DNS and BIND, 5th Edition, by Paul Albitz and Cricket Liu, May 2006

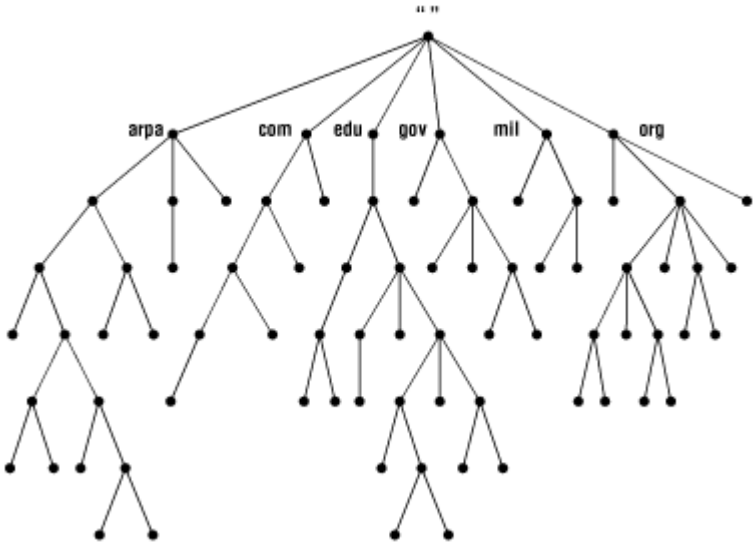
Chapter 2. How Does DNS Work?

The Domain Name System is basically a database of host information. Admittedly, you get a lot with that: funny dotted names, networked nameservers, a shadowy "namespace". But keep in mind that, in the end, the service DNS provides is information about internet hosts. In this chapter, we'll explain and illustrate the mechanisms that make DNS work.

2.1. The Domain Namespace

DNS's distributed database is indexed by domain names. Each domain name is essentially just a path in a large inverted tree, called the *domain namespace*. The tree's hierarchical structure, shown in Figure 2-1, is similar to the structure of the Unix filesystem. The tree has a single root at the top. In the Unix filesystem, this is called the root directory and is represented by a slash (/). DNS simply calls it "the root". Like a filesystem, DNS's tree can branch any number of ways at each intersection point, or *node*. The depth of the tree is limited to 127 levels (a limit you're not likely to reach).

Figure 2-1. The structure of the DNS namespace



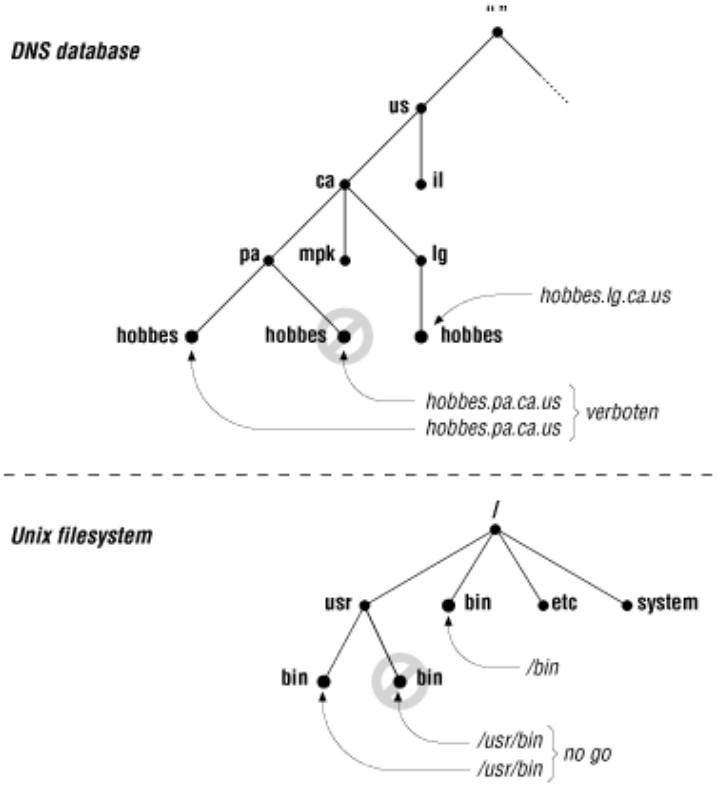
2.1.1. Domain Names

Each node in the tree has a text label (without dots) that can be up to 63 characters long. A null (zero-length) label is reserved for the root. The full *domain name* of any node in the tree is the sequence of labels on the path from that node to the root. Domain names are always read from the node toward the root ("up" the tree), with dots separating the names in the path.

If the root node's label actually appears in a node's domain name, the name looks as though it ends in a dot, as in "www.oreilly.com." (It actually ends with a dot -- the separator -- and the root's null label). When the root node's label appears by itself, it is written as a single dot (.) for convenience. Consequently, some software interprets a trailing dot in a domain name to indicate that the domain name is *absolute*. An absolute domain name is written relative to the root and unambiguously specifies a node's location in the hierarchy. An absolute domain name is also referred to as a *fully qualified domain name*, often abbreviated FQDN. Names without trailing dots are sometimes interpreted as relative to some domain name other than the root, just as directory names without a leading slash are often interpreted as relative to the current directory.

DNS requires that sibling nodes -- nodes that are children of the same parent -- have different labels. This restriction guarantees that a domain name uniquely identifies a single node in the tree. The restriction really isn't a limitation because the labels need to be unique only among the children, not among all the nodes in the tree. The same restriction applies to the Unix filesystem: you can't give two sibling directories or two files in the same directory the same name. As illustrated in Figure 2-2, just as you can't have two *hobbes.pa.ca.us* nodes in the namespace, you can't have two */usr/bin* directories. You can, however, have both a *hobbes.pa.ca.us* node and a *hobbes.lg.ca.us* node, as you can have both a */bin* directory and a */usr/bin* directory.

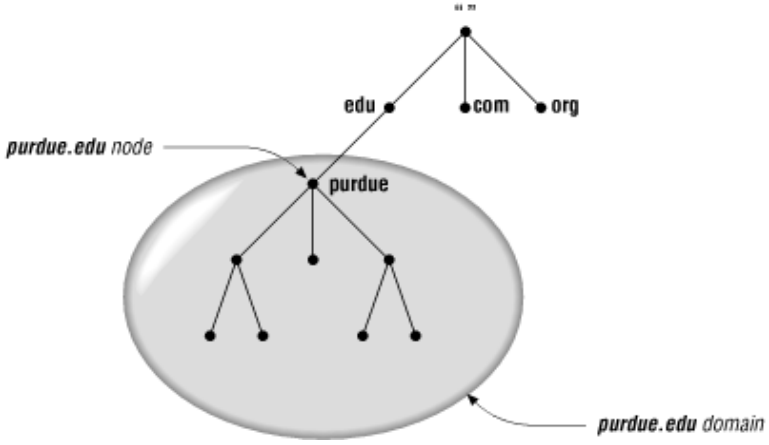
Figure 2-2. Ensuring uniqueness in domain names and in Unix pathnames



