The Domain Name System is basically a database of host information. Admittedly, you get a lot with that: funny dotted names, networked name servers, a shadowy "name space." But keep in mind that, in the end, the service DNS provides is information about internet hosts.

We've already covered some important aspects of DNS, including its client-server architecture and the structure of the DNS database. However, we haven't gone into much detail, and we haven't explained the nuts and bolts of DNS's operation.

In this chapter, we'll explain and illustrate the mechanisms that make DNS work. We'll also introduce the terms you'll need to know to read the rest of the book (and to converse intelligently with your fellow domain administrators).

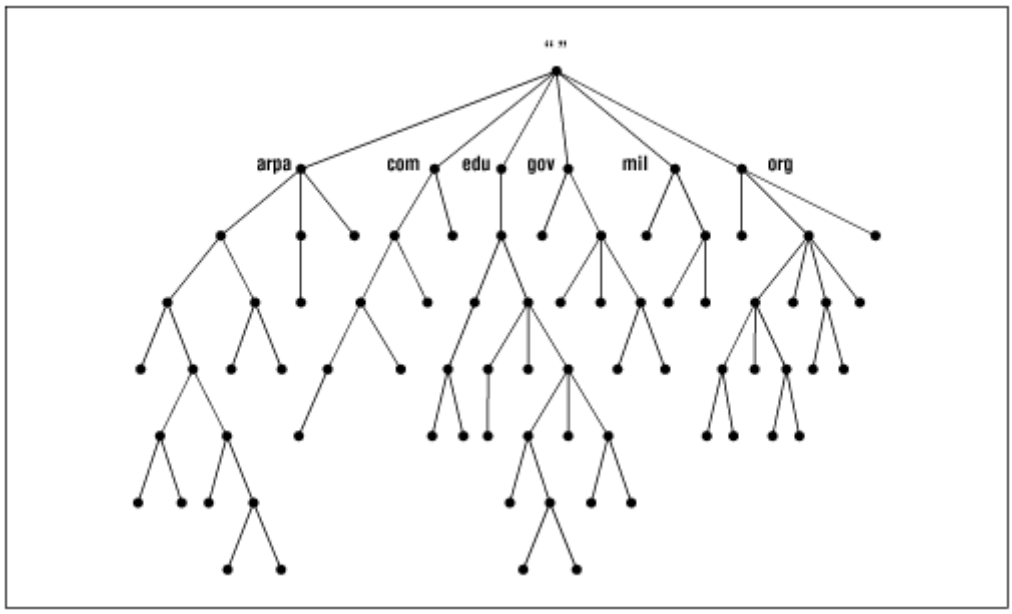
First, though, let's take a more detailed look at concepts introduced in the previous chapter. We'll try to add enough detail to spice it up a little.

2.1 The Domain Name Space

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 name space*. 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.[1] In the UNIX filesystem, this is called the root directory, 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, called a node. The depth of the tree is limited to 127 levels (a limit you're not likely to reach).

[1] Clearly this is a computer scientist's tree, not a botanist's.

Figure 2.1: The structure of the DNS name space



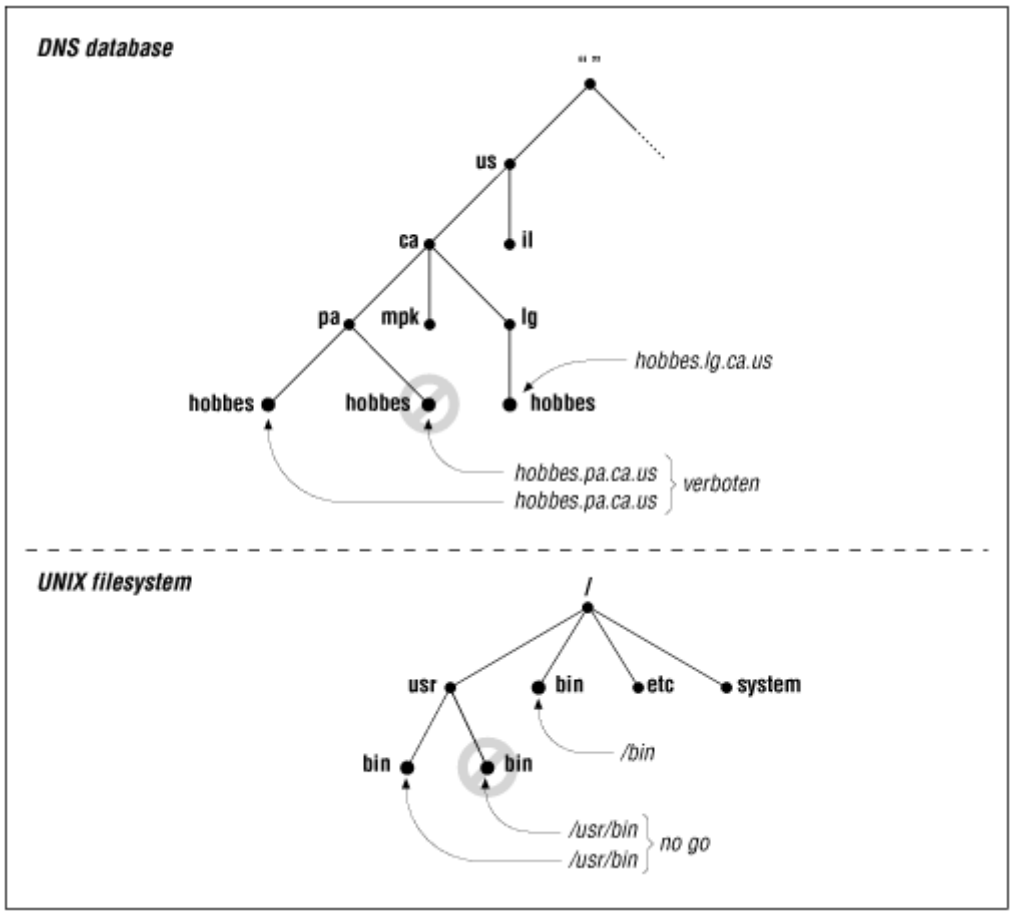
2.1.1 Domain Names

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), and 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 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 only need to be unique 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 the same name. Just as you can't have two *hobbes.pa.ca.us* nodes in the name space, you can't have two */usr/bin* directories (Figure 2.2). You can, however, have both a *hobbes.pa.ca.us* node and a *hobbes.lg.ca.us*, as you can have both a */bin* directory and a */usr/bin* directory.

