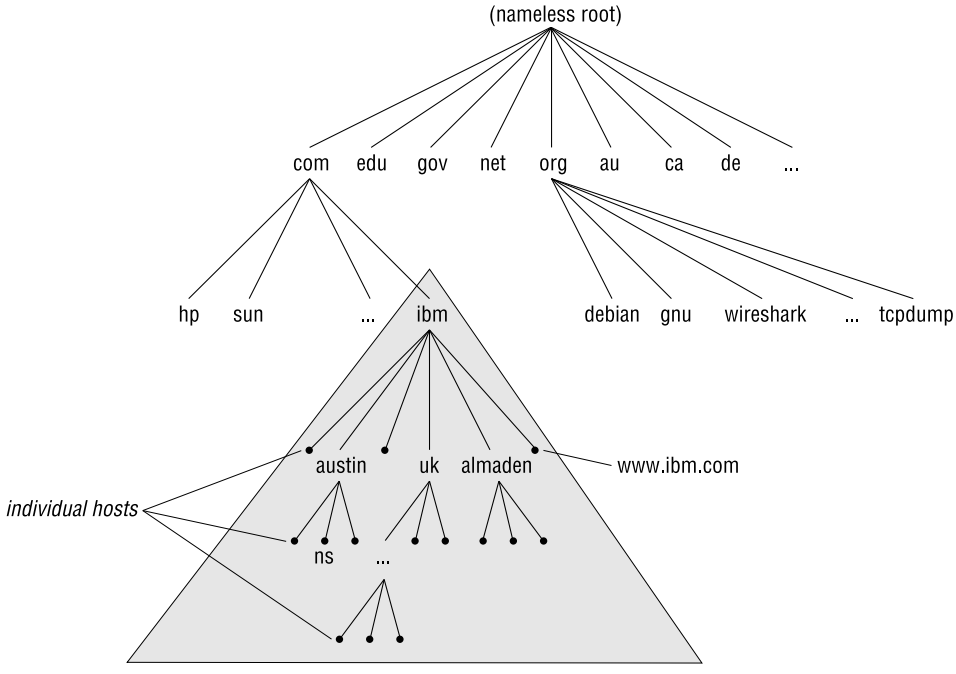

Figure 1.2: Tree structure of DNS names

`/usr/local/doc/readme.txt`). DNS naming works in much the same way. The basic name of a node in the DNS tree is called its *simple name*.

A node's *fully-qualified domain name* (*FQDN*), i.e. its full name, is its simple name followed by the names of each of its parents in turn, separated by dots instead of slashes. In Figure 1.2 the FQDN of the node ns is ns.austin.ibm.com. Note that in DNS the most significant parts are on the right, whereas in a file pathname they are on the left.

An individual node in the DNS tree is also called a *label*; it's limited to 63 characters and must not contain a period (just as in the *nix file system a filename must not contain a slash). The depth of the DNS tree is limited to 127 levels, and a fully qualified domain name is limited to 255 characters (including the dot separators).

A *domain* is a node in the naming tree, plus all its children, grandchildren, etc., if it has any; its domain name is the full name of the node. E.g. ibm.com is the name of the domain consisting of the whole sub-tree highlighted in gray in Figure 1.2, including the ibm node itself, right down to the individual machines. Your DNS domain is a specific part of the name space that is dedicated to you, so you can create your own names without clashing with anyone else's.

A hierarchy or tree structure avoids name clashes if you simply insist that no two children of the same parent have the same name. E.g. two different children of .com can't have the same name, so you're not allowed to have two separate domains called ibm.com and ibm.com. However, IBM can have a machine called www and you can have one called

5. Our server 192.68.1.164 sends, to the client in our desktop PC, the address 192.168.1.20 as the answer to the query for domain name www.qupps.biz.

In other words, our server repeatedly sends queries to carefully chosen name servers, and gets progressively more information about www.qupps.biz each time, until finally it contacts the server that either has the information or can definitely say that there is no such domain name. (Looking at it the other way round, the remote servers queried in Steps 2 and 3 replied: "I don't know what you want, but here's a man who probably does know – ask him instead".)

Now we're going to go through each of the above steps in more detail, looking at how the various DNS servers and other components perform their functions.

1.1.5 Step 1 – the resolver

The *resolver* is the DNS client code that an application program uses to perform DNS queries. "The resolver" is not a separate program. (And later on, when we talk about a "DNS client", the client isn't a separate program either – it's just the resolver in the application program that wants to use the DNS.)

Where the resolver code resides depends on the implementation.

- On *nix the resolver is a set of functions in a library linked into the program (Figure 1.4). The *nix resolver does *not* remember (or "cache") any names or addresses it looked up in the past. When an application program uses the resolver to resolve www.qupps.biz a name, the resolver sends a query over the network to the DNS server, and waits for the answer. If the application immediately calls the resolver again to resolve www.qupps.biz, the resolver will send another query over the network, and wait for its reply, which might not be what you want on a very busy application. We'll see later (Section 1.1.6 on page 10) how you can get over this. This kind of resolver is called a *stub resolver*; it doesn't really do the work of resolution itself but hands it off to to a DNS server that does.

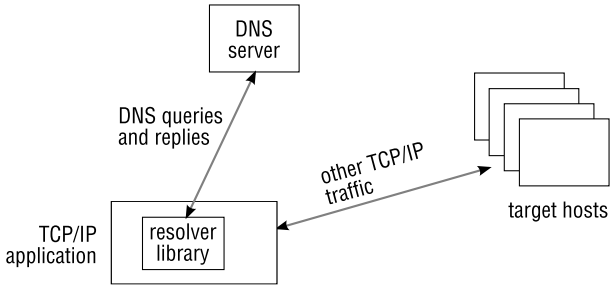

Figure 1.4: The resolver on *nix is implemented as a library

- On Microsoft Windows, resolution is performed in a similar fashion. An application invokes functions in a dynamic link library (DLL) which hands off the work of performing resolution to one or more name servers. When the server(s) return an answer,

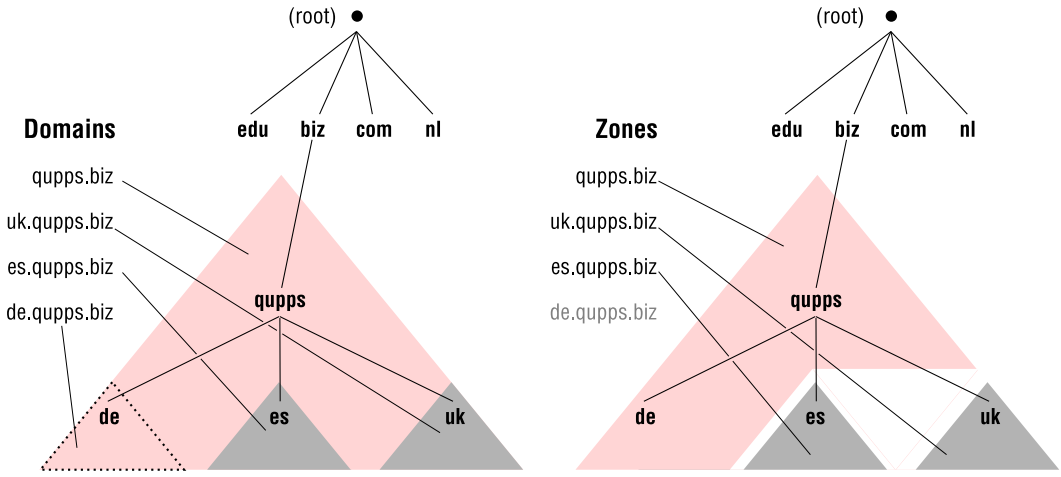

Figure 1.7: DNS domains and zones

1.2.2 Creating redundancy – master/slave, primary/secondary servers

What happens when the authoritative name server for your domain is down and cannot answer queries? Client hosts elsewhere on the Internet wishing to connect to your Web and e-mail servers can't find those servers' addresses: your site is effectively down, even though only the DNS server is unavailable.

To provide reliable service you want redundant servers – i.e. more than one server with identical content. You must also ensure that these servers give identical answers to identical queries. (Imagine your domain qupps.biz has two DNS name servers, but they supply different addresses as answers to queries for www.qupps.biz!) There are two different ways to ensure that all servers have the same content and keep in sync:

A. Set up "master" and "slave" name servers, with some mechanism for the primary to copy its data to the secondary, to keep it in sync.

B. Set up name servers that access a replicated database store. In this case it is the database software, not the name servers, that replicate the zone data.

Now let's look at each of these in detail. (Before we go on, remember that this only applies to authoritative servers; caching servers don't maintain zone data, so there are no issues about getting out of sync.)

A. Master and slave name servers

The QUPPS administrators have decided to set up the name server in Spain so it can also serve the DNS data normally held in the zones in Germany (Figure 1.8), in order to give their DNS service greater resilience. They do this by configuring the authoritative name server located in Germany to be a "master" name server and the server in Spain to be a "slave" name server.

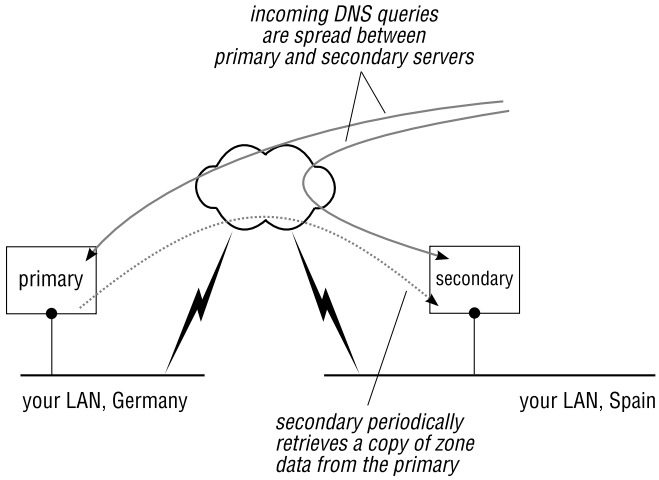

Figure 1.8: Secondary or slave name server

A *master name server* (or *primary name server*) is one that holds the definitive copy of information for one or more zones, which it uses to answer queries from clients. By contrast, a *slave name server* or *secondary name server* is never directly configured with the zone data. Instead, all changes to DNS data are made on the master name server, and the slave transfers this data from its master. A master can have multiple slaves.

In all cases, both master and slave server(s) are authoritative for the zone data. As far as authority is concerned, there is no difference between the master and the slave server(s) of a zone.

The data is transferred from master to slave by *zone transfer*, which duplicates the data from the master to the slave. A zone transfer is a special type of DNS request named AXFR, which allows master and slave to use the DNS protocol as the interchange mechanism, without having to add other special transfer facilities. In contrast to most DNS requests which are performed over UDP, zone transfers are carried out over TCP. Zone transfers performed by a slave server are called incoming or inward transfers. Zone transfers provided by master servers are called outgoing transfers. Note that whenever we say zone transfer or AXFR we use the term strictly (i.e. we mean a zone transfer using the DNS AXFR request) and not as a generic term for replication of DNS data from one name server to another.

