The following paper was originally presented at the
Ninth System Administration Conference (LISA '95)
Monterey, California, September 18-22, 1995

lbnamed: A Load Balancing Name Server in Perl

Roland J. Schemers, III
SunSoft, Inc.

lbnamed: A Load Balancing Name Server in Perl

Roland J. Schemers, III – SunSoft, Inc.

ABSTRACT

Given a cluster of workstations, users have always wanted a way to login to the least-loaded workstation. This paper discusses an attempt to solve that problem using a load balancing name server. This name server also has the ability to serve other dynamic information as well, such as /etc/passwd information (a la Hesiod [2]). The prototype was written in Perl 4 [1], and recently converted to Perl 5. This paper describes the Perl 4 version first and then describes some of the interesting features in the Perl 5 version. This paper assumes the reader has a basic understanding of Perl, DNS, and BIND [3].

The Problem

When I joined the Distributed Computing Operations (DCO) group at Stanford, some of the machines in the public UNIX workstation clusters were overloaded but others were idle. People logging in remotely picked the same machine all the time, or they always picked the same architecture. For example, they would always login to a system running SunOS. If their favorite host was down, they would call the operations staff and complain that the whole system was down.

Users constantly asked for a way to login to the "best" workstation. The best answer consultants could give them was to login to some workstation, then run a program called "sweetload" which would return a sorted list of the loads on all the workstations. The user would then have to pick one workstation from the list, and login to it, possibly logging out of the workstation they ran sweetload on. This was not an ideal solution.

It was at that point I decided to start working on a "real" load balancing name server. I was interested in creating a DNS name server that could receive a request and dynamically create the response.

Stanford has a wide range of machines in the public workstation cluster(s). Figure 1 shows the diversity of machines. A user can login to any one of the workstations and see essentially the same environment: same home directory, mail, etc. This type of environment is well-suited for a load balancing name server.

19 Sparc 2s
37 Sparc 20s
31 Alpha 3000/300s
13 DECstation 5000/240s
10 RS/6000s
15 SGI Indigos

Figure 1: Public clusters run by DCO

Unlike the MIT Athena environment where the typical public workstation only allows console logins, the public workstations at Stanford allow remote logins. At any one time there could be some poor user sitting at the console of a Sparc 2 with limited memory and swap space trying to read their mail while 20 other people were logged in compiling their CS project. If we had the resources we could have disabled remoted logins on all the public workstations and set up some specially configured workstations for remote logins. This is still being investigated, but for historical reasons remote logins are still allowed on public workstations. During the semester we typically saw over a thousand simultaneous unique logins across one hundred workstations.

Other Solutions

At the time I started working on lbnamed there were a number of existing solutions. "Shuffle Addresses" (SA) are one solution implemented by Bryan Beecher. One downside with SA records is they require changes to the DNS specification since they add a new resource record of type T_SA. Another solution is Marshall Rose's "Round Robin" code which is included with current versions of BIND [4]. The problem with "Shuffle Addresses" and the "Round Robin" approaches are they don't factor in load when handing out addresses. For example, the "Round Robin" code just cycles through the A records in a round-robin fashion. The "Round Robin" solution does have its benefits, as it provides some balancing with little to no expense.

At the time this paper was being written, RFC 1794 [5] was also published and describes a load balancing method using a special zone transfer agent that can obtain its information from external sources. The new zone then gets loaded by the name server. One problem with this method is in between zone transfers the weighted information is essentially static, or possibly handed out round-robin. This method also doesn't allow for exotic virtual/dynamic domains where the response is created dynamically based on the name being queried. It does elegantly solve a class of load balancing problems though.

There have also been other load balancing name servers hacked up over the years, but most of them were like my initial lbnamed prototype and not extensible.

Requirements for Initial Implementation

The project had these initial requirements:

a. No changes to the DNS protocol, should be compatible with existing DNS implementations.
b. In between updates of load balancing information from the external source the cached load information should change so the server doesn't end up returning the same information over and over.
c. Must respond fast. Polling for load information will be done by a separate process and loaded back into the load balancing name server.
d. Should be easy to configure and maintain.
e. A host can belong to multiple groups or clusters.
f. Should not preclude having virtual/dynamic domains. The response should be dynamically generated based on the name being queried.
g. Redundancy is handled by multiple, independent servers.
h. The initial implementation is not a general purpose name server. Resolver clients should not be pointed at it and it should not be used in lieu of a real name server like BIND. Remember, don't try this at home.

