MyXDNS: A Request Routing DNS Server With Decoupled Server Selection*

Hussein A. Alzoubi
Case Western Reserve University
hussein.alzoubi@case.edu

Michael Rabinovich
Case Western Reserve University
misha@eecs.case.edu

Oliver Spatscheck
AT&T Research Labs
spatsch@research.att.com

ABSTRACT

This paper presents the architecture and the preliminary evaluation of a request routing DNS server that decouples server selection from the rest of DNS functionality. Our DNS server, which we refer to as MyXDNS, exposes well-defined APIs for uploading an externally computed server selection policy and for interacting with an external network proximity service. With MyXDNS, researchers can explore their own network proximity metrics and request routing algorithms without having to worry about DNS internals. Furthermore, MyXDNS is based on open-source MyDNS and is available to public. Stress-testing of MyXDNS indicated that it achieves its flexibility at an acceptable cost: a single MyXDNS running on a low-level server can process 3000 req/sec with sub-millisecond response even in the presence of continuous updates to server selection policy.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]

General Terms

Design, Performance

Keywords

DNS, Request Routing, Load Balancing, Network Proximity

1. INTRODUCTION

The domain name system (DNS) is a vital part of the Internet infrastructure that provides mapping between human-readable host names (such as "case.edu") and numerical Internet addresses used for packet routing. Since it can resolve the same hostname to different IP addresses for different queries, DNS represents a convenient mechanism to distribute client requests among multiple server replicas. Consequently, the DNS infrastructure has been used extensively in providing scalability to Internet platforms. It is the basis for Web request routing in content delivery networks, for load balancing on a server farm, and for proximity-based request routing to geographically distributed data centers. While the mechanism for using DNS for these purposes is well understood, the algorithms and policies involved in request routing and load balancing are still a subject of active research [4, 5, 11, 19, 3]. In fact, these algorithms and policies represent the "secret sauce" of various content delivery networks such as Akamai [2], SAVVIS [21], Limelight [13], Rapid Edge [20], and AT&T [10].

Unfortunately, the progress in this research area is hampered because DNS servers capable of request routing are either available as black-box appliances from network gear vendors [6, 8, 1, 22] or as complex systems with algorithms already built-in [16, 9]. Thus, researchers have to either modify complex software or base their research on simulation, which does not always produce reliable conclusions.

In this paper, we present the architecture and the preliminary evaluation of a DNS server that decouples request routing algorithms from the rest of DNS functionality. Our DNS server, which we refer to as *MyXDNS*[1], exports a well-defined interface that can be used to install an arbitrary request routing policy. Thus, researchers can implement their own algorithms and not be concerned with the rest of the extensive software comprising a full-featured DNS server. However, should they desire to modify the DNS server itself they can: we built it by modifying MyDNS (an existing open source DNS server implementation) [16] and made our sources available [17].

To demonstrate the flexibility of MyXDNS in implementing request routing policies, we coupled MyXDNS with an external off-the-shelf proximity service. We then used the proximity notion provided by this service to implement two existing request distribution algorithms. We then used these instantiations of MyXDNS to compare the performance of these algorithms in a real testbed rather than simulation.

The flexibility does not come without costs, and MyXDNS introduces some performance overhead compared to original MyDNS. However, our stress-testing shows that this over-

*This material is based upon work supported by the National Science Foundation under Grants No. CNS-0520105, CNS-0551603, and CNS-0615190. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The presentation of this work was made possible, in part, through financial support from School of Graduate Studies at Case Western Reserve University.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

[1]The name, pronounced "mix-DNS", stands for "My eXtensible DNS" and reflects the fact that our system is derived from MyDNS.

head is small, and MyXDNS achieves more than adequate performance. Our instantiation of MyXDNS running on a low-end server was able to sustain 4000 DNS queries per second with sub-millisecond response time in the absence of updates to the request routing policy, and 3000 queries/sec while processing hundreds of updates per second.

2. MYXDNS ARCHITECTURE

A high level architecture of MyXDNS is shown in Figure 1. MyXDNS comprises three major components: a DNS server, a proximity service, and the control process.

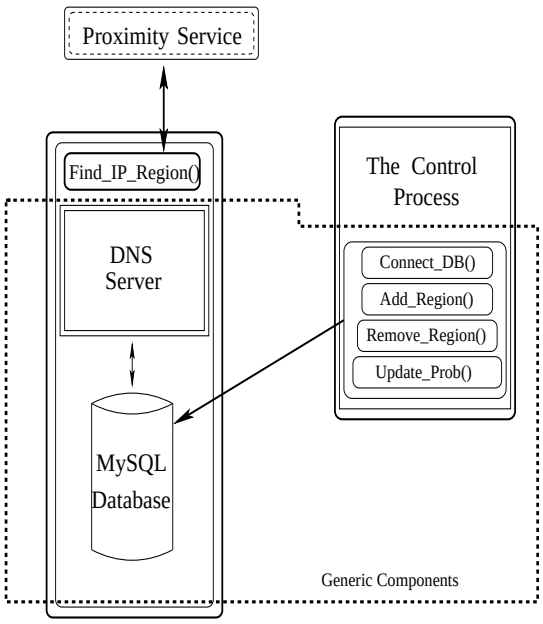

Figure 1: Architecture of MyXDNS.

The DNS server is responsible for receiving and answering external DNS queries. It concentrates the standard functionality of a traditional DNS server. In fact, it is by itself a fully functional load-balancing DNS server: in the absence of the other components it distributes requests among servers according to statically configured server selection probabilities (or uniformly if the probabilities are not supplied). The DNS server stores hostname-to-IP mappings in a database system. If a hostname maps to multiple servers, the database will contain multiple records for this hostname, one for each server.

