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

Prise the back off Linux and find out what really makes it tick.

The Internet Protocol

The Internet Protocol is at the heart of – well – the internet. But what exactly does it do?

What would you consider to be the most important inventions of the last 50 years? Genetic engineering? Post-It Notes? The Teletubbies? How about the Internet Protocol? It underpins the entire internet (obviously) and has found its way into cars, fridges, televisions, smoke alarms, in fact the entire "Internet of Things".

Let's start with the big picture, and talk about protocol stacks. Consider the very common case of a web server sending a web page to a browser. The server and the browser communicate with a protocol called HTTP (Hyper Text Transfer Protocol). So the web server builds an HTTP response packet, which consists of the content of the web page it's sending, with an HTTP header stuck on the front. This header contains the information that the HTTP layer needs to do its job. It's in this header, for example, that you'll find the HTTP status code such as 200 (OK) or 404 (file not found).

Enter the TCP layer

Having assembled the packet, the browser hands it down to the transport layer, TCP (see Figure 1). The task of this layer is to "guarantee" delivery of the packet to the correct program (in this case, the web browser) on the destination machine by providing the illusion of a permanent "circuit" connecting the server and the client. This layer adds its own, rather complicated, header to help it do its job. The TCP layer doesn't know anything about the data it's carrying. For example, it doesn't distinguish the HTTP header from the rest of the packet. As far as the TCP layer is concerned, the whole thing is just the "payload" it's being asked to deliver.

The TCP layer hands the packet down to the IP layer, which is responsible for routing packets across an interconnected set of networks (an "internet") to the correct machine. The IP layer adds its own header, and again, it regards the whole of the packet handed to it from the layer above simply as its payload.

There's at least one more layer below that before the packet actually hits the wire. The detail here depends on what medium is being used to actually transmit the packets; assuming that it's some form of Ethernet, the IP datagram will get encapsulated inside an Ethernet frame, with its own header and its own destination address, as I'll discuss. (Though for a tongue-in-cheek alternative, see RFC1149: A Standard for the Transmission of IP Datagrams on Avian Carriers.)

When the packet reaches its destination (where your browser is running) it proceeds back up the protocol stack, each layer discarding its header before passing its payload up to the layer above. Finally the original HTTP packet is handed up to the browser which (after removing the header) renders the page for you.

Each layer thinks of itself as talking directly to its peer layer – the one at the same level in the stack – at the other end. The application layer talks to the application layer, the TCP layer talks to the TCP layer, and so on. In reality, of course, the data flows down and up the protocol stacks.

IP addressing

Back in issues 6 and 7 I discussed the TCP and UDP protocols in some detail, with emphasis on the "sockets" API that provides access to these protocols from our code. I want to focus on the IP layer this month. Typically, programmers do not interact directly with this layer, although it is possible to create a "raw" socket that lets you craft your own transport layer header. Program like **ping**, and some of the weirder forms of **nmap scan**, use this technique. But we are not really going to look at IP through a programmer's eyes.

To begin at the beginning, every connection from a computer to an internet

IPv6

You probably don't need to be told that we're running out of IPv4 addresses. RIPE (the organisation that allocates these things in Europe) started allocating addresses from its last /8 block two years ago (a /8 block is 2^24 addresses, roughly 16 million, which sounds a lot but is actually less than 0.5% of the IPv4 address space).

The number of addresses available in IPv6, with its 128-bit addresses, is too big to get a proper handle on. I just used two Post-It notes working out that you could allocate the equivalent of an entire IPv4 address space for every square millimetre of the earth's surface – in fact, you could do it 100 million times over. Although IPv6 is on its way, it's slow in arriving. You've been able to build IPv6-only intranets with Linux for years. The latest infographic from RIPE claims that globally, more than 20,000 websites, 240 network operators, and 10 home router vendors now offer IPv6 products and services.

Nonetheless, I think we're still some way away from having full end-to-end IPv6 connectivity from the average home user to the average website. I keep thinking I'll call my ISP and ask them… but they'll just tell me to reboot my router and see if that fixes the problem…

is allocated an IP address, which is 32 bits long and is written in a format called "dotted decimal notation" – you split the address into four lots of 8 bits (called an octet), write each octet's value down as a decimal number (giving a value between 0 and 255) then stick dots in between. So you end up with something like 104.28.7.18. This address is logically split into two parts – a network ID and a host ID. The network ID is the piece that's used for routing (getting packets to the right network); the host ID only comes into play once a packet has reached the right network, when it's used for the final stage of delivery to the destination machine.

