ProxyARP Subnetting HOWTO

Bob Edwards

Robert.Edwards@anu.edu.au

v2.0, 27 August 2000

This HOWTO discusses using Proxy Address Resolution Protocol (ARP) with subnetting in order to make a small network of machines visible on another Internet Protocol (IP) subnet (I call it sub−subnetting). This makes all the machines on the local network (network 0 from now on) appear as if they are connected to the main network (network 1).

This is only relevent if all machines are connected by Ethernet or ether devices (ie. it won't work for SLIP/PPP/CSLIP etc.)

Table of Contents

1. Acknowledgements

This document, and my Proxy ARP implementation could not have been made possible without the help of:

- Andrew Tridgell, who implemented the subnetting options for arp in Linux, and who personally assisted me in getting it working
- the Proxy−ARP mini−HOWTO, by Al Longyear
- the Multiple−Ethernet mini−HOWTO, by Don Becker
- the **arp**(8) source code and man page by Fred N. van Kempen and Bernd Eckenfels

---

2. Why use Proxy ARP with subnetting?

The applications for using Proxy ARP with subnetting are fairly specific.

In my case, I had a wireless Ethernet card that plugs into an 8−bit ISA slot. I wanted to use this card to provide connectivity for a number of machines at once. Being an ISA card, I could use it on a Linux machine, after I had written an appropriate device driver for it − this is the subject of another document. From here, it was only necessary to add a second Ethernet interface to the Linux machine and then use some mechanism to join the two networks together.

For the purposes of discussion, let network 0 be the local Ethernet connected to the Linux box via an NE−2000 clone Ethernet interface on eth0. Network 1 is the main network connected via the wireless Ethernet card on eth1. Machine A is the Linux box with both interfaces. Machine B is any TCP/IP machine on network 0 and machine C is likewise on network 1.

Normally, to provide the connectivity, I would have done one of the following:

- Used the IP−Bridge software (see the Bridge mini−HOWTO) to bridge the traffic between the two network interfaces. Unfortunately, the wireless Ethernet interface cannot be put into "Promiscuous" mode (ie. it can't see all packets on network 1). This is mainly due to the lower bandwidth of the wireless Ethernet (2MBit/sec) meaning that we don't want to carry any traffic not specifically destined to another wireless Ethernet machine − in our case machine A − or broadcasts. Also, bridging is rather CPU intensive!
- Alternatively, use subnets and an IP−router to pass packets between the two networks (see the IP−Subnetworking mini−HOWTO). This is a protocol specific solution, where the Linux kernel can handle the Internet Protocol (IP) packets, but other protocols (such as AppleTalk) need extra software to route. This also requires the allocation of a new IP subnet (network) number, which is not always an option.

In my case, getting a new subnet (network) number was not an option, so I wanted a solution that allowed all the machines on network 0 to appear as if they were on network 1. This is where Proxy ARP comes in. Other solutions are used to connect other (non−IP) protocols, such as netatalk to provide AppleTalk routing.

3. How Proxy ARP with subnetting works

The Proxy ARP is actually only used to get packets from network 1 to network 0. To get packets back the other way, the normal IP routing functionality is employed.

In my case, network 1 has an 8−bit subnet mask (255.255.255.0). I have chosen the subnet mask for network 0 to be 4−bit (255.255.255.240), allowing 14 IP nodes on network 0 (2 ^ 4 = 16, less two for the all zeros and all ones cases). Note that any size of subnet mask up to, but not including, the size of the mask of the other network is allowable here (eg. 2, 3, 4, 5, 6 or 7 bits in this case − for one bit, just use normal Proxy ARP!)

All the IP numbers for network 0 (16 in total) appear in network 1 as a subset. Note that it is very important, in this case, not to allow any machine connected directly to network 1 to have an IP number in this range! In my case, I have "reserved" the IP numbers of network 1 ending in 64 .. 79 for network 0. In this case, the IP numbers ending in 64 and 79 can't actually be used by nodes − 79 is the broadcast address for network 0.

Machine A is allocated two IP numbers, one within the network 0 range for it's real Ethernet interface (eth0) and the other within the network 1 range, but outside of the network 0 range, for the wireless Ethernet interface (eth1).

Say machine C (on network 1) wants to send a packet to machine B (on network 0). Because the IP number of machine B makes it look to machine C as though it is on the same physical network, machine C will use the Address Resolution Protocol (ARP) to send a broadcast message on network 1 requesting the machine with the IP number of machine B to respond with it's hardware (Ethernet or MAC layer) address. Machine B won't see this request, as it isn't actually on network 1, but machine A, on both networks, will see it.