There are two ways a zone transfer can be initiated:

1. A slave server periodically checks with its master server whether the zone has changed, by querying a special DNS record called the Start of Authority (Chapter 2). If this check indicates that the zone's content has changed, the slave transfers it by sending an AXFR query to the master server, requesting a copy of the zone data. This zone is then "downloaded" to the slave server via the DNS. Some server implementations offer so-called *incremental zone transfers*, which allow a slave server to receive only the changed DNS zone data, instead of a full copy of the zone.

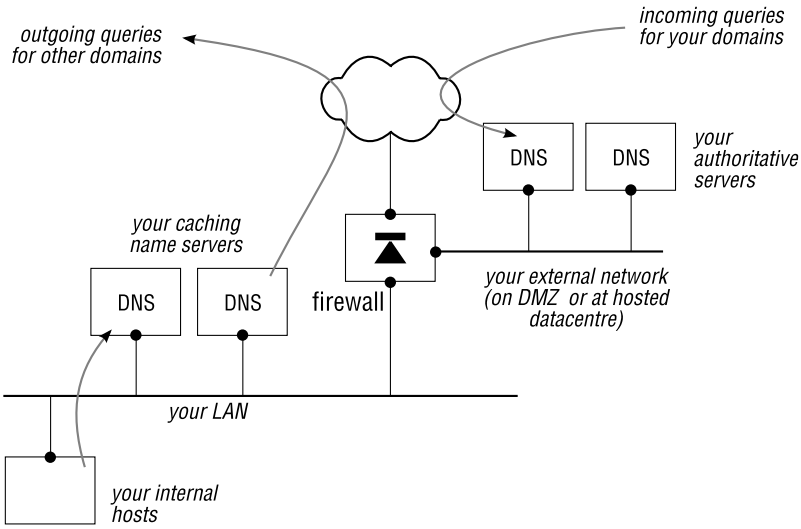

Figure 1.14: A typical setup for a small organization

from spreading their name servers over diverse geographical areas wherever possible, it is not unusual to find name servers on a departmental level. Large organizations often have a mixture of name server brands deployed, and many of the programs we discuss in Part II are sensible choices. Some suggestions:

- Bind DLZ integrates DNS with your LDAP directory server or SQL database environment.
- Bind DLZ with the Berkeley DB High Performance back-end provides database and high performance simultaneously.
- PowerDNS offers a large choice of back-end stores for DNS data, and you can also set it up to be master or slave server.
- Unbound or PowerDNS Recursor installed on separate machines provide your environment with fast recursive resolvers.
- BIND offers an incredible list of features and is often deployed in large organizations.
- If you require great performance, consider the NSD authoritative server.

A large number of combinations of the programs we discuss are possible, and many of them make good sense; the choice is yours. As an example, the DNS infrastructure for a real-life large corporation is shown in detail in the Notes.

This concludes our fast-forwarding over the terminology we'll be using throughout the book. In the next chapter, we discuss how you configure zone data and how different brands of server store it.

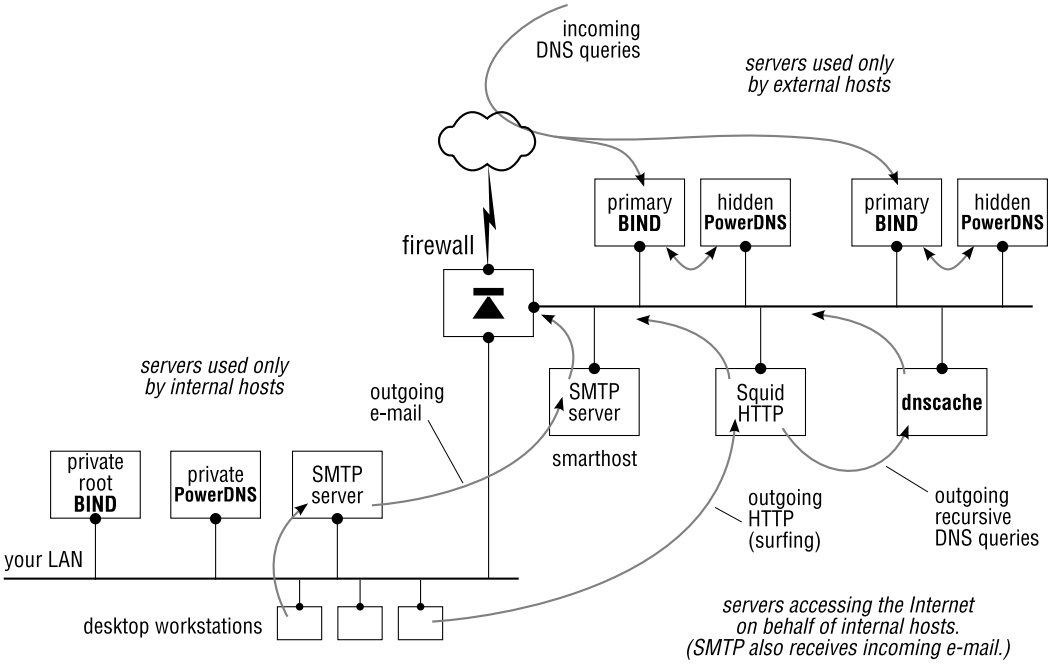

Figure 1.15: Sample of corporate DNS infrastructure

Practical TCP/IP

We highly recommend the companion book in this series, *Practical TCP/IP* by Niall Mansfield. This book is a must-have if you want to have a detailed insight into the workings of TCP/IP and its related protocols. The author dissects the protocols, shows you the content of data packets and gives hands-on tips on how to configure individual services on your network. The book contains three chapters on DNS.

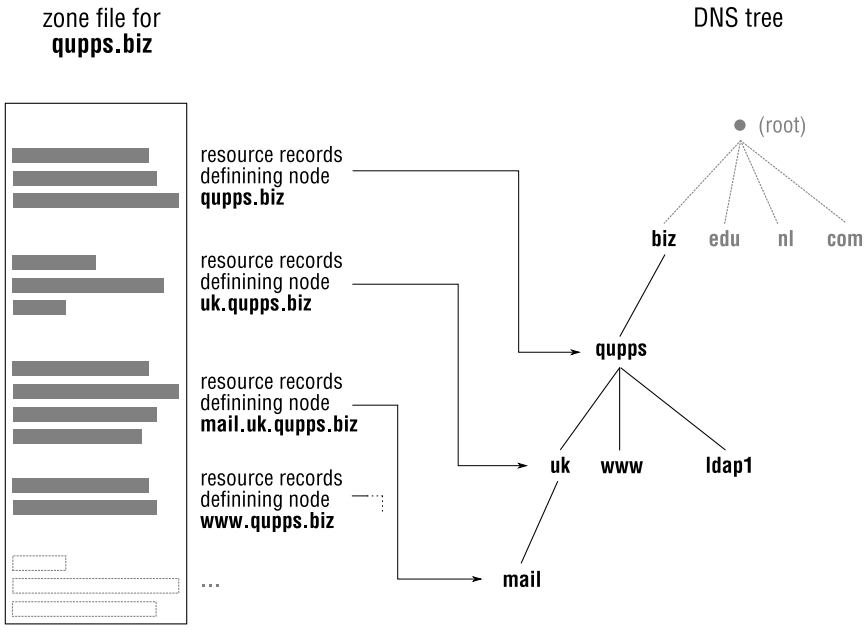

Figure 2.1: Resource records in the zone file define the nodes in the DNS tree

- If it finds any matching records, it returns them in the reply. dig shows them, one per line, in its ANSWER SECTION, and also shows the number of ANSWERs, in its `;; flags` output line.
- If there are no matching records, the server returns a reply containing zero ANSWERs.

Even if there were no matching records, dig outputs `status:NOERROR`: the server successfully found the requested domain, but it happened not to have any RRs of the type we were interested in.

Now we're going to explain in detail the types of resource records you're likely to come across, their contents, and what they are used for. But before we do that, we need to look at the format that's common to all resource records.

2.3.1 The format of a Resource Record

A resource record consists of five fields:

name *TTL* *class* *type* *rdata*

name The name of the domain that this resource record describes. This name is sometimes called an *owner* or the *origin*.

nor implemented or perhaps even get lost in transit, the slave server will attempt to transfer the zone automatically. We recommend you use SOA values as in the examples above.

Name Server (NS) resource records

The resolution example in Section 1.1.4 showed a series of authoritative name servers being queried in turn, to obtain the final answer for a query. The root server had delegated .biz to the .biz servers, which in turn delegated qupps.biz to QUPPS. Delegation is implemented by means of *Name Server* (NS) resource records. The zone file on the parent server contains, in NS records, the addresses of the name servers that the child domain is delegated to (i.e. the NS records contain the names of the servers authoritative for the child domain). Figure 2.2 shows the three delegations from the root to es.qupps.biz.

There can, and should, be more than one Name Server resource record in a zone, each containing the host name of a different authoritative server. As we said in Section 1.2.2, you typically want to create resilience for your name servers, and there are DNS registries that insist on having more than one name server (e.g. the DENIC in Germany).

The name server specified in an NS record must be a canonical name (i.e. a real hostname as you define with an A RR), not an IP address or an alias.

```
name            TTL     class   type    rdata
qupps.biz.      3600    IN      NS      ns1.qupps.biz.
qupps.biz.      3600    IN      NS      dns.my-isp.example.net.
```

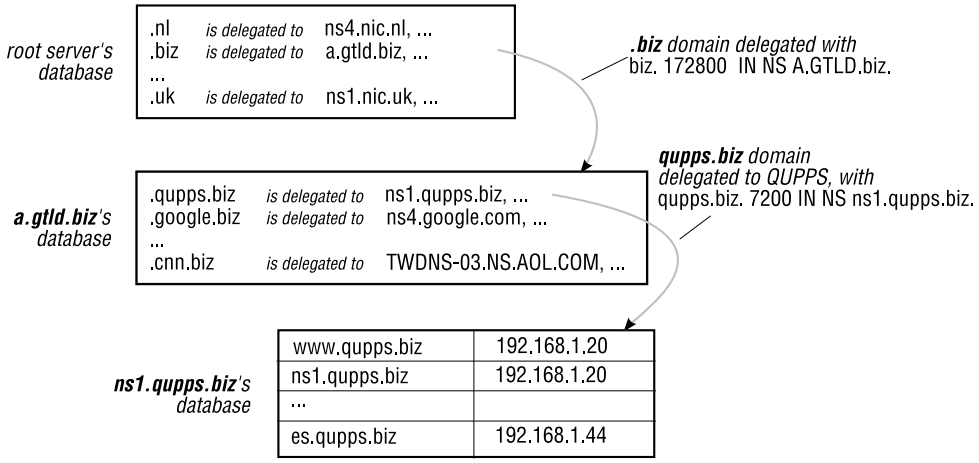

Figure 2.2: How NS resource records are used to delegate sub-domains

If a zone contains an NS record for, say ns9.example.com, but that name server does not exist, or does not answer authoritatively for the zone, you have what is called a *lame delegation*. You have wrongly delegated authority to a name server that isn't (or doesn't believe it is) authoritative for the zone.

