NetAirt: A Flexible Redirection System for Apache

Michał Szymaniak Guillaume Pierre Maarten van Steen

Department of Computer Science
Vrije Universiteit Amsterdam
{*michal,gpierre,steen*}*@cs.vu.nl*

Abstract: In this paper, we present NetAirt – a redirection system that separates redirection mechanisms from redirection policies. NetAirt is implemented in the form of a module for the Apache HTTP server. We propose a few changes to the original Apache source code that enable Apache to service UDP datagrams, and show how we exploit these changes in the NetAirt implementation. Performance measurements prove that the overhead introduced by the redirection mechanisms is low compared to the overall name resolution time.

Keywords: Redirection, DNS, Apache, Autonomous Systems

1 Introduction

With the growing popularity of the Internet, more and more Web sites find it difficult to provide their clients with satisfactory quality of service. A common solution to this problem is to replicate the site content over several machines at specific locations and to redirect clients to a nearby replica. The proximity of the requested replicas can reduce request Round Trip Times (RTTs). Since data transfer constitutes about 80% of the total RTT of a typical Web request, users are likely to observe better access performance [17]. This also leads to load distribution, which may result in faster client request processing. Systems combining content replication and client redirection are usually referred to as Content Delivery Networks (CDNs).

A vital part of every Content Delivery Network is its redirection system. Redirection can be achieved by one of the following three approaches [3]. The first one is to use HTTP-based mechanisms, which redirect client HTTP requests either by rewriting URLs inside Web documents or by using a special HTTP redirection response code. Although HTTP-based mechanisms are easy to implement, they make the redirection visible to the clients, which is often considered undesirable. The second approach utilizes IP-based mechanisms in which the redirector forwards incoming network packets to replicas. IP-based mechanisms are transparent to clients, but they are also complex and do not scale well over wide-area networks. The third approach is to use DNS-based mechanisms in which a custom DNS server responds to name resolution queries with the IP address of a replica rather than with that of the originally named host. DNS-based mechanisms are simple, transparent, and scalable. Because of these advantages, most CDNs use DNS-based redirectors.

A redirection mechanism operates according to a redirection policy. A policy is the part of the redirector that selects which replica a given client should be redirected to. This selection may take various criteria into account, such as the replica-to-client distance and the load of the available replicas. A redirection mechanism and a redirection policy together form a redirection system.

This paper presents NetAirt, a redirection system that separates the implementations of redirection mechanisms and policies. This approach allows for adding new policies, or using existing ones with another redirection mechanism.

We designed NetAirt as a module for the Apache HTTP server to facilitate its integration into Globule, an Apache-based CDN that our group is developing [13]. The advantage of this approach is that it allows NetAirt to cooperate tightly with other Globule components. For example, each time Globule decides to change the placement of its replicas, it must inform NetAirt so that the client got redirected to the new replicas. Since the component responsible for replica placement is just another Apache module, a single function call is enough to bring NetAirt up-to-date after some replica is created or destroyed. Similarly, NetAirt may notice that some replicas are over-

loaded and provide Globule with hints on where new replicas would be most needed.

NetAirt currently supports two redirection mechanisms, HTTP and DNS, and three policies. HTTP redirection mechanisms are trivial to implement in Apache. As for DNS, however, we had to extend Apache by basic UDP server functions. Integration of both HTTP- and DNS-based mechanisms makes NetAirt attractive for more users, regardless of whether they are able to buy and host a DNS domain or not. Also, it allows one to develop redirection policies that use both these mechanisms simultaneously, or switch between them, depending on the circumstances. At the moment, the implemented policies are: *static name-to-address mapping*, *round-robin selection*, and a more complex one, *the shortest AS-path policy*.

We conducted performance experiments which show that our Apache-based DNS implementation is about 2 times faster than the popular Bind DNS server. The shortest AS-path policy introduces an additional overhead of 640 μs on average, which is little in comparison to the overall DNS request-reply delay.

The paper is structured as follows. Section 2 surveys related work. Section 3 presents the design of a DNS-based redirection system. Section 4 describes the implementation of NetAirt as a module for Apache. Finally, Section 5 discusses our performance measurements, and Section 6 concludes.

