RaDaR: A Scalable Architecture for a Global Web Hosting Service

Michael Rabinovich
AT&T Labs
180 Park Ave
Florham Park, NJ 07932
misha@research.att.com

Amit Aggarwal
Department of Computer Science
University of Washington
Seattle, WA 98195
amit@cs.washington.edu

Abstract

As commercial interest in the Internet grows, more and more companies are offering the service of hosting and providing access to information that belongs to third-party information providers. In the future, successful hosting services may host millions of objects on thousands of servers deployed around the globe. To provide reasonable access performance to popular resources, these resources will have to be mirrored on multiple servers.

In this paper, we identify some challenges due to the scale that a platform for such global services would face, and propose an architecture capable of handling this scale. The proposed architecture has no bottleneck points. A trace-driven simulation using an access trace from AT&T's hosting service shows very promising results for our approach.

Keywords: Hosting service, scalable architecture, dynamic replication, migration.

1 Introduction

As commercial interest in the Internet grows, more and more companies are offering hosting services *i.e.* the service of hosting and providing access to objects belonging to third-party information providers. Successful hosting services may in the future host millions of objects on thousands of servers deployed around the globe, resulting in *global information-hosting systems*.

Hosting services commonly use replication, or *mirroring*, to cope with load on popular web sites and to reduce bandwidth consumption in their backbones. Currently, mirroring decisions are done by system administrators, who monitor the demand for information on their sites and decide what content should be replicated and where. Making these decisions is a difficult task, which will become even more daunting as the scale of these systems (and hence the decision space) increases.

A related problem with manual mirroring decisions is that they are necessarily fairly static - resources allocations are reviewed and changed infrequently. This implies that resource allocation has to be done for the worst-case demand scenario. However, practically unlimited number of potential clients for any given object and demand that may vary greatly over time make worst-case resource allocation very wasteful. Considering the large number of objects hosted on a system of the scale we envision, the cost of the worst-case approach clearly becomes prohibitive. Further, not only changes in demand, but any change in the computing environment (addition or removal of servers or network elements, bringing a server down for maintenance, etc.) would require re-considering resource allocation, potentially for the entire system. Without appropriate new technology, system administration and costs related to object placement and resource allocation may become a factor limiting the scale of hosting platforms.

In this paper, we propose an architecture for global information hosting systems that can handle their expected scale. Our approach, the RaDaR (Replicator and Distributor and Redirector) architecture, is based on dynamic object replication and migration. While the idea of dynamic replication is by no means new, the context of the Internet presents important new challenges.

First, two factors must be taken into account in a single framework: server load (it is desirable to balance the load among the servers), and the proximity of clients to servers (it is desirable to place replicas in the

proximity of clients from which most of the requests originate). Moreover, to be scalable, the algorithms for making these decisions can rely only on a limited amount of information about the system.

Second, much of traditional work in dynamic replication has concentrated on protocol aspects, without considering the architecture. We look at the problem as a whole, paying special attention to architectural issues. In fact, specifics of the Internet environment affect the algorithms as well. For instance, choosing to rely on current off-the-shelf routers and TCP implementations rules out approaches where routers interpret web requests and participate in request distribution or replica placement decisions [18]. Recently, there have been some architectural proposals for dynamic web replication, which we discuss later in the related work survey.

Another challenge is administering changes in the system itself, such as adding or removing hosting servers. In a system that contains thousands of servers and spans multiple administrative domains, it is important that the effect of such changes be localized.

Overall, the paper proposes a coherent architecture for distributing load and reducing bandwidth consumption in a global hosting system. The main features of this architecture are the following:

1. The architecture scales with no bottlenecks. In other words, growing load (both due to the increasing number of clients accessing objects and the number of objects hosted by the system) can be handled by adding more off-the-shelf processing nodes rather than increasing the processing power of any single node.

2. The architecture provides a flexible environment for deploying different dynamic replication and request distribution algorithms. In particular, `RaDaR` uses an algorithm that takes into account both server load and client-server proximity when deciding on the number and placement of replicas as well as when choosing a replica to satisfy a particular request. Replica placement is dynamic and may change in response to changes in demand, network topology, or server sets. This not only makes the architecture *administratively* scalable, but also results in better utilization of available resources.

3. Dynamic replica placement makes expanding the system very easy administratively: new hosts are added with no content, and the system automatically decides which objects to migrate to the new servers.

4. Information hiding is used to keep the system manageable despite its scale, by exploiting the hierarchical division of the Internet into administrative units (*autonomous systems* and *OSPF areas* [19]). In particular, host additions and deletions are localized to the affected OSPF areas; that is, no other parts of the system need to be informed.

5. The system is completely transparent to the end-user and does not require any changes on the client side.

A limitation of our system is that complicated objects that require extensive customized environments to process user accesses (e.g., popular search engines) are difficult to migrate. Fortunately, these objects are unlikely to use a public hosting service. Typically, such a service is used to host static and simple dynamic pages, and the environment they need to execute can be reproduced on or migrated to every server.

Finally, dynamic replication involves some overhead as a result of creating new replicas and migrating existing ones. A very high overhead could make a dynamic replication system impractical. For instance, since dynamic replication uses prior accesses in deciding replica placement, there is the potential danger of chasing ever changing demand. In other words, the system would relocate objects based on prior accesses, only to find itself in a suboptimal state again because the demand has changed. Object relocation would only add overhead in this case. One of the contributions of our paper is to show using a real trace that this clearly does not happen. Trace driven simulation on a trace consisting of web accesses to servers hosted by AT&T's hosting service shows that our system reduces bandwidth consumption by about 52% while imposing only a low traffic overhead of below 5% of (already reduced) total traffic.

The rest of the paper is organized as follows: section 2 describes the architecture of `RaDaR`. Section 3 briefly presents the replica placement and request distribution algorithms. Section 4 presents some of the synchronization issues. Section 5 contains details about the simulation and the performance results. In

Figure 1: An example of an internetwork.

this paper, we stress on the architecture. Detailed evaluation of the algorithms, including comparison with alternate static and dynamic replication schemes is presented in [1]. Section 6 describes future work, section 7 describes related work and section 8 contains our conclusions. The Appendix gives details on how network proximity information used in our approach can be derived from routing data.

2 RaDaR Architecture

2.1 The System Model

We assume the Internet environment [19], where individual destinations (nodes and networks) are organized into *OSPF areas* (called *areas* for short), which in turn are organized into *autonomous systems, ASs* (see Figure 1). There is also a proposal to organize autonomous systems into *federations*. Our architecture extends directly for more levels in the hierarchy.

