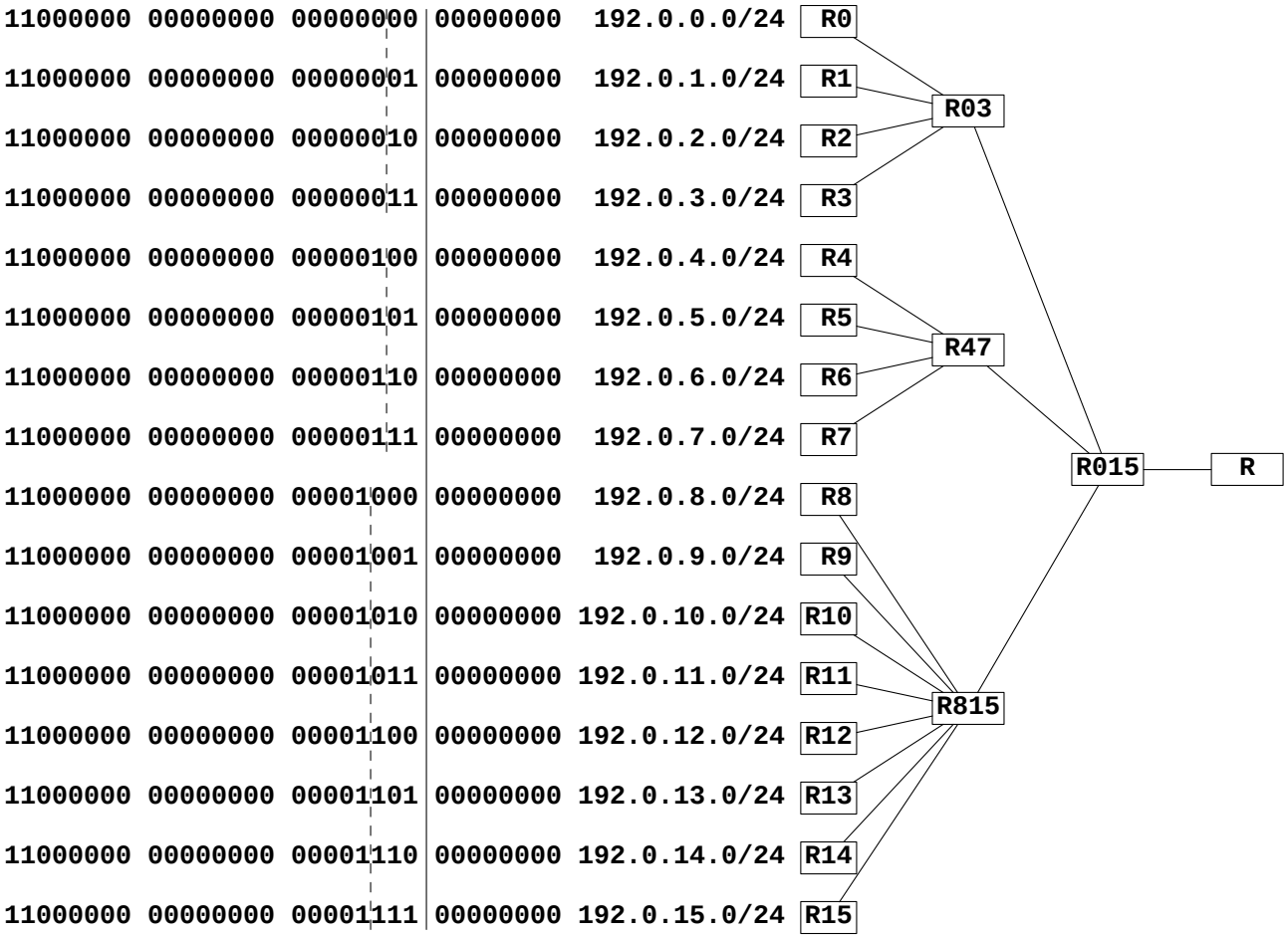
# Fall 2005 Handout 5

**CIDR: Classless Inter-Domain Routing (“cider”)**




```
11000000 00000000 00000000 00000000  192.0.0.0/22
11000000 00000000 00000100 00000000  192.0.4.0/22  R015's routing table
11000000 00000000 00001000 00000000  192.0.8.0/21

11000000 00000000 00000000 00000000  192.0.0.0/20  R's routing table
```

This diagram shows routers and the direct connections between them. We do not show the LAN(s) to which each router is attached, and the non-router hosts on the LANs. We also do not show the interface

numbers. Assume that the interfaces leading leftwards from each router are numbered from top to bottom, starting at zero.

In the vertical column are sixteen Class C networks. (In fact, **192** is the first Class C byte; see Handout 1, p. 13.) Their addresses are shown in binary and dotted decimal. Because they are Class C, each address has 24 network bits and 8 host bits. We show this by writing the **/24**'s.

The undashed vertical lines separate the network bits from the host bits. The dashed vertical lines show that the addresses of the first four networks start with the same 22 bits. Ditto for the next four networks. The addresses of the last eight networks start with the same 21 bits.

Router **R03** is connected to the four routers **R0**, **R1**, **R2**, and **R3**. Its routing table therefore contains four entries telling how to get to the network connected to each router

```
192.0.0.0/24 interface0
192.0.1.0/24 interface1
192.0.2.0/24 interface2
192.0.3.0/24 interface3
```

You would think that router **R015**'s routing table would also contain four entries for these destinations. But **R015** has only one entry, *summarizing* the above four:

```
192.0.0.0/22 interface0
```

**R015** can summarize the four entries into one because they agree in their first bits, and they all lie in the same direction from **R015**: through **R015**'s **interface0** to router **R03**. Summarization is also called *route aggregation* or *supernetting.* See pp. 32−33; RFC's 1517−1520.

Similarly, router **R47** has four entries telling how to get to these destinations:

```
192.0.4.0/24 interface0
192.0.5.0/24 interface1
192.0.6.0/24 interface2
192.0.7.0/24 interface3
```

Router **R015** can summarize them into the single entry

```
192.0.4.0/22 interface0
```

Finally, router **R815** has eight entries telling how to get to these destinations:

```
192.0.8.0/24
192.0.9.0/24
192.0.10.0/24
192.0.11.0/24
192.0.12.0/24
192.0.13.0/24
192.0.14.0/24
192.0.15.0/24
```

Router **R015** can summarize them into the single entry

```
192.0.8.0/21 interface0
```

There's more. We have seen that router **R015** has the following three entries. I'll write them in binary, too:

```
11000000 00000000 00000000 00000000 == 192.0.0.0/22
11000000 00000000 00000100 00000000 == 192.0.4.0/22
11000000 00000000 00001000 00000000 == 192.0.8.0/21
```

They start with the same 20 bits, so router **R** can summarize them into the single entry

```
11000000 00000000 00000000 00000000 == (192.0.0.0/20)
```

A router can summarize only when the networks addresses agree in their leading bits and lie in the same direction from that router. For this reason, an attempt is being made to assign consecutive addresses to networks that are geographically near each other. For example, all newly assigned network addresses in Europe lie in the range **194.0.0.0** to **195.255.255.255**. In the United States, the American Registry for Internet Numbers **http://www.arin.net/** encourages people to trade in older, non-consecutive network addresses for new ones.

Suppose a datagram addressed to **192.0.6.2** arrives at router **R**. The first 20 bits of **192.0.6.2** match the entry for interface 0 in **R**'s routing table, so the datagram will be sent through that interface to **R015**. The first 22 bits of **192.0.6.2** match the entry for interface 1 in **R015**'s routing table, routing table, so the datagram will be sent through that interface to **R47**. The first 24 bits of **192.0.6.2** match the entry for interface 2 in **R47**'s routing table, so the datagram will be sent through that interface to **R6**.

