Enhancing DNS for Improved E-mail Services in a Clustered Environment

Kostas Zorbadelos
Otenet SA
Kifisias Ave. 109 and Sina
15124, Marousi, Greece
kzorba@otenet.gr

Alex Delis
Deprt. of Informatics
The University of Athens
15771, Athens, Greece
ad@di.uoa.gr

Abstract

The Domain Name System (DNS) affects most Internet services including e-mail. It has scaled well with the growth of the Internet but it also has limitations. In normal DNS operation it is likely for users to be directed to "sub-optimal" targets for obtaining service or even to nodes in which there is no service availability at a specific time interval. We propose an enhanced name-server that functions in cooperation with the open-source BIND and takes into consideration the state of the machines that provide the mail service. The main task of our name-server is to effectively carry out host-name resolution to cluster-node IPs even in light of failed or overloaded cluster nodes. We present the server's software architecture and the techniques for load evaluation to attain near-optimal name-to-address resolution.

1 Introduction

E-mail still enjoys tremendous growth worldwide and Internet Service Providers (ISPs) have to continually improve the delivery of e-mail services to millions of subscribers. Even medium-size ISPs have to accommodate the needs of tens of thousands of users. As the volume of subscribers increases, optimization techniques are necessary to keep the service running smoothly and prevent it from ungraceful degradation. Distribution of load in a cluster of machines that provide a service is common. Various criteria are used to evaluate the load of each node so as to direct the client to the "best" possible machine for service. There are various products from many manufacturers that attempt to address the problem of load sharing [11]. However, many of these products perform load evaluation at the network level, without taking into consideration the characteristics of the specific service at the application layer. Furthermore, some depend on specific other products for their operation, or were designed for performance tuning of other product offerings in the first place.

We suggest a solution to the problem of load sharing that takes into consideration the following criteria:

- Depends on core Internet protocols and standards (DNS)

- Is independent of specific vendor product offerings
- Takes into account the characteristics of the mail services at the application layer
- Effectively,transparently and dynamically eliminates the failed nodes in the cluster diverting the load to the other machines
- Performs frequent polls to obtain an as good and as current as necessary picture of the load
- Although described for mail services can be adapted to other applications
- Is based on open source components

Related approaches that use DNS to attain load sharing are discussed in [7, 2]. The most common and widely used, is the Round Robin feature of BIND [1]. In this, the addresses of a specific domain name are returned to the client in a round robin fashion but without more elaborate load evaluation or even check for availability of the destination machine. The *Internet Software Consortium* has expressed an opinion [7] for being reluctant to introduce more sophisticated load balancing code in BIND. They propose as an alternative to use a separate name-server to handle the domain names with any type of reordering suitable for the application at hand. The `lbnamed` has followed this approach and uses a a separate name-server to handle domains with dynamic re-ordering [13].

In this paper, we present the design rationale and algorithms for load evaluation for an enhanced name-server which cooperates with the open source BIND. The name-server in question handles only delegated domains used for load balancing of mail services. It should not be considered a substitute for BIND and provides only a small subset of BIND's capabilities in responding DNS queries.

2 The Operational Environment

Our operational environment is depicted in Figure 1. We have three classes of servers for the mail services and we also have a filer (*Network Attached Storage*) that contains the user mailboxes. The machines that need access to the mailboxes mount the space over NFS. All the servers are multi-homed to distinguish the network traffic towards the outside world, from the internal NFS traffic towards the filer. This is desirable for both security reasons (better access control to the filer) as well as performance. NFS performance is especially sensitive and requires a lot of tuning [14, 8], so having the NFS traffic in a separate LAN is generally a good idea.

As we can see, the machines have names in a virtual domain (lb.ourdomain) that is delegated to the enhanced name-server. The first class (mailgate.lb.ourdomain) contains the servers that are used by subscribers as "outgoing". Users declare the name mailgate.ourdomain as outgoing SMTP server to their Mail User Agents (MUAs) so all mail traffic generated by our subscribers is delivered to the outside world by this class of machines. The domain name mailgate.ourdomain is therefore a CNAME to mailgate.lb.ourdomain. These machines do not need access to the mailboxes of the subscribers.