2 Related Work

The idea of exploiting DNS for client redirection emerged in 1995, yet originally only for load balancing [4]. In essence, a DNS server can be modified to respond to DNS queries concerning a given domain name with IP addresses of different machines. The only constraint is that any of these machines can service client requests targeting the service. For example, a query regarding `www.globule.org` is usually resolved as follows. First, it reaches an (unmodified) DNS server responsible for the `.org` domain, and then it is sent to the (modified) DNS server hosting `.globule.org`, which may return different addresses of `www.globule.org`. An important advantage of DNS redirection is that it requires modifying only the lowest-level DNS server, i.e., the one uniquely associated with the service domain, and leaves all other DNS servers untouched. This is because all DNS responses regarding the service originate from the service DNS server [11].

Redirection decisions are based on a short-lasting view of system conditions. To prevent such decisions from remaining in use for too long, small time-to-live (TTL) values are attached to redirecting DNS responses. A potential problem is that assigning too low TTLs may lead to performance degradation of name resolution. However, it has been shown that reducing TTL values to about 10 minutes does not significantly degrade DNS cache hit rates, while giving the redirecting server sufficient control over redirection [8].

In a wide-area replicated system, the choice of replica can depend on a client's location, so that the client is redirected to a nearby replica. However, DNS queries do not carry any information about the querying client. A redirecting DNS server must therefore approximate the client's location with that of the DNS server that resolves the name on behalf of the client. The assumption here is that clients are located close to their local DNS server. This is generally true, since about 64% of clients are located in the same Autonomous System as their DNS server [9]. An Autonomous System (AS) is a collection of networks that share the same, clearly defined set of rules concerning the exchange of traffic with other ASes [7]. Each AS is connected to some other (one or more) ASes. All ASes together cover the entire Internet.

A redirection policy may redirect clients to their closest replicas according to a certain distance metric. For example, the shortest AS-path policy is based on a graph, where vertices denote Autonomous Systems and edges reflect inter-AS traffic-exchange relationships [10]. The distance between two arbitrary machines is defined as the number of edges in the shortest path between the ASes to which these two machines belong. The graph can be derived from routing tables available at routers running BGP – the inter-AS routing protocol [14].

The scalability of DNS-based redirection can be improved by using what is known as *two-tier* DNS redirection, in which several redirecting DNS servers are deployed in different parts of the Internet [6]. A client of a specific server is periodically redirected to a closest redirecting DNS server by a main redirecting DNS server, and is instructed to use that server for a certain time (e.g., 1 hour). In this way, the load originally handled by the main DNS server is distributed across several *client-proximate* DNS servers, which are likely to service the redirection requests faster.

In the following sections, we discuss how the above ideas can be combined inside a single redirection system. We then show how to implement this system as a module for the Apache HTTP server.

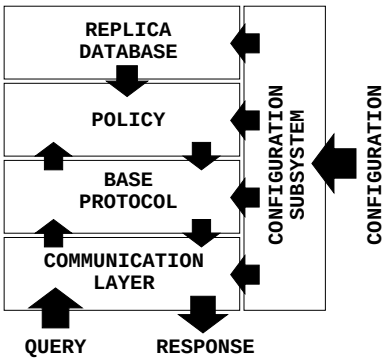

Figure 1: The design of NetAirt

3 System Design

As shown in Figure 1, NetAirt consists of five main components: a replica database, a policy, a base protocol, a communication layer, and a configuration subsystem. The split into the four leading components reflects different phases of handling a redirection request. The configuration subsystem was added to make NetAirt consistent with Apache, which has a separate configuration subsystem as well.

The replica database stores data about the replicas. These include the domain names of replicated services, replica server IP addresses, and TTL values.

The policy component receives invocations containing a service name and executes a service-dependent replica selection algorithm to return one or more replica IP addresses to which the client should be redirected. This algorithm uses the data stored inside the replica database, but may also exploit any policy-specific data stored inside the policy component.

The base protocol layer is responsible for decoding service names from client requests (DNS or HTTP), and passing these names to the redirection policy. The latter returns one or more replica IP addresses, which are subsequently encoded into a protocol-specific response. The possibility of using both DNS- and HTTP-based redirection makes NetAirt more flexible.

The communication layer receives client requests and passes them to the base protocol implementation. The responses returned by the latter are sent back to their respective clients. To service DNS, this layer must support both UDP and TCP.

The configuration subsystem allows for adjusting different settings of the previous four components, such as the contents of the replica database and the number of replica addresses returned. In addition, it can control some policy-specific settings.

