AT&T Labs Research

subject: **Dynamic Replication on the Internet**
Work Project No. 3116-17-7006

date: **March 5, 1998**

from: **Michael Rabinovich**
Dept. HA6177000
FP A107
973-360-8778
misha@research.att.com
HA6177000-980305-01TM

Irina Rabinovich
The Dun and Bradstreet Corp.
Rabinovich2@mail.dbs.com

Rajmohan Rajaraman
DIMACS Center, Rutgers University
rraj@dimacs.rutgers.edu

TECHNICAL MEMORANDUM

This paper proposes a protocol suite for dynamic replication and migration of Internet objects. It consists of an algorithm for deciding on the number and location of object replicas and an algorithm for distributing requests among currently available object replicas. Our approach attempts to place replicas in the vicinity of a majority of requests while ensuring at the same time that no servers be overloaded. The request distribution algorithm uses the same simple mechanism to take into account both server proximity and load, without actually knowing the latter. The replica placement algorithm executes autonomously on each node, without the knowledge of other object replicas in the system.

The proposed algorithms rely on the information available in databases maintained by Internet routers. A simulation study using synthetic workloads and the network backbone of UUNET, one of the largest Internet service providers, shows that the proposed protocol is effective in eliminating hot spots and achieves a significant reduction of backbone traffic and server response time at the expense of creating only a small number of extra replicas.

1 Introduction

Replication, or *mirroring* in Internet parlance, is commonly used to address a scalability problem of popular Internet sites. Currently, mirroring is done manually by system administrators, who monitor the demand for information on their sites and decide what data should be replicated and where. This daunting task is especially difficult and important for a growing number of companies that offer a *hosting service*, i.e., the service of maintaining and providing access to objects belonging to third-party information providers. As the scale of these systems increases (in terms of the number of objects and hosting servers), the decision space for replica placement increases while a brute-force worst case design becomes prohibitively expensive. Without appropriate new technology, system administration related to object placement may become a factor limiting the scale of hosting platforms.

In this paper, we propose a suite of algorithms that dynamically replicate Internet objects in response to changing demand. In addition to automating the placement decisions by system administrators, automatic dynamic replication would make the system more responsive to changes in the demand patterns.

Two fundamental issues any dynamic replication system must address are deciding on the number and placement of replicas and distributing requests among object replicas. In our protocol, replica placement decisions are made by each node autonomously, based on the requests serviced by the node and without any knowledge of other replicas of the same objects that might exist. In fact, each node has very limited knowledge about other hosts that exist in the system. This is essential for the intended scale of the system. For instance, this makes adding or deleting hosts in the system a localized administrative action.

Our request distribution algorithm takes both server load and proximity into account without knowing the actual load of the servers. To illustrate the problem, a simple round-robin request distribution would distribute the load among all replicas but would be oblivious to the proximity of requesters to servers. On the other hand, always directing requests to the closest replica would create problems when a server is swamped with requests originating from its close vicinity: no matter how many additional replicas the server creates, all requests will be sent to it anyway.

Some existing protocols address this problem by collecting load reports from hosting servers and weighing server loads into the proximity-based request distribution. This approach in not well suited for dynamic replication on a global scale (which is evidenced by the fact that products following this approach, e.g., CISCO's Local Director, assume fixed replica sets). First, in the system of this scale, the request re-direction subsystem will be highly distributed. Therefore, each hosting server would have to send its load report to a large number of redirecting servers, increasing the network traffic and the delays in reaching all the redirectors. Second, and more importantly, request distribution for a given object becomes dependent on the popularity of many other objects co-located at the same hosting servers, greatly complicating autonomous replica placement decisions.

In our protocol, a simple deterministic request distribution algorithm allows a hosting server to derive bounds on the number of requests going to potential new replicas based on local knowledge only. Using these bounds has two main advantages. First, it allows a host to decide on object migration and replication autonomously. Second, it enables object relocation to be done *en mass*, without waiting for new access statistics after each move. Without this, a system of our intended scale would be hopelessly slow in adjusting to demand changes.

1.1 Past Work

There is a growing number of commercial products that offer transparent load balancing among multiple Internet sites [11, 22, 40] . They differ in the network level where indirection of requests

to physical replicas occur: CISCO's Distributed Director performs re-direction at the DNS level (a similar idea is used in [23]), while IBM's Net Dispatcher and CISCO's Local Director re-direct at the front-end router level, and Winddance's Web Challenger does it at the application level using redirection features of the HTTP protocol. None of these products offer dynamic replication or migration of replicas.

An up-coming product *Web Repeater* from the Sandpiper start-up company [35] redirects every request to the closest host, and replicates the object to this host. The hope is that there will be other requests from the vicinity of that host, which would amortize replication costs. Such aggressive replication may result in significant performance degradation when the above assumption does not hold. An even more aggressive replication approach is described in [38] in the context of "push caching", where a single object access results in creating copies of the object at multiple proxy caches.

Much of existing work on dynamic replication has concentrated on quorum-based replica management. This work focuses on algorithms for adjusting quorums when replica sets change (e.g., [1, 21, 33]) and on regenerating object replicas in place of failed ones (e.g., [27]). In contrast, our work employs replication and migration for performance, and assumes that asynchronous update propagation and not quorums is used for consistency.

Existing protocols for performance-motivated dynamic replication rely on assumptions that are unrealistic in the Internet context. Wolfson et al [42] propose an *ADR* protocol that dynamically replicates objects to minimize communication costs due to reads and writes. Given that most of Internet objects are rarely written [1], this is not a suitable cost metric for the Internet. In addition, the protocol imposes logical tree structures on hosting servers and requires that requests travel along the edges of these trees. Because of a mis-match between the logical and physical topology, and especially because each node on the way must interpret the request to collect statistics (which requires in practice a separate TCP connection between each pair of nodes), this would result in impractically high delay of request propagation.

Heddaya and Mirdad's *WebWave* dynamic replication protocol [19] was proposed specifically for the Web. 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.

Algorithmically, both ADR and WebWave decide on replica placement based on the assumption that requests are always serviced by the closest replica. Therefore, neither protocol allows load sharing when a server is overloaded with requests from its vicinity. Objects are replicated only between neighbor servers, which would result in high delays and overheads for creating distant replicas, a common case for mirroring on the Internet. Also, ADR requires replica sets to be contiguous, making it expensive to maintain replicas in distant corners of the global network even if internal replicas maintain only control information.

The works of Bestavros [7] and Bestavros and Cunha [8] appear to be the predecessors of WebWave. [7] proposes to reduce network traffic within an intranet by caching organization's popular objects close to the intranet's entry point. In our context, any backbone node in the hosting platform can be such an entry point. Our protocols address the problems of choosing entry points at which to place object replicas and allocating requests to those replicas. These questions are not considered in [7] . In [8] , Bestavros and Cunha discuss the benefits of replicating popular objects from the host server up the request tree. No algorithms are described.

Baentsch et al [6] propose an infrastructure for performing replication on the Web, without describing algorithms for deciding on replica sets. Also, the infrastructure assumes gradual learning

[1] Recent trace studies (e.g., [28]) consistently show that 90% of requests are to static objects, and many of the remaining objects are dynamically generated responses to read-only queries.

of the replica set by clients, which may hurt the responsiveness of the system. Our protocol is complimentary to this work. Gwertzman and Seltzer [18] 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 communication costs 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 [26] for early survey and [3, 4, 5] for more recent work). Early work in this area assumes a central point where decisions on object placement are made by solving an integer programming optimization problem. Even when the search space is heuristically pruned, the scale of our application would make such approaches impractical. Also, this approach requires the decision-making point to have complete information on network topology, server loads, and demand patterns. However, it would be an interesting question for future work to see how much worse the performance of our protocol is compared to the optimal placement obtained by solving the global integer programming optimization problem.

