Introduction to Sockets Programming

Introduction

A *socket* is the mechanism that most popular operating systems provide to give programs access to the network. It allows messages to be sent and received between applications (unrelated processes) on different networked machines.

The sockets mechanism has been created to be independent of any specific type of network. IP, however, is by far the most dominant network and the most popular use of sockets. This tutorial provides an introduction to using sockets over the IP network (IPv4).

This tutorial will not try to cover the entire topic of sockets. There are tutorials on the web that delve into far greater detail. On-line manual pages will provide you with the latest information on acceptable parameters and functions. The interface described here is the system call interface provided by the OS X, Linux, and Solaris operating systems and is generally similar amongst all Unix/POSIX systems (as well as many other operating systems).

Programming with TCP/IP sockets

There are a few steps involved in using sockets:

1. Create the socket
2. Identify the socket
3. On the server, wait for an incoming connection
4. On the client, connect to the server's socket
5. Send and receive messages
6. Close the socket

Step 1. Create a socket

A socket, `s`, is created with the *socket* system call:

```
int s = socket(domain, type, protocol)
```

All the parameters as well as the return value are integers:

domain, or address family —

communication domain in which the socket should be created. Some of address families are AF_INET (IP), AF_INET6 (IPv6), AF_UNIX (local channel, similar to pipes), AF_ISO (ISO protocols), and AF_NS (Xerox Network Systems protocols).

type —

type of service. This is selected according to the properties required by the application: SOCK_STREAM (virtual circuit service), SOCK_DGRAM (datagram service), SOCK_RAW (direct IP service). Check with your address family to see whether a particular service is available.

protocol —

indicate a specific protocol to use in supporting the sockets operation. This is useful in cases where some families may have more than one protocol to support a given type of service. The return value is a file descriptor (a small integer). The analogy of creating a socket is that of requesting a telephone line from the phone company.

For TCP/IP sockets, we want to specify the IP address family (AF_INET) and virtual circuit service (SOCK_STREAM). Since there's only one form of virtual circuit service, there are no variations of the protocol, so the last argument, *protocol*, is zero. Our code for creating a TCP socket looks like this:

```
#include <sys/socket.h>

...

if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("cannot create socket");
        return 0;
}
```

⇒ Download a demo file

Step 2. Indentify (name) a socket

When we talk about *naming* a socket, we are talking about assigning a transport address to the socket (a port number in IP networking). In sockets, this operation is called binding an address and the *bind* system call is used for this. The analogy is that of assigning a phone number to the line that you requested from the phone company in step 1 or that of assigning an address to a mailbox.

The transport address is defined in a socket address structure. Because sockets were designed to work with various different types of communication interfaces, the interface is very general. Instead of accepting, say, a port number as a parameter, it takes a `sockaddr` structure whose actual format is determined on the address family (type of network) you're using. For example, if you're using UNIX domain sockets, *bind* actually creates a file in the file system.

The system call for *bind* is:

```
#include <sys/socket.h>

int
bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

The first parameter, `socket`, is the socket that was created with the *socket* system call.

For the second parameter, the structure `sockaddr` is a generic container that just allows the OS to be able to read the first couple of bytes that identify the address family. The address family determines what variant of the `sockaddr` struct to use that contains elements that make sense for that specific communication type. For IP networking, we use `struct sockaddr_in`, which is defined in the header `netinet/in.h`. This structure defines:

```
struct sockaddr_in {
        __uint8_t       sin_len;
        sa_family_t     sin_family;
        in_port_t       sin_port;
        struct  in_addr sin_addr;
        char            sin_zero[8];
};
```

Before calling *bind*, we need to fill out this structure. The three key parts we need to set are:

sin_family

The address family we used when we set up the socket. In our case, it's AF_INET.

sin_port

The port number (the transport address). You can explicitly assign a transport address (port) or allow the operating system to assign one. If you're a client and won't be receiving incoming connections, you'll usually just let the operating system pick any available port number by specifying port 0. If you're a server, you'll generally pick a specific number since clients will need to know a port number to connect to.

sin_addr

The address for this socket. This is just your machine's IP address. With IP, your machine will have one IP address for each network interface. For example, if your machine has both Wi-Fi and ethernet connections, that machine will have two addresses, one for each interface. Most of the time, we don't care to specify a specific interface and can let the operating system use whatever it wants. The special address for this is 0.0.0.0, defined by the symbolic constant `INADDR_ANY`.

Since the address structure may differ based on the type of transport used, the third parameter specifies the length of that structure. This is simply `sizeof(struct sockaddr_in)`.

The code to bind a socket looks like this:

