A Novel Server Selection Technique for Improving the Response Time of a Replicated Service *

Zongming Fei, Samrat Bhattacharjee, Ellen W. Zegura, Mostafa H. Ammar

Networking and Telecommunications Group, College of Computing
Georgia Institute of Technology, Atlanta, GA 30332
{fei,bobby,ewz,ammar}@cc.gatech.edu

Abstract

Server replication is an approach often used to improve 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 this paper we target an environment in which servers are distributed across the Internet, and clients identify servers using our application-layer anycasting service. Our goal is to allocate servers to clients in a way that minimizes a client's response time. To that end, we develop an approach for estimating the performance that a client would experience when accessing particular servers. Such information is maintained in a resolver that clients can query to obtain the identity of the server with the best response time. Our performance collection technique combines server push with client probes to estimate the expected response time. A set of experiments is used to demonstrate the properties of our performance determination approach and to show its advantages when used within the application-layer anycasting architecture.

1 Introduction

Server replication is an approach that is often used to improve the *scalability* of a service, i.e., its ability to handle a large number of clients with a reasonable quality of service. When replication is used, the primary concern is how a client may discover which of the servers is best to use. Many techniques have been used to guide clients to the use of a particular server among a set of replicated servers. These include:

1. Listing the servers and having the client pick one, based on geographical proximity or some other criteria the client deems appropriate. This technique is not transparent to the user. Furthermore, the closest server geographically may not be the best server to use because it may not be the closest in terms of end-to-end delay over the Internet and/or it may not be the least loaded of the servers.

2. Using Domain Name System (DNS) modifications [1] to return the IP address of one of a set of servers when the DNS server is queried. This technique is transparent to the client. However, often the DNS server uses a round robin mechanism to allocate the servers to clients because it maintains no server performance information on which to base its selection decision. This technique is best suited when the replicated servers are co-located on the same subnet and have the same request processing capacity.

3. Using a more elaborate location mechanism that targets network-wide replication of the servers over potentially heterogeneous platforms. Such a mechanism must allow for the maintenance of information about the servers and their relative performance. One such technique that we have proposed uses the *anycasting* communication paradigm at the application layer [2]. Our anycasting proposal complements IETF standardization efforts aimed at developing a set of specific service location protocols [3].

In this paper we examine one important aspect of the third technique described above, namely how can server performance information be collected at the site where server to client allocation decisions are made. Apart from an enumeration of possible techniques, this issue was largely left unanswered by our previous work [2] which was mainly concerned with the application-layer anycasting architecture. Our goal is to allocate servers to clients in a way that minimizes a client's response time. One challenging aspect of this problem is that the performance of a server is a function of its load (relative to the server capacity) and of the characteristics of the path between the server and the client. We derive an approach that can be used to determine the performance of such servers with respect to specific clients. We then use a set of experiments to show the advantages and implications of using this technique. Our work is done in the context of our proposed anycasting architecture, and we specifically consider replicated web servers. Our proposals do, however, apply to other server types and also have relevance within other server location architectures such as the proposed architecture being designed as part of the IETF's Service Location Working Group.

The rest of this paper is structured as follows. In Section 2 we present an overview of the Application-Layer Anycasting architecture. This is followed in Section 3 with a discussion of previous related work and in particular the relationship of our work to the IETF's Service Location Protocol efforts. Section 4 presents our performance monitoring and measurement technique and discusses the rationale for its adoption. Section 5 shows the

*This work is supported in part by research grants from IBM, Intel and NSF under contract number NCR-9628379.

results from experiments with clients using our anycasting architecture in conjunction with the performance determination technique we developed. The paper is concluded in Section 6.

2 Application-Layer Anycasting

As originally defined [4], anycasting provides:

> "a stateless best effort delivery of an anycast datagram to at least one host, and preferably only one host, which serves the anycast address."

In this definition, an *IP anycast address* is used to identify a group of servers that provide the same service. A sender desiring to communicate with only one of the servers sends datagrams with the IP anycast address in the destination address field. The datagram is then routed using anycast-aware routers to at least one of the servers identified by the anycast address.