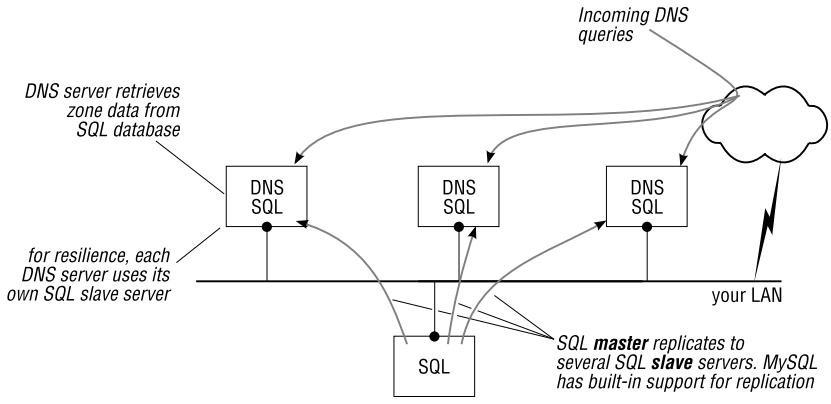

Figure 2.4: Database server and SQL server on one machine

2.5.4 LDAP Directories

Many organizations integrate their configuration data, and security data for authentication and authorization services, into a single, consolidated, data store in the form of an LDAP directory. The LDAP directory is then the single point from which all this data is managed. In such environments it makes sense to store the DNS resource records in the same LDAP directory.

In an LDAP directory, a DNS zone is represented as one or more entries. Each entry contains several attribute types, each representing a specific DNS resource record. (If you are new to LDAP, or want to refresh your memory, we discuss these terms in Appendix A.)

DNS servers that support LDAP

The following name servers can store zone data in LDAP directories: BIND SDB, Bind DLZ, PowerDNS, and ldapdns.

Microsoft Windows DNS Server can be made to store zone data in Active Directory. (Although Active Directory is accessible via LDAP, the format of the entries containing DNS resources is undocumented, which means you cannot manipulate DNS zone data in an Active Directory via LDAP.)

Choice of LDAP directory servers

If you already have a directory server, such as Novell eDirectory, Microsoft Active Directory or similar, you will use that. If you don't yet have a directory server, we recommend OpenLDAP, an Open Source implementation of an LDAP directory server and associated programming libraries and utilities (see `http://www.openldap.org`). We show you in Appendix A how you acquire, install and configure a ready-built package of OpenLDAP.

Replicating LDAP data

To avoid a single point of failure, to spread the performance load, or to service geographically dispersed clients, you can run multiple identical copies of an LDAP directory server (Figure 2.5).

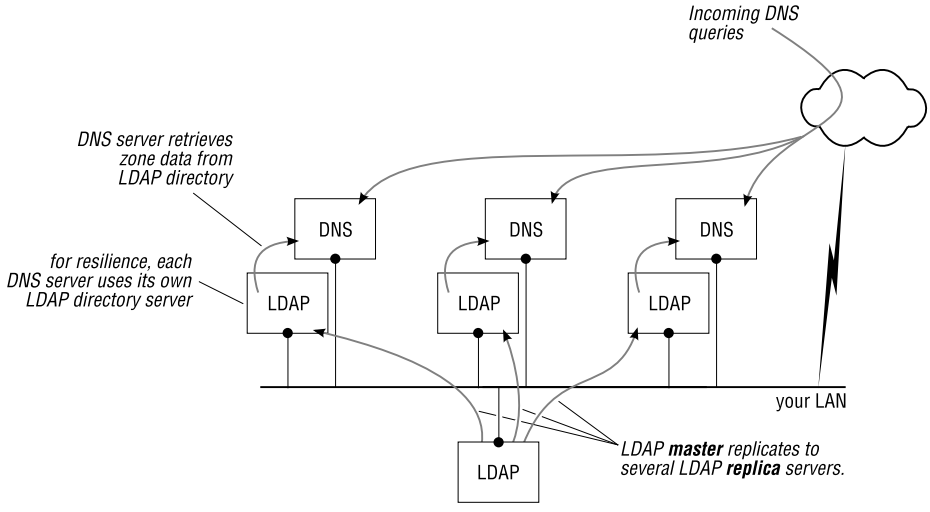

Figure 2.5: Directory server replication ensures resilience

Figure 2.5 shows each DNS server accessing its own directory server, but we recommend you keep the data as close as possible to the DNS server (from a network point of view) and that you deploy both the DNS name server and its LDAP directory server on the same machine, as we showed you above for the SQL databases.

If you're going to use an LDAP directory server as the back-end for a DNS server, replication is essential. You should *never* have more than one name server accessing a single directory server; that would create a single point of failure: if your directory server goes down, all DNS servers querying it are left without a data store. Having more than one DNS server reading entries from a single LDAP directory server is a sure road to disaster.

Manipulating entries in an LDAP directory

The following are editors ("*LDAP browsers*") for creating, modifying or deleting entries in an LDAP directory: task:

- phpLDAPadmin is a Web-based LDAP browser that you can install on a Web server (see `http://phpldapadmin.sourceforge.net/`).
- LDAP Browser/Editor is written in Java. It runs on a variety of platforms (see `http://www-unix.mcs.anl.gov/~gawor/ldap/`).

rr-table
: Specifies the name of the database table containing the resource records for zones. The default value is `rr` and you shouldn't need to modify this.

rr-where
: Analogous to soa-where, this contains an additional `WHERE` clause to append to queries selecting records from the rr-table. You can use this to selectively enable or disable individual resource records.

recursive
: The IP address of a DNS server that accepts recursive DNS queries. If MyDNS receives a query where recursion is desired, and the zone is not local (i.e. MyDNS is not authoritative for it), MyDNS will forward the query to the server at the specified and return the result to the client.

```
recursive = 192.168.1.20
```

If your MyDNS server is Internet-facing, do *not* set this option: you should not make recursive DNS available to the whole Internet.

5.3 Replicating zones

MyDNS can replicate zone data in two ways:

1. Replication by the back-end database system. (This is what PowerDNS calls "native" replication.)

   For two or more peer MyDNS servers serving the same zone data, set up each mydns server with its own SQL back-end. Then use SQL database replication from the master database to the slave database (Figure 5.1).

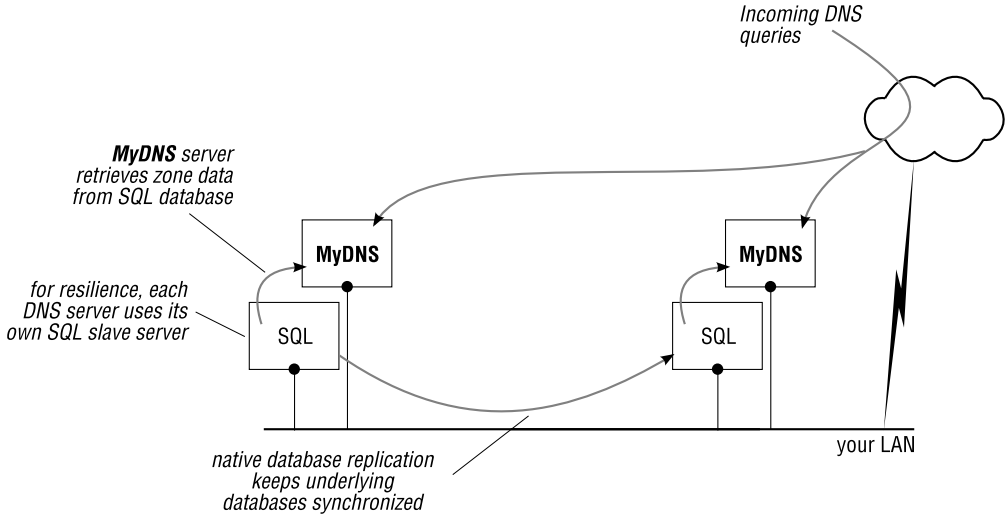

Figure 5.1: MyDNS with "native" database replication

2. AXFR zone transfer: use this where you have a MyDNS master server and a different brand of server as slave(s) (Figure 5.2). MyDNS doesn't support incoming zone transfers, so it can't act as a slave server.

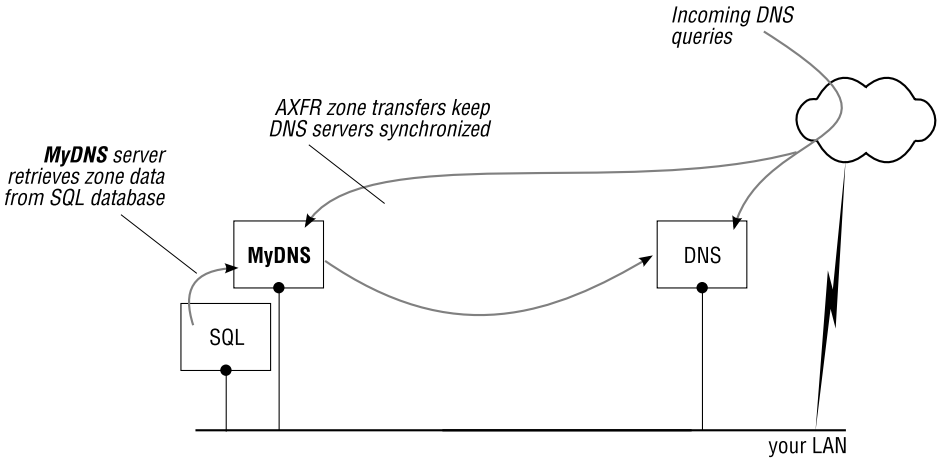

Figure 5.2: MyDNS as a master server

5.3.1 Zone transfers

Zone transfers are allowed only if the allow-axfr variable is "true" in the server configuration. MyDNS supports only outgoing AXFR zone transfers; it does not support incremental transfers (IXFR) or incoming zone transfers. If you enable zone transfers, any client may perform a transfer for any zone in the database. As this is not a good idea, MyDNS provides access controls to limit who may transfer zones.