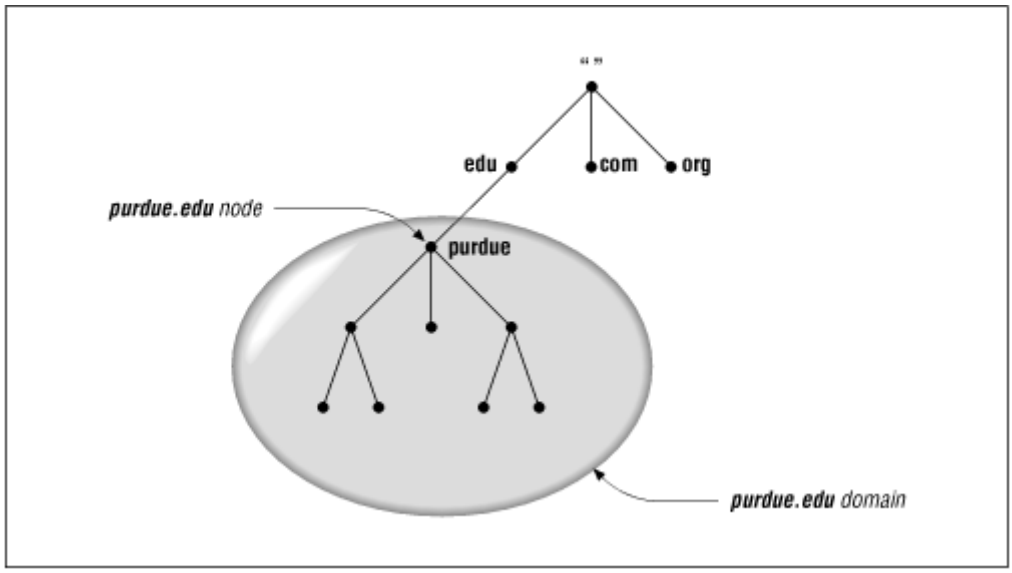
Figure 2.2: Ensuring uniqueness in domain names and in UNIX pathnames



2.1.2 Domains

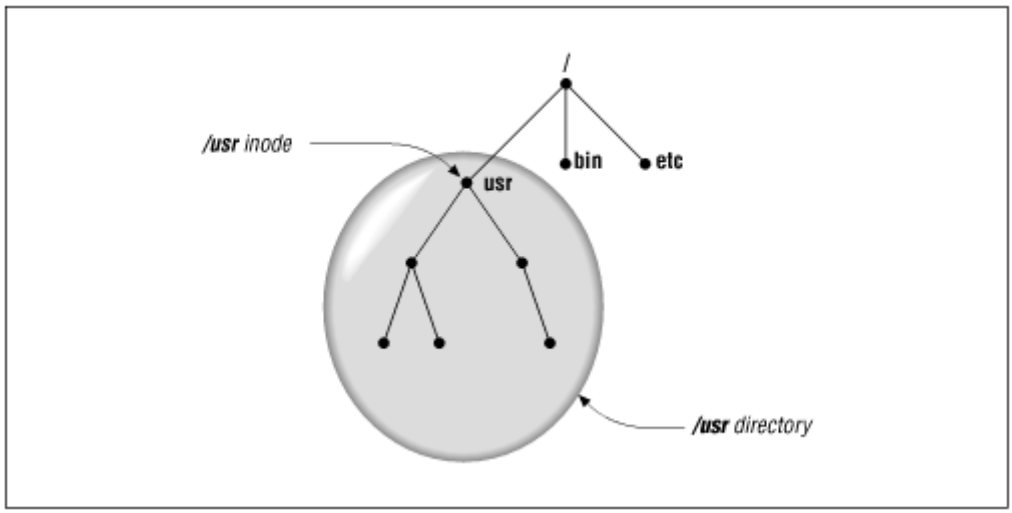
A *domain* is simply a subtree of the domain name space. The domain name of a domain is the same as the domain name of the node at the very top of the domain. So, for example, the top of the *purdue.edu* domain is a node named *purdue.edu*, as shown in Figure 2.3.

Figure 2.3: The purdue.edu domain



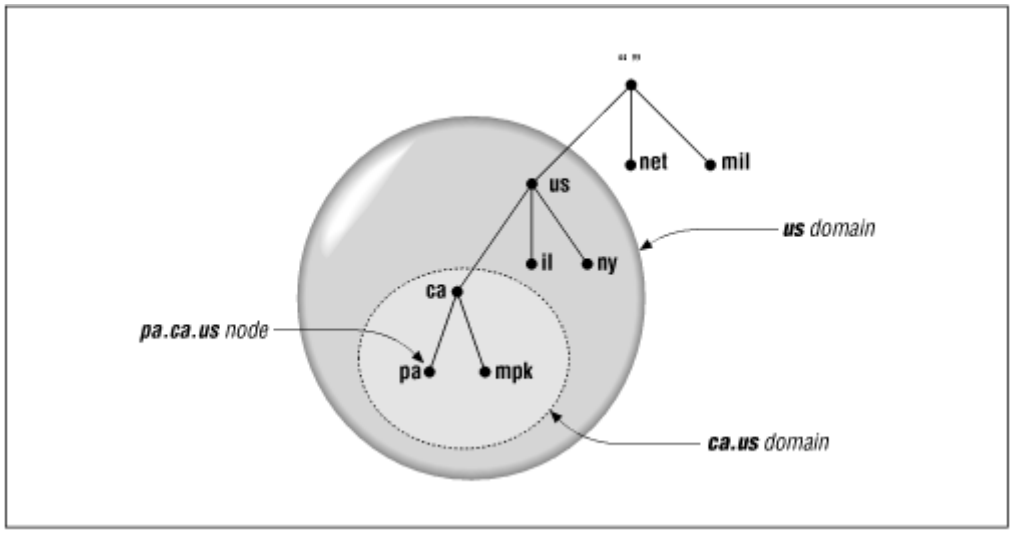
Likewise, in a filesystem, at the top of the */usr* directory, you'd expect to find a node called */usr*, as shown in Figure 2.4.

Figure 2.4: The /usr directory



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 2.5.