The other two components are supplied by the user to implement customized server selection. The proximity service is used to classify requesters' IP addresses into user-defined *regions*, which provide the basis for the request routing policy. The regions are abstract classes of IP addresses and may use any metrics to differentiate requesters. Our instantiation of the proximity service uses geographical regions; other metrics such as network proximity or any other properties of the requesters are also possible. The control process is responsible for periodically recalculating the request routing policy according to a custom server selection algorithm, and for installing this policy at the DNS server. The control process may run on the same host as the DNS server or on a different host.

These three components interacts with each other through interfaces that generic MyXDNS provides.

To implement a custom server selection mechanism, users need to (a) provide their own `Find_IP_Region` function, which might interact with an external proximity service or it might compute the IP's region directly, and (b) implement their own control process, which periodically recomputes the request routing policy and locally calls functions `Add_Region`, `Remove_Region`, and `Update_Prob` to install this policy at the DNS server. These functions encapsulate communication details between the control process and DNS server; these details are transparent to the user since the functions are supplied by MyXDNS. (In fact these functions communicate directly with the database of the DNS server.) Note that because the proximity service and control process are provided by the user, they can in principle communicate with each other, in addition to the interactions discussed here. In fact, our MyXDNS instantiation includes such a communication channel through a shared configuration file (see Section 5).

We will discuss the MyXDNS interfaces in more detail in Section 3.

3. THE API

This section discusses the interactions between the generic components of MyXDNS and the components that implement custom server selection. We will also describe the general process of implementing a custom server selection algorithm.

3.1 IP Regions

The MyXDNS application programming interfaces (APIs) are based on the notion of *regions* - groups of requester IP addresses that are treated similarly by the system for the purpose of request routing. A user can use any properties of the requesters (depending on the system sophistication in what properties it can obtain) to define regions. For example, our instantiation of the proximity service defines regions based on the geographical location of the requesters. Other instantiations could use network proximity metrics, or non-proximity metrics such as the number of clients behind the requesters if this information is available (note that requesters that send DNS queries are client DNS servers).

Regions do not necessarily have to be classified as of geographic or network proximity metrics. A region is a class for which a group of IP addresses belongs to according to the user definition. In other words, `Regions` is a notion for classifying clients into predefined sets. The Classification could be based on any user defined criteria –depending on the data and the capabilities available– for the user.

The specific definition is abstracted by the proximity service through a function call `Find_IP_Region`, which takes an IP address and returns a region number.

Regions form the basis for the description of the *routing policy*, which is computed by the control process and uploaded to the DNS server. For a host name h mapping to a set of servers $s_1, \ldots, s_n$, the routing policy is expressed as a record $(h, (s_1, (P_{11}, TTL_{11}), \ldots, (P_{1m}, TTL_{1m}), (D_1, TTL_1)), \ldots, (s_n, (P_{n1}, TTL_{n1}) \ldots, (P_{nm}, TTL_{nm}) , (D_n, TTL_n)))$ where m is the number of regions, P_{ij} is the probability of using server s_i for a requester from region j, TTL_{ij} is the time-to-live of the corresponding DNS response, D_i is the default probability of using server i for a requester that can-

not be classified into any region (i.e., for whom no region is defined), and TTL_i is the TTL of the corresponding DNS response to such a requester. In the rest of this paper, we refer to value P_{ij} as *region probability* of server s_i.

3.2 Interfaces

MyXDNS interfaces include the following five function calls:

- Find_IP_Region().
- Connect_DB().
- Add_Region().
- Remove_Region().
- Update_Prob().

Function `Find_IP_Region` is a hook called by DNS Server for every DNS query to find the region of the query originator. This function takes the IP address as a string and returns the corresponding region ID as an integer, or zero if no region is defined for the IP address given. The region ID is then used to select the server for the request and to assign time-to-live to the resulting DNS response, according to the corresponding region probabilities. (See Section 3.1.)

Generic MyXDNS provides a stub implementation of function `Find_IP_Region` that always returns region 0, which means the default server probabilities will be used for server selection. In particular, if no control process exists to update the default probabilities, MyXDNS reduces to a simple load-balancing DNS server that selects servers for all requests, no matter where they come from, with a pre-defined probability (or selects servers randomly if no default probabilities are defined).

For performance reasons, `Find_IP_Region` in this studey is implemented as a module of the DNS server. This means that changing to a new proximity service requires MyXDNS to be recompiled. We have also implemented the proximity service as a separate process that interacts with MyXDNS through TCP connections. This new version of MyXDNS (available on [17] along with the original version) does not require the user to recompile MyXDNS to change to a new proximity service. MyXDNS now reads the proximity server IP from the configuration file at startup and communicates with the proximity service accordingly. We plan to explore performance implications of the new implementation of the proximity service as part of our future work.

The remaining four functions are called by the control process to communicate with the DNS server. Function `Connect_DB` initializes the connection of the control process with the database. It is invoked once at the start of the control process.

Function `Add_Region` defines a new region. It takes no arguments and returns an integer that is the ID of the newly created region. This function creates new columns in the database schema, so that the routing policy can include corresponding entries. (See Section4 for the database schema details). This function is typically called only at the DNS configuration time when regions are usually defined. However, the control process can use it to add a new region definition dynamically, or to update region definitions without bringing the DNS service down.