You can enable IP-based access control in MyDNS by adding a predefined (but optional) column called `xfer` to the `soa` table. The server examines the value of this column when a client requests a zone transfer to determine whether the client is allowed to transfer the zone. If you want to add the column to your database, use mysql (or any other program you normally use to alter a database schema):

```
mysql> ALTER TABLE soa ADD COLUMN xfer CHAR(255) NOT NULL;
```

The `xfer` column contains either an asterisk (`*`), or a comma-separated list of individual IP addresses and/or CIDR *network/netmask* pairs, specifying which addresses are allowed to perform AXFR zone transfers. (The asterisk means "allow any client to transfer zones".) For example, to allow zone transfer requests for zone qupps.biz, from hosts with addresses 127.0.0.1 or 192.168.1.1–192.168.1.255:

```
mysql> UPDATE soa SET xfer='127.0.0.1,192.168.1/24' WHERE origin = 'qupps.biz.';
```

8.4.1 Writing a Driver

When a driver is registered with the SDB subsystem, it publicizes its name, list of callback functions and flags. There are five callback functions which can be provided (Figure 8.2):

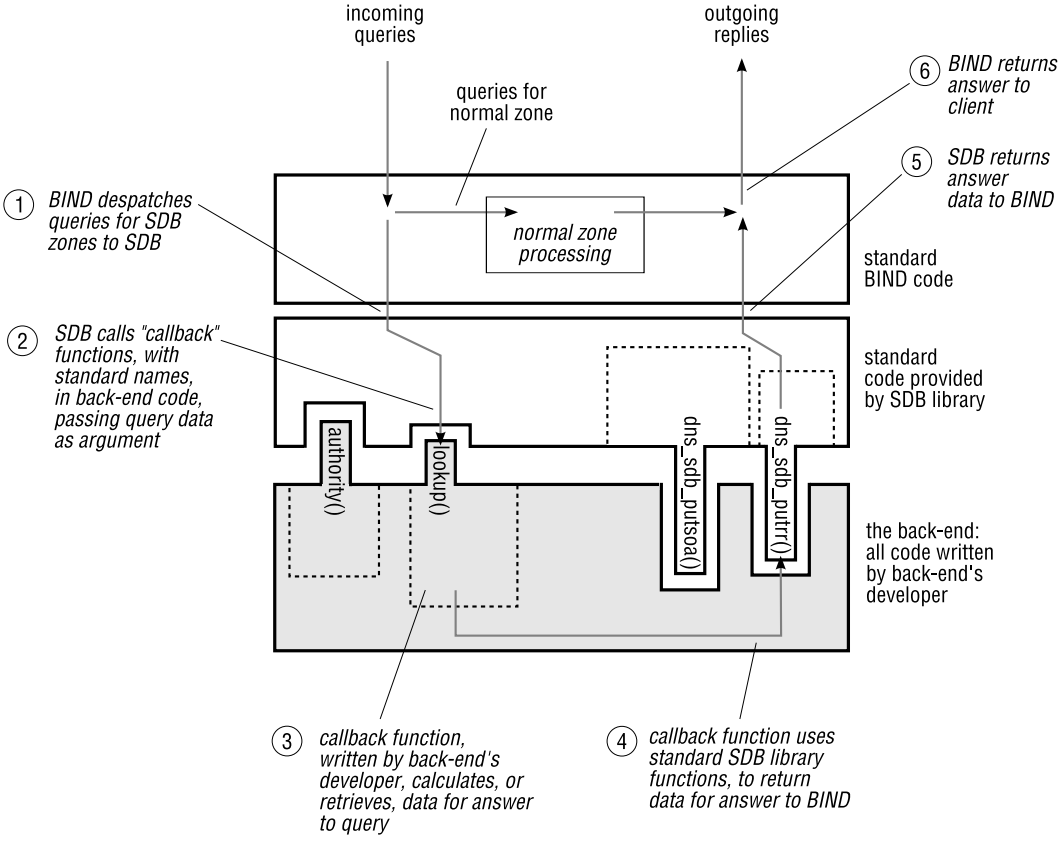

Figure 8.2: The named program, showing the SDB library and driver code

`create`() This is a convenience function that allows the programmer to initialize a zone by connecting to an external database, allocating resources, etc. If the `create`() function is provided, a corresponding `destroy`() can free whatever resources where reserved by `create`().

`lookup`() The `lookup`() function is mandatory. It is invoked by BIND for each query it receives. The function may return whatever resource records are required by the query and, if no `authority`() function is provided, it must return Start of Authority (SOA) and Name Server (NS) records, when queried for the zone apex.

`authority`() The optional `authority`() function must be provided to return SOA and NS records, if the `lookup`() function does not or cannot provide answers to NS

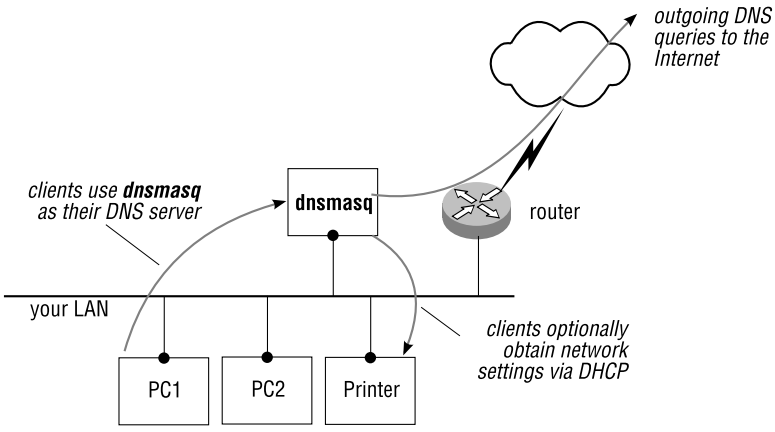

Figure 13.1: Placing dnsmasq on your network

Workstations on the local network set up their resolver to point to the dnsmasq host or get this set up automatically via dnsmasq's built-in DHCP server (in which case you should *disable* any other DHCP server on your network[1]). If you have a single host on your network, you can use dnsmasq as a caching name server for that machine.

If your operating system vendor has not already packaged dnsmasq for you, you will have to build and install it yourself, but that is not difficult (see Notes). dnsmasq is supported on FreeBSD, GNU/Linux, Mac OS X, and Solaris.

13.1 Preliminary explorations

With dnsmasq installed, you can immediately run it to provide seamless DNS to your network, before you explore its more advanced configuration options which we discuss in the next sections. There is no danger at all in setting up a test DNS server on your LAN because your clients will only "see" it when you explicitly configure them to use it.

First add an exotic host name to the `/etc/hosts` file of the machine you will be launching dnsmasq on, to test the DNS server:

```
# echo "192.168.1.17  foozy.local foozy" >> /etc/hosts
```

Now launch dnsmasq in the foreground and make it log queries to the console, so that you can see things as they happen:

```
# dnsmasq --no-daemon --log-queries
dnsmasq: started, version 2.43 cachesize 150
dnsmasq: compile time options: IPv6 GNU-getopt no-ISC-leasefile no-DBus no-I18N
dnsmasq: reading /etc/resolv.conf
dnsmasq: using nameserver 192.168.13.2#53
dnsmasq: read /etc/hosts - 3 addresses
```

[1] Is it just us or do you also get feeling that every device on the market today has a built-in DHCP server? At the time of this writing, we purchased a NAS device, and guess what? Yes.

A name server implemented in Perl can use almost any kind of monitoring software that keeps track of the availability or current performance of hosts. The monitoring software regularly updates a database or file with this information, and then your Perl code in your DNS server uses the information to determine which address (A) to return as answer to the query.

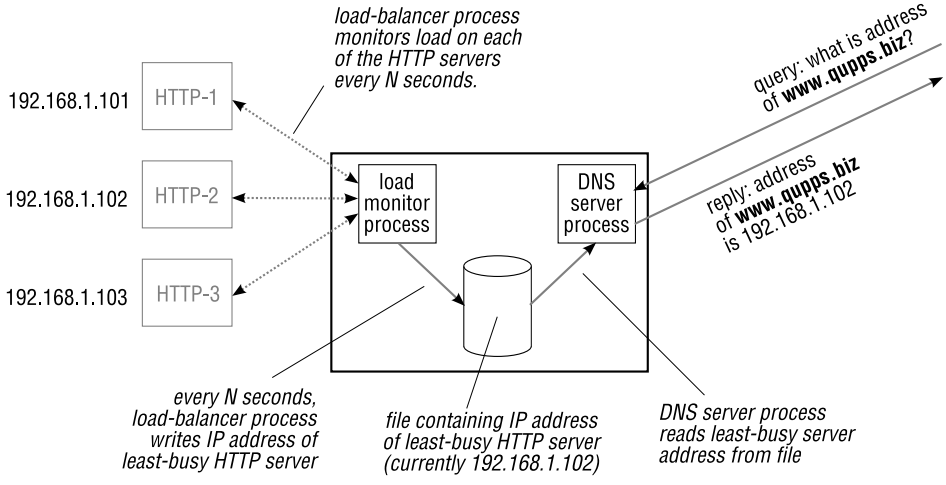

Figure 15.1: An idea for a SNMP-controlled load-balancing DNS server in Perl

15.2.2 Perl tools for creating name servers

There are three different Perl modules you can use to implement a fully dynamic name server:

1. Stanford::DNSserver

   The Perl module Stanford::DNSserver was assembled by Rob Riepel, and it is based on lbnamed by Roland Schemers (see Notes). We describe Stanford::DNSserver in detail in Section 15.3, and use it to develop our own example DNS server.

2. Net::DNS::Nameserver

   The Perl module Net::DNS::Nameserver implements a DNS server class. We show you a simple example in Appendix E. At our request, the module's maintainer, Olaf Kolkman, added a handler to detect DNS NOTIFY requests. We use these in an unusual way to detect if our name servers are sending out notifications (Chapter 24).

3. Net::DNS::Server

   Net::DNS::Server, developed by Luis E. Muñoz, is a set of Perl modules. You use methods to provide answers to specific query types. We show a simple example of how to use it in Appendix E.

Each of these three modules has its own, specific, API. Look at the examples here and in the Appendixes to see which you prefer, or which is closest to your needs.

15.3 Example – Implementing a custom dynamic server using Stanford::DNSserver

Stanford::DNSserver is an extensible name server written in Perl. The Perl code you write determines the answers to the DNS queries it receives (Figure 15.2). The resource records it returns to the DNS client are constructed at runtime by code you supply.