We assume that the hosting platform is distributed globally across multiple ASs. In particular, a message between two nodes belonging to the system may traverse third-party autonomous systems en-route. As an example, in Figure 1, the hosting system contains autonomous systems AS1, AS2, and AS3. Autonomous system AS1 comprises areas AR1, AR2, and AR3. A message from a node in AS1 to a node in AS2 must travel through external routers r1 and r2.

Nodes, areas and ASs belonging to the hosting system will be called *internal* nodes, areas and ASs respectively. Other nodes, areas and ASs will be referred to as *external*. Internal nodes that maintain objects and service client requests are called *hosting servers* or *hosts* for short. For simplicity, we will assume homogeneous hosts. Heterogeneity can be introduced by incorporating weights corresponding to relative power of hosts into the protocol.

We assume the existence of a uniform load measure that allows load comparison of different servers. The length of the ready queue (e.g., the output of the *uptime* command in UNIX) can be used as a metric for the computational load [22]. The load metric may use multiple components; for instance, storage utilization, the number of open connections, etc. With multiple load components, a server is considered overloaded when any component exceeds its threshold. Finally, we assume that a server can estimate the fraction of its total

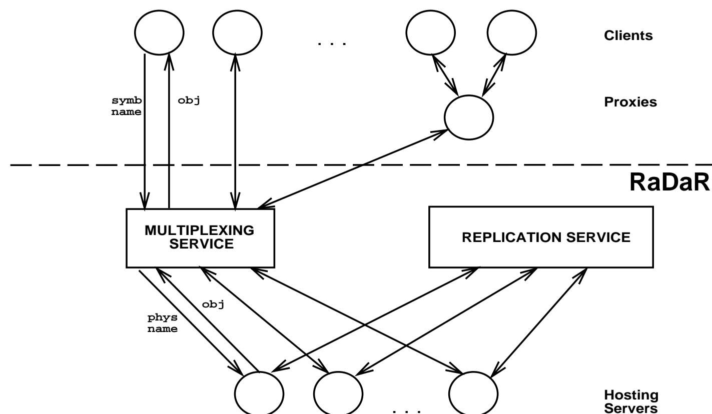

Figure 2: A high-level view of RaDaR.

load due to a particular object on it. For the storage component, object load can be obtained directly; for the computational components, this can be done by keeping track of resource consumption (CPU time, IO operations, etc.) due to requests for individual objects and dividing up the total load proportionally between objects based on their individual consumptions.

Note that a brute-force approach would be to replicate every object on every hosting server. As we argued in the introduction, we consider this approach to be too wasteful for a hosting service of the scale we envision. Thus, we exclude this solution by assuming that a capacity threshold is always reached on any server before all objects can be placed on it.

2.2 The High-Level View

A high-level view of the system is shown in Figure 2. It consists of a *multiplexing service* which maintains a mapping between the *physical* and *symbolic* names of an object. The physical name is the name by which an object is accessed internally by the system. If hosting servers are accessed using HTTP, this would be the replica's *physical URL*. However, the system is free to use a more efficient proprietary protocol to access hosts. The symbolic name (or the *symbolic URL*) is the name used by external web clients to access the object. Due to replication, a symbolic name corresponding to a particular object may map onto multiple physical names. When an object is created, it is placed on one of the hosting servers and is *registered* with the multiplexing service. The registration involves informing the multiplexing service of the physical name of the object and assigning it a symbolic name. The host name portion of any symbolic URL resolves into the multiplexor's identity. Thus, a request for any object is initially directed to the multiplexing service. For a given request, the multiplexing service chooses a corresponding physical name based on host load and client location (section 3.2 describes the request distribution algorithm in greater detail), fetches the object from the corresponding host, and forwards it to the client.

The decision on the number and placement of object replicas is taken by hosting servers in a distributed manner. Each hosting server collects access statistics for each of its objects and periodically decides whether to drop any of its objects, migrate them to other hosts, or create additional replicas on other hosts (Section 3.1 describes the replica placement algorithm). Once a host decides to replicate or migrate an object, the target host is chosen in cooperation with the *replication service*. The replication service also keeps track of all hosts in the system and their load.

2.3 The Replication Service

Replica placement is done in cooperation between hosts and the *replication service*, which is implemented as a *replicator hierarchy* (Figure 3). There is one replicator in each RaDaR OSPF area, one in each RaDaR autonomous system, and one *root* replicator. The main motivation for the hierarchical replication service is to

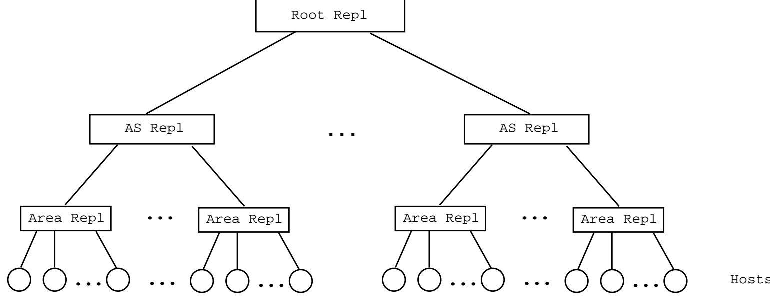

Figure 3: The replicator hierarchy.

hide information about areas and autonomous systems behind their replicators and to make administration of the system manageable, as described below.

The Hosts

A host knows only about other hosts within its OSPF area, OSPF areas within its autonomous system, and other autonomous systems. So, when choosing candidates for migrating or replicating its objects, the host does so in terms of these *entities* it knows about. Once the entity is chosen, the host sends the replication or migration request to the replicator representing this entity. The replicator then chooses a host within its region for placing the object.

Our heuristic for replica placement is based on the routes messages take from the source to the destination.[1] The reason for choosing this metric is our service-centric perspective, with the health of the platform (i.e., backbone traffic and server load) being our primary concern. For a given request that arrived at server s, let $s \rightarrow r_1^1 \rightarrow \ldots \rightarrow r_n^1 \rightarrow r_1^2 \rightarrow \ldots \rightarrow r_m^2 \rightarrow A_1 \rightarrow \ldots \rightarrow A_k \rightarrow c$ be the router path from s to client c that issues the request, where $\{r_i^1\}$ are routers within s's area, $\{r_i^2\}$ are routers in s's AS but outside s's area, and $\{A_i\}$ are ASs other than s's AS. We will refer to a path $h_1 \rightarrow \ldots \rightarrow h_n \rightarrow AR_1 \rightarrow \ldots \rightarrow AR_m \rightarrow AS_1 \rightarrow \ldots \rightarrow AS_k \rightarrow c$, as the *preference path* of the request, where h_i is the closest host to r_i^1, AR_j is the closest internal OSPF area to r_j^2, and AS_q is the closest internal AS to A_q.