More recently, the problem of obtaining distributed solutions for the file allocation has been addressed [3, 4, 5] . In [4] , Awerbuch, Bartal, and Fiat design a distributed file allocation protocol and use the framework of competitive analysis [36] to show that their protocol is nearly optimal in terms of total communication cost and storage capacity of the nodes. However, they do not address the issue of load balancing among different servers. Moreover, while their work is significant from a theoretical standpoint, issues concerning implementation of their protocol over the Internet are not addressed.

1.2 Our Contributions

Our main contributions are as follows.

- We present a new protocol for dynamic replication on the Internet by integrating an algorithm for deciding on the number and placement of object replicas with an algorithm for distributing requests among replicas. The request distribution algorithm uses the same simple mechanism to take into account both the proximity of requests to servers and server load, without actually knowing the latter. The replica placement algorithm relies on the knowledge of how requests would be distributed among potential replicas in making placement decisions autonomously on each node, without even knowing of the existence of other replicas.

- Our protocol relies on practical assumptions derived from the existing Internet technology. We use information available from routing databases and IP headers to drive the protocol. No new functionality is required from the routers.

- System responsiveness to changes in demand patterns is one of the explicit design goals. Responsiveness is improved in two ways. First, unlike existing work, our protocol performs replication between distant nodes directly, without replicating with all intermediate nodes on the path. Second, based on bounds on the load of the new object replicas, the protocol relocates multiple objects at once, without waiting for new access statistics after each move. Given the intended scale of the system, a protocol without this feature could be so slow that by the time it arrives to a desired replication scheme, the demand pattern may change again.

- The protocol employs information hiding to keep the system manageable despite its scale. It uses the hierarchical division into administrative units (autonomous systems and OSPF areas) supported by the Internet for this purpose. In our protocol, each administrative unit has a representative (called a replicator). Each replicator maintains a fixed amount of information

per each child replicator. A host knows only about other hosts within its lowest-level administrative unit, the replicators on the path from itself to the root of the replicator tree, and the siblings of those replicators. This arrangement localizes host additions and deletions and reduces network traffic for finding lower-loaded hosts when offloading of a host is needed.

A simulation study using synthetic workloads on the backbone topology of UUNET, one of the largest commercial Internet Service Providers, shows that our protocol is effective in eliminating hot spots among servers and achieves a significant reduction of backbone traffic and server response time at the expense of creating only a small number of extra replicas. The protocol also showed acceptable responsiveness to changes in demand patterns.

2 The System Model

We assume the Internet environment [20] , where individual destinations (nodes and networks) are organized into *OSPF areas* (called below just *areas* for short), which in their turn are organized into *autonomous systems, ASs* (see Figure 1). There is also a proposal to organize autonomous systems into *federations*. Our protocol extends directly for more levels of the hierarchy. There are two kinds of nodes - routers, which forward data packets, and processing nodes.

We consider a *hosting system* that maintains and provides access to objects. 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, on Figure 1, the hosting system contains autonomous systems AS1, AS2, and AS3. Autonomous system AS1 comprises areas AR1, AR2, and AR3. A message from nodes in AS1 to nodes in AS2 must travel through outside routers r1 and r2. In fact, hosting systems are often provided by Internet service providers (ISP), in which case the hosting platform may be contained in a single autonomous system.

Nodes, areas and ASs belonging to the hosting system will be called *internal* nodes, areas and ASs. 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 could be introduced by incorporating into the protocol weights corresponding to relative power of hosts.

2.1 Host Proximity

We abstract the notion of host proximity by defining *proximity functions*. In the following definition, the notion of "closeness" reflects communication distance (communication costs/delay) according to the routing protocol used. Note that we do not assume that the values of these functions are always known. We will state our assumptions about what information is available later.

Definition 1 (Proximity functions)

- *For a set of internal nodes S and a given Internet node c, function $bestNode(S, c)$ defines the set of all nodes from S that are the "closest" to c;*
- *For a given internal autonomous system A and a node c within this AS, function $bestArea(A, c)$ defines the set of all areas within A that are the "closest" to c. We assume that any node in $bestArea(A, c)$ is closer to c than any node in A outside $bestAS(A, c)$. This assumption holds for so called broadcast areas (e.g., based on the Ethernet local area network) since communication between any nodes in the area has the same cost. For point-to-point areas this assumption by itself may not hold, in which case our protocols may arrive at suboptimal replica placement. To prevent this, the network administrator should organize internal autonomous systems into*

areas in such a way that communication within areas is never much higher than communication between areas.

- *For a set of internal autonomous systems $\mathcal{A}$ and a given Internet node c, function $bestAS(\mathcal{A}, c)$ defines the set of all autonomous systems from $\mathcal{A}$ that are the "closest" to c. We assume that any node in $bestAS(\mathcal{A}, c)$ is closer to c than any internal node outside $bestAS(\mathcal{A}, c)$. This assumption agrees with the way Internet routing protocols consider costs.*

We discuss proximity functions for Internet routing protocols in Section 5. Note that the values of proximity functions are *sets* of, respectively, nodes, autonomous systems, and areas. This reflects the fact that multiple nodes, ASs or areas may be equidistant to node c.

Our heuristics for replica placement are based on the routes messages take getting from the source to the destination. (Granted 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 [31] .)

Definition 2 (Preference path) *Let $s \to r_1^1 \to \ldots \to r_n^1 \to r_1^2 \to \ldots \to r_m^2 \to r_l^3 \to \ldots \to r_k^3 \to c$ be the router path from an (internal) host s to an 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 $\{r_i^3\}$ are routers outside s's AS.*

The preference path *between s and c is a sequence $bestNode(AR_s, r_1^1) \to \ldots \to bestNode(AR_s, r_n^1) \to bestArea(AS_s, r_1^2) \to \ldots \to bestArea(AS_s, r_m^2) \to bestAS(\mathcal{R}, r_1^3) \to \ldots \to bestAS(\mathcal{R}, r_k^3) \to s$, where AR_s is the set of all hosts is s's area, AS_s is the AS to which s belongs, and $\mathcal{R}$ is the set of all internal autonomous systems.*

In some cases, the same internal AS may be the closest to different routers on the path. In these cases we can simplify the preference path by considering the *canonical preference path* instead.

Definition 3 (Canonical preference path) *Let $s \to E_1 \to \ldots \to E_n \to c$ be the preference path from s to c.*

For all $i < j$, if $E_i \bigcap E_j \neq \emptyset$, replace E_i with $E_i \setminus E_j$. If E_i becomes empty, remove it from the path. Repeat this until $E_i \bigcap E_j = \emptyset$ for all pairs of the path elements. The resulting path is called the canonical preference path from s to c.

From now on, we will use the term preference path meaning canonical preference path.

Example 1 *Consider the internetwork on Figure 1, where lines represent the proximity functions. A message from host s to an external client c travels via router path $r_9 \to r_8 \to r_7 \to r_6 \to r_5 \to r_2 \to r_1$, resulting in the preference path $s \to \{h1, h2\} \to \{h3\} \to \{AR2\} \to \{AR2\} \to \{AS3\} \to \{AS2, AS3\} \to c$. The corresponding canonical path is $s \to \{h1, h2\} \to \{h3\} \to \{AR2\} \to \{AS2, AS3\} \to c$.*

A message from host s to client c passes by the hosts on the preference path from s to c, assuming that proximity functions are defined appropriately. Given that Internet routing protocols attempt to choose the shortest routes and the routes from these hosts to c are shorter than from s, it would have been advantageous *for this request* if the request was serviced by one of these hosts. Furthermore, the closer the data is on the preference path to c the greater the benefits. It is this last observation that motivates transforming the preference path to the canonical form.

