Application-Layer Anycasting: A Server Selection Architecture and Use in a Replicated Web Service

Ellen Zegura and Mostafa Ammar and Zongming Fei and Samrat Bhattacharjee

Abstract — **Server replication improves the ability of a service to handle a large number of clients. One of the important factors in the efficient utilization of replicated servers is the ability to direct client requests to the "best" server, according to some optimality criteria. In the anycasting communication paradigm, a sender communicates with a receiver chosen from an anycast group of equivalent receivers. As such, anycasting is well suited to the problem of directing clients to replicated servers.**

This paper examines the definition and support of the anycasting paradigm at the application layer, providing a service that uses an anycast resolver to map an anycast domain name and a selection criteria into an IP address. By realizing anycasting in the application layer, we achieve flexibility in the optimization criteria and ease the deployment of the service.

As a case study, we examine the performance of our system for a key service: replicated web servers. To this end, we develop an approach for estimating the response time that a client will experience when accessing given servers. Such information is maintained in the anycast resolver that clients query to obtain the identity of the server with the best estimated response time. Our performance collection technique combines server push with resolver probes to estimate the expected response time without undue overhead. Our experiments show that selecting a server using our architecture and estimation technique can improve the client response time by a factor of two over nearest server selection and by a factor of four over random server selection.

1 Introduction

Users increasingly view the Internet as providing more than simple connectivity, but rather a range of sophisticated and complex services. As this view becomes prevalent, it becomes important to provide explicit support for the efficient delivery of networked services. Such support must be scalable to a large number of geographically wide-spread users, while maintaining user-perceived quality of service (e.g., response time, throughput, reliability).

Server replication [12] provides scalability by deploying multiple copies of a server and sharing client load across the copies. Server replication is appealing because it offers a relatively straightforward method to potentially improve client performance and reduce network load. A key issue in realizing this potential is the method used for *server selection*. That is, given a set of servers, how does a client select the "best" server?

A server selection system has the obvious design goal of improving client performance. In addition, a server selection system should satisfy the following additional goals. First, it should be flexible in the specification of selection criteria. The "best" server will vary depending on the service and (potentially) on the preferences of the clients. A server selection system should support a rich and flexible set of selection criteria. Second, it should be suitable for wide-area server replication. Although servers can be replicated locally in server farms, our interest is in server selection with global replication across geographically wide-spread locations. Local replication is both easier and more limited in ability to handle request load from wide-spread clients. Third, it should be deployable in the current Internet without modifications to the network infrastructure. Last, it should be scalable to a large number of services, client and client requests.

A number of services are currently replicated, using both local and global replication. The methods currently used for server selection include:

- Domain Name System (DNS) modifications [19] to return one IP address from a set of servers when the DNS server is queried. The DNS server typically uses a round robin mechanism to allocate the servers to clients, thus this technique is best suited to local replication of servers with comparable capacity.
- Network-layer anycasting [25], which associates a common IP anycast address with the group of replicated servers. The routing protocol routes datagrams to the closest server, using the routing distance metric. Standard intra-domain unicast routing protocols can accomplish this, assuming each server advertises the common IP address. The limitations of network-layer anycasting include lack of flexibility in the selection criteria — the routing protocol determines the (single) criteria, typically hop count — and difficulty in extending to wide-area selection.
- Router-assisted server selection, as in Cisco's DistributedDirector product [8, 13]. This product associates a Cisco router with each replicated server to act as the server's agent. Client requests are directed to a central location — the DistributedDirector (DD) — which queries the server agents to determine either hop count or link latency between each server and the client. The DD redirects the client to a server using the query results. This solution is best suited to server selection within a small to moderate-size domain, since it requires significant coordinated deployment of Cisco equipment and relies on routing tables to determine hop counts from a server to a client. For larger do-

E. Zegura, M. Ammar and Z. Fei are with the College of Computing, Georgia Tech. S. Bhattacharjee is with the Department of Computer Science, University of Maryland.

mains, scalability is likely to be an issue.

- Combined caching and server selection systems, such as developed in several recent commercial systems (e.g., Akamai [1], Sandpiper [29]), which operate their own system of caches containing content from a large number of servers. Client requests are directed to a cache, based on cache content and measurements of network and server load. Relatively little information is available regarding the operation and performance details of such systems. The basic premise differs, however, from our focus on "pure" server selection without deployment of caches.

None of the current solutions meet all of the design criteria outlined above.

Our proposed solution begins with network-layer anycasting. We adopt a general view of anycasting as a *communication paradigm* that is analogous to the unicast, broadcast and multicast communication paradigms. In particular, we differentiate between the anycasting *service definition* and the *protocol layer* providing the anycasting service. The original anycasting proposal [25] can, therefore, be viewed as providing an anycasting service definition *and* examining the provision of this service within the IP layer.

We move anycasting to the application layer, allowing us to achieve flexibility in selection criteria, extension to the wide-area, and ease of deployment. For scalability, we retain the best-effort nature of the original network-layer anycasting service definition. This paper describes our application-layer architecture and develops a case study using the architecture for replicated web servers. Our contributions are threefold:

- We generalize the original definition of anycasting to design an anycasting service that offers considerable advantages in flexibility over the traditional network-layer anycasting service.
- We develop an application-layer architecture to realize our anycasting service. Our architecture provides for scalability by using replicated resolvers to handle queries from a set of clients and by organizing the resolvers into a DNS-style hierarchy.
- We examine the performance of our system for client access to replicated web servers. We develop an approach for estimating the client response time that combines server push with resolver probing. This metric is challenging to estimate because the response time is a function of both server load (relative to capacity) and of path load between the server and client. Our experiments show that selecting a server using our architecture and estimation technique can improve the client response time by a factor of two over nearest server selection and by a factor of four over random server selection.

The paper is structured as follows. In Section 2 we define anycasting as a paradigm and identify the components of our application-layer architecture. Section 3 describes a key aspect of the architecture, specifically maintenance of performance metric information. Sections 4 and 5 consider the use of the system for replicated web access. Our technique for estimating response time is developed in Section 4, while Section 5 describes a set of performance evaluation experiments. We describe related work in Section 6 and conclude in Section 7.

2 Application-Layer Anycasting: System Overview

The anycast paradigm shares characteristics with both the multicast and unicast paradigms. Similar to multicast, the anycast paradigm consists of *groups* of destinations, with the semantics that each destination in a given anycast group is equivalent in some sense. Similar to unicast, a sender that communicates with an anycast group typically interacts with one destination, chosen from the anycast group. This section describes our anycasting service and the architecture for providing the service at the application-layer. We conclude the section with an assessment of how well the architecture meets the design goals outlined in the Introduction.

2.1 Architecture

In our architecture, we define an *anycast group* to be a (potentially dynamic) set of unicast or multicast IP addresses. Such a definition allows considerable flexibility in the types of services that our selection method supports. We see two particularly useful consequences of this definition. First, a set of servers may be grouped together based on equivalence *from a user's perspective*. That is, "exact" replication is not required for membership in the same group. A user might define an anycast group to contain, for example, the web sites for CNN Interactive, Time Magazine and USA Today. Second, allowing multicast IP addresses means we can support services that require multiple servers to provide a single instance of the service. For example, a client may wish to merge or edit video clips that can be found on different sets of replicated video servers. The desired service is provided by a group of servers, one per video clip.