Hosts compute preference paths based on the information periodically extracted from the system's routers (see Appendix for details on how this is done). Each host then autonomously chooses candidate entities to migrate or replicate its objects to, based on preference paths of prior requests. An entity that appears frequently on the preference paths of an object would be a good candidate because the object often passes near this candidate on its way to the clients.

Besides improving the client-server network proximity, another reason for a host to replicate or migrate its objects to other hosts is to reduce its load. So, if a host is overloaded and no objects can be migrated or replicated out for proximity reason, the host needs to find *any* under-loaded host to shed some of its load to. To this end, the host sends an *offload* request to its own replicator, which will either find an under-loaded host in its region or forward the request up to its parent in the replicator tree.

The Replicators

A replicator r maintains the following state. First, it keeps the identity h_{min} and load estimate ($load_{min}$) of the least-loaded host among the hosts in its region. It also maintains the average load of the hosts in its region (av_load), referred to as r's average load. Finally, for each child replicator r_i, r maintains the estimate of the average load of r_i's region ($av_load(r_i)$). This state is maintained using periodic report messages from lower-level replicators to their parents.

Each replicator receives periodic load reports from its children and sends its own report to its parent. The report from a host to its area replicators contains its load. Each replicator then computes and sends to its parents the average load of the hosts in its region, the number of hosts, and the load and IP address of

[1] Although different messages between a given pair of nodes can take different routes every time, in practice these routes are usually the same or very similar [26].

its least-loaded host. The next-level replicators then have enough information to compute their state and reports for their parents.

When a replicator receives a request for object placement, it forwards this request to its child replicator with the lowest average load. Eventually the request reaches a leaf (a host), which fulfills it. If any replicator on the request path has average host load exceeding a threshold, the request is denied.

When a replicator receives an *offload* request, it checks if either the average host load of a child replicator, or the load of the least-loaded host is below a threshold. In the first case, the replicator forwards the request down the replicator tree; in the second case, it sends the request directly to the least-loaded host. Otherwise, it cannot fulfill the request and sends it up to its parent, which performs the same actions. If the root cannot handle the request, the request is denied - the system has no hosts whose load allows accepting additional objects.

The replicator uses the least-loaded host in choosing the path for forwarding the offload request because all its child replicators may have average loads over the threshold and yet contain some under-loaded hosts beneath them. Without considering the least-loaded host, a situation could arise where no offloading ever occurs even though there are under-loaded hosts in the system.

In summary, each replicator maintains a fixed amount of information per child replicator, and communicates only with its child and parent replicators to maintain this state. The hierarchical replication service localizes host additions and deletions, which require only the replicator representing the affected OSPF area to be informed. More details on host addition and deletion are provided later.

2.4 The Multiplexing Service

This section describes a scalable architecture for the multiplexing service, which is a potential bottleneck in the global hosting platform of Figure 2.

An important architectural decision is to choose between *iterative* and *recursive* request processing. A multiplexing service can either return a *redirect* response to the client with the physical URL of an object replica, or it can obtain and forward the document to the client. The first alternative (which we call iterative, in analogy with the DNS terminology), reduces the load on the multiplexing service and allows clients to cache name resolutions for future use. The second approach, called recursive, hides the identity (and even existence) of hosts from the clients and eliminates an extra message exchange with the client. We chose the second approach. In addition to the above reasons, it allows the multiplexing service to cache especially hot objects and use more efficient proprietary protocols to communicate with hosts.

In principle, the functionality of the multiplexing service described here could be folded into the DNS infrastructure. The disadvantage of the DNS-based approach is that (1) it is inherently iterative and thus reveals host identities to clients and (2) it stipulates that replica placement be done at the granularity of entire web sites. The advantage is that it leverages the existing DNS infrastructure.

The Interface

The multiplexing service must support the following requests:

1. `RegisterObject(symbolicName,physicalName)`. This request comes from a system administrator upon creating a new object. The multiplexing service makes sure that the symbolic name is unique and creates the initial name mapping.

2. `GetObject(symbolicName)`. This request comes from a client. The multiplexing service resolves the symbolic name into one of the physical names, fetches the chosen object replica and forwards it to the client.

3. `CreateReplica(symbolicName, physicalName)`. This request comes from a hosting server when an additional replica of the object is created. The name service must add the `physicalName` to the set of physical names corresponding to `symbolicName`. The multiplexing service must also acknowledge that the above operation has succeeded.

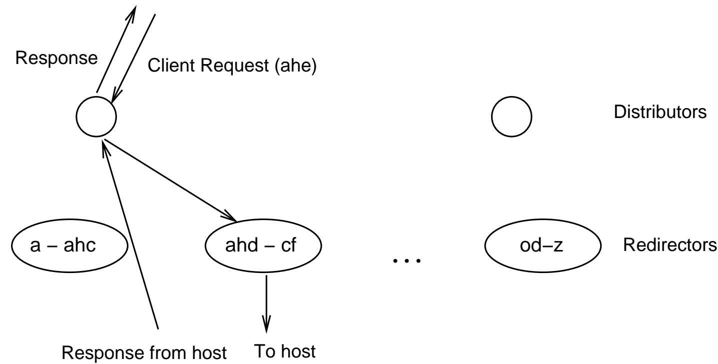

Figure 4: The multiplexing service.

4. `DeleteReplica(symbolicName, physicalName)`. This request comes from a hosting server when a replica with `physicalName` is deleted. The multiplexing service must remove `physicalName` from the set of physical names corresponding to `symbolicName`, unless this is the last physical name in the set (in which case the multiplexing service returns a negative acknowledgment to the sender). The multiplexing service must also send a positive acknowledge to the sender of the request if the above operation succeeds.

5. `CheckOut(symbolicName)` and `CheckIn(SymbolicName,stagedName)`. These requests come from the system administrator when he or she wants to modify an object. The `CheckOut` method creates a temporary replica of the object for modification and returns the physical name of this replica (referred to as "stagedName"). The `CheckIn` method returns the modified object to the multiplexing service, which propagates the update to all physical replicas.