Function `Remove_Region` reverses the above action. It takes the region ID and removes that region's entries from the database schema. Function `Update_Prob` takes a new routing policy description (as defined in Section 3.1 and installs it into the DNS server. This function is called by the control process when it computes a new routing policy according to the server selection algorithm it implements.

3.3 Decoupled Server Selection

MyXDNS provides the user the flexibility of implementing his own server selection policy based on whatever information is available to him, without worrying about DNS server internals. Users may divide IP address space into arbitrary regions using their domain expertise about Internet topology, any policy considerations, or input from their network performance monitoring infrastructure. Users may change this regions definition dynamically, as the conditions and policies change, and may also leave portions (or all) of the IP address space unassigned to any regions.

Users can implement arbitrary server selection algorithms through uploading region-specific server selection probabilities and corresponding time-to-live values, which define how long the returned result will remain valid.

As an extreme example, the user may implement a pure proximity-based server selection as follows:

- Provide the proximity service (through a custom built `Find_IP_Region` function) that defines one region for each server so that each region groups IP addresses for which the corresponding server is the closest (according to the chosen proximity metric). This proximity service is the "magic source" reflecting the domain expertise of the operator.
- Provide the control process that assigns region probability of 100% for each server's own region and 0% for all other regions.

As the result, MyXDNS will always resolve a DNS query using the "closest" server to the query originator.

As another extreme, the user may implement purely load-based server selection as follows:

- Make `Find_IP_Region` function always return zero (use the generic MyXDNS stub).
- Provide the control process that periodically uploads default server selection probabilities based on measured server load.

The user can obviously also implement an algorithm that takes a mixture of these considerations in the account. We implemented two such algorithms that were previously described [3, 19]. The algorithm may also assign different TTLs to responses with different servers or returned to requesters from different regions (allowing for example the implementation of algorithms similar to [7]).

Finally, regions may be defined based on arbitrary considerations completely unrelated to proximity. For example, if a content delivery network knows (somehow) that certain client DNS servers are used mostly by dial-up users, the CDN can assign these servers to a separate region and use requests from these clients to balance the load of edge servers with no consideration for edge server proximity. Indeed, the dial-up clients will see little difference from using a nearby server anyway, so they can be used to even out the server load without reducing their expedience.

4. INTERNALS

This section describes the implementation of MyXDNS's DNS server in more detail. We begin with the description of the database schema and operation, and then present the DNS server details.

4.1 Database

A significant factor in our decision to use MyDNS as the base for our work is that it stores DNS mappings in a database management system. Dealing with DNS records through SQL queries eliminates conflicts that may occur when the Control Process and DNS server are simultaneously accessing the database.

MyDNS uses a database with two tables: Start of Authority records table (SOA) and Resource Records table (RR). The SOA table list DNS zones for which the current server is the authoritative DNS server; it is immaterial for request routing and is not discussed further here. The RR table contains DNS *resource records*, including hostname-to-IP mappings ("type A records") that concern request routing. For each hostname, the RR table includes one record for every server to which this hostname maps.

MyXDNS expands the schema for the RR table as shown in Figure 2. The top part is the original RR table schema. Field `name` lists the the hostname, while the `data` field contains the IP address of the corresponding server. The other two relevant fields, `ttl` and `aux`, although inherited from MyDNS, are used differently in MyXDNS. The `ttl` field specifies the *default TTL*, i.e., the time-to-live value MyXDNS will assign to its reply to a requester from an undefined region (i.e., for whom the `Find_IP_Region` function returns zero) when this server is selected. The `aux` field contains the default probability of selecting this server (among other servers with the same the hostname) if the requester comes from an undefined region.

MyXDNS also adds two new fields for every region i, named Ri and $TTLi$. These fields are created or removed dynamically by the `Add_Region` and `Remove_Region` functions. Ri stores the region probability for this server and $TTLi$ specifies the TTL to be returned to requesters from this region when this server is selected.

The TTL values in DNS responses determine the time period for requester to cache the response. Thus, MyXDNS can assign to responses the TTL that is specific for the chosen server and for the requester's region. For example, if requesters known to be used by a large number of Web clients are grouped together in a region, MyXDNS may implement a server selection algorithm that assigns low TTL to responses to these requesters. Or, when a hostname is mapped to heterogeneous servers, the routing policy may assign lower TTL to less powerful servers [4]. When not specified in the database, MyXDNS returns the default TTL value of 360 seconds.

4.2 DNS Server

The DNS server in MyXDNS is a modified version of MyDNS [16]. When MyXDNS receives a new request it tries to determine the requester's region by consulting the proximity service through `Find_IP_Region` . It then uses the region ID to select the server, among all servers with the given hostname, according to their region probabilities.

MyDNS uses internal caching to reduce the access rate to the database. A cache entry stores all the RR records matching a given hostname. To keep the size of cache entries small despite expanded database records, MyXDNS stores cached entries in the same format as MyDNS but replaces their `aux` values and default TTLs with the appropriate region probabilities and region-specific TTLs. Thus, cache entries in MyXDNS are different for different regions. Consequently, MyXDNS adds the region ID to cache keys that identify cached entries. Whenever the cache is searched for an entry, both the cached hostname and region ID must match the requested hostname and requester's region. Otherwise MyXDNS will issue a database query, add a new entry to the cache, and use this entry to compute its reply to the requester. This technique avoids spending cache space for information about regions from where requests rarely arrive.

Since DNS protocol allows responses to contain multiple answers, MyDNS includes all servers matching the hostname in its response. A load balancing function uses the `aux` value associated with each RR record to sort the answers in the response's addresses.The order of the answers determines the server that will be used by the client: the client uses the first operational server from the list.