In our work [2] we adopt a more general view of anycasting as a *communication paradigm* that is analogous to the 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 [4] can, therefore, be viewed as providing the anycasting service definition *and* examining the provision of this service within the IP layer. Our motivation derives from the fact that network-layer-supported anycasting has several limitations as discussed in [2], the most important being that application-specific metrics cannot be maintained at the network layer.

Whereas network-layer support hinges around the use of anycast IP addresses, our application-layer support makes use of *anycast domain names* (ADNs). An ADN uniquely identifies a (potentially dynamic) collection of IP addresses, which constitutes an *anycast group* [1]. The function of an application-layer anycasting service is thus to map an ADN into one or more IP addresses. An important feature of application-layer anycasting is that it does not require modifications to network layer operations.

Our design centers around the use of *anycast resolvers* to perform the ADN to IP address mapping. Clients interact with the anycast resolvers according to a basic query/response cycle illustrated in Figure 1: a client generates an *anycast query*, the resolver processes the query and replies with an *anycast response*. A key feature of the system is the presence of *metric databases*, associated with each anycast resolver, containing performance data about servers. The performance data can be used in the selection of a server from a group, based on user-specified performance criteria.

The issue of the structure of anycast domain names influences the operation of the anycasting system in general, and the anycast resolver architecture in particular. Our proposed approach is derived from the Internet naming and directory service architecture, which makes it

[1] It is also possible to define an anycast group as a collection of server domain names (or aliases). In this case the anycasting service provides a mapping from an ADN into a host domain name or alias. Obtaining the IP address in this case requires an additional DNS server lookup.

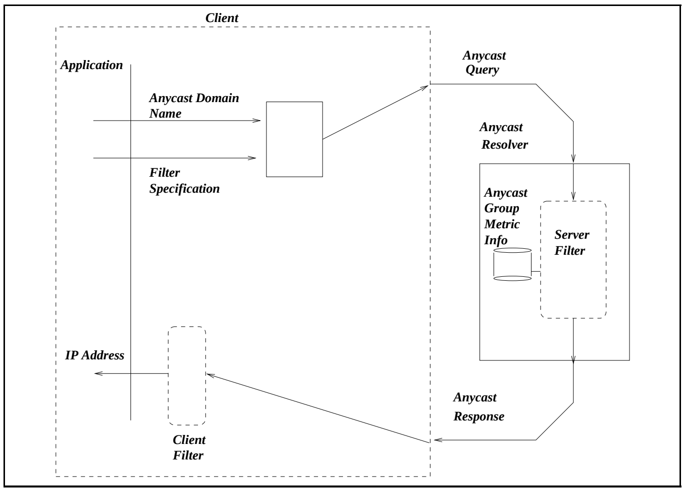

FIGURE 1: Anycast name resolution query/response cycle

straightforward to integrate into the existing infrastructure. An anycast domain name (ADN) is of the form `<Service>%<Domain Name>`. 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. Each network location is preconfigured with the address of its local anycast resolver, which resolves queries and/or consults the authoritative resolver, as in DNS.

An anycast resolver maintains the information necessary to perform the mapping from ADN to IP address. In particular, it maintains the performance information associated with each member of the anycast group. This information is maintained independently at each anycast resolver that has the ADN group membership information cached. Because of the local significance of the metric information, metrics maintained at the authoritative resolver are, in general, of little value to other resolvers. The authoritative resolver may provide its locally maintained metric information whenever it receives a request from another resolver for the anycast group member list for a given ADN. Local resolvers can use this information as "hints" initially as they begin to gather their own metric information. It is this gathering of metric information that is the focus of this paper.

In our proposed approach the anycasting service is accomplished through a set of filters that are applied to the information maintained about the anycast group to obtain an IP address. We distinguish between several types of filters in our general architecture. For the purposes of this paper the relevant filter is the metric-based filter which picks the server with the best (numerically lowest or highest) metric value. We also propose in [2] some approaches that can be used to specify the filter.

3 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 how to find the desired resource over the network knowing only its name or property. Many techniques were proposed and investigated. These include: 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., [5, 6]) , 2) the use of various name server architectures in order to lookup the location of the resource (e.g., [7, 8, 9]) and 3) the use of caching of a resource's location (not content) at sites where the resource is frequently accessed [10].

