

Using Linux as a Router

Jonathan Feldman

Linux can be a terrific "poor man's" router. It takes a little more startup configuration than a typical hardware router, but once it is going, very little will stop it, at least in our experience. We've been using the router setup detailed below in production for six months now (a few months more worth of pilot-testing), and the only faults we've encountered were either service-provider related or due to water-soaked cables -- unbelievable, but true, and a really long story I won't get into here!

Concepts

Since I'm about to describe how to roll your own router, it will help if you understand how most TCP/IP routing works. Any router, whether Cisco, Proteon, or Linux, is based upon the premise that packets need to be forwarded. Why? Because, presumably, the packets that come in on one interface are not local to the other interfaces in the router. Therefore, the router in question must be able to take a packet, look at its destination, and forward it to the appropriate interface.

How does the router map network destinations to interfaces? Well, just as every router needs to be able to forward packets, so too it needs the ability to consult, build, and update a lookup table, called a routing table, that maps destination networks to interfaces.

If a packet comes in for network X, the routing table is consulted and the packet is dumped to the appropriate interface -- a local node that is either another gateway or the packet's final destination. If network X does not exist in the routing table, the packet is dumped to the default destination, also denoted by IP address 0.0.0.0. If you have not specified a default destination, the packet is dropped, and an ICMP "Destination Unreachable" message is sent back to the originating network client.

How does the router maintain the routing table? In two ways: statically and dynamically. Static routes are added by you, in a known, and sometimes tedious, fashion. Typically, default routes are static routes.

Dynamic routes are "learned" routes, learned by some sort of routing protocol. For our installation, we were only interested in one routing protocol, RIP (Routing Information Protocol). It is simple, and widely available.

When a RIP router sees a RIP packet (which is a broadcast packet -- good to know for troubleshooting purposes), it updates the routing table so that future packets will follow the learned route to the new destination.

By the same token, the router "advertisers" when it has a destination that other machines should know about. Obviously, routers that need to talk to each other need to speak the same protocol, or all is confusion!

Although RIP is okay for small networks with non-variable subnet masks, you will want to check out OSPF and BGP if you are about to embark on a complex networking plan. First of all, RIP is a broadcast protocol, which means your network is subject to broadcast storms. Second, RIP gives no "weight" to any particular route -- one route is considered as good as another. For example, RIP would consider your 9.6KBps backup link to a site to be just as good as a your primary 56KBps link.

Implementation

Our needs were extremely simple: a routing system that would connect one remote site to a shared Internet router that spoke RIP on our end. The site had one LAN to start with, but would probably be adding a few more.

We decided to use Linux because it was inexpensive enough for a pilot project: it would cost us two PCs plus labor. Also, we would be able to have one machine handle routing services, mail services, limited FTP services, and name services. This saved us the startup costs of a dedicated router plus a UNIX or dedicated DOS box to manage DNS and POP. Our goal, and current implementation, is shown in Figure 1.

We built the first machine out of a surplus (and highly generic) 80486 motherboard with 8Mb of RAM and a 120Mb IDE hard drive. We purchased and installed the Slackware distribution of Linux off a borrowed NEC CD-ROM, making sure to install kernel source code, the PPP daemon, and the various user-level services we wanted to run.

The first thing to do was to modify the kernel to act as a packet forwarder. Slackware came with many excellent precompiled kernels, but IP forwarding was not an option. Apparently, some random RFC specifies that IP forwarding is not to be turned on by default. This actually makes sense: why spend the processor cycles dealing with it if you don't need to? So, compiling a kernel ourselves was our only option. However, it's not the only option you have: if you want a precompiled Linux 1.2.0 kernel with IP forwarding, ethernet, and token-ring support, get it from

```
ftp://moria.co.chatham.ga.us/pub/vmlinuz.tr.eth.ipfwd
```

At first, the notion of compiling a kernel seemed daunting. Before this, we'd only re-linked OEM UNIX kernels. It turned out to be much easier than expected; the processor and hard drive were the only things that had to work hard.

