Introduction to Sockets and Sockets Programming

Programming TCP/IP in Unix is based on sockets, while Windows uses winsock. Both are similar but the implementation is somewhat different. Here we will focus on Unix BSD sockets. However, the same concepts apply to Windows sockets (although there are higher-level libraries for network programming on Windows that are easier to use, based on event-driven programming).

Ports

Each process that wants to communicate with another process identifies itself to the TCP/IP protocol suite by one or more ports. A port is a 16-bit number, used by the host-to-host protocol to identify to which higher-level protocol or application program (process) it must deliver incoming messages. In the OSI model, a port is referred to as an SAP (Service Access Point).

As some higher-level programs are themselves protocols, standardized in the TCP/IP protocol suite, such as TELNET and FTP, they use the same port number in all TCP/IP implementations. Those "assigned" port numbers are called *well-known ports* and the standard applications are known as *well-known services*. Both UDP and TCP use the same port numbers.

The "well-known" ports are controlled and assigned by the Internet Assigned Numbers Authority (IANA) and on most systems can only be used by system processes or by programs executed by privileged users. The assigned "well-known" ports occupy port numbers in the range 0 to 1023. The ports with numbers in the range 1024-65535 are not controlled by the IANA and on most systems can be used by ordinary user-developed programs. Client-developed programs will generally request an available port from the operating system, so ports may change from one invocation to the next.

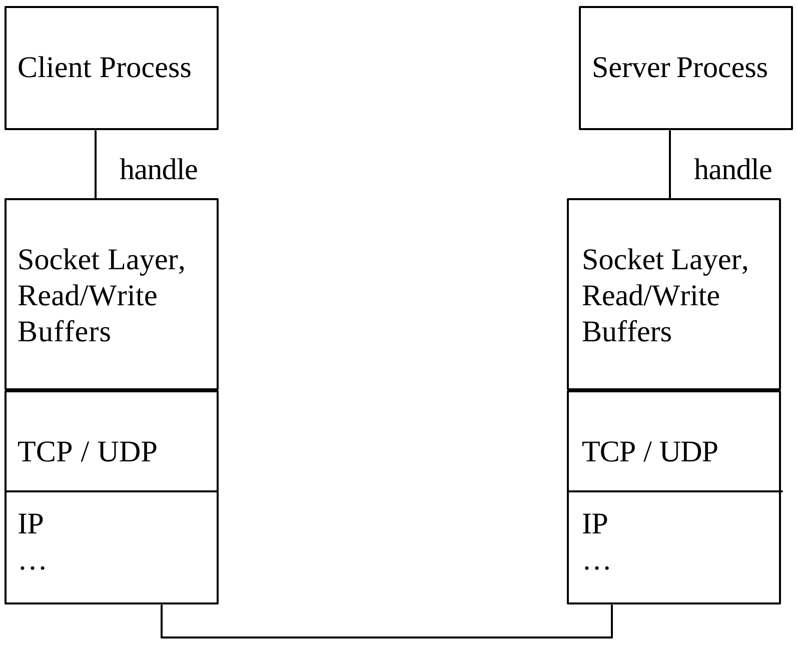
Sockets

A socket is what allows a process to communicate with other processes. This means that your applications will need a socket to communicate with TCP. From the perspective of your program, a socket is a lot like a Unix file handle, but it requests network services from the operating system.

A socket address is the triple: {protocol, local-address, local-port}
In the TCP/IP suite, for example: {tcp, 137.229.134.205, 6666}

In client/server processing, you will need to have a socket on the server side and a separate socket on the client side. In this case, the client might have a socket of {tcp, 137.229.134.204, 10304} and it communicates with a server at {tcp, 137.229.134.205, 25}.