Figure 2.5: A node in multiple domains



So in the abstract, a domain is just a subtree of the domain name space. 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 ten different hosts, each of them on a different network and each one perhaps even in a different country, all in the same domain.[2]

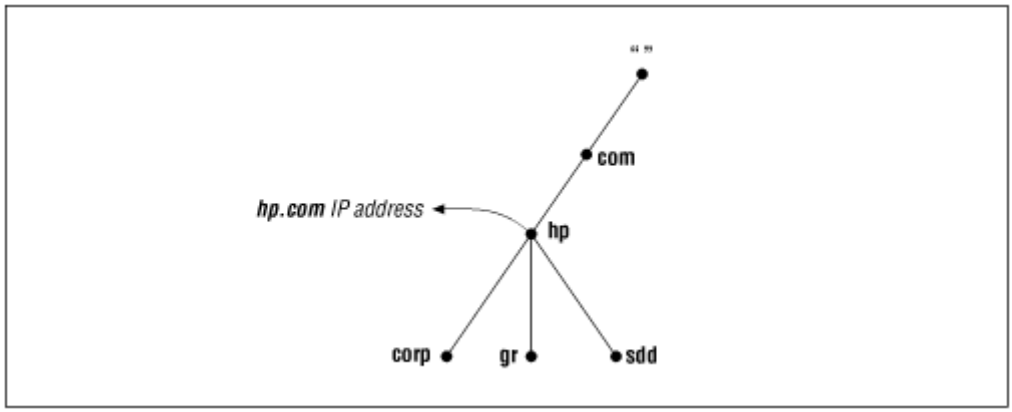
[2] One note of caution: Don't confuse domains in the Domain Name System with domains in Sun's Network Information Service (NIS). Though an NIS domain also refers to a group of hosts, and both types of domains have similarly structured names, the concepts are quite different. NIS uses hierarchical names, but the hierarchy ends there: hosts in the same NIS domain share certain data about hosts and users, but they can't navigate the NIS name space to find data in other NIS domains. NT domains, which provide account management and security services, also don't have any relationship to DNS domains.

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* can point to information about the domain. Interior domain names aren't restricted to one or the other. They 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 the domain name of a host that runs 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* would return mail routing information, while `telneting` to the domain name would look up the host information (in Figure 2.6, for example, *hp.com*'s IP address).[3]

[3] The terms *domain* and *subdomain* are often used interchangeably, or nearly so, in DNS and BIND documentation. Here, 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.

Figure 2.6: An interior node with both host and structural data



A simple way of deciding whether 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*. Similarly, it's 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 name space:

- 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.

2.2 The Internet Domain Name Space

So far, we've talked about the theoretical structure of the domain name space and what sorts of data are 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 particular level. When you manage a part of the domain name space, 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 name space, however, has some self-imposed structure to it. Especially in the upper-level domains, the domain names follow certain traditions (not rules, really, as they can be and have been broken.) These traditions help 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 name space 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

Networking organizations, such as NSFNET (*nsf.net*)

org

Noncommercial organizations, such as the Electronic Frontier Foundation (*eff.org*)

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 host names 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 the examples: all are primarily U.S. 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 domains are called *generic top-level domains*, or gTLDs. By the time you read this, we may have quite a few more of these, such as *firm*, *shop*, *web*, and *nom*, to accommodate the rapid expansion of the Internet and the need for more domain name "space." For more information on a proposal to create new gTLDs, see http://www.gtld-mou.org/.

To accommodate the internationalization of the Internet, the implementers of the Internet name space 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.[4] ISO 3166 establishes official, two-letter abbreviations for every country in the world. We've included the current list of top-level domains as Appendix C, *Top-Level Domains*, of this book.

[4] 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.

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 U.S.'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 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.

That's not true of the *us* top-level domain, however. The *us* domain has fifty subdomains that correspond to - guess what? - the fifty U.S. states.[5] Each is 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 is still largely geographical: most subdomains correspond to individual cities. Beneath the cities, the subdomains usually correspond to individual hosts.

[5] Actually, there are a few more domains under *us*: one for Washington, D.C., one for Guam, and so on.

2.2.3 Reading Domain Names

Now that you know what most top-level domains represent and how their name spaces 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've got 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 (in fact, we gave you this earlier, too). Their *corp* subdomain is undoubtedly their 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 knew your San Francisco Bay Area geography. (And no, it's not the same Menlo Park that Edison lived in - that one's in New Jersey.)

daphne.ch.apollo.hp.com

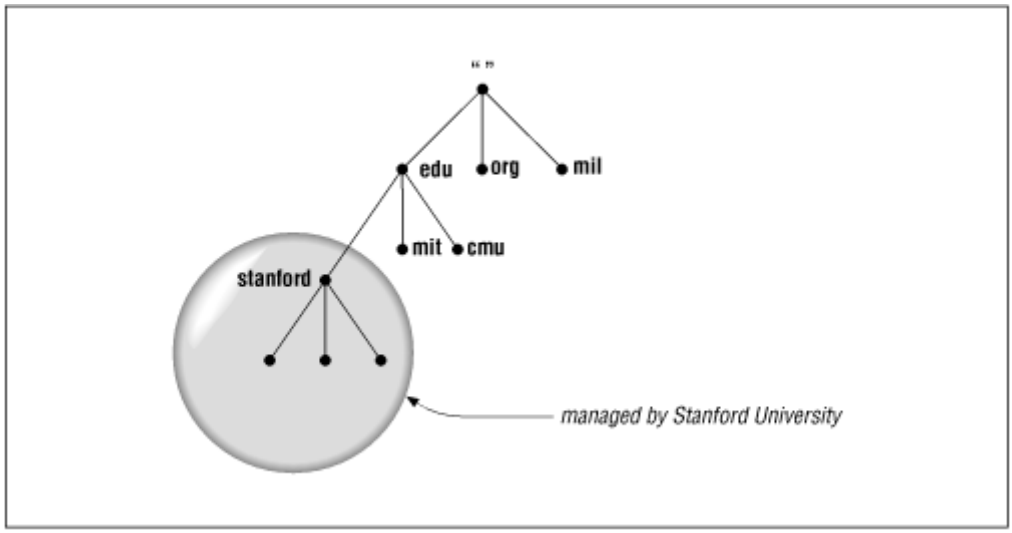
> We've included this example just so you don't start thinking that all domain names have only four labels. *apollo.hp.com* is the former Apollo Computer subdomain of the *hp.com* domain. (When HP acquired Apollo, it also acquired Apollo's Internet domain, *apollo.com*, which became *apollo.hp.com*.) *ch.apollo.hp.com* is Apollo's Chelmsford, Massachusetts, site. And *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 of those subdomains can be *delegated* to other organizations. This 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 up into more subdomains and delegate those. The parent domain contains 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 2.7).

Figure 2.7: stanford.edu is delegated to Stanford University



Not all organizations delegate away their whole domain, just as not all managers delegate all their work. A domain may have several subdomains and also 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 U.S. would fit better under *acme.com* than under either subdomain.