The directory /usr/src/linux is, by convention, a symbolic link to the current production version of Linux running on your machine. Finding it is easy: The README file contained therein is fairly short and to the point, and describes how to configure the kernel and install it.

To make a kernel, we typed:

```
# cd /usr/src/linux
# make config
...
*
* Networking options
*
TCP/IP networking (CONFIG_INET) [y]
IP forwarding/gatewaying (CONFIG_IP_FORWARD) [n] y

# make dep ; make clean
# make zImage
```

Then, before installing the kernel, in addition to backing up the current kernel, we also made a boot disk, just in case. After putting a floppy in the A: drive (for you DOS types), we typed:

```
# dd if=/vmlinuz of=/dev/fd0
# cp /vmlinuz /vmlinuz.safe
# mv /usr/src/linux/arch/i386/boot/zImage /vmlinuz
# lilo
Added linux
# sync ; reboot
```

A Few Caveats

If you have played with LILO (the LInux LOader), /vmlinuz may very well not be your system kernel. Check /etc/lilo.conf if you're unsure.

Make sure you use networking tools that match your kernel version -- all sorts of strange things can happen with a mismatch! Specifically, watch out for arp, route, ifconfig, and gated. A good rule of thumb is to stick to the versions that come on the CD-ROM distribution. If you're getting your files from the Internet, pay close attention to READMEs.

Stick to the defaults when configuring by hitting ENTER at the prompts. Be sure you don't configure a device that you don't have -- if you do, you risk lockups and erratic system behavior.

In our case, we also had to scour the earth for a Token Ring driver for Linux, and compile it into our kernel, but that's another long story. If you are looking for Token Ring support, and are willing to live without busmastering and TI chipset support, check out Peter De Schrijver's Wonderful Token Ring driver:

ftp://linux3.cc.kuleuven.ac.be/pub/Linux

or

ftp://ftp.cs.kuleuven.ac.be/pub/unix/linux

See Terry Dawson's comprehensive NET2-HOWTO for current information on available networking technology support. Look in:

ftp://sunsite.unc.\
edu/pub/Linux/docs/\
linux-doc-project/network-guide

Token Ring may well be included in many distributions by the time you read this. I will mention, however, that we had problems using kernels earlier than 1.2.

Once the kernel was recompiled and ready to forward packets according to its routing table, we needed to give it routing "smarts," so we started up routed, the ancient routing daemon. This was a mistake; routed is obsolete and flakey, and should not be used here. Instead, use gated. It will speak every routing protocol you're ever likely to need, and it is easy to configure for RIP (see Figure 2). If you need to use something other than RIP, say, OSPF, you can rest easy knowing that the gated distribution has quite a few sample configuration files for you to copy.

Any router needs more than one interface to be useful, and our Linux router was no exception. We chose to use PPP (Point-to-Point Protocol) in conjunction with ISDN terminal adaptors, because ISDN service is extremely inexpensive here. (ISDN was not supported with most Linux distributions at the time, but it

may be by the time you read this.)

Also, pppd, the point-to-point daemon, supports "proxy arp," which lets you avoid wasting a network on the point-to-point link. Basically, proxy arp makes your remote machine's point-to-point interface appear as if it is on your local network, in our case, the 167.195.160 network (see Figure 1).

The pppd support for a Token-Ring frame type was not complete at the time of this writing. The version of pppd that we used (2.1.2) would proxy-arp, but with a frame type of Ethernet, not Token Ring. The files /etc/ppp/ip-down and /etc/ppp/ip-up (Figure 3) were necessary in order to kludge the proxy arp on the local end. You don't have to do this if you're using Ethernet, or if this has been fixed in your version of the pppd.

We chose terminal adaptors that would turn a synchronous 64Kbps channel into an asynchronous 57.6Kbps channel, so that we could use a standard 16550 UART serial port, /dev/ttys0 on both machines. Note that you must use a 16550 or other buffered or intelligent serial port to run at this speed (or 128K synch transformed into 115K asynch). If you do not, you will experience data overruns and thus network errors.

