Socket Programming

Mahalingam Ramkumar
Mississippi State University, MS

August 27, 2014
1. Introduction

2. Socket Programming
   - Opening a Socket
   - Bind, Listen, Accept
   - Connect, Send, Recv
   - TCP Client and Server

3. UDP Socket Commands
   - Socket Address Structure

4. Client and Server Code

5. Odds and Ends
   - Blocking vs Non Blocking Sockets
Every computer connected to the Internet has a unique IP address.

Corresponding to each IP address, there may be many application processes willing to accept connection requests at a specific port number.

Servers *listen* — wait for connection requests.

Clients *initiate* connection requests.

Once a connection is established, both client and server can send and receive (any number of) bytes.

Connections can then be closed.
## Client Server Applications

<table>
<thead>
<tr>
<th>Server</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  Listen at some port number</td>
<td>1  Request Connection</td>
</tr>
<tr>
<td>2  Accept connection request</td>
<td>3  Send Query</td>
</tr>
<tr>
<td>4  Process Query</td>
<td>6  Receive response</td>
</tr>
<tr>
<td>5  Send Response</td>
<td>7/8 Close connection</td>
</tr>
<tr>
<td>7/8 Close Connection</td>
<td></td>
</tr>
</tbody>
</table>
Addressing servers and Clients

- Client needs to know IP address and port number of the server.
- Usually clients know only the domain name (for e.g., yahoo.com)
- DNS (domain name system/service) — an application that translates domain names to IP addresses (like 411 Directory service);
- Port number depends on the type of application (standard port numbers)
- Connection request made by specifying IP address and port number of the server,
- Also conveys IP address and port number used by the client.
Sockets and Socket Programming

- Software library for network application development;
- Sockets are similar to file handles
- File handles bound to a file name (and path); Sockets bound to socket address;
- Socket address has two components — an IP address and a port number
- Two types of sockets - TCP (transport control protocol) and UDP (user datagram protocol)
- TCP sockets can be connected.
- The socket library provides various system calls like socket(), bind(), listen(), connect(), accept(), send(), recv(), close()
Socket (System) Calls

- `sd = socket(OPTIONS)` Creates a socket. `sd` is a handle to the socket.
- `bind(sd,FROMADDRESS)` (local socket address).
- `listen()`
- `connect(sd,TOADDRESS)` (remote socket address).
- `sd = accept(sd)` (accept() returns a connected socket)
- `send(sd,buffer, num_bytes)`.
- `recv(sd,buffer, num_bytes)`.
- `close(sd)`.
- These functions are the interfaces provided by the transport layer (to the application layer above).
Opening a Socket

int socket(int, int, int);
Output: socket handle
Inputs: domain, type, protocol
TCP:

int sd;
sd = socket(AF_INET, SOCK_STREAM, 0);

UDP:

sd = socket(AF_INET, SOCK_DGRAM, 0);

Returns an integer (handle) sd or -1 on failure.
int bind(int, sockaddr*, int);

int check = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));

bind() returns -1 on failure.
int listen(int, int);
Inputs: socket handle, BACKLOG

check = listen(sd,BACKLOG);

listen() returns -1 on error;
BACKLOG is the number of connection requests that can be queued
int accept(int, void *, int *);
Inputs: socket handle, pointer to client address, size of returned address.

sockaddr_in clientaddr;
int addresssize, newsd;
newsd = accept(sd, (void *)&clientaddr, &addresssize);
accept() returns -1 on error
Connect

int connect(int, sockaddr *, int);
Returns -1 on failure
Inputs: socket handle, destination address, size of address.
**Send and Receive**

```c
int send(int, const void *, int, int);
Output: Number of bytes sent, 0, or -1
Input: socket handle, message buffer, buffer length in bytes, and FLAGS
```

```c
int recv(int, const void *, int, int);
Output: Number of bytes received, 0, or -1
Input: socket handle, message buffer, maximum buffer length in bytes, and FLAGS
```
TCP Client and Server

**Server**
- lsd = socket();
- bind(lsd, address);
- listen(lsd);
- sd = accept(lsd);
- recv(sd, · · ·);
- send(sd, · · ·);
- close(sd);

**Client**
- sd = socket();
- connect(sd, dest_address);
- send(sd, · · ·);
- recv(sd, · · ·);
- close(sd);
send() and recv() can be used only after connection has been established
Can not use send() and recv() for UDP
Use sendto() and recvfrom() instead

```c
int sendto(int sd, const void *msg, size_t len, int flags, const struct sockaddr *to, int len);

int recvfrom(int sd, void *buf, size_t len, int flags, struct sockaddr *from, int *len);
```

