Distributed Systems

Programming with UDP sockets

Introduction

In our discussion of sockets, we covered an example of programming with connection-oriented sockets: sockets that use the TCP/IP protocol. Here, we'll briefly look at an example using connectionless sockets over UDP/IP.

This tutorial provides an introduction to using UDP sockets over the IP network (IPv4).

As with TCP sockets, this tutorial will focus on the basics. 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 UDP/IP sockets

There are a few steps involved in using sockets:

1. Create the socket
2. Identify the socket (name it)
3. On the server, wait for a message
4. On the client, send a message
5. Send a response back to the client (optional)
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 UDP/IP sockets, we want to specify the IP address family (AF_INET) and datagram service (SOCK_DGRAM). Since there's only one form of datagram service, there are no variations of the protocol, so the last argument, *protocol*, is zero. Our code for creating a UDP socket looks like this:

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

...

if ((fd = socket(AF_INET, SOCK_DGRAM, 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 to do 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 don't need a well-known port that others can use to locate you (since they will only respond to your messages), you can 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 which to address messages.

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 the size of the internet address structure, `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 initiate communication here - it will */
/* just send responses */
/* 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

This example asks the operating system to pick any available port number by setting the port to 0. If you want to use a specific port number, change the line

```
myaddr.sin_port = htons(0);
```

to

```
myaddr.sin_port = htons(myport)
```

where `myport` is the variable (short int) that contains your port number.

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. Using the macros, however, ensures that your code remains portable regardless of the architecture to which you compile. 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. Send a message to a server from a client

With TCP sockets, we had to establish a connection before we could communicate. With UDP, our sockets are connectionless. Hence, we can send messages immediately. Since we do not have a connection, the messages have to be addressed to their destination. Instead of using a *write* system call, we will use *sendto*, which allows us to specify the destination. The address is identified through the `sockaddr` structure, the same way as it is in *bind* and as we did when using *connect* for TCP sockets.

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

int
sendto(int socket, const void *buffer, size_t length, int flags, const struct sockaddr *dest_addr,
         socklen_t dest_len)
```

The first parameter, `socket`, is the socket that was created with the *socket* system call and named via *bind*. The second parameter, `buffer`, provides the starting address of the message we want to send. `length` is the number of bytes that we want to send. The `flags` parameter is 0 and not useful for UDP sockets. The `dest_addr` defines the destination address and port number for the message. It uses the same `sockaddr_in` structure that we used in *bind* to identify our local address. As with *bind*, the final parameter is simply the length of the address structure: `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 sending a message 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 */
char *my_messsage = "this is a test message";
```

```
/* 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);

/* send a message to the server */
if (sendto(fd, my_message, strlen(my_message), 0, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
        perror("sendto failed");
        return 0;
}
```

Step 3b. Receive messages on the server

With TCP sockets, a server would set up a socket for listening via a *listen* system call and then call *accept* to wait for a connection. UDP is connectionless. A server can immediately listen for messages once it has a socket. We use the *recvfrom* system call to wait for an incoming datagram on a specific transport address (IP address and port number).

The *recvfrom* call has the following syntax:

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

int
recvfrom(int socket, void *restrict buffer, size_t length, int flags, struct sockaddr *restrict src_addr,
         socklen_t *restrict *src_len)
```

The first parameter, `socket` is a socket that we created ahead of time (and used *bind*. The port number assigned to that socket via the *bind* call tells us on what port *recvfrom* will wait for data. The incoming data will be placed into the memory at `buffer` and no more than `length` bytes will be transferred (that's the size of your buffer). We will ignore `flags` here. You can look at the man page for *recvfrom* for details on this. This parameter allows us to process out-of-band data, peek at an incoming message without removing it from the queue, or block until the request is fully satisfied. We can safely ignore these and use 0. The `src_addr` parameter is a pointer to a `sockaddr` structure that you allocate and will be filled in by *recvfrom* to identify the sender of the message. The length of this structure will be stored in `src_len`. If you do not care to identify the sender, you can set both of these to zero but you will then have no way to reply to the sender.

The *recvfrom* call returns the number of bytes that were read into `buffer`

Let's examine a simple server. We'll create a socket, bind it to all available IP addresses on the machine but to a specific port number. Then we will loop, receiving messages and printing their contents.

```

#define PORT 1153
#define BUFSIZE 2048

int
main(int argc, char **argv)
{
        struct sockaddr_in myaddr;      /* our address */
        struct sockaddr_in remaddr;     /* remote address */
        socklen_t addrlen = sizeof(remaddr);            /* length of addresses */
        int recvlen;                    /* # bytes received */
        int fd;                         /* our socket */
        unsigned char buf[BUFSIZE];     /* receive buffer */

        /* create a UDP socket */

        if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
                perror("cannot create socket\n");
                return 0;
        }

        /* bind the socket to any valid IP address and a specific port */

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

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

        /* now loop, receiving data and printing what we received */
        for (;;) {
                printf("waiting on port %d\n", PORT);
                recvlen = recvfrom(fd, buf, BUFSIZE, 0, (struct sockaddr *)&remaddr, &addrlen);
                printf("received %d bytes\n", recvlen);
                if (recvlen > 0) {
                        buf[recvlen] = 0;
                        printf("received message: \"%s\"\n", buf);
                }
        }
        /* never exits */
}
```

⇒ Download a demo file

Step 4. Bidirectional communication

We now have a client sending a message to a server. What if the server wants to send a message back to that client? There is no connection so the server cannot just write the response back. Fortunately, the *recvfrom* call gave us the address of the server. It was placed in `remaddr`:

```
recvlen = recvfrom(s, buf, BUFSIZE, 0, (struct sockaddr *)&remaddr, &addrlen);
```

The server can use that address in *sendto* and send a message back to the recipient's address.

```
sendto(s, buf, strlen(buf), 0, (struct sockaddr *)&remaddr, addrlen)
```

⇒ Download a demo file

Step 5. Close the socket

With TCP sockets, we saw that we can use the *shutdown* system call to close a socket or to terminate communication in a single direction. Since there is no concept of a connection in UDP, there is no need to call *shutdown*. However, the socket still uses up a file descriptor in the kernel, so we can free that up with the *close* system call just as we do with files.

```
        close(fd);
```