We will assume the availability of the following information:

- For any client c and any subset S of (internal) hosts, the value of $BestNode(S, c)$;

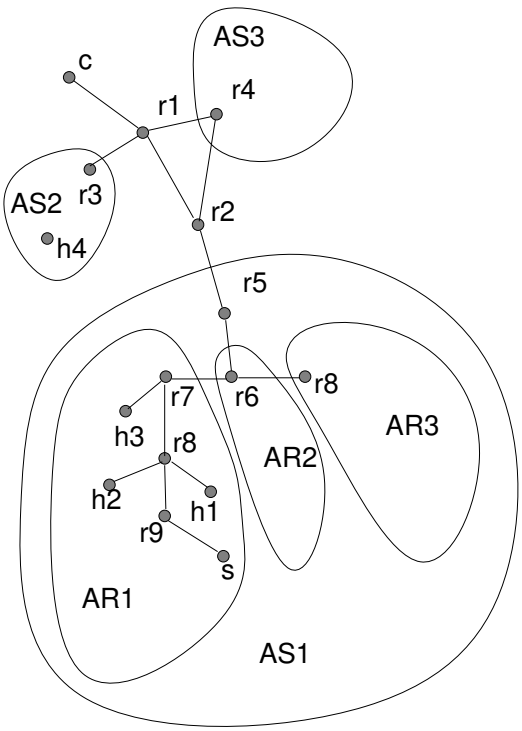


Figure 1: An example of an internetwork.

- For any client c and any host s, the (canonical) preference path between c and s.

We will discuss how to obtain this information efficiently in the context of actual IP routing protocols in Section 5. This information will be extracted from the routes databases maintained by the routers, and hence, correspond directly to the notions of "closeness" used by the routers, currently the number of hops taken by messages en route from one node to another. 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 routes databases and therefore will be picked up by our proximity functions automatically.

2.2 Load Metrics

We assume the existence of a uniform load measure that allows load comparison of different servers. In general, the load metric may reflect multiple components, notably computational load and storage utilization. With multiple load components, one can use *load vectors*, with the following operation rules:

- $load1 > load2$ if $load1$ is greater than $load2$ in *some* component; $load1 < load2$ if $load1$ is smaller than $load2$ in *all* components. Note that $load1 > load2$ does not entail $load2 < load1$, so one must be careful with the order of operands. These operations are used, e.g., to compare the server load with its capacity load.
- $load1 + load2$ is a load vector whose components are obtained by component-wise addition of operands. This operation is used, e.g., to express the combined load due to hosting two objects.
- $N * load$, where N is the number, is a load vector whose computational components are obtained by multiplying the corresponding components of vector $load$ by N, and whose storage components are equal to the corresponding components of vector $load$. This operation is used, e.g., to express the load due to hosting an object when the object access rate increases N times.

For compactness of the presentation, we will assume that load metric represents a single computational component. The length of the ready queue (e.g., the output of the *uptime* command in UNIX) can be used as the measure of computational load [24] . Extending the algorithms to allow vector loads as defined above is trivial.

We assume that an individual server can estimate the fraction of its total load due to a given object on this server. 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 between objects proportionally to their consumption.

One last technical issue is that load metrics are usually averaged over some *sampling interval* preceding the time the metric is recorded. So, a measurement taken right after an object relocation event on a host will not reflect the change in the set of hosted documents. To deal with this technicality, we assume that once a host accepts an object, it uses an upper-limit estimate of what its load would be after acquiring the object in deciding whether or not to honor further requests for accepting objects from other hosts. The host returns to using actual load metrics only when its sampling interval *starts* after the last object had been acquired.[2] Similarly, the host decides it needs to offload based on a lower-limit estimate of its load. The derivation of these estimates is enabled by our request distribution algorithm.

[2]When frequent object relocations make most of sampling intervals contain a relocation event, a host can always periodically halt relocations to take fresh load measurements.

ChooseReplica(c, x):
/* Executed by the name service */
let X be the set of hosts that have a replica of x;
let p be a node in $bestNode(X, c)$ with the smallest value of $ratio1 = \frac{rcnt(x_p)}{aff(x_p)}$ and q be the host that has a replica of x with the smallest value of $ratio2 = \frac{rcnt(x_q)}{aff(x_q)}$;
if $\frac{ratio1}{2} < ratio2$
choose p;
$rcnt(x_p) = rcnt(x_p) + 1$;
else
choose q;
$rcnt(x_q) = rcnt(x_q) + 1$;
endif
end

Figure 2: The algorithm for choosing a replica.

2.3 Notation and Terminology

In the rest of the paper, x_p will denote a replica of object x on server p; $load(p)$ will denote the load of node p, and $load(x_p)$ will denote the load on node p due to object x. An object is said to be *geo-migrated* or *geo-replicated* to host s if it is placed on s for the reason of proximity to client requests. Otherwise (when an object is migrated or replicated due to load considerations), the object is said to be *load-* migrated or replicated to s.

A request from client c is called *local* to host s if $s \in BestNode(All_hosts, c)$. The preference path of a local request contains just the host s.

3 Request Distribution Algorithm

The challenge in designing the algorithm for assigning requests to replicas that would service them is that the algorithm must combine the goal of distributing load with the goal of choosing a replica that is close to the request. As an example, consider a hosting system with just two hosts, one in America and the other in Europe. Choosing replicas in the round-robin manner would neglect the proximity factor. If roughly half of requests come from each region, the round-robin distribution may well result in directing American requests to the European replica and vice versa. On the other hand, always choosing the closest replicas could result in poor load distribution. Assume that the American site is overloaded due to "local" requests. Then, if the closest replicas were always chosen, creating additional replicas in Europe would not help the overloaded site - all requests would still be directed to this site anyway. Our goal is an algorithm that would direct requests to their closest hosts in the first case while distributing requests among both hosts (regardless of the origin of requests) in the second case. Finally, the algorithm must be simple, for two reasons. First, it lies on the critical path of servicing the request. Second, it should allow simple reasoning about load bounds on existing and potential object replicas, which would enable the replica placement algorithm to make autonomous placement decisions.

The algorithm is shown in Figure 2. For each replica x_s, the algorithm keeps a count of the number of times it chooses this replica, the request count $rcnt(x_s)$. It also maintains the *replica affinity*, $aff(x_s)$. Replica affinity is initially equal to one and then is modified by the replica

placement algorithm (see Section 4.) We refer to the ratio $\frac{rcnt(x_s)}{aff(x_s)}$ as a *relative request count*, since it reflects the request count per affinity unit.

When a request from client c arrives, the algorithm begins by identifying a replica x_q with the smallest relative request count, and a replica x_p that is the closest to the client. It then chooses the replica (among these two) by comparing the relative request count of the least-requested replica with the relative request count divided by 2 of the closest replica (a different constant can be chosen, with corresponding modifications to the replica placement algorithm).

Applying this algorithm to the above example, in the first case, both replicas will have roughly the same request count, and therefore every request will be directed to the closest replica (assuming both replicas have affinity one). In the second case, the American site will receive all requests until its request count exceeds the request count of the European site by a factor of two, at which point the European site will be chosen. Therefore, the load on the American site will be reduced by one-third on average. Creating more replicas would reduce its load even further. Assume that n replicas of an object are created. Even if the same replica is the closest to all requests, it is easy to see that this replica will have to service only $2N/(n+1)$, where N is the total number of requests. Thus, by increasing the number of replicas, we can make the load on this replica arbitrarily low. Still, whenever an odd request arrives from another replica's region, this request will be directed to its local replica.

Replica affinities allow the protocol to be very flexible in request distribution. Continuing with our example, assume that request patterns change from being equally divided between the American and European replicas to the 90%-10% split. If neither site is overloaded, the replica placement algorithm can set the affinity of the American replica to 4. With regular request interspacing (i.e., when a request from Europe arrives after every nine requests from America), the request distribution algorithm would direct 1/9 (11%) of all requests, including all those from Europe, to the European site and the rest to the American site.