**Private IP addresses and NAT**

A network connected to the outside world must use IP addresses issued by an ISP, which is a hassle and costs money. A network that is not connected to the outside world is free to use any IP addresses you want. Of course, you'll have to change them if you ever connect to the Internet—they might duplicate someone else's addresses.

For this reason, RFC 1918 set aside three blocks of IP addresses for internal use only:

```
Class A: 10.0.0.0/8     i.e., 10.0.0.0    to 10.255.255.255
Class B: 172.16.0.0/12  i.e., 172.16.0.0  to 172.31.255.255
Class C: 192.168.0.0/16 i.e., 192.168.0.0 to 192.168.255.255
```

For example, the Mac OSX I usually use at NYU is **ftcg5faculty2.edlab.its.nyu.edu** and has the RFC 1918 IP address **192.168.20.196**. But packets from hosts with RFC 1918 addresses cannot be routed outside of NYU. That's why I can't **ping** or **traceroute** to an outside host from the Mac.

When I point the Mac's web browser at

```
http://www.whatismyip.com/
```

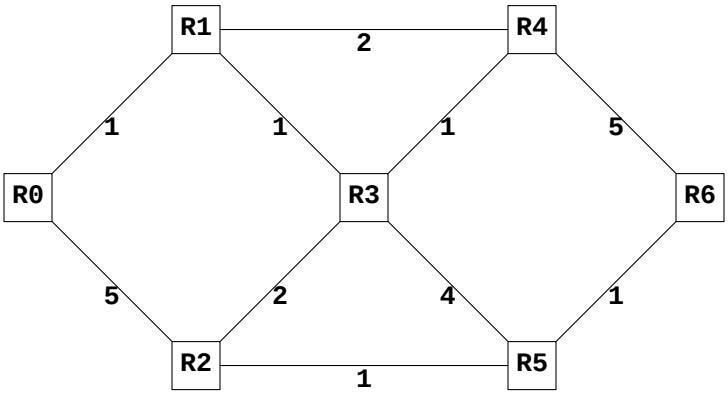
I am told that the Mac's address is **128.122.128.41** because of a NAT translation box (pp. 85−88) at NYU. In fact, several of the NYU Mac's with RFC 1918 addresses are told by **whatismyip** that their address is **128.122.128.41**. The name of **128.122.128.41** is **PROXY-DEV.NS.NYU.EDU**.

A NAT box can also perform ''port address translation'' (PAT). TCP or UDP packets coming into NYU that are addressed to port 10000 of **128.122.128.41** can be routed to my Mac; those addressed to port 10001 of **128.122.128.41** can be routed to another Mac. This is how you survive if your organization has more hosts than IP addresses, at least if all the hosts will not be in use at the same time.

**Flooding**

Used mostly by the military and by wireless networks.

**Static routing**




Here is an algorithm for finding the shortest path between two routers. The adjacency table has a zero for every pair of routers that are not connected directly to each other. It also has zero for every router's distance from itself.

| | R0 | R1 | R2 | R3 | R4 | R5 | R6 |
|---|---|---|---|---|---|---|---|
| R0 | 0 | 1 | 5 | 0 | 0 | 0 | 0 |
| R1 | 1 | 0 | 0 | 1 | 2 | 0 | 0 |
| R2 | 5 | 0 | 0 | 2 | 0 | 1 | 0 |
| R3 | 0 | 1 | 2 | 0 | 1 | 4 | 0 |
| R4 | 0 | 2 | 0 | 1 | 0 | 0 | 5 |
| R5 | 0 | 0 | 1 | 4 | 0 | 0 | 1 |
| R6 | 0 | 0 | 0 | 0 | 5 | 1 | 0 |

The shortest path algorithm is used by OSPF. For simplicity, we assume that there is only at most one direct link between any two routers.

—On the Web at
**http://i5.nyu.edu/~mm64/x52.9547/src/shortest.c**

```
 1 #include <stdio.h>
 2 #include <stdlib.h>   /* for exit and EXIT_SUCCESS */
 3 #include <limits.h>   /* for INT_MAX */
 4
 5 #define N 7   /* the number of nodes */
 6 int distance[N][N] = {
 7     {0, 1, 5, 0, 0, 0, 0},
 8     {1, 0, 0, 1, 2, 0, 0},
 9     {5, 0, 0, 2, 0, 1, 0},
10     {0, 1, 2, 0, 1, 4, 0},
11     {0, 2, 0, 1, 0, 0, 5},
12     {0, 0, 1, 4, 0, 0, 1},
13     {0, 0, 0, 0, 5, 1, 0}
14 };
15
16 void shortest_path(int begin, int end);
17
18 int main(int argc, char **argv)
19 {
20     shortest_path(0, 6);
21     return EXIT_SUCCESS;
22 }
```

```
23
24 void shortest_path(int begin, int end)
25 {
26     struct router {
27         int predecessor;
28         int length;
29         enum {permanent, tentative} label;
30     };
31     struct router a[N];
32
33     int i;
34     int current;
35     int new_distance;
36     int min;
37
38     /* Initialize the state of each node. */
39     for (i = 0; i < N; ++i) {
40         a[i].predecessor = -1;
41         a[i].length = INT_MAX;
42         a[i].label = tentative;
43     }
44
45     current = begin;
46     a[current].length = 0;
47     a[current].label = permanent;
48
49     while (current != end) {
50         printf("When current == R%d, we are probing ", current);
51
52         /*
53         Probe each tentatively labeled node i adjacent to the current
54         node.  Compute the distance from the beginning to i via the
55         current node.  If this new distance is the shortest distance
56         we've seen so far from the beginning to i, record the new
57         distance as the tentative distance from the beginning to i.
58         */
59         for (i = 0; i < N; ++i) {
60             if (a[i].label == tentative && distance[current][i] > 0) {
61                 printf(" R%d", i);
62
63                 new_distance =
64                     a[current].length + distance[current][i];
65
66                 if (new_distance < a[i].length) {
67                     a[i].length = new_distance;
68                     a[i].predecessor = current;
69                 }
70             }
71         }
72
73         printf("\n");
74
75         /*
76         Let the new current node be the tentatively labeled node with
```

```
 77          the smallest distance to the end.
 78          */
 79          min = INT_MAX;
 80          current = 0;
 81
 82          for (i = 0; i < N; ++i) {
 83              if (a[i].label == tentative && a[i].length < min) {
 84                  min = a[i].length;
 85                  current = i;
 86              }
 87          }
 88
 89          if (min == INT_MAX) {
 90              printf("There is no path from R%d to R%d.\n",
 91                  begin, end);
 92              return;
 93          }
 94
 95          a[current].label = permanent;
 96      }
 97
 98      printf("\nThe shortest path from R%d to R%d, listed backwards, is\n",
 99          begin, end);
100
101      for (current = end;; current = a[current].predecessor) {
102          printf("R%d\n", current);
103          if (current == begin) {
104              break;
105          }
106      }
107  }
```

```
1$ cd ~mm64/public_html/x52.9547/src
2$ gcc -o ~/bin/shortest shortest.c
3$ ls -l ~/bin/shortest

4$ shortest
When current == R0, we are probing  R1 R2
When current == R1, we are probing  R3 R4
When current == R3, we are probing  R2 R4 R5
When current == R4, we are probing  R6
When current == R2, we are probing  R5
When current == R5, we are probing  R6

The shortest path from R0 to R6, listed backwards, is
R6
R5
R2
R3
R1
R0
```

(1) When the current node is **R0**, we assign a tentative length of 1 to **R1** and 5 to **R2**.

(2) When the current node is **R1**, we assign a tentative length of 2 to **R3** and 3 to **R4**.

(3) When the current node is **R3**, we assign a tentative length of 6 to **R5**. The tentative length of **R4** is already 3, so we don't change it. We change the tentative length of **R2** from 5 to 4.