The first bit of magic now happens as the Linux kernel arp code on machine A, with a properly configured Proxy ARP with subnetting entry, determines that the ARP request has come in on the network 1 interface (eth1) and that the IP number being ARP'd for is in the subnet range for network 0. Machine A then sends it's own hardware (Ethernet) address back to machine C as an ARP response packet.

Machine C then updates it's ARP cache with an entry for machine B, but with the hardware (Ethernet) address of machine A (in this case, the wireless Ethernet interface). Machine C can now send the packet for machine B to this hardware (Ethernet) address, and machine A receives it.

Machine A notices that the destination IP number in the packet is that of machine B, not itself. Machine A's Linux kernel IP routing code attempts to forward the packet to machine B by looking at it's routing tables to determine which interface contains the network number for machine B. However, the IP number for machine B is valid for both the network 0 interface (eth0), and for the network 1 interface (eth1).

At this point, something else clever happens. Because the subnet mask for the network 0 interface has more 1 bits (it is more specific) than the subnet mask for the network 1 interface, the Linux kernel routing code will match the IP number for machine B to the network 0 interface, and not keep looking for the potential match with the network 1 interface (the one the packet came in on).

Now machine A needs to find out the "real" hardware (Ethernet) address for machine B (assuming that it doesn't already have it in the ARP cache). Machine A uses an ARP request, but this time the Linux kernel arp code notes that the request isn't coming from the network 1 interface (eth1), and so doesn't respond with the Proxy address of eth1. Instead, it sends the ARP request on the network 0 interface (eth0), where machine B will see it and respond with it's own (real) hardware (Ethernet) address. Now machine A can send the packet (from machine C) onto machine B.

Machine B gets the packet from machine C (via machine A) and then wants to send back a response. This time, machine B notices that machine C in on a different subnet (machine B's subnet mask of 255.255.255.240 excludes all machines not in the network 0 IP address range). Machine B is setup with a "default" route to machine A's network 0 IP number and sends the packet to machine A. This time, machine A's Linux kernel routing code determines the destination IP number (of machine C) as being on network 1 and sends the packet onto machine C via Ethernet interface eth1.

Similar (less complicated) things occur for packets originating from and destined to machine A from other machines on either of the two networks.

Similarly, it should be obvious that if another machine (D) on network 0 ARP's for machine B, machine A will receive the ARP request on it's network 0 interface (eth0) and won't respond to the request as it is set up to only Proxy on it's network 1 interface (eth1).

Note also that all of machines B and C (and D) are not required to do anything unusual, IP−wise. In my case, there is a mixture of Suns, Macs and PC/Windoze 95 machines on network 0 all connecting through Linux machine A to the rest of the world.

Finally, note that once the hardware (Ethernet) addresses are discovered by each of machines A, B, C (and D), they are placed in the ARP cache and subsequent packet transfers occur without the ARP overhead. The ARP caches normally expire entries after 5 minutes of non−activity.

---

4. Setting up Proxy ARP with subnetting

I set up Proxy ARP with subnetting on a Linux kernel version 2.0.30 machine, but I am told that the code works right back to some kernel version in the 1.2.x era.

The first thing to note is that the ARP code is in two parts: the part inside the kernel that sends and receives ARP requests and responses and updates the ARP cache etc.; and other part is the **arp**(8) command that allows the super user to modify the ARP cache manually and anyone to examine it.

The first problem I had was that the **arp**(8) command that came with my Slackware 3.1 distribution was ancient (1994 era!!!) and didn't communicate with the kernel arp code correctly at all (mainly evidenced by the strange output that it gave for "**arp** −a").

The **arp**(8) command in "net−tools−1.33a" available from a variety of places, including (from the `README` file that came with it) ftp.linux.org.uk:/pub/linux/Networking/base/ works properly and includes new man pages that explain stuff a lot better than the older arp(8) man page.

Armed with a decent **arp**(8) command, all the changes I made were in the `/etc/rc.d/rc.inet1` script (for Slackware − probably different for other flavours). First of all, we need to change the broadcast address, network number and netmask of eth0:

```
NETMASK=255.255.255.240 # for a 4-bit host part
NETWORK=x.y.z.64        # our new network number (replace x.y.z with your net)
BROADCAST=x.y.z.79      # in my case
```

Then a line needs to be added to configure the second Ethernet port (after any module loading that might be required to load the driver code):

```
/sbin/ifconfig eth1 (name on net 1) broadcast (x.y.z.255) netmask 255.255.255.0
```

Then we add a route for the new interface:

```
/sbin/route add -net (x.y.z.0) netmask 255.255.255.0
```

And you will probably need to change the default gateway to the one for network 1.