In MyDNS, a low `aux` value increases the likelihood of the corresponding server to be placed high in the list. However, the exact relationship between the `aux` values and probabilities of a server being selected is difficult to express or compute.

Algorithm 1 Reply sorting function in MyXDNS

```
1: Input: Requester region j, requested hostname h
2: Output: Response_List
3: for Every server s_i corresponding to requested hostname do
4:    if j = 0 then
5:       Prob(s_i) = aux(s_i)
6:    else
7:       Prob(s_i) = R_j(s_i)
8:    end if
9: end for
10: Let S be the set of all servers corresponding to h
11: Let Response_List = ∅
12: while S ≠ ∅ do
13:    Let Total_Prob be the total probability of all servers in S
14:    if Total_Prob = 0 then
15:       Response_List = Concatenate(Response_List , RandomPermutation(S))
16:    else
17:       for Each server s_i ∈ S do
18:          Normalized_Prob(s_i) = Prob(s_i)/Total_Prob
19:       end for
20:       Select next_server according to Normalized_Prob(s_i)
21:       Concatenate(Response_List,next_server)
22:       S = S − { next_server }
23:    end if
24: end while
```

MyXDNS modified the reply sorting function so that the `aux` and the region probability values (`Rx`) directly specify probabilities of the corresponding server being selected. Specifically, the reply sorting function in MyXDNS follows the algorithm shown above. The function is invoked with the appropriate region probabilities or aux values for requesters

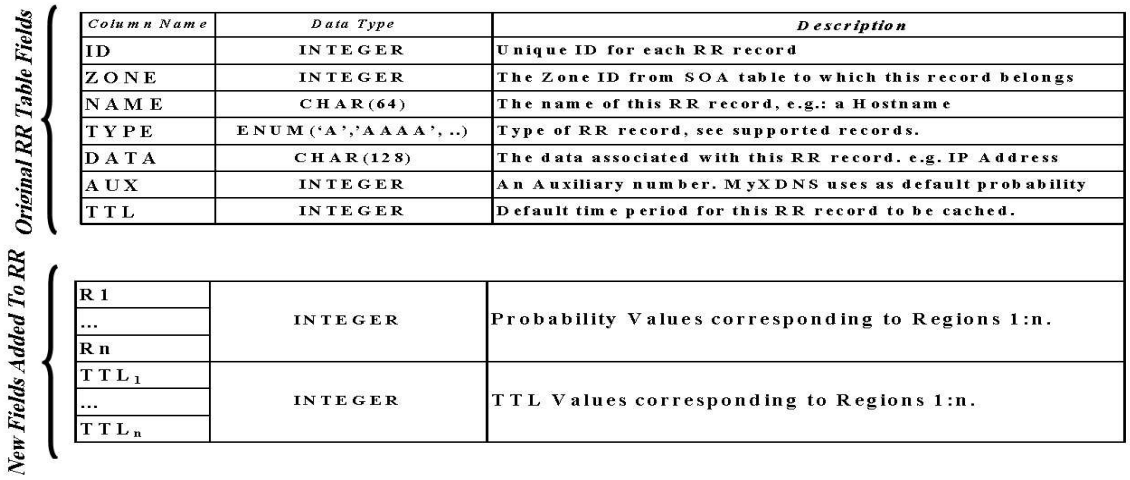

Figure 2: The RR table schema.

from an undefined region. It recursively puts the next server on the list according to its normalized region probability. If all probabilities for the servers that remain to be sorted are zero, they are all put into the list in random order.

5. INSTANTIATION

In this section, we describe an instantiation of MyXDNS that we implemented to demonstrate its benefits and study its performance. It is this instantiation that we tested in Section 6.2. While representing just one instance of the possible server selection mechanism, it itself allows fairly flexible variations of this mechanism as described below. From now on, we mean this instantiation when referring to MyXDNS.

5.1 External Proximity Service (GeoIP)

Our MyXDNS instantiation classifies IP addresses to regions according to their geographical location. It obtains the geographical location using GeoIP Country Lite database from MaxMind [15]. We used GeoIP to demonstrate the ability for MyXDNS to be coupled with an external off-the-shelf proximity service.

GeoIP provides geographical data including two-character country codes for any IP addresses. We grouped neighboring countries into eight regions, thus defining IP-to-region mapping for the resulting proximity service. Our proximity service defines these regions using a configuration file named `code-to-region`, which lists country codes and region IDs to which these countries belong. The user creates this configuration file manually. The control process reads this file at the startup and calls the `Add_Region` function for each region defined, thereby adding the corresponding Ri and $TTLi$ fields into the database schema (see figure 2). After each `Add_Region` call, the control process modifies the `code-to-region` file to ensure that the region IDs in this file match the region IDs generated by the `Add_Region` calls.

Each time a DNS request is received, the DNS Server calls `Find_IP_Region` function, which employs GeoIP APIs to determine the country code for the requester's IP address and then utilizes region definitions from the `code-to-region` file to extract the region ID for that country code. The DNS server then uses the region ID to load the appropriate region probabilities of the servers and select the server for the query right probabilities according to the current routing policy that has been installed by the control process.

5.2 The Control Process

Our instantiation of the control process periodically measures the load of the servers, recomputes the routing policy according to a given server selection algorithm, and installs this policy at the DNS server using the `Upload_Prob` function call. In our proof-of-concept instantiation, the control process uses the UNIX *uptime* command to measure server load. The control process is co-located with the DNS server on the same machine, and remotely executes the uptime command on each of the application servers. In real environments, the control process would likely use a large number of load metrics, including network consumption, memory utilization, open socket descriptors, disk I/O, etc., each with its own thresholds. To demonstrate the flexibility of MyXDNS, we implemented two existing request routing algorithms, one mentioned in [3] and the other proposed in [19]. Both algorithms try to take the proximity and load factors into account in their server selection.