Note that the path followed by the ''current node'' is not necessarily the same as the shortest path from the beginning to the end. For example, the current node is momentarily **R4**, but **R4** is not on the final path.

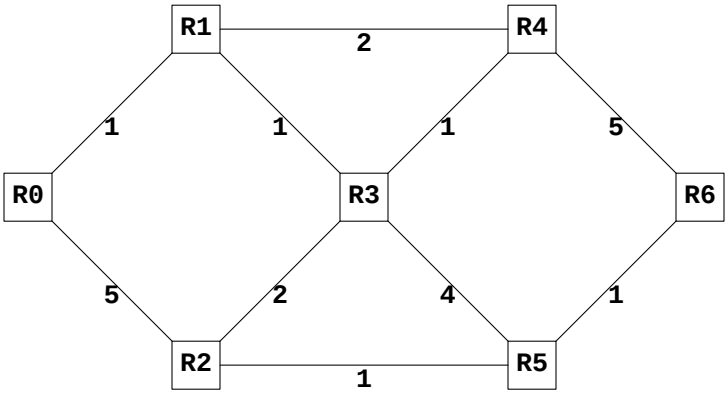
**RIP: Routing Information Protocol**

RIP is a *distance vector* routing algorithm. See pp. 179−184; RFC's 1058 (RIP Version 1) and 2453 (RIP Version 2).

Suppose that sending a packet directly from one router to another takes one unit of time. Or perhaps it costs one unit of money. Or perhaps it introduces errors that cause one unit of pain.

We will say that the direct distance between the two routers is 1. (''Direct'' means that there are no intervening routers between them.) RIP always believes that the distance between routers is 1, but a smarter distance vector protocol might know that the distances can be greater than one.

Here's the graph we used before:




**Propagate information about R0 from router to router**

Each router broadcasts to its immediate neighbors, but does not send information to anyone else. The broadcasts are repeated every 30 seconds. We assume for simplicity that they all happen at the same time, but they really don't.

(1) **R0** finds out how far it is from its immediate neighbors **R1** and **R2** and broadcasts the results to them.

(2) **R1** hears from **R0** that **R1** is one unit away from **R0** and broadcasts this information to all its immediate neighbors **R4**, **R3**, and **R0**. (Of course, **R0** ignores this broadcast coming back from **R1**.)

(3) **R2** hears from **R0** that **R2** is 5 units away from **R0** and broadcasts this information to all its immediate neighbors **R3**, **R5**, and **R0**. (Of course, **R0** ignores this broadcast coming back from **R0**.)

(4) **R3** hears from **R2** that **R2** is 5 units away from **R0**. **R3** concludes that **R3** is 7 units away from **R0** via a path whose first step is **R2**.

But more or less simultaneously, **R3** also hears from **R2** that **R2** is 1 unit away from **R0**. **R3** concludes that **R3** is only 2 units away from **R0** via a path whose first step is **R1**. **R3** then forgets about the longer path from **R3** to **R0** whose first step is **R2**.

**R3** broadcasts to its immediate neighbors **R1**, **R2**, **R4**, and **R5** that **R3** is 2 units away from **R0**. You can see why this broadcast will be ignored by **R1**: **R1** is only 2 units away from **R0**. But this broadcast will be avidly absorbed by **R2**. **R2** knows that it is 5 units away from **R0** along a direct path, but is glad to find out that it is 4 units away from **R0** along a path whose first step is **R3**. **R3** then forgets about the longer path from **R3** directly to **R0**.

**Good news travels fast over RIP.**

Suppose a link of length 2 comes up between **R0** and **R5**. **R5** will hear the good news along the new link from **R0** and will be delighted to know that it is only 2 units from **R0**. **R2**, **R3**, and **R6** will then hear from **R5** that **R5** is 2 units away from **R0**. **R3** will ignore this broadcast because **R3** is already 2 units away from **R0**. But **R2** will conclude that **R2** is now only 3 units away from **R0** along a path whose first step is **R5**, and **R6** will conclude that **R6** is now only 3 units away from **R0** along a path whose first step is **R5**.

It will take routers two 30-second intervals to readjust their distances from **R0**.

**Counting to infinity: regain equilibrium after 10½ minutes**

Bad news travels slowly over RIP. Let's use a much simpler group of routers:


**R0** should assure **R1** every 30 seconds that it is still there. If 180 seconds passes without **R1** hearing from **R0**, **R1** will assume that **R0** is down, or that the link between them is down.

RIP can count up only to 15. It thinks that any greater distance is infinity. It will therefore take three minutes plus 15 periods of 30 seconds = 10½ minutes for **R1** to realize that it has no way to reach **R0**. During those three minutes, a packet addressed to **R0** that finds itself at either **R1** or **R2** would go back and forth between the latter two routers until its time to live counted down to zero.

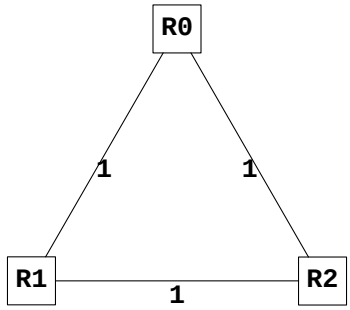
RFC 1058, pp. 13−14: ''If a [router such as the above **R0**] becomes completely inaccessible, we want counting to infinity [i.e., 16] to be stopped as soon as possible. Infinity must be large enough so that no real route is that big. But it shouldn't be any bigger than required. Thus the choice of infinity is a tradeoff between network size and speed of convergence in case counting to infinity happens.'' A small infinity will make the network converge more rapidly, but it will limit the size of the network.

**Split horizon: regain equilibrium after 3 minutes**

In actual RIP, a router does not broadcast information back to the source from which it heard it. In the above diagram, **R2** hears from **R1** that **R1** is 1 unit away from **R0**. **R2** then concludes that **R2** is 2 units away from **R0** via a path whose first step is **R1**. But **R2** does not broadcast (''advertise'') this conclusion to **R1**. If the link between **r0** and **R1** goes down, **R1** will resign itself to this fact after 3 minutes.

**Split horizon with poison reverse: p. 183**

Will the above split horizon scheme always prevent a count to infinity? If **R0** goes down, **R1** and **R2** will perform their own count to infinity. With split horizon and poison reverse, this will be prevented and we will regain equilibrium after 3 minutes.




RFC 1058, pp. 14−15: ''However, poisoned reverse does have a disadvantage: it increases the size of the routing messages. Consider the case of a campus backbone connecting a number of different buildings. In each building, there is a gateway connecting the backbone to a local network. Consider what routing updates those gateways should broadcast on the backbone network. All that the rest of the network really needs to know about each gateway is what local networks it is connected to. Using simple split horizon,

only those routes would appear in update messages sent by the gateway to the backbone network. If split horizon with poisoned reverse is used, the gateway must mention all routes that it learns from the backbone, with metrics of 16. If the system is large, this can result in a large update message, almost all of whose entries indicate unreachable networks.''

**Triggered updates**

Will the above split horizon with poison reverse always prevent a count to infinity?

**The DNS resolver**

Given a host's fully qualified domain name **`i5.nyu.edu`**, how can you find the corresponding IP address **`128.122.253.152`**? Conversely, given an IP address, how can you find the corresponding fully qualified domain name?

The file **`nsswitch.conf`** (Handout 1, p. 24) tells us to look in a file first. See pp. 52−54.

```
1$ awk '$1 == "hosts:"' /etc/nsswitch.conf
hosts:      files dns
```

In this case, **`files`** means the file **`/etc/hosts`**:

```
2$ cat -n /etc/hosts | more
     1 #
     2 # Internet host table
     3 #
     4 127.0.0.1          localhost
     5 ## 128.122.253.152 i6.home.nyu.edu loghost
     6 128.122.253.152    i5.nyu.edu i5 loghost
```

If we can't find what we're looking for in **`/etc/hosts`**, **`nsswitch.conf`** tells us to try DNS second. The Domain Name System is a database of fully qualified domain names and IP addresses. Each record of the database is called a Resource Record (RR, pp. 216−217). The database is resides in thousands or millions of programs. These programs (and the hosts on which they run) are called *domain name servers.*

Instead of talking directly to a domain name server, you use a package of functions called a *resolver.* See pp. 60, 206, and **`resolver`**(3resolv). If there is no server running on your machine, your resolver will communicate with a server running on another machine via UDP.