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 being placed on servers resulting in 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. Two notable studies in this area are: 1) the original work by Partridge, Mendez and Milliken [4] proposing the idea of anycasting and discussing its network-layer support and 2) a recent study by Guyton and Schwartz [11] which addresses the problem of locating the nearest server. The latter work also presents a classification of "best"-server location schemes. The work is related to earlier work on the `Harvest` system [12] 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. Another related project is the SONAR network proximity service [13] in which the authors define a service which can return the closest (in hops) server from among a list presented to it.

By choosing to define anycasting as a communication paradigm, we deviate somewhat from the Guyton and Schwartz classification which considers the original definition of anycasting as a network-layer-supported service. Our work investigates the complete design of application-layer anycasting and not just the metric probing aspect as discussed in [11]. We also consider using a variety of metrics (not just hop-distance as provided by SONAR) in order to provide a closer match to the application requirement.

Recent work by Carter and Corvella [14, 15] has also addressed 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 [15]).

Remote measurement and monitoring of system performance has been explored as part of the extensive work on distributed system monitoring [16, 17].

3.1 Service Location Protocols

The Service Location Working Group of the IETF has been developing protocols to facilitate the discovery and selection of network services [3]. Thus, the high level objectives of their work and ours are quite similar. However, their focus differs considerably from ours, and the solutions we and they have developed are mostly complementary. We briefly describe their focus, then contrast our contributions with theirs.

The current focus of the Service Location Working Group is on selecting network services within an enterprise network, not the global Internet. One of the goals is to ease network system administration, by supporting dynamic discovery of services in new environments. While selection of the "best" among replicated servers is possible, it does not appear to be the primary motivation for this work. The protocols are well-suited to services such as printers or file servers, with long-lived attributes (e.g., location of a printer). The attribute values are updated via a push from a Server agent co-located with the Service and deposited at a Directory. While not explicitly prohibited, it is not at all clear whether path-dependent attributes could be incorporated within the attribute update protocol.

In contrast, our focus is on the development of metric collection techniques, and the evaluation of the system performance in a wide-area network. We have deliberately chosen a very simple application programmer interface, requiring only a modification to `gethostbyname()`. We have also focused on a single type of "server," namely an IP address. Despite the differences in implementation, our performance evaluation should provide important information as the Service Location group expands their protocols to include a global internet [18].

4 Metric Collection

4.1 Metrics of Interest

As mentioned in the Introduction, the focus of this paper is on maintaining the metric information needed to select amongst replicated web servers. The particular metric we consider is the *response time experienced at the client*, measured from just prior to sending a query to just after receiving the complete response. With information about the size of the response data, this time could also be used to compute server-to-client throughput.

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

4.2 Goals and Constraints

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. 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.

4.3 A Proposed Technique

Server Push and Client Probe: To build a metric collection technique meeting the goals and constraints outlined above, we combine two well-known methods for collecting performance data in a distributed system.

In the *server push* technique [19, 20], the server monitors its performance and pushes this information to the resolvers when interesting changes are observed. For additional scalability, the update information can be network-layer multicast to all resolvers that maintain information about the server. The server can control the network traffic generated by this mechanism 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 measured.

In the *probing* technique, periodic queries are made to the servers to determine the performance that a client would experience. These queries must be designed to have performance that is similar to legitimate client requests. For scalability, these queries should be made from probing agents (possibly co-located with resolvers) that collect probe data relevant to a large number of clients. A clear tradeoff exists between the accuracy of the probe data and the number of probing agents. The advantages of this technique are that it measures network path performance, and it does not require server modification; on the other hand, the load on the network and servers may be significant.

Note that probing gives the most accurate estimate of what the probing client should expect 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: In general, 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. Note that the task of updating link state in a distributed routing environment has precisely the same criteria. We have adopted the link state update algorithm used in the ARPANET [21]. 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. (Note that 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.

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 for each access the time from just after the server assigns the process until just before the server does the first read. These measured values are averaged and smoothed (as described in Section 5) before being used in the algorithm described above.

Client 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 time 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.