Solution

lbnamed: A Load Balancing Name Server in Perl

Lbnamed is a load balancing name server written in Perl. It was meant to be a prototype that would get re-written in C and/or integrated with a special version of BIND. It has worked well enough (and I've been too busy with other things) that I've left it in Perl. Lbnamed allows you to create dynamic groups of hosts that have one name in a DNS domain. A host may be in multiple groups at the same time. For example, when someone types:

```
telnet elaine.best.stanford.edu
```

they get connected to one of 57 different SPARCstations named elaine1-elaine57. Since the Elaines contain both Sparc 2 and Sparc 20 class machines I also wanted a way for people to be able login to the "best" Sparc 2 or Sparc 20, partly for fear that people who knew the difference wouldn't want to use the "elaine.best" alias because chances are one of the Sparc 2s would have the lowest load. Therefore, someone can also type:

```
telnet sparc20.best.stanford.edu
```

or even:

```
telnet sparc2.best.stanford.edu
```

And get connected to the "best" Sparc 20 or Sparc 2.

The Server(s)

The server side consists of two Perl programs, lbnamed and poller. These programs run in parallel and communicate using signals and configuration files.

Poller

The poller daemon contacts the client daemon running on the hosts being polled. It reads a configuration file that tells the poller which hosts to poll. The poller periodically sends out requests and receives the responses asynchronously. After it has received all the responses it dumps the information into a configuration file and sends a signal to lbnamed which then reloads the configuration file. If the poller does not receive a response from one of the hosts being polled it removes it from the configuration file it feeds to lbnamed.

The poller program is also the program that calculates the weight of each system. This logic was placed in the poller program so the weight formula could easily be changed without having to modify all the poller client programs.

The formula used to determine the weight of a host is:

```
$WT_PER_USER = 100;
$USER_PER_LOAD_UNIT = 3;

$fudge = ($tot_user - $uniq_user) *
          $WT_PER_USER/5)
$weight = $uniq_user * $WT_PER_USER
        + ($USER_PER_LOAD_UNIT* $load)
        + $fudge;
```

Where the variables are:

$tot_user	total number of users logged in.
$uniq_user	unique number of users logged in.
$load	the load average over the last minute multiplied by 100.
$WT_PER_USER	the pseudo weight for each user.
$USER_PER_LOAD_UNIT	the number to multiple the load by.
$fudge	fudge factor for users logged in more then once.

The formula tries to favor hosts with the least amount of unique logins, and lower load averages. It has worked well, but could be improved.

A situation still exists when a host responds to poller requests, and has a low load, but no one can login because of a problem (such as lack of swap). A future version may be smarter and watch for trends where a host is constantly handed out but the weight of that host never changes.

The poller daemon was inspired by a previous program I wrote called fping. See the appendix for a brief description of fping.

Lbnamed

The lbnamed script reads the configuration file generated by the poller and loads it into a number of different data structures. Each group of machines is stored in an array, while the weights of all the hosts are stored in one hash table. When a request for a particular group comes in, the array for that group is sorted based on the weight of each host in that group. The host with lowest weight is then returned as the best host, and its weight is increased by adding two times the constant $WT_PER_USER to it. By increasing the weight we ensure the same host won't be returned over and over.

The best way to understand how the data is stored internally is an example. Consider the configuration file created by the poller shown in Figure 2 where the format of the file is:

```
weight host ipaddress group1 [...]
```

Upon reading the configuration file, lbnamed will create the arrays shown in Figure 3. The @group_ arrays are created using "eval":

```
eval "push(@group_$group,ost);";
```

The groups array contains all the dynamic groups and the number of members in each group. This array serves two purposes. It is used to determine if a particular group exists and to reset the current groups before the configuration file is reloaded:

```
foreach $group (%groups) {
     eval "@group_$group=();";
}
%groups=();
```

The weight array contains the weight of each host and is used to assist in sorting a particular group when a query is made. To find the host with the lowest weight, the eval function is used:

```
$the_host =
    eval "&get_best(*group_$qname);";
```

The get_best function just sorts the array passed to it using the "by_weight function:

```
sub by_weight {
    $weight{$a} <=> $weight{$b};
}

sub get_best {
   local(*group)=@_;
   local($best);
   @group = sort by_weight @group;
   $best = @group[0];
   $weight{$best} += $WT_PER_USER * 2;
   return $best;
}
```

Also note the weight of the host returned is updated so the weight does not remain static in between polls. Another option would have been to sort each array once after the configuration file has been loaded and to hand out names in a round-robin fashion until the next poll. The current method will degrade to round-robin in the case where all the hosts are equally loaded, but will tend to favor the least loaded systems in the normal case. There is room for improvement in this algorithm.