```
#include <sys/socket.h>

...

struct sockaddr_in myaddr;

/* bind to an arbitrary return address */
/* because this is the client side, we don't care about the address */
/* since no application will connect here: */
/* INADDR_ANY is the IP address and 0 is the socket */
/* htonl converts a long integer (e.g. address) to a network representation */
/* htons converts a short integer (e.g. port) to a network representation */

memset((char *)&myaddr, 0, sizeof(myaddr));
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr = htonl(INADDR_ANY);
myaddr.sin_port = htons(0);

if (bind(fd, (struct sockaddr *)&myaddr, sizeof(myaddr)) < 0) {
        perror("bind failed");
        return 0;
}
```

⇒ Download a demo file

Note: number conversions (*htonl, htons, ntohl, ntohs****)***

You might have noticed the *htonl* and *htons* references in the previous code block. These convert four-byte and two-byte numbers into network representations. Integers are stored in memory and sent across the network as sequences of bytes. There are two common ways of storing these bytes: *big endian* and *little endian* notation. Little endian representation stores the least-significant bytes in low memory. Big endian representation stores the least-significant bytes in high memory. The Intel x86 family uses the little endian format. Old Motorola processors and the PowerPC (used by Macs before their switch to the Intel architecture) use the big endian format.

Internet headers standardized on using the big endian format. If you're on an Intel processor and set the value of a port to 1,234 (hex equivalent `04d2`), it will be be stored in memory as `d204`. If it's stored in this order in a TCP/IP header, however, `d2` will be treated as the most significant byte and the network protocols would read this value as 53,764.

To keep code portable – and to keep you from having to write code to swap bytes and worry about this &ndash a few convenience macros have been defined:

htons

host to network short : convert a number into a 16-bit network representation. This is commonly used to store a port number into a `sockaddr` structure.

htonl

host to network long : convert a number into a 32-bit network representation. This is commonly used to store an IP address into a `sockaddr` structure.

ntohs

network to host short : convert a 16-bit number from a network representation into the local processor's format. This is commonly used to read a port number from a `sockaddr` structure.

ntohl

network to host long : convert a 32-bit number from a network representation into the local processor's format. This is commonly used to read an IP address from a `sockaddr` structure.

For processors that use the big endian format, these macros do absolutely nothing. For those that use the little endian format (most processors, these days), the macros flip the sequence of either four or two bytes. In the above code, writing `htonl(INADDR_ANY)` and `htons(0)` is somewhat pointless since all the bytes are zero anyway but it's good practice to remember to do this at all times when reading or writing network data.

Step 3a. Connect to a server from a client

If we're a client process, we need to establish a connection to the server. Now that we have a socket that knows where it's coming *from*, we need to tell it where it's going *to*. The *connect* system call accomplishes this.

```
#include <sys/types.h>
#include <sys/socket.h>

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

The first parameter, `socket`, is the socket that was created with the *socket* system call and named via *bind*. The second parameter identifies the *remote*transport address using the same `sockaddr_in` structure that we used in *bind* to identify our local address. As with *bind*, the third parameter is simply the length of the structure in the second parameter: `sizeof(struct sockaddr_in)`.

The server's address will contain the IP address of the server machine as well as the port number that corresponds to a socket listening on that port on that machine. The IP address is a four-byte (32 bit) value in network byte order (see *htonl* above).

In most cases, you'll know the name of the machine but not its IP address. An easy way of getting the IP address is with the *gethostbyname* library (libc) function. *Gethostbyname* accepts a host name as a parameter and returns a `hostent` structure:

```
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 */
};
```

If all goes well, the `h_addr_list` will contain a list of IP addresses. There may be more than one IP addresses for a host. In practice, you should be able to use any of the addresses or you may want to pick one that matches a particular subnet. You may want to check that `(h_addrtype == AF_INET)` and `(h_length == 4)` to ensure that you have a 32-bit IPv4 address. We'll be lazy here and just use the first address in the list.