One problem with this algorithm is that when a new replica is created, it will be chosen for all requests until its request count catches up with the rest of replicas. This may cause a temporary overloading of new replicas. To avoid that, the algorithm resets all request counts to 1 whenever the replica set for the object changes.

4 Replica Placement Algorithm

Decisions on replica placement are done in cooperation between hosts and the *replication service*, which is implemented as the *replicator hierarchy* (Figure 3). There is one replicator in each internal area, one in each internal autonomous system, and one *root* replicator. It is also convenient to consider hosts as trivial replicators, with a single subordinate host.

Replicators act as representatives of their regions to outside hosts. If a host decides to place a replica in another area or AS, it sends its request to the replicator of this area or AS, and the replicator chooses an underloaded host (if any) within its region to place the object. This arrangement facilitates information hiding and is essential for manageability and scalability of the system when it spans multiple areas and ASs. Adding and deleting hosts becomes an administrative action localized to the area involved; adding or deleting an entire area concerns only the replicator of the parent AS. Computing preference paths by a host becomes reliant only on information available from the routing database of the host's area and autonomous system (see Section 5).

Note that our request distribution algorithm precludes a trivial solution that would replicate every object on every server. Indeed, since this algorithm is oblivious to server loads, it distributes requests to all available replicas. Thus, excessive replicas would cause many requests to be sent to

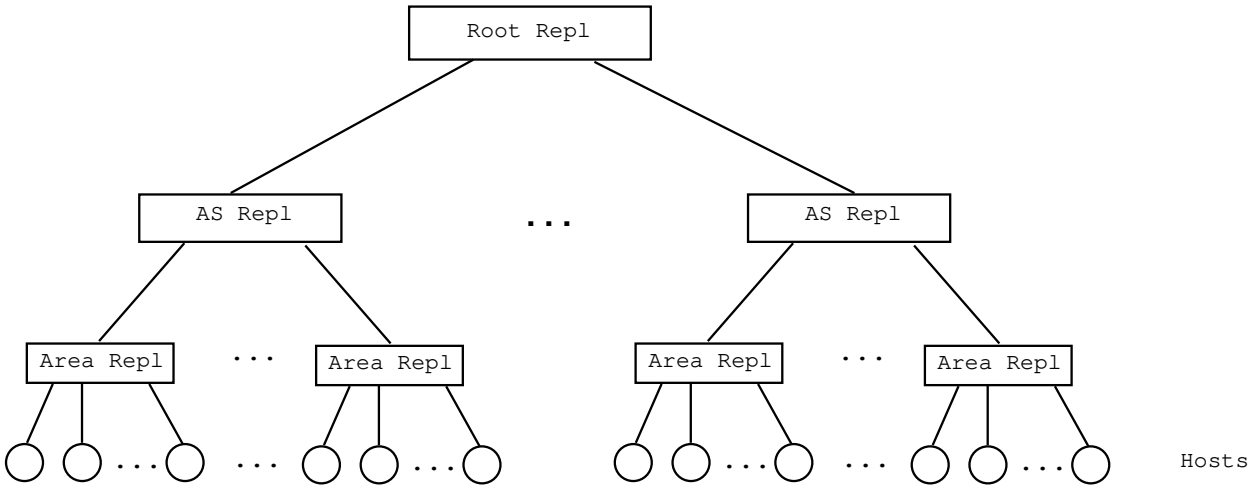


Figure 3: The replicator hierarchy.

distant hosts. The replica placement protocol therefore creates new replicas only if it is likely to be beneficial for either client proximity or server load reasons.

4.1 The Control State

Each host s maintains the following state for each object it keeps, x_s.

For each entity E (which can be hosts, areas, or autonomous systems) that appeared on preference paths of some requests to x_s in the last sampling interval, host s keeps the count of the number of these appearances, $cnt(E, x_s)$, referred to as the *access count* of E. In particular, $cnt(s, x_s) = cnt(x_s)$ gives the total access count for x_s. We know that, for a given request, nodes on its preference path represent preferable locations for the object. So, an entity that frequently appears in preference paths may be a good candidate for placing an object replica.

Host s also keeps the distance from s to E on preference paths, averaged over the requests for x_s, $dist(E, x_s)$. The distance is averaged because preference paths may change over time.

Finally, s maintains the load due to each object x_s it keeps, $load(x_s)$, and replica *affinity*, $aff(x_s)$. Affinity is a compact way of representing multiple replicas of the same object on the same host. When the replica is first created, its affinity is initialized to 1; when an object is migrated or replicated to a host that already has a replica of this object, its affinity is incremented. Similar to relative request counts, we will call a ratio $\frac{cnt(E,x_s)}{aff(x_s)}$ a *relative access count* of candidate E.

A replicator r maintains the *minimum host load*, $min_load(r_i)$, for each of its child replicators r_i. For a child r_i, $min_load(r_i)$ is the higher-bound load estimate of the the least-loaded host among r_i's descendant hosts. This state is maintained as follows: the lowest-level replicators (hosts) periodically report their load to their parent replicators. Upon collecting all reports, the parent replicator records the minimum host loads, and in turn sends the smallest reported load to its parent. Between rounds of load reports, a replicator may also modify its state during execution of the protocols below.

4.2 The Algorithm

Periodically, a host s runs the algorithm on Figure 4 to decide on replica placement for its objects. There are several tunable parameters in the protocol:

DecidePlacement():
/* Executed by host s */
if $load(s) > hw$ $offloading = Yes$;
if $load(s) < lw$ $offloading = No$;
for each x_s
  if $\frac{cnt(s,x_s)}{aff(x_s)} < u$
    decrement $aff(x_s)$ if it was greater than 1, or drop x_s otherwise
    (unless x_s is the sole replica of x in the system);
  else
    Loop through $candidates(x_s)$, in the decreasing order of $dist(E, x_s)$.
    For each candidate E such that $\frac{cnt(E,x_s)}{cnt(s,x_s)} > 0.5$
      send MigrateRequest(x_s, $\frac{load(x_s)}{aff(x_s)}$) to E's replicator, r_E;
      if r_E responded with "OK"
        decrement $aff(x_s)$ if it was greater than 1, or drop x_s otherwise;
        break from loop;
      endif
    endloop
  endif
  if x_s has not been dropped or migrated AND $\frac{cnt(s,x_s)}{aff(x_s)} > m$
    Loop through $candidates(x_s)$, in the decreasing order of $dist(E, x_s)$.
    For each candidate E such that $\frac{cnt(E,x_s)}{aff(x_s)} \geq m/2$ OR $\frac{cnt(E,x_s)}{cnt(s,x_s)} > 1/6$
      send ReplicateRequest(x_s,$\frac{load(x_s)}{aff(x_s)}$) to r_E, where r_E is the replicator of E;
      if r_E responded with "OK"
        break from loop;
      endif
    endloop
  endif
endfor
if $offloading = Yes$ AND no objects were dropped, migrated or replicated
  send OffloadRequest(s) to the parent replicator of s;
endif

Figure 4: Replica placement algorithm.

- Low and high watermarks for server load, lw and hw;
- Deletion threshold u and replication threshold m.

The parameters must be chosen subject to a constraint $4u < m$. A replica can be dropped if its count falls below u, it can only migrate if its count is between u and m, and it can either migrate or be replicated if its count is above m.

The request distribution algorithm guarantees that the load on any replica of an object after replication will be at least one quarter of the lowest pre-replication replica load among replicas that can cause replication (see proof of Lemma 8.3). Knowing this, and by adhering to the above condition on u and m, each host can make an autonomous decision to replicate an object based just on its own replica load, without creating a vicious cycle of replica creations and deletions. Indeed, every replica after replication will have the load exceeding u, so no replicas will be dropped.