Both algorithms use as input a table of IP region ranks for each server, which specifies the server proximity to the IP regions. Region ranks can vary between 1 (highest) and n (lowest), where n is the number of regions defined in the system. Region ranks for a server need not be unique: a server that is equidistant to multiple regions will have the same region rank for these regions.

Server Selection Algorithm I. The algorithm takes a load threshold as a parameter and produces the probability of selecting a given server for requests from each region. For a given region R, it executes the following steps [3]:

- Eliminate all servers that are overloaded.
- From the remaining servers, keep those servers who serve region best according to their ranking.
- For servers identified in step 2, distribute the load among them according to any available information about their capacity and load.

- In addition, we added an extra functionality to split the load among all servers if all of them were overloaded.

Server Selection Algorithm II This algorithm uses two load thresholds: high watermark, *HW* and low watermark, *LW*. Given a region, the algorithm assigns server selection probabilities in three passes over the servers as follows [19].

- The first pass assigns server weights based on load: servers with load above *HW* receive zero weight; servers below *LW* receive unity weight, and other servers receive a weight between zero and unity depending on where the server load falls between the high and low watermarks.
- The second pass adjusts the above weights to favor higher-ranked servers.
- The third pass normalizes these weights and converts them into selection probability by normalizing them to sum up to one.

6. PERFORMANCE OF MYXDNS

To evaluate the performance of MyXDNS, we performed stress-testing of generic MyXDNS, original MyDNS, and our instantiation of MyXDNS coupled with GeoIP-based external proximity service. We considered two metrics in our experiments: the percentage of successfully processed DNS requests (out of the total requests submitted) and the average response time for DNS queries. Both metrics were measured at the client side. Measuring the response time at the client side introduces a slight overestimation of the DNS processing time due to added network round-trip delay. However, this overestimation is very small because we used an isolated Gigabit Ethernet segment for our experiments, and it affects equally all three server configurations we compare.

6.1 Experiment Setup

Our test setup for stress-testing of MyXDNS included a DNS server and a client machine that emitted DNS queries with a given rate. To avoid possible effects of network congestion, we connected both machines as an isolated Gigabit Ethernet segment. Our DNS server is a low-end server (Sun Fire x2100 with Opteron 175 CPU and 2G memory) running Linux 2.4 kernel. The client machine is Dell Optiplex GX620 with 3.0GHz Pentium 4HT processor and 1 GB RAM. Both machines are equipped with Gigabit network cards and are interconnected via a Gigabit Ethernet switch.

In our experiments, the DNS server under study is configured to be an authoritative DNS server for zone `example.com`. The hostname `example.com` maps to six type `A` RR records in the database. To mimic requests coming from around the globe, our clients use raw sockets to generate DNS requests with source IP addresses randomly selected from 8500 different IP addresses. The 8500 IP addresses are chosen to be distributed among all regions we defined. There are around 1000 IP addresses for each region plus a few hundred addresses belonging to region zero, signifying clients whose region could not be determined (e.g., for which GeoIP returned NULL country code, or whose countries are not listed in our `code-to-region` file).

6.2 Performance with a Stable Request Routing Policy

Our first experiment investigates the performance of the DNS servers under study in the absence of updates to the routing policy. Figure 3 shows the average response time and Figure 4 shows query success rate for our MyXDNS instantiation and original MyDNS servers for different request rate, with and without internal caching at the DNS servers.

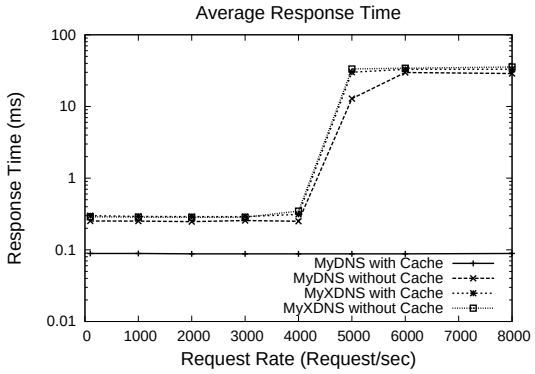

Figure 3: Average Response Time.

Figure 3 indicates that MyXDNS and MyDNS are capable of processing around 4000 request per second at a sub–millisecond response time. The difference between response times for MyXDNS and MyDNS at request rates less than 4000 is less than 0.2 milli-seconds when cache is disabled. With enabled cache, MyDNS's performance improves markedly: it processed any request rate we offered with no degradation in response time. At the same time, enabling caching did not appreciably improve the performance of our MyXDNS instantiation.

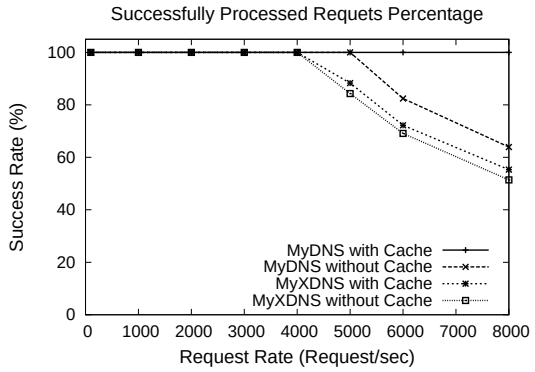

Figure 4: Percentage of Successfully Processed DNS Requests.