Before our host is fully booted, our resolver may not be able to talk to other hosts because UDP is not up yet. In this case, our only hope is the **`/etc/hosts`** file.

After our host is fully booted, our resolver looks at the **`/etc/resolv.conf`** file to see which domain name server(s) it should talk to. For example, there might be a domain name server running right here on our host. But disappointingly, our **`/etc/resolv.conf`** file mentions no domain name server on **`i5.nyu.edu`** (**`128.122.253.152`**):

```
3$ cat -n /etc/resolv.conf | more
     1  domain nyu.edu
     2  nameserver 128.122.253.37
     3  nameserver 128.122.128.24
     4  nameserver 128.122.253.92
     5  search nyu.edu home.nyu.edu es.its.nyu.edu
```

Our resolver therefore sends questions to the three domain name servers

```
128.122.253.37  NYUNSB.NYU.EDU
128.122.128.24  EGRESS.NYU.EDU
128.122.253.92  CMCL2.NYU.EDU
```

**The search command**

We can abbreviate the name of other machines whose names end with **nyu.edu**. See p. 59. For example, we can say

```
1$ ping i4
i4 is alive
```

instead of

```
2$ ping i4.nyu.edu
```

The **search** command in line 5 of **/etc/resolv.conf** is what lets us get away with this. Page 208 says that the **resolv.conf** file should never contain both of the commands **domain** and **search**, but ours does.

**Why isn't i5.nyu.edu running a domain name server?**

A dæmon runs as long as the machine is up, so it must be launched in the background with an ampersand. See **sh**(1) pp. 1, 3, 7 for background jobs in the Bourne Shell. For example, the **inetd** dæmon is launched in the background when we come up to runlevel 2 in Handout 4, p. 4. If our host was going to run the name server **/usr/sbin/in.named**, it would be launched in the background when we come up to runlevel 2:

```
1$ cd /etc
2$ ls -li init.d/inetsvc
392 -rwxr--r--   5 root     sys         7353 Mar 18  2004 init.d/inetsvc

3$ ls -li 'find . -inum 392 2> /dev/null'
392 -rwxr--r--   5 root     sys         7353 Mar 18  2004 ./init.d/inetsvc
392 -rwxr--r--   5 root     sys         7353 Mar 18  2004 ./rc0.d/K42inetsvc
392 -rwxr--r--   5 root     sys         7353 Mar 18  2004 ./rc1.d/K42inetsvc
392 -rwxr--r--   5 root     sys         7353 Mar 18  2004 ./rc2.d/S72inetsvc
392 -rwxr--r--   5 root     sys         7353 Mar 18  2004 ./rcS.d/K42inetsvc

4$ cat -n init.d/inetsvc | sed -n 72,80p
72  #
73  # If this machine is configured to be an Internet Domain Name System (DNS)
74  # server, run the name daemon.  Start named prior to: route add net host,
75  # to avoid dns gethostbyname timout delay for nameserver during boot.
76  #
77  if [ -f /etc/named.conf ] && [ -f /usr/sbin/in.named ]; then
78      echo 'starting internet domain name server.'
79      /usr/sbin/in.named &
80  fi
```

To see what the **-f** in the **[**single square brackets**]** means,

```
5$ man test
```

See pp. 212−216 for the **named.conf** file, which we don't have.

```
6$ ls -l /etc/named.conf /usr/sbin/in.named
/etc/named.conf: No such file or directory
-r-xr-xr-x   1 root     bin       404536 Feb 24  2004 /usr/sbin/in.named
```

**Ask the resolver a question in C via gethostbyname**

You can ask the resolver a question in C by calling the **gethostbyname** function in line 8. The resolver will send your question to a domain name server, and get an answer from the server. I picked a machine that has three IP addresses; most have only one. We saw the **inet_ntop** in line 18 in Handout 1, p. 25, line 57.

—On the Web at
**http://i5.nyu.edu/~mm64/x52.9547/src/gethostbyname.c**

```
 1 /* Convert a fully qualified domain name into its IP address(es). */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <netdb.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     struct hostent *entry = gethostbyname("indira.uu.nl");
 9     char **pp;
10
11     if (entry == NULL) {
12         fprintf(stderr, "%s: h_errno == %d\n", argv[0], h_errno);
13         return EXIT_FAILURE;
14     }
15
16     for (pp = entry->h_addr_list; *pp != NULL; ++pp) {
17         char buffer[INET6_ADDRSTRLEN];
18         inet_ntop(entry->h_addrtype, *pp, buffer, sizeof buffer);
19         printf("%s\n", buffer);
20     }
21
22     return EXIT_SUCCESS;
23 }
```

Compile with the **-lnsl** option for the ‘‘Networking Services Library’’ **/usr/lib/libnsl.a**. Type two lowercase L’s, not ones.

```
1$ man libnsl
2$ gcc -o ~/bin/gethostbyname gethostbyname.c -lnsl
3$ ls -l ~/bin/gethostbyname

4$ gethostbyname
131.211.5.1
131.211.16.1
131.211.17.1
```

If the **gethostbyname** function fails, the value of **h_errno** in the above line 12 will be one of

```
5$ cat -n /usr/include/netdb.h | sed -n 403,406p
403 #define HOST_NOT_FOUND 1 /* Authoritive Answer Host not found */
404 #define TRY_AGAIN      2 /* Non-Authoritive Host not found, or SERVERFAIL */
405 #define NO_RECOVERY    3 /* Non recoverable errors, FORMERR, REFUSED, NOTIMP */
406 #define NO_DATA        4 /* Valid name, no data record of requested type */
```

See if your machine has **hstrerror**, which is a better way of printing an error message.

```
6$ man hstrerror
```

There is an ‘‘address family’’ macro for each possible value of the **h_addrtype** field in the above line 18:

```
7$ cat -n /usr/include/sys/socket.h | sed -n '162p;186p'
162 #define AF_INET  2  /* internetwork: UDP, TCP, etc. */
186 #define AF_INET6 26 /* Internet Protocol, Version 6 */
```

The following program converts in the opposite direction, from an IP address to a fully qualified domain name. For the **in_addr_t** in the following line 8, see Handout 1, p. 14.

—On the Web at
**http://i5.nyu.edu/~mm64/x52.9547/src/gethostbyaddr.c**

```
 1 /* Convert an IP address into its fully qualified domain name. */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <netdb.h>
 5
 6 int main(int argc, char **argv)
 7 {
 8     in_addr_t ip = inet_addr("131.211.17.1");
 9     struct hostent *entry;
10
11     if ((int)ip == -1) {
12         fprintf(stderr, "%s: bad argument to inet_addr\n", argv[0]);
13         return EXIT_FAILURE;
14     }
15
16     entry = gethostbyaddr((char *)&ip, sizeof ip, AF_INET);
17     if (entry == NULL) {
18         fprintf(stderr, "%s: h_errno == %d\n", argv[0], h_errno);
19         return EXIT_FAILURE;
20     }
21
22     printf("%s\n", entry->h_name);
23     return EXIT_SUCCESS;
24 }
```

```
8$ gcc -o ~/bin/gethostbyaddr gethostbyaddr.c -lnsl
9$ ls -l ~/bin/gethostbyaddr
10$ gethostbyaddr
Indira.uu.nl
```

**Ask the resolver a question in Perl via gethostbyname**

In the above C program **gethostbyname.c**, **entry** was the following structure:

```
1$ cat -n /usr/include/netdb.h | sed -n 116,123p
   116  struct  hostent {
   117       char*h_name;/* official name of host */
   118       char**h_aliases;/* alias list */
   119       int h_addrtype;  /* host address type */
   120       int h_length;    /* length of address */
   121       char**h_addr_list;   /* list of addresses from name server */
   122  #define h_addr  h_addr_list[0]   /* address, for backward compatiblity */
   123  };
```

The **h_addr_list** in line 16 of the above C program above was the fifth field of the structure. This field

is an array of IP addresses.