The Architecture

The scale of the system suggests that the multiplexing server be highly replicated for scalability and availability. Further, since every object is routed through the multiplexing service to the client, we would want it to be replicated so as to increase its proximity to the clients. However, the high frequency of updates to the mapping database due to replication and migration of objects requires that the number of replicas be kept low, since updating a large number of replicas is impractical.

To solve this dilemma, we split the functionality of the multiplexing server between two components (Figure 4). The *redirector* maintains the mapping from symbolic names to physical names, which is potentially very dynamic due to frequent migrations and replications. Scalability is achieved by partitioning the symbolic name space among redirectors based on some hash function. (The simplest way is to partition names lexicographically, as shown in the figure). The second component consists of *distributors*, which store the hash function mapping the symbolic names to the corresponding redirectors. This function is fairly static allowing the distributors to be highly replicated, for scalability and increasing proximity to clients.

A client request first arrives at one of the distributors. The distributor is chosen with the goal of increasing its proximity to the client. The distributor sends the request to the appropriate redirector by applying the hash function to the symbolic name in the request. The redirector chooses a physical replica of the requested object (e.g., the closest replica to the distributor that sent the corresponding request) and forwards the requests to the replicas' host. The host sends the object directly to the distributor, which forwards it back to the client.

Note that routing requests through the (potentially remote) redirectors does not affect the backbone traffic significantly, since the requests are typically small UDP messages. Any additional traffic they generate is more than offset by savings due to serving content from hosts that are close to the distributors that, in turn are close to the clients. While the latency is more affected by the request detour, the savings still outweigh the extra delay due to the detour (section 5).

The issue that remains to be addressed is how a client finds a distributor to send a request to. We rely

```
for each document x_s /* x_s is replica of document x on host s */
    if cnt(x_s) < U /* cnt(x_s) is the number of accesses to x_s */
        delete x_s
    else
        Attempt to migrate x_s to farthest host c st
        cand_cnt(c) > MIGR_RATIO × cnt(x_s)
        /* cand_cnt(c) is the number of preference paths
        for x_s containing c */
    If migration not successful
        if cnt(x_s) > M
            Attempt to replicate x_s to farthest candidate c st
            cand_cnt(c) > REPL_RATIO × cnt(x_s)
```

Figure 5: Replica Placement Algorithm

on the DNS infrastructure to locate distributors. The DNS server(s) of the hosting platform can map the domain names of all web sites hosted by the system into the set of IP addresses of distributors. In response to a client query, the DNS server will send the IP address of a distributor that is the closest in the network to the client. (See [9] and, commercially, [11] for implementations of DNS name resolution based on the origin of the requester.) Note that this usage of the DNS service conforms to the assumption behind the existing DNS infrastructure that the mapping of DNS name into IP addresses changes slowly. (Unlike sets of replicas for a given object, the set of distributors changes infrequently). Thus, existing caching mechanisms will keep the load on the DNS service manageable. Alternatively, the *anycast* service [25] could be used for this purpose when it becomes available. In any case, the existence of the multiplexing service is completely transparent to the clients. In particular, clients can establish HTTP1.1 persistent connections with the distributor, just as they would with the origin servers directly.

3 Algorithms

The `RaDaR` architecture provides a scalable and flexible framework for implementing dynamic replica placement and request distribution algorithms. The replication service provides the ability to place an object on any host without having complete knowledge of all the hosts in the system. Moreover, all the hosts have access to preference paths which facilitate replica placement decisions. The multiplexing service allows scalable request distribution based on per-object state information.

Various algorithms for replica placement and request distribution can be implemented in such an environment. [1] presents detailed evaluation of a number of algorithms. In this section, we present a particular scheme which has been used in our simulation study.

3.1 Replica Placement Algorithm

The replica placement algorithm has two goals - to distribute load and increase client server proximity. Client server proximity is increased by migrating or replicating objects to candidates which appear on a large number of preference paths for the object. Load is distributed by ensuring that objects are created only at those hosts whose load is below a certain threshold. Moreover, the offloading mechanism provides the ability to unload documents when the load on a particular host increases above the threshold (section 2.3). During offloading, objects are migrated even if it is not advantageous from the proximity point of view.

Figure 5 shows the replica placement algorithm which is executed at each host periodically. A document is deleted if its load is below the deletion threshold U and it is not the sole replica in the system. If the load is above U, the document is migrated to a candidate which occurs on a majority of the preference paths (i.e., $MIGR_RATIO > 0.5$). This heuristic ensures that migration improves proximity. In case migration fails,

then we attempt to replicate the document to a candidate which occurs on at least $REPL_RATIO$ of the preference paths. The condition for replication is much weaker than that for migration ($REPL_RATIO < MIGR_RATIO$). Replication is done only if the replica load is greater than the replication threshold M, so that the benefit of replication outweights its cost.

3.2 Request Distribution Algorithm

The request distribution algorithm also attempts to distribute load and increase client server proximity. A fundamental choice for the algorithm is whether to use load feedback for taking request distribution decisions or not. [1] describes and compares the two approaches in detail. In this paper, we focus on the non-feedback approach because of its ability to predict the effect of an object movement, which is essential for any dynamic algorithm. A redirector sends a client request to the closest replica such that the following invariant is maintained: *The access count of the most heavily loaded replica is at most a constant times the count of the most lightly loaded replica.* Thus, the redirector maintains a count of the number of requests sent to each replica of an object. The invariant makes it possible to predict the effect of adding or deleting a replica of an object. In particular, we can derive load bounds for a host after the set of object on it changes. This has a number of desirable effects - accurate load estimates facilitate load balancing; documents can be moved en-masse without waiting for load statistics after each move; placement decisions can be evaluated leading to more optimal replica placement. Finally, the scheme does not suffer from stability problems due to feedback delay or global synchronization. See [27] for a detailed description of the non-feedback approach and the derivation of the load bounds.

4 Synchronization Issues

4.1 Creation and Deletion of Replicas

Creation and deletion of replicas must be coordinated with modification of the name mappings at the multiplexing service. Otherwise, there may be periods of inconsistency between the mapping database at the multiplexing service and the actual replica sets. Such inconsistencies may lead to a situation where the multiplexing service resolves a symbolic name to the physical name of a replica that no longer exists.