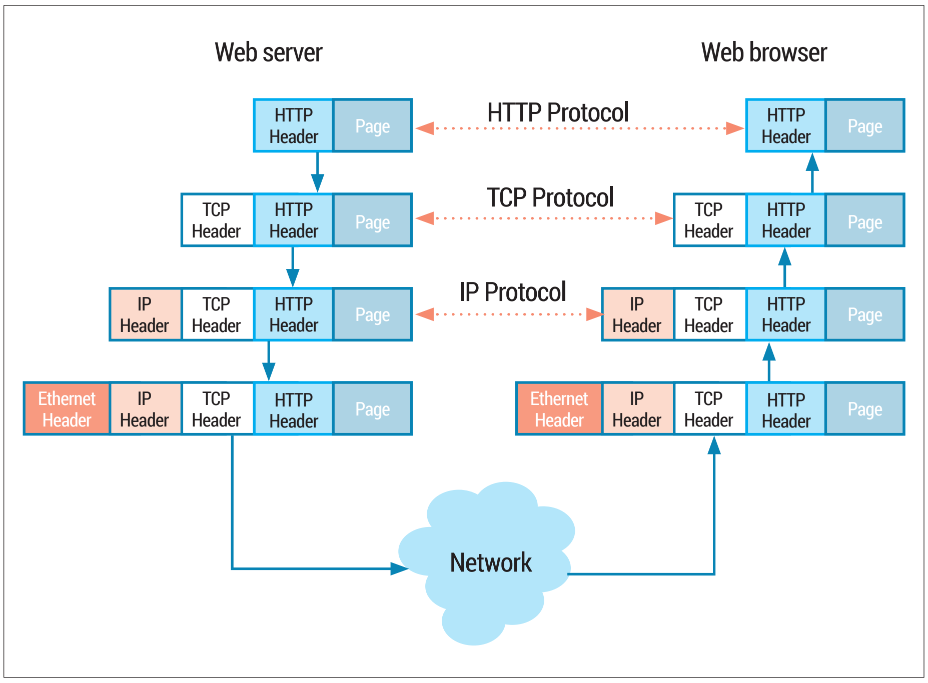


Figure 1: As a packet passes down through the layers of a protocol stack, each layer's header forms part of the payload of the layer below.

What's a subnet mask?

The division between the network piece and the host piece is specified by the "subnet mask", which is also usually written in dotted decimal notation. For example, a subnet mask of 255.255.255.0 converted to binary gives us 24 ones followed by 8 zeros, meaning that the top 24 bits of the IP address are network ID, and the remaining 8 bits are the host ID. There's a more compact way of representing this. We might say that a machine is on the network 192.168.1.0/24, meaning that the top 24 bits of this (the 192.168.1 piece) specify the network and the remaining 8 bits select the host.

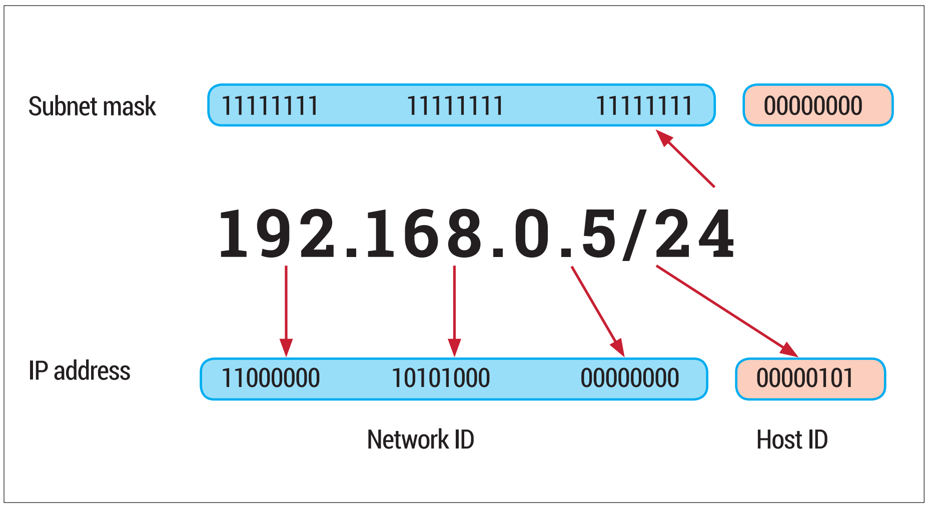
Figure 3 shows a typical small internet. Machines A, B, C and D are connected to the upper network 192.168.0/24; machines P, Q, and R are connected to the lower network 192.168.1/24. Additionally, machine S is connected to both networks (it has two network cards) and can route packets between them. Finally, machine D has a connection to the outside world.