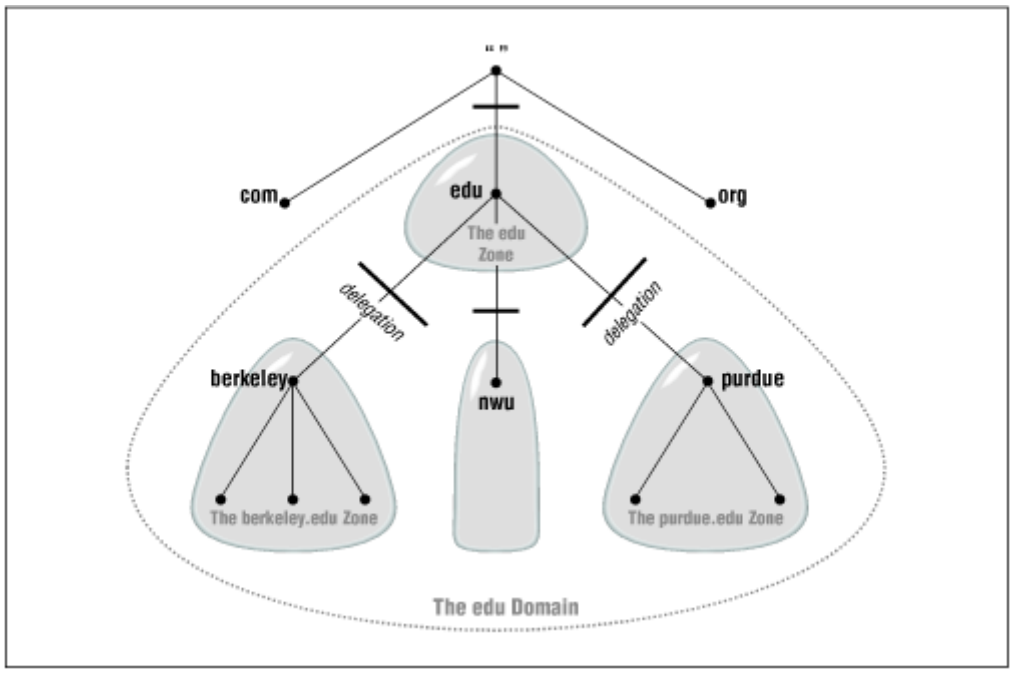
We'll explain how to create and delegate subdomains later. For now, it's only important that you understand that the term *delegation* refers to assigning responsibility for a subdomain to another organization.

2.4 Name Servers and Zones

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

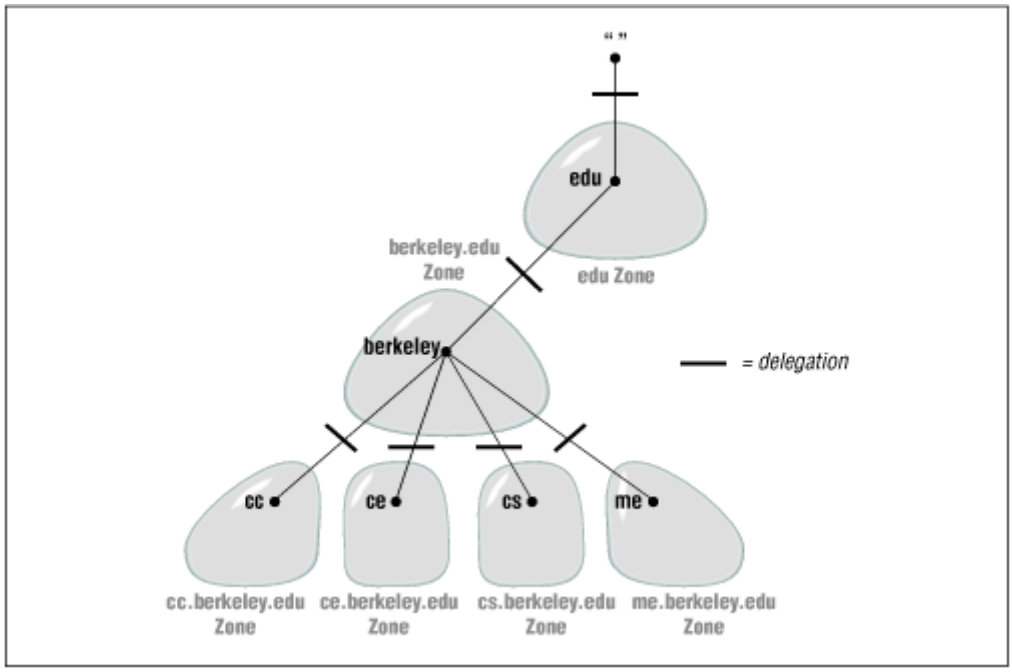
The difference between a zone and a domain is important, but subtle. All top-level domains, and many domains at the second level and lower, like *berkeley.edu* and *hp.com*, are broken into smaller, more manageable units by delegation. These units are called zones. The *edu* domain, shown in Figure 2.8, 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 would contain mostly delegation information to subdomains of *edu*.

Figure 2.8: The edu domain broken into zones



The *berkeley.edu* subdomain is, in turn, broken up into multiple zones by delegation, as shown in Figure 2.9. There are delegated subdomains called *cc*, *cs*, *ce*, *me*, and more. Each of these subdomains is delegated to a set of name servers, some of which are also authoritative for *berkeley.edu*. However, the zones are still separate, and may have a totally different group of authoritative name servers.

Figure 2.9: The berkeley.edu domain broken into zones



A zone contains the domain names that the domain with the same domain name contains, except for domain names in delegated subdomains. For example, the top-level domain *ca* (for Canada) may have the subdomains *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 name servers in each of the provinces. The *domain ca* contains all the data in *ca* plus all the data in *ab.ca*, *on.ca*, and *qc.ca*. But the *zone ca* contains only the data in *ca* (see Figure 2.10), which is probably mostly pointers to the delegated subdomains.

If a subdomain of the domain isn't delegated away, however, the zone contains the domain names and data in the subdomain. So the *bc.ca* and *sk.ca* (British Columbia and Saskatchewan) subdomains of the *ca* domain may exist, but might not be delegated. (Perhaps the provincial authorities in B.C. 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 name space 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 2.11.

Figure 2.10: The domain ca...