The probing of servers is performed by *probing clients* that would normally be co-located with the resolver but may also be running at other locations. We choose to logically separate the two. The probe client is server protocol dependent, and if incorporated within the resolver, the resolver would have to be aware of all the protocols of all the servers in its database. Also note that each probing client is acting as a proxy for real clients within a certain region. The farther away a client is (in Internet "distance") from a probing client, the less representative are the probe measurements of the clients experience.

A Hybrid Push/Probe Technique: We combine the time 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. We further ask the server to include in the well-known file the most recent time value measured as described above. Let S denote the server time value reported in the file during the most recent probe.

Inbetween 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

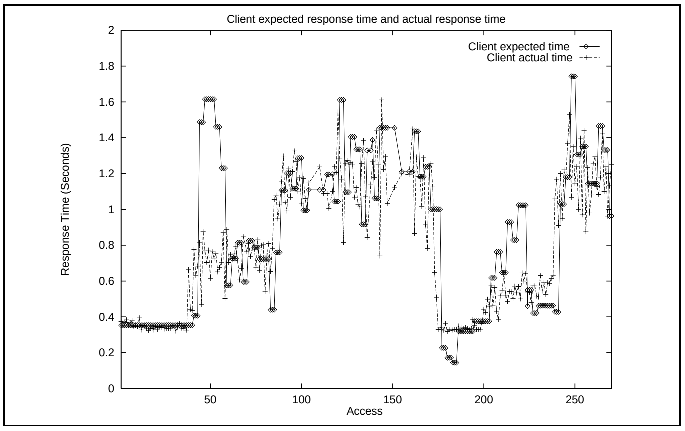

FIGURE 2: Response time estimation using proposed metric collection technique

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.

Experimental Evaluation: To determine the accuracy of the metric collection technique described, we experimented with various locations and capabilities of servers and resolvers, variations in server load, and alternative file sizes. Figure 2 shows a typical result, plotting the estimated and actual values of response time over 270 queries. In this particular experiment, the client was at the University of Maryland, and the server was located 12 Internet hops away at Georgia Tech. The client made requests to the server according to an access log file from a real server. The x-axis indicates the index of each query. 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. The client probed for the well-known file every fifty accesses.

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. As we will see later, this relative accuracy is sufficient for our purposes.

Avoiding Oscillations: 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

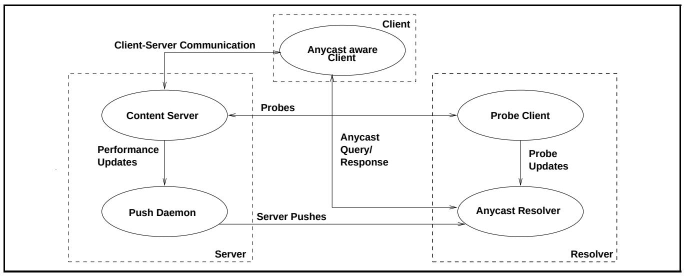

FIGURE 3: Software components of the application-layer anycasting architecture

within a threshold of best performance. This set of equivalent servers is recomputed every time a new pushed value is received at the resolver. The resolver picks at random from this set of equivalent servers to answer an anycast query. The set ES is computed according to the following algorithm:

When new push or probe measurement is received:
Compute the new estimated response time for server;
Sort servers according to estimated response time;
$ES = ES \cup \{$Server with lowest response time estimate$\}$;
$R_{top} =$ minimum response time estimate;
For each server $j \in ES$
  if ($\mid R_j - R_{top} \mid > \tau_\ell$)
    then $ES := ES$ - {j};
For each server $j \notin ES$ (except those just removed in above step)
  if ($\mid R_j - R_{top} \mid \leq \tau_j$)
    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 Section 5, 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 examined.

5 Experimental Evaluation

In this section we describe a set of experiments that we conducted over the Internet, using our proposed anycasting architecture and metric collection techniques. We first describe the experimental setup, then discuss results from experiments to assess the metric collection technique and 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 3.

Anycast-aware Servers: We modified the Apache HTTP [2] daemon to act as a performance monitoring

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

server as described in Section 4. 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 one of the files maintained on disk and an amount of data equal to the size of the original file is read and transmitted. 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 measure file that the server shares with the performance push daemon. The performance push daemon executes the update algorithm given in the Section 4 and passes the average server performance data through a low pass filter [3] to achieve some smoothing. 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 clients. 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 [22].