Also note that when using Linux, if you want to use serial speeds above 38,400Bps, you must use the setserial program to make the 38,400 transform into either 57,600Bps (spd_hi) or 115,000Bps (spd_vhi) (see Figure 4 for details).

Testing and Troubleshooting

Once the first system was built, we tested the PPP connection locally by connecting a DOS-based PC to it, with FTP Software's PC/TCP PPP stack. Everything looked fine; a ping from the PC to the Linux box worked, which meant that PPP was happening. Then we pinged the Internet router successfully, which meant that the IP forwarding also worked.

At this point, it was time to "clone" the first machine to create the remote router. A quick modification of the Slackware root diskette was in order. We deleted tar and replaced it with cpio, so that we could restore a backup of the entire filesystem, including device files. We backed up the first system over the network to the unixhost:

```
cd / ; find . -print | cpio -ocv | rsh unixhost -l username
dd of=/dev/rmt1
```

Then, we booted the second machine, a Dell 486/33SL with 8Mb and a 250Mb hard drive. We started with the Slackware boot diskette, and inserted the modified root diskette when prompted for the root diskette.

We used the convenient Slackware setup program to format and setup the root and swap partitions identically to the first machine. Specifically, we made the root partition and the swap partition the same partition letter as the first machine, /dev/hda3 and /dev/hda2 respectively. This ensured that the root filesystem and swap partition would be the "cloned" kernel expected them to be. Then we exited setup, started the network, mounted the new root filesystem, and restored the filesystem from the unixhost tape drive.

```
#ifconfig tr0 167.195.160.6 netmask 255.255.255.0 broadcast 167.195.160.255
```

```
#route add -net 167.195.160 netmask 255.255.255.0
#mount /dev/hda3 /mnt
#cd /mnt ; rsh unixhost -l username "dd if=/dev/rmt1" | cpio -icv
```

We shut down the clone, removed it from the LAN, and rebooted. We then changed the IP number and network on the clone machine as shown in Figure 1 (full startup files for both machines are detailed in Figure 4).

We set the clone up three feet away from the first machine, using a directly connected null-modem cable, in order to test the PPP without the complications of ISDN or geographic distance. We connected a hub and a PC to the cloned machine to simulate conditions at the remote site. This was where routed caused everything to fall apart. As soon as we replaced routed with gated, all was well.

We moved the clone to the remote site, connected the ISDN terminal adaptors, auto-dialed between them, then plugged them into both of our new Linux routers. Of course, one of the ISDN modems turned out to be configured for the wrong type of channel. As soon as that was corrected, everything worked perfectly.

The Future

Since Linux is extremely well documented, and supports a wide variety of protocols and networking technologies (including Novell's IPX/SPX), it's hard to see any reason to stop using it. During the project, we were consistently amazed at how many well-written and detailed "howto" guides there were. Some of them are truly better than their commercial counterparts.

One day we may have to replace the Linux routers with dedicated hardware routers -- but it's hard to say when and if this will happen. If the system doesn't break, we won't fix it.

A measurement tool to tell us what kind of routing volume we're getting and with what kind of latency would be nice. Right now, our only way of measuring traffic is to check ifconfig output and do an average per sample period. We could just wait until the users complain that the network's too slow, but this is probably a method to be avoided if at all possible. We're looking into coding a program to automate the router monitoring process.

We've managed to save ourselves the startup costs associated with hardware routers. And, due to the nonproprietary nature of Linux, we've left ourselves "open" to a very flexible future.

About the Author

Jonathan Feldman works with UNIX and NetWare at the Chatham County Government in Savannah, Georgia. He likes to keep things simple so that even he can understand them. When he is not chasing around with his 18-month-old son, he likes to write, grow roses with his lovely wife, and play guitar with his bare feet. He is reachable via email at jonathan@co.chatham.ga.us.

