[0]); /* input error: need port no! */
        return -1;
    }
    if ( (s = socket(AF_INET, SOCK_STREAM, 0 ) ) < 0 ) { /* create socket*/
        perror("socket"); return -1;
    }
    sin.sin_family = AF_INET;              /*set protocol family to Internet */
    sin.sin_port = htons(atoi(argv[1]));  /* set port no. */
    sin.sin_addr.s_addr  = INADDR_ANY;   /* set IP addr to any interface */
    if (bind(s, (struct sockaddr *)&sin, sizeof(sin) ) < 0 ) {
        perror("bind"); return -1;
    }
    sin.sin_family = AF_INET;              /*set protocol family to Internet */
    sin.sin_port = htons(3000);           /* set port no. */
    sin.sin_addr.s_addr = htonl(INADDR_ANY);  /* set IP addr to any interface */
    if (bind(s, (struct sockaddr *)&sin, sizeof(sin) ) < 0 ) {
        perror("bind"); return -1;
    }
    t = accept(s);                        /* accept a connection */
    if ( t < 0 ) {
        perror("accept"); return -1;
    }
    /* read message */
    if ( recvfrom(t, (char *)msg, 80, 0, &sin, &sinlen) < 0 ) {
        perror("read"); return -1;
    }
    /* print message */
    printf("%s\n", msg);
    /* write message */
    if ( sendto(t, msg, strlen(msg), 0, &sin, sinlen) < 0 ) {
        perror("sendto"); return -1;
    }
    return 0;
}
```
Example of Stream Server: echo (cont'd)

```c
/* server indicates it's ready, max. listen queue is 5 */
if (listen(s, 5)) {
    printf("listen"); /* listen error*/
    return -1;
} sinlen = sizeof(sin);
while (1) {
    /* accepting new connection request from client, 
    socket id for the new connection is returned in t */
    if ( (t = accept(s, (struct sockaddr *) &sin, &sinlen) ) < 0 ) {
        perror("accept"); /* accept error*/
        return -1;
    }
    printf( "From %s:%d.
", 
        inet_ntoa(sin.sin_addr), ntohs(sin.sin_port) );
    if ( read(t, msg, sizeof(msg) ) <0 ) { /* read message from client */
        perror("read"); /* read error */
        return -1;
    }
    /* close connection, clean up sockets */
    if (close(t) < 0) { perror("close"); return -1;}
    return 0;
}
```

Example of Stream Client: echo

```c
/* stream client: send a message to server */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char *argv[]) {
    int s, n;
    struct sockaddr_in sin;
    struct hostent *hptr;
    char msg[80] = "Hello World!";
    if ( argc < 3 ) {
        printf ( "%s host port
", argv[0] );   /* input error: need host & port */
        return -1;
    }
    if ( (s = socket(AF_INET, SOCK_STREAM, 0 ) ) < 0) { /* create 
        socket*/
        perror("socket");  /* socket error */
        return -1
    } sin.sin_family = AF_INET;              /*set protocol family to Internet */
    sin.sin_port = htons(atoi(argv[2]));  /* set port no. */
    if ( (hptr =  gethostbyname(argv[1]) ) == NULL){
        fprintf(stderr, "gethostname error: %s", argv[1]);
        return = -1;
    }
    memcpy( &sin.sin_addr, hptr->h_addr, hptr->h_length);
    if (connect (s, (struct sockaddr *)&sin, sizeof(sin) ) < 0 ){
        perror("connect"); return -1;   /* connect error */
    }
    if ( write(s, msg, strlen(msg)+1) < 0 ) { /* send message to server */
        perror("write");    return -1; /*  write error */
    }
    if ( ( n = read(s, msg, sizeof(msg) ) ) <0 ) { /* read message from server */
        perror("read");return -1; /* read error */
    }
    printf( "%d bytes: %s
", n, msg);  /* print message to screen */
    /* close connection, clean up socket */
    if (close(s) < 0) {
        perror("close"); return -1;}
    return 0;
}
```

Compiling and Executing

```bash
hobbes% g++ -o echo-server echo-server.c -lsocket -lnsl
hobbes% g++ -o echo-client echo-client.c -lsocket -lnsl
hobbes% echo-server 5700 &
hobbes% echo-client kepler 5700

From 128.101.34.75:32938.
12 bytes: Hello World!
```

A Few Words about Port Numbers

- 1-255: standard services (21 ftp, 25 SMTP, 80 HTTP)
- 1-1023: available only to system
- 1024-4095: usable by system & users
- 5000+: usable only by users
What We Have Learned

• BSD Unix C Socket Programming API
  - Socket operations: system calls into OS

What we will learn next

• Writing server and client to handle asynchronous input output from multiple socket descriptors

scenario

Think about the case when you write a program

• Will take input from the std input and also from other multiple connected peers at the same time.
• Input from all of descriptors are asynchronous

One possible solution -

Probing all the descriptors in round robin manner for activity
Problem
Wastage of CPU time

Select()

• Select() is a system call that lets you probe the operating system to discern whether or not there is I/O to be done on various file descriptors (monitoring several sockets at the same time).
• It is way for program to instruct the kernel to wake up this process whenever there is an event detected in any of the descriptor passed to the function as argument.

Select (cont)

```c
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>

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

Select Arguments

• The nfds argument specifies the range of file descriptors to be tested. The select() function tests file descriptors in the range of 0 to nfds-1.
• If the readfds argument is not NULL, it points to an object of type fd_set that on input specifies the file descriptors to be checked for being ready to read, and on output indicates which file descriptors are ready to read.

Select Arguments (cont)

• If the writefds argument is not NULL, it points to an object of type fd_set that on input specifies the file descriptors to be checked for being ready to write, and on output indicates which file descriptors are ready to write.
• If the exceptfds argument is not NULL, it points to an object of type fd_set that on input specifies the file descriptors to be checked for error conditions pending, and on output indicates which file descriptors have error conditions pending.
Select Arguments (timeout)

- the timeout argument is not a NULL pointer, it points to an object of type struct timeval that specifies a maximum interval to wait for the selection to complete.
- If the timeout argument points to an object of type struct timeval whose members are 0, select() does not block.

Select Arguments (return value)

- If successful, select() returns the number of ready descriptors that are contained in the bit masks. If the time limit expires, select returns zero and sets errno to EINTR. On failure, it returns -1 and sets errno to one of the following values:
  - EBADF
  - EINTR
  - EISDIR
  - EINVAL

How to set descriptor values for select() arguments

- FD_CLR(fd, &fdset) -- clears the bit for the file descriptor fd in the file descriptor set fdset.
- FD_ISSET(fd, &fdset) -- check for readiness
  - Returns a non-zero value if the bit for the file descriptor fd is set in the file descriptor set pointed to by fdset, and 0 otherwise.

select() Example

servsock = startserver();
FD_ZERO(&livesdset);
FD_SET(servsock, &livesdset);
livesdmax = servsock+1;
While(1)
FD_ZERO(&readset);
readset = livesdset;
nfound = select( livesdmax, &readset, NULL, NULL, NULL);
if( fnound < 0 ) {
  if( errno == EINTR ) { /* system interrupt handling */
    continue;
  }
  perror( "Select call problem ");
  exit(1);
}
/* look for messages from live clients */
for (frsock=3; frsock < livesdmax; frsock++) 
  }