A host s can be in one of the two modes of operation. If its load exceeds high-water mark hw, it switches to an $offloading$ mode, where it sheds objects to other hosts, even if it is not beneficial from the proximity perspective. Once in this mode, the host continues in this manner until its load drops below a low water mark, lw. Then, it moves objects only if it is improves their proximity to the requesters, and stays in this mode until its load again exceeds hw. Water-marking is a standard technique to add stability to the system. These two modes of operations are described in the next two subsections.

4.2.1 Geo-Migration and Replication

After establishing its mode of operation, s examines access counts of each of its objects to decide on its placement. The main components of the placement decision are:

- Choose objects whose placement must change, and decide between dropping, migrating, or replicating these objects.
- For each object that is to be migrated or replicated, choose the new location to host this object.

An affinity unit of an object is dropped if its relative access count is below a deletion threshold u. (It should not delete the sole replica of an object in the system. The request redirection service is a convenient place to ensure that, since it can arbitrate among competing replica deletions. Details are provided in [32] .)

An object x_s is chosen for migration if there is a candidate E that appears in preference paths of over 50% of requests for this object. This restriction is a heuristic that aims at migrating an object to a node that would be closer to most of the requests. In addition, it prevents an object from migrating back and forth between nodes in most cases.

An object x_s is chosen for replication if it has not been migrated and if (a) its total relative access count exceeds m and (b) there is a candidate E such that either (ba) the access count of E is over one-sixth of the total access count for the replica, or (bb) the per affinity unit access count of E exceeds $\frac{m}{2}$. Condition (a) is to avoid a vicious cycle of replica creations and deletions as discussed earlier. Condition (b) is a heuristic that replicates to a candidate only if it was closer than s to some minimum number of requests, either relative to the total number of requests received by s, or in terms of the absolute per-affinity unit value.

Condition (ba) assures that, when s has the sole replica of object x, creating an additional replica will statistically improve the proximity of requests to their servicing hosts. Condition (bb) allows

```
ReplicateRequest(x_s,load(x_s)):
/* Executed by replicator r. */
  if r is a host
    if load(r) > lw
      send "Refuse" to invoker;
      exit;
    endif
    if r does not have x already
      copy x from host s;
      aff(x_r) = 1;
    else
      aff(x_r) = aff(x_r) + 1
    endif
    load(r) = load(r) + 4load(x_s);
    send Done(load(r)) to invoker;
    exit;
  endif
  /* r is a non-leaf replicator */
  let r_min be the replicator with the smallest minimum host load among child replicators of r;
  send ReplicateRequest(x_s,load) to r_min
  if r_min responded with Done(load)
    UpdateMinLoads(r_min, load);
    send Done(min_load) to invoker;
  else
    send "Refuse" to invoker;
  endif
end
UpdateMinLoads(r_min, load):
  min_load(r_min) = load;
  min_load = min{min_load(r_1), ..., min_load(r_m)} where r_1, ..., r_m are child replicators of r;
end
```

Figure 5: Protocol for object replication.

bootstrapping the replication process of an extremely popular object even if the first replica to be created would hurt the request proximity. By demanding the candidate to have minimum access count of $m/2$, it prevents the new replica from migrating right back, giving the protocol time to create further replicas elsewhere.[3] Note that these are just heuristics, and it is easy to provide examples of detrimental replication.

When object x_s is to be replicated or migrated, s attempts to place the replica on the farthest among all qualified candidates. This is a heuristic that improves the responsiveness of the system.

To replicate object x_s on a candidate E, host s initiates the protocol on Figure 5. Host s sends a replication request to E's replicator r_E, which includes the ID of the object to be replicated and the load on host s generated due to x_s. r_E forwards this request along the path with the lowest-loaded hosts. When this request reaches a host, the host checks if its load is below the low watermark, in

[3]Indeed, if condition (bb) holds before replication, for the new replica to migrate back (i.e., have host s appear in more than 50% of request preference paths), its total access count must exceed m. As we will see shortly, the protocol explicitly prohibits migrating a replica whose total access count exceeds m to a recipient that already has a replica of this object.

which case it copies the object from s or increments the affinity of an existing replica of the object, updates its upper-bound load estimate, and sends acknowledgment back up the tree to r_E. Each replicator on the path updates its upper-bound estimates of the minimum host load for itself and the child replicator involved. These estimates are based on the fact that, when a host q acquires a new replica x_q, extra load imposed on q is bounded by four times the load of an existing replica, $load(x_s)$ (see Appendix A). When the load of the leaf host exceeds the low watermark, it sends a refusal up the replicator tree, which eventially reaches the host that requested the replication .

The protocol for geo-migration to the chosen candidate is the essentially same as the protocol for geo-replication of Figure 5. The only difference is that, in addition to the requirement that the receiver's load be below the low watermark, there are two additional restrictions: (a) a host that already has a replica of x will accept the migration request only if the relative access count of x on the sending host is below m; and (b) the recipient host accepts the migration request only if its upper-bound load estimate *after* the proposed migration is below the high watermark. Both these conditions are heuristics targeting especially heavily accessed objects. Condition (a) gives a popular object a chance to replicate on multiple nodes even if a majority of requests are local to a single node (cf. condition (bb) for replication). Condition (b) prevents migration to a host in cases when a single object migration would bring the recipient's load from below the low watermark to above the high watermark. Without it, a vicious cycle could occur when an object load-migrates from a locally overloaded site, only to geo-migrate back to the site.

Note the absence of a similar restriction in the replication heuristics. Overloading a recipient site temporarily may be necessary in this case in order to bootstrap the replication process. The vicious cycle is not a danger here because each replication brings the system into a new state.

4.2.2 Host Offloading

When host s is in *Offloading* mode, it migrates or replicates objects to other nodes even if it is not beneficial from the proximity perspective. The protocol contains two stages. In the first stage the protocol identifies a recipient of the objects from s. The recipient is chosen to be the nearest node to s whose load is below lw.

This stage, shown in Figure 6, is initiated by the request for offloading from s to its parent replicator. This request travels up the tree until it find the replicator that has a child replicator with the minimum host load below lw. Then, this replicator send RecipientRequest message down the tree along the path with the lowest load until it reaches a host q.

Upon receiving RecipientRequest, host q starts the second stage of the protocol by sending OffloadOffer to s. The algorithm for this stage is given in Figure 7. The offloading host goes through all its objects, starting with those that have a higher rate of "foreign" requests, and attempts to send them to the recipient. Unlike in the *DecidePlacement* algorithm, the offloading host does not try to migrate heavily loaded objects - objects with the relative access count above replication threshold m can only be replicated. The reason is that load-migrating these objects out might undo a previous geo-replication.

To decide conservatively when to stop the offloading process, the sending node re-calculates the lower-bound estimate of its load and the upper-bound estimate of the recipient load upon every object migration or replication (see Appendix A for derivations of these estimates). When the former falls below or the latter rises above the low watermark, the offloading stops until actual load measurements are available.

Note that these estimates reflect the load changes only due to the migration or replication performed; concurrent changes in replica placement on other nodes can obviously affect the load of the nodes involved in the offloading. However, this is true for even a single object transfer, when

```
OffloadRequest(s):
/* Executed by the replicator r */
  let r_min be the replicator with the smallest minimum host load among child replicators of r;
  if min_load(r_min) < lw
    send RecipientRequest(s) to r_min;
    if r_min responded with OffloadDone(load)
      UpdateMinLoads(r_min, load);
      exit;
    elseif r_min responded with OffloadFailed(load)
      UpdateMinLoads(r_min, load);
      send OffloadRequest(s) to parent unless r is the root;
    endif
  else
  send OffloadRequest(s) to parent unless r is the root;
end

RecipientRequest(s):
/* executed by node q */
  if q is a host
    if load(q) < lw
      initiate offloading from s by sending OffloadOffer(load(q)) to s;
      if any objects were offloaded
        send OffloadDone(load(q)) to invoker;
      else
        send OffloadFailed(load(q)) to invoker;
      endif
    else
      send OffloadFailed(load(q)) to invoker;
    endif
    exit;
  endif
  let r_min be the replicator with the smallest minimum host load among child replicators of q;
  if min_load(r_min) < lw
    send RecipientRequest(s) to r_min;
    if r_min responded with OffloadDone(load)
      UpdateMinLoads(r_min, load);
      send OffloadDone(min_load(q)) to invoker;
    elseif r_min responded with OffloadFailed(load)
      UpdateMinLoads(r_min, load);
      send OffloadFailed(min_load(q)) to invoker;
    endif
  else
    send OffloadFailed(min_load(q)) to invoker;
  endif
end
```

Figure 6: Stage 1 of the protocol for host offloading: finding the recipient.