To other name servers, lbnamed looks like a standard DNS name server, with the exception that it doesn't answer recursive queries. It only handles requests for the dynamic groups it maintains. lbnamed gets a normal DNS request and, based on the name in the request, it calculates the host to return. lbnamed then constructs a standard DNS response and sends it back to client that requested it. The time to live (TTL) value in the response is set to 0. This prevents the response from being cached by other name servers.

```
1364 elaine1 36.215.0.117 elaine sparc2 sparc sunos
1264 elaine2 36.215.0.118 elaine sparc2 sparc sunos
1602 elaine40 36.218.0.88 elaine sparc20 sparc sunos
1827 elaine41 36.218.0.89 elaine sparc20 sparc sunos
```

Figure 2: Poller configuration file

```
@group_sparc20 = ( "elaine40","elaine41");
@group_sparc2  = ( "elaine1", "elaine2");
@group_elaine = ( "elaine1", "elaine2", "elaine40","elaine41");
@group_sparc = ( "elaine1", "elaine2", "elaine40","elaine41");
@group_sunos = ( "elaine1", "elaine2", "elaine40","elaine41");

%groups = ('sparc20',2,'sparc2',2,'elaine',4,'sparc',4,'sunos',4);

%weight = ('elaine1',1364,'elaine2',1264,'elaine40',1602,'elaine41',1827);
```

Figure 3: Arrays created from configuration file

For example, Figure 4 shows the use of "dig" (which is distributed with the latest version of BIND[4]) to see data returned from a query to the load balancing name server. Figure 5 shows a second query for the same domain with a different returned value.

There are a few things to note in the data returned. First, a dynamic CNAME is returned, not a dynamic A record. By returning a dynamic CNAME we can leverage off other data associated with the real domain name (such as an MX record). In addition, by returning a CNAME the resolver client doesn't end up with an A record that doesn't have a corresponding PTR record.

Secondly, note that we have returned the address of the host to which the CNAME points. This should save an extra lookup by the resolver client. If we just returned the CNAME record, the client would then have to lookup the A record for that name as well. We already have the IP address because the poller needs it to poll the host.

Load Balancing Client Daemon

Hosts that are going to be polled by the poller need to run a special daemon, the load balancing client daemon (lbcd). lbcd responds to poller requests (over UDP) using a simple protocol. The protocol format is described in the appendix. I wrote yet-another-remote-statistics daemon because initially I had grand plans for having it do a number of different system management tasks. Later, I used Sysctl [6] for those tasks.

lbcd is written in C, although it probably could have been written in Perl with a few helper programs written in C to read information from the kernel. The important thing to remember is the client and poller can easily be replaced with something else, as long as the poller program creates the lbnamed configuration file in the correct format.

Configuring the load balancing name servers

For the load balancing name server to answer requests it must be delegated a virtual domain to serve. This is normally done in the parent domain by adding NS records. In the stanford.edu domain the load balancing name server uses the best.stanford.edu domain, so in the DNS configuration file for the stanford.edu domain there are two NS records:

```
best   IN   NS dsodb.stanford.edu.
best   IN   NS sunlight.stanford.edu.
```

These two NS records delegate the best.stanford.edu domain to the lbnamed's running on dsodb and sunlight. Now when the primary servers for the stanford.edu domain get a request like elaine.best.stanford.edu they know to forward it to the lbnamed's. Note that the two lbnamed's don't communicate with each other; they both operate independently for simplicity and redundancy.

```
# dig elaine.best.stanford.edu

; <<>> DiG 2.1 <<>> elaine.best.stanford.edu
;; res options: init recurs defnam dnsrch
;; got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6
;; flags: qr aa rd ra; Ques: 1, Ans: 2, Auth: 0, Addit: 0
;; QUESTIONS:
;;      elaine.best.stanford.edu, type = A, class = IN

;; ANSWERS:
elaine.best.stanford.edu.     0     CNAME elaine19.stanford.edu.
elaine19.stanford.edu.  3600  A     36.216.0.207

;; Total query time: 60 msec
;; FROM: cardinal1.Stanford.EDU to SERVER: default -- 36.56.0.150
;; WHEN: Thu Jul  6 22:13:57 1995
;; MSG SIZE  sent: 42  rcvd: 114
```

Figure 4: Dig results of first query

```
# dig elaine.best.stanford.edu

 [... header deleted ...]

;; ANSWERS:
elaine.best.stanford.edu.     0     CNAME elaine16.stanford.edu.
elaine16.stanford.edu.  3600  A     36.216.0.204
```

Figure 5: Dig results from second query

Perl 5 Server

Although the Perl 4 version has worked fine, it serves a single purpose: handing out load information in the “best.stanford.edu” domain. It is possible to modify it to serve other information was well, but doing so using Perl 4 would not have been easy due to the lack of nested data structures. After this paper was accepted, I decided to re-write the name server in Perl 5, using Perl 5 features like references to achieve nested data structures, and the Socket module to provide portability.