One could avoid such inconsistency by running replica deletion and mapping update as one distributed transaction, and name resolution as another. This, however, would lead to interference of load distribution activity with user requests. The protocol for deleting and creating replicas in `RaDaR` avoids distributed transactions by maintaining an invariant that the set of physical names to which a symbolic name is mapped always forms a subset of currently valid replicas. To this end, when a new replica is created, the mapping on the appropriate redirector is modified only after the replica has been created. When a replica is to be deleted, the hosting server first requests the redirector to exclude this replica from the mapping. Only after obtaining confirmation that the redirector successfully did so, does the hosting server delete the replica. In addition, the confirmation from the redirector includes a count of the number of requests that were directed to the replica being deleted. This is useful in order to prevent the replica from being deleted before serving all the requests that have already been directed to it. Object migration is accomplished by replica creation on the recipient node followed by replica deletion on the source node. This ensures that symbolic names are always resolved to valid replicas.

4.2 Addition and Deletion of Hosts

An important feature of `RaDaR` is that it makes administering a global-scale system manageable. Adding a host requires simply pointing the host to its area's replicator. The "empty" host will start sending its load reports to the replicator, and the replicator will gradually fill the host with objects as it receives object placement requests.

Removing a host is also easy. The administrator instructs the host to prepare for being removed. The host starts acting as if it were overloaded: its load reports to its replicator always contain a high value so that no new objects are ever placed on the host. The host also sends *offload* requests to the replicator until no

objects remain, at which point the host notifies the administrator that it can be turned off. Unless all other hosts in the system are overloaded (which is an exceptional case requiring extensive manual intervention), the host will empty itself automatically.

4.3 Replica Consistency

Multiple studies (e.g., [24, 15]) have shown that a large majority of web object accesses are to static objects. These objects, as well as dynamic objects that do not change as a result of user accesses, can be replicated freely. In addition, objects whose per-access updates are commutative can also be replicated if a mechanism is provided for merging updates recorded by different replicas.

Of course, objects may also change as a result of updates by the content provider. However, the overall rate of these modifications is negligible compared to the access rate, so it should not play a role in replica placement decisions. Consistency of these updates can be maintained by using the primary copy approach [2]. The location of the primary copy can be tracked by the object's redirector.

Finally, there are objects that change their state as a result of user accesses, with non-commutative changes. These objects can be freely migrated, but their replication strategy must depend on the mix of read-only and update accesses. Some ideas from database research are relevant to this problem (e.g., [32]). Given a small number of these objects, we can assume for simplicity that they can only be migrated.

5 Performance Results

We have simulated the `RaDaR` system and evaluated it using a trace of accesses to servers hosted by Easy-WWW, AT&T's low-end hosting service. This paper concentrates on the performance comparison with a centralized hosting system, reflecting the current architecture of EasyWWW. We compare various dynamic replication approaches to static replication in [1], which also studies the performance effects of changing various parameters in the algorithms.

5.1 Simulation Model

Our simulation study uses the EasyWWW trace, which is a log of HTTP requests to servers hosted by AT&T's hosting service. The trace covers a three month period, contains around 111 million requests to more than 100 sites and 0.2 million documents. Note that these are accesses reaching the server after being filtered through any browser or proxy caches that clients might have used. In order to make the simulation efficient, we pruned uninteresting documents whose peak access rate was below 0.01 requests per second. This led to a 72% reduction in the number of documents while only a 1.3% reduction in the number of requests and 1.1% reduction in total number of bytes requested.

For simulation, we used the 14-node Worldnet backbone topology. A few of these nodes are border routers (or *IGR*s) through which external traffic enters the Worldnet backbone. The remaining nodes serve as entry points for traffic from AT&T customers and are called *POP*s (*Points of Presence*). The topology consists of a single AS and OSPF area. This implies that the replicator hierarchy collapses into two levels - the root replicator with all the 14 nodes as its children. We assume that each of the 14 nodes contains a hosting server. Further, there is a distributor co-located with each of the nodes. We placed a single redirector at the "central" node, whose average distance to all other nodes in the network is the smallest.

Client requests are directed to the distributor that is co-located with the exit point of traffic from Worldnet backbone to the client. Determining this exit point is easy for Worldnet customers since each customer is connected to a unique Worldnet node. For external client, the exit point is determined by using the BGP tables; inconclusive BGP data are resolved using geographical proximity information [1].

The sizes of the various documents were obtained from the trace. Initially all the documents were placed at a particular host (called the *data center*), which is similar to what is done in reality today. Our dynamic replication algorithm then migrated and replicated objects to other nodes in the system. Response latency is computed as the sum of link latencies on the request and response paths, the document transfer time and the delay at the hosting server used. The latency and bandwidth of various links in the topology were determined using delay measurements for various packet sizes. The host delays were obtained assuming that

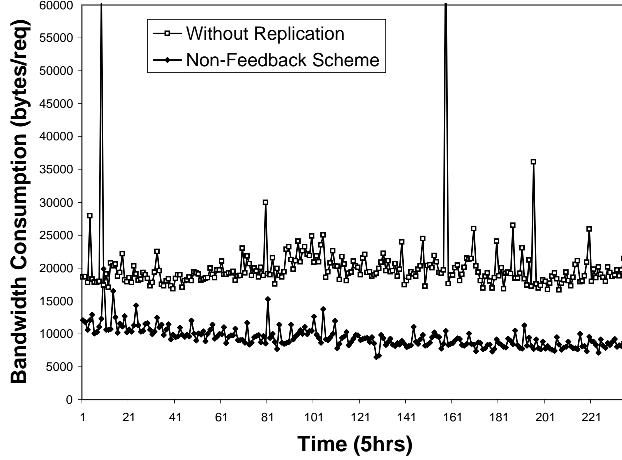

Figure 6: Bandwidth consumption.

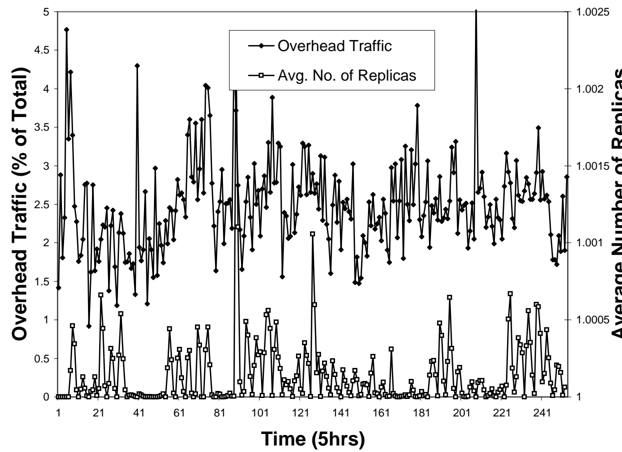

Figure 7: Overhead traffic and average number of replicas.

they process requests in FIFO order, with a fixed service time for each request. While this model for latency is not as realistic as for our bandwidth results, it suffices to compare our approach with the existing platform.