Figure 4 shows that MyXDNS instantiation managed to successfully process 100% of all incoming request up to the request's sending rate of 4000 request per second with or without internal cache enabled. However, increases beyond this rate lead to rapid degradation of the success rate. In fact the plato in response time above 5000 requests per second shown in Figure 3 is due to the fact that all additional

requests are simply dropped by the server. MyDNS exhibited around 25% higher capacity without the cache; however with the cache enabled, it kept processing 100% of the incoming requests even at 8000 request per second rate, the highest rate we offered.

We can expect that the extra functionality offered by the MyXDNS instantiation must cost some performance penalty, but we were intrigued by its virtual negation of benefits from caching. To investigate the reason for this behavior, we tested the performance of the generic MyXDNS server, which uses a stub `Find_IP_Region` function that always returns 0. This experiment showed that the generic MyXDNS server performance is virtually indistinguishable from MyDNS server performance in terms of both response time and success rate. In particular, MyXDNS with caching showed the same flat 100% request processing rate at all the loads we offered. We conclude that the performance difference between MyXDNS and MyDNS is due to the off-the-shelf external proximity service, GeoIP.

It is not the purpose of this research to justify the performance of an external proximity service. Depending on its implementation and functionality, it may cause lower or higher overhead than what we observed in our instantiation. However, we note that the 4000 requests per second at a sub-millisecond response time is more than adequate for most practical purposes. For instance, Wolman et al. report that they observed a peak Web request rate of about 300 requests/sec from a population of 23 thousand users [24]. Under the extremely conservative assumption that every Web request is preceded by a DNS query, our performance testing indicates that the MyXDNS instantiation can serve 300,000 users with sub-millisecond delay. And if we assume (still conservatively) a TTL of 30s for DNS responses, this user population increases 30-fold, to roughly 9 million. Scaling beyond this population requires a distributed two-level DNS platform similar to the one used by Akamai [2].

6.3 Performance in the Presence of Routing Policy Updates

We now turn to the effects of routing policy updates on MyXDNS performance. Note that internal caching in the DNS server would delay the effect of the policy update until the cached entry expires. Thus, to see the effect of extreme update rates on MyXDNS performance, we desabled the cache for these experiments[2]. In practice, one can expect MyXDNS to execute with enabled cache, resulting in higher performance than the performance observed in this section.

Figure 5 shows both the response time and success rate of our MyXDNS instantiation for various request rates in the presence of one routing policy update per second. Because internal cache would virtually negate the effect of the policy update, we disabled the cache for these experiments. We must mention here that caching was only disabled for this experiment and not for general operation.

According to Figure 5, MyXDNS managed to process all incoming requests up to the rate of 3000 req/sec. At this rate MyXDNS processing time is still below 1ms. Beyond this

[2]Enabling the cache would negate the effect of policy updates because the DNS server would not access the database until the cached entries expire. The default expiration time of cache entries in MyDNS is 60 seconds, much higher than our update rate.

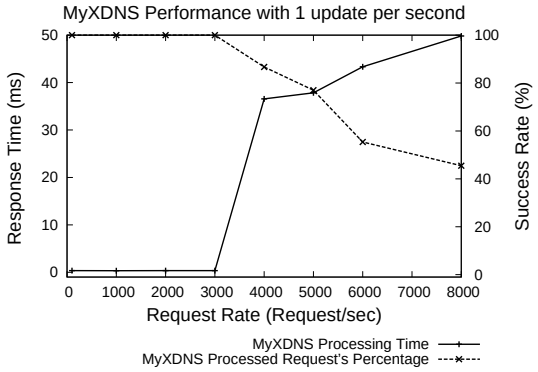

Figure 5: MyXDNS Performance for one update per second.

point, MyXDNS response time jumps rapidly passing the 30ms level for 4000 req/sec. The percentage of successfully processed requests also starts degrading beyond the 3000 req/sec rate. We repeated this experiment for the generic MyXDNS, and its performance was indistinguishable from the MyXDNS instantiation for request rates of up to 3000 req/sec. We conclude that in the presence of one update per second, the MyXDNS capacity becomes 3000 req/sec. Thus, while decreasing from the 4000 req/sec capacity for a stable policy, the MyXDNS capacity remains respectable. We will use this rate for our next experiment, studying how MyXDNS capacity is affected by an increasing update rate.

In Figure 6, we investigated the effects of different update rates on the response time at request rate of 3000 request per second. Again, caching is disabled in this experiment and both MyXDNS instantiation and the generic MyXDNS were tested.

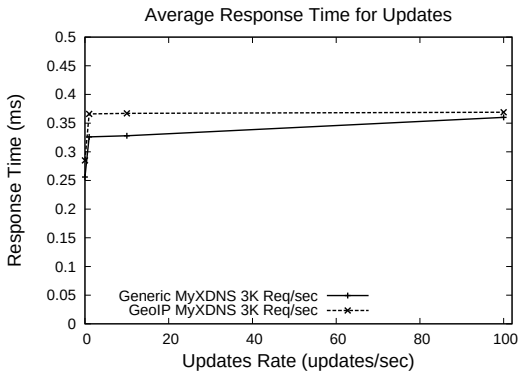

Figure 6: Response Time for Different Updates Rates.

Figure 6 shows growing update rate results in a very small increase in the response time: the difference in response time is less than 0.1ms for up to 100 updates/sec. We also tested the percentage of successfully processed requests for the generic MyXDNS and MyXDNS instantiation under the same workloads and found that both servers processed 100% of the received requests in all experiments. Obviously, the request routing policy cannot be expected to change nearly

as often as 100 updates/sec. Thus, we can confidently assume the 3000 req/sec capacity for MyXDNS under changing request routing policies.