The second class (mail.lb.ourdomain) is advertised as MX record for our domain. To be precise, what is advertised is mail.ourdomain which is a CNAME to mail.lb.ourdomain. Although MX

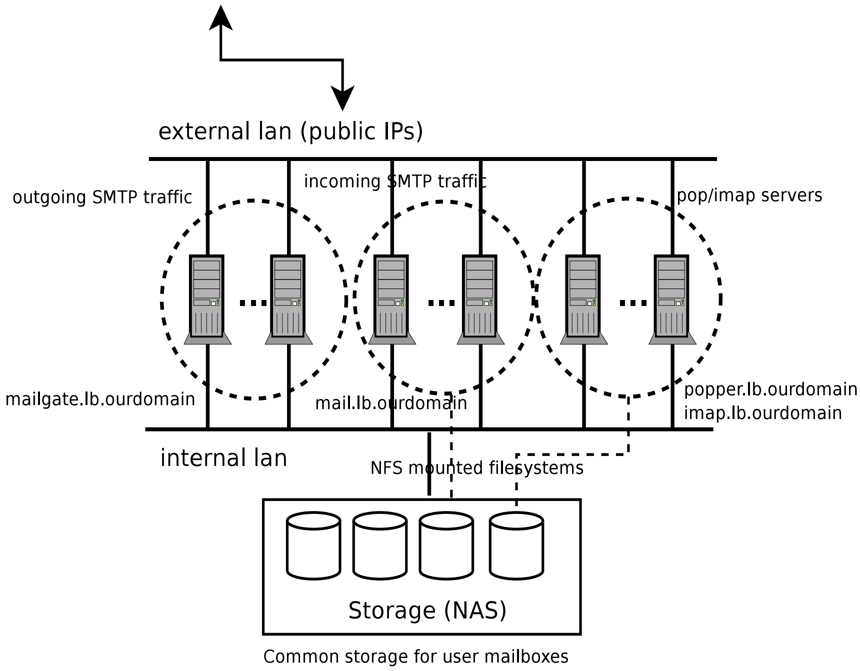

Figure 1: The clustered environment for mail services

records should contain only canonical domain names and not CNAMES to avoid mailer loops, in our case there is only one MX record with the lowest preference. This record points to a different IP address each time because of the way our enhanced name-server works. In the rare case where a mailer trying to deliver to ourdomain is directed to an IP which is not available, the mailer differs delivery and as it retries it finally obtains a different IP in the mail.lb.ourdomain pool. We do not expect the client resolver to have cached previous IP addresses due to the short TTL returned by our enhanced name-server. The machines in this class are the only machines that deliver mail to the users' mailboxes.

Finally the third class (popper.lb.ourdomain or imap.lb.ourdomain) contains the servers that provide access to users' mailboxes via pop or imap. Subscribers declare popper.ourdomain or imap.ourdomain in the configuration of the tools they use to fetch their mail to their local machines, or their MUAs. The popper.ourdomain and imap.ourdomain are CNAMES for popper.lb.ourdomain and imap.lb.ourdomain respectively.

The relevant part of the setup in the parent ourdomain zone is presented in Figure 2. With this setup, the delegated virtual sub-domain lb remains transparent to the final users.

3 Service Classes Affected by DNS

In the case of an ISP subscriber that tries to send mail, she most probably declares to her MUA software a host-name to use as an outgoing SMTP server. This server is configured to relay mail

```
@    IN    MX    0  mail.ourdomain.

mailgate IN CNAME mailgate.lb.ourdomain.
mail     IN CNAME mail.lb.ourdomain.
popper   IN CNAME popper.lb.ourdomain.
imap     IN CNAME imap.lb.ourdomain.
```

Figure 2: The ourdomain zone configuration

for subscribers. She could also use the MTA running on her local machine and configure it to forward all mail to the ISP's SMTP server ("smarthost" configuration). In any case, the IP of the relay server is required to establish a SMTP connection. The MUA or local MTA therefore performs a DNS query to locate the IP and as a result, it finally receives a record containing the wanted IP. The same scenario is performed whenever a user tries to establish a session with a pop or imap server to retrieve mail from her remote mailbox. In our environment and for either case, the user is directed to one of the machines in the cluster (mailgate group in the case of sending mail, popper or imap for accessing her mailbox).