2.1.2. Domains

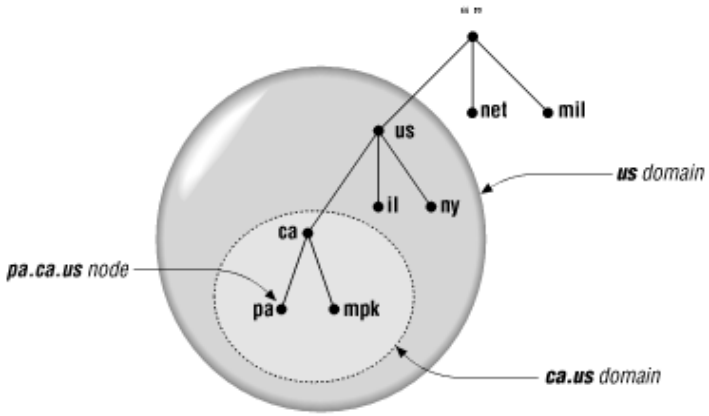
A *domain* is simply a subtree of the domain namespace. The domain name of a domain is the same as the domain name of the node at the very top of the domain. So, for example, the top of the *purdue.edu* domain is a node named *purdue.edu*, as shown in Figure 2-3.

Figure 2-3. The *purdue.edu* domain



Any domain name in the subtree is considered a part of the domain. Because a domain name can be in many subtrees, a domain name can also be in many domains. For example, the domain name *pa.ca.us* is part of the *ca.us* domain and also part of the *us* domain, as shown in Figure 2-4.

Figure 2-4. A node in multiple domains



So in the abstract, a domain is just a subtree of the domain namespace. But if a domain is simply made up of domain names and other domains, where are all the hosts? Domains are groups of hosts, right?

The hosts are there, represented by domain names. Remember, domain names are just indexes into the DNS database. The "hosts" are the domain names that point to information about individual hosts, and a domain contains all the hosts whose domain names are within the domain. The hosts are related *logically*, often by geography or organizational affiliation, and not necessarily by network or address or hardware type. You might have 10 different hosts, each of them on a different network and perhaps even in different countries, all in the same domain.

Domain names at the leaves of the tree generally represent individual hosts, and they may point to network addresses, hardware information, and mail-routing information. Domain names in the interior of the tree can name a host *and* point to information about the domain; they aren't restricted to one or the other. Interior domain names can represent both the domain they correspond to and a particular host on the network. For example, *hp.com* is both the name of the Hewlett-Packard Company's domain and a domain name that refers to the hosts that run HP's main web server.

The type of information retrieved when you use a domain name depends on the context in which you use it. Sending mail to someone at *hp.com* returns mail-routing information, while ssh-ing to the domain name looks up the host information.

A domain may have several subtrees of its own, called *subdomains*[1]. A simple way of determining if a domain is a subdomain of another domain is to compare their domain names. A subdomain's domain name ends with the domain name of its parent domain. For example, the domain *la.tyrell.com* must be a subdomain of *tyrell.com*, because *la.tyrell.com* ends with *tyrell.com*. It's also a subdomain of *com*, as is *tyrell.com*.

Besides being referred to in relative terms, as subdomains of other domains, domains are often referred to by *level*. On mailing lists and in Usenet newsgroups, you may see the terms *top-level domain* or *second-level domain* bandied about. These terms simply refer to a domain's position in the domain namespace:

- A top-level domain is a child of the root.
- A first-level domain is a child of the root (a top-level domain).
- A second-level domain is a child of a first-level domain, and so on.

[1] The terms 'domain' and 'subdomain' are often used interchangeably. We use subdomain only as a relative term: a domain is a subdomain of another domain if the root of the subdomain is within the domain.

2.1.3. Resource Records

The data associated with domain names is contained in *resource records*, or RRs. Records are divided into classes, each of which pertains to a type of network or software. Currently, there are classes for internets (any TCP/IP-based internet), networks based on the Chaosnet protocols, and networks that use Hesiod software. The internet class is by far the most popular. (We're not really sure if anyone still uses the Chaosnet class, and use of the Hesiod class is mostly confined to MIT.) In this book, we concentrate on the internet class.

Within a class, records come in several types, which correspond to the different varieties of data that may be stored in the domain namespace. Different classes may define different record types, though some types are common to more than one class. For example, almost every class defines an *address* type. Each record type in a given class defines a particular record syntax to which all resource records of that class and type must adhere.

2.2. The Internet Domain Namespace

So far, we've talked about the theoretical structure of the domain namespace and what sort of data is stored in it, and we've even hinted at the types of names you might find in it with our (sometimes fictional) examples. But this won't help you decode the domain names you see on a daily basis on the Internet.

The Domain Name System doesn't impose many rules on the labels in domain names, and it doesn't attach any *particular* meaning to the labels at a given level of the namespace. When you manage a part of the domain namespace, you can decide on your own semantics for your domain names. Heck, you could name your subdomains A through Z, and no one would stop you (though they might strongly recommend against it).

The existing Internet domain namespace, however, has some self-imposed structure to it. Especially in the upper-level domains, the domain names follow certain traditions (not rules, really, because they can be and have been broken). These traditions help to keep domain names from appearing totally chaotic. Understanding these traditions is an enormous asset if you're trying to decipher a domain name.

2.2.1. Top-Level Domains

The original top-level domains divided the Internet domain namespace organizationally into seven domains:

> ***com*** -- Commercial organizations, such as Hewlett-Packard (*hp.com*), Sun Microsystems (*sun.com*), and IBM (*ibm.com*).
>
> ***edu*** -- Educational organizations, such as U.C. Berkeley (*berkeley.edu*) and Purdue University (*purdue.edu*).
>
> ***gov*** -- Government organizations, such as NASA (*nasa.gov*) and the National Science Foundation (*nsf.gov*).
>
> ***mil*** -- Military organizations, such as the U.S. Army (*army.mil*) and Navy (*navy.mil*).
>
> ***net*** -- Formerly organizations providing network infrastructure, such as NSFNET (*nsf.net*) and UUNET (*uu.net*). Since 1996, however, *net* has been open to any commercial organization, like *com* is.
>
> ***org*** -- Formerly noncommercial organizations, such as the Electronic Frontier Foundation (*eff.org*). Like *net*, though, restrictions on *org* were removed in 1996.
>
> ***int*** -- International organizations, such as NATO (*nato.int*).

Another top-level domain called *arpa* was originally used during the ARPAnet's transition from host tables to DNS. All ARPAnet hosts originally had hostnames under *arpa*, so they were easy to find. Later, they moved into various subdomains of the organizational top-level domains. However, the *arpa* domain remains in use in a way you'll read about later.

You may notice a certain nationalistic prejudice in our examples: we've used primarily U.S.-based organizations. That's easier to understand -- and forgive -- when you remember that the Internet began as the ARPAnet, a U.S.-funded research project. No one anticipated the success of the ARPAnet, or that it would eventually become as international as the Internet is today.