In the following Perl program, **@entry** is a list. The fifth, sixth, seventh, etc. elements (**$entry[4]**, **$entry[5]**, **$entry[6]**, etc.) are a sublist of IP addresses. The **$#entry** in line 8 is the largest subscript in the **@entry** list. In this example, **$#entry** is 6 because **indira** has the three IP addresses in **$entry[4]**, **$entry[5]**, **$entry[6]**.

Since **$#entry** is 6, the expression **@entry[4 .. $#entry]** in line 8 means the following list of words:

```
($entry[4], $entry[5], $entry[6])
```

The **foreach** loop in line 8 means the same thing as the Korn Shell loop

```
for ip in ${entry[4]} ${entry[5]} ${entry[6]}
do
```

See **ksh**(1) pp. 7−8 for Korn Shell arrays.

We saw the **inet_ntoa** in line 9 in Handout 1, p. 27, line 37.

—On the Web at
**http://i5.nyu.edu/~mm64/x52.9547/src/gethostbyname.pl**

```
 1 #!/bin/perl
 2 #Convert a fully qualified domain name into its IP address(es).
 3 use Socket;
 4
 5 @entry = gethostbyname('indira.uu.nl')
 6     or die "$0: couldn't find indira.uu.nl\n";
 7
 8 foreach $ip (@entry[4 .. $#entry]) {
 9     print inet_ntoa($ip), "\n";
10 }
11
12 exit 0;
```

```
2$ gethostbyname.pl
131.211.16.1
131.211.17.1
131.211.5.1
```

The following program converts in the opposite direction, from an IP address to a fully qualified domain name. The **$entry[0]** in line 8 is the first element of the **@entry** list. It corresponds to the first field of the above C structure. We saw the **inet_aton** in line 5 in Handout 1, p. 17, line 8.

—On the Web at
**http://i5.nyu.edu/~mm64/x52.9547/src/gethostbyaddr.pl**

```
1 #!/bin/perl
2 #Convert an IP address to its fully qualified domain name.
3 use Socket;
4
5 $ip = inet_aton('131.211.17.1')      or die "$0: inet_aton failed";
6 @entry = gethostbyaddr($ip, AF_INET) or die "$0: gethostbyaddr failed";
7
8 print $entry[0], "\n";
9 exit 0;
```

```
3$ gethostbyaddr.pl
Indira.uu.nl
```

**Non-interactive nslookup**

The easiest way to ask your resolver to send a question to a domain name server is by running **nslookup**. **nslookup** even tells you which domain name server it referred your question to.

```
1$ /usr/sbin/nslookup indira.uu.nl
Server:  NYUNSB.NYU.EDU
Address:  128.122.253.37

Non-authoritative answer:
Name:    indira.uu.nl
Addresses:  131.211.17.1, 131.211.5.1, 131.211.16.1


2$ /usr/sbin/nslookup 131.211.17.1
Server:  NYUNSB.NYU.EDU
Address:  128.122.253.37

Name:    Indira.uu.nl
Address:  131.211.17.1
```

**Non-interactive dig**

**dig** gives more detailed output than **nslookup**. See **man resolver** for the resolver options.

```
1$ /usr/sbin/dig indira.uu.nl | cat -n | more
     1
     2  ; <<>> DiG 8.3 <<>> indira.uu.nl
     3  ;; res options: init recurs defnam dnsrch
     4  ;; got answer:
     5  ;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 2
     6  ;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 3, ADDITIONAL: 4
     7  ;; QUERY SECTION:
     8  ;;  indira.uu.nl, type = A, class = IN
     9
    10  ;; ANSWER SECTION:
    11  indira.uu.nl.        7h59m58s IN A    131.211.17.1
    12  indira.uu.nl.        7h59m58s IN A    131.211.5.1
    13  indira.uu.nl.        7h59m58s IN A    131.211.16.1
    14
    15  ;; AUTHORITY SECTION:
    16  uu.nl.           4h18m49s IN NS   ns.uu.nl.
    17  uu.nl.           4h18m49s IN NS   ns1.surfnet.nl.
    18  uu.nl.           4h18m49s IN NS   ns2.uu.nl.
    19
    20  ;; ADDITIONAL SECTION:
    21  ns.uu.nl.        11h47m9s IN A    131.211.4.5
    22  ns1.surfnet.nl.      13h53m1s IN A    192.87.106.101
    23  ns1.surfnet.nl.      21h3m26s IN AAAA  2001:610:1:800a:192:87:106:101
    24  ns2.uu.nl.       1h53m49s IN A    131.211.4.6
    25
    26  ;; Total query time: 2 msec
    27  ;; FROM: i5.nyu.edu to SERVER: default -- 128.122.253.37
    28  ;; WHEN: Thu Dec 22 08:49:51 2005
    29  ;; MSG SIZE  sent: 30  rcvd: 215
    30
```

**Interactive nslookup**

When **script** captures all the input and output of an interactive program, it annoyingly stores a carriage return and a newline at the end of each line except the first one and last two. The carriage return appears in **vi** as a **^M**, since the ASCII code of a carriage return is 13 and **M** is the thirteenth letter of the alphabet. Unix needs only need the newline, so use **vi** to remove the carriage return from the end of every line, except the first one and last two.

```
1$ cd
2$ script myscript
$ /usr/sbin/nslookup     script gives you a dollar sign prompt.
$ exit                   Turn off script when you're done with nslookup.

3$ cd
4$ vi myscript           The carriage returns will appear as ^M's.
:2,$-2s/.$//             Remove last character (carriage return) from each line except first and last two.
:w                       write
:q                       quit
5$
```

See pp. 216−217. The **>** is the **nslookup** prompt.

```
6$ /usr/sbin/nslookup
Default Server:  NYUNSB.NYU.EDU
Address:  128.122.253.37

> set type=any
> set all
Default Server:  NYUNSB.NYU.EDU
Address:  128.122.253.37

Set options:
  nodebug     defname     search  recurse
  nod2        novc        noignoretc  port=53
  querytype=ANY    class=IN    timeout=5   retry=2
  root=ns.internic.net.
  domain=nyu.edu
  srchlist=nyu.edu/home.nyu.edu/es.its.nyu.edu

> nyu.edu
Server:  NYUNSB.NYU.EDU
Address:  128.122.253.37

nyu.edu  internet address = 128.122.108.35
nyu.edu  preference = 10, mail exchanger = SMTP.nyu.edu
nyu.edu  record type LOC, interpreted as:
nyu.edu.     4D IN LOC   40 42 51.000 N 74 00 23.000 W 0.00m 1.00m 10000.00m 10.00m
nyu.edu  nameserver = NS1.nyu.edu
nyu.edu  nameserver = NS2.nyu.edu
nyu.edu  nameserver = NYU-NS.BERKELEY.edu
nyu.edu  nameserver = LAPIETRA.NYU.FLORENCE.IT
nyu.edu
    origin = NS1.nyu.edu
    mail addr = HOSTMASTER.nyu.edu
    serial = 200512220
    refresh = 86400 (1D)
    retry   = 7200 (2H)
    expire  = 2592000 (2592000)
    minimum ttl = 345600 (4D)
nyu.edu  nameserver = NS1.nyu.edu
nyu.edu  nameserver = NS2.nyu.edu
nyu.edu  nameserver = NYU-NS.BERKELEY.edu
nyu.edu  nameserver = LAPIETRA.NYU.FLORENCE.IT
SMTP.nyu.edu internet address = 128.122.109.18
NS1.nyu.edu  internet address = 128.122.253.83
NS2.nyu.edu  internet address = 128.122.253.42
NYU-NS.BERKELEY.edu  internet address = 128.32.222.90
> exit
7$
```

The **4D** in the Location record is the ‘‘time to live’’ in p. 217—four days. RFC 1876 gives the other fields: latitude, longitude, altitude, diameter, horizontal precision, vertical precision. **m** is meters; the default is centimeters.