In the early days of the internet every single machine that used TCP/IP had a globally unique IP address assigned to it. We could establish direct end-to-end connectivity between any two machines. But the internet grew way beyond expectations and we started running out of addresses. So in 1996, the Internet Assigned Numbers Authority designated three "private" address blocks as follows:

10.0.0.0–10.255.255.255 (10/8 prefix)
172.16.0.0–172.31.255.255 (172.16/12 prefix)
192.168.0.0–192.168.255.255 (192.168/16 prefix)

The idea was that machines that only needed to communicate within their own private "intranet" could use IP addresses from these private blocks and didn't need to apply for an address allocation from a central registry. More than anything else, this strategy has staved off the exhaustion of IPv4 addresses, as countless corporate networks around the world re-use these private address blocks.

In our diagram, there is only one globally unique IP address – that's the 176.13.4.92 address of the outward-facing connection of machine D.



An IP address is split into a network ID and a host ID. The "CIDR" notation shown here specifies the boundary between the two pieces

The routing routine

So, what does the IP layer do, exactly? Well, it has the job of delivering a packet to a specified destination IP address. To figure out how to send IP packets on their way, each machine maintains a routing table. Machines on "stub" networks, like machine P in the diagram, only need to know two things – which network they're connected to (192.168.1/24 in this case) and where to send packets destined for other networks (192.168.1.254 in this example); this is usually called the default gateway.

If we examine the routing table of machine P, we'll see something like this:

```
$ route -n
Kernel IP routing table
Destination   Gateway        Genmask        Flags Iface
0.0.0.0       192.168.1.254  0.0.0.0        UG    eth0
192.168.1.0   0.0.0.0        255.255.255.0  U     eth0
```

I'll explain all this in a minute, but first let's

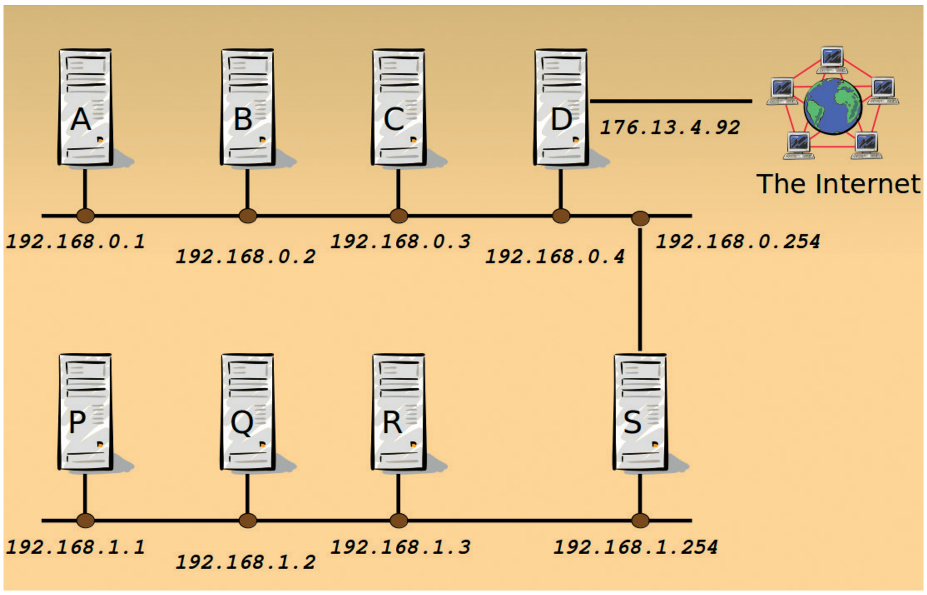


Figure 3: The internet in miniature – two networks connected by a gateway.

look at the routing table for machine A, which has an extra entry because it needs a route onto the lower network, so it might look like this:

```
$ route -n
Kernel IP routing table
Destination   Gateway         Genmask         Flags Iface
0.0.0.0       192.168.0.4     0.0.0.0         UG    eth0
192.168.1.0   192.168.0.254   255.255.255.0   U     eth0
192.168.0.0   0.0.0.0         255.255.255.0   U     eth0
```

Here's how it works. The IP layer works through the entries in the routing table in turn. For each one, it takes the packet's destination IP address, bit-wise ANDs it with the value in the Genmask column and compares it to the value in the Destination column. If they match this is considered as a potential route. If more than one entry in the routing table matches, the most specific route – the one with the longest Genmask – wins.

