Socket Programming

Mahalingam Ramkumar
Mississippi State University, MS

January 19, 2009
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 some port number.

Servers listen - wait for incoming attempts to establish a connection.

Clients initiate connection requests.

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

- **Server**
  - Listen (at some port number)
  - If a connection attempt is sensed
    - accept connection request
    - process query (send response)
    - close connection

- **Client**
  - request connection
  - send query
  - receive response
  - close connection
Client Server Applications

Server
0. Listen at some port number
2. Accept connection request
4. Process Query
5. Send Response
7/8. Close Connection

Client
1. Request Connection
3. Send Query
6. Receive response
7/8. Close connection
Addressing servers and Clients

- Client needs to know IP address and port number to send the connection request to the server.
- Usually clients know only the domain name (yahoo.com)
- DNS (domain name system/service) - an application that translates domain names to IP addresses
- 411 Directory service?
- How do servers determine IP address / port number of clients?
Overview
Socket Programming

Sockets and Socket Programming

- Software tools (library) for application developers to interact with the transport layer
- Sockets bound to some IP and port number
- Sockets can be connected.
- The socket library provides various system calls
- Examples: socket(), bind(), listen(), connect(), accept(), send(), recv(), close()
Socket Programming

Socket (System) Calls

- \( sd = \text{socket}(\text{OPTIONS}) \). Creates a socket. \( sd \) is a handle to the socket.
- \( \text{bind}(sd, \text{FROMADDRESS}) \).
- \( \text{connect}(sd, \text{TOADDRESS}) \).
- \( \text{send}(sd, \text{buffer}, \text{num\_bytes}) \).
- \( \text{recv}(sd, \text{buffer}, \text{num\_bytes}) \).
- \( \text{close}(sd) \).
- Two more functions are needed in the server side
  - \( \text{listen}(sd) \)
  - \( \text{newsd} = \text{accept}(sd) \)
- These functions are the *interfaces provided by the transport layer* (to the application layer above).
Web Browser / Web Server

- Say “http://www.abc.com/def/page.html” entered in the browser address bar
- Browser parses string - separates domain name - “www.abc.com” and file name / path - “/def/page.html”
- DNS (uses a function call gethostbyname() ) to discover IP address from domain name
- Server port number 80 for HTTP
- Create a socket, send connection request to server
- Server accepts connection
- Client sends application data “HTTP 1.1 GET /def/page.html”
- Server sends page.html as response
- Client and server close connection.
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.
Address Format

```c
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

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;
```
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
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);
    ProcessRequest(newsd);
}
close(sd);

Typically ProcessRequest() spawns a new thread of execution so that the server can go back to waiting on accept().
Connect

int connect(int, sockaddr *, int);
Returns -1 on failure
Inputs : socket handle, destination address, size of address.
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() - converts character string “X.Y.Z.W” to unsigned long
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

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
Shutdown()

Closing down one half of a socket connection;
int shutdown(int sd, int how);
how: SHUT_RD, SHUT_WR, SHUT_RDWR
int csd, check, numbytes;
unsigned short port = 4349;
char * serverIP = "123.134.245.123";
char sbuf[256], rbuf[256];
sockaddr_in serveraddr;
CreateQuery(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);
ProcessResponse(rbuf);
ProcessRequest() in Server

```c
char rbuf[256], sbuf[256];
int numbytes;
umbytes = recv(newsd, (const void *)rbuf, 256, 0);
CreateResponse(rbuf, sbuf);
umbytes = send(newsd, (const void *)sbuf, 200, 0);
close(newsd);
```
int main(int argc, char* argv[]) { //Client.c
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);
memset(&(serveraddr.sin_zero), '\0', 8);
connect(csd, (struct sockaddr *)&serveraddr, sasize);
send(csd, "Hi", 3, 0);
shutdown(csd, SHUT_WR);
recv(csd, buf, 255, 0);
close(csd);
}
int main(int argc, char* argv[]) { //Server.c
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); }
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
send() and recv() can be used only after *connection has been established*
Obviously, you cannot 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);
```

```c
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 = socket();`
- `bind(sd, address);`
- `recvfrom() - equivalent to listen() and accept()`
- `ProcessRequest();`
- `sendto()`

Client
- `sdc = socket();`
- `sendto();`
- `recvfrom();`
- `DoWhatever();`
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

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 -lInsl -lsocket (for Solaris)
- gcc server.c -o server -lInsl -lsocket -lresolv
- gcc server.c -o server -lInsl -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
```
Blocking vs Non Blocking sockets

- 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?
select()
select()

```c
sdt = socket(.SOCK_STREAM, .); bind(); listen();
sdu = 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*)
}"
```
Socket Programming in Windows

- Windows sockets are used only for network programming
- `select()` can be used *only for sockets*