5.2 Performance

Figure 6 plots the average bandwidth consumption per request using our system as opposed to the case when there is no replication. The bandwidth consumption is computed as the sum of data transmitted over all the links for every request. With dynamic replication, the bandwidth requirements reduce by upto 52% as compared to the case when there is no replication. Moreover, our algorithm is able to smooth out bursts in bandwidth demand. Most of the bandwidth improvement is due to the increased client-server proximity. Client latencies show an improvement of about 13%. The modest latency reduction is due to the extra communication with the redirectors. However, as already pointed out, being a server-centric architecture, the bandwidth is a more important concern. Figure 7 shows that the algorithm imposes a low traffic overhead of less than 5% of already reduced total traffic. Further, the number of replicas created is very small.

Although the average number of replicas in the system at any time is small, it could happen that the maximum resource requirement at all the hosts is very high. In such a case, dynamic replication would be no better than worst case static allocation. Further, a system administrator managing a RaDaR system faces the challenge of determining the processing and storage capacity of each of the hosts offline. We simulated

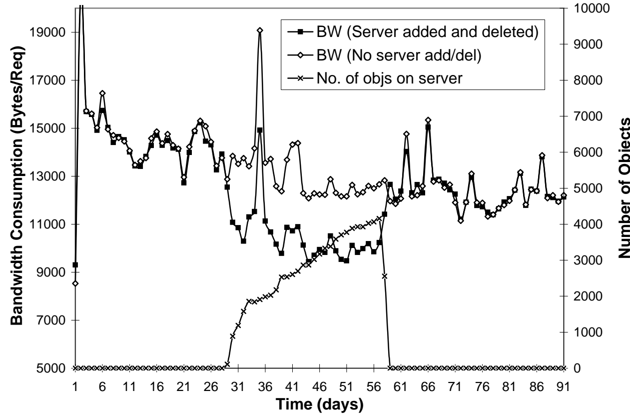

Figure 8: Effect of addition/deletion of a server

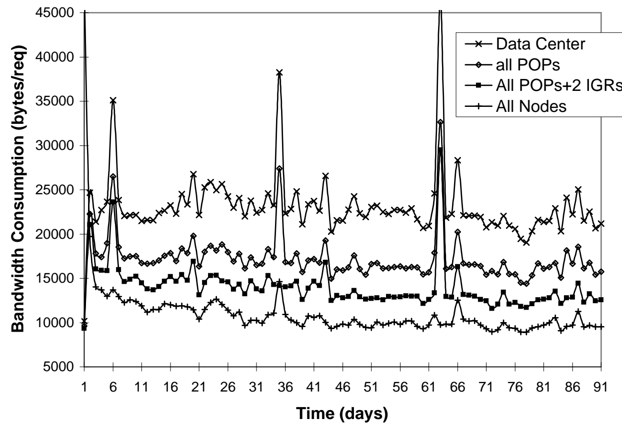

Figure 9: Effect of adding resources to the system

a small part of the trace to determine the resource requirements on each of the hosts offline. Using these estimates, we limited the resources in the system, and simulated the remaining trace. Our results show that most of the performance gain can be achieved with a very low resource overhead of about 14% *ie.* we get a performance improvement of over 50% by using only 14% extra storage as compared to the case when there is no replication.

One of the features of `RaDaR` is the ease with which hosts can be added and removed from the system. In order to demonstrate that the system adjusts to changes in the set of hosting servers, we conducted an experiment in which one of the *IGR* nodes did not initially contain a hosting server, and then its hosting server was added to the system and later removed. Figure 8 plots the bandwidth consumption for the system as well as the number of objects on the server. When the server is added to the system, documents are migrated to it, thereby improving the performance as compared to the case when the server is not added. During the time that the server is up, the system performance improves by as much as 19% on the average. Moreover, the improvement in performance is visible almost as soon as the server is added. This indicates that, while the added server continues to accumulate objects for a long time, it acquires the most valuable objects quickly. When the server is removed, all the objects on it are offloaded and the performance deteriorates again.

The architecture of `RaDaR` uses replication and partitioning for scalability and preventing bottlenecks. Additional load can be handled by adding new distributors or further partitioning the symbolic name space among redirectors. A related and important question concerns the scalability of the dynamic replication algorithm *ie.* does performance scale with the available resources. In order to demonstrate that our system

effectively utilizes all the available resources, we incrementally added servers to the system and compared the performance. Figure 9 plots the bandwidth consumption for various configurations of the hosting service. As the number of servers available for hosting objects increases, the performance of the system improves. This is because the added resources are utilized for increasing client-server proximity by the system.

6 Future Research

We view the work presented in this paper as a starting point that should provide an infrastructure for future work in policies and protocols for a global hosting service. We now mention a few areas for future research.
Accounting for Link Load and Capacity. Our current approach relies on proximity metrics used in routing messages. One could also attempt to account for bandwidth and congestion of network links. Several methods have been proposed to probe the network to determine its available bandwidth (e.g., [10]), which could provide input for these decisions.
Replication Strategy. In this paper, we adopted a replication strategy that starts with a small number of object replicas and then creates additional replicas as the demand for certain objects grows. An alternative strategy could be to always have as many replicas as possible, and then drop replicas when the demand for space grows (due to hosting new objects or the need to create additional replicas of the hottest objects). It would be interesting to study the tradeoff between these strategies.
Object Granularity. Our simulations assumed that the placement of each Web page can be considered separately. The other extreme is to consider placement of entire Web sites. Various intermediate object granularities are also possible. A finer granularity imposes higher overhead for collecting and maintaining access statistics and for placement decisions. Coarser granularity may increase the overhead for transferring objects between sites. In some cases of dynamically generated pages, a group of files, executables, and other environment state *must* be migrated together, since they are used together for page generation. The cost of transferring such bundled objects may be high and should be taken into account in a replica placement algorithm. A language for specifying these bundled objects must be designed. Marimba's DRP protocol is an example of work in this direction [13]. In addition to manual bundling of objects, some objects could conceivably be bundled together automatically based on similarity of access patterns. The policies for (and the feasibility of) doing this is an issue for future work.