In our architecture, a client interacts with an anycast group via a query-response protocol illustrated in Figure 1. The anycast query contains the *anycast domain name* (ADN), which identifies the group, and the selection criteria to be used in choosing from the group. The anycast response contains the IP address for the selected server. As illustrated in Figure 1, the architecture centers around the use of a hierarchy of *anycast resolvers* that perform the ADN to IP address mapping. The resolver receives the anycast query and applies a *filter* to control the selection. A filter operates on a set of anycast group members and returns a (possibly empty) subset. A second filter may be applied at the client. Filters may be content-independent (e.g., select any member at random), or based on performance metrics or policy information.

To do the mapping, the resolvers maintain two types of

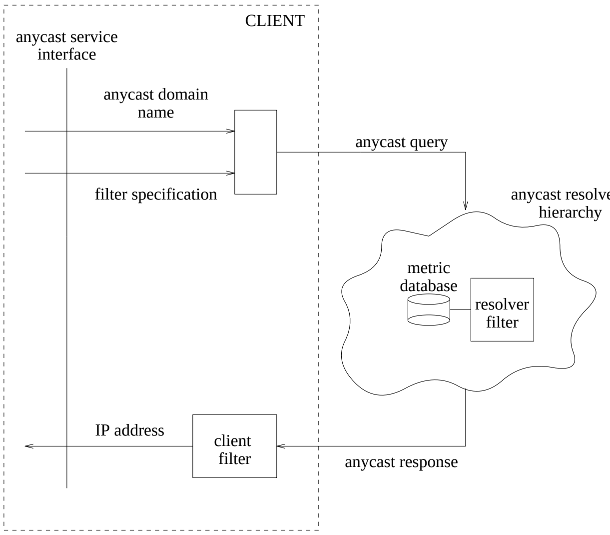

Figure 1: Anycast Name Resolution Query/Response Cycle

information, (1) the list of IP addresses that form particular anycast groups, and (2) a metric database of information associated with each member of the anycast group. As described further below, authoritative resolvers maintain the definitive list of IP addresses for a group, whereas local resolvers cache this information. A membership protocol updates the anycast group information, and a service creation protocol defines new anycast groups. We do not discuss the details of such protocols here; some effort in this area has been undertaken in the IETF [33]. Many of the metrics are locally significant, thus they are maintained independently at each anycast resolver that has the ADN group membership information cached. The authoritative resolver may provide its locally-maintained metric information as a "hint" whenever it receives a request from another resolver for the anycast group member list for a given ADN.

The structure of anycast domain names influences the operation of the anycasting system in general, and the anycast resolver architecture in particular. We use a DNS-style naming and directory service architecture for scalability and ease of integration into the existing Internet infrastructure. While the anycast resolver is logically distinct from other name servers like DNS [22], the functions of an anycast resolver could be integrated with the operation of DNS. In our scheme, an anycast domain name is of the form `<Service>%<Domain Name>`. Such a name will typically be used as an argument to a library call that invokes the anycasting service and results in the mapping of this ADN to an IP address. The `Domain Name` part of the system indicates the location of the *authoritative* anycast resolver for this ADN. The `Service` part of the ADN identifies the service within the authoritative resolver.

The architecture for handling anycast requests is shown in Figure 2. Each network location is preconfigured with

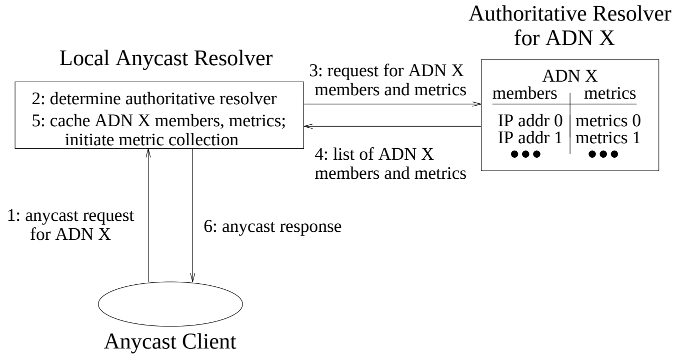

Figure 2: Anycast Request-Handling Architecture

the address of its local anycast resolver in the same way local DNS servers are configured. An anycast client makes its initial anycast query to its local resolver. If the resolver is authoritative for the ADN in the query or if it has cached information about the ADN, it can process the query immediately and return the appropriate response. Otherwise, the local resolver determines the address of the authoritative resolver for the `Domain Name` part of the ADN and obtains the anycast group information from this resolver. Determining the address of the authoritative anycast resolver for a particular domain can be done using techniques similar to DNS authoritative name determination [22].

2.2 Design Goals Revisited

With respect to the design goals presented in the Introduction, the proposed architecture clearly meets the first three goals. The user specifies the selection criteria by way of the filters, thus supporting flexibility in selection. The resolvers maintain lists of servers and explicitly track metrics associated with each server. These metrics may include both path and server load characteristics, as is necessary for wide-area server selection. The architecture does not rely upon changes to the network infrastructure, thus it is deployable in the current Internet. As we will see in the next section, modest changes to the servers can facilitate metric collection.

Whether the architecture meets the scalability design goal is less clear. The architecture attempts to achieve scalability in three ways. First, the service is best-effort, thus explicitly allowing techniques that improve scalability at some sacrifice in optimal performance. For example, a given resolver might only track the performance at a subset of servers that are deemed to be most promising, based on some (longer time-scale) mechanism. We have not, however, fully explored the performance trade-offs associated with such scalability techniques. Second, we use DNS-style replication and hierarchy in the resolvers, thus reducing the load on any one resolver. Third, we have developed a relatively efficient mechanism to track server response time, using a combination of light-weight server pushes and less frequent, heavier-weight probes. Various methods for metric maintenance are discussed next. The hybrid push-probe mechanism is discussed in detail in Section 4, and the performance is evaluated in Section 5.

3 Maintenance of Metric Information in Resolvers

The methods used by the resolver to maintain selection metrics are key to the performance of the architecture. Metrics fall into three general categories: those that depend only on server characteristics, those that depend on characteristics of the client-server path, and those that depend on both server and path. A variety of techniques will be used to maintain metric information, depending on factors such as the category, the accuracy required and the cost of burdening the network and/or the server. Examples of maintenance techniques include:

Remote Server Performance Probing:
In this technique, a probing agent makes periodic queries to the servers to estimate the performance that a client would experience. These queries appear to the server to be legitimate client requests, and thus they measure expected client peformance. Probing agents would normally be co-located with resolvers but may also be running at other locations. Each probing agent acts as a proxy for real clients within a certain region, thus the farther away a client is (in Internet "distance") from a probing agent, the less useful the probe measurements.
This technique measures network path performance and does not require server modification; on the other hand, the load on the network and servers may be significant.

Server Push:
In the server push technique [17, 18], the server monitors its performance and pushes this information to the resolvers when interesting changes occur. For additional scalability, the update information can be network-layer multicast to all resolvers that maintain information about the server. The anycast resolvers can join well-known multicast groups for each server that they are interested in, allowing the servers to disseminate performance information without knowing the identities of the resolvers.
The server can control the network traffic generated by this mechanism by adjusting the monitoring and push schedules. The primary advantages of this technique are scalability and accurate server measurements; the disadvantages are that the servers must be modified and the network path performance is not easily measured. Some properties of the one-way path from the server to each resolver could be measured as part of the multicast push. For example, the hop count from the server to the resolver could be determined via use of the TTL.