Server Organization

The Perl 5 server is organized into four different files: DNS.pm, lbnamed, lbnamed.conf, and LBDB.pm.

DNS.pm

DNS.pm is a Perl 5 package containing constants and functions that assist in creating and parsing DNS messages. It was created by starting with /usr/include/arpa/nameser.h and converting it into Perl 5. From there functions were added to expand compressed domain names, create resource records, etc. After importing DNS.pm, programs can use these functions and constants. For example:

```
$flags |= QR_MASK | AA_MASK | NOTIMP;
```

Functions are provided for encoding data into resource records (RRs). After the RR is encoded it is returned as string. Only the common RRs were implemented, see Figure 6.

An answer to a DNS query consists of of 6 pieces of data. The domain name which the resource records pertains to, the type of the RR, the class of the RR, the time-to-live (TTL) of the data, the length of the data, and the data the itself. The dns_answer function is used to create an answer:

```
$answer = dns_answer(QPTR, T_TXT,
    C_IN, 60, rr_TXT($date));
```

Note that the length of the resource data is not passed to the dns_answer function because the resource data is passed in as a string. The dns_answer function uses the string’s length as the resource data’s length since Perl strings can contain null characters, unlike null-terminated C strings. Also note the special constant “QPTR”. QPTR is a compressed domain name pointer which points to the original question in the DNS message. If the domain name of RR that is getting added to the answer section is the same as the domain name in the question you should use QPTR. QPTR takes only two bytes as opposed to duplicating the original domain name.

lbnamed

lbnamed is the main server. It loads lbnamed.conf, sets up the TCP and UDP sockets, and then answers requests after performing a select(2) on the TCP and UDP sockets. Upon receiving a request it calls the do_dns_request function which attempts to parse the DNS request. If the request is invalid (i.e., a unsupported operation is requested or the request could not be parsed), an error is returned. Otherwise lbnamed first checks to see if there is a static answer available. If not, it attempts to find a dynamic domain that will answer the question. If neither a static or dynamic answer is found then the NXDOMAIN (non-existent domain) error is returned.

A static domain name is a domain name that does not change from one query to the next. A dynamic domain name is a domain that can possibly change from query to query. Static and dynamic domains are discussed in detail a little later.

Figure 7 shows the code for checking for static and dynamic domain names. The response to the query is generated using Perl’s “pack” function, as shown in Figure 8. Note that the LBDB::check_static and LBDB::check_dynamic functions are free to modify the various variables in the $dnsmsg associative array, such as setting the response code (rcode) and adding data to the answer, authority, and additional section of the response.

```
$data = rr_A($ipaddress);
$data = rr_CNAME("stanford.edu");
$data = rr_HINFO("PowerPC","Solaris/2.5");
$data = rr_MX(10,"leland.stanford.edu");
$data = rr_NS("bestserver.stanford.edu");
$data = rr_NULL;
$data = rr_PTR("leland.stanford.edu");
$data = rr_SOA("foo.stanford.edu","root.stanford.edu",
            1234, 1200, 300, 604800,86400)
$data = rr_TXT("this is text");
```

Figure 6: Prototypes for implemented common resource record encodings

```
if (LBDB::check_static($qname,
     $qtype,$qclass,$dnsmsg)) {
   # return answer
} elsif (LBDB::check_dynamic($qname,
     $qtype,$qclass,$dnsmsg)) {
   # return answer
} else {
   $dnsmsg->{'rcode'} = NXDOMAIN;
}
```

Figure 7: Coding to check static/dynamic domain names

lbnamed.conf

lbnamed.conf is the place to put local modifications, and to define two function hooks which are called from lbnamed: do_maint, clean_exit. The do_maint function is called from the answer_requests function in lbnamed if the variable need_maint is set. For example, lbnamed.conf can install a signal handler to catch the HUP signal. This signal handler would set the need_maint variable so the do_maint function would get called. clean_exit is a function which cleanly shuts down the server.

lbnamed.conf also contains calls to the LBDB::add_static and LBDB:add_dynamic functions to add static and dynamic data. Under normal circumstances lbnamed.conf is the only file that needs to be changed.