So, taking the three entries in turn: the first entry always matches, because any destination IP address AND-ed with 0.0.0.0 is going to give 0.0.0.0. So this route will be used if there isn't a more specific match – it says that 192.168.0.4 is our default gateway. The second entry defines the route onto the lower network; basically it says "to reach the 192.168.1/24 network, go via 192.168.0.254". The third entry has no gateway defined; it says that traffic to the 192.168.0/24 network doesn't need to go via a gateway because that's the network we're actually connected to. In all three cases, packets will go out via network interface eth0. That's a bit of a no-brainer because it's the only one we've got. Let's take a look at the routing table on machine S:

```
$ route -n
Kernel IP routing table
Destination   Gateway       Genmask         Flags Iface
0.0.0.0       192.168.0.4   0.0.0.0         UG    eth0
192.168.0.0   0.0.0.0       255.255.255.0   U     eth0
192.168.1.0   0.0.0.0       255.255.255.0   U     eth1
```

A careful examination of this (the last two lines) shows that the machine has direct connections to two networks, 192.168.0/24 (via its "upper" network connection eth0), and 192.168.1/24 (via its lower connection eth1).

To get a feel for how IP routing and packet delivery works, let's consider three routing scenarios in turn:

1 Machine A to machine C.
2 Machine A to machine Q.
3 Machine A to a machine somewhere in the outside world.

Machine A to machine C

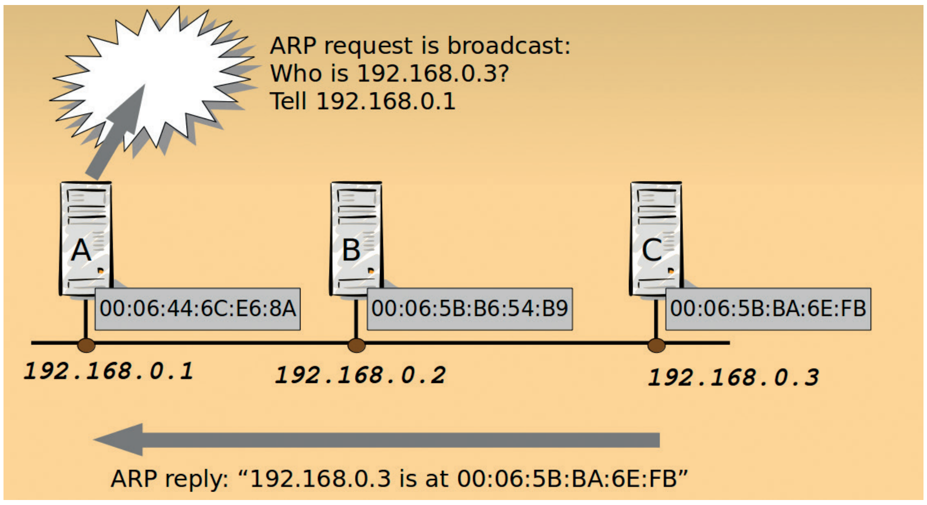
This is the easy case, because the destination address of the packet, 192.168.0.3, is on the same network as machine A, as determined by the third entry in machine A's routing table. But we're not quite home and dry, because the packet needs to be encapsulated into an Ethernet frame for transmission, and we need to know the Ethernet address of machine C. Ethernet addresses are 48 bits long and are written down as a group of 6 pairs of hexadecimal digits, separated by colons, for example 00:06:5B:BA:6E:FB. Ultimately, it's this address that's used to get the packet to the right machine.

Keep in mind though, that it's pointless addressing a packet to an Ethernet address that isn't on your network -- these addresses are not used for routing. The Address Resolution Protocol (ARP) is used to discover the Ethernet address. Essentially, machine A broadcasts an ARP request onto its local network that says "Who is 192.168.0.3? Please tell 192.168.0.1". All the machines pick up and ponder this request but only machine C, recognising its own IP address, responds with the reply: "192.168.0.3 is at 00:06:5B:BA:6E:FB". Finally, machine A is able to build an Ethernet frame and send it out on the wire in the reasonable expectation that it will reach machine C.

Broadcasting an ARP request every single time you want to send an IP datagram is clearly not smart, so machine A will keep the result for a while (60 seconds by default) in its ARP cache. You can examine this cache with the **arp** command:

```
$ arp -a
? (10.0.2.2) at 52:54:00:12:35:02 [ether] on eth0
```

You can also manually add and delete ARP



A broadcast ARP request is used to find the Ethernet address of a directly connected machine whose IP address is known.

cache entries with this command, though there shouldn't be any need to.