Probing for Locally-Maintained Server Performance:
A variation on the probing technique allows the probing agent to obtain server load information. Specifically, each server can maintain its own locally-monitored performance metrics in a globally readable file. Remote probing locations can then read the information in the file (as opposed to attempting to exercise the server) to obtain the desired information. Since probes merely read from a locally-maintained file, they may represent less of a burden on the server than the probes that mimic client requests.

User Experience:
Users currently make server access decisions based, in part, on past experience. Collecting information about past experience offers a coarse method of maintaining server performance. The primary advantage of this method is that the information is collected for free; no additional burden is placed on the server or the network. The quantity and accuracy of the information can be increased by sharing of experience among clients. For example, a gateway into a campus might maintain server performance information based on the experience of all clients on the campus. An architecture for collection and sharing of such information is being developed in the SPANDS project [31].

Table 1 summarizes the four techniques based on performance and cost dimensions. The first three columns are measures of system overhead. The Net Load column represents the number of messages generated per unit time to obtain the metric data from one server, where P is the number of probing agents, T_p is the period of probing, T_s is the period of server push. The Server Push messages can be multicast rather than unicast, reducing their burden. The Server Mod column indicates whether the server must be modified to allow the metric to be collected. The Server Load column expresses (relatively) how much additional load is placed on the server by the collection of the metric data. The last two columns are performance measures, indicating whether the method exercises network path, and (relatively) how accurately the method is able to maintain the metrics that it can evaluate.

The appropriate technique to use for maintaining performance metric information is highly dependent on the service details and context. In the next two sections we examine in detail a technique that is well-suited to selection among replicated web servers.

4 Case Study: Web Server Response Time

We turn our focus to the issue of how our application-layer architecture can be used for selection amongst replicated web servers. In particular, we design and evaluate a performance monitoring system for the estimation of service response time experienced at a client and use the estimates to guide selection of servers within our system.

The response time metric is important because it directly correlates with a user's perception of the quality of service.

	Net Load	Server Mod	Server Load	Exercises Net Path	Accuracy
Probing	$2PT_p$	No	High	Yes	Moderate
Server Push	T_s	Yes	Low	No*	High
Reading Server Log	$2PT_p$	Yes	Moderate	Yes	High
User Experience	None	No	None	Yes	Low/Varies

(* Can measure one-way path information)

Table 1: Comparison of Metric Collection Techniques

In addition it is a very difficult metric to monitor since it depends on server capabilities (e.g., speed and number of processors at the server), current server load (e.g., number of queries currently being served), network path characteristics (e.g., propagation delay on the path) and current path load. Thus, the metric collection technique must measure both server and path performance.

The metric collection technique should meet two basic goals. First, it should be scalable to a large number of servers, anycast groups and clients. The load placed on any component of the system — servers, network resources, resolvers, clients — in collecting metric data must be kept "reasonable". Second, the metric collection should be *relatively* accurate. The service provided by anycasting can inherently deal with inaccuracy in the absolute values of the metrics, since the service makes a relative selection amongst servers. The service is also somewhat robust against errors in the relative values of the metrics, due to the best-effort nature of the service. The performance penalty associated with out-of-date or slightly inaccurate metric data will not typically be severe; rather than selecting the "best" server, the service may identify a "nearly-best" server.

The two goals constrain the design of the metric collection technique in the following ways. First, metric updates should occur primarily in response to significant changes in metric value, rather than on a periodic basis. This implies monitoring of metric values to determine when updates are needed. Second, servers should have some control over the load incurred due to metric collection. A server should be able to decrease metric collection load, if desired.

4.1 Overview: Metric Collection Technique

To build a metric collection technique meeting the goals and constraints outlined above, we combine the probing and server push techniques described in Section 3. Probing gives the most accurate estimate of what the probing agent expects in terms of server response time. Probes, however, can represent a significant overhead if performed frequently. Server pushes, while more lightweight, are less accurate predictors of response time since they only propagate server performance information. Our technique combines server push with less frequent periodic probing.

Server Push Algorithm: The server will measure its performance and push performance information according to an update algorithm. To define the way the server measures its performance, consider the server response cycle

```
assign process to handle query
parse query
locate requested file
repeat until file is written:
    read from file
    write to socket
```

To assess its performance, the server measures the time from just after assigning the process until just before doing the first read. These measured values are averaged and smoothed before being used in the update algorithm described below. (Note that this is the cycle used by the Apache server. We expect that other web servers will have a similar high level processing structure. If this is not the case, the server measurements will need to be modified accordingly.)

We want the server to push performance information whenever its measured performance has changed sufficiently to be "interesting," with some constraint on the maximum frequency of updates so as to bound the overhead of the updating mechanism. The task of updating link state in a distributed routing environment has precisely the same criteria, thus we have adopted the link state update algorithm used in the ARPANET [26]. The update algorithm is parameterized by a measurement interval I, a maximum threshold T and a reduction factor R. The algorithm maintains a current threshold C, initialized to T. The server measures its performance over each interval I. If the new measured value changes from the previous measurement by at least C, the new measurement is pushed, and C is reset to T. If the state does not change by at least C, C is reduced by R. When C becomes 0, the state will be pushed, and C will be reset to T. The algorithm will send updates at least every $T/R * I$ time units and at most every I time units.

Agent Probe Mechanism: The probe is made to a well-known file that is maintained at anycast-aware servers specifically to service probe requests. The file contains the most recent measured performance value by the server and is padded with dummy data. Each probe results in a response time measurement, taken from just before sending the query to just after receiving the complete response. This time depends on server and path characteristics and on the size of the file being probed.

A Hybrid Push/Probe Technique: We combine the performance value pushed by the server with the response time measured by the probes to keep an estimate of server response time. The idea is to use the probes to get a measurement of the response time that includes the network path. The measurement is then used to calibrate the more frequently pushed server time value to get an expected response time at a given resolver.

Specifically, let R denote the most recent measurement of response time when probing for the well-known file. As indicated earlier, the server includes in the well-known file the most recent performance value measured as described above. Let S denote the server time value reported in the file during the most recent probe. In between consecutive probes the server typically pushes a sequence of server values. Let $S(i)$ denote the i-th value pushed by the server. The resolver adjusts the server-reported value $S(i)$ by multiplying by an adjustment factor $A = R/S$. Thus, the resolver estimates the current response time as $R(i) = A * S(i)$. Typically, the probes will occur less frequently than the server pushes its measured time value, thus a given adjustment factor will be used to adjust a sequence of pushed server values, until the next probe occurs and updates A.

As will be shown by the results of the experiments, this technique works quite well for our purposes. To understand the intuition behind it, we note that the value of S is the average time until the server *begins* to serve a page and includes delays incurred because of the need to process other requests at the server. In a sense S is the time required for the request to receive one unit of service and $A = R/S$ is an estimate of the number of units of service required to service a page. While S is a function of server load, the value of R and, consequently A, is strongly dependent on the characteristics of the path from server to client.