For the fields in the Start of Authority record, see pp. 572−576. **refresh** is how often a slave server will compare the serial number of its most recently downloaded information with the master’s current serial number. If the master doesn’t respond, a slave will wait **retry** seconds before trying again. A

slave will discard its downloaded information after **`expire`** seconds. A remote server will remember for **`minimum ttl`** seconds that a host in this zone doesn't exist.

```
8$ bc
scale = 5
2592000 / 60 / 60 / 24
30.00000
control-d
9$
```

**A zone transfer**

A *zone file* stores the fully qualified domain names and IP addresses of all the hosts in a zone. The textbook makes several attempts to define a *zone.* Page 60: ''A particular domain's database file is called a *zone file* ...'' Page 206: ''A *zone* is a piece of the domain namespace over which a nameserver holds authority.'' Page 572: ''The Start of Authority Record (SOA) marks the beginning of a zone...''

The names of the hosts in a zone all end with the same suffix. For example, all the hosts whose names end with **`.uu.nl`** constitute a zone. We can get the zone's domain name server to send us all its resource records (RR's). See pp. 60, 207, 231−232.

Not all domain name servers are willing to do this for us—the NYU domain name servers aren't. See the **`allow-transfer`** option on pp. 554, 557.

```
1$ /usr/sbin/nslookup
Default Server:  NYUNSB.NYU.EDU
Address:  128.122.253.37

> set type=any
> uu.nl
Server:  NYUNSB.NYU.EDU
Address:  128.122.253.37

Non-authoritative answer:
uu.nl
    origin = ns.uu.nl
    mail addr = hostmaster.uu.nl
    serial = 2005120102
    refresh = 28800 (8H)
    retry   = 7200 (2H)
    expire  = 604800 (1W)
    minimum ttl = 86400 (1D)
uu.nl   nameserver = ns.uu.nl
uu.nl   nameserver = ns1.surfnet.nl
uu.nl   nameserver = ns2.uu.nl
```

```
Authoritative answers can be found from:
uu.nl   nameserver = ns.uu.nl
uu.nl   nameserver = ns1.surfnet.nl
uu.nl   nameserver = ns2.uu.nl
ns.uu.nlinternet address = 131.211.4.5
ns1.surfnet.nl  internet address = 192.87.106.101
ns1.surfnet.nl  IPv6 address = 2001:610:1:800a:192:87:106:101
ns2.uu.nl   internet address = 131.211.4.6
> server ns1.surfnet.nl
Default Server:  ns1.surfnet.nl
Addresses:  192.87.106.101, 2001:610:1:800a:192:87:106:101

> ls uu.nl
[ns1.surfnet.nl]
$ORIGIN uu.nl.
ns.bio          8H IN A     131.211.48.32
ns2.bio         8H IN A     131.211.48.98
ns.bureau       8H IN A     131.211.98.100
cao1        8H IN A     131.211.17.28
cao2        8H IN A     131.211.17.9
```

*etc.*

```
> exit
2$
```

The **1D** is the ‘‘time to live’’ in p. 217—one day. **IN A** means ‘‘internet address’’.

**Configure and run the name server dæmon in.named on i5.nyu.edu**

RFC 1918 says that the class B NAT addresses **172.16.0.0** through **172.31.255.255** are available for an organization’s internal use, without having to be registered with any authority. See p. 86. If you **traceroute** to these addresses from within NYU, you never go beyond NYU.

Let’s pretend that we have the following three machines:

```
172.31.0.1      host1.x529547.com
172.31.0.2      host2.x529547.com
172.31.0.3      host3.x529547.com
```

**host1.x529547.com** will also be known by the names **www.x529547.com** and **ftp.x529547.com**

We have to create five files:

```
named.conf
named.ca
named.local
172.31.rev
x529547.com.hosts
```

The superuser would put the files in **/var/named**; see **filesystem**(5) p. 5 for the **/var** directory. I put them in my directory **~mm64/public_html/x52.9547/src/named**. You can see this directory on the web at

**http://i5.nyu.edu/~mm64.nyu.edu/x52.9547/src/**

**The configuration file named.conf**

See pp. 212−216; 550−568. In our version of Unix, there must be a semicolon after every **}** except the last. As you'll see, **in.named** will complain about that missing semicolon. But if you supply the missing semicolon, **in.named** will drop dead.

```
1$ man -s 4 named.conf
```

The directory in line 4 is prepended to the filenames in lines 6, 11, 26, 31, 36, and 41. The **print-time** in line 12 will record the date and time in the logging file. The quoted strings in lines 24, 29, 34, and 39 give the default suffix for the names in the first field of the records in each file.

```
2$ cat -n ~mm64/public_html/x52.9547/src/named/named.conf | more
     1  #pp. 212-216; 550-568
     2
     3  options {
     4      directory "/home1/m/mm64/public_html/x52.9547/src/named"; #pp. 553, 213
     5      listen-on port 10566 {128.122.253.152;};                  #pp. 554, 557
     6      pid-file "named.pid";                                     #pp. 553, 555
     7  };
     8
     9  logging {
    10      channel my_channel {
    11          file "named.channel";   #or null; p. 562
    12          print-time yes;         #p. 562
    13      };
    14
    15      category queries {          #individual questions and answers, p. 563
    16          my_channel;
    17      };
    18
    19      category xfer-out {         #outbound zone transfers, p. 563
    20          my_channel;
    21      };
    22  };
    23
    24  zone "." {                      #This file converts 13 names to IP addresses.
    25      type hint;
    26      file "named.ca";            #".ca" stands for "cache"
    27  };
    28
    29  zone "0.0.127.in-addr.arpa"    { #This file converts one IP address to a name.
    30      type master;
    31      file "named.local";
    32  };
    33
    34  zone "x529547.com" {            #This file converts many names to IP addresses.
    35      type master;
    36      file "x529547.com.hosts";
    37  };
    38
    39  zone "31.172.in-addr.arpa" {    #This file converts many IP addresses to names.
    40      type master;
    41      file "172.31.rev";
    42  }   #no ; in BIND Version 8, even though it complains about the missing ;
```

The name server usually sends and receives packets via port 53. (The wildcard with square brackets contains one blank and one tab.)

```
3$ lynx -source http://www.iana.org/assignments/port-numbers | grep '[    ]53/'
domain          53/tcp    Domain Name Server
domain          53/udp    Domain Name Server

4$ awk '$1 == "domain"' /etc/services
domain      53/udp
domain      53/tcp

5$ man -s 4 services
```

But only the superuser can run a program that inputs packets from ports below 1024:

```
6$ cat -n /usr/include/netinet/in.h | sed -n 193,200p
   193  /*
   194   * Ports < IPPORT_RESERVED are reserved for
   195   * privileged processes (e.g. root).
   196   * Ports > IPPORT_USERRESERVED are reserved
   197   * for servers, not necessarily privileged.
   198   */
   199  #define  IPPORT_RESERVED      1024
   200  #define  IPPORT_USERRESERVED  5000
```

That's why the above line 5 listens on port 10566.

