A veteran Unix and Linux enthusiast, Chris Brown has written and delivered open source training from New Delhi to San Francisco, though not on the same day.

CORE TECHNOLOGY

Dive under the skin of your Linux system to find out what really makes it tick.

Sockets, UDP and TCP

This month, we'll get plugged in with sockets, which span the world.

The term "sockets" refers to an API (a set of library routines) that enables us to write clients and servers that communicate with the TCP and UDP protocols. A socket is sometimes described as a communication endpoint; it's where the transport provider (the TCP or UDP layer) and the application programs meet.

Let's talk first about the underlying IP layer. IP (Internet Protocol) has the important job of delivering a datagram to the correct recipient machine (based on its IP address). The IP layer handles all the routing decisions. However, IP does not offer guaranteed delivery. If a packet goes missing, it just goes missing; there's no mechanism to automatically re-send it. Also, there's no guarantee that the packets will arrive in the same order they were sent. Successive datagrams sent over a wide-area network may get routed differently and arrive in the wrong order.

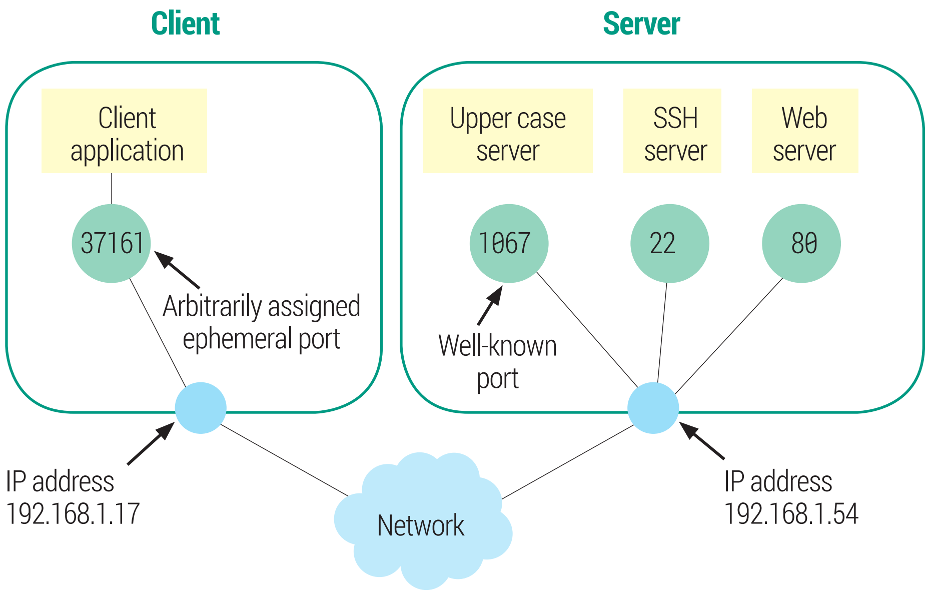
The TCP and UDP protocols lie on top of IP and are of more direct interest to us if we're writing socket-based programs.

UDP

UDP stands for User Datagram Protocol and is the simpler of the two. In fact it adds very little on top of IP except for the concept of a port number (a 16-bit value) that enables datagrams to be sent to a specific endpoint (effectively, to a specific application or service) on the destination machine.

UDP sockets provide a "connectionless" communication model, and sometimes they're compared to the postal service. Suppose I'm in regular postal contact with Barney Rubble at Bedrock. Each and every letter I send to him has to have Barney's address on the envelope; based on that information alone the letter is routed through the postal system and delivered. I don't get notification of delivery; I don't even get notification of non-delivery.

Every datagram sent by a program has to have a destination address (an IP address and port number) specified. When a program transmits a UDP datagram there is no guarantee that it will arrive and no notification if it doesn't. UDP simply inherits the Internet Protocol's failure of not guaranteeing to deliver the datagrams or get them in the right order. Of course, if the communication is taking place within a local network it's difficult to see how datagrams might become mis-ordered; nonetheless, UDP makes no guarantees about delivery order. It is sometimes known (unfairly) as the "Unreliable Datagram Protocol".



Clients use arbitrary "ephemeral" ports, server bind "well-known" ports. The tuple {client port, client IP, server port, server IP} identifies a TCP connection

Information exchange

For many UDP-based services the lack of a "first sent, first received" guarantee isn't an issue. Take DNS for example: it listens on UDP port 53, receives a lookup request, and sends a reply. Next request, next reply. There is never a sequence of datagrams that form part of the same interaction.

