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.

UDP: Get plugged in

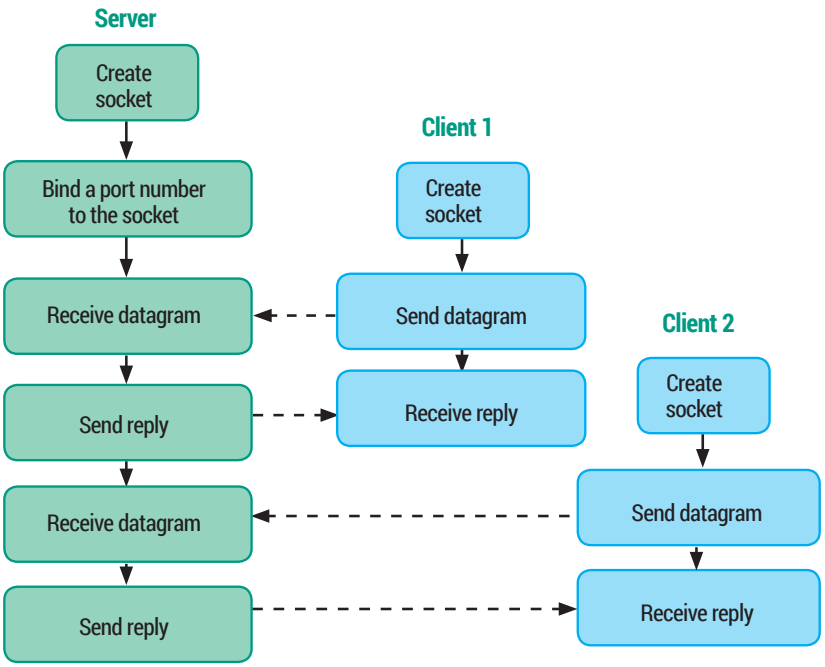
Peek inside your machine to find out how it transmits data packets.

Last issue we implemented a simple server using the TCP protocol, which turned any string you typed into it into upper case letters. While the following examples should make sense on their own, we've put that article up as a PDF at **www.linuxvoice.com/coretech06/** so you can read it alongside this month's.

This month I want to re-cast this server to use UDP. In some ways it's simpler than TCP – there are no **connect** or **accept** operations. I've drawn a flowchart that shows the typical sequence of operations for a UDP-based service. Here we see a single server interacting with two clients. The server has a single endpoint (UDP socket) and may well find itself retrieving datagrams from several clients in an arbitrary, interleaved order. If the server is stateless (that is, if it does not need to remember anything from one client interaction to the next) then this does not present a problem. The server simply reads a request, formulates a reply, returns it to the client, then forgets about it. Classic UDP-based services such as DNS are stateless in this sense.

Things get more complicated for servers that maintain state. One approach is to parcel up the per-client state information into a structure, and place them into some sort of indexed data structure that uses as its search key a composite value formed from the client's IP address and port number. Another approach is for the server to create a child process for each client it finds itself dealing with. Each child can create a new UDP socket, whose port number is duly reported back to the client, and which is used by the client for the remainder of the interaction. The TFTP (Trivial File Transfer) server works this way, for example.

Peer-to-peer architecture using UDP broadcasts



A connectionless server interacts with multiple clients using a single socket.

The power of Python

Most of our code examples this month are in Python, because Python hides some of the fiddly data structures that would be exposed if we wrote them in C. So here's a UDP version of our upper-case server:

```
import socket
port = 4444
s = socket.socket(socket.AF_INET, socket.SOCK_
DGRAM)
s.bind(("", port))
while 1:
  data, addr = s.recvfrom(1024)
  s.sendto(data.upper(), addr)
```

Pretty simple, huh? We create a socket and bind our "well-known" port to it. Then we enter our service loop, retrieving messages from clients, converting them to upper case, and sending them back. Last month, in showing the equivalent code for a TCP server, I briefly made the point that the server doesn't really need to know the address of the client, unless it wants to use it for logging or access control. Here it's different -- we definitely need the address of the client's endpoint (**addr** in the example) so

Try It Out – Create a UDP server

To create and test the upper-case server, place its code in a file called **ucserver.py**.