Clients: We modified the NCSA Mosaic [4] client to have the desired behavior. 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 to generate client requests. As described above, the file sizes are distributed to the servers. By reading the access log file, the client calculates the *relative* time for each access (with respect to a specified initial access time). 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.

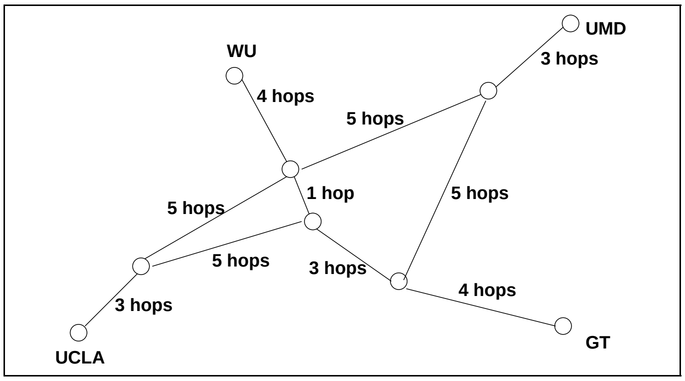

FIGURE 4: Experimental topology

Anycast Resolver and Probing Client: For the probing client 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 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 performance monitoring 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 4.)

5.2 Evaluating Selection Technique

In this section, we describe several experiments to evaluate the performance of the server selection technique described in Section 4.3.

We have 20 clients running: 16 at Georgia Tech, Atlanta and 4 at the University of Maryland, College Park. The clients are divided into groups of 5. 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 which tests 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.

[3] The precise algorithm at the push daemon is as follows. The push daemon monitors server performance at intervals of length I time units, starting with interval 1. At the end of interval i, the push daemon averages the data recorded by the server (say avg_i) during the i-th interval. Performance P_i at this time is determined to be $P_0 = 0, P_i = \alpha(avg_i) + (1-\alpha)P_{i-1}$. We set α to be 0.9.

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

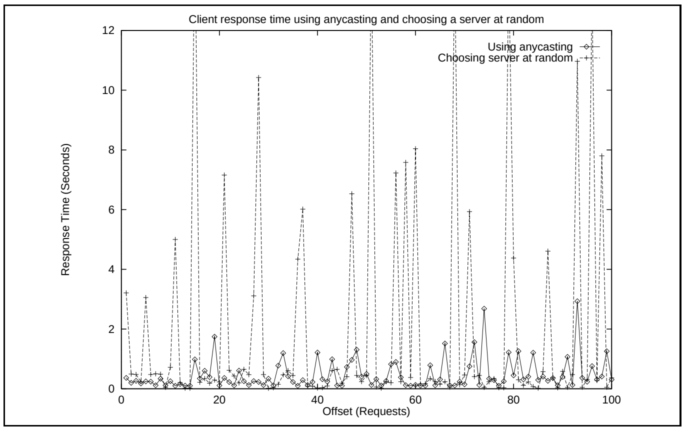

FIGURE 5: Comparison of performance of a client in all anycasting and all random experiment

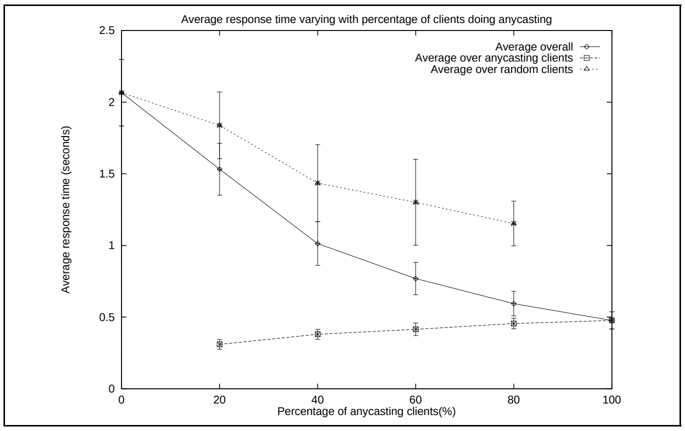

FIGURE 6: Response time varying with percentage of clients using anycasting

In Figure 5, we compare the performance experienced by a client in two settings. In one of them, the clients all use random server selection; in the other, the clients all use anycasting for server selection. In the anycasting case, the probe client accessed the servers once every 4 minutes, and 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, over a sequence of 100 requests. 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 1: Performance of server location schemes