The separation of the redirection mechanism (by means of the base protocol layer) and the redirection policies makes it possible to build complex redirection systems. For example, NetAirt can use the same mechanism to service several domain names, each configured to use a separate policy. More importantly, NetAirt can be used to implement the two-tier DNS redirection discussed in Section 2, where the main redirector uses different policies than the redirectors scattered across the Internet.

4 System Implementation

We start by explaining how we built NetAirt as an Apache module for doing HTTP redirection. Then we describe which modifications had to be made to Apache to allow it to service DNS requests as well.

4.1 HTTP Redirection

Apache splits the servicing of HTTP requests into three steps: accepting a connection, processing the request (including response generation), and closing the connection [1]. The processing step is further divided into several HTTP-specific phases.

Each phase is handled by so-called hook functions. Whenever a given phase is reached, the corresponding hook functions are called. Hook functions are provided by modules, which can be a part of the standard Apache distribution, or can be provided by third parties. This allows one to easily extend the functionality of Apache.

An HTTP redirector can easily be implemented as an Apache module. This module provides one hook function that generates redirecting HTTP responses. The behavior of the hook function follows the request processing model described in Section 3. It extracts the service host name encoded inside the URL, and invokes the redirection policy to find a replica IP address for that name. It then prepares a new URL, in which the service host name is replaced with the replica IP address, and packs this URL into an HTTP response that carries the “302 Temporarily moved” HTTP response code. Upon retrieval of this response, the client is expected to follow the new URL and access the corresponding replica.

4.2 DNS Redirection

Supporting DNS redirection requires Apache to service DNS queries that come, for example, from resolving `www.globule.org` as we explained before. These queries can be carried by either TCP or UDP [12]. TCP-carried DNS queries can be handled similarly to HTTP requests – by defining a hook function that overtakes the entire connection. The only difference is that the function must now decode *DNS requests* to identify target service names, and pack the replica IP addresses into *DNS responses*.

Servicing UDP-carried DNS queries is more complicated because Apache was originally designed as a TCP server. Supporting UDP requests required modifying the Apache code. However, the required modifications are relatively small.

Apache encapsulates listening sockets into so-called "listener" structures, which contain a network socket descriptor and a pointer to a custom "accept" function. When data has arrived to one of the listening sockets, Apache calls the associated "accept" function to establish the connection, and then passes the connection to the standard connection-processing subsystem. All "accept" function calls are serialized with a mutex in order to avoid several processes to try to simultaneously accept the same connection.

Usually, the socket descriptor is that of a TCP socket, but it can easily be replaced by a UDP socket descriptor. In UDP, however, connections do not need to be *accepted*, so the corresponding listener must have its "accept" function set to a void function. Unfortunately, this modification is not sufficient, as the standard request-processing subsystem implicitly considers the socket descriptor to use TCP. For example, it operates on this socket using TCP-specific functions, such as `read()` and `write()`, whereas for UDP `recvfrom()` and `sendto()` should be used. We solved that problem by adding another function pointer to the listener. This pointer indicates a custom request-processing function that should be called *instead* of the standard one. The pointer remains unused in TCP listeners, for which the standard Apache request-processing subsystem is still invoked.

With this small modification to the Apache listener structure, a UDP server such as a DNS redirector can easily be implemented as an Apache module. A special hook function creates a UDP listener at server startup and adds it to the server-wide collection of listeners. The remaining parts of request processing are essentially identical to that of its TCP-based counterpart.

4.3 Redirection Policies

NetAirt currently supports three redirection policies. The first policy implements static name-to-address mapping. It always returns the same IP addresses for a given service name and enables NetAirt to behave like a simple DNS server. The second policy implements round-robin address selection, which allows for simple load balancing. Since the implementation of these two policies is quite straightforward, we do not discuss it in this paper.

The third policy is the shortest AS-path policy described in Section 2. It assumes physical distribution of replicas, and aims at improving the client-perceived service performance by exploiting replica locality.

The implementation of the shortest AS-path policy must fulfill two tasks. Firstly, it must match replica and client IP addresses to their respective ASes. Secondly, it must build a graph of ASes based on which distance calculations can be performed. The necessary AS-related data are derived from BGP routing tables.