Other UDP-based services handle things differently. For example, the Trivial File Transfer Protocol (TFTP) (which does use a sequence of datagrams that form part of the same interaction), solves the problem by having every datagram explicitly acknowledged within the application layer, so that the whole thing stays in lock-step.

Now let's look at TCP (Transmission Control Protocol). Like UDP, it's also layered

Try It Out – Build and test the upper-case server

You can easily build this server. Either type the code in, or grab it from our magazine landing page on **www.linuxvoice.com**.

Put it into a file called **ucserver.c**. Next, ensure you've got the **gcc** compiler installed. On Ubuntu, for example it's installed simply as:

```
$ sudo apt-get install gcc
```

Now build your executable:

```
$ gcc ucserver.c -o ucserver
```

Assuming it compiles without errors, run it in the foreground:

```
./ucserver
```

Now open a new terminal window and verify that the server is listening on the expected port:

```
$ lsof -i TCP:1067
COMMAND   PID  USER   FD   TYPE DEVICE SIZE/OFF
NODE NAME
ucserver 12575 chris   3u  IPv4 272606      0t0  TCP
*:1067 (LISTEN)
```

We can use **netcat** (also know as **nc**) as a handy client program to test our server by telling it to connect to port 1067. All nc is doing is connecting to the port and ferrying lines of text back and forth, like this:

```
$ nc localhost 1067
Hello World
HELLO WORLD
This is a test
THIS IS A TEST
^D
```

Success! Not the world's most exciting service I grant you. But what do you expect for 40 lines of code? You can also run the test "non-interactively" by piping into **nc** like this:

```
$ echo "Hello World" | nc localhost 1067
HELLO WORLD
```

over IP, and like UDP it adds the notion of port numbers to identify a specific endpoint. But TCP is considerably more complicated, offering a reliable, sequenced, connection-oriented service.

Sometimes people refer to TCP as a "guaranteed" delivery service, but we should be clear what is meant by that. If you unplug the network cable from your server, TCP won't guarantee to deliver packets to it, nor can you write in and ask for your money back when it fails. The reliability of a TCP connection results from a process of "positive acknowledgement with re-transmission" – essentially, senders expect a confirmation of receipt of the segments of data they transmit, and will re-send them if they don't get one.

The main thing we need to understand is the connection-oriented nature of the service, and a common analogy is with a telephone call. Edward Bear wants to call Maisie Bear to invite her to a picnic. He provides addressing information "up front" when he dials Maisie's phone number (analogous to specifying an IP address and a port number when establishing a TCP connection) and Maisie needs to answer the phone (accept the connection) but from then on they have the illusion of a piece of copper wire connecting their phones. In my analogy, the client and server have a file descriptor referencing the connection. Edward and Maisie simply speak into the phone. Maisie doesn't need to repeatedly tell the telephone company "please deliver this sentence to Edward Bear", there is a connection and they simply speak on the phone. In my analogy, the client and server simply read and write on their file descriptor.