Table 1 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. In addition to the random and anycasting selection methods, the table also includes the nearest hop count method. We note that while always choosing nearest server improves upon random selection, another factor of two improvement is possible with the 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 6, 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. We note that average response time over all clients decreases from 2.1 to 0.5 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 to 0.5 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 6 is the confidence interval for each point with 90% confidence level. The confidence intervals of response time of 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 better, on average, than the random clients across the full range of the experiment.

In Figure 7, we analyze the performance of the anycasting system for varying values of probe and push frequencies. All clients are using the anycasting mechanism. The average response time is recorded over a period of 30 minutes for a range of push and probe frequency values. 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. 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

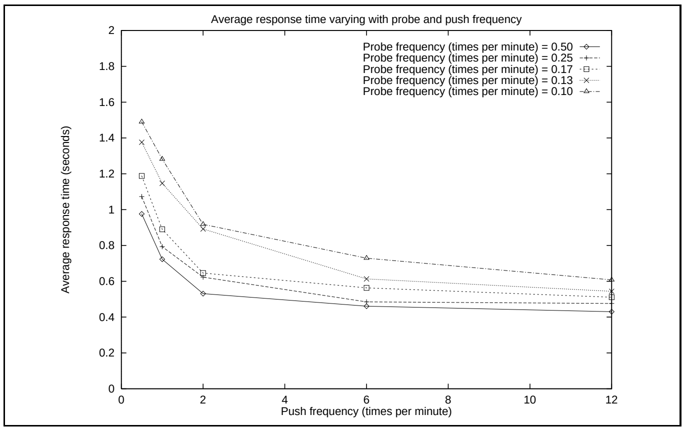

FIGURE 7: Response time varying with push and probe frequency

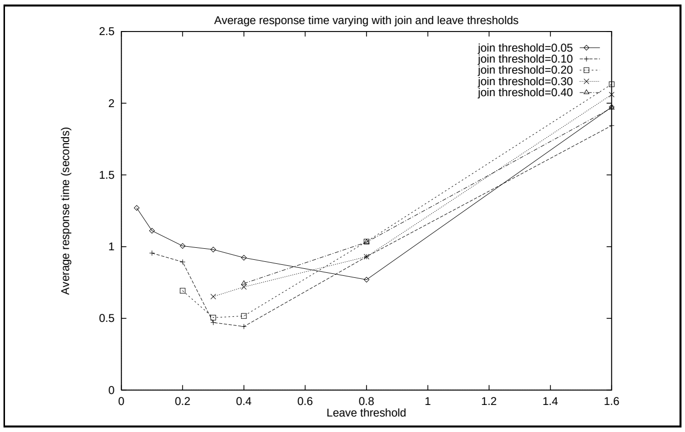

FIGURE 8: Response time varying with join and leave thresholds

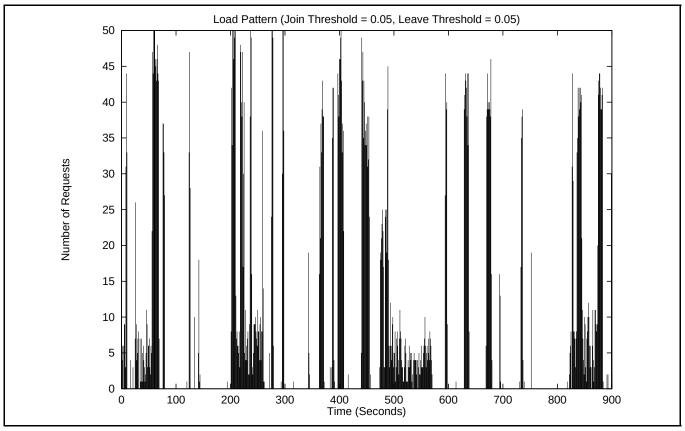

FIGURE 9: Server load with join threshold = 0.05 and leave threshold = 0.05

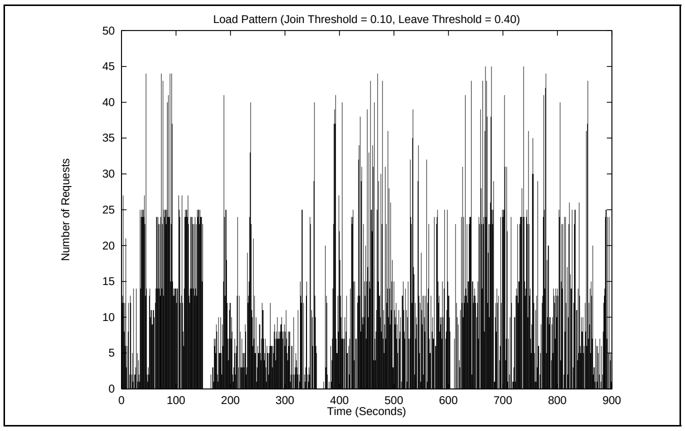

FIGURE 10: Server load with join threshold = 0.10 and leave threshold = 0.40

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. When the thresholds are high, the addition process is aggressive, while the removal process is conservative. Figure 8 shows the results of our evaluation. 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 all random case. This is because the server will never leave the equivalent group after joining it. The resolvers actually 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 9. It shows the number of requests for each 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 10 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 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. They are the costs due to the push daemon, the probe client, the resolver, and client anycast query. The client anycast query 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 the 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 being done infrequently, this will not represent a significant burden on servers.

6 Concluding Remarks

The efficient utilization of a set of replicated servers hinges upon the ability to appropriately allocate servers to clients. In this paper we derive a technique for such allocation. Our goal is to allocate a server that would minimize a client's response time. Although designed to work within our Application-Layer Anycasting architecture, the technique has wider applicability and complements on-going server location efforts within the IETF. We focus in our development on HTTP servers but many of the ideas are applicable to other kinds of services. Our architecture targets an environment in which replicated servers can be widely distributed over the Internet.