To see the port numbers currently in use,

```
7$ netstat -an -f inet -P udp | more                          or -P tcp
   Local Address         Remote Address     State
      *.111                                 Idle
      *.*                                   Unbound
      *.32771                               Idle
      *.*                                   Unbound
      *.123                                 Idle
      *.32778                               Idle
      *.*                                   Unbound
      *.*                                   Unbound
```

**The cache file named.ca**

See pp. 219−221. To download the file, I said

```
1$ cd ~mm64/public_html/x52.9547/src/named
2$ pwd

3$ lynx -source ftp://ftp.internic.net/domain/named.root > named.ca
4$ chmod 444 named.ca
5$ ls -l named.ca
-r--r--r--   1 mm64     users       2517 Nov 17 13:08 named.ca
```

In the **named.conf** file, every statement (except the last) ended with a semicolon. But in the following files, the semicolon is a comment delimiter. The time to live is 3,600,000 seconds = 1,000 hours = 41.67 days.

```
6$ bc
scale = 2
3600000 / 60 / 60 / 24
41.66
```

DNS searches for a name in the first field of each record, and returns the item in the last field of each record.

```
7$ cat -n ~mm64/public_html/x52.9547/src/named/named.ca | sed -n 15,29p
    15  ;
    16  ; formerly NS.INTERNIC.NET
    17  ;
    18  .                        3600000  IN  NS    A.ROOT-SERVERS.NET.
    19  A.ROOT-SERVERS.NET.      3600000      A     198.41.0.4
    20  ;
    21  ; formerly NS1.ISI.EDU
    22  ;
    23  .                        3600000      NS    B.ROOT-SERVERS.NET.
    24  B.ROOT-SERVERS.NET.      3600000      A     192.228.79.201
    25  ;
    26  ; formerly C.PSI.NET
    27  ;
    28  .                        3600000      NS    C.ROOT-SERVERS.NET.
    29  C.ROOT-SERVERS.NET.      3600000      A     192.33.4.12
```

The current status of the 13 root servers is at

**http://netmon.grnet.gr/pings/rootnsping/**

**The loopback file named.local**

See p. 222. The **@** in the first field of line 4 stands for the default suffix **0.0.127.in-addr.arpa** in line 29 of the file **named.conf**.

Line 12 has no first field, so by default it gets the first field of the previous line.

The **1** in the first field of line 14 has no dot after it. That's why the suffix **0.0.127.in-addr.arpa** in line 29 of **named.conf** is appended to it, yielding **1.0.0.127.in-addr.arpa**. The **i5.nyu.edu** in line 12 does have a dot after it. That's why the suffix **0.0.127.in-addr.arpa** is not appended to it.

```
1$ cat -n ~mm64/public_html/x52.9547/src/named/named.local | more
     1  ;Time to live: 60 * 60 * 24 seconds
     2  $TTL 86400
     3
     4  @ IN SOA i5.nyu.edu. mm64.i5.nyu.edu. (
     5     2004062401 ;serial number, p. 573
     6          21600 ;refresh: how often slave server should check serial numbers
     7           1800 ;retry: how long slave server should wait before trying again
     8         604800 ;expire: how long slave server should wait before discarding
     9            900 ;negative cache ttl: how long slave server should cache bad news
    10  )
    11
    12      IN  NS  i5.nyu.edu.
    13  0   IN  PTR loopback.    ;the imaginary network
    14  1   IN  PTR localhost.   ;the one machine on the imaginary network
```

**The reverse zone file 172.31.rev**

See pp. 223−225. The **@** in the first field of line 4 stands for the default suffix **31.172.in-addr.arpa** in line 39 of the file **named.conf**.

**$GENERATE** is a ‘‘for’’ loop. You can use it for **PTR** records, but not for **A** records. See pp. 219, 224−225, 570.

```
1$ cat -n ~mm64/public_html/x52.9547/src/named/172.31.rev | more
     1  ;Time to live: 60 * 60 * 24 seconds
     2  $TTL 86400
     3
     4  @ IN SOA i5.nyu.edu. mm64.i5.nyu.edu. (
     5     2004062401 ;serial number, p. 573
     6          21600 ;refresh: how often slave server should check serial numbers
     7           1800 ;retry: how long slave server should wait before trying again
     8         604800 ;expire: how long slave server should wait before discarding
     9            900 ;negative cache ttl: how long slave server should cache bad news
    10  )
    11
    12      IN NS i5.nyu.edu.
    13
    14  1.0 IN PTR host1.x529547.com.
    15  2.0 IN PTR host2.x529547.com.
    16  3.0 IN PTR host3.x529547.com.
    17
    18  ;$GENERATE 1-3/1 $.0 PTR host$.x529547.com.   ;Can't say "IN".
```

**The forward zone file x529547.com.hosts**

See pp. 225−227. The **@** in the first field of line 4 stands for the default suffix **x529547.com** in line 34 of the file **named.conf**.

```
1$ cat -n ~mm64/public_html/x52.9547/src/named/x529547.com.hosts | more
     1  ;Time to live: 60 * 60 * 24 seconds
     2  $TTL 86400
     3
     4  @ IN SOA i5.nyu.edu. mm64.i5.nyu.edu. (
     5     2004062401 ;serial number, p. 573
     6          21600 ;refresh: how often slave server should check serial numbers
     7           1800 ;retry: how long slave server should wait before trying again
     8         604800 ;expire: how long slave server should wait before discarding
     9            900 ;negative cache ttl: how long slave server should cache bad news
    10  )
    11
    12            IN TXT "Networking and Unix X52.9547"
    13            IN LOC 40 N 75 W 0m   ;latitute, longitude, altitude in meters
    14            IN NS i5.nyu.edu.
    15            IN MX 10 i5.nyu.edu.
    16
    17  localhost IN A  127.0.0.1
    18  host1     IN A  172.31.0.1
    19  host2     IN A  172.31.0.2
    20  host3     IN A  172.31.0.3
    21
    22  ;host1.x529547.com will also be known as www.x529547.com
    23  www       IN CNAME host1
    24
    25  ;www.x529547.com will also be known as ftp.x529547.com
    26  ;(demonstrate that CNAME's can be chained together)
    27  ftp       IN CNAME www
```

**Run the name server dæmon in.named on i5.nyu.edu**

```
1$ cd ~mm64/public_html/x52.9547/src/named
2$ pwd

3$ /usr/sbin/in.named -v            See the version number.
in.named BIND 8.3.3 Wed Feb 18 23:46:02 PST 2004
    Generic Patch-5.9-May 2002
```

**-d 1** turns on the lowest level of debugging. **in.named**(1M) says that **in.named** will put itself in the background without an **&**; look at the **-f** option. But as we have seen, **/etc/rc2.s/S72inetsvc** launches **in.named** with an ampersand, because **in.named** will not put itself into the background until it has bound itself to a port.

```
4$ /usr/sbin/in.named -c named.conf -d 1          minus d one

5$ cat named.pid                                  so you know which process to kill
24553

6$ ps -f -p 'cat named.pid'
     UID   PID  PPID  C    STIME TTY      TIME CMD
    mm64 24553     1  0 08:49:55 ?       0:00 /usr/sbin/in.named -c named.conf -d 1
```

```
7$ head -3 named.run                created because we asked for debugging with -d 1
Debug level 1
Version = in.named BIND 8.3.3 Wed Feb 18 23:46:02 PST 2004
    Generic Patch-5.9-May 2002
```

The following syntax error is caused by the missing semicolon in the **named.conf** file. This problem is fixed in Version 9 of BIND (‘‘Berkeley Internet Name Domain’’), the Unix version of a DNS server.

```
8$ sed -n 42p named.run
named.conf:43: syntax error near <end of file>
```

Individual questions are asked and answered via UDP:

```
9$ netstat -a -f inet -P udp | awk '2 <= NR && NR <= 4 || $1 ~ /\.10566$/'
UDP: IPv4
   Local Address         Remote Address     State
-------------------- -------------------- -------
i5.10566                                   Idle
```