Today, these original seven domains are called *generic top-level domains*, or gTLDs. The "generic" contrasts them with the country-code top-level domains, which are specific to a particular country.

2.2.1.1. Country-code top-level domains

To accommodate the increasing internationalization of the Internet, the implementers of the Internet namespace compromised. Instead of insisting that all top-level domains describe organizational affiliation, they decided to allow geographical designations, too. New top-level domains were reserved (but not necessarily created) to correspond to individual countries. Their domain names followed an existing international standard called ISO 3166[2]. ISO 3166 establishes official, two-letter abbreviations for every country in the world.

2.2.1.2. New top-level domains

In late 2000, the organization that manages the Domain Name System, the Internet Corporation for Assigned Names and Numbers, or ICANN, created seven new generic top-level domains to accommodate the rapid expansion of the Internet and the need for more domain name "space". A few of these were truly generic top-level domains, like *com*, *net*, and *org*, while others were closer in purpose to *gov* and *mil*: reserved for use by a specific (and sometimes surprisingly small) community. ICANN refers to this latter variety as sponsored TLDs, or sTLDs, and the former as unsponsored gTLDs. Sponsored TLDs have a charter, which defines their function, and a sponsoring organization, which sets policies governing the sTLDs and oversees their operation on ICANN's behalf. Here are the new gTLDs:

aero -- Sponsored; for the aeronautical industry.

biz -- Generic.

coop -- Sponsored; for cooperatives.

info -- Generic.

museum -- Sponsored; for museums.

name -- Generic; for individuals.

pro -- Generic; for professionals

More recently, in early 2005, ICANN approved two more sponsored TLDs, *jobs*, for the human resources management industry, and *travel*, for the travel industry. Several other sponsored TLDs were also under evaluation, including *cat*, for the Catalan linguistic and cultural community, *mobi*, for mobile devices, and *post*, for the postal community. So far, only *mobi* has been delegated from the root. You can check out ICANN at http://www.icann.org.

2.2.2. Further Down

Within these top-level domains, the traditions and the extent to which they are followed vary. Some of the ISO 3166 top-level domains closely follow the United States's original organizational scheme. For example, Australia's top-level domain, *au*, has subdomains such as *edu.au* and *com.au*. Some other ISO 3166 top-level domains follow the *uk* domain's lead and have organizationally oriented subdomains such as *co.uk* for corporations and *ac.uk* for the academic community. In most cases, however, even these geographically oriented top-level domains are divided up organizationally.

---

[2] Except for Great Britain. According to ISO 3166 and Internet tradition, Great Britain's top-level domain name should be gb. Instead, most organizations in Great Britain and Northern Ireland (i.e., the United Kingdom) use the top-level domain name uk. They drive on the wrong side of the road, too.

That wasn't originally true of the *us* top-level domain, though. In the beginning, the *us* domain had 50 subdomains that corresponded to -- guess what? -- the 50 U.S. states. Each was named according to the standard two-letter abbreviation for the state, the same abbreviation standardized by the U.S. Postal Service. Within each state's domain, the organization was still largely geographical: most subdomains corresponded to individual cities. Beneath the cities, the subdomains usually corresponded to individual hosts.

As with so many namespace rules, though, this structure was abandoned when a new company, Neustar, began managing *us* in 2002. Now *us* -- like *com* and *net* -- is open to all comers.

2.2.3. Reading Domain Names

Now that you know what most top-level domains represent and how their namespaces are structured, you'll probably find it much easier to make sense of most domain names. Let's dissect a few for practice:

lithium.cchem.berkeley.edu -- You've got a head start on this one, as we've already told you that *berkeley.edu* is U.C. Berkeley's domain. (Even if you didn't already know that, though, you could have inferred that the name probably belongs to a U.S. university because it's in the top-level *edu* domain). *cchem* is the College of Chemistry's subdomain of *berkeley.edu*. Finally, *lithium* is the name of a particular host in the domain -- and probably one of about a hundred or so, if they have one for every element.

winnie.corp.hp.com -- This example is a bit harder, but not much. The *hp.com* domain in all likelihood belongs to the Hewlett-Packard Company. Its *corp* subdomain is undoubtedly its corporate headquarters. And *winnie* is probably just some silly name someone thought up for a host.

fernwood.mpk.ca.us -- Here, you'll need to use your understanding of the *us* domain. *ca.us* is obviously California's domain, but *mpk* is anybody's guess. In this case, it would be hard to know that it's Menlo Park's domain unless you know your San Francisco Bay Area geography.

daphne.ch.apollo.hp.com -- We've included this example just so you don't start thinking that all domain names have only four labels. *apollo.hp.com* is the former Apollo Computer subdomain of the *hp.com* domain. (When HP acquired Apollo, it also acquired Apollo's Internet domain, *apollo.com*, which became *apollo.hp.com*.) *ch.apollo.hp.com* is Apollo's Chelmsford, Massachusetts site. *daphne* is a host at Chelmsford.

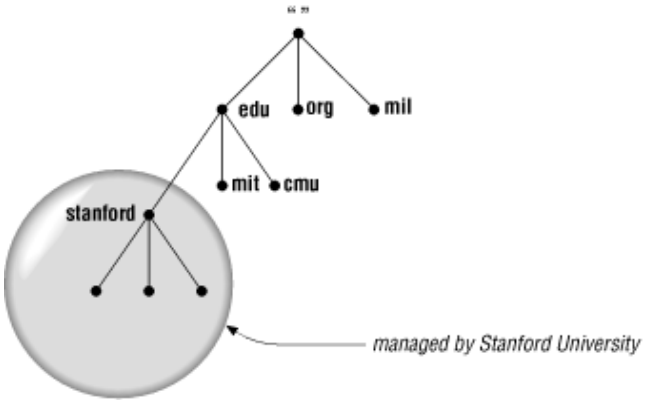
2.3. Delegation

Remember that one of the main goals of the design of the Domain Name System was to decentralize administration? This is achieved through *delegation*. Delegating domains works a lot like delegating tasks at work. A manager may break up a large project into smaller tasks and delegate responsibility for each of these tasks to different employees.

Likewise, an organization administering a domain can divide it into subdomains. Each subdomain can be *delegated* to other organizations, which means that an organization becomes responsible for maintaining all the data in that subdomain. It can freely change the data and even divide its subdomain into more subdomains and delegate those. The parent domain retains only pointers to sources of the subdomain's data, so that it can refer queriers there. The domain *stanford.edu*, for example, is delegated to the folks at Stanford who run the university's networks (Figure 2-5).

Not all organizations delegate away their whole domain, just as not all managers delegate all their work. A domain may have several delegated subdomains and contain hosts that don't belong in the subdomains. For example, the Acme Corporation (it supplies a certain coyote with most of his gadgets), which has a division in Rockaway and its headquarters in Kalamazoo, might have a *rockaway.acme.com* subdomain and a *kalamazoo.acme.com* subdomain. However, the few hosts in the Acme sales offices scattered throughout the United States would fit better under *acme.com* than under either subdomain.