4.2 Evaluation of Push-Probe Technique

In Section 5, we examine the performance of the overall anycasting system. Prior to combining all parts of the system, we evaluate the accuracy of the metric collection technique in isolation. To do this, we experimented with various locations and capabilities of servers and resolvers, variations in server load, and alternative file sizes. Figure 3 shows a typical result, plotting the estimated and actual values of response time over 270 queries. The x-axis indicates the index of each query.

In this particular experiment, the probing agent was at the University of Maryland, and the server was located 12 Internet hops away at Georgia Tech. The agent made requests to the server according to an access log file from a real server. That is, the access log file was used to determine the time of the query (relative to the starting time of the experiment) and the particular file to request. Approximately 26 minutes elapsed from the first to the last access, thus the average interarrival time of accesses was 5.8 seconds. The server was loaded by other clients that also made log-driven requests. The server pushed its load using the push algorithm with threshold value (T) 0.01, reduction factor (R) 0.002 and measure interval (I) 4 seconds. Thus, a push occurred at most every 4 seconds and at least every 20 seconds. The agent probed for the well-known file every fifty accesses.

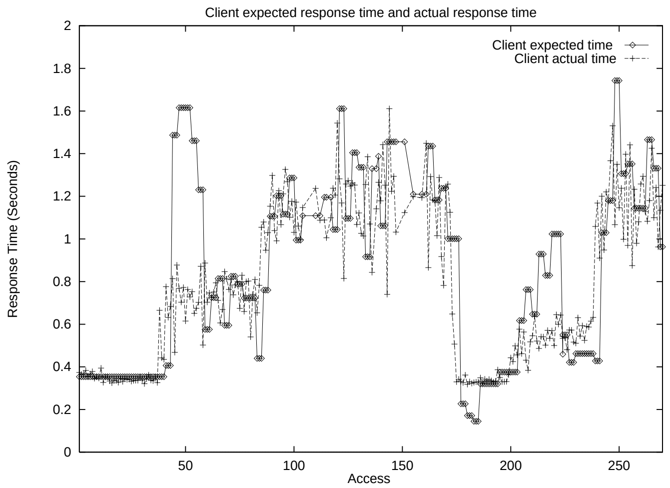

Figure 3: Response Time Estimation using Proposed Metric Collection technique

This plot demonstrates that our method yields estimated response times that are relatively similar to the actual response times. That is, the estimate tends to increase when the actual time increases, and vice versa. However, the estimate is not always accurate in the absolute sense, with the estimated time generally higher than the actual time.

4.3 Refinement

A potential problem with an approach that identifies the best server to clients is that of *oscillation* among servers. As clients discover a newly designated best server, they all divert their workload to that server thus off-loading one or more servers which now become the designated best servers, and so on.

To address this issue we introduce the concept of *the set of equivalent servers (ES)* which is defined as the subset of the replicated servers whose measured performance is within a threshold of best performance. This set of equivalent servers is recomputed every time a new pushed value is received at the resolver. In answering an anycast query, the resolver picks at random from this set of equivalent servers. The set *ES* is computed according to the following algorithm, executed whenever a new push or probe measurement is received:

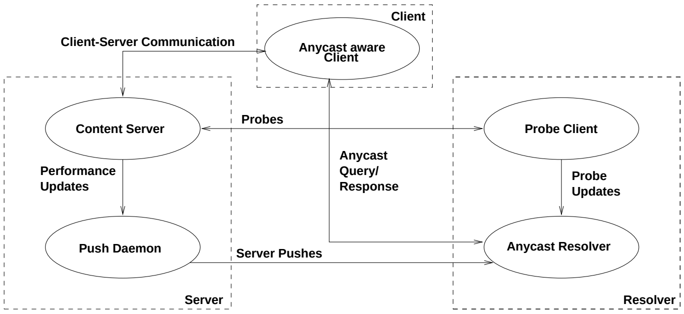

Figure 4: Software components of the application-layer anycasting architecture

compute the new estimated response time for server
sort servers according to estimated response time
$ES = ES \cup$ {Server with minimum response time estimate}
R_{min} = minimum response time estimate
for each server $j \in ES$
    if $(R_j - R_{min} > \tau_\ell)$
        then $ES := ES$ - {j};
for each server $j \notin ES$
    if $(R_j - R_{min} \leq \tau_j)$
        then $ES := ES \cup$ {j};

The leave and join thresholds τ_ℓ and τ_j are such that $\tau_\ell \geq \tau_j$. Together they allow a form of hysteresis used to achieve stability in the constitution of the set ES.

As will be illustrated by the results of the experiments in the next section, this algorithm successfully deals with the oscillation problem. When a server reports good performance, the resolver will direct more (but not all) clients to this server. The selection of the thresholds and their effects are studied in Section 5.

5 Experimental Evaluation

In this section we describe a set of experiments that we conducted over the Internet, using our proposed anycasting architecture and the technique described in Section 4 to monitor response time. We first describe the experimental setup, then discuss results from experiments to assess the overall system performance.

5.1 Experimental Setup

The experimental setup is designed to mimic as much as possible the true operation of a set of Web clients and servers using the anycasting architecture, depicted in Figure 4.

Anycast-aware Servers: We modified the Apache HTTP [1] daemon to act as a performance monitoring server as described in Section 4. We wanted to distribute anycast-aware web servers, that serve realistic content, without distributing a large amount of data. The modified web server emulates the real server in the number of bytes sent for each request without actually maintaining the files of the real server. We achieve this by having our modified server maintain a small set of *dummy* files, and a table containing tuples of the type `(filename, size)`. The table is constructed using file sizes taken from an actual server. Whenever an access is made to our modified server, a table lookup is performed to determine the file size. Then the name of the requested web page is mapped into the name of the next largest file maintained on disk. This mapping is such that the same web page name will always map to the same file. An amount of data equal to the size of the original file is then read and transmitted in response to the request. Four files of each size are kept, to partially avoid false caching effects. This method of server emulation comes close to the operation of a real server, and allows any optimizations that the server may perform (e.g., by caching data read from disk) to be reflected in our experimental results, albeit approximately.

The modified server was also instrumented to monitor its performance by recording time values as described previously. These values are recorded in a performance measurement file that the server shares with the performance push daemon. The performance push daemon executes the update algorithm given in the Section 4. If the variation in server performance or the expiration of an epoch warrants a performance push, the performance measure is "pushed" to the resolvers.

Finally, the modified web server maintains a special file that is requested by probing agents. The probing file is the same size at all servers. We use the average size of files recorded in the log file as the size of this special file. Note that the validity of the probing technique does not depend on the probe file size being "representative" of the server files: the response time measured at probing can be used to compare the relative performance of servers as long as the sizes of probing files are the same at all servers. This is important, since web item size distributions have been observed to have infinite variance [2].

Clients: We modified the NCSA Mosaic [2] client to have the desired behavior. For our prototype we use a simple form of API that encodes the specification of the anycast group and selection criteria in a name used to reference the service. Specifically, we use a Metric-Qualified Anycast Domain Name (MQ-ADN) of the form `<Filter>.<ADN>` where `Filter` provides information about the filter to be used by the resolver. In the prototype, the resolver returns a host name, which we pass to DNS to determine the IP address.