Figure 1: Linux Router Network Diagram

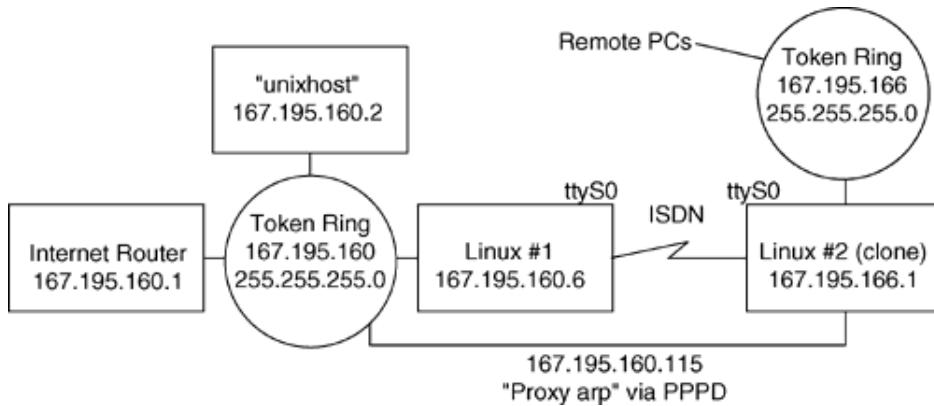


Figure 2: RIP Gated configuration

```
/etc/gated.conf:
#
tracefile "/var/tmp/gated.log" replace size 100k files 2;
traceoptions general route kernel update icmp;

rip yes ;
```

Figure 3: Token-Ring proxy arp pppd kludge

```
/etc/ppp/ip-up:
#!/bin/sh
# pppd calls this file when TCP/IP comes up. It supplies parameters:
# interface-name tty-device speed local-ip-address remote-ip-address
/etc/ppp/arp -d 167.195.160.115
# remove ppp's Ethernet hardware type
/etc/ppp/arp -t tr -s 167.195.160.115 00:60:8C:24:A2:BD pub
# add "by hand"

# Use your own IP addresses and hardware addresses, please!
# you can get your own hardware address ##:##:## from ifconfig's output
exit

/etc/ppp/ip-down:
# called by pppd when ip goes down. Same parameters supplied as ip-up.
# For some reason, with ip-up kludge, gateway daemon needs to be restarted
# when the link goes down...
/sbin/gdc restart
```

Figure 4: Startup files for Linux routers

```
/etc/rc.local startup script, both machines:
#! /bin/sh
### "router" stuff 6/95 jf
setserial /dev/ttyS0 spd_hi
sh /etc/ppp/ppplogin

/etc/rc.inet2 modifications, both machines:
#! /bin/sh
...
```

```

if [ -f ${NET}/gated ]
then
echo -n " gated"
${NET}/gated
fi
...

/etc/ppp/ppplogin script, local machine:
#!/bin/sh
## Token-ring with proxyarp
/usr/lib/ppp/pppd crtscts netmask 255.255.255.0 proxyarp passive
:167.195.160.115 /dev/ttys0 38400
echo "Point-to-point network started."
sleep 3
exit

/etc/ppp/ppplogin script, remote machine:
#!/bin/sh
/usr/lib/ppp/pppd /dev/ttys0 38400 netmask 255.255.255.0 passive defaultroute
echo "Point-to-point network started."
sleep 3
exit

/etc/ppp/rc.inet1 modifications, local machine:
...
# Edit for your setup.  Please use your own IP addresses and networks!
IPADDR="167.195.160.6"
NETMASK="255.255.255.0"
NETWORK="167.195.160.0"
BROADCAST="167.195.160.255"
GATEWAY="167.195.160.1" # Internet router
...
/sbin/route add default gw ${GATEWAY} metric 1
...

/etc/ppp/rc.inet1 modifications, remote machine:
...
##### JF: CHANGED eth0 to tr0, for obvious reasons ;-) , got rid of GATEWAY
##### stuff because ppp does its own default route.
# Edit for your setup.
IPADDR="167.195.166.1"
NETMASK="255.255.255.0"
NETWORK="167.195.166.0"
BROADCAST="167.195.166.255"
# GATEWAY="167.195.160.6"          # ppp will handle gateway
...
# /sbin/route add default gw ${GATEWAY} metric 1
...

```
