UDP made simple

Abstract: This page describes how to write a simple UDP client/server system in a C/Unix environment. The code is explained step by step.

Motivation: I needed a page like this when working with a small test program for my master's thesis at Appius / Fält Communications. It is quite hard to remember all the socket API details off the top of your head, so I wanted a small reference page to get me started quickly without having to wade through tons of `man` pages. As I did not find a page I liked, I decided to write one. I hope it will be useful for others, too.

Caveats: The code is known to work under recent (fall 1999) versions of Linux. It should work on other Unices too, though some of the header files may be located in other directories on your system.

The server

```
 1  #include <arpa/inet.h>
 2  #include <netinet/in.h>
 3  #include <stdio.h>
 4  #include <sys/types.h>
 5  #include <sys/socket.h>
 6  #include <unistd.h>
 7
 8  #define BUFLEN 512
 9  #define NPACK 10
10  #define PORT 9930
11
12  void diep(char *s)
13  {
14    perror(s);
15    exit(1);
16  }
17
18  int main(void)
19  {
20    struct sockaddr_in si_me, si_other;
21    int s, i, slen=sizeof(si_other);
22    char buf[BUFLEN];
23
24    if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
25      diep("socket");
26
27    memset((char *) &si_me, 0, sizeof(si_me));
28    si_me.sin_family = AF_INET;
29    si_me.sin_port = htons(PORT);
30    si_me.sin_addr.s_addr = htonl(INADDR_ANY);
31    if (bind(s, &si_me, sizeof(si_me))==-1)
32        diep("bind");
33
34    for (i=0; i<NPACK; i++) {
35      if (recvfrom(s, buf, BUFLEN, 0, &si_other, &slen)==-1)
36        diep("recvfrom()");
37      printf("Received packet from %s:%d\nData: %s\n\n",
38             inet_ntoa(si_other.sin_addr), ntohs(si_other.sin_port), buf);
```

```
39    }
40
41    close(s);
42    return 0;
43 }
```

Comments

- Lines 8-10 define the buffer size (quite arbitrary), the number of packets to receive and the UDP port number to listen at. You could use any port number above 1023, although `bind()` will fail if someone else is using the same port simultaneously.
- The function `diep()` is used for error handling.
- 21: Declare receive buffer.
- 22: `sockaddr_in` is a structure containing an Internet socket address. Basically, it contains:
  - an address family (always AF_INET for our purposes)
  - a port number
  - an IP address

  `si_me` defines the socket where the server will listen. `si_other` defines the socket at the other end of the link (that is, the client).
- 24: Create a socket. `AF_INET` says that it will be an Internet socket. `SOCK_DGRAM` says that it will use datagram delivery instead of virtual circuits. `IPPROTO_UDP` says that it will use the UDP protocol (the standard transport layer protocol for datagrams in IP networks). Generally you can use zero for the last parameter; the kernel will figure out what protocol to use (in this case, it would choose IPPROTO_UDP anyway).
- 27: We need to initialize the `si_me` structure. The first step is to fill it with binary zeroes, which is done on this line. (I doubt this step is actually necessary in modern Unix implementations, but better safe than sorry.)
- 28: We will use Internet addresses.
- 29: Here, the port number is defined. `htons()` ensures that the byte order is correct (Host TO Network order/Short integer).
- 30: This line is used to tell what IP address we want to bind to. Most machines have more than one network interface (for example, 127.0.0.1 for the loopback interface and some other address for the network card; there may be more than one network card). In the general case, you want to accept packets from any interface, so you use INADDR_ANY instead of a specific address.
- 31: Now we are ready to bind the socket to the address we created above. This line tells the system that the socket `s` should be bound to the address in `si_me`.
- 35: This call says that we want to receive a packet from `s`, that the data should be put info `buf`, and that `buf` can store at most `BUFLEN` characters. The zero parameter says that no special flags should be used. Data about the sender should be stored in `si_other`, which has room for `slen` byte. Note that `recvfrom()` will set `slen` to the number of bytes actually stored. If you want to play safe, set `slen` to `sizeof(si_other)` after each call to `recvfrom()`.
- 37: The information about the sender we got from `recvfrom()` is displayed (IP:port), along with the data in the packet. `inet_ntoa()` takes a `struct in_addr` and converts it to a string in dot notation, which is rather useful if you want to display the address in a legible form.

The client

```
 1  #define SRV_IP "999.999.999.999"
 2  /* diep(), #includes and #defines like in the server */
 3
 4  int main(void)
 5  {
 6    struct sockaddr_in si_other;
 7    int s, i, slen=sizeof(si_other);
 8    char buf[BUFLEN];
 9
10    if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
11      diep("socket");
12
13    memset((char *) &si_other, 0, sizeof(si_other));
14    si_other.sin_family = AF_INET;
15    si_other.sin_port = htons(PORT);
16    if (inet_aton(SRV_IP, &si_other.sin_addr)==0) {
17      fprintf(stderr, "inet_aton() failed\n");
18      exit(1);
19    }
20
21    for (i=0; i<NPACK; i++) {
22      printf("Sending packet %d\n", i);
23      sprintf(buf, "This is packet %d\n", i);
24      if (sendto(s, buf, BUFLEN, 0, &si_other, slen)==-1)
25        diep("sendto()");
26    }
27
28    close(s);
29    return 0;
30  }
```

Note: The client is quite similar to the server. Only the differences will be discussed.

- 1: You need to know the IP address of the machine where the server runs. If you run the client and the server on the same machine, try 127.0.0.1. "999.999.999.999" is not a legal IP address; you need to substitute your own server's address.
- 12: You may call `bind()` after the call to `socket()`, if you wish to specify which port and interface that should be used for the client socket. However, this is almost never necessary. The system will decide what port and interface to use.
- 13-19: Here, we set up a `sockaddr_in` corresponding to the socket address where the server is listening. `inet_aton()` is used to convert a string in dotted-decimal ASCII notation to a binary address.
- 24: Send `BUFLEN` bytes from `buf` to `s`, with no flags (`0`). The receiver is specified in `si_other`, which contains `slen` byte.

General tips

- Remember to always check return values from system calls! By doing so, you *will* save time in the long run, I promise. Many people do not test return values in small quick-and-dirty test programs. However, in such cases it is *especially* important to check return values, because if you don't really know what you are doing you are much more likely to make a mistake. The checks help you understand what went wrong and why.

- There is a tool called `netcat` (the actual command is `nc`) which is very useful for testing and debugging socket code. Check the `man` page if you are curious (of course, it might not be installed on your system).
- If you want to cut and paste the code above, use `cut -c9-` to get rid of the line numbers. (The exact syntax of `cut` may be different on your system, and you may have to remove more or less than 9 characters.)
- The command `netstat` can be useful to check which sockets are active. Try `netstat -a`.
- For an overview over some of the structures used in socket programming, check out the code examples from lecture 13 on my course in Unix system programming. You will also find some material on TCP programming there. Disregard the initial material on `select()` and friends. There are some comments in Swedish, but most of the page is written in C.

Last modified 07-10-12 by Gunnar Gunnarsson. Comments, bug fixes, and suggestions welcome.