To test the server, start it in one terminal window:

```
$ python ucserver.py
```

Now we can open a second terminal and test server using the the jack of all trades **nc** (network client) command:

```
$ nc -v -u localhost 4444
Connection to localhost 4444 port [udp/*] succeeded!
XXXXXThis is a test
THIS IS A TEST
it works!
IT WORKS!
^C
```

The **XXXXX** string appearing in the output above is an artifact resulting from a series of probe datagrams that **nc** apparently sends to the server (and which our server duly echoes back). The **connection suceeded** message is a little confusing; this is UDP and there is no connection as such. If you omit the **-v** (verbose) command option you won't see the **X**'s or the message. But we can clearly see that messages we enter are returned in upper case – our server is working.

We can extend the experiment. Leave the **nc** program running in the second terminal window, open a third terminal window and run the same **nc** command there as well. You should find that you can interact with the server via both windows. That's the simplicity of a connectionless service; you don't need any multi-processing or multi-threading or other fancy tricks in the server to get concurrent operation with multiple clients.

that we know where to send the reply. Python's dynamic typing is hiding a little complexity here, because **addr** is actually a (host, port) pair. See the box above if you would like to build and test this server. We could write a little Python program to act as a client to our upper-case server, but let's switch to a different example. There is an ancient UDP-based service called **daytime**, which listens on port 13 and simply sends back a string with the current time and date. It's a sort of speaking clock but without the speaking. This service is implemented by a daemon called **xinetd**; the box below shows how to install and enable it.

Once the **daytime** service is up and running, we can write a client for it. Again using Python, it looks like this:

```
#!/usr/bin/python
# UDP daytime client
import sys
import socket
# Get server host name from command line
host = sys.argv[1]
port = 13
s = socket.socket(socket.AF_INET, socket.SOCK_
DGRAM)
# Send an empty datagram to wake the server up
s.sendto("", (host, port))
data, addr = s.recvfrom(1024)
print "time from", addr, " is ", data
```

And we can run it like this:

```
$ chmod u+x daytimeclient.py
$ ./daytimeclient localhost
time from ('127.0.0.1', 13)  is  01 JUL 2014 10:26:42
BST
```

Notice how Python automagically converts the client address into a printable (host, port) representation.

Broadcasting

One thing that you can do with UDP sockets that you cannot do with TCP is broadcasting. That is, you can send a single copy of a message and have it received by many listeners. The constraints are that all the listeners must be using the same port, and they must all be on the same network, because routers and gateways are almost never configured to pass broadcast traffic. There isn't a lot to it really; you have to explicitly enable broadcasting on the socket, and use a special destination host address of "all ones", or 255.255.255.255 in dotted decimal notation.

The example presented here is both a client and server all rolled into one. The idea is that the client piece periodically generates an item of data, which it broadcasts. The server piece receives the broadcasts and displays the item of data. For simplicity the 'item of data' is a simple randomly generated integer, but could be something more interesting in the real world – a weather forecast or a stock price, perhaps.

The code here is in C, and it looks more complicated than the Python examples we've seen so far, but conceptually it's not really any harder. As always, the line numbers are for reference; they are not part of the code:

```
1 #include <stdlib.h>
2 #include <netdb.h>
3 #include <stdio.h>
4 #include <arpa/inet.h>
5
6 #define UPDATE_PORT  2066
7
8 void main ()
9 {
10  int sock;                /* Socket descriptor  */
11  struct sockaddr_in server; /* Broadcast address
*/
12  struct sockaddr_in client;
13  int client_len, yes = 1;
14  int value;
15
16  /* Create a datagram socket and enable
broadcasting */
17  sock = socket (AF_INET, SOCK_DGRAM, 0);
18  setsockopt (sock, SOL_SOCKET, SO_BROADCAST,
(char *) &yes, sizeof yes);
19
20  /* Bind our well-known port number */
21  server.sin_family = AF_INET;
22  server.sin_addr.s_addr = htonl (INADDR_ANY);
23  server.sin_port = htons (UPDATE_PORT);
24  bind (sock, (struct sockaddr *) &server, sizeof
server);
25
26  server.sin_family = AF_INET;
27  server.sin_addr.s_addr = 0xffffffff;
28  server.sin_port = htons (UPDATE_PORT);
29
30  /* Create an additional process. The parent acts
as the client,
31     periodically broadcasting values to anyone who
happens to be
32     listening on port 2066.  The child acts as the
server,
33     receiving the broadcasts and displaying the
data.
34  */
```