Design choices

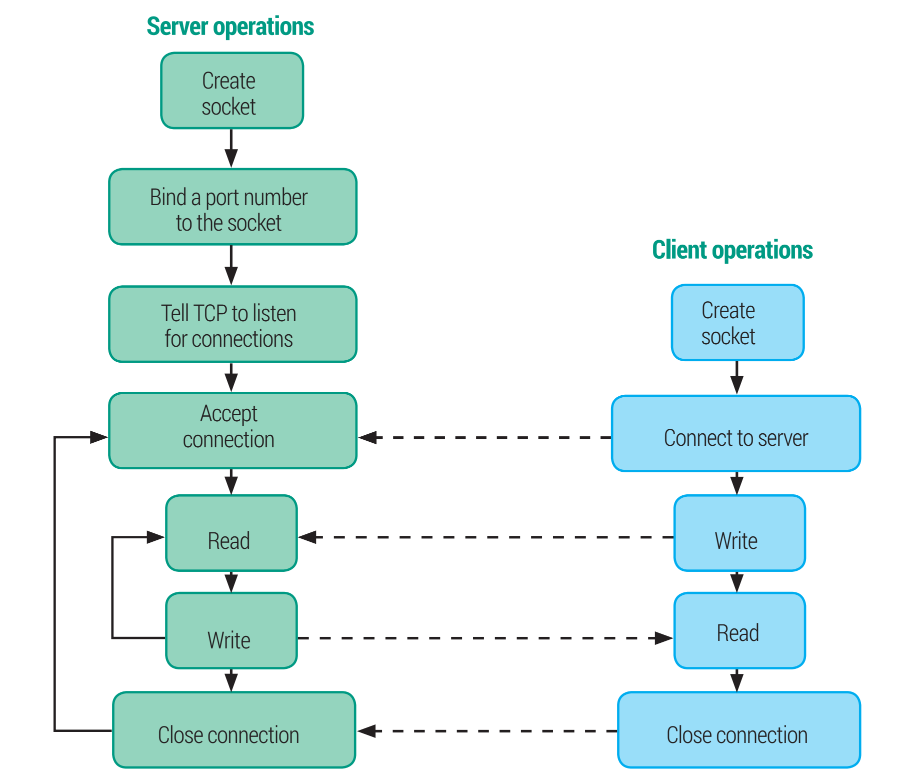
The choice of UDP or TCP protocols has a fundamental effect on the way in which services are written, particularly if they want to be able to service multiple clients simultaneously. A UDP-based service such as TFTP is basically holding out a bucket labelled "Port 69". Any client can lob a datagram into the bucket, and in general the server will find itself fishing datagrams out of the bucket from various clients in some arbitrarily interleaved order. Usually, a UDP-based server is single-threaded, with the thread looping round on the request to "get the next datagram from the bucket". TCP-based services are different; each accepted connection results in a new file descriptor. In the simplest case, a server might complete its dialog with one client before accepting a connection from the next, but this will keep subsequent clients waiting if a client has opened the connection and then gone for a cup of tea. More commonly, TCP servers use a process-per-client or thread-per-client model to achieve concurrency when serving multiple clients.

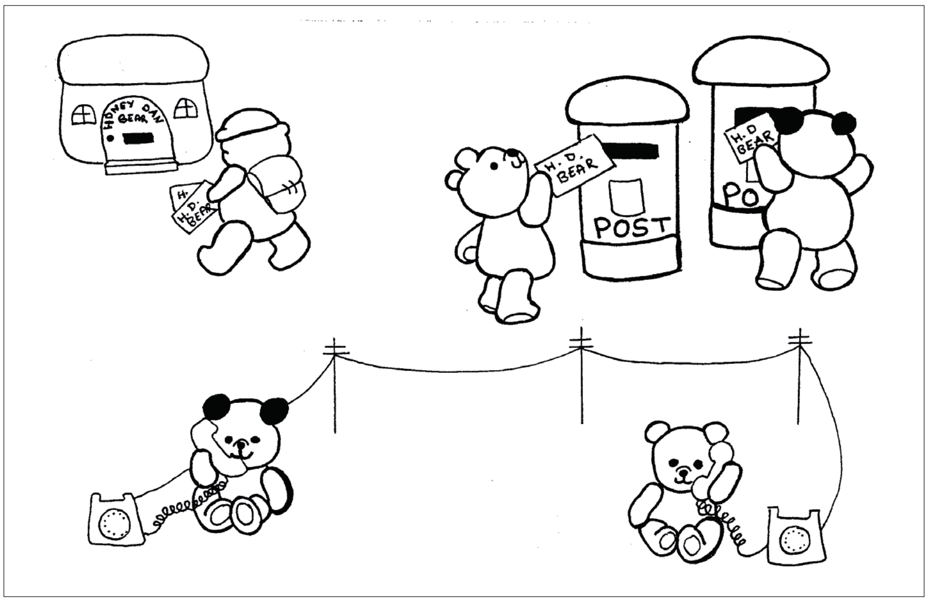
I'm going to inflict some C code on you to show how sockets really work. Our example is a simple TCP-based server; the "service" is simply to convert any text received by the client to upper case, and send it back.

Here's the code. (The lines numbers are just for reference; they aren't part of the file)

```
1  #include <stdio.h>
2  #include <netinet/in.h>
3
4  #define SERVER_PORT 1067
5  #define BUFSIZE 100
6
7  /* ------ service() ------- */
8
```