OffloadOffer($recipient_load$):
/* Executed by the offloading host s */
  while $load(s) > lw$ AND $recipient_load < lw$ AND not all objects have been examined
    let x_s be the unexamined object with the highest value of $\frac{cnt(E,x_s)}{cnt(s,x_s)}$ for some E
    if $\frac{cnt(x_s)}{aff(x_s)} \leq m$
      send MigrateRequest(x_s, $\frac{load(x_s)}{aff(x_s)}$) to q;
      $load(s) = load(s) - \frac{load(x_s)}{aff(x_s)} - (3/4) * \frac{load(x_s)}{aff(x_s)-1}$;
      $recipient_load = recipient_load + 4\frac{load(x_s)}{aff(x_s)}$;
      decrement $aff(x_s)$ if it was greater than 1, or drop x_s otherwise;
    else
      send ReplicateRequest(x_s,$\frac{load(x_s)}{aff(x_s)}$) to q;
      $load(s) = load(s) - (3/4) * \frac{load(x_s)}{aff(x_s)}$;
      $recipient_load = recipient_load + 4\frac{load(x_s)}{aff(x_s)}$;
    endif;
  endwhile;
end

Figure 7: Stage 2 of the protocol for host offloading: re-distributing objects.

no load estimates are used. Using estimates does not worsen our load prediction while allowing transferring objects in bulk, which is essential for responsiveness of the system to demand changes.

5 Obtaining Proximity Functions and Preference Paths

A fundamental implementation question in any replicated system is the architecture of the indirection module responsible for distributing requests among physical replicas. As mentioned in Section 1.1, different products supporting replication on the Internet offer different solutions to this issue. In this paper, we concentrate on algorithms, leaving architectural issues out. (See Appendix B for our proposal on a scalable architecture of indirection service that is well suited to the algorithms described here.) Instead, we describe in this section how information required by the proposed protocol can be obtained in the context of today's Internet. Specifically, we answer the following questions:

1. For a given request, how to find the closest host having a replica of the requested object. This information is used in the request distribution algorithm.

2. How the host receiving a client request obtains the preference path for this request. While the `traceroute` tool [37] 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 slightly 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 [20] .

5.1 Background: Internet Routing

Under BGP/OSPF, the whole Internet is divided into administrative domains called *autonomous systems (AS)*, identified by unique *AS numbers*. All message traffic external to an AS is handled by *border routers*. Inter-AS routing is based on *path vectors* maintained by the border routers. A path vector on a given border router contains the numbers of ASs on the best path from this router to every network in the Internet.

All border routers of the same AS communicate between themselves to decide, for each external destination X, on the best router(s) to deliver packets to X. Then, the best router(s) advertises within the AS that it can deliver traffic to this destination. If multiple border routers advertise this, one is chosen on the case by case basis, depending on the cost of reaching the border router from the source of the message within the AS. Thus, the BGP routing database within an autonomous system X contains the following information, for every external destination A: (a) which border routers are the best to reach A from nodes within X - we denote this set of routers $BorderRouters(A, X)$ below - and (b) the path vectors from these routers to A. In particular, the function $BGP_hops(A, X)$ that gives the number of hops from X to every autonomous system on the Internet A is available (this is the number of elements in the corresponding path vector),

A large autonomous system can be further divided into OSPF *areas*. Areas within the same AS are connected by *area-border routers*, which belong to both areas they connect and handle all inter-area traffic. Within a given area, the area routers maintain a *link state database*, which contains *link records* and *summary records*. Link records describe all links within the area and the cost metrics for sending messages over these links. This information is sufficient to compute the closest path between any two hosts in the area. Summary records are advertised by area-border routers to the areas they connect. Summary records advertised by a router r to its area contain the total cost of delivering a message from r to each destination i outside this area but within the autonomous system. In combination, link and summary records in the link state database of an OSPF area allow computing the cost of delivering a message from any node i within the area to any node j within the autonomous system, denoted $OSPF_metric(i, j)$ below.

Normally, OSPF areas within an AS are organized hierarchically: there is a *backbone area* that contains all border routers in the AS, and *leaf areas*, each connected to the backbone area by one or more area-border routers. To be specific, we assume this organization in the rest of this section.

5.2 Obtaining the Data from the Routing Databases

The request distribution service extracts the following information from the routing databases of internal autonomous systems, using for example the functionality provided by Distributed Director from CISCO [11] .

The BGP routing database in every internal autonomous system IAS is queried to obtain functions $BorderRouters(A, IAS)$ and $BGP_hops(A, IAS)$. The OSPF link state database of the backbone area in each IAS is queried to obtain function $OSPF_metric(br, s)$, for every border router br and every host s.

Using this information, the request indirection service computes the mapping $(A \times host) \rightarrow distance$ for each autonomous system A and internal host $host$, where $distance$ is represented as a pair in which the first component is the number of BGP hops from $host$'s autonomous system to A, and the second component is the OSPF cost of delivering a message from $host$ to the nearest border router that advertized the external route to A within $host$'s autonomous system. Formally,

$$distance = (BGP_hops(A, IAS(host)), \min\{OSPF_metric(r, host) | r \in BorderRouters(A, IAS(host))\}),$$

where $IAS(host)$ denotes the (internal) AS to which *host* belongs. 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 indirection service uses this mapping to find the closest host to a given client request, thereby implementing a function $bestNode(X, c)$ in the request distribution algorithm.

Each internal host h extracts the following information (much of it is a specialization of information obtained by the indirection service to h's IAS). Let $IAS(h)$ and $area(h)$ be, respectively, the autonomous system and OSPF area to which h belongs.

- From the BGP routing database of $IAS(h)$, host h obtains function $BorderRouters(A) = BorderRouters(A, IAS(h))$, for every external destination A;
- From the function $BGP_hops(A, IAS)$, h computes function $BestIAS(A)$, which gives internal autonomous system(s) with the minimal paths to A.
- From the link state database of $area(h)$, host h obtains function $OSPF_metric(br, h)$ for each border router br in $IAS(h)$ and function $BestHost(lr_i)$, which, for each router lr_i within $area(h)$, gives the closest host(s) to lr_i. In other words,

  $$BestHost(lr_i) = \{h_{best} \in area(h) | OSPF_metric(lr_i, h_{best}) = \min_{h_j \in area(h)} \{OSPF_metric(lr_i, h_j)\}.$$

- From the link state database of the backbone area of h's autonomous system, h obtains function $BestABR(r_i)$ that, for each backbone area router r_i, gives the closest area-border router(s) to r_i, among area-border routers connecting the backbone area to leaf areas.

Functions $BestHost$, $BestABR$, and $BestIAS$ represent specializations of proximity functions *bestNode*, *bestArea* and *bestAS* to sub-domains needed by host h. ($BestABR$ corresponds to *bestArea* because there is a mapping from area-border routers to the OSPF leaf areas they connect.) With the exception of $BestIAS$, which can be obtained from the request indirection service, all other information is obtained from routing databases within h's autonomous system. Host h uses this information to compute a database of preference paths to every destination A on the Internet using the following procedure.

