by Peter Collinson, **Hillside Systems**

DENISE ORTAKALES

The Domain Name Service

One of the many reasons for the success of the Internet has been the Domain Name Service, or DNS. (Incidentally, the "S" in DNS can also stand for "Server.") The primary task of DNS is to translate machine or service names into IP numbers. By using a clever ruse, which I'll get to later, it will also translate IP numbers back into names.

However, how the DNS works is poorly understood. When I hear someone say, "Oh, I couldn't get to your Web site. It said something like 'Site Unknown,'" I always ask if they tried again, because I know that what's happened is the DNS lookup for the machine name has failed. This usually happens because the information request was taking too long and, when they try again, the necessary information has probably arrived and can be accessed.

In the beginning, on the ARPANET, name-to-address translation was done using a file known as `HOSTS.TXT`. The single file was maintained by the Stanford Research Institute (SRI) Network Information Center, known as "The NIC," and distributed from a single machine, `SRI-NIC`. Administrators would periodically grab copies of the file using FTP. The file was the basis for the UNIX `/etc/hosts` file, which lists numbers and machine names.

The scheme did not scale. Not only did the size of the file increase as the network expanded but also the traffic from hosts accessing `SRI-NIC` grew. At the time, all the machines were called by a single-word name, and coordinating the name space became increasingly difficult. Anyway, using a unique single word as a name for each machine was obviously not going to work forever. Paul Mockapetris was responsible for designing the architecture of a system that would replace the `HOSTS.TXT` file. He came up with the Domain Name Scheme (yet another translation for the "S").

DNS has two principal good ideas. First, it provides a way of expressing machine names in a hierarchy–a tree of names where each word in the name only needs to be unique at one particular level. The tree means that you, like trillions of other people, could call your machine "zaphod," but your zaphod would be distinguished from my zaphod because the remainder of its fully qualified domain name would be different. Douglas Adams has a lot to answer for. As you know, the hierarchy is written "bottom up," starting with the name of the machine, and as we move from left to right we move up through the separate domains to the top, each level is separated by a "dot."

Second, DNS was envisaged as a distributed database, consisting of many small local databases scattered over the planet, each managed by a program called "a name server." The name server is usually called `named`, although on a Sun machine it's called `in.named`. The job of the name server is to answer queries from local programs and to communicate with other name servers, answering their queries about the local domain.

The key operational idea that makes

this distributed database work is "delegation." At any level in the hierarchy, a name server is able to say, "Well, I don't know about that name. Ask the name server that you will find here." Delegation allows a server higher up the domain tree to pass lookup requests onto a name server further down the tree. For example, if you are looking up `strange.cpg.com`, your application will send the information request to your local name server. Assuming that your local name server doesn't know the answer already, it must traverse all the name servers from the top domain down to find one that contains the information you need. Thus, your local server will start work at the right-most part of the address looking for information on the `.com` domain.

Your name server will already know where to find top-level domain information. When `named` starts, it is given a file that contains the names and addresses of a set of top-level servers. It uses this information to ask one of those top-level servers to send its "zone," a copy of the information that a name server can access from its own disks. A zone contains all the information from a particular level in the domain hierarchy and is logically a single file on the name server. Most `named` implementations allow "include" directives so this logical file can be split over several physical files, should that be desirable. Because the name server is pulling the information from its own disks, it is certain that the information is correct and is said to be "authoritative" about the zone.

The zone data from a top-level server tells your `named` where it can find name servers for all the top-level domains that are recognized in the world. So when you make your request for `strange.cpg.com`, your name server already knows where to find a server that can give it relevant information about the `.com` domain and will move to the next part of the domain name, looking up `cpg.com`.

The `.com` name server will know where to find two name servers for the `cpg.com` domain because this information was supplied when the `cpg.dom` domain was registered. Your name server will contact one of these servers to find the address of Computer Publishing Group's name server. Finally, your name server is able to look up the full address `strange.cpg.com`, getting information from a machine located in CPG's offices.

The ability to locally administer your own name space is a great strength of DNS. When you want to make changes, you don't have to tell some poor overworked person in a central administrative office to change the contents of a file, you can do this yourself and have those changes reflected around the world very quickly.

Of course, it would be desperately slow to have to go through the rigmarole of asking several servers for information for every address that you need to look up, so name servers cache all the information they get. Caching means that a second lookup on the same name will use the information that has just been laboriously obtained. Also, it means that a second lookup on a domain in `.com` can use the information that was obtained before. When a name server has cached information about a zone, it's said to be "nonauthoritative" about that zone.

When you cache information, you have to accept that the data will become incorrect at some random point in time, depending on the whim of disorganized humans. DNS places the control of behavior of the remote caches in the hands of the information owner by allowing them to place a "time to live" (TTL) on the zone data. When that time expires, remote DNS servers know that they must flush the information from their cache.