For example, suppose you want to find the addresses for google.com. The code will look like this:

```
#include <stdlib.h>
#include <stdio.h>
#include <netdb.h>

/* paddr: print the IP address in a standard decimal dotted format */
void
paddr(unsigned char *a)
{
        printf("%d.%d.%d.%d\n", a[0], a[1], a[2], a[3]);
}

main(int argc, char **argv) {
        struct hostent *hp;
        char *host = "google.com";
        int i;

        hp = gethostbyname(host);
        if (!hp) {
                fprintf(stderr, "could not obtain address of %s\n", host);
                return 0;
        }
        for (i=0; hp->h_addr_list[i] != 0; i++)
                paddr((unsigned char*) hp->h_addr_list[i]);
        exit(0);
}
```

Here's the code for establishing a connection to the address of a machine in `host`. The variable `fd` is the socket which was created with the *socket* system call.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>      /* for fprintf */
#include <string.h>     /* for memcpy */

struct hostent *hp;     /* host information */
struct sockaddr_in servaddr;    /* server address */

/* fill in the server's address and data */
memset((char*)&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(port);

/* look up the address of the server given its name */
hp = gethostbyname(host);
if (!hp) {
        fprintf(stderr, "could not obtain address of %s\n", host);
        return 0;
}

/* put the host's address into the server address structure */
memcpy((void *)&servaddr.sin_addr, hp->h_addr_list[0], hp->h_length);
```

```
/* connect to server */
if (connect(fd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
        perror("connect failed");
        return 0;
}
```

Step 3b. Accept connections on the server

Before a client can connect to a server, the server should have a socket that is prepared to accept the connections. The *listen* system call tells a socket that it should be capable of accepting incoming connections:

```
#include <sys/socket.h>

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

The second parameter, *backlog*, defines the maximum number of pending connections that can be queued up before connections are refused.

The *accept* system call grabs the first connection request on the queue of pending connections (set up in *listen*) and creates a new socket for that connection. The original socket that was set up for listening is used *only* for accepting connections, not for exchanging data. By default, socket operations are synchronous, or blocking, and accept will block until a connection is present on the queue. The syntax of accept is:

```
#include <sys/socket.h>

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

The first parameter, *socket*, is the socket that was set for accepting connections with *listen*. The second parameter, *address*, is the address structure that gets filed in with the address of the client that is doing the *connect*. This allows us to examine the address and port number of the connecting socket if we want to. The third parameter is filled in with the length of the address structure.

Let's examine a simple server. We'll create a socket, bind it to any available IP address on the machine but to a specific port number. Then we'll set the socket up for listening and loop, accepting connections.

```
#include <sys/types.h>

...
{
        int port = 1234;        /* port number */
        int rqst;       /* socket accepting the request */
        socklen_t alen;       /* length of address structure */
        struct sockaddr_in my_addr;    /* address of this service */
        struct sockaddr_in client_addr;  /* client's address */
        int sockoptval = 1;


        /* create a TCP/IP socket */
        if ((svc = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                perror("cannot create socket");
                exit(1);
        }

        /* allow immediate reuse of the port */
```

```
        setsockopt(svc, SOL_SOCKET, SO_REUSEADDR, &sockoptval, sizeof(int));

        /* bind the socket to our source address */
        memset((char*)&my_addr, 0, sizeof(my_addr));  /* 0 out the structure */
        my_addr.sin_family = AF_INET;   /* address family */
        my_addr.sin_port = htons(port);
        my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(svc, (struct sockaddr *)&my_addr, sizeof(my_addr)) < 0) {
                perror("bind failed");
                exit(1);
        }

        /* set the socket for listening (queue backlog of 5) */
        if (listen(svc, 5) < 0) {
                perror("listen failed");
                exit(1);
        }