LBDB.pm

LBDB.pm is a Perl 5 package that contains the functions for adding data to and checking for static and dynamic domain data.

Registering Static Domains

Static domain data (data that does not vary from query to query) is added using the LBDB::add_static function as shown in Figure 9. The database for static information is implemented using a four-level hash table:

```
$static_domain{$domain} ->
     {$dns_class} -> {$dns_type} =
            { ...data... }
```

The first-level hash table is indexed by the domain name of the data, the second level is indexed using the class of the data (such as C_IN), the third level is indexed using the type of the data (such as T_A), and the fourth level contains the information associated with the data of that domain, class, and type. This layout simplifies finding all the data associated with a given domain name, even when confronted with a query that contains C_ANY or T_ANY.

Static domain data can be used for any type of data, but will probably be used mainly for answering SOA queries for a dynamic domain. For example, to register an SOA record for the “best.stanford.edu” domain you would make the following call in the “lbnamed.conf” file:

```
LBDB::add_static("best.stanford.edu",
     T_SOA,
     rr_SOA(hostname, $hostmaster,
        time, 86400, 86400, 86400, 0)
);
```

```
$flags |=  QR_MASK | AA_MASK | $dnsmsg->{'rcode'};
$response = pack("n n n n n n", $id, $flags, $qdcount,
    $dnsmsg->{'ancount'}, $dnsmsg->{'nscount'}, $dnsmsg->{'arcount'})
        . $question
        . $dnsmsg->{'answer'}
        . $dnsmsg->{'auth'}
        . $dnsmsg->{'add'};
```

Figure 8: Creating response to query

```
sub add_static {
  my($domain,$type,$value,$ancount,$class,$ttl) = @_;

  $ancount = 1            unless $ancount;
  $class   = C_IN         unless $class;
  $ttl     = $default_ttl unless $ttl;

  $static_domain{$domain} -> {$class} -> {$type} = {
             "answer" => dns_answer(QPTR,$type,$class,$ttl,$value),
             "ancount" => $ancount
  };
}
```

Figure 9: LBDB::add_static

Registering Dynamic Domains

Dynamic domains (data that gets created dynamically, based on the name being queried) are added using the LBDB::add_dynamic function as shown in Figure 10. The database for dynamic information is implemented using a hash table:

```
$dynamic_domain{$domain} = $handler;
```

The hash table is indexed by the domain name of the data, and the value returned is a reference to a function which gets called at the time the query is made. The function for a dynamic domain is called with the following arguments:

```
&$dfunc($domain, $residual, $qtype,
     $qclass, $dnsmsg);
```

$domain is the dynamic domain (i.e., best.stanford.edu)
$residual is the data to the left of the dynamic domain (i.e., elaine)
$qtype is the type of the of the query (i.e., T_A)
$qclass is the class of the query (i.e., C_IN)
$dnsmsg is a reference to a hash table which is used to return information to the load balancing name server.

The function returns 1 if it executed successfully (i.e., the results in $dnsmsg should be used) or 0 otherwise.

```
sub add_dynamic {
  my($domain, $handler) = @_;
  $dynamic_domain{$domain} = $handler;
}
```

Figure 10: LBDB::add_dynamic

The algorithm for finding a dynamic domain attempts to find the longest dynamic domain name that matches the query. For example, if we had the following dynamic domains registered:

```
stanford.edu
best.stanford.edu
```

And the following query came in:

```
elaine.best.stanford.edu
```

Then the handler for the "best.stanford.edu" domain would be called, since it is the longest match for "elaine.best.stanford.edu" Here is how the algorithm matches "elaine.best.stanford.edu":

```
     domain                  residual  match
"elaine.best.stanford.edu"  ""          no
"best.stanford.edu"         "elaine"   yes
```

If the query was "foo.bar.stanford.edu" then the match would look like:

```
     domain             residual  match
"foo.bar.stanford.edu"  ""          no
"bar.stanford.edu"      "foo"       no
"stanford.edu"          "foo.bar"  yes
```

Dynamic domains are the heart of the load balancing name server as they allow you to create answers dynamically based upon the name being queried. The best way to explain dynamic domains is with an example.