The prototype processing is shown in Figure 5. We intercept calls to the `gethostbyname()` library call and check if the argument is an ADN, by checking for a `.any` suffix. For ADN arguments, an anycast resolver query is formulated, and the reply is used as input to the original `gethostbyname()` procedure to return the desired IP

[1] http://www.apache.org/dist/

[2] ftp://ftp.ncsa.uiuc.edu/Mosaic/Unix/source/Mosaic-src-2.6.tar.Z

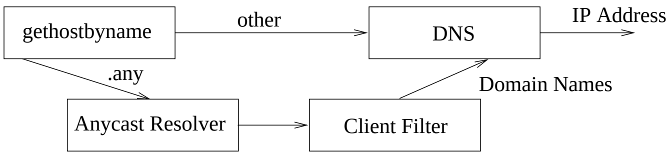

Figure 5: Implementation using Metric-Qualified ADNs

address. If the `gethostbyname()` argument is other than an ADN then the usual procedure is called directly. This allows us to use traditional applications without modification and gives the option of using them with or without the anycasting feature.

The behavior at our clients consists of two parts. First, a client must determine a set of requests to make to the replicated service over time. Second, for each request, a client must determine which server to access. For the first part, we use the access logs from a moderately busy web server (approximately 4000 accesses per hour) to generate client requests. The client reads the access log file to determine a filename to access and a time, relative to the start of the experiment, to issue the request. The clients schedule each subsequent access at this calculated relative access time. In this way, the clients replay the logged accesses.

We want to compare the performance experienced by clients that are anycasting, as compared to clients that use other methods to select a server. We support three methods for determining which server to access for a request: (1) querying the anycast resolver to determine a best server, (2) choosing a server at random, and (3) choosing the server that is closest to the client based on hop count. We control which method is used on a per-client basis.

Anycast Resolver and Probing Agent: For the probing agent we used a modified NCSA Mosaic client that periodically queries each server in the anycast group for the probing file (with the performance data), and communicates the end-to-end response time and performance data to the resolver. The anycast resolver can process two different types of performance updates, corresponding to server pushes and client probes. It maintains a database of anycast group members, and their current push and probe data. In response to query messages, the resolver returns one of the *best* servers as computed using the algorithm given in Section 4.

Client and Server Internet Locations: In each of the experiments there are four anycast-aware servers, one running at the University of California, Los Angeles, one running at Washington University, St. Louis and two running at Georgia Tech, Atlanta. The anycast resolvers were run at the University of Maryland, College Park and Georgia Tech, Atlanta. The anycast-aware clients were located at the University of Maryland, College Park and Georgia Tech, Atlanta. (See Figure 6.)

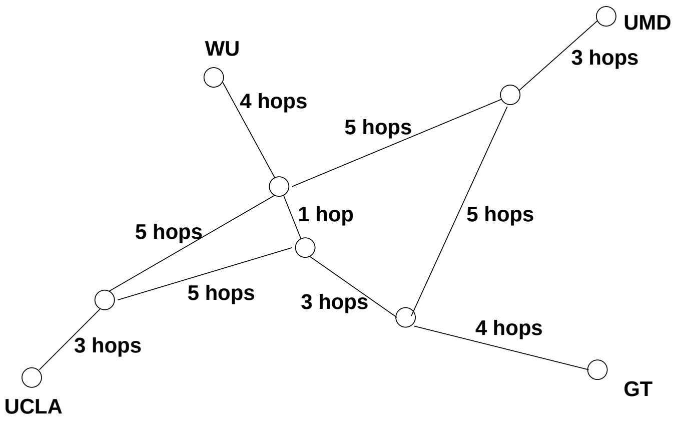

Figure 6: Snapshot of Experimental Topology

5.2 Evaluation of Anycasting System

In this section, we describe several experiments to evaluate the performance of the system. The experiments answer five primary questions. When all clients use the same method to select a server, how does the performance of anycasting, random, and nearest server selection compare? When some clients use anycasting and some use random access, how does the performance vary? What is the effect of variation in the push and probe frequencies? What is the effect of the join and leave thresholds? Does the equivalent server technique effectively reduce oscillations?

Our results are based on a set of experiments involving four servers and 20 clients. They show that anycasting *can* offer a considerable performance benefit over random or nearest server selection. Specifically, we observe a factor of four improvement in response time for anycasting over random selection and a factor of two improvement over nearest server selection. The performance experienced by another set of clients and servers will, of course, be dependent on the loading patterns for the systems and the client-server paths. The question of the expected performance gain for a "typical" set of clients and servers remains open.

We have 20 clients running, 16 at Georgia Tech and 4 at the University of Maryland. The clients are organized in four groups of five clients each. Each group generates requests equal to one log file with each client within the group allocated one-fifth of the requests in the log file. In order to put enough load on the servers, we also scale the log by a scaling factor k. For each access read, the client requests the same document k successive times. By changing the value k, we can generate different levels of traffic on the servers and use the log of our moderately busy server to get the effect of a busy server. In our experiments, we set k to 3. Except in the last experiment testing the effects of join and leave threshold on the performance, the resolvers set the join threshold to 0.1 and leave threshold to 0.3.

In Figure 7, we depict the performance experienced by a client in two settings. In one setting, the clients all use random server selection; in the other, the clients all use anycasting for server selection. In the anycasting case, the probe agent accessed the servers once every 4 minutes,

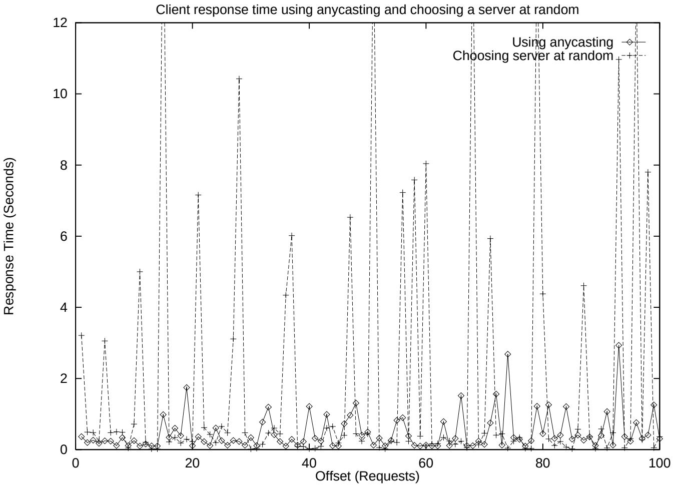

Figure 7: Comparison of performance of a client in all anycasting and all random experiment

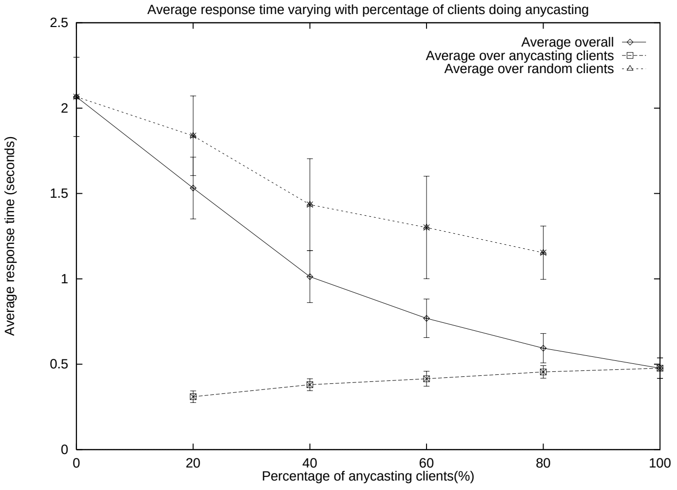

Figure 8: Response Time Varying with Percentage of Clients Using Anycasting

however the duration of the experiment short enough (2 minutes) that only one probe was performed at the beginning. The server performance monitor was run once every 10 seconds. (Note that this is the interval at which the server logs its own performance; depending upon the state of the system, a push update may or may not be issued at every 10 second interval. However, regardless of activity at the servers, a push update was issued at least once every 50 seconds.)