Try It Out – Install the daytime service

To get the **daytime** service running we first need to install **xinetd** (this is on Ubuntu, but the story should be similar on other distros):

```
$ sudo apt-get install xinetd
```

Even if **xinetd** were already installed, the **daytime** service is probably disabled. So, edit the file **/etc/xinetd.d/daytime**, find the stanza that relates to the UDP version of the service and change the line **disable = yes** to read **disable = no**. Now restart **xinetd**:

```
$ sudo invoke-rc.d xinetd reload
```

(On a Red Hat-style system you would need **service xinetd restart** instead.) Now verify that the **daytime** server is listening:

```
$ sudo lsof -i | grep daytime
xinetd   27465   root   5u  IPv4 165011      0t0  UDP
*:daytime
```

If you don't see an encouraging line of output here, you'll need to investigate before moving forwards. We can test this service using **nc** again:

```
$ nc -u localhost daytime
01 JUL 2014 10:16:20 BST
```

You will need to send the **daytime** server a datagram of some sort (just enter a blank line).

The Berkeley sockets library, dating from 1983, remains the standard sockets API to this day.

```
35  if (fork ())
36    {                       /* PARENT (client) here */
37      while (1)
38      {
39        value = rand () % 1000;
40        /* Broadcast update packet to servers */
41        sendto (sock, (char *) &value, sizeof value, 0,
42              (struct sockaddr *) &server, sizeof
server);
43        sleep (1);
44      }
45    }       /* End of parent (client) code */
46
47 /* ------------------------------------------------
*/
48
49  else
50    { /* CHILD (server) here */
51      /* Enter service loop, receiving values and
displaying them */
52      while (1)
53      {
54        /* Receive an update packet */
55        client_len = sizeof client;
56        recvfrom (sock, (char *) &value, sizeof value,
0,
57              (struct sockaddr *) &client, &client_len);
58
59        /* Display the broadcast value and where it
came from */
60        printf ("got %3d from %s\n", value, inet_ntoa
(client.sin_addr));
61      }
62    }     /* End of child (server) code */
63 }
```

Now there's quite a bit of code here, and some of it is messy. So grab a brown paper bag (so that you can breathe into it for a bit if you start to panic) and let's work through it. First, the declarations at lines 11 and 12 refer to the endpoint addresses used for sending and receiving. (The name **sockaddr_in** means 'internet socket address'. When I first met this years ago I thought that the "in" meant "input", and spent some time looking for a **sockaddr_out**, which my sense of symmetry told me must be there, like the Higgs Boson. But I digress.)

At line 17 we create our socket and at line 18 we set the **SO_BROADCAST** option on it. Just look at the hoops we have to jump through to pass in a Boolean TRUE value.

Lines 20–24 bind our chosen port number (2066) to the socket. At lines 26–28 we re-use the 'server' structure to hold the broadcast address. Notice the **0xffffffff** value, which is the "all ones" of the broadcast address.

Now we get cunning, rolling the client and server pieces of the application into one program by creating another process. The parent process (lines 37–45) is the client. Once a second, it generates a random integer value, and broadcasts it in a tiny 4-byte datagram. The child process (lines 52–62) is the server. It receives the broadcast packets and prints out each value, along with the IP address of the client that sent it. The important thing to keep in mind here is that this loop is not only receiving the broadcasts from its own client, it will also receive the broadcasts from all other instances of the client running elsewhere on the network.