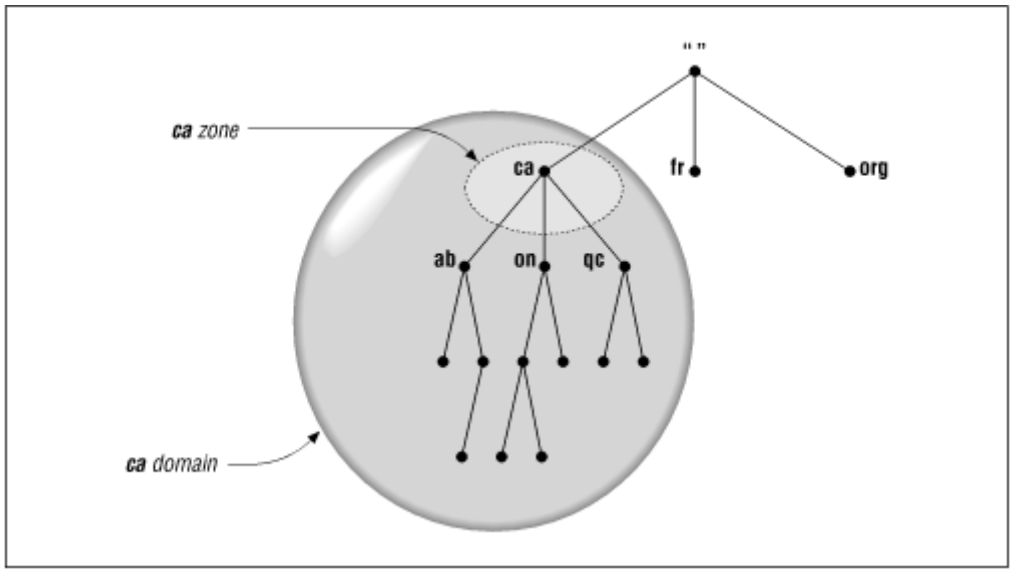
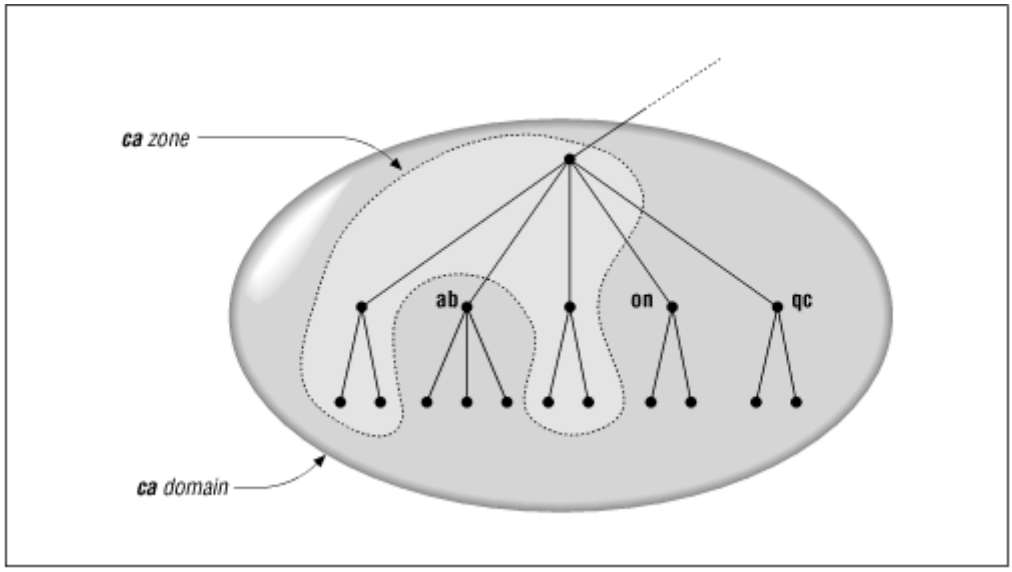


Figure 2.11: ...versus the zone ca



Now it's clear why name servers load zones instead of domains: a domain might contain more information than the name server would need.[6] A domain could contain data delegated to other name servers. Since a zone is bounded by delegation, it will never include delegated data.

[6] Imagine if a root name server loaded the root domain instead of the root zone: it would be loading the entire name space!

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

2.4.1 Delegating Domains

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 domain 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 your subdomains to different name servers. (Note that we said "name servers," not just "name server.")

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

2.4.2 Types of Name Servers

The DNS specs define two types of name servers: *primary masters* and *secondary masters*. A *primary master* name server for a zone reads the data for the zone from a file on its host. A *secondary master* name server for a zone gets the zone data from another name server that is 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 name server and, if necessary, pulls the zone data over. This is referred to as a *zone transfer*. Nowadays, the preferred term for a secondary master name server is a *slave*, though many people (and much software, including Microsoft's DNS Manager) still call them secondaries.

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

Slave name servers are important because it's a good idea to set up more than one name server 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 name server close by. Using slave name servers makes this administratively workable.

Calling a *particular* name server a primary master name server or a slave name server is a little imprecise, though. We mentioned earlier that a name server can be authoritative for more than one zone. Similarly, a name server can be a primary master for one zone and a slave for another. Most name servers, 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 name server a primary or a slave, we mean that it's the primary master or a slave for *most* of the zones it loads.

2.4.3 Data Files

The files from which primary master name servers load their zone data are called, simply enough, zone data files or just data files. We often refer to them as *db files*, short for *database files*. Slave name servers can also load their zone data from data files. Slaves are usually configured to back up the zone data they transfer from a master name server to data files. If the slave is later killed and restarted, it will read the backup data files first, then check to see

whether the data are 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 data files contain resource records that describe the zone. The resource records describe all the hosts in the zone and mark any delegation of subdomains. BIND also allows special directives to include the contents of other data files in a data file, much like the `#include` statement in C programming.

2.5 Resolvers

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

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

In BIND, the resolver is just a set of library routines that is linked into programs such as `telnet` and `ftp`. It's not even a separate process. 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 name server. The DNS specs call this kind of resolver a *stub resolver*.

Other implementations of DNS have had smarter resolvers, which can do more sophisticated things such as build up a cache of information already retrieved from name servers.[7] But these aren't nearly as common as the stub resolver implemented in BIND.

2.6 Resolution

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

Because the name space is structured as an inverted tree, a name server needs only one piece of information to find its way to any point in the tree: the domain names and addresses of the root name servers (is that more than one piece?). A name server can issue a query to a root name server for any name in the domain name space, and the root name server will start the name server on its way.

2.6.1 Root Name Servers

The root name servers know where there are authoritative name servers for each of the top-level domains. (In fact, most of the root name servers *are* authoritative for the generic top-level domains.) Given a query about any domain name, the root name servers can at least provide the names and addresses of the name servers that are authoritative for the top-level domain that the domain name is in. And the top-level name servers can provide the list of name servers that are authoritative for the second-level domain that the domain name is in.