Client and server ends in a TCP connection





Even teddy bears have a choice between connectionless and connection-oriented protocols.

```
9  void service(int in, int out)
10 {
11   char buffer[BUFSIZE];
12   int i, len;
13   while ((len = read(in, buffer, BUFSIZE)) > 0)
14   {
15     for (i = 0 ; i < len; i++)
16       buffer[i] = toupper(buffer[i]);
17     write(out, buffer, len);
18   }
19 }
20
21 /* ---- Main program ---- */
22
23 void main()
24 {
25   int sock, fd, client_len;
26   struct sockaddr_in server, client;
27
28   sock = socket(AF_INET, SOCK_STREAM, 0);
29   server.sin_family = AF_INET;
30   server.sin_addr.s_addr = htonl(INADDR_ANY);
31   server.sin_port = htons(SERVER_PORT);
32   bind(sock, (struct sockaddr *)&server,
sizeof(server));
33
34   listen(sock, 5);
35   while (1) {
36     client_len = sizeof(client);
37     fd = accept(sock, (struct sockaddr *)&client,
&client_len);
38     printf("accepted connection on fd %d\n", fd);
39     service(fd, fd);
40     close(fd);
41   }
42 }
```

I'll not trouble you with the details of all the data structures; some of them (like the **sockaddr_in** structure) look more complicated than you might think they should, but that's because the API is designed to support a wide variety of protocols, not just TCP and IPV4.

The action starts at line 28, where our server creates a socket, requesting an address family of **AF_INET** (which means that the socket will be identified by an IP address and a port number) and a **SOCK_STREAM** transport (which means we want TCP not UDP). Lines 29–32 bind our socket to port 1067 (as defined by **SERVER_PORT** up at line 4); the mysterious constant **INADDR_ANY** that appears here means that we're willing to listen for connections on any of our server's network interfaces. If the machine has only one interface this isn't really an issue, but if a machine has, say, an inward-facing connection and an internet-facing connection, you could choose to accept connections on only one of them.

Choose your socket

At line 34 we tell the system that we want to accept connections on this socket; specifically we set up a queue (of length 5) for incoming connection requests. Then at line 35 we enter our service loop. Line 37 will block until a client comes along to connect; when this happens the **accept()** call wakes up and returns a new file descriptor (**fd**) referencing the connection. At line 39 we call our little **service()** function, which actually carries out the conversation with the client, then when this returns we're careful to close the file descriptor. If we didn't do this we would consume a new descriptor for each

Try It Out – Create a concurrent server

You can turn our iterative "upper case" server into a concurrent server by forking a new child process to deal with each client. Because child processes inherit file descriptors from their parents, this is rather easy. The schema looks like this:

```
#include <signal.h>
#include <stdlib.h>
....
while(1) {
  fd = accept( ... );
  if (fork() == 0) {
    service(fd, fd); /* Child */
    exit(0);
  }
  else {
    close(fd);     /* Parent */
}
```

To test, compile the program and run it as before. Then open three or so terminal windows and conduct a Telnet session with the server in each of them, like we did before. Convince yourself that you are conducting a separate conversation with the server in each window. So far, so good. Now quit out of a couple of those Telnet sessions (leave one open) and look at the process table:

```
$ ps -e | grep ucc
11760 pts/4    00:00:00 ucconcurrent
11837 pts/4    00:00:00 ucconcurrent
11839 pts/4    00:00:00 ucconcurrent <defunct>
11858 pts/4    00:00:00 ucconcurrent <defunct>
```

Here, process 11760 is the original parent process and 11837 is a child that's connected to the one remaining Telnet client. The other two are the children that were servicing the now-closed Telnet sessions. Unfortunately, because their parent isn't waiting for them they have entered the zombie state. (We discussed the formation of zombies in detail in LV004.) If you kill the original parent (just type **^C** in the window where you started it) the zombies will disappear. But this isn't a satisfactory situation, because for a long-lived server those zombies will eventually fill up the process table. Fortunately they're easy to prevent. Just add the line:

```
signal(SIGCHLD, SIG_IGN);
```

in **main()**, before the **while** loop. This will stop the zombies. You'll also need an extra header file:

```
#include <signal.h>
```

You can download the completed code for the concurrent server from the issue landing page on **www.linuxvoice.com.**

There's another way to write a concurrent server, without using child processes, and that's to use the **select()** call. But the details are messy, and we won't consider it further here.

When I first met concurrent servers, I didn't understand how it was possible to maintain multiple client connections on the server, when they're all using the same port. The answer is that the TCP connection is really defined by four items: the port number and IP address of the server, and the port number and IP address of the client. Provided at least one of these four is different, it's a different connection!

client that connects and would eventually run out. Then we simply loop back and await the next client.

The **service()** function at lines 9–19 is really just symbolic of providing a real, useful service. The key points to note are that we read a request from the client using the descriptor returned by the **accept()**, and write a response using the same descriptor. Also notice the loop control at line 13. Each **read()** will block until the client sends another request; when the client eventually closes the connection this read will return 0, and the loop terminates.