For POSIX-2 compliance the last (6th) parameter in sendto() and recvfrom() should be socklen_t instead of int
UDP Client and Server

Server:
- \( sd = \text{socket}(); \)
- \( \text{bind}(sd, \text{address}); \)
- \( \text{recvfrom}() - \text{equivalent to listen()} \) and \( \text{accept}() \)
- process request; create response
- \( \text{sendto}() \)

Client
- \( sdc = \text{socket}(); \)
- create query
- \( \text{sendto}(); \)
- \( \text{recvfrom}(); \)
- process response
int bind(int, sockaddr *, int);
Inputs: socket handle, pointer to address structure, size of address structure

struct sockaddr {
    unsigned short sa_family;
    char sa_data[14];}

struct sockaddr_in {
    short int sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];}

struct in_addr {unsigned long s_addr;}

Address Format
Address Format

```c
unsigned short port = 4345;
sockaddr_in soaddr;
soaddr.sin_family = AF_INET;
soaddr.sin_port = htons(port);
memset(&soaddr.sin_zero, 0, 8);

htons() - host-to-network byte order conversion for shorts
htons(), htonl(), ntohs(), ntohl()
```
Specifying IP Address

Automatically fill local IP Address

```c
soaddr.sin_addr.s_addr = INADDR_ANY;
```

IP address specified as a string

```c
char * IPaddr = "123.134.245.123";
inet_aton(IPaddr, &soaddr.sin_addr);
```

IP from domain name through a DNS query

```c
hostent * h;
char * dname = "yahoo.com";
h = gethostbyname(dname);
soaddr.sin_addr = (struct in_addr *)h->h_addr;
```
struct hostent * = gethostbyname(char *);

struct hostent { ...
char ** h_addr_list;
}
#define h_addr h_addr_list[0];

h_addr is a pointer to a sequence of four characters
Cast h_addr to a in_addr pointer
Recall that in_addr is just unsigned long
int sd, addsize, backlog=10, newsd;
sockaddr_in serveraddr, clientaddr;
unsigned short port = 4349;
sd = socket(AF_INET, SOCK_STREAM, 0);
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(port);
serveraddr.sin_addr.s_addr = INADDR_ANY;
memset(&(serveraddr.sin_zero), 0, 8);
check = bind(sd, (struct sockaddr *)&serveraddr, 
sizeof(serveraddr));
check = listen(sd, backlog);
while(1) {
    newsd = accept(sd, (void *)&clientaddr, &addsize);
    //connection specific recv() and send()
    //process request by
    //recv() ing from newsd and
    //send() ing using newsd;
}
close(sd);

Typically connection specific tasks performed in a new thread / process so that the server can go back to waiting on accept().
int csd;
unsigned short port = 4349; // dest port
char * serverIP = "123.134.245.123"; // dest IP
sockaddr_in serveraddr;
csd = socket(AF_INET, SOCK_STREAM, 0);
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(port);
inet_aton(serverIP, &serveraddr.sin_addr);
connect(csd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));

inet_aton() - coverts character string “X.Y.Z.W” to unsigned long
int csd, check, numbytes;
unsigned short port = 4349;
char * serverIP = "123.134.245.123";
char sbuf[256], rbuf[256];
sockaddr_in serveraddr;
//create application specific query in sbuf
...
csd = socket(AF_INET, SOCK_STREAM, 0);
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(port);
inet_aton(serverIP, &serveraddr.sin_addr);
connect(csd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
numbytes = send(csd, (const void *)sbuf, 100, 0);
shutdown(csd, SHUT_WR);
numbytes = recv(csd, (const void *)rbuf, 256, 0);
close(csd);
//Process response (in rbuf) from server ...

shutdown(csd, SHUT_RD), shutdown(csd, SHUT_RDWR) can also be used. shutdown(csd, SHUT_RDWR) closes both reading and writing.
send() and recv() in Server

char rbuf[256], sbuf[256];
int numbytes;
numbytes = recv(newsd, (const void *)rbuf, 256, 0);
//create application specific response in sbuf
numbytes = send(newsd, (const void *)sbuf, 200, 0);
close(newsd);
int main(int argc, char* argv[]) {
    int sd, newsd; char buf[256];
    struct sockaddr_in serveraddr, clientaddr;
    socklen_t sasize = sizeof(struct sockaddr_in);
    sd = socket(AF_INET, SOCK_STREAM, 0);
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(atoi(argv[1]));
    serveraddr.sin_addr.s_addr = INADDR_ANY;
    bind(sd, (struct sockaddr*)&serveraddr, sasize);
    listen(sd, 10);
    while(1) {
        newsd=accept(sd,(struct sockaddr*)&clientaddr,&sasize);
        recv(newsd, buf, 256, 0); send(newsd, "Hello", 6, 0);
        close(newsd); }
    close(sd);
}
int main(int argc, char* argv[]) {
    int csd;
    char buf[256];
    struct sockaddr_in serveraddr;
    socklen_t sasize = sizeof(struct sockaddr_in); //16
    csd = socket((AF_INET, SOCK_STREAM, 0);
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(atoi(argv[2]));
    inet_aton(argv[1], &serveraddr.sin_addr);
    connect(csd, (struct sockaddr *)&serveraddr, sasize);
    send(csd, "Hi", 3, 0); shutdown(csd, SHUT_WR);
    recv(csd, buf, 255, 0);
    close(csd);
}
Local and Remote Address