This plot shows the response time experienced at a single client, in each of the two settings, over a sequence of 100 requests. The experiments for the two settings each took about 2 minutes each; the two experiments were run one right after the other. The experimental setup keeps the total server load constant from one experiment to the next, however we could not control variation in the load on network paths. Thus, the response times for a given request cannot be directly compared. Instead, the important feature of the plot is the relative values over the complete set of accesses. The response time in the anycasting case is always less than 4 seconds, and generally less than 2 seconds, while the response time in random case can exceed 12 seconds.

Server Location Algorithm	Avg. Response Time (sec.)	Standard Deviation (sec.)
Anycasting	0.49	0.69
Nearest Server	1.12	2.47
Random	2.13	6.96

Table 2: Performance of Server Location Schemes

Table 2 summarizes the comparison between the different methods for identifying servers. Mean and standard deviation of response time are reported, based on the values experienced by all clients participating in the experiment. Each value in this table results from an experiment that lasted about 30 minutes with about 2000 accesses. The three separate experiments were run back-to-back. In addition to the random and anycasting selection methods, the table also includes the nearest hop count method. We note that choosing the nearest server improves upon random selection, however another factor of two improvement is possible with anycasting selection. The nearest server and random selection methods also exhibit much higher standard deviation than the anycasting selection, leading to a more unpredictable service.

In the previous experiments, all of the clients used the same server selection method. In Figure 8, we vary the percentage of the clients using the anycasting method as opposed to random selection. The x-axis indicates the percentage of clients that are anycasting, ranging from 0% (i.e., all clients use random selection) to 100% (i.e., all clients use anycasting). Three response time curves are shown: average response time for clients who anycast, average response time for clients who select randomly, and overall average response time. For each x-axis value, we run five separate 30 minute experiments. Each experiment produces an average response time for anycasting clients and an average response time for random selection clients. We average the five values and take confidence intervals to produce the curves. We note that average response time over all clients decreases from 2.1 s to 0.5 s when the percentage of anycasting clients increases from 0% to 100%. The average response time for the clients selecting at random also improves as more clients use anycasting, due to the better load balancing achieved on the servers. The average response time for anycasting clients increases from 0.3 s to 0.5 s when the percentage of anycasting clients changes from 20% to 100%. The anycasting mechanism performs relatively better when server load is unbalanced, since more lightly loaded servers can be identified and used. However, even when all clients are anycasting, the performance is quite good.

Also included in Figure 8 is the confidence interval for each point with 90% confidence level. The confidence inter-

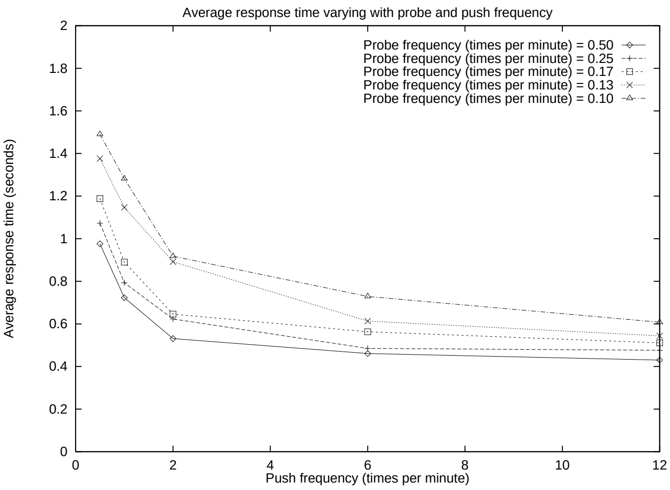

Figure 9: Response Time Varying with Push and Probe Frequency

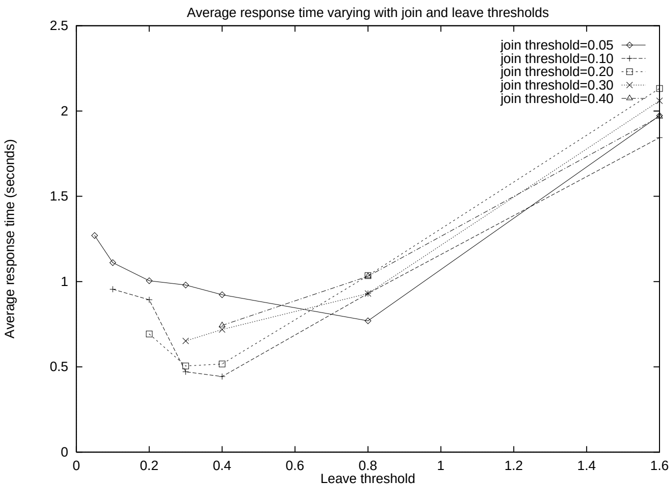

Figure 10: Response Time Varying with Join and Leave Thresholds

vals of response time for anycasting clients are rather small while the confidence intervals of response time of random clients are much larger. This is not surprising if we consider the variation observed earlier for random accesses. We can conclude from the data that the anycasting clients perform a good deal better, on average, than the random clients across the full range of the experiment.

In Figure 9, we analyze the performance of the anycasting system for varying values of probe and push frequencies. The value of push frequency is the maximum value; pushes will occur less often if the conditions warrant. All clients are using the anycasting mechanism. For each data point, the average response time is recorded over a period of 30 minutes. In this experiment, we notice rather sharp regions in the probe frequency-push frequency space, beyond which increases in frequency do not result in appreciable improvements in performance. Most notably, more than 2 pushes per minute yields little performance gain unless the probing frequency is very low. Further, a tradeoff between push and probe frequencies is evident. For high push frequencies, e.g. 12 pushes per minute, the average response time when probing every 10 minutes (probe frequency = 0.10 probes/min) is comparable to response times achieved by one probe every 2 minutes (probe frequency = 0.50 probes/min) with 1 push per minute. This ability to tradeoff probes for server pushes leads in general to a more scalable system: server pushes can be connectionless and multicast, with push frequency controlled by some server-specific process (e.g., taking current server load and policies into account), while probes will be connection-oriented, unicast transactions.

In our last experiment, we explore the effects of the join and leave thresholds in the resolver algorithm. Recall that these determine when a server is added or removed from the set of equivalent servers. When the thresholds are low, the algorithm is conservative in adding servers to the set, and aggressive in removing servers from the set. Thus, the set tends to be small. When the thresholds are high, the addition process is aggressive, while the removal process is conservative. Thus, the set tends to be large. Figure 10 shows the results of our evaluation. Each data point is the result of an experiment that lasts about 30 minutes. For a join threshold less than 0.3, we observe an initial performance improvement as the leave threshold is increased, followed by performance degradation as the leave threshold continues to increase. For join thresholds at least 0.3, the performance degrades with an increase in the leave threshold. When the leave threshold increases to 1.60, the response time is almost the same as in the random selection case. This is because a server will rarely leave the equivalent group after joining it. The resolvers essentially perform a random selection among the servers.