The **accept()** call on line 37 also returns a **sockaddr_in** structure (called 'client' in the code) that contains (among other things) the IP address of the client end of the connection. We don't use it in this example, but many real-world servers use this information for logging or access control.

This is a simple iterative server – it talks to just one client at a time and any other that try to connect in the meantime will have to wait. The first five will have their connection requests placed on the queue we created; any beyond that will have their connections refused. However, it is not difficult to turn this into a concurrent server.

What about the client?

We can use a program such as **nc** (or even Telnet) as a client to test our upper-case server, but for completeness' sake let's write a client program, too. I can't bring myself to inflict more C on you, so this one's in Python:

"Once they are connected, the client and server are really just 'peers', and the application protocol will determine the exchange from then on."

```
#!/usr/bin/python
from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 1067))
s.send('Hello World')
reply = s.recv(1000)
print reply
```

Just as in the server we begin by creating a stream (TCP) socket using the 'INET' address family. Then, we actively connect to the server at port 1067. Once the socket is connected, we can send a request to it, and read the reply. We see here the key distinction between the client and the server: the server passively listens for and accepts connections; the client actively connects.

Notice that we don't explicitly bind a port number for the client; the system will bind an (arbitrary) port for it to use. Once they are connected, the client and server are really just 'peers', and the application protocol will determine the exchange from then on. Typically the client transmits first (some sort of request) and the server returns a reply. For some services the server will transmit first, for example the old **Daytime** server on port 13 (which nobody runs any more) works like that – you just connect to it and it sends you a string. TCP is a wonderful protocol, providing the illusion of a reliable, connection-oriented transport on top of an unreliable connectionless one (IP). But there are some things TCP doesn't do. For one thing, it doesn't preserve message boundaries. If a client opens a connection and writes, say, 300 bytes then 50 bytes then 100 bytes down the connection, the server will simply find that it has 450 bytes waiting to be read. Usually, it's left to the application protocol to make it clear where the message boundaries lie.

It's a privilege

There's an important rule in the Linux world that says ports below 1024 are "privileged" -- that is, a program can only bind a privileged port if it's running as root. This rule goes back many years and probably seemed like a good idea at the time, because in theory it prevents regular (non-root) users from masquerading as bona-fide servers and capturing sensitive login information.

Nowadays when many users have root access to their PCs the rule makes very little sense, and forces servers to run as root simply so that they can bind their well-known privileged port. A well-written server will drop its user ID to a non-root account as soon as it's got the port bound. LV

Command of the month: netcat

Netcat (also known as **nc**) has been described as the Swiss Army knife of networking commands. You can use it to create simple TCP or UDP clients or servers, and it's designed to be easy to script around. We've already seen it in use as a client to test our upper case server. Because it acts as a filter, reading stdin and writing stdout like filters usually do, we can perform all the usual tricks of redirecting the streams. Here, we pass the contents of a local file **greet** to our server and capture the result in **greet2**:

```
$ nc < greet localhost 1067 > greet2
```

Using the **-l** (**listen**) option, you can also cast **nc** in a server role. Here it is, being the world's most boring web server, delivering the same file every time:

```
$ nc -l 8080 < somecontent.html
```

Now we can browse to port 8080 and see the content. Of course, it's not a real web server so we don't get a proper HTTP response header back, but the browser doesn't seem to mind. Also useful is the standard output from **nc**, which shows us the actual HTTP request from the browser (somewhat trimmed here):

```
$ nc -l 8080 < greet2
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml
User-Agent: Mozilla/5.0
```

Of course it's a one-shot affair; **nc** will terminate after serving the page once. But with a bit of shell scripting we can wrap a loop around it like this:

```
$ while true; do nc -l 8080 < somecontent.html; done
```

Now here's a challenge for you: I attempted to use **nc** in conjunction with **tr** and a named pipe to cobble together an equivalent to the upper case server at the command line. Following a very similar example in the man page, I tried:

```
$ mkfifo /tmp/f
$ cat /tmp/f | tr 'a-z' 'A-Z' | nc -l localhost 1234 > /
tmp/f
```

... then on the "client" side I ran:

```
$ nc localhost 1234
```

I can enter text, but then the whole thing hangs. If you can figure out why this doesn't work (and especially if you can fix it) drop me a line at **drchrisbrown@linuxvoice.com**. I'd be delighted to hear from you!