7. PERFORMANCE OF SERVER SELECTION ALGORITHMS

Finally, we perform a preliminary experiment with the two server selection algorithms we implemented in our instantiation. This performance study is by no means intended to be complete; it is merely meant to show the potential of MyXDNS as a research tool. In contrast to [19] which studies these two algorithms using simulations, MyXDNS allows us to compare them using a realistic testbed.

7.1 Experiment Setup

For this experiment, we configured an isolated Gigabit Ethernet segment comprising a MyXDNS server, two Tomcat application servers and one client workload generator that sends requests for a servlet hosted by both Tomcat servers. We used the same Sun Fire server for MyXDNS and Dell PowerEdge 1950 servers with dual Intel Xeon 5140 2.33GHz CPUs and 4G RAM for both the application servers and the workload generator.

The workload generator is implemented to mimic clusters of browser machines behind local DNS servers. Each browser cluster issues one DNS query to MyXDNS every TTL period, and then the associated browsers generate HTTP requests to the returned IP address until the response to the DNS query expires. To add realistic randomization to load balancing, we staggered the starting times of different clusters, so that they will send their DNS queries at different times.

MyXDNS is configured to direct client requests to Tomcat server 1 or server 2 according to the routing policy computed by the server selection algorithm. All client clusters are assumed to belong to one region, and the server ranks for this region are configured to be 1 for server 1 and 2 for server 2. In other words, server 1 is closer to all the clients than server 2. We used a dummy servlet that simply executes a certain number of floating-point multiplications. In this experiment, the workload generator was configured to emulate 20 browser clusters, the DNS TTL is 30 seconds, and each cluster issues between 1 and 3 HTTP requests per second. The policy recomputation occurs roughly every 10 seconds (this interval varies slightly because it is affected by the response time of sometimes overloaded Tomcat servers to MyXDNS's load probes). The load threshold for algorithm 1 is set to 0.5; for algorithm 2, the high-watermark is set to 0.6 and low watermark to 0.4. (Recall our MyXDNS instantiation measures server load as the output of the uptime command.)

7.2 Results

The results are shown in Figure 7 for algorithm 1 and Figure 8 for algorithm 2. They show that, at least in this experiment, both algorithms result in constantly oscillating routing policies. The more complex algorithm 2 exhibits oscillations that seem as severe as those produced by the simpler algorithm 1. However, algorithm 1 results in virtually no overall preference for the closest server (as indicated by the close load curves in the bottom graph of Figure 7). Further, neither algorithm shows distinct preference for the closest

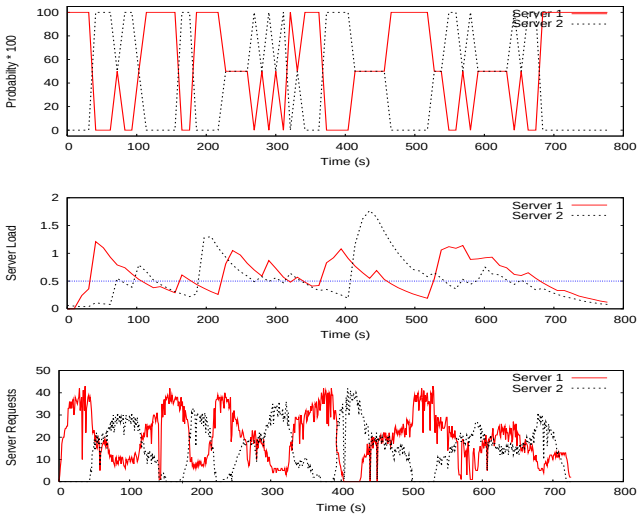

Figure 7: Performance of Algorithm I.

server in its request routing behavior. These results differ markedly from the conclusions in the simulation study from [19]. While our current preliminary experiment is clearly far from being representative and implements an admittedly artificial scenario, it still shows a concrete example that deviated from the behavior predicted by the simulation study.

8. RELATED WORK

Cardellini et al. [5] present a taxonomy of request distributed algorithms and discuss performance trade offs between them. They focus in particular on the interaction of DNS-based global load balancing and what they refer to as second-level dispatching at data centers.

NCSA Web site was among the first to utilize DNS load balancing [11, 12], which they implemented using a simple round-robin algorithm. Several studies examined the effectiveness of DNS-based request routing [23, 14, 18].

A number of algorithms for DNS-based server selection has been proposed [4, 7, 5, 11, 19]. MyXDNS will lower the barrier for studying these and other algorithms and metrics.

Several publicly available DNS implementations, such as MyDNS [16] and SuperSparrow [9], offer load balancing capability but they do not offer APIs to plug in one's own server selection algorithm. In particular, MyDNS allows servers to be selected with pre-assigned probability, and SuperSparrow utilizes BGP routing information to select the closest server.

MyXDNS stems from our previous work on IDNS, a specialized request-routing DNS server [3]. Unlike IDNS, our MyXDNS is a general DNS server and exposes well-defined APIs allowing it to be coupled with a custom proximity service and server selection.