        /* loop, accepting connection requests */
        for (;;) {
                while ((rqst = accept(svc, (struct sockaddr *)&client_addr, &alen)) < 0) {
                        /* we may break out of accept if the system call */
                        /* was interrupted. In this case, loop back and */
                        /* try again */
                        if ((errno != ECHILD) && (errno != ERESTART) && (errno != EINTR)) {
                                perror("accept failed");
                                exit(1);
                        }
                }
                /* the socket for this accepted connection is rqst */
                ...
        }
}
```

⇒ Download a demo file

This code contains two special operations, marked in bold:

1. `setsockopt(svc, SOL_SOCKET, SO_REUSEADDR, &sockoptval, sizeof(int))`

   The *setsockopt* system call allows us to set special options on a socket. In this case, we use *setsockopt* to set `SO_REUSEADDR`. This allows us to reuse the port immediately as soon as the service exits. Some operating systems will not allow immediate reuse on the chance that some packets may still be en route to the port. Allowing us to reuse the port immediately is a huge help when debugging requires us to start and restart the server over and over again.

2. `if ((errno != ECHILD) && (errno != ERESTART) && (errno != EINTR))`

   accept is a blocking system call that returns when an incoming connection has been received. However, on some operating systems it may return if other signals have been received, such as a death of a child process (`ECHILD`), an interrupted system call that should be restarted (`ERESTART`), or just an interrupted system call (`EINTR`). If any of these signals have been received then we loop back and continue waiting for a connection. Otherwise we print an error message and exit.

Step 4. Communicate

We *finally* have connected sockets between a client and a server! Communication is the easy part. The same *read* and *write* system calls that work on files also work on sockets. We can send 20 bytes from the client to server with:

```
char buffer[MAXBUF];
...
nbytes = write(fd, buffer, 20); /* write 20 bytes in buffer */
```

We can read a bunch of data from the client with:

```
char buffer[MAXBUF];
...
nbytes = read(fd, buffer, MAXBUF); /* read up to MAXBUF bytes */
```

One important thing to keep in mind is that TCP/IP sockets give you a byte stream, not a packet stream. If you write 20 bytes to a socket, the other side is not guaranteed to read 20 bytes in a *read* operation. It may read 20. It may read less if there was some packet fragmentation. In that case, you'll need to loop back and keep reading. If you write 20 bytes to a socket and then write another 20 bytes to a socket, the other side may read all 40 bytes at once ... or not.

If the amount of data you want to read matters then it's up to you to create a protocol that lets you know what to read, such as sending a length count prior to sending data.

A few additional system calls were added. We can ignore them most of the time but should know they exist for the times we may need them. All of these functions support an extra *flags* parameter. The actual capabilities may differ among different operating systems, so check the documentation on a specific system. Some features that are supported are quality of service control, out-of-band data transmission, the ability to peek before reading incoming data abd and the ability to bypass routing (mostly for debugging).

The *send* and *recv* calls are similar to *read* and *write* with the addition of the *flags* parameter.

The *sendto* and *recvfrom* system calls are like *send* and *recv* but also allow callers to specify or receive addresses of the peer with whom they are communicating. This is most useful for connectionless sockets.

Finally, *sendmsg* and *recvmsg* let one use a `msghdr` structure to minimize the number of directly supplied arguments and also minimize the numner of system calls by using scatter/gather to read/write from/to a number of distinct memory blocks.

Could this have been designed cleaner and simpler? Probably. The *read*/*write* or *send*/*recv* calls are designed be used primarily for connection-oriented communication while sendto/recvfrom or sendmsg/recvmsg are normally used for connectionless communication since they allow one to specify the address.

Step 5. Close the connection

When we're done communicating, the easiest thing to do is to close a socket with the *close* system call — the same *close* that is used for files.

There's another way to close connections: the *shutdonw* system call.

```
#include <sys/socket.h>

int
shutdown(int socket, int how);
```

The second parameter, *how*, allows us to tell the socket what part of the full-duplex connection to shut down:

- A value of `SHUT_RD` will disallow further receives on that socket.
- A value of `SHUT_WR` will disallow further sends on that socket.
- A value of `SHUT_RDWR` will disallow both further sends and receives on that socket.

Closing a socket with `close(s)` is identical to using `shutdown(s, SHUT_RDWR)`.

Synchronous or Asynchronous?

Network communication (or file system access in general) system calls may operate in two modes: synchronous or asynchronous. In the synchronous mode, socket routines return only when the operation is complete. For example,

accept returns only when a connection arrives. In the asynchronous mode, socket routines return immediately: system calls become non-blocking calls (e.g., *read* does not block, waiting until data arrives). You can change the mode with the fcntl system call. For example,

```
fcntl(s, F_SETFF, FNDELAY);
```

sets the socket `s` to operate in asynchronous mode.