If you are supporting a zone with DNS, then you will need to choose a TTL for your data. The decision is a trade-off between the load that external sites will present to your system and the accuracy of the picture that you are presenting to the outside world. A short TTL means that remote sites will be accessing your server more frequently, increasing the load on your system. A long TTL means that remote sites will take time to recognize any changes in your information. My name servers set a TTL of one day, which seems a common choice. It means that I only have to wait 24 hours before I know that local changes are reflected in the outside world.

Name Server Types

So you sit in front of your screen typing away, generating DNS requests that may be satisfied by the cache in your local `named`, or more often by reaching out to some remote name server to obtain the necessary information. It all works well in a reliable world, but networking is not reliable. Machines may be down being rebooted, or worse, they've crashed. Perhaps the route to them is temporarily unavailable while the routers across the world struggle to make the network operational because a train derailment in New Jersey has done serious damage to the fiber infrastructure that normally carries the packets.

Well, if we are thinking about reliability, a name server is obviously a single point of failure. If there is only one, and it's down when someone needs information from it, then they are stuck. DNS helps with this problem by allowing you to define a secondary master name server for a particular zone. The basic idea is that the secondary name server mirrors the master, getting its information by pulling data from a master server that is authoritative about the zone and storing it locally.

Of course, you can mix and match the definitions for a particular name server; an invocation of `named` can be primary for one zone and secondary for another. It can just support primary information, just secondary information, or both, or actually neither. It's often desirable to set up a name server that supplies no information but simply acts as a cache to speed up local name searches.

A name server that's acting as a secondary for a particular zone will pull the data from the zone master on startup and will then obey some directives in that information, ensuring that this new cache of information remains current. First, each zone file will contain a serial number that's advanced whenever the file is changed. I use the common convention of placing a reversed date in this field: *yyyymmddSS*, where *yyyy*, *mm*, *dd* are the current year, month and day numeric values, and *SS* is an advancing number, allowing me to make more than one change per day. Forgetting to advance the serial number is a common mistake made by DNS administrators. When a secondary server contacts a master, it first asks for the serial

number and will only pull the data if the serial on the master is greater than the value that it holds.

Second, each zone file contains a *refresh* interval that tells the secondary how often to wake up and ask the master for the zone data. Again, there's a performance and consistency choice to be made here. If the refresh interval is too short, then you may be transferring zones much too often and gobbling bandwidth; if it's too long, then users will get restive because the address of their new workstation hasn't appeared in the copy of `named` that they need to access. The screaming users can be pacified by restarting the appropriate `named`, forcing a reload of its secondary files. However, a restart may not be an option if the machine is remote. On my small network, where my machines are within reach, I set a 12-hour refresh period.

Third, there is a *retry* value. If the secondary server cannot get to the master, it will delay for this period before trying again. Finally, there's an *expire* value, used to completely expire all the data if the name server cannot be reached. I set this to seven days, and I'm considering increasing it to 15.

So there is excellent support for secondary name servers, and this support is endorsed by the name registration bodies that insist that a new domain should exist in two distinct name servers before they will register it. Duplicating name servers makes great sense. The secondary for my domain is run by my Internet service provider (ISP), so that if I have some disaster that persists for a significant period (like the five-day power outage that happened as a result of the great southern English hurricane in 1987), my names and addresses won't disappear from the Net, and mail will not be bounced. The notice of my Net death will not be premature.

Sensible placement of your secondary is a must, and the need to get this right is often neglected. I've recently started to host someone's domain on my server because they had had reports that there were often lookup errors for their machines. Their ISP has two name servers, but they are on the same network, and I suspect that when their network becomes congested, DNS access suffers and fails. If their secondary was placed on another network, with different access characteristics, then I suspect these problems would disappear.

What's in a Zone?

For any zone, the file contains several different types of record. Each record has a key, which is a domain name, a class (which is always IN for Internet), a type and some associated value(s). The number of values will depend on the record type. It's worth knowing a little about the record types because it helps to understand the results of name server access tools like `nslookup`.

Each zone file will begin with a Start of Authority (SOA) record that has a bunch of arguments. First, it has the name of the name server that supplied the data. Second, it has a "fake" machine name that supplies the email address of the person who controls the data. The SOA record for `hillside.co.uk` has `pc.hillside.co.uk` as its second argument; you can replace the first dot with @ and derive my mail address. This information is not used by name servers but is intended for use by humans, providing them with a contact address at the site. The remaining SOA arguments are the TTL and the values used by secondary name servers to time out their copies of the data.

The basic record used to map a domain name to an IP address is the "A" or address record. An IP address defines an interface, and it's often the case that a machine will have several interfaces, so it's perfectly possible to have several A records mapping the same name to different IP addresses.