1. Let br be the border router with the minimal OSPF cost from h, according to function $OSPF_metric(br, h)$, among all border routers in $BorderRouters(A)$.[4] Let $br \rightarrow A_1 \ldots A_p \rightarrow A$ be the AS path from br to A from br's path vector database. Replace each A_i that is not an internal autonomous system with the closest IAS(s) $R_i = BestIAS(A_i)$, resulting in a path $br \rightarrow R_1 \ldots R_p \rightarrow A$. Drop the first element (br) from the path.
2. Let abr be the area-border router on the shortest path from h to br. From the link state database of the backbone OSPF area of h's IAS, obtain the router path from abr to br, $abr \rightarrow r_1 \ldots r_m \rightarrow br$. Replace each router r_i with the id of leaf OSPF area(s) $a_i = area(abr_i)$ where $abr_i = BestABR(r_i)$, and router br with the id of OSPF area $a_{br} = area(abr_{br})$, where $abr_{br} = BestABR(br)$. The resulting path is $abr \rightarrow a_1 \ldots a_m \rightarrow a_{br}$. Drop the first element (abr) from the path.
3. Let $h \rightarrow r_1 \ldots r_n \rightarrow abr$ be the router path from h to abr according to the link state database of $area(h)$; Replace each r_i (including abr) with the closest host $h_i = BestHost(r_i)$, resulting in a path $h \rightarrow h_1 \ldots h_n \rightarrow h_{abr}$.

[4] For simplicity, we do not consider a case where multiple border routers can have the minimal cost; including this case is a technical matter.

4. Concatenate all paths and use the resulting path

$$h \to h_1 \dots h_n \to h_{abr} \to a_1 \dots a_m \to a_{br} \to R_1 \dots R_p \to A$$

and compute the canonical path to A according to its definition.

6 Replica Consistency

All objects can be divided into the following types:

1. Objects that do not change as the result of user accesses. These objects can be either static HTML pages, or dynamic pages that retrieve information, e.g., weather reports, or map-drawing services.
2. Objects in which the only per-access modification involves collecting access statistics or other commuting updates.
3. Objects where updates that result from user accesses do not commute.

Objects in the first category can change only as a result of updates by the content provider. Consistency of these updates can be maintained by using the primary copy approach, with the node hosting the original copy of the object acting as the primary. Depending on the needs of the application, updates can propagate from the primary asynchronously to the rest of currently existing replicas either immediately or in batches using epidemic mechanisms [16] . These objects can be replicated or migrated freely, with the provision that either the primary copy of an object never migrates or the location of the primary copy is tracked by the redirection module. In the case of batched update propagation, techniques used in Web caching to prevent (or bound) staleness of cached copies (see, e.g., Alex adaptive copy expiry [10] , and piggybacked validation [25]) can also be used here to prevent serving stale content to clients. Multiple studies (e.g., [28]) have shown that an overwhelming majority (over 90%) of Web object accesses are to this category of objects.

Objects in the second category can still be replicated using our protocol if a mechanism is provided for merging access statistics recorded by different replicas. The problem arises if content served to clients includes some access statistics, as is the case of access counter appearing in some pages. If application requires this information to always be served current, then such objects become equivalent to objects in the third category for the purpose of our protocol.

Objects in the third category, in general, can only be migrated in our protocol. In the case when the application can tolerate some inconsistency, updates can be propagated asynchronously, either immediately after the access or in batches. In this case, one may still find it beneficial to allow the protocol to create a limited number of replicas. The protocol itself remains the same subject to the additional restriction that the total number of replicas remain within the limit.

7 Performance

We performed an event-driven simulation to study the performance of our protocol. Since it is logical for an ISP to also provide hosting services, we chose the backbone of one of the largest existing ISPs, UUNET, as a testbed. The backbone (see [39] for the map) contains 53 nodes in North America, Europe, Pacific Rim, and Australia.

Parameter	Value
Number of objects	100000
Placement decision frequency	Every 200 seconds
Node request rate	20 and 40 requests per sec
Server capacity	200 requests per sec
Network delay	10ms per hop
High watermark	60 requests/sec
Low watermark	40 requests/sec
Deletion threshold u	0.03 requests/sec
Replication threshold m	$6u$, or 0.18 requests/sec

Table 1: Simulation parameters.

7.1 Simulation Model

We assume that all the backbone nodes belong to the same OSPF area, so the replicator hierarchy in this case collapses into a single node. In the simulation, each backbone node is responsible for generating client requests originated from its vicinity (e.g., regional networks connected to this backbone node). Also, instead of modeling hosting servers separate from backbone nodes, we assume that the backbone nodes themselves maintain and service requests for objects that would have been maintained by hosts connected to the corresponding backbone nodes. This simplifies simulation setup while still permitting correct analysis of the backbone traffic.

Objects are initially distributed among the nodes in the round-robin fashion, so that object i is assigned to node i mod 53. Each request is assigned to a physical replica according to the algorithm of Figure 2 and is routed to the node that hosts this replica along the shortest path, which is also the preference path for this request. When there are equidistant paths between nodes i and j, one path is chosen for all requests from i to j.

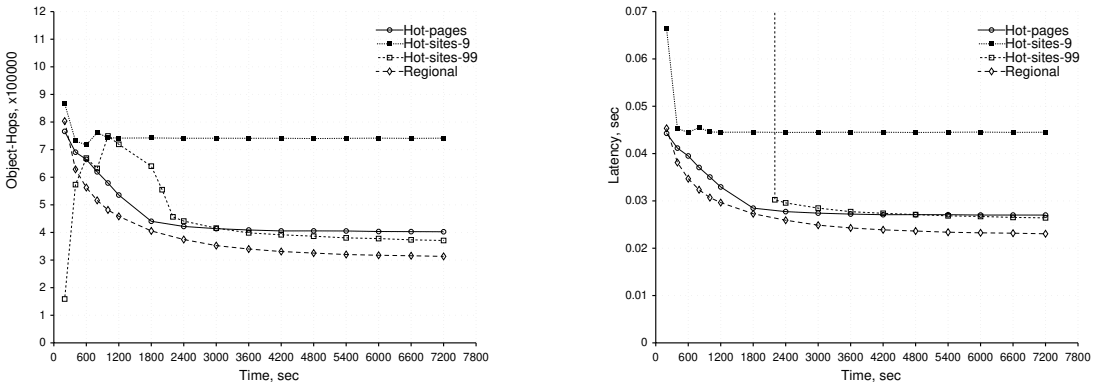
We explore the following workloads. One can expect than a real-life workload would be some mix of workloads similar to the ones considered.

- **Hot-pages.** Previous studies [15, 2] observed that the popularity of pages requested by clients, as well as popularity of pages on a given Web site, follows Zipf's law, which basically says that if pages are ranked according to their access frequency, then page i is twice as popular as page $2i$. Thus, in this workload, we assume that every next page requested by any client is chosen according to Zipf's law, where the page number corresponds to its popularity rank.[5] The hot-pages workload models the situation when some objects are uniformly more popular than others.

- **Hot-sites** All sites are divided randomly into hot and cold, with p fraction of sites going to the cold bucket and the rest to the hot bucket. A client chooses a random page among those initially assigned to hot sites, with probability p, and a random document from a cold site, with probability $1 - p$. We refer to the **hot-sites** workload with $p = 0.9$ and $p = 0.99$ as **hot-sites-9** and **hot-sites-99**, respectively. This workload models the situation when entire Web sites vary in popularity.

- **Regional.** All nodes are divided into four regions: Western North America, Eastern North America, Europe, and Pacific and Australia. Each region is assigned a contiguous set of object

[5] We use an approximation of Zipf's distribution due to Jim Reeds, where the next page number is calculated as the value of $e^{u(0,1)*\log(n)}$ rounded to the closest integer, where $u(0,1)$ returns a random real number uniformly distributed between 0 and 1, and n is the number of objects [34] .



(a) Network traffic.

(b) Average response latency.