7 Related Work

Existing commercial products offer transparent request distribution among a fixed set of server replicas. So-called load-balancers use a a front-end IP multiplexing device (see [14] for a survey of some of these products) to balance load among Web servers on a local area network. Several approaches use DNS servers for the same purpose [21, 12]. CISCO Distributed Director [11], IBM Network Dispatcher [20], and WindDance Web Challenger [31] allow request distribution over a wide-area network. None of these systems offer dynamic object replication or migration.

ArrowPoint's Content Smart Switch allows multiplexing requests among replicas on a per-page basis [3]. While their white papers mention dynamic content replication, no mechanisms for performing it are described.

Our approach to use routing databases to determine proximity information is similar to CISCO Distributed Director. However, unlike CISCO, we extract this information asynchronously with client requests, thereby reducing request latency at the expense of potential staleness of the proximity information. The reason for using the routing-based proximity metric is our network-centric view, in which reducing the backbone bandwidth is an overriding concern. Guyton and Schwartz describe various other methods that can be used to determine proximity of nodes in the network [16].

Several research proposals for server selection in a wide-area network have been described (e.g., [8, 28, 10, 29]). All these proposals focus on replica selection and do not consider the dynamic replication.

Heddaya and Mirdad have proposed a *WebWave* protocol for replicating Web objects from the content server up the router tree towards the clients [18]. The works of Bestavros [7] and Bestavros and Cunha [6] appear to be the predecessors of WebWave. WebWave achieves almost perfect load balancing among replicas.

However, it burdens the Internet routers with the task of maintaining replica locations for web objects and intercepting and interpreting requests for Web objects. It also assumes that each request arrives in a single packet. As the authors note, this protocol cannot be deployed in today's networks.

Approaches by Yoshikawa et al [33] and Baentsch et al [5] perform replica selection at the client. In [33], the URL of an object actually points to a Java applet, which embeds the knowledge about the current replica set and the procedure for replica selection. This approach requires an extra TCP communication to download the applet. The approach of [5] propagates information about replica sets to clients in HTTP headers. It requires changes to clients (proxy servers in this case) to process extra headers and to implement replica selection.

Gwertzman and Seltzer [17] motivate the need for geography-based object replication. They propose to base replication decisions on the geographical distance (in miles) between clients and servers. This measure may not correctly reflect bandwidth consumption for fetching an object, since the network topology often does not correspond to the geographical distances.

The problem of placing objects in the proximity of requesting clients has also been addressed in research on file allocation (see [23] for an early survey and [4] and references therein for more recent work). In particular, Awerbuch, Bartal, and Fiat have proposed a distributed file allocation protocol that it is nearly optimal in terms of total communication cost and storage capacity of the nodes [4]. However, their protocol does not address the load balancing issue. Moreover, architectural issues are not considered.

8 Conclusion

This paper proposes a scalable architecture for providing access to information on a global network. While ISPs work hard on increasing the bandwidth of their backbones, they still cannot catch up with just as fast-growing demand. In fact, we are on the verge of another qualitative jump in Internet load levels, as current slow modems, which act as flood-gates limiting user access to the Internet, are replaced with much faster alternatives like ISDN lines and cable modems. Moreover, with practically unlimited number of clients to access an information server, it is very hard to design a server for the worst load case. Thus, the approach adopted in our architecture involves monitoring the load of a large pool of servers and migrating or replicating objects among them to distribute their load and to move replicas closer to the requesting clients. A scalable mechanism that accomplishes this is proposed. Our system has a number of desirable features. It is scalable, which means that increase in load can be handled by adding additional hardware. We use dynamic migration and replication in order to distribute the load and increase client-server proximity. Dynamic replication, apart from being administratively scalable, utilizes the available resources more effectively. The system uses information hiding for higher scalability and manageability. Finally, it is completely transparent to the end user. Experiments on a real trace show that our system improves performance by as much as 52% while imposing a low overhead of only about 5%.

References

[1] A. Aggarwal and M. Rabinovich. Performance of replication schemes on the Internet. Technical Report HA6177000-981030-01-TM, AT&T Labs, October 1998. Also available as http://www.research.att.com/~misha/radar/tm-perf.ps.gz.

[2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2d International Conference on Software Engineering*, pages 627–644, October 1976.

[3] ArrowPoint Comunications. `http://www.arrowpoint.com`.

[4] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. In *Proc. of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 574–583, January 1996.

[5] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. Enhancing the web infrastructure – from caching to replication. *IEEE Internet Computing*, pages 18–27, Mar-Apr 1997. Also available at http://neumann.computer.org/ic/books/ic1997/pdf/w2018.pdf.

[6] A. Bestavros and C. Cunha. Server-initiated document dissemination for the www. *Bulletin of the Computer Society Technical Committee on Data Engineering*, 19:3–11, September 1996.

[7] Azer Bestavros. Demand-based document dissemination to reduce traffic and balance load in distributed information systems. In *Proc. IEEE Symp. on Parallel and Distr. Processing*, pages 338–345, 1995.

[8] S. Bhattacharjee, M. Ammar, E. Zegura, V. Shah, and Z. Fei. Application-layer anycasting. In *INFOCOM*, 1997.

[9] H.-W. Braun and K. C. Claffy. An experimental means of providing geographically oriented responses relative to the source of domain name server queries. Technical report, San Diego Supercomputing Center, April 1994.

[10] Robert L. Carter and Mark E. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. In *Proceedings of Infocom-97*, 1997.

[11] Cisco Systems, Inc. DistributedDirector. White paper. http://www.cisco.com/warp/public/734/distdir/dd_wp.htm.

[12] M. Colajanni, Ph. S. Yu, and D. M. Dias. Scheduling algorithms for distributed web servers. In *Proc. 17th IEEE Intl. Conf. on Distributed Computing Systems*, May 1997.

[13] The http distribution and replication protocol. A WWW Consortium submission, August 1997. http://www.w3.org/TR/NOTE-drp.

[14] R. Farrell. Distributing the web load. *Network World*, pages 57–60, September 22 1997.

[15] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *INFOCOM-99, to appear*, 1999.

[16] J. Guyton and M. Schwartz. Locating nearby copies of replicated Internet sarvers. In *Proceedings of SIGCOMM'95*, 1995.

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

[18] A. Heddaya and S. Mirdad. Webwave: Globally load balanced fully distributed caching of hot published documents. In *Proc. 17th IEEE Intl. Conf. on Distributed Computing Systems*, May 1997.

[19] C. Huitema. Routing in the Internet. Prentice Hall, 1995.