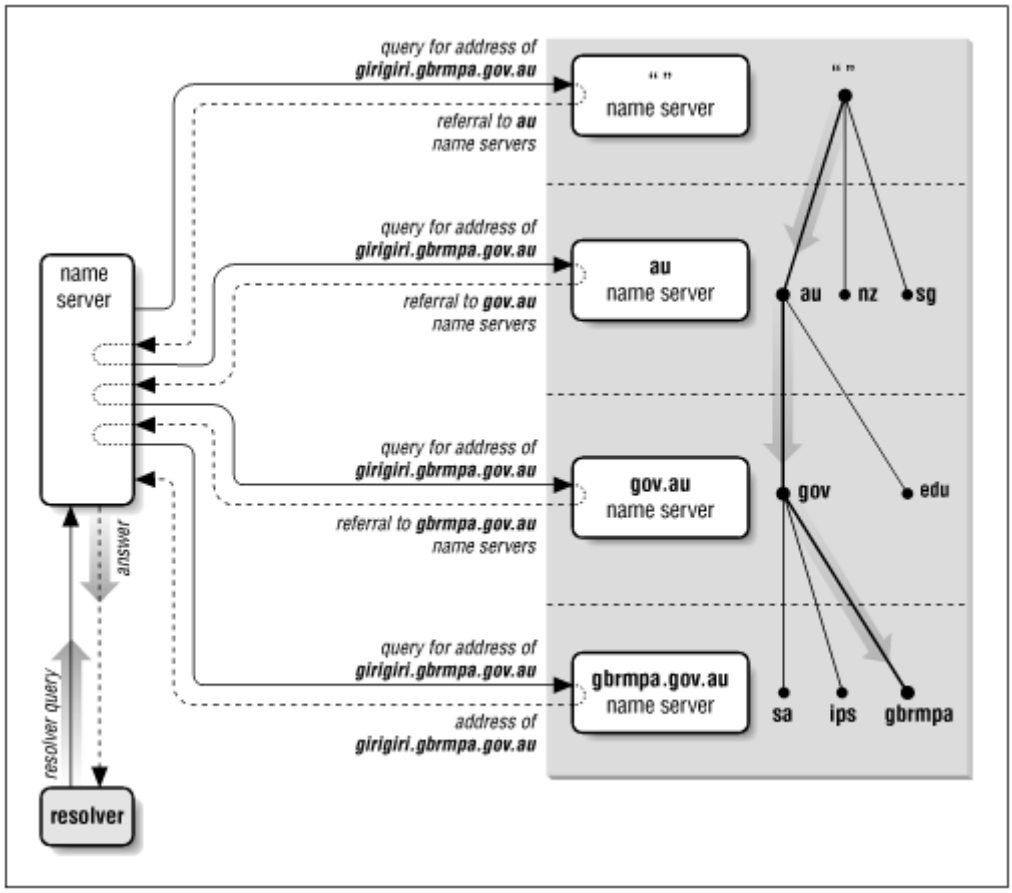
Each name server queried gives the querier information about how to get "closer" to the answer it's seeking, or it provides the answer itself.

The root name servers 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 name servers. But in the absence of other information, resolution has to start at the root name servers. This makes the root name servers crucial to the operation of DNS; if all the Internet root name servers were unreachable for an extended period, all resolution on the Internet would fail. To protect against this, the Internet has thirteen root name servers (as of this writing) spread across different parts of the network. Two are on the MILNET, the U.S. military's portion of the Internet; one is on SPAN, NASA's internet; two are in Europe; and one is in Japan.

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

Despite the load placed on root name servers, resolution on the Internet works quite well. Figure 2.12 shows the resolution process for the address of a real host in a real domain, including how the process corresponds to traversing the domain name space tree.

Figure 2.12: Resolution of girigiri.gbrmpa.gov.au on the Internet



The local name server queries a root name server for the address of *girigiri.gbrmpa.gov.au* and is referred to the *au* name servers. The local name server asks an *au* name server the same

question, and is referred to the *gov.au* name servers. The *gov.au* name server refers the local name server to the *gbrmpa.gov.au* name servers. Finally, the local name server asks a *gbrmpa.gov.au* name server 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 name servers in the previous example. Four of the name servers simply returned the best answer they already had - 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 name server - the one queried by the resolver - had to follow successive referrals until it received an answer.

Why couldn't the local name server simply have referred the resolver to another name server? Because a stub resolver wouldn't have had the intelligence to follow a referral. And how did the name server 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 name server. *Recursion*, or *recursive resolution*, is just a name for the resolution process used by a name server when it receives recursive queries.

Iteration, or *iterative resolution*, on the other hand, refers to the resolution process used by a name server when it receives iterative queries.

In recursion a resolver sends a recursive query to a name server for information about a particular domain name. The queried name server is then obliged to respond with the requested data or with an error stating that data of the requested type don't exist or that the domain name specified doesn't exist.[8] The name server can't just refer the querier to a different name server, because the query was recursive.

[8] The BIND 8 name server can be configured to refuse recursive queries; see Chapter 10, *Advanced Features and Security*, for how and why you'd want to do this.

If the queried name server isn't authoritative for the data requested, it will have to query other name servers to find the answer. It could send recursive queries to those name servers, thereby obliging them to find the answer and return it (and passing the buck). Or it could send iterative queries and possibly be referred to other name servers "closer" to the domain name it's looking for. Current implementations are polite and do the latter, following the referrals until an answer is found.[9]

[9] The exception is a name server configured to forward all unresolved queries to a designated name server, called a *forwarder*. See Chapter 10 for more information on using forwarders.

A name server that receives a recursive query that it can't answer itself will query the "closest known" name servers. The closest known name servers are the servers authoritative for the zone closest to the domain name being looked up. For example, if the name server receives a recursive query for the address of the domain name *girigiri.gbrmpa.gov.au*, it will first check whether it knows the name servers for *girigiri.gbrmpa.gov.au*. If it does, it will send the query

to one of them. If not, it will check whether it knows the name servers 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, since every name server knows the domain names and addresses of the root name servers.