getpeername() tells you who is connected at the other end of the socket
getsockname() tells you who is connected at your end of the socket

```c
int getpeername(int sd, struct sockaddr *their_address, socklen_t *namelen);
int getsockname(int sd, struct sockaddr *my_address, socklen_t *namelen);

gethostname() - get host name (from /etc/hosts)

int gethostname(char *name, size_t len);
```
#include <sys/socket.h>  //socket, send, recv, bind, listen, 
                          //accept, getsockname, getpeername..  
#include <netdb.h>  //for hostent, gethostbyname()  
#include <netinet/in.h>  //definitions of protocols  
#include <arpa/inet.h>  //inet_ntoa, inet_aton etc
Command line switches for gcc or g++

- gcc server.c -o server -lnsl -lsocket (for Solaris)
- gcc server.c -o server -lnsl -lsocket -lresolv
- gcc server.c -o server -lnsl -lsocket -lresolv -lxnet
- gcc server.c -o server (should do for Linux / MAC OS-X)
- gcc server.c (will result in a.out)
- chmod +x server (to make the output executable)
# Socket Programming in Windows

```c
#include <winsock.h>
#include <netinet/in.h> //definitions of protocols
#include <arpa/inet.h> //inet_ntoa, inet_aton etc
{
  int sd; ..... 
  WORD wVersionRequested;
  WSADATA wsaData;
  wVersionRequested = MAKEWORD( 1, 1 );
  WSAStartup(wVersionRequested, &wsaData) //returns 0
  ..... //on success
  ....
  closesocket(sd); //for unix just close(sd);
  WSACleanup();
}

In Visual studio, project settings, link wsock32.lib
```
By default accept(), recv(), recvfrom() functions block
For example nb = recv(sd, buf, 256, 0) would not return unless
- 256 bytes have been received (nb=256), or
- less than 256 bytes received (say nb=100), but the sender does not have anything to send anymore, or
- sender closed connection (nb=0), or
- an error occurs (nb=-1)
Blocked sockets just wait till the transaction is “completed”
Non-blocked sockets can return - for instance they can return with nb=5. It is up to the programmer to make sure that all bytes are received, by calling recv again to get the remaining bytes.
Or call recv till 0 is received (or -1)
Setting a socket to non-blocking mode

#include <fcntl.h>
...
sd = socket(SF_INET, SOCK_STREAM, 0);
fcntl(sd, F_SETFL, O_NONBLOCK);
...

If a socket is set to non blocking we have to periodically poll the socket to see if any bytes have been received / sent.
Why do we need non-blocking sockets?
Is there a better way?
int select(int, fd_set*, fd_set*, fd_set*, struct timeval*)

maxfd - highest file descriptor - first input

fd_set fdR, fdW, fdE (file descriptor sets - handles that have to be monitored for Reading, Writing and Exceptions)

struct timeval - specifies timeout for select()

FD_ZERO(), FD_SET(), FD_CLR(), FD_ISSET()

Say we need to listen to a TCP socket and UDP socket.

If we did not have select(), execution thread will wait at accept() for TCP socket and at recvfrom() for UDP socket

select() needs to monitor two handles - TCP socket and UDP socket

Assign both handles to fdR using FD_SET()
socket (. , SOCK_STREAM , .); bind (); listen ();
socket (. , SOCK_DGRAM , .); bind ();
int nfd = sdt > sdu ? sdt+1 : sdu+1;
for (;;) {
    fd_set fdR;
    FD_ZERO(&fdR); //FD_ZERO(fd_set*);
    FD_SET(sdt, &fdR); //FD_SET(int, fd_set*)
    FD_SET(sdu, &fdR); //FD_CLR(int, fd_set*)
    select(nfd, &fdR, NULL, NULL, NULL); //execution thread
    if (FD_ISSET(sdt, &fdR)) {} //waits here
    if (FD_ISSET(sdu, &fdR)) {} //FD_ISSET(int, fd_set*)
}