Raw sockets

I won't inflict any more code on you this month, but I wanted to wrap up by mentioning two more socket types. First, raw sockets enable an application program to reach right down to the IP layer and 'hand-craft' the headers of whatever the overlying protocol is. For example, the port scanner *Nmap* uses raw sockets to build non-conformant TCP headers for its own special purposes. As another example, **ping** (which sends and receives ICMP packets) also uses raw sockets. On Linux, the rule is that only processes running with root privilege can use raw sockets. This is a security precaution, because a program using raw sockets can intercept all traffic entering the system. A common way to deal with this is to have the program run "set UID to root". This enables it to create its raw socket, then drop its privilege back to a non-root user. If you look at the **ping** program for example, you'll find it runs setuid for this reason:

```
$ ls -l /bin/ping
-rwsr-xr-x 1 root root 44168 May  7 22:51 /bin/ping
```

Notice the **s** in the permissions.

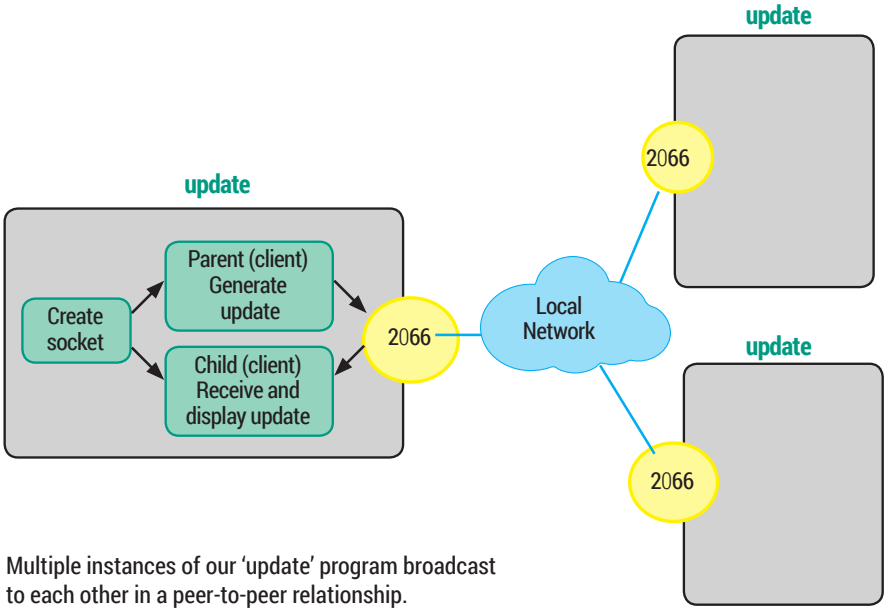
Staying in the Unix domain

We've focussed here on sockets in the internet domain, which means that (among other things) the socket is identified by an IP address and a port number. But there are other naming domains for sockets; in

Get the code

A tarball of the programs used in this tutorial can be downloaded from **www.linuxvoice.com/mag_code/lv07/coretech007.tar**.

Timeline for connectionless server



Multiple instances of our 'update' program broadcast to each other in a peer-to-peer relationship.

particular, the so-called "Unix domain sockets" are identified by a name within the filesystem. You can find all of these with the command:

```
$ sudo find / -type s
```

A classic example is **/dev/log**, which **syslog** (or **rsyslog**) uses to collect log messages from local applications. But you will probably find many others.

There's no command for creating a named socket analogous to **mkfifo** for creating named pipes. Your server creates the socket and binds the name to it, and your client needs to know that name in order to connect. Unix domain sockets only support communication between processes running on the same machine. There are also anonymous Unix domain sockets created with the **socketpair()** system call; these are somewhat similar to good ol'anonymous pipes (see Core Technologies in LV005) but unlike pipes, which are unidirectional, a socket pair is bidirectional. Also, you can create both stream and datagram socketpairs, whereas pipes are inherently stream-oriented.

Try It Out – Build a peer-to-peer update service

If necessary, install the gcc compiler:

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

Enter the code into a file called update.c and compile it:

```
$ gcc update.c -o update
```

Run it like this:

```
$ ./update
got 383 from 192.168.1.69
got 649 from 192.168.1.73
got 886 from 192.168.1.69
got 421 from 192.168.1.73
```

To do a meaningful test you'll need to copy the executable across onto at least one other machine on the same network, and run it there as well. (You can't run multiple instances on the same machine -- why not?) In the output above, you'll see that we're receiving interleaved broadcasts from two machines.