The relatively poor performance for low values of both join and leave threshold is caused by oscillation in server load. Since the thresholds are low, the equivalent server set at a resolver will usually contain only one server. Since all the clients are using anycasting, all requests to the same resolver will be directed to this single server. After a while, the performance of the server degrades and another server will be favored by the resolver. All the clients will then be shifted to that server. The oscillation phenomenon can be clearly seen in Figure 11. It shows the number of requests per second on one particular server in the experiment with join threshold and leave threshold both equal to 0.05. We note that this oscillation effect may be exaggerated by our experimental setup; a larger number of resolvers and less frequent client accesses will also tend to reduce oscillations.

When join and leave thresholds become a little bit larger, the equivalent server set is likely to contain more than one server. This helps avoid the oscillation observed earlier. In our experiment, the best performance was achieved when the join threshold is 0.1 or 0.2 and leave threshold is 0.3 or 0.4. This is the area where oscillation is reduced and the anycasting mechanism does play a role. Figure 12 shows the number of requests on a server using join threshold 0.1 and leave threshold 0.4. We note that the load on the

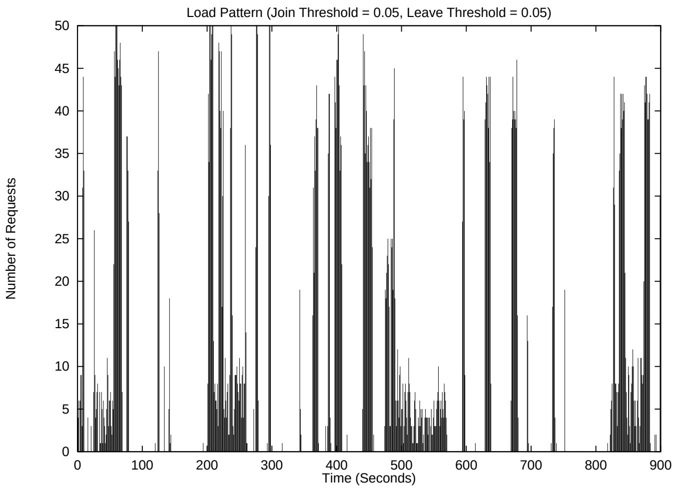

Figure 11: Server Load with Join Threshold = 0.05 and Leave Threshold = 0.05

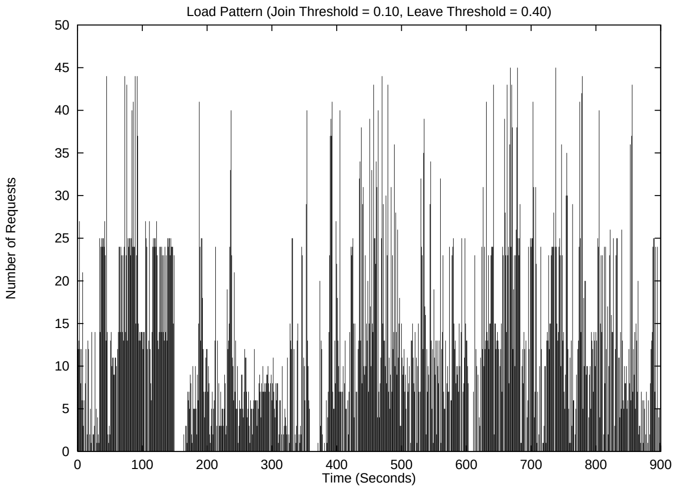

Figure 12: Server Load with Join Threshold = 0.10 and Leave Threshold = 0.40

server in this case does not exhibit the rapid changes observed earlier. We are re-assured by this experiment that the oscillation problem can be solved by introducing join and leave threshold to the resolvers and selecting appropriate values for them.

5.3 The Costs of Anycasting

Four new costs are incurred by the servers and the network due to our anycasting architecture. The costs are due to the server monitoring and push, the agent probe, and the client query resolution. The client anycast query resolution has cost equivalent to the usual domain name resolution. The anycast resolver architecture is such that anycast resolution can be combined with DNS lookup, making this cost negligible.

The server push daemon periodically computes a smoothed average of the measured server performance data. The cost of this computation is linear in the number of server performance measurements. Push messages will typically be small, making bandwidth consumption negligible compared to the actual server accesses. Also, the push updates can be multicast to multiple resolvers to save on bandwidth.

The cost of each probe to the network and to the server being probed is exactly the cost of an additional access to that server. With probes done infrequently —Figure 10 suggests perhaps once every 6-8 minutes— this will not represent a significant burden on servers.

Several techniques can be used to increase the scalability of the anycasting service. Naturally, each of these techniques will have an effect on the performance. First, the server can control the overhead for monitoring and pushes by adjusting the appropriate performance monitoring parameters. Likewise, the probing agent can control the overhead for probing via the frequency parameter. On a more coarse grain, the overall system can limit the number of anycast groups that are tracked. These groups might be restricted to those that are popular and can benefit substantially from an application-layer server selection technique. In a similar vein, a given resolver can track a subset of the "most promising" servers in a given group, using longer time scale information to identify which servers are most promising.

6 Related Work

The server or resource finding problem has been the subject of much investigation for over a decade. Initially, with low to moderate server loads, the problem was finding the desired network resource knowing only its name or property. Many techniques were investigated, including: 1) the use of multicast or broadcast communication to "touch" all the locations where the resource may reside in an attempt to find it (e.g., [3, 24]) , 2) the use of various name server architectures to lookup the location of the resource (e.g., [22, 15, 4]) and 3) the use of caching a resource's location (not content) at sites where the resource is frequently accessed [32]. This early work typically dealt with a single instance of the resource. The case of a mobile resource was addressed through interesting techniques such as the use of forwarding addresses [14].

Beginning with initial services like `ftp`, `archie`, and `gopher` and culminating more recently with the World-Wide Web, the Internet has experienced a dramatic growth in the use and provision of information services. This has resulted in heavy demands placed on servers and thus the desire to replicate (or mirror) servers. This adds a new dimension to the server finding problem: it is now important to find the "best" server from among many content-equivalent servers. Studies in this area include:

- The original work by Partridge et al. [25], proposing the idea of anycasting and discussing its network-layer support.
- A study by Guyton and Schwartz [16] which addresses the problem of locating the nearest server. The latter work also presents a classification of "best"-server lo-

cation schemes. The work is related to earlier work on the Harvest system [5] which provides a set of tools for gathering information from various servers and efficiently indexing and searching through this information. Tools for caching and replication of indices are also used in the Harvest system in order to improve the scalability of the service.

- The SONAR network proximity service [23] in which the authors define a service that can return the server that is the closest (in hops) from among a list presented to it.
- Work by Carter and Corvella [11, 7] addressing the issue of server selection. Their selection, however, has been primarily based on the characteristics of the path leading to the server. While they acknowledge the desirability of using server load information as a guide to server selection, their work does not address this issue (except for a limited experiment reported in [6]).
- The work by Colajanni and Yu [9, 10] considering how client requests may be scheduled among a set of replicated web servers.
- Cisco's DistributedDirector product which supports server selection based on hop count proximity and link latency. The system relies upon a distributed set of agents (i.e., routers with appropriate software) that can supply hop count information (from their own routing tables) and probe servers for path latency.