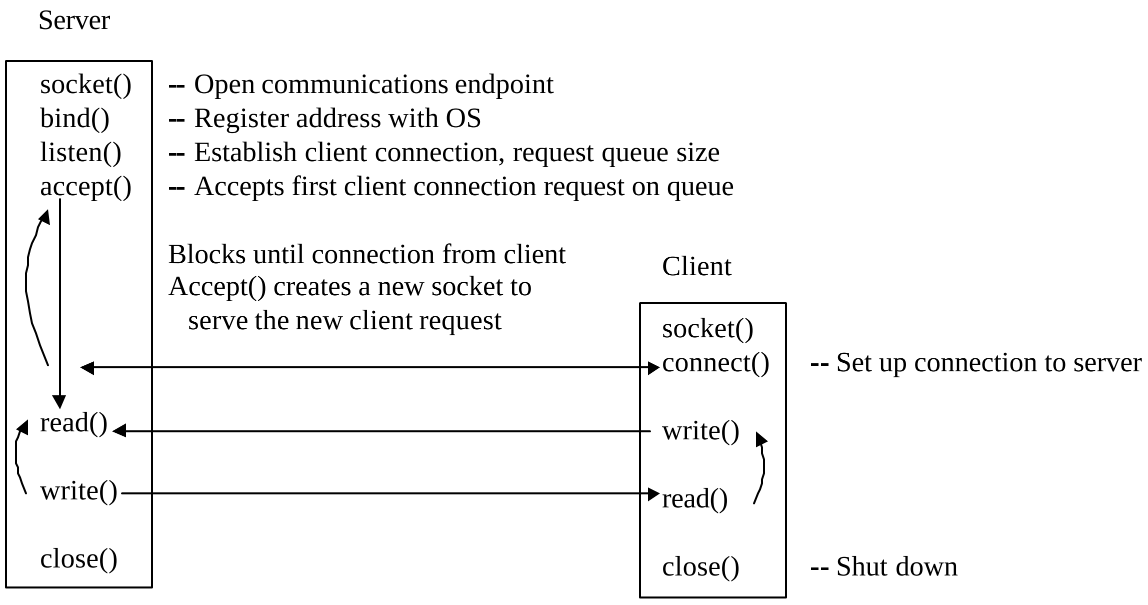
An overview of where sockets lie in terms of the TCP/IP layers is shown below:


Client Process
Server Process
handle
handle
Socket Layer, Read/Write Buffers
Socket Layer, Read/Write Buffers
TCP / UDP
TCP / UDP
IP
…
IP
…

TCP Sockets

Let’s start with using sockets with TCP. Recall that TCP is connection-oriented, so we will need to first set up our connection, acknowledge the connection, then send our data before shutting down. Since TCP is reliable and in-order, we should have no errors and data will be received in the order it was sent.

The figure below illustrates the sequence of calls the client and server must make:


Server
socket()
bind()
listen()
accept()
– Open communications endpoint
– Register address with OS
– Establish client connection, request queue size
– Accepts first client connection request on queue
Blocks until connection from client
Accept() creates a new socket to serve the new client request
read()
write()
close()
Client
socket()
connect()
write()
read()
close()
-- Set up connection to server
-- Shut down

Here is a description of these calls in more detail. To use these from C/C++, you will typically need to include the following header files:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

Some others you might want/need to include are:

```
#include <sys/socketvar.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/time.h>
```

When compiling, you will likely need the –lnsl and –lsocket flags on Solaris and SunOS systems.

socket(2): Both clients and servers, initialize a socket

```
int  socket(int family, int type, int protocol);

family = AF_INET , could be AF_UNIX, AF_APPLETALK, etc.
type = socket type.  SOCK_STREAM for TCP, SOCK_DGRAM for UDP
        SOCK_RAW (access to innards, need root)
protocol = 0   (default)

ex:   s = socket(AF_INET, SOCK_STREAM, 0);
```

bind(2): Server mostly (client rarely), associate socket with a port address

```
int  bind(int socket, const struct sock_addr *address, size_t address_len);
```

This function returns –1 if there is an error, 0 if success.

socket = A valid socket returned from socket()
address = A struct sockaddr_in, described soon. The port goes in here.
address_len = sizeof struct sockaddr_in

Use a port of 0 to have the system assign a port.

```
ex: bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

listen(2): Server only, listen for socket connections and buffer queue

```
int listen(int socket, int backlog);
```

socket = A valid socket from socket()
backlog = Size of queue, enables concurrent connections

This call also initialized the TCP state machine to start getting connections.
It returns –1 if error, 0 otherwise.

Ex: `listen(sockfd, 5);`

accept(2) : Server only, accepts a new connection upon client connect()