4.3.1 IP-to-AS Matching

A BGP routing table contains a list of entries. Each entry describes a single route to the set of machines whose addresses fall within a given IP address range, such as the one below:

```
10.2.3.0/24 ... 64003 64001 64112 i
```

This entry means that the route to any machine which has "10.2.3" as the first 24 bits of its IP address passes through ASes 64003, 64001, and 64112. The triple dots denote other available information, which is irrelevant here. The list of AS numbers is called an *AS-sequence*. The trailing character, called "origin," denotes the information source: "i" means that the route was learned using an Internal Gateway Protocol. In other words, the route ends in the last AS in the AS-sequence.

BGP route descriptions allow one to match IP addresses to their "home" AS. Given an IP address, we first determine which route leads to this address. We then consider the last AS in the corresponding AS-sequence as the home AS for the IP address.

To speedup the search of relevant routes, NetAirt organizes all routes into a prefix-based tree and then applies a classical longest-prefix search algorithm [16]. As a result, mapping an IP address to its home AS takes on average less than 1 microsecond when running on a PIII/1.1GHz.

Treating the last AS in an AS-sequence as the home AS is not always correct. For example, BGP tables

can contain entries that represent a set of aggregated routes. In this case, the AS-sequence is followed by an *unordered* set of ASes, through which the traffic *can* go further. Since we have no certainty concerning which element from the AS-set is actually at the end of the route, we cannot accurately determine the home AS. Instead, we can consider the last AS in the AS-sequence only as the best available approximation. According to our tests, AS-sets occur in less than 1% of all routes in a typical BGP table.

Another potential problem is posed by incomplete route descriptions, which inform that at some point routers switch from BGP to another routing protocol. Such entries have a special "origin" mark: "?" instead of "i." Like in the previous case, the last AS in an AS-sequence is only the best available approximation of the home AS. Moreover, switching to another routing protocol also means switching to another network distance metric. Since our aim is to express network distance in inter-AS hops, ignoring this new metric does not influence our distance calculations. We noticed that incomplete descriptions constitute about 12% of the BGP table we used.

4.3.2 Building a Map of ASes

BGP route descriptions also allow one to build a map of the Internet in the form of a graph of ASes. NetAirt links two ASes if they are adjacent in at least one AS-sequence. We assume here that inter-AS traffic exchange is bidirectional.

A potential problem is that one BGP routing table can provide a view of the Internet topology as seen only from a given router. This view contains routes only from this router to other ASes, and thus misses a lot of transversal links. NetAirt solves this problem by exploiting a global routing table, available from the RouteViews project [2]. This table merges several views of the Internet, contributed by routers located in different parts of the world. It therefore contains many more links than a normal routing table.

4.3.3 Policy Implementation

Given an IP-to-AS translation tree and an AS graph, the shortest AS-path policy operates as follows. For each redirection request, it starts with translating the IP addresses of the client and every candidate replica to their home ASes. Then, it applies a classical Breadth-Search-First (BSF) algorithm to the AS-graph, and traverses the graph starting from the client's home AS. While traversing, it builds a list containing the visited home ASes of the candidate replicas. The search terminates once the required number of replicas has been found, or when the entire graph has been traversed. Finally, the policy returns the just-built list of candidate replicas as its redirection recommendation.

NetAirt keeps the AS-lookup tree and the AS graph in a shared memory chunk to make them available to all the Apache processes. Two such chunks are used alternatively: when one stores the currently used data structures, the other one is available for their new version to be generated when a new version of the BGP routing table is downloaded. In this way, one version of the data structures is available at any time, which enables the policy to work continuously, even while a new data version is being constructed (which may take up to 2 minutes on a PIII/1.1GHz).

A possible optimization could be to precompute responses that would be returned to clients located in each AS. A potential problem, however, is that in such approach the responses would have to be recomputed every time a replica is created or destroyed. Therefore, whether it is beneficial to precompute them depends on the replica placement dynamics. We plan to investigate this optimization in the future.

5 Performance Measurements

We measured the performance of NetAirt by conducting two experiments. First, we measured the performance of the DNS transport layer alone, and compared it to the performance both of Bind 9.1.3 and of (unmodified) Apache 2.0.32. Second, we performed micro-benchmarks on the shortest AS-path policy. Both experiments were performed on a standard PC, equipped with one PIII/1.1GHz CPU.

5.1 DNS Transport Layer

In this experiment, we evaluated how much time our redirector needs to service a typical DNS request. We conveyed separate tests for UDP- and TCP-based DNS queries. For each test we used the two simplest (thus fastest) policies: the static address mapping policy and the round-robin policy. We ran the tests on four redirector configurations, each returning a different number of addresses to the client.