9. CONCLUSIONS

In this paper we presented the architecture and the preliminary evaluation of a request routing DNS server that decouples server selection algorithms from the rest of DNS functionality. Our DNS server, which we refer to as MyXDNS, exposes well-defined APIs for uploading an externally com-

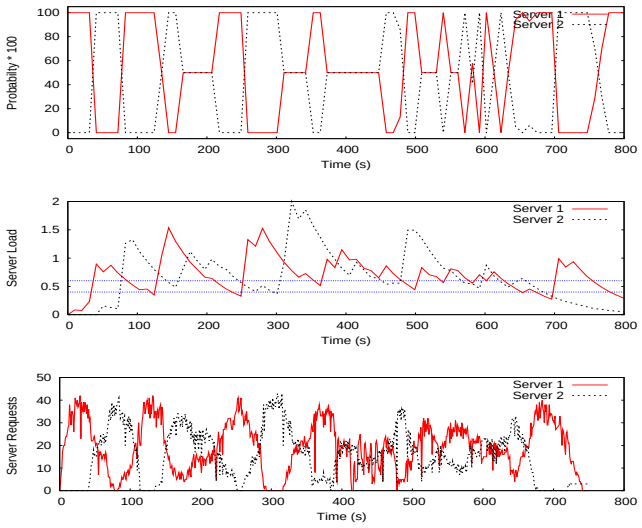

Figure 8: Performance of Algorithm II.

puted server selection policy and for interacting with an external network proximity service. With MyXDNS, researchers can explore their own network proximity metrics and request routing algorithms without having to worry about DNS internals. Furthermore, MyXDNS is based on open-source MyDNS and is available to public.

To demonstrate the flexibility of MyXDNS, we implemented an instance of a request routing DNS, which uses an external off-the-shelf proximity service, GeoIP, to determine the proximity of requesters to the servers. We then used this externally provided notion of proximity as the basis for implementing two previously described server selection algorithms. Besides demonstrating MyXDNS flexibility, this allowed us to compare the performance of these two algorithms in a real testbed as opposed to simulations. Our preliminary stress-testing of MyXDNS achieved over 3000 resolutions/sec with sub-millisecond response on a low-end server. This is more than adequate for a DNS server: using independently observed peak request rates from Web clients [24], we can estimate that a single MyXDNS server we tested can serve over six million Web users.

10. REFERENCES

[1] 3-DNS controller: Make the most of your network. F5 Networks. http://www.f5networks.de/f5products/3dns/.

[2] Akamai Technologies. http://www.akamai.com/html/technology/index.html.

[3] A. Biliris, C. Cranor, F. Douglis, M. Rabinovich, S. Sibal, O. Spatscheck, and W. Sturm. CDN Brokering. In *6th Int. Workshop on Web Caching and Content Distribution*, June 2001.

[4] V. Cardellini, M. Colajanni, and P. S. Yu. Redirection algorithms for load sharing in distributed web-server systems. In *ICDCS*, pages 528–535, 1999.

[5] V. Cardellini, M. Colajanni, and P. S. Yu. Request redirection algorithms for distributed web systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):355–368, 2003.

[6] Cisco GSS 4400 series global site selector appliances. http://www.cisco.com/en/US/products/hw/contnetw/ps4162/index.html.

[7] M. Colajanni, P. S. Yu, and V. Cardellini. Dynamic load balancing in geographically distributed heterogeneous web servers. In *ICDCS*, pages 295–302, 1998.

[8] Enhancing web user experience with global server load balancing. white paper. Alteon WebSystems. http://www.nortel.com/products/library/collateral/intel_int/gslb_wp.pdf.

[9] S. Horman. Globally distributed content (using BGP to take over the world). http://www.supersparrow.org/ss_paper/html/ss_paper.html, 2001.

[10] Intelligent content distribution service. AT&T Inc. http://www.business.att.com/service_fam_overview.jsp?repoid=ProductSub-Category&repoitem=eb_intelligent_content_distribution&serv_port=eb_hosting_storage_and_it&serv_fam=eb_intelligent_content_distribution&segment=ent_biz.

[11] E. D. Katz, M. Butler, and R. McGrath. A scalable http server: The ncsa prototype. *Computer Networks and ISDN Systems*, 27(2):155–164, 1994.

[12] T. T. Kwan, R. McCrath, and D. A. Reed. NCSA's world wide web server: Design and performance. *IEEE Computer*, 28(11):68–74, 1995.

[13] http://www.limelightnetworks.com/contentdelivery.html.

[14] Z. M. Mao, C. D. Cranor, F. Douglis, M. Rabinovich, O. Spatscheck, and J. Wang. A precise and efficient evaluation of the proximity between web clients and their local dns servers. In *USENIX Annual Technical Conference, General Track*, pages 229–242, 2002.

[15] Maxmind - IP geolocation and online fraud prevention. `http://www.maxmind.com/`.

[16] Mydns. `http://mydns.bboy.net/`.

[17] http://eecs.case.edu/ misha/MyXDNS.html.

[18] J. Pang, A. Akella, A. Shaikh, B. Krishnamurthy, and S. Seshan. On the responsiveness of dns-based network control. In *Internet Measurement Conference*, pages 21–26, 2004.

[19] M. Rabinovich, Z. Xiao, and A. Aggarwal. Computing on the edge: A platform for replicating Internet applications. In *the Eighth International Workshop on Web Content Caching and Distribution*, Sept. 2003.

[20] http://www.rapidedgecdn.com/.

[21] http://www.savvis.net/corp/Products+Services/Digital+Content+Services/Content+Delivery+Services/.

[22] ServerIron DNSProxy. Fountry Networks. http://www.foundrynet.com/products/appswitch/app-switch-appl/app-global-load-bal.html.

[23] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of dns-based server selection. In *INFOCOM*, pages 1801–1810, 2001.

[24] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of Web-object sharing and caching. In *Usenix Symposium on Internet Technologies and Systems*, Oct. 1999.