The case of mail delivery is different though. For the mail delivery, another type of DNS RR (Resource Record) is utilized, namely the MX records. The exact algorithm used by Mail Transfer Agents to deliver a mail to its final destination is described in RFC 974 [12]. The MTA performs an MX query to find the relevant records for the destination domain with the corresponding preferences. It then tries each MX in turn starting with the one that has the lowest value associated with it. To avoid mail loops the MTA checks if itself is part of the list, in which case it discards all records with preference values greater than or equal to its own. Finally, of course, the MTA needs to find the IP address of the corresponding MX so as to attempt the delivery. Most MTAs are tolerant in case they don't find MX records for a destination domain, and try as a last resort to deliver the mail to the host included in the address. In our environment, the delivery of mail items destined to ourdomain will end up in one of the machines in the "mail" group.

4 Bottlenecks and Load Factors

One of the most sensitive factors that can affect the performance of the entire service, is the NFS protocol used in the mounted file system containing the users' mailboxes. For the NFS server part, there are optimized filer products such as the *Netapp Filers* which can provide very good I/O throughput and minimize the NFS server bottleneck. For the client part, there are performance tuning techniques, but require experimentation for calibration. These methods also depend upon the specific platform NFS is implemented. [14, 8] offer a number of guidelines for NFS performance tuning and problem diagnosis. Since NFS is a network based protocol, the underlying network performance is also vital to its operation.

At the application layer, all MTAs use one or more queues to store mail items that wait to be delivered. Queue analysis [3, 10, 4] can provide valuable metrics regarding the load of each MTA. Generally speaking, the mail items' count that wait in the queue for delivery and their size, provide a good estimate of an MTA's load. Another bottleneck point can be the locking that is required in the delivery of mails to local mailboxes. We can overcome this by using maildir [9] format for the mailboxes. Document [16] provides a benchmark for the use of mbox vs maildir in IMAP services.

In the next section, we provide formulas that take into account the aforementioned factors. Our overall objective is to maximize the throughput (number of mail items delivered to their destination per second) by distributing the load evenly across the machines that constitute the cluster.

5 Load Evaluation and Distribution Algorithms

For every class of machines deployed (mailgate, mail, popper/imap) we establish different criteria. The common part in all cases, is that the load is expressed as a single number and load estimation takes into account a "history window" that is, the load samples within a specific time frame.

• **Outgoing Mail:** In the case of the mailgate class of machines, we take into consideration the general load average of the machine and the size of the mail queue. In this case NFS is not involved, since the MTAs in these machines do not interact with the mailboxes. At the time when the pollerc performs a sampling, it breaks the mail queue into time intervals. The recent time intervals have fine granularity while the older ones have geometrically less fine granularity. The messages contained in the recent intervals are given greater weight in the computation of load while the older ones less. This happens, because in general the most recent messages are the ones that are being processed while the older ones are more likely "differed" because of an earlier failed delivery attempt. The differed messages are retried in a growing time frame between retransmissions. Therefore it is logical to assume that the younger messages affect the current load of the MTA more than the older ones. The following formula expresses all the above:

$$l_s = l_m \cdot \sum_{i=1}^{k} c_i \cdot n_i \cdot \bar{S}_i \qquad (1)$$

where, l_s is the sample of the load at a specific time; l_m is the load average of the machine (as given for example by the top [15] command), c_i is the weight of the messages in the i-th time interval of the queue (with the first interval containing the younger messages), n_i is the count of the mail items contained in the i-th time interval, $\bar{S}_i$ is the average size (in KB or MB) of the mail items of the i-th time interval, and finally k is the count of the time intervals we break the queue into. We also have $c_i > c_{i+1}$ that is the younger messages affect the current load more than the older ones.