We measured the average Round Trip Time (RTT) for 1000 DNS queries. To avoid network-related overhead, the client and the redirector were located on the same machine. Also, to avoid concurrency-related inaccuracies, we configured the redirector to use only

Transport Layer and Policy	No. of Addresses Returned			
	1	2	3	4
TCP Static	588	595	595	599
TCP Round-robin	590	596	596	600
UDP Static	387	391	392	395
UDP Round-robin	393	400	400	402

Table 1: RTTs for different policies (in μs)

one request-processing thread. The results of these experiments are summarized in Table 1.

As can be observed, TCP-oriented queries are about 50% slower than UDP-oriented ones. We believe that this difference is caused only by the cost of establishing TCP connections.

The overhead introduced by the round-robin policy, compared to the static address mapping policy, turns out to be negligible (about 2% of the total RTT). Also, returning different numbers of addresses (between 1 and 4) does not significantly influence the RTT.

We also compared the efficiency of our Apache-based DNS server with that of the popular Bind DNS server [5]. Similar to the TCP-vs-UDP comparison, we placed the client and the server on the same machine and measured the average RTT for 1000 DNS queries. It turned out to equal 821 μs for UDP-based queries, and 1514 μs for TCP-based ones. These values indicate that our DNS implementation services a typical DNS query over 2 times faster than Bind. A possible interpretation of these results is that Bind supports many advanced DNS features, whereas our implementation deals only with the classical queries. Consequently, the DNS request-processing code of NetAirt is much simpler.

Finally, we compared the performance of DNS queries versus HTTP queries. We measured the overall latency of downloading a 0-byte long HTML file using the "ApacheBench" utility. Similarly to the above experiments, the server was configured to use a single request-processing thread, and both the client and the server were placed on the same machine. The average latency turned out to be 588 μs per request, which comes close to the latency of TCP-oriented DNS requests. This shows that handling simple DNS queries in Apache can be just as efficient as processing HTTP requests.

5.2 The Shortest AS-path Policy

In this experiment, we investigated the internal latency of the shortest AS-path policy. To measure only the delay caused by the AS-related processing (and not by Apache), we removed the redirection policy function from NetAirt, and embedded it into a single-threaded C program. We measured the time needed to return a list of replica addresses, excluding costs due to the transport layer.

We performed the following experiment. First, we selected an arbitrary AS where a simulated client was located. Then, we placed one simulated replica in every AS, and instructed the redirector to treat each of these replicas as the only replica of a certain site. In other words, we put a non-replicated site in every AS. Finally, for each site, we measured the time needed by the policy to prepare a redirecting response for the client.

The results turned out to be independent of the client's location. Figure 2a shows the distribution of search times for replicas located at increasing distances from the client. When a replica is very close to the client, the BSF search algorithm inside the policy explores only a few AS-graph nodes before finding it, so the search is very fast. When the distance grows, the number of nodes to explore dramatically increases, leading to longer search. Finally, only very few nodes are located far away from the client. Searches for these nodes correspond to a nearly full exploration of the AS graph. This interpretation is supported by Figure 2b, which shows that most nodes are indeed located at distances 3 to 5 from the clients.

Knowing the search times, we can derive the time needed to locate the n replicas closest to a given client: this time is equal to the search time of the n-th replica. For example, if the n-th replica is located at distance 4 from the client, then the complete search will take between 0.05 and 0.8 milliseconds (see Figure 2a). Also, search time is bounded by 3.1 milliseconds, which corresponds to the time of a full graph exploration. The mean search time, however, is much lower: 0.64 milliseconds. These values are very low compared to typical name resolution times, which have been evaluated between 60 and 200 milliseconds [15].

Knowing the search times, we can also derive the theoretical maximum server throughput. In a pessimistic case, when all queries require complete AS-graph exploration, the redirector can handle about 290 queries per second. In a typical case, however, the throughput will be over 950 queries per second, as the mean request processing time equals 1.04 millisecond