At this point, it is appropriate to add the Proxy ARP entry:

```
/sbin/arp -i eth1 -Ds ${NETWORK} eth1 netmask ${NETMASK} pub
```

This tells ARP to add a static entry (the s) to the cache for network ${NETWORK}. The −D tells ARP to use the same hardware address as interface eth1 (the second eth1), thus saving us from having to look up the hardware address for eth1 and hardcoding it in. The netmask option tells ARP that we want to use subnetting (ie. Proxy for all (IP number) & ${NETMASK} == ${NETWORK} & ${NETMASK}). The pub option tells ARP to publish this ARP entry, ie. it is a Proxy entry, so respond on behalf of these IP numbers. The −i eth1 option tells ARP to only respond to requests that come in on interface eth1.

Hopefully, at this point, when the machine is rebooted, all the machines on network 0 will appear to be on network 1. You can check that the Proxy ARP with subnetting entry has been correctly installed on machine A. On my machine (names changed to protect the innocent) it is:

```
bash$ /sbin/arp -an
Address                 HWtype  HWaddress           Flags Mask            Iface
x.y.z.1                 ether   00:00:0C:13:6F:17   C     *               eth1
x.y.z.65                ether   00:40:05:49:77:01   C     *               eth0
x.y.z.67                ether   08:00:20:0B:79:47   C     *               eth0
x.y.z.5                 ether   00:00:3B:80:18:E5   C     *               eth1
x.y.z.64                ether   00:40:96:20:CD:D2   CMP   255.255.255.240 eth1
```

Alternatively, you can examine the `/proc/net/arp` file with eg. **cat**(1).

The last line is the proxy entry for the subnet. The CMP flags indicate that it is a static (Manually entered) entry and that it is to be Published. The entry is only going to reply to ARP requests on eth1 where the requested IP number, once masked, matches the network number, also masked. Note that **arp**(8) has automatically determined the hardware address of eth1 and inserted this for the address to use (the −Ds option).

Likewise, it is probably prudent to check that the routing table has been set up correctly. Here is mine (again, the names are changed to protect the innocent):

```
#/bin/netstat -rn
Kernel routing table
Destination     Gateway         Genmask         Flags Metric Ref Use    Iface
x.y.z.64        0.0.0.0         255.255.255.240 U     0      0       71 eth0
x.y.z.0         0.0.0.0         255.255.255.0   U     0      0      389 eth1
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        7 lo
0.0.0.0         x.y.z.1         0.0.0.0         UG    1      0      573 eth1
```

Alternatively, you can examine the `/proc/net/route` file with eg. **cat**(1).

Note that the first entry is a proper subset of the second, but the routing table has ranked them in netmask order, so the eth0 entry will be checked before the eth1 entry.

5. Other alternatives to Proxy ARP with subnetting

There are several other alternatives to using Proxy ARP with subnetting in this situation, apart from the ones mentioned about (bridging and straight routing):

- IP−Masquerading (see the IP−Masquerade mini−HOWTO), in which network 0 is "hidden" behind machine A from the rest of the Internet. As machines on network 0 attempt to connect outside through machine A, it re−addresses the source address and port number of the packets and makes them look like they are coming from itself, rather than from the machine on the hidden network 0. This is an elegant solution, although it prevents any machine on network 1 from initiating a connection to any machine on network 0, as the machines on network 0 effectively don't exist outside of network 0. This effectively increases security of the machines on network 0, but is also means that servers on network 1 cannot check the identity of clients on network 0 using IP numbers (eg. NFS servers use IP hostnames for access to mountable file systems).
- Another option is IP in IP tunneling, which isn't supported on all platforms (such as Macs and Windoze machines) so I opted not to go this way.
- Use Proxy ARP without subnetting. This is certainly possible, it just means that a separate entry needs to be created for each machine on network 0, instead of a single entry for all machines (current and future) on network 0.
- Possibly IP Aliasing might also be useful here, but I haven't looked into this at all.

---

6. Other Applications of Proxy ARP with subnetting

There is only one other application that I know about that uses Proxy ARP with subnetting, also here at the Australian National University. It is the one that Andrew Tridgell originally wrote the subnetting extensions to Proxy ARP for. However, Andrew reliably informs me that there are, in fact, several other sites around the world using it as well (I don't have any details).

The other A.N.U. application involves a teaching lab set up to teach students how to configure machines to use TCP/IP, including setting up the gateway. The network used is a Class C network, and Andrew needed to "subnet" it for security, traffic control and the educational reason mentioned above. He did this using Proxy ARP, and then decided that a single entry in the ARP cache for the whole subnet would be faster and cleaner than one for each host on the subnet. Voila...Proxy ARP with subnetting!

7. Copying conditions