Let's create a domain called "random. stanford.edu", which will return a different random number between 0 and 10 every time it is called. We register that domain by adding the following calls to lbnamed.conf:

```
sub handle_random {
   my($domain, $residual, $qtype,
            $qclass, $dm) = @_;
   $dm->{'answer'} .= dns_answer (
        QPTR, T_TXT, C_IN,
        60, rr_TXT(int(rand(10))));
   $dm->{'ancount'} += 1;
   return 1;
}

LBDB::add_dynamic(
    "random.stanford.edu" =>
    \&handle_random);
```

By calling LBDB::add_dynamic we are requesting that the load balancing name server call our function whenever a request comes in for the name "random.stanford.edu". The first statement calls the dns_answer function which creates the binary data which will be placed in the answer section of the DNS message:

```
$dm->{'answer'} .= dns_answer(
         QPTR,  T_TXT, C_IN,
         60, rr_TXT(int(rand(10)))
);
```

QPTR is a constant defined in DNS.pm that is a compressed domain name pointer which points to the original question in the DNS message. T_TXT is the type of data being returned. C_IN is the class of the data being returned. 60 is the time-to-live (TTL) of the data being returned (in seconds), and rr_TXT is a function which given a text string returns a text resource record. The second statement increments the answer count in the reply message. The response code for the reply is set to NOERROR by default, so there is nothing else for us to set.

Let's say we also want to define a domain that returned a random number between 0 and 100. It would be easy to do something like:

```
LBDB::add_dynamic(
    "random100.stanford.edu"
       => \&handle_random_100);
```

But that solution does not scale. The solution is to modify the original handle_random function so that it examines the residual part of the domain name passed to it. For example:

```
sub handle_random {
   my($domain, $residual, $qtype,
              $qclass, $dm) = @_;

   $residual = 10 unless $residual;

   $dm->{'answer'} .= dns_answer(
       QPTR, T_TXT, C_IN, 60,
       rr_TXT(int(rand($residual))));

   $dm->{'ancount'} += 1;
   return 1;
}
```

This enables us to make a query in the form:

```
random.stanford.edu
```

or:

```
N.random.stanford.edu
```

where the return value will be between 0 and N; see Figure 11 for an example. While "random.stanford.edu" is not a useful domain, it helps show the basic concepts involved in creating dynamic domains. A more useful example would create a dynamic domain that mimicked the Hesiod "passwd" domain in the Athena environment.

Using a standard BIND server that understands class HS and type TXT records, you need a database entry in the domain file for each user in the password file; see Figure 12 for an example. If you have a large password file (like Stanford's 22,000 users), then BIND will consume a lot of memory loading every single passwd entry. It also means that to add/delete/update an entry you'll have to reload the whole file. Using lbnamed you register a dynamic domain:

```
sub handle_passwd_request {
  my($domain, $residual, $qtype,
                  $qclass, $dm) = @_;

  my($name, $passwd, $uid, $gid,
     $q, $c, $gcos, $dir, $shell) =
      getpwnam($residual);
  my($entry);

  if ($name) {
     $entry = "$name:*:$uid:$gid:".
                          "$gcos:$dir:$shell";
  } else {
       $dm->{'rcode'} = NXDOMAIN;
       return 1;
  }

  $dm->{'answer'} .=  dns_answer(QPTR,
      T_TXT, C_HS, 3600,
      rr_TXT($entry));
  $dm->{'ancount'} += 1;
  return 1;
}

LBDB::add_dynamic(
    "passwd.ns.stanford.edu"  =>
            \&handle_passwd_request);
```

Now if someone attempts to lookup the name "root.passwd.ns.stanford.edu", the lookup will get re-directed to the handle_passwd_request, which will lookup the passwd information and construct the correct response dynamically. Note that depending on the OS, the getpwnam call could be getting the password information from a local file, a DBM file, or even NIS/NIS+. You could also replace the getpwnam call with your own function that obtains information from a DBM file or even a relational database.

Results/Conclusions

Overall lbnamed has been a big win at Stanford. It has helped distribute the load among a large number of workstations. It also enables system administrators to take systems down temporarily without interrupting users who use the load balancing name. It also allows systems to be transparently added and removed from groups.