```
int accept(int socket, struct sockaddr *address, size_t *address_len);
```

socket = A valid socket from socket()
address = A struct sockaddr_in, described soon.
address_len = Length of the sockaddr_in

This call normally **blocks** (could use select). **Note that address_len is called by value-result**. We must initially set it to the length, but it returns the client's port/IP/len in the address and len parameters.

The function returns –1 if error, a new socket otherwise.

Ex:

```
  clilen = sizeof(cli_addr);
  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
```

not:

```
  clilen = sizeof(cli_addr);
  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, clilen);
```

sockaddr Structure:

sockaddr is a generic structure . It looks like the following:

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

This will store the IP address and port. This generic structure maps into two more detailed structures, struct sockaddr_in for INET, and struct sockaddr_un for Unix. We care about the sockaddr_in format here:

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

struct in_addr {
   unsigned long s_addr;
};
```

These are overlay structures, with the sockaddr_in being overlaid on the sockaddr. The sin_family should be AF_INET, and you will need to set the port and the address. The sin_zero is used to pad the structure, and should be set to all 0's.

Important: The address and port must be in **network byte order**. In NBO, the most significant byte comes first. In a computer's internal representation, the least significant byte may come first. Fortunately there are functions to help you do the conversion:

htons() – Host to Network Short, convert a short (use for PORT)
htonl() – Host to Network Long, convert a long (use for ADDRESS)
ntohs() – Network to host, short
ntohl() – Network to host, long

Fortunately you won't need to use many of these except perhaps htons to set the port, because there are other functions that will look up IP addresses for you and return them in network byte order.

Ex (this could be called before bind()) :

```
   struct sockaddr_in serv_addr;
   bzero((char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sin_family      = AF_INET;
   serv_addr.sin_addr.s_addr = htonl(ADDR);
   serv_addr.sin_port        = htons(PORT);
```

gethostbyname(3) : This looks up an IP address by name

```
struct hostent * gethostbyname(char *name);
```

name = The alphabetic name, e.g. www.yahoo.com

This function takes a DNS name and maps it to an IP address "somehow". It might be through /etc/hosts, DNS, yellow pages, or buffered in the OS.
If successful, it returns a pointer to the struct hostent. Otherwise, it returns null.

Given the struct hostent*, you can access the IP address via the h_addr field:

Ex:

```
struct hostent *hp;
hp = gethostbyname("www.math.uaa.alaska.edu");
sin.sin_addr = *((struct in_addr *)hp->h_addr);
```

connect(2) : Client call to initiate connection

```
int connect(int socket, const struct sockaddr *address, size_t address_len);
```

socket = A valid socket
address = A struct sockaddr_in, described above.
address_len = Length of the sockaddr_in

This is used by the client to connect to the server's port and address defined in the struct sockaddr_in parameter. The client does an implicit bind and a port is assigned by the OS to the client. If there is an error, -1 is returned. You should check for these error conditions, because TCP might not be able to successfully connect with the server.

read(2)/write(2) : Read or write from the socket as a file descriptor.

```
int read(int sock, void *buffer, size_t num_bytes);
int write(int sock, void  *buffer, size_t num_bytes);
```

sock = socket to read/write to
buffer = data to send
num_bytes = Size of the buffer

This call is used to read/write data. Both return –1 if there is an error, otherwise it returns the number of bytes that were read/written.

With read, you may need to use a loop because TCP might return less than you expect. For example if you ask for 1024 bytes, you might get two 512 byte packets. Write will block until your data is sent, so there is no need for a loop.

You can also use **send(2)** and **recv(2)** to accomplish these same tasks. See the man pages for information on these; recv has the advantage of an additional flag , MSG_PEEK, that allows you to peek at the data but not take it out of the buffer until the next recv call.

close(2) : Shuts down the socket

```
int close(int fd);
```

fd = Your socket.