Finally, we take into consideration the N previous samplings so we have a history window of

$N \cdot t_s$ size (t_s is the time between the samplings). So the final load of a node is determined as:

$$l = \frac{1}{N} \sum_{i=1}^{N} l_{s_i} = \bar{l_s} \tag{2}$$

• **Local Mail Delivery:** The machines that belong in the “mail” group perform the deliveries to the local user mailboxes. In this case, we take into consideration the general load of the machine, the size of the mail queue (items that wait to be delivered) and also the NFS overhead in the load evaluation. The following formula binds these factors together:

$$l_s = l_m \cdot n_q \cdot \bar{S_q} \cdot t_{svctm} \tag{3}$$

where, l_s is the sample of the load at a specific time, l_m is the load average of the machine, n_q is the count of the mail items contained in the queue, $\bar{S_q}$ is the average size (in KB or MB) of the mail items in the queue, and lastly t_{svcmt} is the average service time (in milliseconds) for I/O requests that were issued to the NFS file system.

Considering the *N* previous samplings for a history window of $N \cdot t_s$ size (t_s is the time between the samplings), we finally obtain:

$$l = \frac{1}{N} \sum_{i=1}^{N} l_{s_i} = \bar{l_s} \tag{4}$$

• **POP/IMAP Services:** The machines in the popper/imap class have the greatest NFS interaction with the mailboxes. The number of concurrent user sessions, the size of the session mailbox and as always the general load of the machine are the factors contributing to the load computation:

$$l_s = l_m \cdot t_{svctm} \cdot \sum_{i=1}^{N_s} n_{m_i} \cdot \bar{S_{m_i}} \tag{5}$$

where, l_s is the sample of the load at a specific time, l_m is the load average of the machine, t_{svcmt} is the average service time (in milliseconds) for I/O requests that were issued to the NFS file system, N_s is the count of the pop/imap sessions, n_{m_i} is the count of the mail items in the user mailbox corresponding to the i-th session, and $\bar{S_{m_i}}$ is the average size (in KB or MB) of the mail items in the mailbox corresponding to the i-th session.

When taking into consideration the history window of the *N* previous samplings, we have:

$$l = \frac{1}{N} \sum_{i=1}^{N} l_{s_i} = \bar{l_s} \tag{6}$$

6 The Enhanced Name-Server

Our enhanced name-server consists of two parts, the main enh-named and a pollerc. The multi-threaded enh-named's software architecture appears in Figure 3. The program starts, binds to the

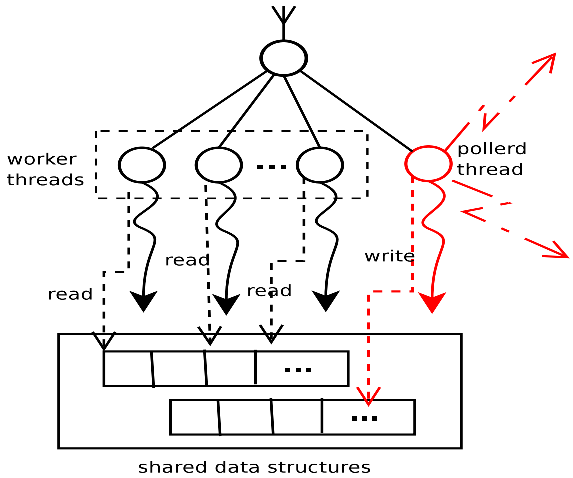

Figure 3: The multi-threaded enh-named

machine's address and listens for incoming requests on TCP and UDP ports 53 (domain service). Upon start, it reads a configuration file that contains among other things the pools of the hosts that provide the mail services and the port each pollerc listens for probes by the pollerd daemon.

```
#-------------------------------------------------------
# enh-named configuration
#-------------------------------------------------------

zone lb.ourdomain {
  ...
  pollerd_wakeup = num
  ...
  # The config for the first pool of machines (mailgate).
  # This creates the name mailgate.lb.ourdomain
  group mailgate {
    host mailgate1 {
      # the canonical name of the host
      cn=mailgate1.ourdomain
      # the host ip address
      ip=ip1
      # the port that the pollerc listens on that machine
      pollerc_port= port1
    }
    ...
    host mailgateN {
      cn=mailgateN.ourdomain
      ip=ipN
      pollerc_port= portN
    }
  }

  # The config for the second pool of machines (mail).
  # This creates the name mail.lb.ourdomain
  group mail {
    host mail1 {
      cn=mail1.ourdomain
      ip=ip1
      pollerc_port= port1
    }
    ...
```

```
    host mailN {
      cn=mailN.ourdomain
      ip=ipN
      pollerc_port= portN
    }
  }

  # Finally the config for the popper/imap pool of machines.
  # This creates the name popper.lb.ourdomain
  group popper {
    host popper1 {
      cn=popper1.ourdomain
      ip=ip1
      pollerc_port= port1
    }
    ...
    host popperN {
      cn=popperN.ourdomain
      ip=ipN
      pollerc_port= portN
    }
  }

  # This creates the name imap.lb.ourdomain
  group imap {
    host imap1 {
      cn=imap1.ourdomain
      ip=ip1
      pollerc_port= port1
    }
    ...
    host imapN {
      cn=imapN.ourdomain
      ip=ipN
      pollerc_port= portN
    }
  }
  ...
}
```