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

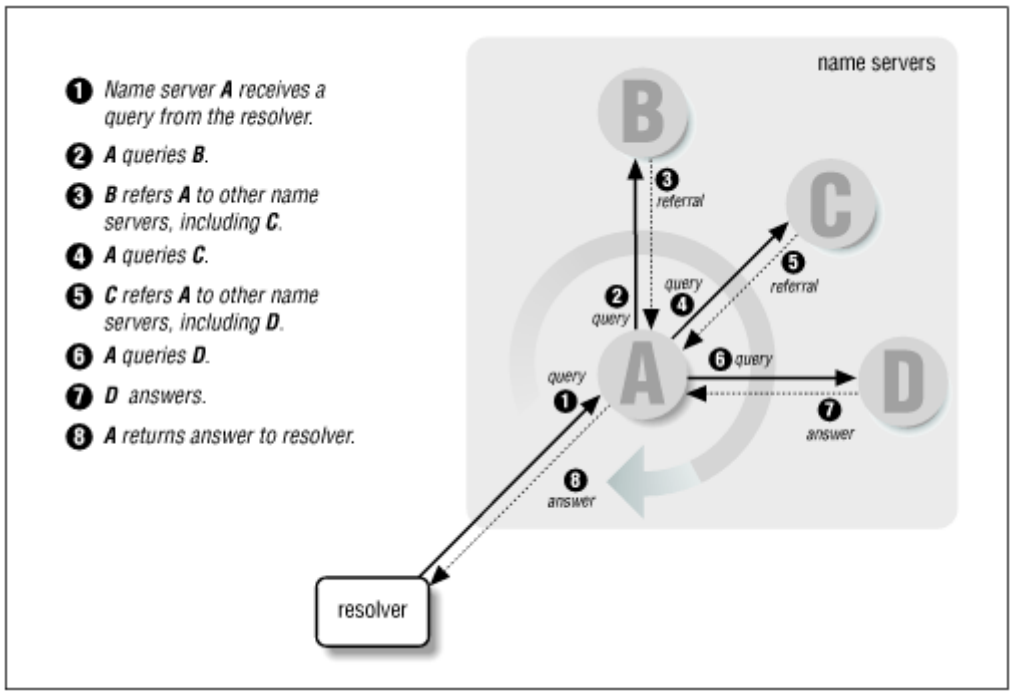
The name server that receives the recursive query always sends the same query that the resolver sends it, for example, for the address of *waxwing.ce.berkeley.edu*. It never sends explicit queries for the name servers for *ce.berkeley.edu* or *berkeley.edu*, though this information is also stored in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

2.6.3 Iteration

Iterative resolution, on the other hand, doesn't require nearly as much work on the part of the queried name server. In iterative resolution, a name server simply gives the best answer *it already knows* back to the querier. No additional querying is required. The queried name server consults its local data (including its cache, which we're about to talk about), looking for the data requested. If it doesn't find the data there, it makes its best attempt to give the querier data that will help it continue the resolution process. Usually these are the domain names and addresses of the closest known name servers.

What this amounts to is a resolution process that, taken as a whole, looks like Figure 2.13.

Figure 2.13: The resolution process



A resolver queries a local name server, which then queries a number of other name servers in pursuit of an answer for the resolver. Each name server it queries refers it to another name server that is authoritative for a zone further down in the name space and closer to the domain name sought. Finally, the local name server queries the authoritative name server, which returns an answer.

2.6.4 Mapping Addresses to Names

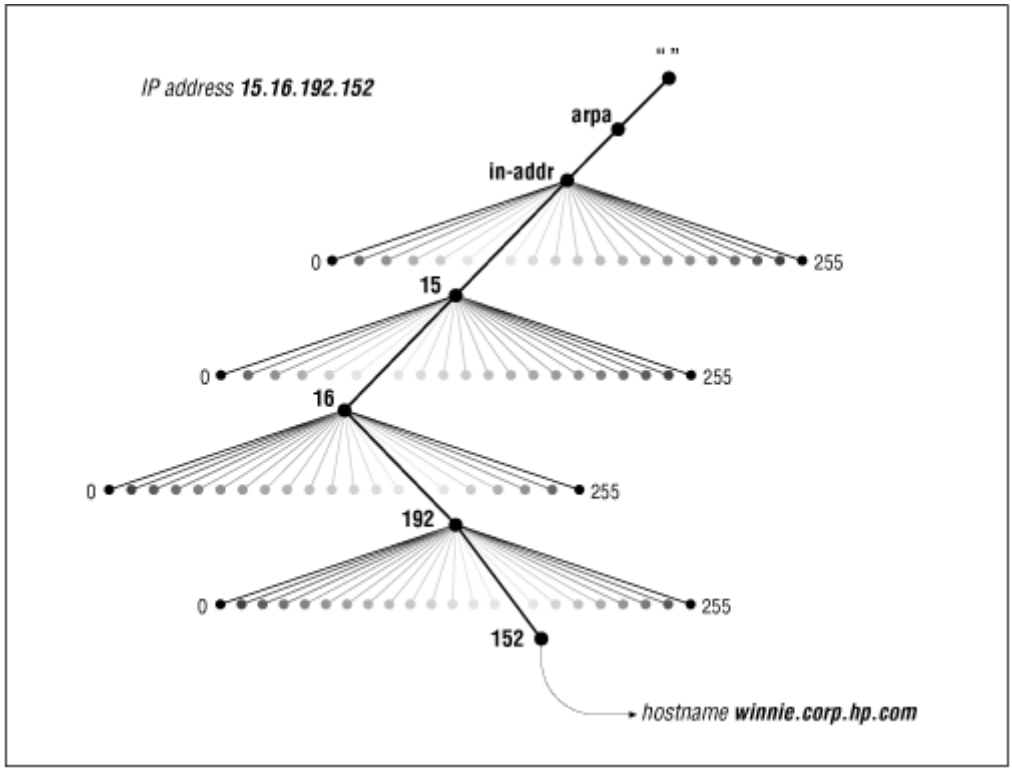
One major piece of functionality missing from the resolution process as explained so far is how addresses get mapped back to names. Address-to-name mapping is used to produce output that is easier for humans to read and interpret (in log files, for instance). It's also used in some authorization checks. 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 host name listed. In DNS, however, address-to-name mapping isn't so simple. Data, including addresses, in the domain name space are 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 name space that uses addresses as labels? In the Internet's domain name space, this portion of the name space is the *in-addr.arpa* domain.

Nodes in the *in-addr.arpa* domain are labelled after 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, could have up to 256 subdomains, one corresponding to each possible value in the first octet of an IP address. Each of these subdomains could 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 or network at that IP address. That makes for an awfully big domain: *in-addr.arpa*, shown in Figure 2.14, is roomy enough for every IP address on the Internet.

Figure 2.14: addr.arpa domain

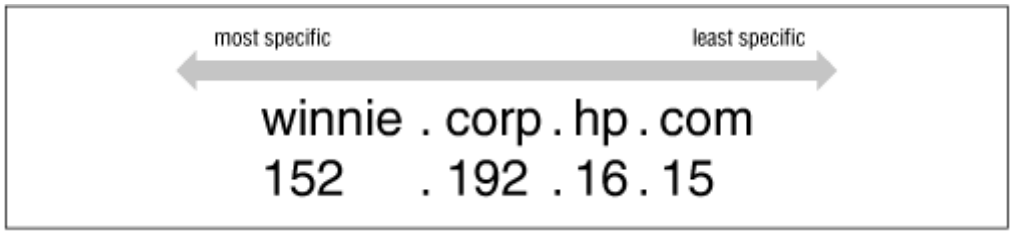


Note that when read in a domain name, the IP address appears backward because the name is read from leaf to root. For example, if *winnie.corp.hp.com*'s IP address is 15.16.192.152, the corresponding *in-addr.arpa* subdomain is *152.192.16.15.in-addr.arpa*, which maps back to the domain name *winnie.corp.hp.com*.