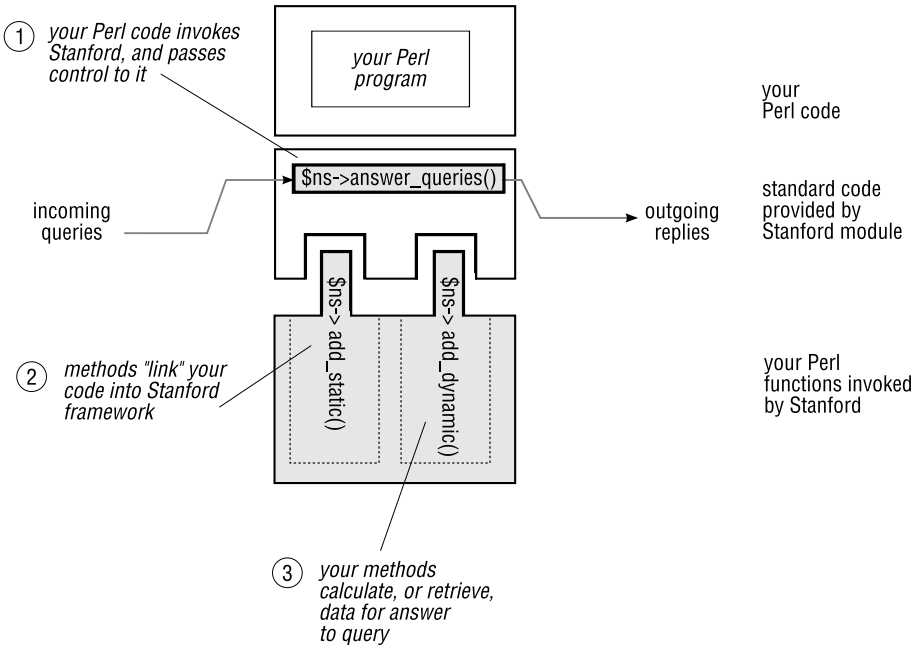

Figure 15.2: Architecture of Stanford::DNSserver

Installing Stanford::DNSserver

After downloading the code, apply the usual Perl incantation:

```
$ wget http://www.stanford.edu/~riepel/lbnamed/Stanford-DNSserver/↩
        Stanford-DNSserver.tar.gz
$ tar xvzf Stanford-DNSserver.tar.gz
$ cd Stanford-DNSserver-1.2.0
$ perl Makefile.PL
$ make
$ make test
$ make install
```

```
alexi.info.qupps.biz.  3600  IN  TXT  "name: Ilexa Snem"
alexi.info.qupps.biz.  3600  IN  TXT  "phone: +34 71 10019345"
```

5. Our own Perl code has all the data it requires now. It passes it back to the main code of the Perl server, which returns the DNS reply to the sysadmin's dig (via the caching server).

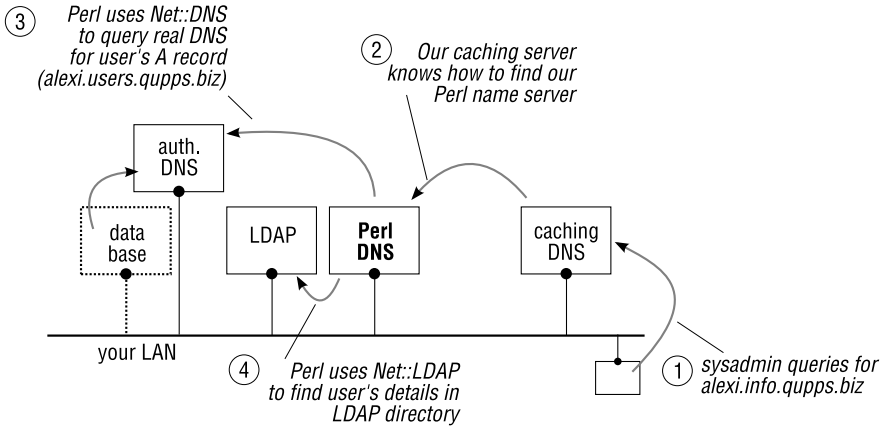

Figure 15.3: Stanford::DNSserver Perl server queries DNS and LDAP

The reason we use two distinct domains is that we used info.qupps.biz to get the authoritative "information" records, which is held on our Perl server (strictly, on the LDAP directory used by the Perl server). Then, we use the separate domain, users.qupps.biz, to get the authoritative A record for the user's machine.
Could we have consolidated the data into a single name server? Yes, but then we would have had to store the users' machine addresses – their A records – in our LDAP directory, instead of using a normal authoritative server as we did above.

The code

The name server we implement is a program is called clidnsd.pl. We go through the source ode explaining the most interesting portions here. The full code is in Appendix E.

1. We are going to use a "modern" DNS Service (SRV) record to find the LDAP directory server:

```
my $ldapsrv     = '_ldap._tcp.qupps.biz';
my $ldapbase    = 'ou=usr,dc=qupps,dc=biz';
```

2. Define defaults for our DNS server:

```
my $myname      = hostname();
my $tld         = 'info.qupps.biz';
my $atld        = 'users.qupps.biz';
my $ttl         = 3600;
```

16.1 Why would I want to implement a DNS blacklist?

Reasons to implement a DNS blacklist on your own systems include:

- You are sick and tired of Spam. Your users already have a method in place with which they report Spam, and you want to use addresses gleaned from that "database" to populate a blacklist.
- You subscribe to (or otherwise use) a DNS blacklist service. Instead of using bandwidth over the Internet to query their DNS servers (which you mitigate with a caching name server), you want to become self-sufficient in case the DNSBL provider's name servers become unreachable.
- You maintain a "white list" of hosts that you *know* would never send spam, and wish to make that list available to friendly sites (or to the public at large) via DNS.

16.2 How to use an existing blacklist in your e-mail server

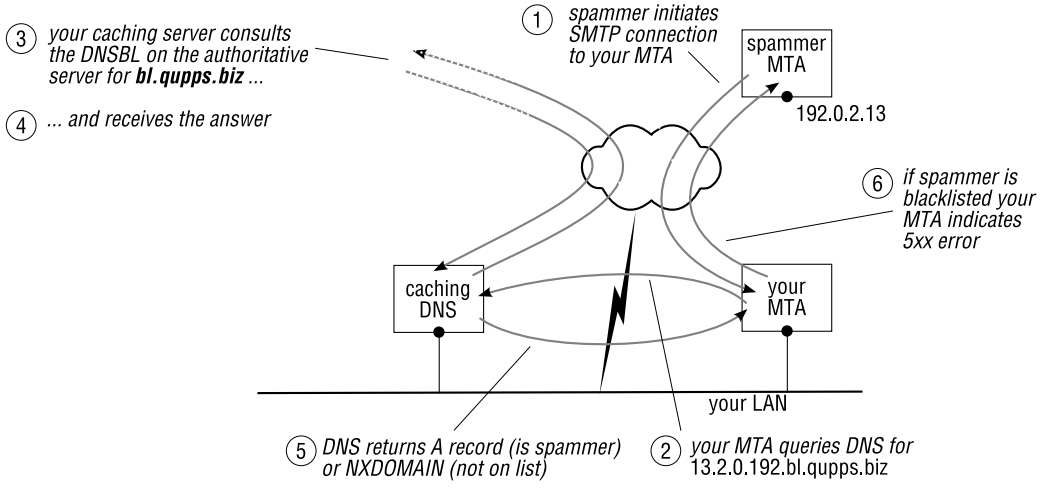

Figure 16.1: DNS blacklists attempt to combat spam

When your mail server receives a connection from a remote mail server (which acts as SMTP client, to your SMTP server) and wants to check that against a DNSBL, the process is more or less as follows: (Figure 16.1):

1. The (remote) sending server establishes a connection to your server. Your server examines the connection to find the IP address (192.0.2.13, say) of the sending server.
2. Your server creates a special "pseudo-domain name" by reversing the IP address and appending the domain name of the blacklist. For example, if the incoming address is

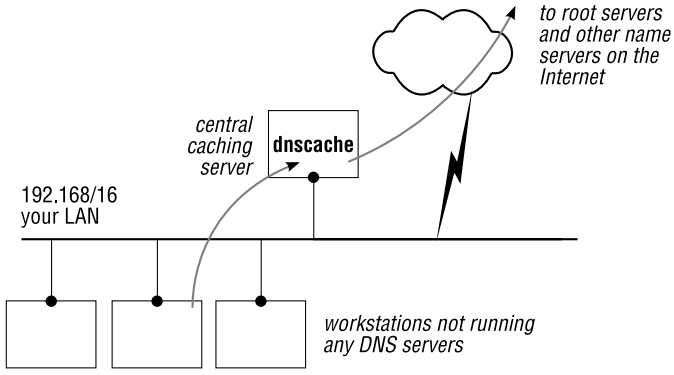

Figure 17.5: Central dnscache

servers contained in `/etc/resolv.conf`, return the answer and forget about it. If the program requesting an answer does not explicitly "remember" (i.e. cache) the answer itself, it is lost and has to be repeated for subsequent identical queries. Users of Microsoft Windows 2000 and above are used to the workstation caching answers; this is handled explicitly by the DNS Client service. In essence, setting up a dnscache on a UNIX or GNU/Linux workstation emulates that feature of Microsoft Windows. On a workstation or computer with Internet connectivity, if you set up dnscache with a default configuration, it contacts the root servers specified in its `$ROOT/servers/@` file and caches results of queries (see Figure 17.6). In this configuration, each workstation's `/etc/resolv.conf` points to the local dnscache IP address at 127.0.0.1.

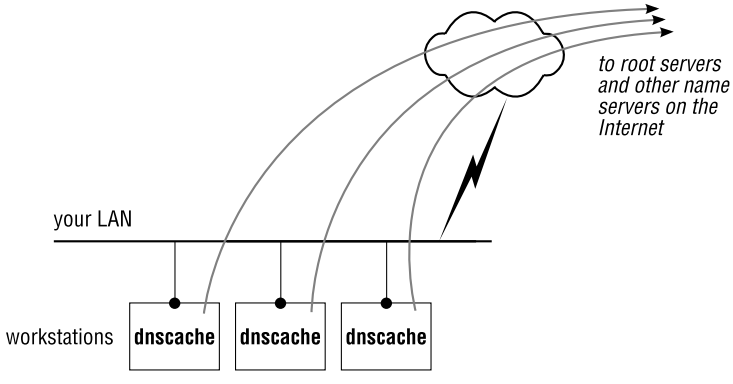

Figure 17.6: dnscache on a workstation

The difference between this and the scenario in Figure 17.5, is that in Figure 17.5 one dnscache services all the machines on the local network, and not just the machine that dnscache is running on.

Workstation as forwarder

The large central cache (Figure 17.5) may also be used as a forwarder for workstations: dnscache runs on each workstation and forwards queries to this central dnscache. You might have a large DNS cache on a central machine in your network, (Figure 17.7, left), or at your ISP (Figure 17.7, right), in which case your workstations can profit from the work it has already done in caching queries. In this case, you set up a local dnscache on a workstation, which forwards its requests to the upstream cache.