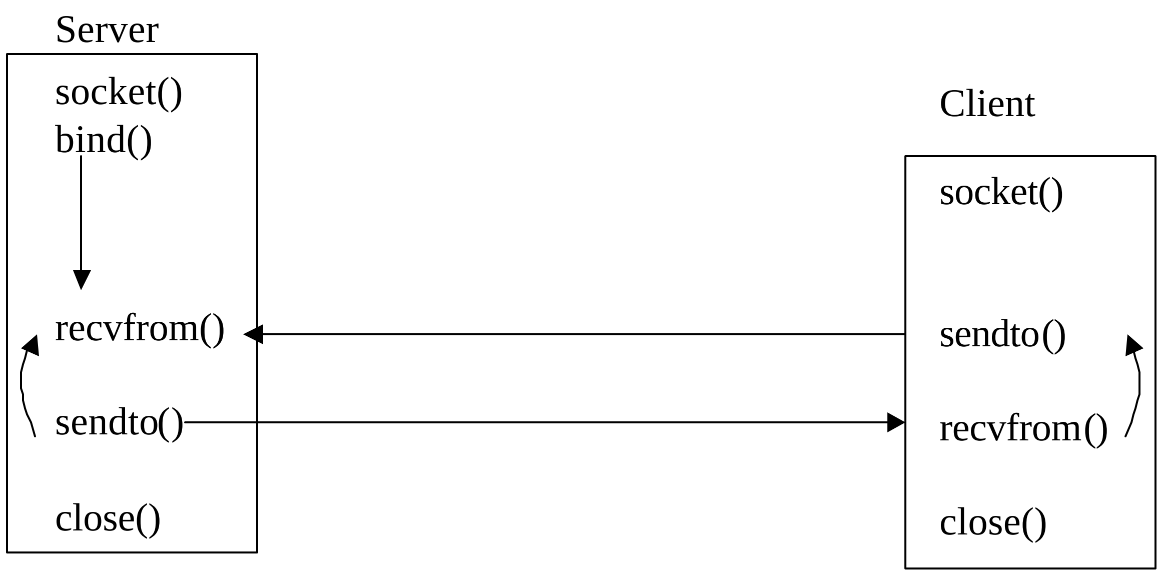
This closes down the socket and terminates communications.

Here is a complete sample client that will print the time from CWOLF:

```
/* Simple TCP stream client that connects to CWOLF's port for time
   prints the results and quits.
*/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 13         /* daytime port */
#define MAXDATASIZE 500 /* max number of bytes we can get at once */
#define SERV "cwolf.uaa.alaska.edu"

int main()
{
 int sockfd, numbytes;
 char buf[MAXDATASIZE];
 struct hostent *he;
 struct sockaddr_in their_addr; /* connector's address information */

 if ((he=gethostbyname(SERV)) == NULL) {  /* get the host info */
      perror("gethostbyname");
        exit(1);
 }

 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      perror("socket");
        exit(1);
 }

 their_addr.sin_family = AF_INET;      /* host byte order */
 their_addr.sin_port = htons(PORT);    /* short, network byte order */
 their_addr.sin_addr = *((struct in_addr *)he->h_addr);
 bzero(&(their_addr.sin_zero), 8);     /* zero the struct */

 if (connect(sockfd, (struct sockaddr *)&their_addr,
             sizeof(struct sockaddr)) == -1) {
            perror("connect");
            exit(1);
 }

 if ((numbytes=recv(sockfd, buf, MAXDATASIZE, 0)) == -1) {
            perror("recv");
            exit(1);
 }

 buf[numbytes] = '\0';
 printf("Received: %s",buf);
 close(sockfd);
 return 0;
}
```

UDP Sockets

Now that we've described using TCP for sockets, UDP is actually much simpler because we don't have to set up a connection and do all the handshaking. Instead it is connectionless. The figure below depicts the basic steps to follow:



In UDP, we don't need to listen and accept a connection. We do need to do some initialization for the sockets, but once that is done we just blast data out the socket and hope that it is received. A server can receive packets, or datagrams, from multiple clients and won't be able to tell which is which unless the payload has some identifying information.