IP addresses could have been represented the opposite way in the name space, 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 2.15 shows what we mean.

Figure 2.15: Hierarchical names and addresses



Making the first octets in the IP address appear highest in the tree gives administrators the ability to delegate authority for *in-addr.arpa* domains along network lines. For example, the *15.in-addr.arpa* domain, 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 domain to try to delegate.

2.6.5 Inverse Queries

The *in-addr.arpa* name space is clearly only useful for IP address-to-domain name mapping. Searching for a domain name that indexes an *arbitrary* piece of data - something besides an address - in the domain name space would require another specialized name space like *in-addr.arpa* or an exhaustive search.

That exhaustive search is to some extent possible, and it's called an *inverse query*. An inverse query is a search for the domain name that indexes a given datum. It's processed solely by the name server receiving the query. That name server searches all of its local data for the item sought and returns the domain name that indexes it, if possible. If it can't find the data, it gives up. No attempt is made to forward the query to another name server.

Because any one name server only knows about part of the overall domain name space, an inverse query is never guaranteed to return an answer. For example, if a name server receives an inverse query for an IP address it knows nothing about, it can't return an answer, but it also doesn't know that the IP address doesn't exist, because it only holds part of the DNS database. What's more, the implementation of inverse queries is optional according to the DNS specification; BIND 4.9.7 still contains the code that implements inverse queries, but it's commented out by default. BIND 8 no longer includes that code at all, though it does recognize inverse queries and can make up fake responses to them.[10] That's fine with us, because very little software (such as archaic versions of `nslookup`) actually still uses inverse queries.

2.7 Caching

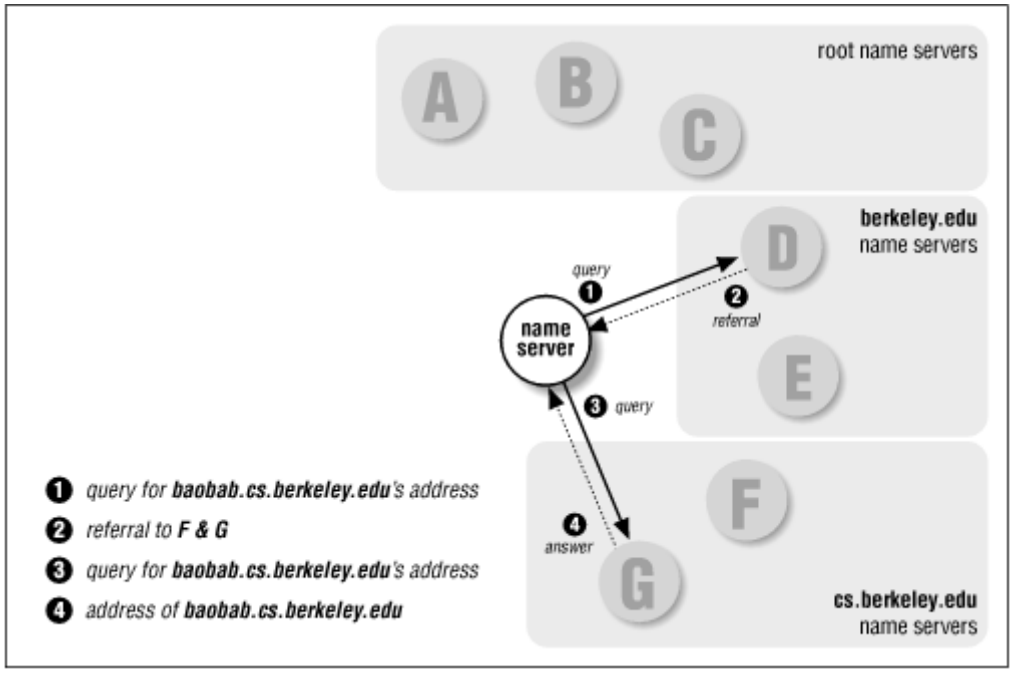
The whole resolution process may seem awfully convoluted and cumbersome to someone accustomed to simple searches through the host table. Actually, it's usually quite fast. One of the features that speeds it up considerably is *caching*.

A name server 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 name space as it does so. Each time it's referred to another list of name servers, it learns that those name servers are authoritative for some zone, and it learns the addresses of those servers. And, 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. With version 4.9 and all version 8 BINDs, name servers even implement *negative caching*: if an authoritative name server responds to a query with an answer that says the domain name or data type in the query doesn't exist, the local name server will temporarily cache that information, too. Name servers cache all of this data to help speed up successive queries. The next time a resolver queries the name server for data about a domain name the name server knows something about, the process is shortened quite a bit. The name server 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 name servers that are authoritative for the zone the domain name is in and be able to query them directly.

For example, say our name server 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* name servers (plus *eecs.berkeley.edu*'s IP address). Now if a resolver were to query our name server for the address of *baobab.cs.berkeley.edu*, our name server could skip querying the root name servers. Recognizing that *berkeley.edu* is the closest ancestor of *baobab.cs.berkeley.edu* that it knows about, our name server would start by querying a *berkeley.edu* name server, as shown in Figure 2.16. On the other hand, if our name server had discovered that there was no address for *eecs.berkeley.edu*, the next time it received a query for the address, it could simply have responded appropriately from its cache.

Figure 2.16: Resolving baobab.cs.berkeley.edu



In addition to speeding up resolution, caching prevents us from having to query the root name servers again. This means that we're not as dependent on the roots, and they won't suffer as much from all our queries.

2.7.1 Time to Live

Name servers can't cache data forever, of course. If they did, changes to that data on the authoritative name servers would never reach the rest of the network. Remote name servers would just continue to use cached data. Consequently, the administrator of the zone that contains the data decides on a *time to live*, or *TTL*, for the data. The time to live is the amount of time that any name server is allowed to cache the data. After the time to live expires, the name server must discard the cached data and get new data from the authoritative name servers. This also applies to negatively cached data; a name server must time out a negative answer after a period, too, in case new data has been added on the authoritative name servers. However, the time to live for negatively cached data isn't tunable by the domain administrator; it's hardcoded to ten minutes.

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

A large TTL will shorten the average time it takes to resolve information in your domain because the data can be cached longer. The drawback is that your information will be inconsistent for a longer time if you make changes to your data on your name servers.

Enough of this theory - I'll bet you're antsy to get on with this. There's some homework necessary before you can set up your domain and your name servers, though, and we'll assign it in the next chapter.