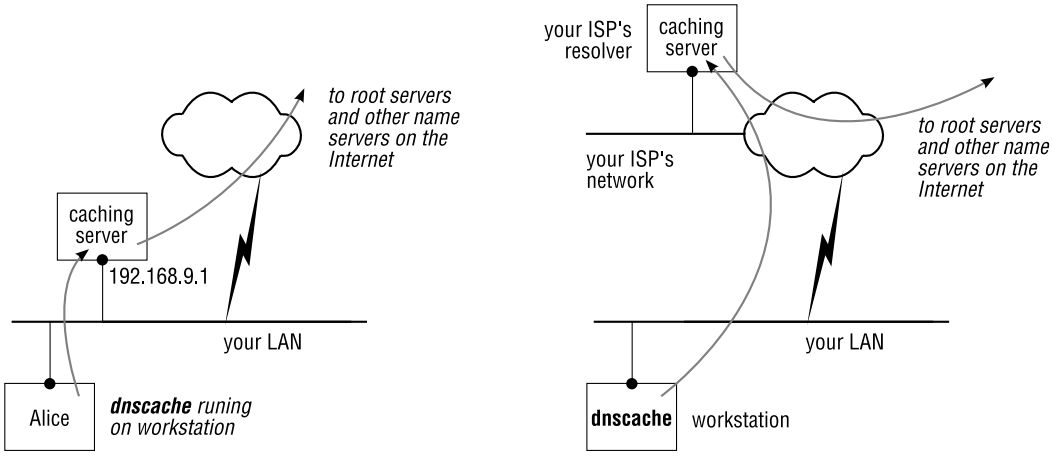

Figure 17.7: dnscache as a forwarder

In Figure 17.7, left, Alice's stub resolver points to her own local dnscache IP address on 127.0.0.1. (Alice's dnscache listens on 127.0.0.1 only, because it doesn't have to handle queries from any other host.) To enable forwarding, edit Alice's `$ROOT/servers/@` file, adding the IP address(es) of the upstream cache(s), and set `$FORWARDONLY`. For example, if the upstream cache has IP address 192.168.9.1, you configure Alice as follows, and then restart dnscache:

```
alice# echo 192.168.9.1 > /usr/local/dnscache/root/servers/@
alice# echo 1 > /usr/local/dnscache/env/FORWARDONLY
```

Central cache with forwarding

This is an extension of the previous scenario. Suppose you have a tinydns (or other authoritative) server that serves a few zones. You add forwarding for those zones to dnscache so that it directs queries for those zones directly to your authoritative name server, without using the root servers (Figure 17.8). You configure your workstations' resolvers to query dnscache; they don't query tinydns directly. Adding a forwarder for a zone is easy enough:

```
# echo 192.168.1.20 > /usr/local/dnscache/root/servers/qupps.biz
```

After restarting dnscache, queries for qupps.biz are directed to 192.168.1.20, whereas all other queries are resolved via the root servers (specified in the file `$ROOT/servers/@`).

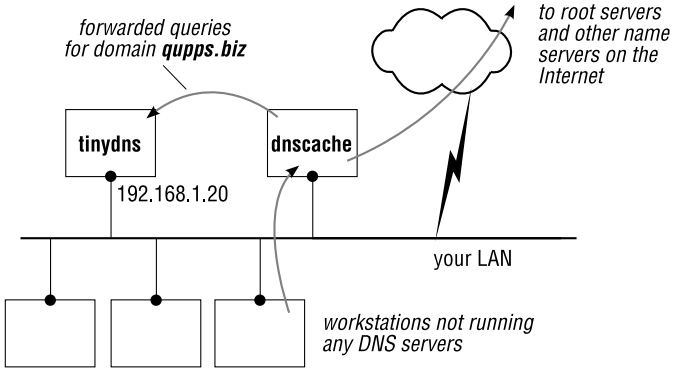

Figure 17.8: Central dnscache with forwarders

Inbound cache

Authoritative DNS content servers that make heavy use of database back-ends and offer no caching of their own can profit from a front-end dnscache as shown in Figure 17.9. Here,

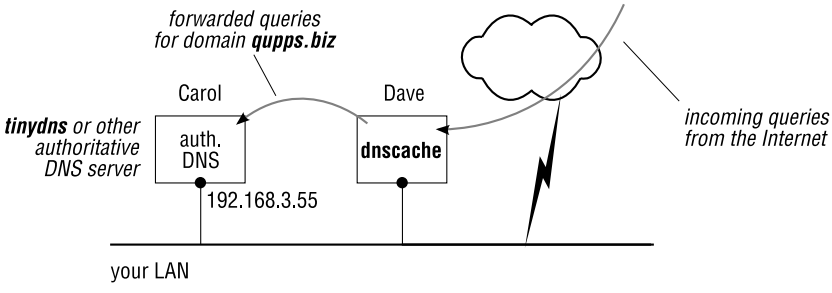

Figure 17.9: Inbound dnscache for content servers

dnscache is accessible on an external IP address (the address to which zones have been delegated to in the worldwide DNS). dnscache forwards all queries to an internal authoritative DNS server. If the authoritative server is on Carol (192.168.3.55), you configure dnscache on Dave with:

```
dave# echo 192.168.3.55 > /usr/local/dnscache/root/servers/@
dave# echo 1 > /usr/local/dnscache/env/FORWARDONLY
```

This configuration provides good caching for your content servers. However, a side effect is that dnscache will never return answers for your zones with the "aa" authority bit set in the replies (because it is the tinydns server that is authoritative, and it's not the one replying). Some DNS registries complain when they check that zones for which your servers are authoritative, aren't answered with the "aa" bit set.

17.4.3 Detailed dnscache configuration options

dnscache uses a handful of environment variables and some files in `$ROOT` (Figure 17.10). When you use the default run script, the environment is primed from files in the `env` directory: as we saw with tinydns, the filename is used as the environment variable name, and the file content becomes the variable's value. If you want to have dnscache start with `$ROOT` set to `/dns` you would:

```
# echo /dns > /usr/local/dnscache/env/ROOT
```

After modifying a variable's value you must restart dnscache for the change to take effect.

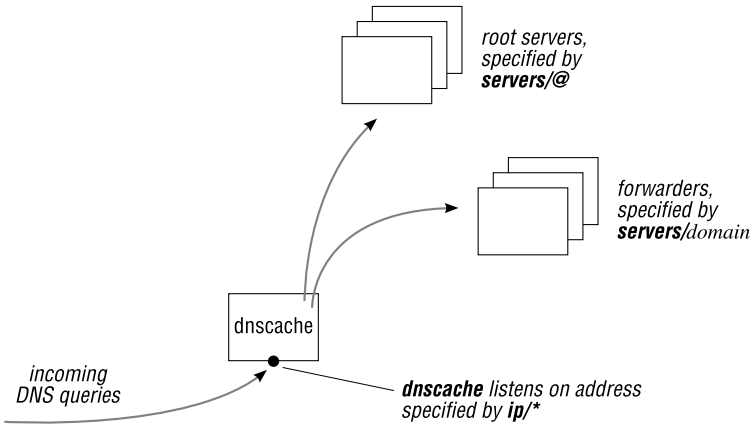

Figure 17.10: Files used by dnscache

Here are the variables and filenames used by dnscache:

$CACHESIZE The size of dnscache's in-memory cache. Default: one million bytes.

$DATALIMIT The run script sets the limit on the maximum size of the process's data segment to `$DATALIMIT`, using the `setrlimit()` system call, effectively limiting the maximum size of the dnscache process. The default is three million bytes.

$FORWARDONLY If this variable is set to any value, dnscache treats the addresses in `$ROOT/servers/@` as a list of forwarders (i.e. addresses of other caching name servers), not root servers. Any domain in the `servers` directory will be contacted directly while all other queries go to the back-end caching name server(s) that you specify in the `$ROOT/servers/@` file.

If you do set this variable, ensure that your `@` file does *not* contain addresses of root servers; the addresses listed must be of name servers willing to handle recursive queries.

The following two methods don't use RFC 2136:

D. If the DNS server doesn't support RFC 2136: the DHCP server writes leases (i.e. IP address allocations) to the file `dhcpd.leases`, and a program can automatically scan the file for modifications and update your DNS from that.

E. The "poor man's" way. We discuss this in Section 19.8 below.

We cover each of these in later sections. But first, if we are going to use method A, B, or C, we must configure the DNS server so it will accept dynamic update requests.

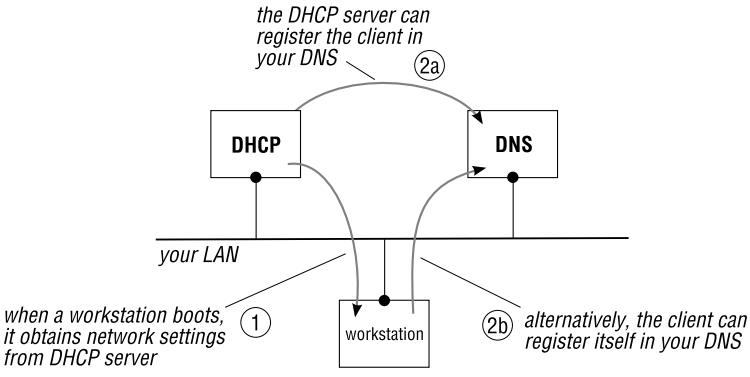

Figure 19.6: DHCP and dynamic DNS updates

To have your DHCP server perform dynamic DNS updates:

Configure your DNS server

Set up your name server to allow incoming RFC 2136 updates:

BIND For BIND, create one or more keys using rndc-keygen, and include an allow-update statement in each of the zones that is to be dynamically updatable:

```
zone "mens.de" IN {
  type master;
  file "mens.zone";
  allow-update { key our; };
};
```

MyDNS For MyDNS, add the update_acl column to the `soa` database table, populate it with the address of the DHCP server, and set allow-update in `mydns.conf`:

```
allow-update = yes
```

We wrote a single executable client program, pmc, that can be compiled on multiple platforms, and distributed it to the workstations. The implementation ensures that the complexity is centralized on the server, not on the clients. This approach has the following benefits:

- Because the program distributed to user's workstations is so simple, it requires no maintenance or updates.
- Authentication and authorization are performed by the server. The server decides if the DNS entry is allowed, and if so, how to add it.
- The back-end DNS servers can be replaced if necessary, without requiring the client program to be altered.

To limit the complexity of the client program, we determine the client's IP address address *on the server*, not on the client. At this point, you must be asking yourself, whether we've gone off our rocker. Hang on.

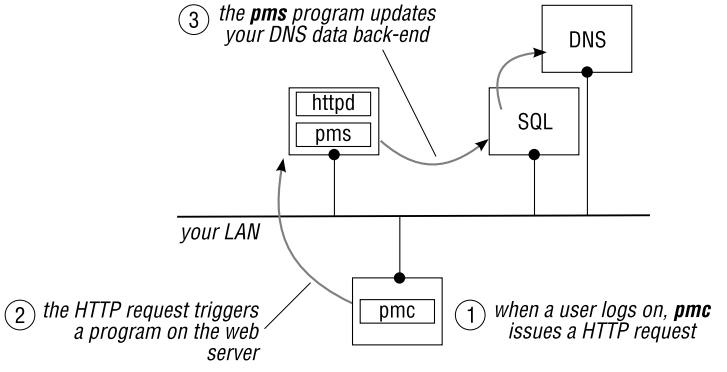

Figure 19.7: poor man's Dynamic DNS

The server program, pms, runs under the control of a Web server. The client contacts the server via HTTP, and in the HTTP request, gives the username of the user currently logged on (Figure 19.7). The server, pms, then determines the client's IP address: it is passed to the HTTP server in the `$REMOTE_ADDR` variable, which is generally made available to a Common Gateway Interface (CGI) program or to a PHP script. pms then creates a DNS resource record or two (an Address (A) and a PTR) for the client.