Figure 2-5. *stanford.edu* is delegated to Stanford University

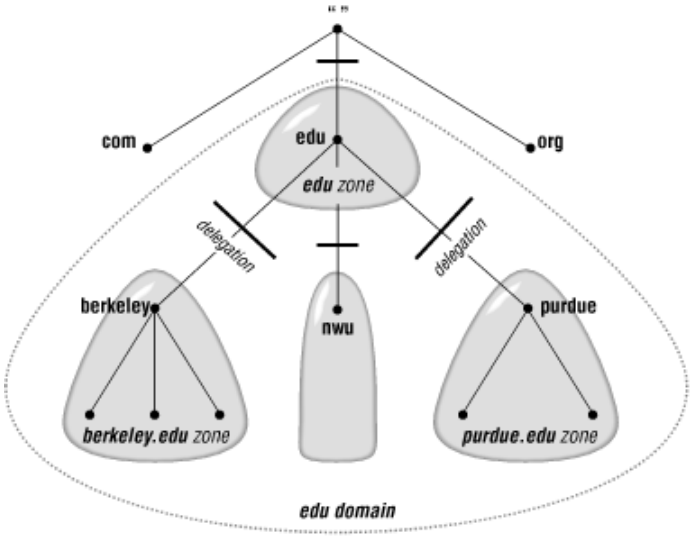


2.4. Nameservers and Zones

The programs that store information about the domain namespace are called *nameservers*. Nameservers generally have complete information about some part of the domain namespace, called a *zone*, which they load from a file or from another nameserver. The nameserver is then said to have *authority* for that zone. Nameservers can be authoritative for multiple zones, too.

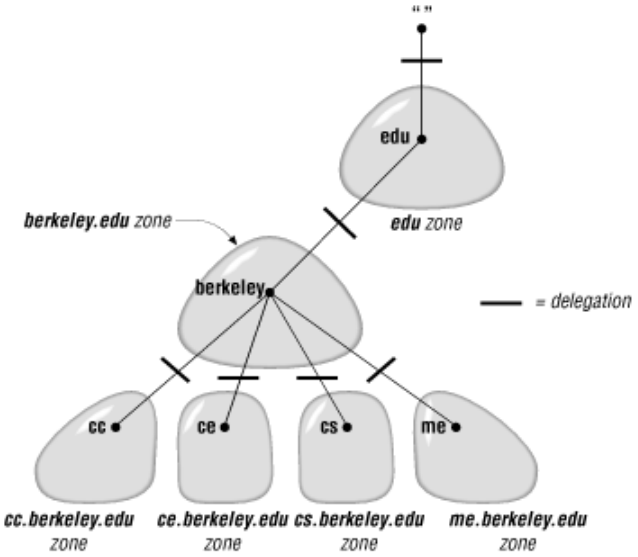
The difference between a zone and a domain is important, but subtle. All top-level domains and many domains at the second level and lower, such as *berkeley.edu* and *hp.com*, are broken into smaller, more manageable units by delegation. These units are called zones. The *edu* domain, shown in Figure 2-6, is divided into many zones, including the *berkeley.edu* zone, the *purdue.edu* zone, and the *nwu.edu* zone. At the top of the domain, there's also an *edu* zone. It's natural that the folks who run *edu* would break up the *edu* domain: otherwise, they'd have to manage the *berkeley.edu* subdomain themselves. It makes much more sense to delegate *berkeley.edu* to Berkeley. What's left for the folks who run *edu*? The *edu* zone, which contains mostly delegation information for the subdomains of *edu*.

Figure 2-6. The *edu* domain broken into zones



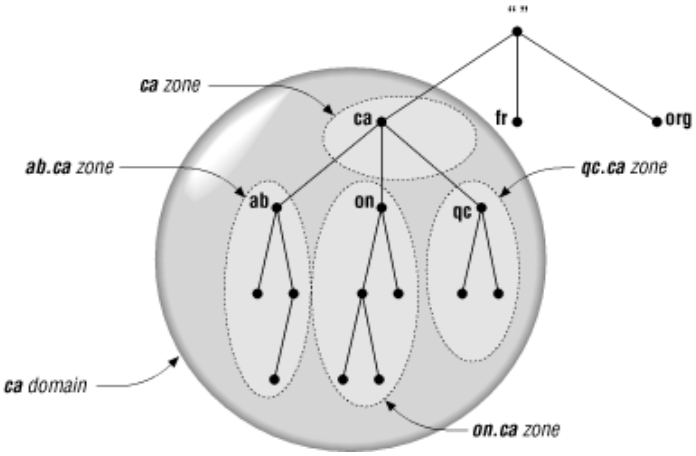
The *berkeley.edu* subdomain is, in turn, broken up into multiple zones by delegation, as shown in Figure 2-7. There are delegated subdomains called *cc*, *cs*, *ce*, *me*, and more. Each subdomain is delegated to a set of nameservers, some of which are also authoritative for *berkeley.edu*. However, the zones are still separate and may have totally different groups of authoritative nameservers.

Figure 2-7. The *berkeley.edu* domain broken into zones



A zone contains all the domain names the domain with the same domain name contains, except for domain names in delegated subdomains. For example, the top-level domain *ca* (for Canada) has subdomains called *ab.ca*, *on.ca*, and *qc.ca*, for the provinces Alberta, Ontario, and Quebec. Authority for the *ab.ca*, *on.ca*, and *qc.ca* domains may be delegated to nameservers in each province. The domain *ca* contains all the data in *ca* plus all the data in *ab.ca*, *on.ca*, and *qc.ca*. However, the zone *ca* contains only the data in *ca* (see Figure 2-8), which is mostly pointers to the delegated subdomains. *ab.ca*, *on.ca*, and *qc.ca* are separate zones from the *ca* zone.

Fig 2-8. The domain *ca* …



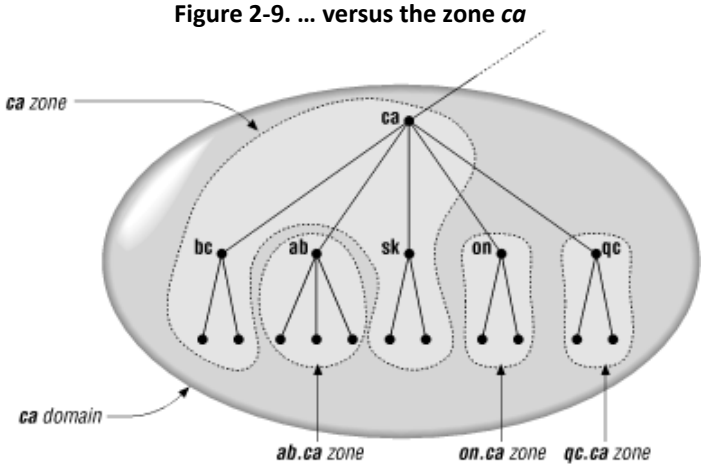
The zone also contains the domain names and data in any subdomains that aren't delegated away. For example, the *bc.ca* and *sk.ca* (British Columbia and Saskatchewan) subdomains of the *ca* domain may exist but not be delegated. (Perhaps the provincial authorities in British Columbia and Saskatchewan aren't yet ready to manage their subdomains, but the authorities running the top-level *ca* domain want to preserve the consistency of the namespace and implement subdomains for all the Canadian provinces right away). In this case, the zone *ca* has a ragged bottom edge, containing *bc.ca* and *sk.ca* but not the other *ca* subdomains, as shown in Figure 2-9.