A list of all hosts is initially created. This list has a record for each host, that contains the name(s) of the machine, its IP, a boolean field that indicates whether it is alive, the canonical domain name and the machine's current load. For each group of machines an array is created that contains references to the hosts list. This situation is depicted in Figure 4.

Our enh-named is a multi-threaded program that has a number of worker threads that service the incoming DNS requests, plus a special thread that is our poller daemon (pollerd). The job of the pollerd thread is to wake every specified time interval and communicate with each external machines' pollerc to obtain their corresponding load. The pollerd uses UDP protocol for this communication and sends asynchronously requests to each pollerc running on the mail server cluster nodes. Upon receiving an answer that contains the load of the specified machine, the pollerd updates the relevant data structure with the current load. If pollerd doesn't receive an answer from a machine after a timeout period, it marks it as dead and updates the data structures accordingly. pollerd is the only thread that writes to the shared data structures with all others being readers.

The worker threads' role in enh-named, is the actual service of the incoming DNS queries.

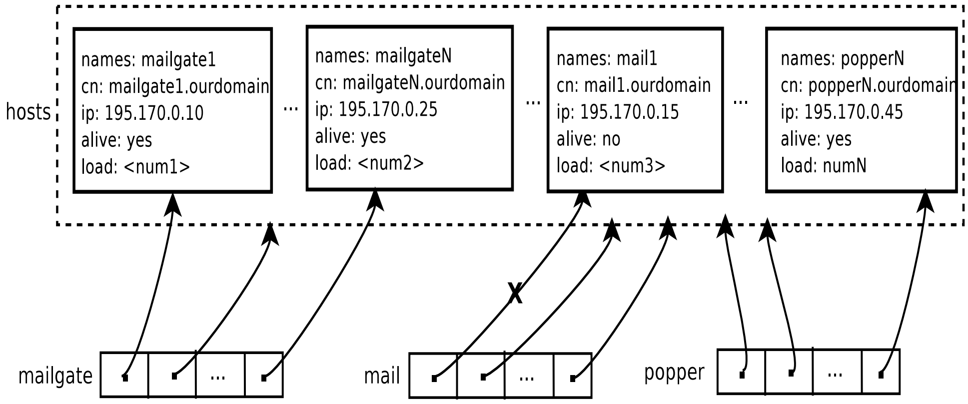

Figure 4: enh-named pertinent data structures

Our name-server is not a full featured domain server like BIND, so it understands a subset of the possible DNS queries (no recursive queries for example) and answers only for the domains it knows about. Client resolvers should not be pointed to our name-server directly. For each incoming query that has to do with a machine class, a worker thread returns an answer, based on the current load of the machines in the class. The array relevant to the name being asked is sorted according to 'live machines with the lowest load' and the machine with the lowest load is returned as an answer. The TTL of the answer is very small, to avoid its caching by other resolver clients. Such caching would defeat the whole purpose of dynamic load-balancing that we try to achieve. The TTL should be equal to or less than the pollerd's polling interval.