If a machine has aliases, for example, my machine `wooded.hillside.co.uk` is also known as `ftp.hillside.co.uk` and `www.hillside.co.uk`, then the alias is included as a canonical name (CNAME) record. A CNAME maps one name to its canonical form. Canonical forms are defined by A records and give an IP address for their key. When the name server is given a name to look up and finds a CNAME, it replaces the name it is looking up with the canonical name and does another lookup to find the IP address.

It would be possible for me to define these aliases using A records to map `www.hillside.co.uk` directly to an IP address. However, some applications, notably `sendmail`, need to know that a particular name is an alias so they can change any mail name to its canonical form. It's better to mark an alias as such by using a CNAME record.

DNS also provides routing information for the email by using a mail exchanger (MX) record. When processing mail, the handling program will process a destination address such as `pc@cpg.com` by looking up the name that appears after the @ symbol as an MX query. MX records have priority value and a canonical name. It's usual to have several MX records for any given mail address, and the sending machine will try each starting with the lowest priority value until it manages to deliver the mail. Mailbox addresses can be names of machines but are often domain names forcing the delivery of mail to one mail gateway machine on a local network.

Reverse lookups, translating an IP address into a machine name, are handled by a cunning use of the DNS mechanism with a special pointer record (PTR). The IP address is reversed and used to construct a name in the `in-addr.arpa` domain. For example, my machine `craggy.hillside.co.uk` has an IP address of `194.205.42.1`, but a site wishing to perform a lookup on the address will look up `1.42.205.194.in-addr.arpa`.

The reason for the reversal of the IP address is simple. IP addresses are hierarchical, being split into *classes*. However, the number reads from left to right with the left-most bits used to denote the class and the right-most bits used for machine numbering or further subnetting on a site. I have a Class C address, and my network number is `194.205.42`. My local addresses use the last value of the dotted quad.

To allow me to control my reverse name space, the registration authority needs to delegate my network number to my name server in the form `42.205.194.in-addr.arpa` so that I can associate the correct machine name with the correct number. I create a special zone file for this delegated domain and make PTR record entries that map my IP numbers onto machine (or service) names.

Applications and DNS

Any application on your machine that needs to access the DNS will send a message to its local `named` using routines that are compiled into the program binary. The program will use a set of routines known as the *resolver library* to format the query and interpret the response from the local `named`. The cache that `named` supports plays an important role in making DNS lookup operate quickly.

When the lookup fails, it's actually your `named` that gives up and times out. I'm sure you've experienced a Web browser "getting stuck" after you type a URL; you sit there waiting for something to happen, get bored, but then nothing happens when you hit the Stop button. The browser is hanging because it has looked the name up in the DNS, and very probably a name server somewhere is not reachable. The lookup mechanism tends to take time to realize that something isn't working but will time out, eventually.

Because the lookup routines are compiled into applications, there has to be some mechanism to configure how lookups are to be done if there is no `named` running, or you want to use a `named` on another machine. There are extra complications on SunOS and Solaris machines because of the interaction with the name service formerly known as Yellow Pages, now known as NIS and NIS+.

In the traditional implementation, the resolver routines consult `/etc/hosts` if there is no `named` running on the machine. This means that machines on island networks that are not connected to the Internet can quite happily talk to each other using IP addresses taken from a static file. NIS provides a way to transport this file around an island network, so larger sets of machines can share one master copy of the file.

Once the network is connected to the Internet, then there's a need to run a regular name server to support access to external Internet addresses and also to provide the world with local address mappings. It's still not necessary to run a `named` on every machine on the network. It's possible to establish a file called `/etc/resolv.conf`, which tells the resolver routines where on the local network a name service can be found. However, it's probably better to run a simple cache-only name server in these circumstances; users will get a faster response for name lookups.

Sun has struggled with the interaction between NIS and DNS for several years, meaning that its version of `named` has often lagged considerably behind the current public release. When I was running SunOS on a single machine that was connected to the Internet, I was forced to run YP to ensure that I had a working `named` on the machine. The alternative was to install a version of the Berkeley Internet Name Daemon (BIND), but this meant editing some shared libraries, which I was always queasy about doing.

With recent releases of Solaris, Sun has sorted out many of the name service problems that occurred in earlier systems. The resolver routines inspect `/etc/nsswitch.conf` to find out which sources of name information are available in a particular machine.

Finally

I haven't managed to say that most versions of `named` derive from BIND, which is freely distributable and has been maintained in recent years by Paul Vixie of Vixie Enterprises (`http://www.vix.com`). The System Administrator's Guild (SAGE) awarded Paul its SAGE Outstanding Achievement Award in 1997 for this work.

The bible for DNS and its associated lookup routines is *DNS and Bind* by Paul Albitz and Cricket Liu (ISBN 1-56592-236-0). It's published by O'Reilly & Associates Inc. and is now in its second edition.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.