Now it's clear why nameservers load zones instead of domains: a domain may contain more information than the nameserver needs because it can contain data delegated to other nameservers[3]. Since a zone is bounded by delegation, it will never include delegated data.

[3] If a root nameserver loaded the root domain instead of the root zone, it would be loading the entire namespace!

If you're just starting out, your domain probably won't have any subdomains. In this case, since there's no delegation going on, your domain and your zone will contain the same data.

Figure 2-9. … versus the zone *ca*



2.4.1. Delegating Subdomains

Even though you may not need to delegate parts of your domain just yet, it's helpful to understand a little more about how the process of delegating a subdomain works. Delegation, in the abstract, involves assigning responsibility for some part of your domain to another organization. What really happens, however, is the assignment of authority for a subdomain to different nameservers. (Note that we said "nameservers", not just "nameserver").

Your zone's data, instead of containing information in the subdomain you've delegated, includes pointers to the nameservers that are authoritative for that subdomain. Now if one of your nameservers is asked for data in the subdomain, it can reply with a list of the right nameservers to contact.

2.4.2. Types of Nameservers

The DNS specs define two types of nameservers: primary masters and secondary masters. A *primary master* nameserver for a zone reads the data for the zone from a file on its host. A *secondary master* nameserver for a zone gets the zone data from another nameserver authoritative for the zone, called its *master server*. Quite often, the master server is the zone's primary master, but that's not required: a secondary master can load zone data from another secondary. When a secondary starts up, it contacts its master nameserver and, if necessary, pulls the zone data over. This is referred to as a *zone transfer*. Nowadays, the preferred term for a secondary master nameserver is a *slave*, though many people (and some software, including Microsoft's DNS console) still use the old term.

Both the primary master and slave nameservers for a zone are authoritative for that zone. Despite the somewhat disparaging name, slaves aren't second-class nameservers. DNS provides these two types of nameservers to make administration easier. Once you've created the data for your zone and set up a primary master nameserver, you don't need to copy that data from host to host to create new nameservers for the zone. You simply set up slave nameservers that load their data from the primary master for the zone. The slaves you set up will transfer new zone data when necessary.

Slave nameservers are important because it's a good idea to set up more than one authoritative nameserver for any given zone. You'll want more than one for redundancy, to spread the load around and to make sure that all the hosts in the zone have a nameserver close by. Using slave nameservers makes this administratively workable.

Calling a *particular* nameserver a primary master nameserver or a slave nameserver is a little imprecise, though. We mentioned earlier that a nameserver can be authoritative for more than one zone. Similarly, a nameserver can be a primary master for one zone and a slave for

another. Most nameservers, however, are either primary for most of the zones they load or slave for most of the zones they load. So if we call a particular nameserver a primary or a slave, we mean that it's the primary master or a slave for most of the zones for which it's authoritative.

2.4.3. Zone Datafiles

The files from which primary master nameservers load their zone data are called, simply enough, zone datafiles. We often refer to them as datafiles. Slave nameservers can also load their zone data from datafiles. Slaves are usually configured to back up the zone data they transfer from a master nameserver to datafiles. If the slave is later killed and restarted, it reads the backup datafiles first, then checks to see whether its zone data is current. This both obviates the need to transfer the zone data if it hasn't changed and provides a source of the data if the master is down. The datafiles contain resource records that describe the zone. The resource records describe all the hosts in the zone and mark any delegation of subdomains.

2.5. Resolvers

Resolvers are the clients that access nameservers. Programs running on a host that need information from the domain namespace use the resolver. The resolver handles:

- Querying a nameserver
- Interpreting responses (which may be resource records or an error)
- Returning the information to the programs that requested it

In BIND, the resolver is a set of library routines that is linked to programs such as ssh and ftp. It's not even a separate process. The resolver relies almost entirely on the nameservers it queries: it has the smarts to put together a query, to send it and wait for an answer, and to resend the query if it isn't answered, but that's about all. Most of the burden of finding an answer to the query is placed on the nameserver. The DNS specs call this kind of resolver a *stub resolver*.

Other implementations of DNS have had smarter resolvers that could do more sophisticated things that had more advanced capabilities, such as following referrals to locate the nameservers authoritative for a particular zone.

2.6. Resolution

Nameservers are adept at retrieving data from the domain namespace. They have to be, given the limited intelligence of most resolvers. Not only can they give you data about zones for which they're authoritative, they can also search through the domain namespace to find data for which they're not authoritative. This process is called *name resolution*, or simply *resolution*.

Because the namespace is structured as an inverted tree, a nameserver needs only one piece of information to find its way to any point in the tree: the domain names and addresses of the root nameservers. A nameserver can issue a query to a root nameserver for any domain name in the domain namespace, and the root nameserver will start the nameserver on its way.

2.6.1. Root Nameservers

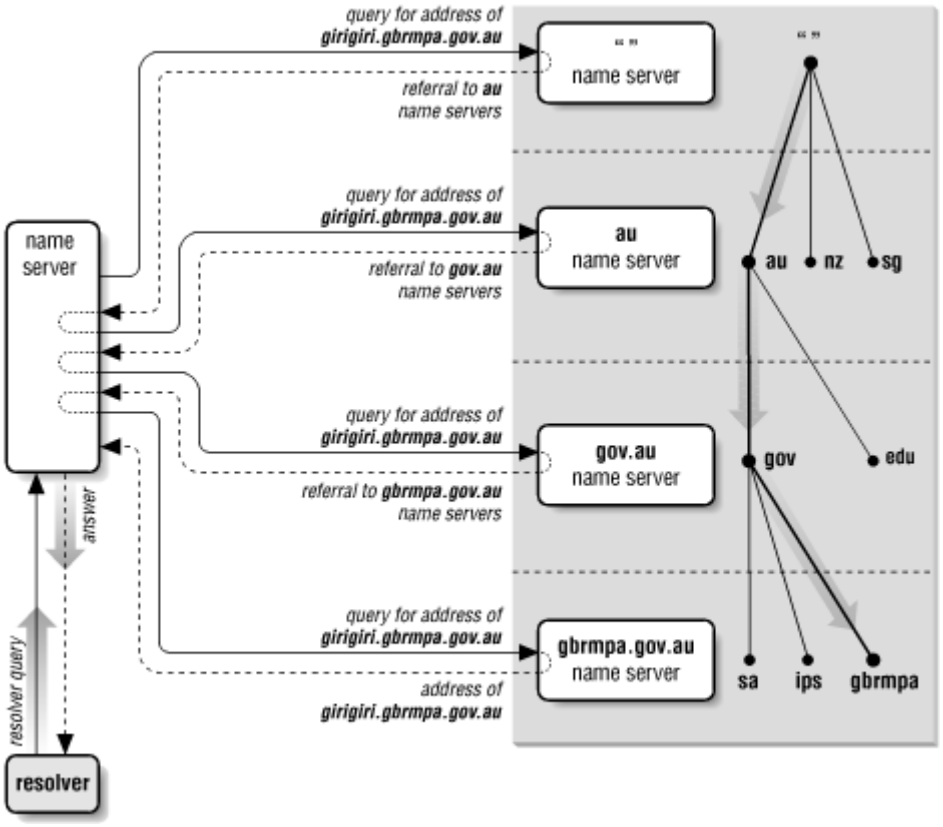
The root nameservers know where the authoritative nameservers for each of the top-level zones are. (In fact, some of the root nameservers are authoritative for some of the generic top-level zones). Given a query about any domain name, the root nameservers can at least provide the names and addresses of the nameservers that are authoritative for the top-level zone the domain name ends in. In turn, the top-level nameservers can provide the list of authoritative nameservers for the second-level zone that the domain name ends in. Each nameserver queried either gives the querier information about how to get "closer" to the answer it's seeking or provides the answer itself.