The pollerc program runs on every machine in the cluster. Upon startup, it reads its configuration file, binds to the host's address and listens for probes from the pollerd thread to the port specified. The pollerc part is the one that implements the algorithms described in section 5. When contacted by the pollerd on the UDP port it listens to, pollerc returns its machine's current load. The configuration file contains all necessary information the pollerc needs to perform its role.

```
#-------------------------------------------------------
# Pollerc configuration
#-------------------------------------------------------
pollerc {
       # the UDP port pollerc listens for
       # requests from pollerd
       port=<num>
       # the type of algorithm pollerc implements
       # for the evaluation of load
       type=<outgoing-smtp|local-delivery|popper|imap>
       # sample time interval
       # every that interval pollerc will perform a
       # sample of the machine's load
       polling_interval=<num>
       # this affects the history window described
       # in the algorithms section
       # this number of previous samplings are taken
       # into consideration for the current evaluation
       # of load
       history_polls=<num>
}
```

In order for our setup to successfully operate, we need to delegate a "virtual" zone from our domain to the *enh-named*. This is done by adding the proper NS records in ourdomain zone file (figure 5). With this setup, we have 2 enhanced named servers that operate independently of each other for redundancy. Each DNS query for the machines in the mail cluster, arrives in our name-server who responds with the alive machine that has the lowest load at the time.

A subtle point that is worth mentioning, is that our name-server responds with a CNAME record and not with an A. The CNAME points to the canonical domain name of the machine (which is the reason we need the canonical name in the configuration file). The answer returned also contains the IP of the machine in order to avoid the extra query to our parent name-server. The IP is known to our name-server as we need it for the communication with the pollerc and we include it in the configuration file. By returning a CNAME instead of an A record we avoid returning an answer to the resolver client that doesn't have a corresponding PTR record. Finally, as we have earlier mentioned, the answer has a small TTL to avoid caching, typically the polling interval or a fraction of it.

```
lb   IN  NS lbns1.ourdomain.
lb   IN  NS lbns2.ourdomain.
```

Figure 5: Delegation of the lb zone from the parent ourdomain

7 Conclusions

Our enhanced name-server is a proposal for load balancing that is independent of specific product offerings, takes into account the characteristics of the services at the application layer and dynamically eliminates failed nodes, redirecting their load to other machines. This way it also provides a way for easier scheduling of server downtime for upgrades and maintenance. Although mainly it is a work in progress, the ideas behind it address the issues of load balancing using open, widely implemented standards. There is room for improvements and also possibility to deploy the name-server to other applications, by writing and putting into service pollerc programs that perform the load evaluation with different algorithms.

Acknowledgments: we would like to thank Edwin Kremer for commenting on earlier drafts of this note as well as Achilles Voliotis and Kostas Tavernarakis of Otenet for very informative discussions.

References

[1] Paul Albitz and Cricket Liu. *DNS and BIND*. Number ISBN: 0-596-00158-4. O Reilly and Associates, Inc., 2002.

[2] Thomas P. Brisco. RFC 1794: DNS Support for Load Balancing, April 1995.

[3] Victor Duchovni. Postfix Bottleneck Analysis. `http://www.postfix.org/QSHAPE_README.html`.

[4] Victor Duchovni. qshape(1) man page, Postfix documentation.

[5] Terry Gray. Message Access Paradigms and Protocols. `ftp://ftp.cac.washington.edu/mail/imap.vs.pop`, September 1995.

[6] iostat(1) man page.

[7] BIND and Load Balancing. `http://www.isc.org/products/BIND/docs/bind-load-bal.html`.

[8] Chuck Lever. Using the Linux NFS Client with Network Appliance Filers. Technical report, Network Appliance, March 2004.

[9] maildir(5) man page.

[10] mailq(1) man page, included in sendmail, postfix, exim.

[11] Netcraft faq. `http://uptime.netcraft.com/up/accuracy.html`.

[12] Craig Partridge. RFC 974: Mail Routing and the Domain System, January 1986.

[13] Roland J. Schemers. *lbnamed*: A Load Balancing Nameserver in Perl. `http://www.stanford.edu/~riepel/lbnamed`, September 1995.

[14] Seth Vidal Tavis Barr, Nicolai Langfeldt and Tom McNeal. Linux NFS HowTo. `http://www.tldp.org/HOWTO/NFS-HOWTO/index.html`, August 2002.

[15] top(1) man page.

[16] Sam Varshavchik. Benchmarking `mbox` vs `maildir`. `http://www.courier-mta.org/mbox-vs-maildir`, March 2003.