The Service Location Working Group of the IETF has been developing protocols to facilitate the discovery and selection of network services [33]. Thus, the high level objectives of their work and ours are quite similar. However, their focus has been on selecting network services within an enterprise network, not the global Internet. The Service Location group has just begun to consider the modifications necessary to support location in a global internet [28, 27], while maintaining backward compatibility with the solutions developed for an enterprise network.

The concept of probing the network and the servers to determine performance measures is related to various tools and systems that are used for network management purposes. An extensive set of tools and systems are available; the CAIDA project maintains a living list [3]. Examples of tools to monitor characteristics of paths and links include `ping`, `traceroute`,[4] and `pathchar`[5]. Examples of systems to monitor web service include commercial products such as Keynote[6] and NetScore,[7] and public domain software such as Timeit[8]. These monitoring systems are not integrated with server selection.

Protocol approaches designed to estimate various network conditions are also related to our probing concept. The packet-pair technique [20] is one such technique designed to estimate the bottleneck rate on a path. Remote measurement and monitoring of system performance has also been explored as part of the extensive work on distributed system monitoring [30, 21].

[3] http://www.caida.org/Tools/taxonomy.html

[4] ftp://ftp.ee.lbl.gov/traceroute.tar.Z

[5] ftp://ftp.ee.lbl.gov/pathchar/

[6] http://www.keynote.com

[7] http://penta.ufrgs.br/gereint/anacapa

[8] ftp://ftp.va.pubnix/com/pub/uunet/timeit-2.1.tar.gz

7 Concluding Remarks

As the Internet continues to grow, server replication will be increasingly important as a technique to scale services. The effective utilization of a set of replicated servers hinges upon the ability to appropriately allocate servers to clients. Simple techniques such as round-robin or nearest selection cannot accomodate the diversity of selection criteria that developing services will demand.

To address this, we have developed an application-layer architecture for the anycasting paradigm. Our architecture enables server selection based on a wide variety of criteria, including both performance and policy concerns. Our architecture achieves scalability by using replicated resolvers to handle queries from a set of clients and by organizing the resolvers into a DNS-style hierarchy.

We examined the performance of our architecture in some detail for an important server selection criteria, namely client response time. Our approach estimates the client's expected response time at each server using a combination of a relatively light-weight server push approach with a client-probe approach. Measured path-independent server performance (the pushed data) is calibrated using path-dependent response time measurements obtained via relatively infrequent probes. While we focused on HTTP servers operating within our architecture, the estimation technique has wider applicability to other types of servers and within other contexts.

We developed an experimental setup that allows us to distribute servers around the Internet without actually requiring them to maintain real data. Experiments that we conducted using our setup show that significant response time improvement that can be achieved with this technique over the use of performance-independent allocation mechanisms, including random and nearest selection.

Avenues for future work include scalable techniques to select different types of servers, the use of network-layer support to collect path performance metrics, and variations on the spectrum between multicasting and anycasting. We are currently investigating the potential for active networking to provide path performance metrics and to provide native support for network-layer anycasting.

References

[1] Akamai freeflow. http://www.akamai.com/.

[2] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proceedings of ACM SIGMETRICS '96 Conference*. ACM Press, 1996.

[3] J. Bernabeu, M. Ammar, and M. Ahamad. Optimizing a generalized polling protocol for resource finding over

a multiple access channel. *Computer Networks and ISDN Systems*, 27:1429–1445, 1995.

[4] A. Birrel, R. Levin, and M. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[5] C. M. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz. The harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28:119–125, 1995.

[6] R. Carter and M. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. Technical Report BU-CS-96-007, Computer Science Department, Boston University, Boston, MA, 1996.

[7] R. Carter and M. Crovella. Server selection using dynamic path characterization in wide-area networks. In *Proceedings of INFOCOM 97*, 1997.

[8] Cisco distributeddirector. http://www.cisco.com/.

[9] M. Colajanni and P. Yu. Adaptive ttl schemes for load balancing of distributed web servers. *Performance Evaluation Review - ACM SIGMETRICS*, 25(2):36–42, September 1997.

[10] M. Colajanni, P. S. Yu, and D. M. Dias. Scheduling algorithms for distributed web servers. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, May 1997.

[11] M. Crovella and R. Carter. Dynamic server selection in the internet. In *Proceedings of Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS'95)*, August 1995.

[12] P. Danzig, D. Delucia, and K. Obraczka. Massively replicating services in wide-area internetworks. Technical report, University of Southern California, 1994.

[13] Kevin Delgadillo. Cisco distributeddirector. http://www.cisco.com/warp/public/cc/cisco/mkt /scale/distr/tech/dd_wp.htm.

[14] R. Fowler. *Decentralized Object Finding Using Forwarding Addresses*. PhD thesis, University of Washington, 1985.

[15] I. Gopal and A. Segall. Directories for networks with casually connected users. In *Proceedings of INFOCOM 88*, pages 1060–1064, 1988.

[16] J. Guyton and M. Schwartz. Locating nearby copies of replicated Internet servers. In *Proceedings of SIGCOMM 95*, pages 288–298, 1995.

[17] J. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proceedings of the 1995 Workshop on Hot Topics in Operating Systems*, 1995.

[18] M. Humes. Netscape's server push, client pull and CGI animation. http://www.emf.net/mal/animate.html.

[19] E. D. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27:155–164, 1994.

[20] S. Keshav. Packet-pair flow control. *IEEE/ACM Transactions on Networking*, 1997.

[21] F. Lange, R. Kroeger, and M. Gergeleit. Jewel: Design and implementation of a distributed measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):657–671, Novemeber 1992.

[22] P. Mockapetris and K. Dunlap. Development of the domain name system. *Proceedings of SIGCOMM 88*, pages 123–133, 1988.

[23] K. Moore, J. Cox, and S. Green. SONAR - a network proximity service. *Internet Draft (work in progress) draft-moore-sonar-01.txt*, February 1996.

[24] D. Oppen and Y. Dalal. The clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems*, 3(1):230–253, July 1983.

[25] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. *RFC 1546*, November 1993.

[26] E. C. Rosen. The updating protocol of ARPANET's new routing algorithm. *Computer Networks*, (4):11–19, 1980.

[27] J. Rosenberg, E. Guttman, R. Moats, and H. Schulzrinne. WASRV architectural principles. *Internet Draft (work in progress) draft-ietf-rosenberg-wasrv-arch-00.txt*, February 1998.

[28] J. Rosenberg, B. Suter, and H. Schulzrinne. Wide area network service location. *Internet Draft (work in progress) draft-ietf-svrloc-wasrv-00.txt*, July 1997.

[29] Sandpiper footprint. http://www.sandpiper.com/.

[30] B. Schroeder. On-line monitoring: A tutorial. *IEEE Computer*, 28(6):72–78, June 1995.

[31] S. Seshan, M. Stemm, and R. Katz. SPAND: Shared passive network performance discovery. *1st Usenix Symposium on Internet Technologies and Systems (USITS '97)*, 1997.

[32] D. Terry. Caching hints in distributed systems. *IEEE Transactions on Software Engineering*, 13(1):48–54, January 1987.

[33] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol. *RFC 2165*, June 1997.