The root nameservers are clearly important to resolution. Because they're so important, DNS provides mechanisms -- such as caching, which we'll discuss a little later -- to help offload the root nameservers. But in the absence of other information, resolution has to start at the root nameservers. This makes the root nameservers crucial to the operation of DNS; if all the Internet root nameservers were unreachable for an extended period, all resolution on the Internet would fail. To protect against this, the Internet has 13 root nameservers[4] (as of this writing) spread across different parts of the network. One is on PSINet, a commercial Internet backbone; one is on the NASA Science Internet; two are in Europe; and one is in Japan

Being the focal point for so many queries keeps the roots busy; even with 13, the traffic to each root nameserver is very high. A recent informal poll of root nameserver administrators showed some roots receiving tens of thousands of queries per second.

Despite the load placed on root nameservers, resolution on the Internet works quite well. Figure 2-10 shows the resolution process for the address of a real host in a real domain, including how the process corresponds to traversing the domain namespace tree.

Figure 2-10. Resolution of *girigiri.gbrmpa.gov.au* on the Internet



The local nameserver queries a root nameserver for the address of *girigiri.gbrmpa.gov.au* and is referred to the *au* nameservers. The local nameserver asks an *au* nameserver the same question, and is referred to the *gov.au* nameservers. The *gov.au* nameserver refers the local nameserver to the *gbrmpa.gov.au* nameservers. Finally, the local nameserver asks a *gbrmpa.gov.au* nameserver for the address and gets the answer.

2.6.2. Recursion

You may have noticed a big difference in the amount of work done by the nameservers in the previous example. Four nameservers simply returned the best answer they already had --

[4] In fact, the 13 "logical" root nameservers comprise many more physical nameservers. Most of the root servers are either load-balanced behind a single IP address, a "shared unicast" group of distributed nameservers that use the same IP address, or some combination of the two.

mostly referrals to other name servers -- to the queries they received. They didn't have to send their own queries to find the data requested. But one nameserver -- the one queried by the resolver -- had to follow successive referrals until it received an answer.

Why couldn't the local nameserver simply have referred the resolver to another nameserver? Because a stub resolver wouldn't have had the intelligence to follow a referral. And how did the nameserver know not to answer with a referral? Because the resolver issued a recursive query.

Queries come in two flavors, *recursive* and *iterative*, also called *nonrecursive*. Recursive queries place most of the burden of resolution on a single nameserver. *Recursion*, or *recursive resolution*, is just a name for the resolution process used by a nameserver when it receives recursive queries. As with recursive algorithms in programming, the nameserver repeats the same basic process (querying a remote nameserver and following any referrals) until it receives an answer. *Iteration*, or *iterative resolution*, on the other hand, refers to the resolution process used by a nameserver when it receives iterative queries.

In recursion, a resolver sends a recursive query to a nameserver for information about a particular domain name. The queried nameserver is then obliged to respond with the requested data or with an error stating either that data of the requested type doesn't exist or that the domain name specified doesn't exist. The nameserver can't just refer the querier to a different nameserver, because the query was recursive.

If the queried nameserver isn't authoritative for the data requested, it will have to query other nameservers to find the answer. It could send recursive queries to those nameservers, thereby obliging them to find the answer and return it, or it could send iterative queries and possibly be referred to other nameservers *closer* to the domain name it's seeking. Current implementations are polite and by default do the latter, following the referrals until an answer is found.

A nameserver that receives a recursive query that it can't answer itself will query the "closest known" nameservers. The closest known nameservers are the servers authoritative for the zone closest to the domain name being looked up. For example, if the nameserver receives a recursive query for the address of the domain name *girigiri.gbrmpa.gov.au*, it first checks whether it knows which nameservers are authoritative for *girigiri.gbrmpa.gov.au*. If it does, it sends the query to one of them. If not, it checks whether it knows the nameservers for *gbrmpa.gov.au*, and after that *gov.au*, and then *au*. The default, where the check is guaranteed to stop, is the root zone, because every nameserver knows the domain names and addresses of the root nameservers.

Using the closest known nameservers ensures that the resolution process is as short as possible. A *berkeley.edu* nameserver receiving a recursive query for the address of *waxwing.ce.berkeley.edu* shouldn't have to consult the root nameservers; it can simply follow delegation information directly to the *ce.berkeley.edu* nameservers. Likewise, a nameserver that has just looked up a domain name in *ce.berkeley.edu* shouldn't have to start resolution at the root to look up another *ce.berkeley.edu* (or *berkeley.edu*) domain name; we'll show how this works in the "Caching" section.

The nameserver that receives the recursive query always sends the same query that the resolver sent it for example, for the address of *waxwing.ce.berkeley.edu*. It never sends explicit queries for the nameservers for *ce.berkeley.edu* or *berkeley.edu*, though this information is also stored in the namespace. Sending explicit queries could cause problems: for example, there may be no *ce.berkeley.edu* nameservers (that is, *ce.berkeley.edu* may be part of the *berkeley.edu* zone). Also, it's always possible that an *edu* or *berkeley.edu* nameserver would know *waxwing.ce.berkeley.edu*'s address. An explicit query for the *berkeley.edu* or *ce.berkeley.edu* nameservers would miss this information.