The problems have been minor. The biggest problem has been hosts that respond to load balancing queries, but don't allow logins (due to other problems). This could be fixed by falling back to a strict round-robin scheme in between polls, or some variant, such as changing the weight of the host handed out to be slightly more than the host in the middle of the list, for example.

```
# dig 100.random.stanford.edu

 [... header deleted ...]

;; ANSWERS:
100.random.stanford.edu.      60    TXT   "97"

 [... trailer deleted ...]
```

Figure 11: Querying N.random.stanford.edu

```
root.passwd HS TXT "root:*:0:1:Root Account:/:/bin/sh"
```

Figure 12: Sample database entry for root user

The other problem has been resolver clients that don't deal well with TTL values of 0. This has only happened in a few cases and generally only happens in clients with old software.

Some people may not feel comfortable using a TTL of 0, but I personally don't have any trouble sleeping at night because I chose it. As mentioned in RFC 1794, there are plenty of versions of BIND that treat anything less than 300 seconds as 300 seconds, which can defeat the whole purpose of trying to balance the load. I figured the added load on the name servers was worth the benefit of getting a truly dynamic response to every query. When trying to load balance your cache can be trash...

Future Directions

One of the reasons I decided to write this paper was to get people thinking about "exotic" name servers. There are a number of directions someone could take the concepts presented here, some of which have already been hinted at in a future version of BIND. For example, the registration of dynamic domains could be added to BIND by loading shared objects at runtime, or by allowing external daemons to register with BIND and communicate via IPC mechanisms.

As far as the Perl implementation of lbnamed is concerned, a number of improvements immediately come to mind:

- recognizing when a particular host is consistently handed out as being the best, even though no one can login to it.
- adding more factors to the determine the weight of a host, such as a swap space, free memory, number of processes, CPU model, etc.
- modifying the poller protocol so poller clients can specify their weight rather then letting the poller calculate it for them. For example, you could load balance requests to a name such as "www.stanford.edu" based on the average number of requests over the last few minutes, etc.
- modifying the poller so it periodically reloads the IP addresses of clients into its cache.
- adding logging and statistics back to the Perl 5 version.
- generalize support for domain name compression in answers.

Acks

The Perl 4 version was written while I was at Stanford. The Perl 5 code was written (in my free time after work) after I left Stanford, and after this paper was accepted. Special thanks to Shirley Gruber at Stanford University and Kevin Kluge at SunSoft for finding and correcting plenty of errors in early versions of this paper. My English is a little better and they probably know a little more about DNS ;-)

Availability

The code is available using the following URL: http://www-leland.stanford.edu/~schemers/dist/lb.tar Use the code at your own risk. The Perl 4 version has been in use at Stanford for over two years.

Author Info

Roland Schemers received his M.S degree in Computer Science from Oakland University in Rochester, Michigan. He currently is working in the DCE engineering group at SunSoft. He previously worked in the Distributed Computing Group at Stanford, and helped manage and maintain such campus-wide services as AFS, Kerberos, and DNS, as well as the public workstation clusters and servers. He can be reached electronically at <schemers@ eng.sun.com>.

While at Stanford he also co-authored a chapter in the book *Distributed Computing, Implementation and Management Strategies*, Raman Khanna, Editor, which probably would have sold better if it had the words "Client/Server" in the title.

He is patiently waiting for the day when he can login into any UNIX system and access /usr/bin/perl.

References

[1] Larry Wall, Randal L. Schwartz, *Programming perl*, O'Reilly and Associates, Sebastopol, CA.

[2] Stephen P. Dyer, *The Hesiod Name Server*, Proc. USENIX Winter Conference, 1988.

[3] Paul Albitz, Cricket Lui, *DNS and BIND*, O'Reilly and Associates, Sebastopol, CA.

[4] Paul Vixie, BIND, http://www.isc.org/isc/

[5] Thomas P. Brisco, *DNS Support for Load Balancing*, RFC 1794.

[6] Salvatore DeSimone, Christine Lombardi, *Sysctl: A Distributed System Control Package*, Proc. USENIX LISA Conference, November 1993.

[7] Dan Farmer, Wietse Venema, SATAN, satan@fish.com.

APPENDIX

Poller configuration file

The poller configuration file tells which hosts the poller should poll, and which dynamic groups those hosts are in. The format is:

```
host    weight-multiplier   group1 [group2 ...]
```

The weight-multiplier field is currently not used but could be used in the future to allow for better selection among different hardware in the same group.