The service provided by our "poor man's dynamic DNS" is comparable to that offered by dyndns.org (see Notes) and similar providers. We developed this idea for a closed corporate environment. Consider using it if Dynamic DNS Updates a la RFC 2136 aren't possible or are too cumbersome.

If you haven't forbidden your name server answering to version.bind queries, you can check with a simple DNS query:

```
$ dig +short @192.168.1.20 version.bind ch txt
"9.2.4"

$ dig +short @192.168.1.164 version.bind ch txt
"NSD 3.0.7"
```

- Here's an example of a cheap form of load balancing that we implemented for a customer, which makes good use of the monitoring system. The site has two Web servers; one is a hot-standby for the other. By default, the DNS entry for www.*site* points to the primary Web server. We implemented a Nagios plug-in that determines whether the primary is responding correctly (Figure 24.1). As soon as the primary doesn't respond, Nagios updates the back-end database of the DNS server, and changes the www.*site* Address (A) record to point to the secondary Web server. The TTL of the DNS record is set to a very low value, so that the switch from primary to secondary happens quickly.

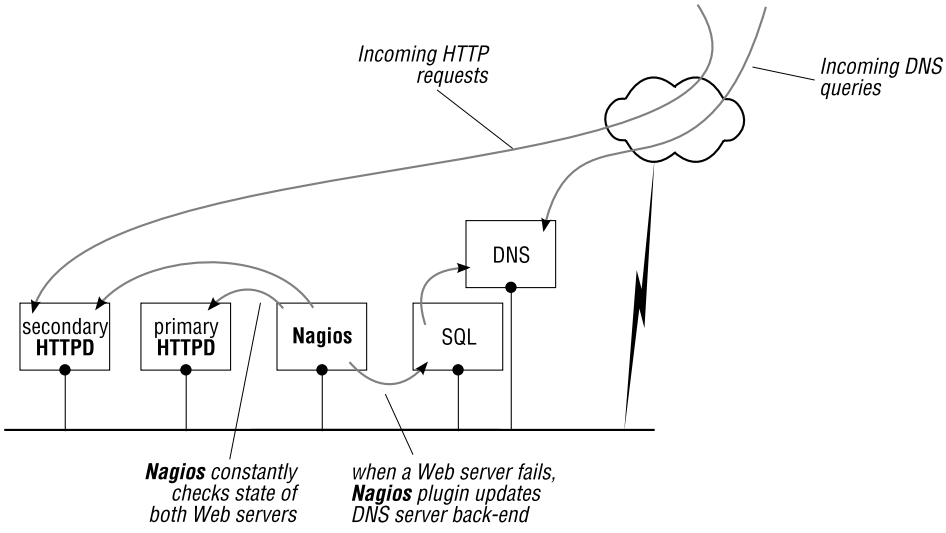

Figure 24.1: Nagios monitors Web servers and updates DNS

If your DNS servers make use of an SQL database or an LDAP directory server, you should also monitor these components:

- Connectivity between the DNS server and the external database or LDAP server, if these components are not both on the same physical machine (c.f. plug-ins, item 4, page 576)
- Replication between back-ends. Let's say you have two name servers, each with an LDAP back end. If the back-ends are not replicating correctly, a query to one DNS

server could obtain a different answer than from the other DNS server, but only while the back-ends are out of sync. These intermittent errors are hard to detect.

Here's an easy way to check your SQL or LDAP replication. Periodically, update a counter in the master back end; you can use cron or your monitoring system to do this regularly. Then use another tool to check the value of this counter in each instance of your back-end; if the counter is the same on all back-end instances, replication is working correctly Figure 24.2).

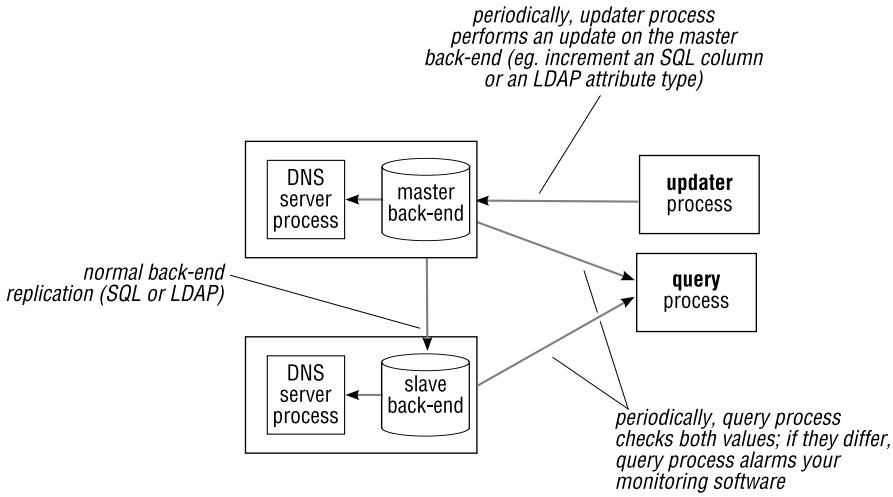

Figure 24.2: Using a "probe" to check back-end replication

That concludes our discussion on monitoring of your DNS service. In the following section we look at how you can gather statistics about your DNS operation.

24.3 Gathering statistics about your DNS operation

In the chapters about the individual brands of name servers, we discussed how (if at all) they produce their own statistics. Some don't produce the detailed statistics you might want, in which case you can use one or more of the tools below. These tools are name server agnostic: they sniff DNS packets on the wire, so they work with any name server implementation.

Even though some servers can produce their own statistics, if you are using more than one brand, or think you might change later, you might want to deploy a solution that caters for any brand of name server, as described in the next section.

24.3.1 DNS Statistics Collector: dsc

DNS Statistics Collector, dsc, is a program by The Measurement Factory, written by Duane Wessels and Ken Keys (see `http://dns.measurement-factory.com/tools/dsc/`). It is designed to collect and aggregate statistics from busy authoritative servers, such as those used by TLD and root server operators, but you can use it to collect statistics for any DNS servers you use. The program consists of two major components (Figure 24.3):

1. The collector process sniffs DNS messages received and sent on a network interface. You typically run it either on the machine on which your DNS server is located or on a system connected to a switch port configured with port mirroring, in which case you monitor the port that mirrors your DNS traffic. A configuration file specifies which datasets the collector should collect. The collector dumps the datasets to XML files every 60 seconds.

   Each machine on which DNS is monitored is called a "node", and nodes that have something in common (e.g. location) are called a "system" or "system cluster". For example, you could group the DNS servers for Spain (i.e. the hosts ns1.es.qupps.biz and ns2.es.qupps.biz) into a system you call "Spain".

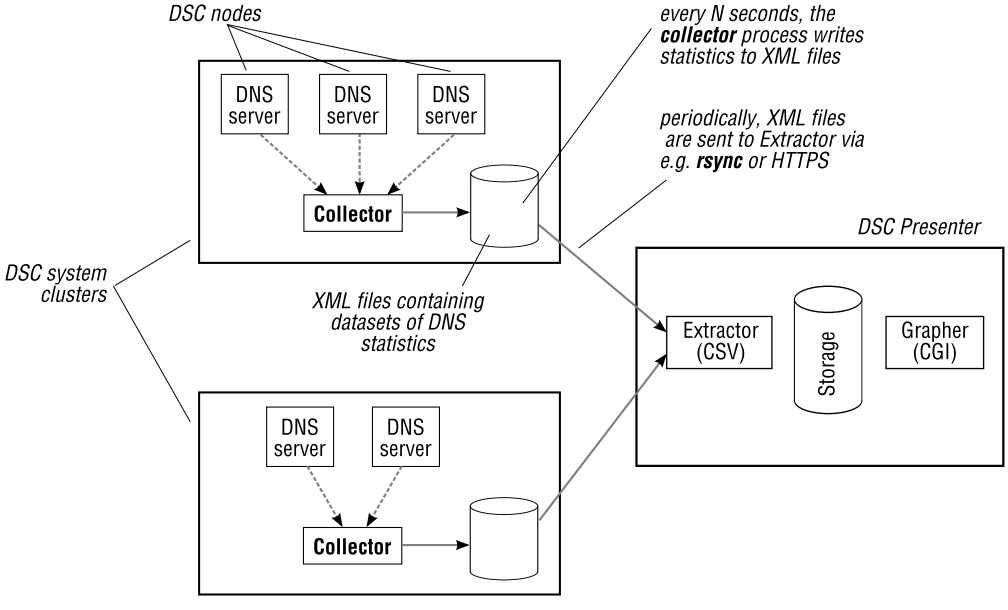

Figure 24.3: Architecture of the DNS Statistics Collector

2. The presenter component receives the XML files from collectors. It uses an extractor process to parse and convert them to a different text-based format. dsc provides scripts for "uploading" the XML files from your collectors to the presenter, but you are free to use any means at your disposal.

- You may wish to grant permissions to a group of individuals (or an application) based on the directory structure. For example you will want to allow only the HR department to access employee payroll data, but allow anybody to see the DNS data in your directory.
- You can combine replication with individual branches of the tree. For example, you might replicate the branch containing data for a department to a dedicated distant server for that department.

LDAP data is stored hierarchically in a tree starting at a root (usual local to your LDAP server), and branching down into individual entries, just like the DNS or a computer's file system (Figure A.1). The top level entry, just below the root of the hierarchy, typically represents your organization. Under that in the hierarchy might be entries for smaller groupings, such as departments, etc. The hierarchy might end with entries for individual people or resources.