Command of the month: dig

My command of the month is **dig**. According to its man page it stands for "domain information groper", though that sounds like a retrofitted acronym if ever I heard one! Anyway, **dig** is a command-line tool for performing DNS queries.

In my view, **dig** has two main uses. First, you can use it to test your DNS service. Second, you can use it as an exploration tool. It's this second use we'll focus on here. We'll use *Linux Voice*'s own site as a target for our exploration. First let's just find the IP address of the website; this is the simplest type of lookup:

```
$ dig www.linuxvoice.com
;; QUESTION SECTION:
;www.linuxvoice.com.            IN      A
;; ANSWER SECTION:
www.linuxvoice.com.  600  IN  CNAME  linuxvoice.
com.
linuxvoice.com.      600  IN  A      213.138.101.172
```

I've edited a lot of detail from this output but you'll see it shows that we requested an 'A' record from DNS for the name **www.linuxvoice.com**. ('A' records are the records in DNS that map machine names to IPV4 addresses.) What we actually got was a CNAME record (an alias, in effect) pointing to the name **linuxvoice.com**. Dig then kindly looked up the A record for **linuxvoice.com**, finally reporting the IP address.

There are command-line options for **dig** that control how much output we see. For example, **+noquestion** suppresses the **question** section from the output. You can turn off other output sections using options such as **+nocomments**, **+noauthority**, **+noadditional** and **+noanswer**, or you can turn everything off using **+noall** and then explicitly enable the sections you want to see. For example, this shows just the **ANSWER** section:

```
$ dig ubuntu.com +noall +answer
; <<>> DiG 9.9.5-3-Ubuntu <<>> ubuntu.com +noall
+answer
;; global options: +cmd
ubuntu.com.  577  IN  A  91.189.94.156
```

If there are options you always want to specify, just put them into **~/.digrc**. For example, if you put this line into the file:

```
+noall +answer
```

then by default your **dig** queries will only show the **ANSWER** section.

The **+short** option really cuts to the chase and shows just a bare-bones response:

```
$ dig +short linuxvoice.com
213.138.101.172
```

Dig deeper

Next let's investigate who handles mail for the **linuxvoice.com** domain. For that we need to get the MX (Mail Exchanger) record:

```
$ dig +short linuxvoice.com mx
10 smtp.linuxvoice.com.
$ dig +short smtp.linuxvoice.com
213.138.101.172
```

So... mail is handled by a machine called **smtp.linuxvoice.com**, and it turns out that this is the same machine (same IP address) as the web server. So, Linux Voice apparently hosts its own web and mail servers on a single machine. No surprises there.

Let's try a reverse lookup on that machine. That is, let's look up the PTR record for that IP address and convert it back to a machine name. The PTR records are stored under the **in-addr.arpa** domain, and because DNS names are written in a "little endian" form, we end up with the four octets of the IP address reversed. So here's the hard way to do the lookup:

```
$ dig +short 172.101.138.213.in-addr.arpa ptr
mainsite.default.linuxvoice.uk0.bigv.io.
```

An easier way is to use the **-x** option of **dig**, which lets us enter the IP address in the usual format:

```
$ dig +short -x 213.138.101.172
mainsite.default.linuxvoice.uk0.bigv.io.
```

What's interesting is that the IP address is allocated to a machine in the **bigv.io** domain. A quick search reveals that BigV is a virtual machine hosting provider – now we know who Linux Voice uses to host its site.

A handy tool for stalkers

Let's try something a little different, by asking DNS where its root name servers are:

```
$ dig +short . ns
f.root-servers.net.
i.root-servers.net.
d.root-servers.net.
```

Here, **ns** means we're looking for name server records and **.** refers to the top level domain. It is analogous to **/** in a filename, which names the root directory (the top-level directory) in a filesystem. I've cut the output down again; there are actually 13 root name servers, I've shown only three.

By default, **dig** consults the file **/etc/resolv.conf** to figure out which name server to consult, just as normal DNS lookups do. But we can direct **dig** to a specific DNS server like this:

```
$ dig @8.8.8.8 +short jamieoliver.com
85.233.160.22
```

Here, 8.8.8.8 is the IP address of Google's public DNS service.