The following is a sample poller configuration file with some lines removed to save space.

```
#
# groups
# -------------------------
# sweet     all machines
# elaine    elaine1-elaine57
# sparc     elaine1-elaine57
# sunos     elaine1-elaine57
# sparc2    sparc2 (elaine1-elaine19)
# sparc1    sparc1 (elaine20-elaine57)
# adelbert  adelbert1-adelbert26
# ultrix    adelbert1-adelbert26
# dec       adelbert1-adelbert26
# dec5000   adelbert1-adelbert13
# dec3100   adelbert14-adelbert26
# rs        rs1-rs10
# rs6000    rs1-rs10
# aix       rs1-rs10
#
rs1    1    rs rs6000 aix
rs2    1    rs rs6000 aix
rs10   1    rs rs6000 aix
#
elaine1    1    elaine sparc2 sparc sunos sweet
elaine2    1    elaine sparc2 sparc sunos sweet
elaine19   1    elaine sparc2 sparc sunos sweet
#
elaine20   1    elaine sparc1 sparc sunos sweet
elaine21   1    elaine sparc1 sparc sunos sweet
elaine57   1    elaine sparc1 sparc sunos sweet
#
adelbert1  1    adelbert dec5000 dec ultrix sweet
adelbert2  1    adelbert dec5000 dec ultrix sweet
adelbert13  1    adelbert dec5000 dec ultrix sweet
#
adelbert14  1    adelbert dec3100 dec ultrix sweet
adelbert26  1    adelbert dec3100 dec ultrix sweet
#
```

lbnamed configuration file

The lbnamed configuration file tells lbnamed what the weight of each host is, what its IP address is, and which dynamic groups a hosts is in. The format is:

```
weight host ipaddress group1 [group2 ...]
```

The following is a sample lbnamed configuration file with some lines removed to save space.

```
2200 elaine11 36.214.0.127 elaine sparc2 sparc sunos sweet
639 adelbert10 36.211.0.81 adelbert dec5000 dec ultrix sweet
651 elaine20 36.215.0.208 elaine sparc1 sparc sunos sweet
2336 elaine3 36.212.0.119 elaine sparc2 sparc sunos sweet
...
```

```
866 adelbert6 36.211.0.76 adelbert dec5000 dec ultrix sweet
243 adelbert26 36.212.0.201 adelbert dec3100 dec ultrix sweet
```

Protocol

The protocol between the poller and client daemon is simple. Everything is in network byte order. I used UDP so I could easily send out multiple polls at the same time and receive responses asynchronously. The packet format (described by C structures) is:

```
#define PROTO_PORTNUM 4330
#define PROTO_MAXMESG 2048    /* max udp message to receive */
#define PROTO_VERSION 2

typedef enum P_OPS {
    op_lb_info_req             =1,  /* load balance info, request and reply */
} p_ops_t;

typedef enum P_STATUS {
    status_request             =0,  /* a request packet */
    status_ok                  =1,  /* ok */
    status_error               =2,  /* generic error */
    status_proto_version       =3,  /* protocol version error */
    status_proto_error         =4,  /* any other protocol error */
    status_unknown_op          =5,  /* unknown operation requested */
} p_status_t;

typedef struct {
  u_short   version;  /* protocol version */
  u_short   id;       /* requestor's uniq request id */
  u_short   op;       /* operation requested */
  u_short   status;   /* set on reply */
} P_HEADER,*P_HEADER_PTR;

typedef struct {
  P_HEADER h;
  u_int boot_time;
  u_int current_time;
  u_int user_mtime;  /* time user information last changed */
  u_short l1; /* (int) (load*100) */
  u_short l5;
  u_short l15;
  u_short tot_users;  /* total number of users logged in */
  u_short uniq_users; /* total number of uniq users */
  u_char  on_console; /* true if someone on console */
  u_char  reserved;   /* future use, padding... */
} P_LB_RESPONSE, *P_LB_RESPONSE_PTR;
```

The protocol was meant to be extensible but I have yet to use the daemon for anything but load balancing requests.

fping

The poller daemon was inspired by a previous program I wrote called fping. fping is a ping(8) like program which uses the Internet Control Message Protocol (ICMP) echo request to determine if a host is up. fping is different from ping in that you can specify any number of hosts on the command line, or specify a file containing the lists of hosts to ping. Instead of trying one host until it times out or replies, fping will send out a ping packet and move on to the next host in a round-robin fashion. If a host replies, it is noted and removed from the list of hosts to check. If a host does not respond within a certain time limit and/or retry limit it will be considered unreachable. fping is used by SATAN [7] to quickly ping a list of hosts and/or ip addresses.

fping is currently being maintained and updated by R. L. "Bob" Morgan <morgan@networking.stanford.edu> and can be obtained via the following URL:

```
ftp://networking.stanford.edu/pub/fping/fping.2.0.tar.gz
```