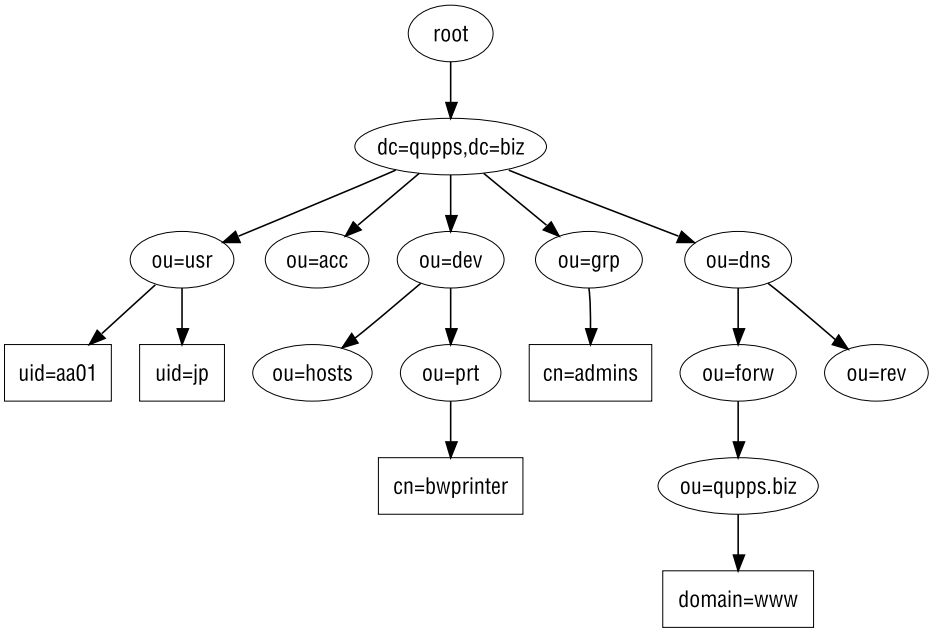

Figure A.1: LDAP tree

The tree structure is called the *Directory Information Tree* (*DIT*). Each entry in a directory is an *object*. Objects are of two types: containers and leaves. A *leaf* is an object at the end of a branch. A *container* is like a folder: it contains other containers or leaves. (In Figure A.1 we show containers as ovals and leaves as rectangles). Note that containers can be empty, and leaves can, at any time, be converted to containers by creating subordinate leaves. In other words, there is nothing really special about a container: you can create a container as a "person" object and later decide to change it to store "furniture" objects in it.

- Suppose further you want to find either all entries with a surname of "Doe" and a given name of "Jane", or those that have a room number A1. You join the two sub-filters with a boolean OR:

```
(|(&(sn=doe)(givenname=jane))(roomnumber=A1))
```

- You want to search your directory for all names starting with the string "Smith" to find people called "Smith", as well as "Smithsonian". The asterisk allows you to specify a substring in a search:

```
(sn=Smith*)
```

A.3.6 Search scopes

When searching an LDAP directory, you specify how "deeply" within the DIT the directory server should search for you. The depth is called the *search scope*. The search scope defines the set of entries at the search base that the directory server should consider for a search operation (Figure A.4). (Remember that the search base specifies the DN in the hierarchy where your search starts.) There are three search scope values:

base — The search operation should be performed only against the entry specified as the search base DN. No subordinate entries will be considered.

one — The search operation is performed against entries that are immediate subordinates (i.e. children) of the entry specified as the search base DN. The base entry itself is not included, nor are any entries below the immediate children.

subtree — The directory searches the entry specified as the search base DN and all of its subordinates to any depth.

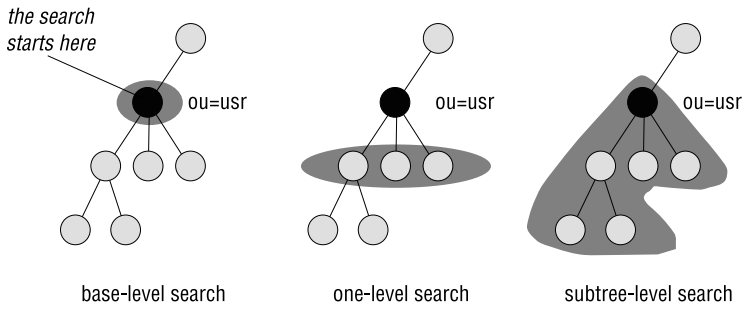

Figure A.4: Search scopes

The areas your in directory that are searched through, for the respective scopes, when you start searching at `ou=usr,dc=qupps,dc=biz` are shown in Figure A.4.

3. Reload the master (stealth) name server so that it performs an AXFR zone transfer.

The program in Section 6.9.3, page 154, automatically enumerates the zones in your back-end database to create a `zones.incl` file, but how do you invoke the program? There are several options:

- Try to remember to launch it manually. This is error-prone – if you forget, your master name server won't serve the zone.
- Periodically run the program via cron. This is wasteful of resources if you have many zones. You need a mechanism that creates the list of zones only when you add or remove a zone from the database.

We love automation (because we're lazy) so we use a database trigger, a User Defined Function, a pinch of make and a squeeze of cron (Figure F.1) to automate the whole process.

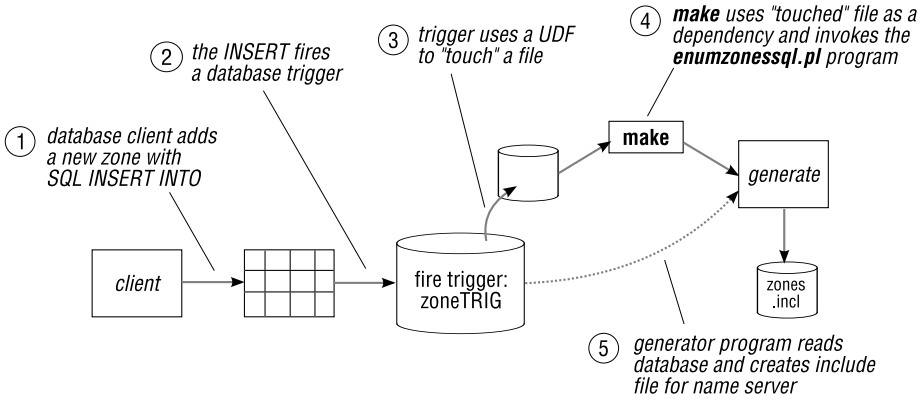

Figure F.1: Trigger an update to a file via MySQL with a UDF

1. With your chosen database client (e.g. MySQL command-line client, Web interface, Perl utility, etc.) you create a zone in your database by performing an SQL `INSERT` into the appropriate table.

2. The `INSERT` on the table fires a database trigger:

```
DELIMITER $$

CREATE TRIGGER zoneTRIG AFTER INSERT ON domains
FOR EACH ROW
BEGIN
      SET @error = newzone(NEW.name);
END $$
DELIMITER ;
```

This example is for the table used by PowerDNS; adapt the table and column name for your brand of name server accordingly.

Appendix

H Scripting PowerDNS Recursor with the Lua programming language

Two weeks before this book was delivered to the printers, a new version of PowerDNS Recursor was released. This version includes scripting capabilities, using the Lua language. You can use this to create or modify on the fly answers to queries received by PowerDNS Recursor. We give you a (very) short introduction to Lua, an overview of how PowerDNS Recursor uses it, and explain how you use Lua functions to manipulate DNS queries.

H.1 A (very) short overview of Lua

Lua is a powerful, fast, light-weight, embeddable scripting language, designed by Waldemar Celes, Roberto Ierusalimschy, and Luiz Henrique de Figueiredo in 1993.

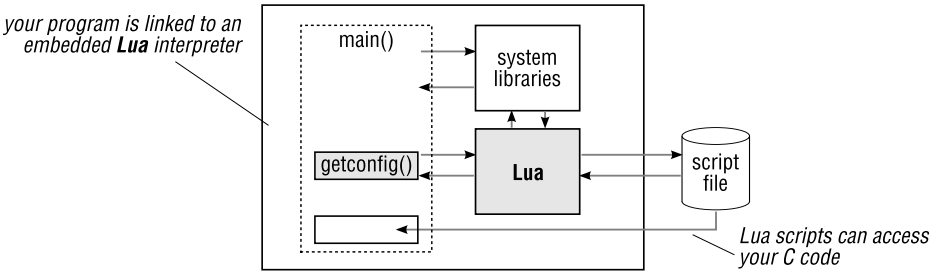

Figure H.1: You can embed Lua in your applications

It has much of the functionality found in a modern scripting language: scope, control structures, iterators, and standard libraries for processing strings, performing mathematical operations and interacting with a machine's environment. Tables are the the only data-structuring mechanism in Lua. Tables are dynamic: they grow when data is added to them (by assigning a value to a hitherto non-existent field) and shrink when data is removed from them (by assigning `nil` to a field). You can use a table as an array (indexed from 1), an associative array (sometimes called a hash or a dictionary), a record, etc. The contents of tables in Lua can include any combination of types. Tables' keys can be of any Lua value, including a function or another table. (For a complete description of the Lua language, see `http://www.lua.org/manual/5.1/`, and the book *Programming in Lua* at `http://www.lua.org/pil/`. There is also a very lively and helpful mailing list at `http://www.lua.org/lua-l.html`.)

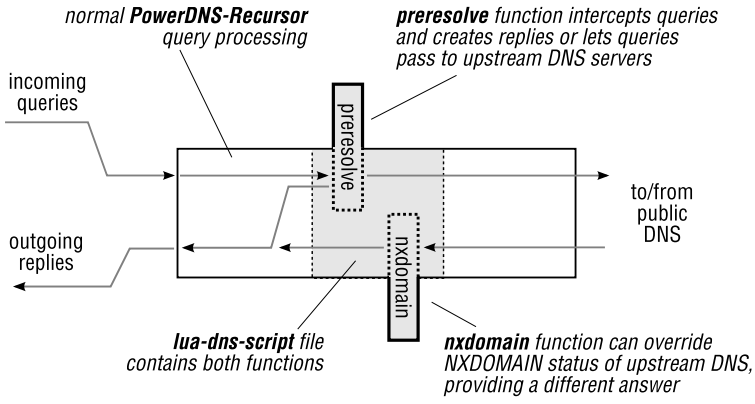

Figure H.2: How PowerDNS Recursor uses the Lua script functions during query processing

Possible scenarios for using Lua functions in your PowerDNS Recursor server include:

- Load balancing within your own organization. You can implement a preresolve function that answers a query for a specific hostname from a list of addresses managed by your Lua script. (Note that this load-balancing is used by your internal hosts only, because it is implemented on your caching server, and not on your Internet-facing authoritative servers.)
- Catch typos. You know that your general manager always mistypes www.qupps.biz (he uses an "s" instead of the "z"), so you add an nxdomain script that returns the correct address anyway. (It avoids you having to answer his questions about the reliability of your Web servers...)
- Advertisement blocking. You want to "protect" your users from certain Web sites, so you configure a preresolve function that catches queries for those and returns a harmless address, say 127.0.0.1 instead. Fredrik Danerklint has implemented such a system, using an externally acquired `hosts` file[1], together with an optional patch to PowerDNS Recursor. The patch adds a call-lua-function that dumps statistics of the function's use and lets you load new hosts (see `http://www.fredan.org/os/`). (If you want to block such domains for Web browsers only, you can use squid instead.)

H.2.1 Configure PowerDNS Recursor to use Lua scripts

Before attempting to get scripts running, we recommend you configure PowerDNS Recursor normally and get that working (Section 17.3), because the additional level of complexity introduced by scripting can make errors difficult to find. Then:

1. Create your Lua script, containing a preresolve and/or an nxdomain function. You can name the file however you want to. (We've called it `qupps.lua`.)

[1] Danerklint doesn't maintain the `hosts` file himself; note the terms of use (which are included in the file itself).