Zone transfers are delivered via TCP:

```
10$ netstat -a -f inet -P tcp | awk '2 <= NR && NR <= 4 || $1 ~ /\.10566$/'
TCP: IPv4
   Local Address        Remote Address    Swind Send-Q Rwind Recv-Q  State
-------------------- -------------------- ----- ------ ----- ------ -------
i5.10566                   *.*                0      0 49152      0 LISTEN
```

**Ask our name server a question**

```
1$ /usr/sbin/nslookup                    with no arguments
Default Server:  NYUNSB.NYU.EDU
Address:  128.122.253.37

> server i5.nyu.edu
Default Server:  i5.nyu.edu
Address:  128.122.253.152

> set port=10566
> host1.x529547.com
Server:  i5.nyu.edu
Address:  128.122.253.152

Name:    host1.x529547.com
Address:  172.31.0.1

> www.x529547.com
Server:  i5.nyu.edu
Address:  128.122.253.152

Name:    host1.x529547.com
Address:  172.31.0.1
Aliases:  www.x529547.com

> ftp.x529547.com
Server:  i5.nyu.edu
Address:  128.122.253.152

Name:    host1.x529547.com
Address:  172.31.0.1
Aliases:  ftp.x529547.com, www.x529547.com

> exit
2$

3$ cat -n named.channel | more      created because named.conf asked for logging
1   22-Dec-2005 08:50:30.350 XX+/128.122.253.152/host1.x529547.com/A/IN
2   22-Dec-2005 08:50:30.358 XX+/128.122.253.152/www.x529547.com/A/IN
3   22-Dec-2005 08:50:30.360 XX+/128.122.253.152/ftp.x529547.com/A/IN

4$ /usr/sbin/dig @i5.nyu.edu host1.x529547.com -p 10566

5$ cat -n named.channel | more
1   22-Dec-2005 08:50:30.350 XX+/128.122.253.152/host1.x529547.com/A/IN
2   22-Dec-2005 08:50:30.358 XX+/128.122.253.152/www.x529547.com/A/IN
3   22-Dec-2005 08:50:30.360 XX+/128.122.253.152/ftp.x529547.com/A/IN
4   22-Dec-2005 08:50:30.467 XX+/128.122.253.152/host1.x529547.com/A/IN
```

**Request a zone transfer from our name server**

```
1$ /usr/sbin/nslookup
Default Server:  NYUNSB.NYU.EDU
Address:  128.122.253.37

> server i5.nyu.edu
Default Server:  i5.nyu.edu
Address:  128.122.253.152

> set port=10566
> set type=any
> x529547.com
Server:  i5.nyu.edu
Address:  128.122.253.152

x529547.com  preference = 10, mail exchanger = i5.nyu.edu
x529547.com  nameserver = i5.nyu.edu
x529547.com  record type LOC, interpreted as:
x529547.com.     1D IN LOC    40 00 00.000 N 75 00 00.000 W 0.00m 0.00m 0.00m 0.00m
x529547.com  text = "Networking and Unix X52.9547"
x529547.com
    origin = i5.nyu.edu
    mail addr = mm64.i5.nyu.edu
    serial = 2004062401
    refresh = 21600 (6H)
    retry   = 1800 (30M)
    expire  = 604800 (1W)
    minimum ttl = 900 (15M)
x529547.com  nameserver = i5.nyu.edu
i5.nyu.edu  internet address = 128.122.253.152
> ls x529547.com
[i5.nyu.edu]
$ORIGIN x529547.com.
host2           1D IN A     172.31.0.2
localhost       1D IN A     127.0.0.1
host3           1D IN A     172.31.0.3
host1           1D IN A     172.31.0.1
> ls -t ptr 31.172.in-addr.arpa
[i5.nyu.edu]
$ORIGIN 31.172.in-addr.arpa.
2.0         1D IN PTR   host2.x529547.com.
3.0         1D IN PTR   host3.x529547.com.
1.0         1D IN PTR   host1.x529547.com.
> exit
2$
```

```
3$ cat -n named.channel | more          created because named.conf asked for logging
1   22-Dec-2005 08:50:30.350 XX+/128.122.253.152/host1.x529547.com/A/IN
2   22-Dec-2005 08:50:30.358 XX+/128.122.253.152/www.x529547.com/A/IN
3   22-Dec-2005 08:50:30.360 XX+/128.122.253.152/ftp.x529547.com/A/IN
4   22-Dec-2005 08:50:30.467 XX+/128.122.253.152/host1.x529547.com/A/IN
5   22-Dec-2005 08:50:30.623 XX+/128.122.253.152/x529547.com/ANY/IN
6   22-Dec-2005 08:50:30.628 XX /128.122.253.152/x529547.com/AXFR/IN
7   22-Dec-2005 08:50:30.628 zone transfer (AXFR) of "x529547.com" (IN) to [128.122.253
8   22-Dec-2005 08:50:30.632 XX /128.122.253.152/31.172.in-addr.arpa/AXFR/IN
9   22-Dec-2005 08:50:30.633 zone transfer (AXFR) of "31.172.in-addr.arpa" (IN) to [128

    4$ dig @i5.nyu.edu x529547.com axfr -p 10566
    5$ dig @i5.nyu.edu -x 172.31 axfr -p 10566

6$ cat -n named.channel | more
1   22-Dec-2005 08:50:30.350 XX+/128.122.253.152/host1.x529547.com/A/IN
2   22-Dec-2005 08:50:30.358 XX+/128.122.253.152/www.x529547.com/A/IN
3   22-Dec-2005 08:50:30.360 XX+/128.122.253.152/ftp.x529547.com/A/IN
4   22-Dec-2005 08:50:30.467 XX+/128.122.253.152/host1.x529547.com/A/IN
5   22-Dec-2005 08:50:30.623 XX+/128.122.253.152/x529547.com/ANY/IN
6   22-Dec-2005 08:50:30.628 XX /128.122.253.152/x529547.com/AXFR/IN
7   22-Dec-2005 08:50:30.628 zone transfer (AXFR) of "x529547.com" (IN) to [128.122.253
8   22-Dec-2005 08:50:30.632 XX /128.122.253.152/31.172.in-addr.arpa/AXFR/IN
9   22-Dec-2005 08:50:30.633 zone transfer (AXFR) of "31.172.in-addr.arpa" (IN) to [128
10  22-Dec-2005 08:50:30.749 XX /128.122.253.152/x529547.com/AXFR/IN
11  22-Dec-2005 08:50:30.749 zone transfer (AXFR) of "x529547.com" (IN) to [128.122.253
12  22-Dec-2005 08:50:30.771 XX /128.122.253.152/31.172.in-addr.arpa/AXFR/IN
13  22-Dec-2005 08:50:30.772 zone transfer (AXFR) of "31.172.in-addr.arpa" (IN) to [128
```

Do not leave your name server running. To kill it,

```
7$ cd ~mm64/public_html/x52.9547/src/named
8$ pwd

9$ cat named.pid
24553

10$ kill -9 ‘cat named.pid‘
11$ ps -Af | grep abc1234
12$ rm named.pid
```

**▼ Homework 5.1: a resource record for an IPv6 address**

Can you make a resource record of type **AAAA** or **A6**? What about **MX** or **RP**?

▲

**▼ Homework 5.2: run a slave server on i5.nyu.edu**

Run a slave server for the domain **x529547.com** on your machine. (Your machine could be i5.nyu.edu). See pp. 214−216. The **masters** statements in the slave's **named.conf** file will have to specify the port number of the master server:

```
masters port 10566 {128.122.253.152;};
```

Hand in the configuration file(s) you had to create, and an **nslookup** or **dig** session to prove that it worked. It the slave doesn't work, see if there are error messages in the **named.run** file.

▲
□