[20] IBM Interactive Network Dispatcher. http://www.ics.raleigh.ibm.com/netdispatch/.

[21] E. Katz, M. Butler, and R. McGrath. A scalable Web server: the NCSA prototype. *Computer Networks and ISDN Systems*, (27):155–164, September 1994.

[22] O. Kremlin and J. Kramer. Methodical analysis of adaptive load sharing algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 6(3):747–760, November 1992.

[23] Q. Kure. *Optimization of File Migration in Distributed Systems. PhD Dissertation.* University of California (Berkeley), 1988.

[24] Stephen Manley and Margo Seltzer. Web facts and fantasy. In *Proceedings of the Symposium on Internet Technologies and Systems*, pages 125–133. USENIX, December 1997.

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

[26] V. Paxson. End-to-end routing behavior in the Internet. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 25–39, August 1996.

[27] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A dynamic object replication and migration protocol for an Ineternet hosting servece. Submitted for publication, 1998.

[28] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek. Selection algorithms for replicated web servers. In *Workshop on Internet Server Performance*, June 1998.

[29] S. Seshan, M. Stemm, and R. Katz. SPAND: shared passive network performance discovery. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pages 135–146, 1997.

[30] R.W. Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley, Reading, MA, 1994.

[31] WindDance Networks Corp. Webchallenger. http://www.winddancenet.com/webchallenger/products/frame_products.htm.

[32] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems (TODS)*, 22(4):255–314, June 1997.

[33] Chad Yoshikawa, Brent Chun, Paul Eastham, Amin Vahdat, Thomas Anderson, and David Culler. Using smart clients to build scalable services. In *1997 Annual Technical Conference, January 6–10, 1997. Anaheim, CA*, pages 105–117. USENIX, January 1997.

Appendix: Obtaining Proximity Information

This section describes how information required by our architecture can be obtained in the context of today's Internet. Specifically, we answer the following questions:

1. How do redirectors find the closest host having a replica of the requested object. We use hop count as the metric of distance, which corresponds directly to the notion of closeness used by routers. As routers become more sophisticated and start using more elaborate metrics (e.g., link bandwidth, link congestion, usage fees), these metrics will be reflected in the router databases and will be picked up by our proximity functions automatically. Moreover, using hop counts is justified by our server-centric approach in which bandwidth reduction is the primary motivation.

2. How the host receiving a client request obtains the preference path for this request. While the `traceroute` tool [30] can give the router path information, it generates prohibitively high network traffic to be used for routine statistics collection.

The answer to these questions would differ depending on the protocols considered. To be specific, we will assume that BGP is used to route IP messages between autonomous systems (AS) of the Internet, and OSPF is used to route IP messages within an autonomous system, the most common (and recommended) open routing protocols [19].

8.1 Internet Routing Model

Under BGP/OSPF, the whole Internet is divided into administrative domains called *autonomous systems (AS)*, identified by unique *AS numbers*. Routing within an AS uses the link state protocol. Thus every node in the AS knows the shortest distance to every other node in the same AS. Traffic between ASs is handled by the *border routers* using a path-vector protocol with number of ASs as the cost metric. Each border router knows its best possible path (in terms of ASs) to all other ASs in the Internet. A node which wants to send a packet to a node outside its AS chooses the border router which has the shortest path length to the destination AS. In case multiple border routers have the same shortest distance, then the one closest to the sender is chosen.

The following information can be obtained from the routing tables:

- BGP tables inside an AS X provide information about the best border router for sending packet to any other AS A. We denote this by $BorderRouter(A, X)$. Also $BGP_hops(A, X)$ gives the number of AS hops between X and A.
- The OSPF routing tables provide the shortest distance $OSPF(i, j)$ between any two nodes i and j in the AS.

8.2 Host Proximity

Definition 1 (Distance)
The distance between a node host and any other node dest is given by the tuple:
$\{BGP_hops(AS(dest), AS(host)), \min\{OSPF(r, host) | r \in BorderRouters(AS(dest), AS(host))\}\}$ *where* $AS(h)$ *denotes the AS of the host h.*

In agreement with Internet routing, which first uses BGP to route a message to the destination's autonomous system and then OSPF to deliver it within the autonomous system, we assume that distance (d_1, d_2) is greater than distance (d'_1, d'_2) if either $d_1 > d'_1$ or $d_1 = d'_1$ and $d_2 > d'_2$.

The redirectors use the above definition to find the distance between two nodes. Computing the distance from a node to any other node just involves querying the BGP and OSPF tables. For a given client request, the redirector computes the distance of the distributor to all the hosts that contain a replica of the requested object. The minimum distance determines the closest host for the request. Note that all the distances can be pre-computed and the routing tables need not be queried on every request. Also note that the distance is computed from the distributor and not the client, since all responses pass through the distributor.

8.3 Preference Paths

Definition 2 (Preference path)
Let $s \to r_1^1 \to \ldots \to r_n^1 \to r_1^2 \to \ldots \to r_m^2 \to A_1 \to \ldots \to A_k \to c$ *be the router path from internal node s to external client c, where* $\{r_i^1\}$ *are routers within s's area,* $\{r_i^2\}$ *are routers in s's AS but outside s's area, and* $\{A_i\}$ *are ASs other than s's AS. We refer to a path* $h_1 \to \ldots \to h_n \to AR_1 \to \ldots \to AR_m \to AS_1 \to \ldots \to AS_k \to c$*, as the* preference path *of the request, where* h_i *is the closest host to* r_i^1*,* AR_j *is the closest internal OSPF area to* r_j^2*, and* AS_q *is the closest internal AS to* A_q*.*

For every external client, a host can compute the router path to it using the routing tables. More specifically, the OSPF tables can be used to compute the route within the host's AS (from the host to the border router). The border router in the path is determined by the BGP tables ($BorderRouter(A, X)$). For ASs outside the host's AS, path vector information from the border router is used.

The closest AS to a given AS can be computed using the information in the BGP tables. Similarly the closest host and area to a give router can be computed using the OSPF tables. This enables the hosts to determine the preference path corresponding to a router path.

Vitae

Michael Rabinovich is a Principal Technical Staff Member in the Database Research Department at AT&T Labs. His interests include distributed systems, Web performance, and transaction management. He holds a PhD from the University of Washington.

Amit Aggarwal is a graduate student in computer science at the University of Washington. He received a B.Tech. in computer science from the Indian Institute of Technology, Delhi. His current interests include operating systems, networks and wide-area distributed systems.