2.6.3. Iteration

Iterative resolution doesn't require nearly as much work on the part of the queried nameserver. In iterative resolution, a nameserver simply gives the best answer *it already knows* back to the querier. No additional querying is required. The queried nameserver consults its local data (including its cache, which we'll talk about shortly), looking for the data requested. If it doesn't

find the answer there, it finds the names and addresses of the nameservers closest to the domain name in the query in its local data and returns that as a referral to help the querier continue the resolution process. Note that the referral includes *all* nameservers listed in the local data; it's up to the querier to choose which one to query next.

2.6.4. Choosing Between Authoritative Nameservers

You may be wondering how the nameserver that receives the recursive query chooses among the nameservers authoritative for the zone. For example, we said that there are 13 root nameservers on the Internet today. Does the nameserver simply query the one that appears first in the referral? Does it choose randomly?

BIND nameservers use a metric called *roundtrip time*, or RTT, to choose among nameservers authoritative for the same zone. Roundtrip time is a measurement of how long a remote nameserver takes to respond to queries. Each time a BIND nameserver sends a query to a remote nameserver, it starts an internal stopwatch. When it receives a response, it stops the stopwatch and makes a note of how long that remote nameserver took to respond. When the nameserver must choose which of a group of authoritative nameservers to query, it simply chooses the one with the lowest roundtrip time.

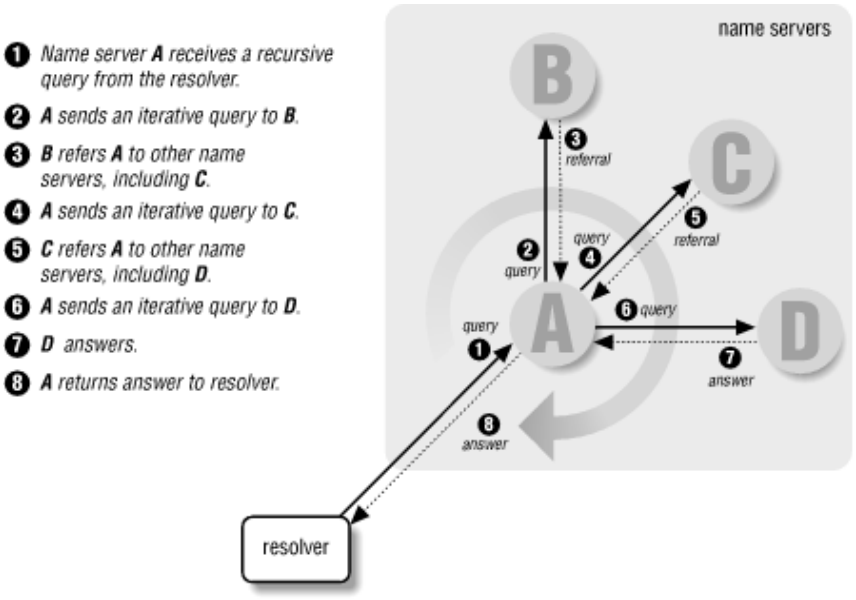
Before a BIND nameserver has queried a nameserver, it gives it a random roundtrip time value lower than any real-world value. This ensures that the BIND nameserver queries all nameservers authoritative for a given zone in a random order before playing favorites.

On the whole, this simple but elegant algorithm allows BIND nameservers to "lock on" to the closest nameservers quickly and without the overhead of an out-of-band mechanism to measure performance.

2.6.5. The Whole Enchilada

What this amounts to is a resolution process that, taken as a whole, looks like Figure 2-11.

Figure 2-11. The resolution process



A resolver queries a local nameserver, which then sends iterative queries to a number of other nameservers in pursuit of an answer for the resolver. Each nameserver it queries refers it to another nameserver that is authoritative for a zone further down in the namespace and closer to the domain name sought. Finally, the local nameserver queries the authoritative nameserver, which returns an answer. All the while, the local nameserver uses each response it receives -- whether a referral or the answer -- to update the RTT of the responding nameserver, which will help it decide which nameservers to query to resolve domain names in the future.

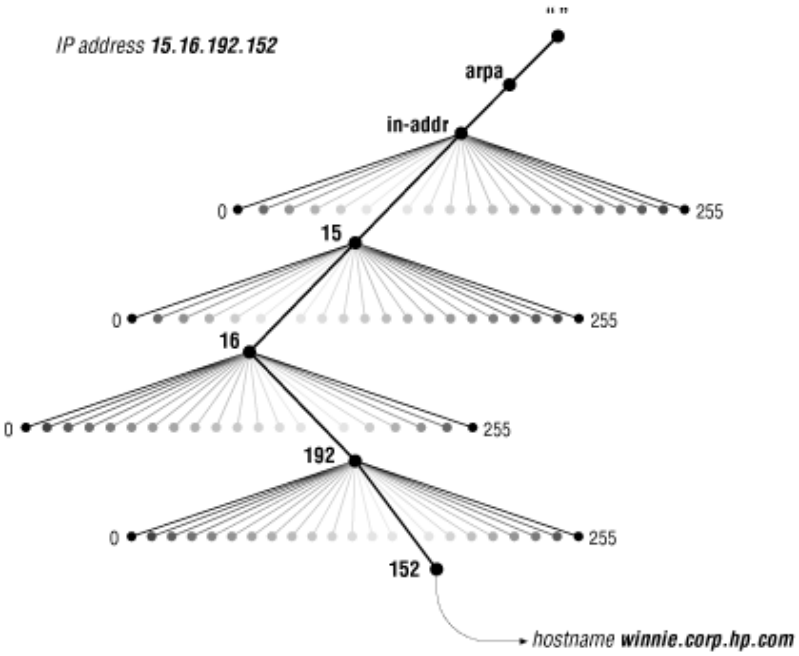
2.6.6. Mapping Addresses to Names

One major piece of functionality missing from the resolution process as explained so far is how addresses get mapped back to domain names. Address-to-name mapping produces output that is easier for humans to read and interpret (in logfiles, for instance). It's also used in some authorization checks. Unix hosts map addresses to domain names to compare against entries in *.rhosts* and *hosts.equiv* files, for example. When using host tables, address-to-name mapping is trivial. It requires a straightforward sequential search through the host table for an address. The search returns the official hostname listed. In DNS, however, address-to-name mapping isn't so simple. Data, including addresses, in the domain namespace is indexed by name. Given a domain name, finding an address is relatively easy. But finding the domain name that maps to a given address would seem to require an exhaustive search of the data attached to every domain name in the tree.

Actually, there's a better solution that's both clever and effective. Because it's easy to find data once you're given the domain name that indexes that data, why not create a part of the domain namespace that uses addresses as labels? In the Internet's domain namespace, this portion of the namespace is the *in-addr.arpa* domain.

Nodes in the *in-addr.arpa* domain are labeled with the numbers in the dotted-octet representation of IP addresses. (Dotted-octet representation refers to the common method of expressing 32-bit IP addresses as four numbers in the range 0 to 255, separated by dots). The *in-addr.arpa* domain, for example, can have up to 256 subdomains, one corresponding to each possible value in the first octet of an IP address. Each subdomain can have up to 256 subdomains of its own, corresponding to the possible values of the second octet. Finally, at the fourth level down, there are resource records attached to the final octet giving the full domain name of the host at that IP address. That makes for an awfully big domain: *in-addr.arpa*, shown in Figure 2-12, is roomy enough for every IP address on the Internet.