Machine A to machine Q

Our second scenario, machine A sending to machine Q, is a little more complicated. The destination IP address for machine Q is 192.168.1.2. From the second entry in machine A's routing table, it discovers that to reach this network it needs to send the packet to 192.168.0.254, the upper network connection of machine S. So it will check its ARP cache for an entry for this IP address, and use the associated Ethernet address if it finds one, or broadcast an ARP request if it doesn't. Note that machine A has absolutely no idea what will happen to the packet after it reaches machine S.

The focus of attention now turns to machine S, the gateway. Tasked with delivering the packet to 192.168.1.2, it discovers from its routing table that one of its network interfaces (eth1) is directly connected to that network. So it will broadcast an ARP request on eth1 to get an Ethernet address for machine Q, and finally send the packet to its destination.

Machine A to the outside world

In our final routing scenario, machine A wants to send a packet to a machine out in the internet -- perhaps to Linux Voice's web server at 104.28.6.18. Machine A quickly discovers that its only hope is to go via its default gateway (machine D at 192.168.0.4). Now we haven't looked at machine D's routing table, but it will in turn discover that the packet needs to go out on the 176.13.4.92 interface to its own default gateway -- a machine operated by the site's Internet service provider.

But there's a problem. Sending the packet out with a destination address of 104.28.6.18 and a source address of 192.168.0.1 will work fine, but getting a reply back is a different matter: 192.168.0.1 is a private address; you can't route packets to it across the internet.

So here's what happens. Machine S picks an unused TCP port on its outward-facing interface. Suppose it picks port 13348. It then re-writes the SOURCE IP address and port number on the packet to be 176.13.4.92 and 13348, and sends the packet on to Linux Voice's web server. This server thinks the request originated at machine S and sends the reply back there; that is, to 176.13.4.92 port 13348. Machine S, meanwhile, has remembered the IP address and port number that this request originally came from – ie, machine A. So it now re-writes the DESTINATION IP address and port number of the reply packet and sends it back to machine A.

"NAT is fundamental to how machines on private internal networks interact with the outside world."

This trick is known as NAT (Network Address Translation) and is fundamental to how machines on private internal networks can interact with the outside world. If you browse the web from home, your broadband router does this on every single packet you send. Note that machine A has no idea that NAT is taking place – as far as it's concerned, it's sending the packet to machine S simply because it's the default gateway to the outside world.

NAT is, in a sense, extending the IP address space by using the port number as part of the address. This form of NAT is sometimes called IP masquerading, because it hides the internal structure of our network from the outside world. It only works when a network connection is initiated from a machine within the local intranet. A web browser running somewhere "out there" cannot connect to a web server running on our intranet. In this sense, NAT offers a kind of firewall, protecting our systems from external attack.

So... if you get the impression that all this routing stuff can get complicated... well, you're right. But keep in mind that the operations I've described occur thousands of times on maybe a dozen machines, just for a single visit to a website. Long live IP!

So long and thanks for all the fish

I've decided to stop writing for Linux Voice, as part of a process that I am told is called "retirement". So this will be my last piece. I'd like to thank the Editorial Team for offering me the opportunity over the last months, and wish them every success as they take the magazine through into its second year. Thanks to all of you who have read what I've written – that's what makes it all worthwhile. Oh, and the cheques of course.

Command of the month: ip

The **ip** command is the main administrative tool for things down at the IP layer. It's intended to replace commands like **ifconfig**, **route** and **arp**. As such, it's a bit of a jack-of-all-trades, with an extensive command syntax. Commands are basically of the form:

```
# ip object action
```

where the objects you can perform actions on include addresses, network interfaces (**ip** calls them links), **arp** cache entries, and routes. The actions you can perform depend on the object you're operating on, but typically you can show, add or delete them. Here are a few examples:

To show all addresses assigned to all interfaces (roughly analogous to the old **ifconfig -a**):

```
$ ip address show
```

To list just the IPv6 addresses assigned to eth0:

```
$ ip -family inet6 address show dev eth0
```

To show the routing table (similar to **route -n**):

```
$ ip route show
```

To add the static route from machine A to the bottom network in our example:

```
$ sudo ip route add 192.168.1.0/24 via
192.168.0.254 dev eth0
```

...and to delete it again:

```
$ sudo ip route del 192.168.1.0/24
```

The **help** option of the command makes it, to some extent, self-documenting. For example:

```
$ ip help
```

will give you a list of the object you can operate on, and drilling down a level further:

```
$ ip route help
```

will show you the actions you can perform on a route.

I get the impression that the **ip** command hasn't gained quite the level of adoption that it perhaps deserves, a result (I suspect) of its extensive command syntax, and the inertia of the sysadmin community