We've already described most of these functions, except for the following differences:

socket(2):

To use UDP, instead of SOCK_STREAM use SOCK_DGRAM:
ex: s = socket(AF_INET, SOCK_DGRAM, 0);

sendto(2) : Send data, DGRAM method

```
size_t sendto(int socket, void *message, size_t message_len, int flags,
              struct sockaddr *address, size_t address_len);
```

socket = A valid socket
message = Pointer to the data you wish to send
message_len = Number of bytes to send starting at the message's location
flags = Just use 0
address = sockaddr_in structure with the port/IP address to send to
address_len = size of sock_addr_in structure

sendto returns the number of characters sent if successful, otherwise –1 if error.

recvfrom(2): Receive data, DGRAM

```
size_t recvfrom(int socket, void *buffer, size_t buffer_len, int flags,
                struct sockaddr *address, size_t *address_len);
```

socket = A valid socket
buffer = Pointer to the buffer to store your data
buffer_len = Number of bytes your buffer can hold
flags = Just use 0 unless you want to peek at message data
address = sockaddr_in structure filled in with the port/IP address received from
address_len = pointer to size of sock_addr_in structure **using value result**

This function returns the number of bytes received, or –1 if there was an error.

These functions are typically used in a loop of some kind:

```
for (;;) {
        sendto(…);
        recvfrom(…);
}
```

For some sample UDP programs, see the class website and look at "Sample Sockets Programs in C" and then look in the "sockets_examples_src" directory.

UDP vs. TCP

When might you want to use UDP instead of TCP? In general, UDP is better for short messages where you don't need error checking. Here are some comparisons:

TCP:

- Stream
- Reliable
- Point-to-Point
- Connect/Accept give us addresses, don't need addresses in read/write (or send/recv)
- Checksum on data

UDP:

- Discrete packets
- Unreliable
- Can broadcast or multicast
- One server can receive from multiple clients
- Need addresses in recvfrom and sendto
- May or may not have a checksum on data

Typically in a TCP server, we must fork() a process and have one server per connection. The master process accepts new connections, and the slave process handles the communications tasks.

```
TCP Server:
    socket()
    bind()
    listen()
    loop
        accept()
        if (fork()==0)
            read/write
        close()
    close()
    waitpid()
```

An example of this type of master server is the **inetd** daemon. inetd listens on all the "well-known" ports and spawns processes that do all of the work. See /etc/inetd.conf for those programs that get executed.

Blocking

Many of the calls we have described so far will block; e.g. accept, recv, etc. Sometimes this is undesirable, because you might want your programs to do other things in the interim. The way around this problem is to use select, which can poll a set of file descriptors for you and see if they are available or have data waiting to be read.

Select

The select call handles synchronous I/O multiplexing and uses the clock (which is why you need to include sys/time.h and also unistd.h). Here is the definition:

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

nfds = This value should be 1 + the socket (ignored by some implementations)
readfds = A pointer to a set of file and socket descriptors that are to be polled for non-blocking reading and writing operations.
writefds = Same as above, but for non-blocking writing detection. Usually NULL.
exceptfds = Same as above, but for non-blocking errors. Usually NULL.
timeout = Pointer to a timeval struct that specifies how long the select call should wait until it polls the descriptors for activity. If the timeout value is 0, then select returns immediately and if the timeout value is NULL then select will block until at least one descriptor is ready for I/O.

The function returns the number of open handles ready for I/O, or 0 if the timeout.

To manipulate the set of file descriptors, there are four macros available:

FD_CLR(fd, *set)	– Removes fd from the set
FD_ISSET(fd, *set)	– Zero if fd is not in the set, nonzero if it is in the set.
FD_SET(fd, *set)	– Adds fd to the set
FD_ZERO(*set)	– Initialize set to empty

The following snippet illustrates the usage:

```
fd_set fds;
struct timeval tv;

tv.tv_sec = 4;
tv.tv_usec = 300000;
// tv now represents 4.3 seconds, 4 seconds and 300,000 microseconds

FD_ZERO(&fds);
FD_SET(sock, &fds);          // adds some valid socket “sock” to the file descriptor set
FD_SET(STDIN, &fds);         // adds STDIN to the set

// Wait 4.3 seconds for any data to be available on the socket or STDIN
select(sock+1, &fds, NULL, NULL, &tv);

if (FD_ISSET(sock, &fds))
        recvfrom(s, buffer, buffer_len, 0, &sa, &sa_len);
else if (FD_ISSET(STDIN, &fds))
        … someone pressed a key
etc.
```