Our proposed approach is centered around estimation of a client's expected response time at each server. Such estimation is performed by a resolver that acts as a proxy for a number of closely-located clients. The most interesting aspect of our design is its ability to combine a relatively light-weight server push approach with a client-probe approach to produce useful estimates of response time. Measured path-independent server performance (the pushed data) is calibrated using path-dependent response time measurements obtained via relatively infrequent probes.

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 the significant response time improvement that can be achieved with this technique over the use of random, or other performance-independent allocation mechanisms.

References

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

[2] S. Bhattacharjee, M. H. Ammar, E. W. Zegura, V. Shah, and Z. Fei, "Application layer anycasting," in *Proceedings of INFOCOM 97*, 1997.

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

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

[5] 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*, vol. 27, pp. 1429–1445, 1995.

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

[7] P. Mockapetris, "Domain names – concepts and facilities," *RFC 1034*, November 1987.

[8] I. Gopal and A. Segall, "Directories for networks with casually connected users," in *Proceedings of INFOCOM 88*, pp. 1060–1064, 1988.

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

[10] D. Terry, "Caching hints in distributed systems," *IEEE Transactions on Software Engineering*, vol. 13, pp. 48–54, January 1987.

[11] J. Guyton and M. Schwartz, "Locating nearby copies of replicated Internet servers," in *Proceedings of SIGCOMM 95*, pp. 288–298, 1995.

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

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

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

[15] R. L. Carter and M. E. Crovella, "Dynamic server selection using bandwidth probing in wide-area networks," Tech. Rep. BU-CS-96-007, Computer Science Department, Boston University, Boston, MA, 1996.

[16] B. Schroeder, "On-line monitoring: A tutorial," *IEEE Computer*, vol. 28, pp. 72–78, June 1995.

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

[18] J. Rosenberg, H. Schulzrinne, and B. Suter, "Wide area network service location," *Internet Draft (work in progress) draft-ietf-svrloc-wasrv-01.txt*, November 1997.

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

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

[21] E. C. Rosen, "The updating protocol of arpanet's new routing algorithm," *Computer Networks*, no. 4, pp. 11–19, 1980.

[22] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: The search for invariants," in *Proceedings of SIGMETRICS'96*, ACM Press, 1996.