Figure 2-12. The *in-addr.arpa* domain

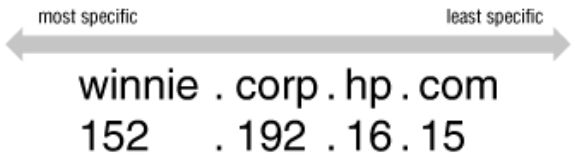


Note that when read in a domain name, the IP address appears backward because the name is read from leaf to root. For example, if *winnie.corp.hp.com*'s IP address is 15.16.192.152, the corresponding node in the *in-addr.arpa* domain is *152.192.16.15.in-addr.arpa*, which maps back to the domain name *winnie.corp.hp.com*.

IP addresses could have been represented the opposite way in the namespace, with the first octet of the IP address at the bottom of the in-addr.arpa domain. That way, the IP address would have read correctly (forward) in the domain name. IP addresses are hierarchical,

however, just like domain names. Network numbers are doled out much as domain names are, and administrators can then subnet their address space and further delegate numbering. The difference is that IP addresses get more specific from left to right, while domain names get less specific from left to right. Figure 2-13 shows what we mean.

Figure 2-13. Hierarchical names and addresses



Making the first octets in the IP address appear highest in the tree enables administrators to delegate authority for *in-addr.arpa* zones along network lines. For example, the *15.in-addr.arpa* zone, which contains the reverse-mapping information for all hosts whose IP addresses start with 15, can be delegated to the administrators of network 15.0.0.0. This would be impossible if the octets appeared in the opposite order. If the IP addresses were represented the other way around, *15.in-addr.arpa* would consist of every host whose IP address ended with 15 -- not a practical zone to try to delegate.

2.7. Caching

The whole resolution process may seem awfully convoluted and cumbersome to someone accustomed to simple searches through the host table. Actually, though, it's usually quite fast. One of the features that speeds it up considerably is *caching*.

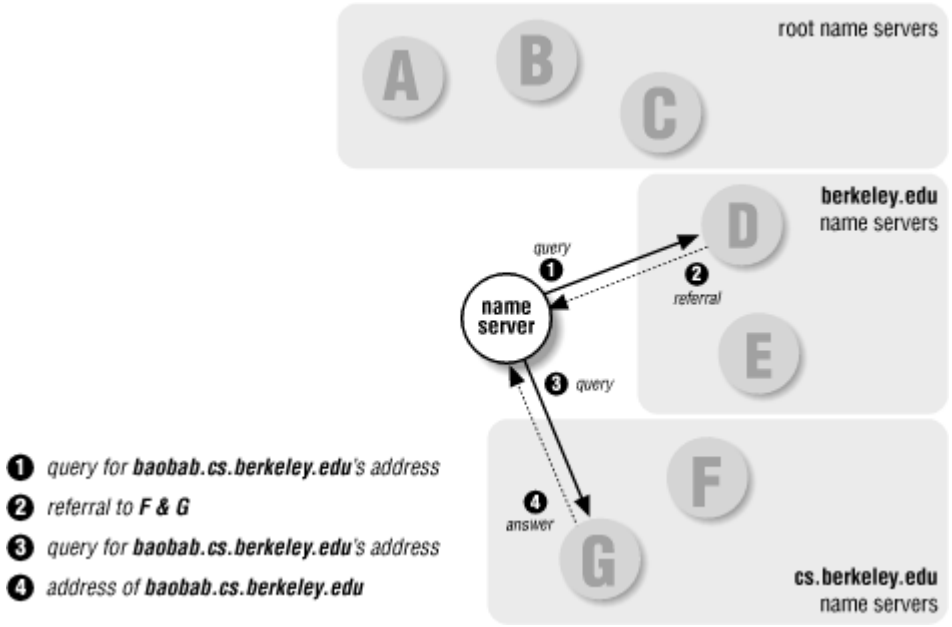
A nameserver processing a recursive query may have to send out quite a few queries to find an answer. However, it discovers a lot of information about the domain namespace as it does so. Each time it's referred to another list of nameservers, it learns that those nameservers are authoritative for some zone, and it learns the addresses of those servers. At the end of the resolution process, when it finally finds the data the original querier sought, it can store that data for future reference, too. The BIND nameserver even implements *negative caching*: if a nameserver responds to a query with an answer that says the domain name or data type in the query doesn't exist, the local nameserver will also temporarily cache that information.

Nameservers cache all this data to help speed up successive queries. The next time a resolver queries the nameserver for data about a domain name the nameserver knows something about, the process is shortened quite a bit. The nameserver may have cached the answer, positive or negative, in which case it simply returns the answer to the resolver. Even if it doesn't have the answer cached, it may have learned the identities of the nameservers that are authoritative for the zone the domain name is in and be able to query them directly.

For example, say our nameserver has already looked up the address of *eecs.berkeley.edu*. In the process, it cached the names and addresses of the *eecs.berkeley.edu* and *berkeley.edu* nameservers (plus *eecs.berkeley.edu*'s IP address). Now if a resolver were to query our nameserver for the address of *baobab.cs.berkeley.edu*, our nameserver could skip querying the root nameservers. Recognizing that *berkeley.edu* is the closest ancestor of *baobab.cs.berkeley.edu* that it knows about, our nameserver would start by querying a *berkeley.edu* nameserver, as shown in Figure 2-14. On the other hand, if our nameserver discovered that there was no address for *eecs.berkeley.edu*, the next time it received a query for the address, it could simply respond appropriately from its cache.

In addition to speeding up resolution, caching obviates a nameserver's need to query the root nameservers to answer any queries it can't answer locally. This means it's not as dependent on the roots, and the roots won't suffer as much from all its queries.

Figure 2-14. Resolving *baobab.cs.berkeley.edu*



2.7.1. Time to Live

Nameservers can't cache data forever, of course. If they did, changes to that data on the authoritative nameservers would never reach the rest of the network; remote nameservers would just continue to use cached data. Consequently, the administrator of the zone that contains the data decides on a *time to live* (TTL) for the data. The time to live is the amount of time that any nameserver is allowed to cache the data. After the time to live expires, the nameserver must discard the cached data and get new data from the authoritative nameservers. This also applies to negatively cached data: a nameserver must time out a negative answer after a period in case new data has been added on the authoritative nameservers.

Deciding on a time to live for your data is essentially deciding on a trade-off between performance and consistency. A small TTL helps ensure that data in your zones is consistent across the network, because remote nameservers will time it out more quickly and be forced to query your authoritative nameservers more often for new data. On the other hand, this increases the load on your nameservers and lengthens the average resolution time for information in your zones.

A large TTL reduces the average time it takes to resolve information in your zones because the data can be cached longer. The drawback is that your information will be inconsistent longer if you make changes to the data on your nameservers.