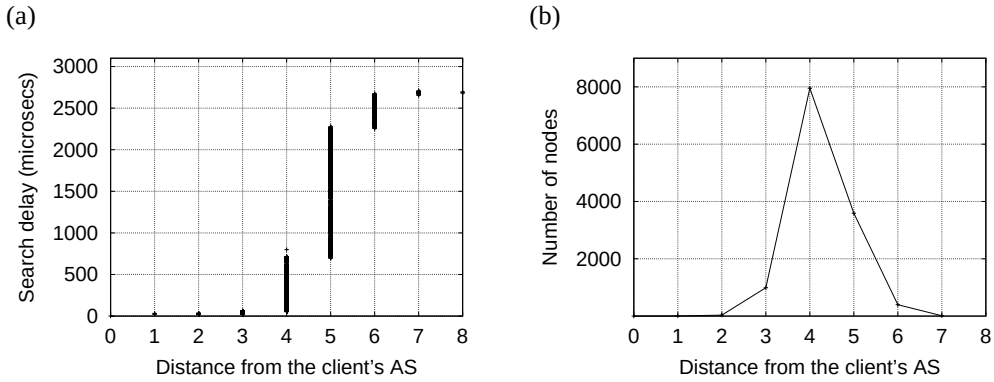

Figure 2: The results of the shortest AS-path policy analysis: search times (a), and distance distribution (b)

(including the overhead due to both UDP and DNS processing). This shows that the throughput of our redirector is close to that of Bind, even when taking into account the computational cost of the shortest AS-path policy. Also note that the throughput of a DNS redirection system can be increased to almost any level by combining several redirectors.

6 Conclusion

We have presented NetAirt, a redirection system that separates redirection mechanisms from redirection policies. NetAirt is implemented in the form of a module for the Apache HTTP server. HTTP redirection can be performed by simply compiling this module into Apache. To perform DNS redirection, however, we had to propose a few changes to the original Apache source code that enable it to service UDP datagrams.

NetAirt currently supports two basic policies and a more elaborated one, the shortest AS-path policy. If a need for other policies is ever identified, NetAirt provides a simple framework for implementing it without having to code redirection mechanisms again.

Performance measurements show that even a relatively complex redirection policy can be supported with low overhead compared to the overall name resolution time.

NetAirt will soon be released for public use. We hope that it can contribute to the development of worldwide-distributed services, and thus help sustaining the continuous growth of the Internet.

References

[1] *The Apache HTTP Server Project*, `http://httpd.apache.org/`.

[2] *The RouteViews Project*, `http://www.routeviews.org/`.

[3] A. Barbir, B. Cain, F. Douglis, M. Green, M. Hoffman, R. Nair, D. Potter, and O. Spatscheck, *Known CN Request-Routing Mechanisms*, Internet Draft, IETF, May 2002.

[4] T. Brisco, *DNS support for load balancing*, RFC1794, IETF, April 1995.

[5] Internet Software Consortium, *Berkeley Internet Name Domain*, `http://www.isc.org/products/BIND/`.

[6] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, *Globally Distributed Content Delivery*, IEEE Internet Computing **6** (2002), no. 5, 50–58.

[7] L. Gao, *On Inferring Autonomous System Relationships in the Internet*, IEEE/ACM Transactions on Networking **9** (2001), no. 6, 733–745.

[8] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, *DNS Performance and the Effectiveness of Caching*, IEEE Transactions on Networking **10** (2002), no. 5, 589–603.

[9] Z. Morley Mao, C. 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*, Proc. of USENIX Annual Technical Conference, 2002.

[10] P. R. McManus, *A Passive System for Server Selection within Mirrored Resource Environments using AS-Path Length Heuristics*, Tech. report, Applied Theory, Inc., June 1999.

[11] P. Mockapetris, *Domain Names – Concepts and Facilities*, RFC1034, IETF, November 1987.

[12] P. Mockapetris, *Domain Names – Implementation and Specification*, RFC1035, IETF, November 1987.

[13] G. Pierre and M. van Steen, *Globule: a Platform for Self-Replicating Web documents*, Proc. of the 6th International Conference on Protocols for Multimedia Systems, October 2001, pp. 1–11.

[14] Y. Rekhter and T. Li, *A Border Gateway Protocol 4 (BGP-4)*, RFC1771, IETF, March 1995.

[15] A. Shaikh, R. Tewari, and M. Agrawal, *On the Effectiveness of DNS-based Server Selection*, Proc. of IEEE INFOCOM, April 2001, pp. 1801–1810.

[16] H. Hong-Yi Tzeng, *Longest Prefix Search using Compressed Trees*, Proc. of IEEE Globecom, November 1998.

[17] M. Zari, H. Saiedian, and M. Naeem, *Understanding and Reducing Web Delays*, IEEE Computer **34** (2001), no. 12, 30–37.