Figure 8: Effects of dynamic replication.

numbers totaling 1% of all objects, representing a preferred object set for the region. Then, with probability 90%, each node requests a random object from the preferred set for this node; with probability 10% a random object from the entire set of objects is chosen. The regional workload models the situation when object popularity varies with the region.

The time the system takes to adjust to a given workload, starting from the initial document assignment, indicates the responsiveness of the protocol to a change in the demand pattern. Each node services requests one by one in the first-come-first-serve order. A node's load is measured as the rate of serviced requests and is averaged over the period between consecutive placement decisions. (In other words, the sampling interval is equal to the interval between placement decisions.) A request incurs latency due to queuing and processing at the server and network delays. Table 1 lists simulation parameters for our experiments.

7.2 Results

Figure 8 depicts the network traffic due to sending objects to clients and average response time, while Figure 9 shows the traffic generated by our protocol due to moving objects between nodes. The latter occurs in addition to the request traffic and represents the traffic overhead of our protocol.

Our first observation is that the traffic overhead introduced by the protocol never exceeds 2% of the total client traffic, and subsides to around 0.5% in the first 20 minutes. At the same time, our protocol achieves the client traffic reduction of 14.5% for **hot-sites-9**, 47.5% for **hot-pages**, and 61% for **regional**, compared to the initial random distribution of non-replicated objects to nodes. The reason for the modest traffic reduction in the **hot-sites-9** case is that the demand for pages on hot sites does not exceed the replication threshold, so virtually no new replicas are created (see Table 2 below); at the same time, since hot sites are accessed uniformly from all clients of the network, geo-migration is rarely warranted also, except for pages on the network periphery.

It may appear from Figure 8a that the protocol is detrimental for the **hot-sites-99** workload. However, the increase of the total client traffic in this case is due to the increased number of requests processed by the system. The reason is that, until the protocol offloads pages from hot

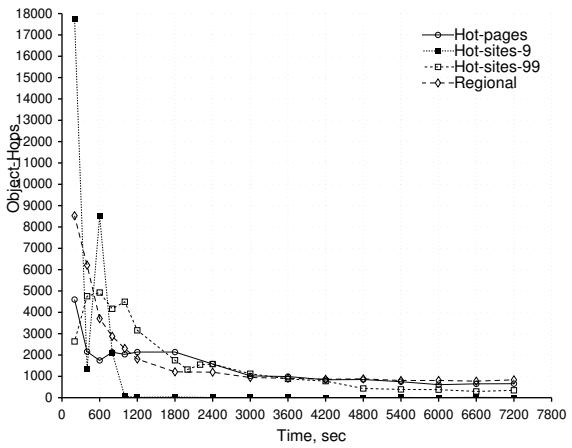


Figure 9: Network overhead.

sites, virtually all requests go to a small number of sites, which can only process a small fraction of the arriving requests. As the system adopts to the workload, not only it becomes capable of servicing all generated requests, but it also clears the backlog of requests that had been enqueued at the hot sites early on. This is evidenced by the latency change, which drops from hundreds of seconds to the levels comparable to other workloads. In fact, the average number of hops an object travels from the server to the client decreases in the **hot-sites-99** case by 53.6%.

Latency reduction is also significant - 33% for **hot-sites-9**, 39.1% for **hot-pages**, and 49% for **regions**. In the case of **hot-sites-99**, our protocol successfully eliminated hot spots in the network, which caused ever-increasing backlogs of requests and huge latencies (in a real system, many of these requests would time out). This allowed all requests to be processed in reasonable time. For **hot-pages** and **regional** workloads, none of the servers were ever overloaded since popular pages were initially distributed evenly among all sites. Thus, latency reduction was entirely due to fewer hops between clients and hosts. A disproportionally large latency reduction in the **hot-sites-9** case was because some queuing occurred at hot sites until objects were re-distributed. Therefore, the protocol helps in this case by both reducing the hop distance and eliminating hot spots.

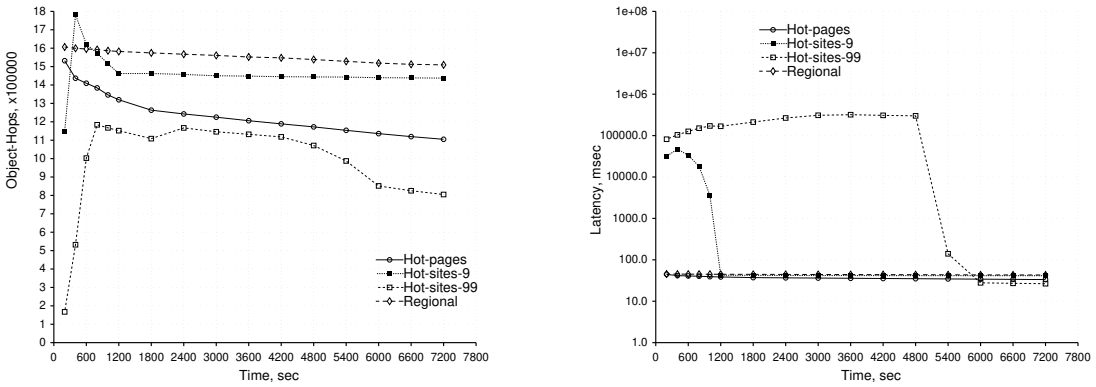
With respect to responsiveness to demand changes, Figure 9 shows that the protocol activity largely subsided within 40 minutes after the demand change. In fact, in all cases, much of the performance gains were obtained earlier than that. By decreasing placement decision frequency, one could further improve the responsiveness of the protocol at the expense of an increased risk of reacting to minute changes in demand patterns. Previous studies indicate that Web traffic is very bursty for intervals less than half a minute [17] . That motivated our choice for the placement decision frequency of once every 200 seconds.

Table 2 shows the average number of object replicas at time 7200, when the protocol activity subsides to a background noise. It shows that the protocol achieves its traffic and latency reduction at the expense of creating at most 10% additional replicas for the workloads studied. The disproportionate gains in traffic and latency reduction in each of the workloads indicate that protocol's heuristics are successful in deciding when and where to perform replication.

Figure 10 shows system behavior under heavy load, when each node generates 40 requests per second. This, on average, places the low watermark load on every server. For **hot-sites**, the system still has no problem eliminating hot spots, evidenced by a dramatic drop in latency on Figure 10b.

Workload	Ave. replicas per object
hot-pages	1.03948
hot-sites-9	1.00004
hot-sites-99	1.09468
Regional	1.07471

Table 2: Average number of object replicas.



(a) Network traffic.

(b) Average reponse latency.

Figure 10: Effects of dynamic replication under heavy load.

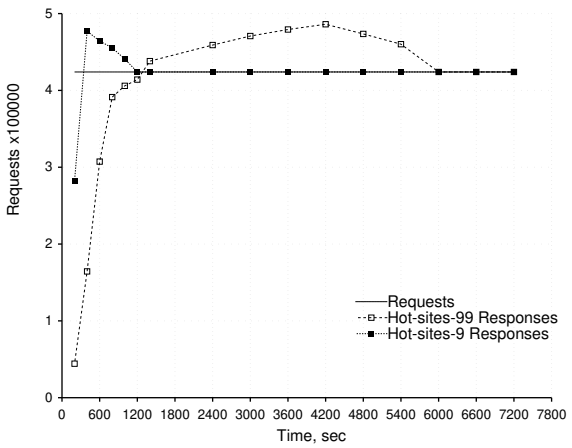


Figure 11: Arrived and processed request numbers in the system under heavy load.

For **hot-pages**, however, performance improvements diminished, to 27.9% in client traffic and 25.1% in latency reduction.[6] For **regional**, the improvements all but disappeared, with both traffic and latency reductions in single digits.

The reason for this behavior is that the protocol has no mechanism for exchanging pages between overloaded nodes. For instance, if node i has a page accessed mostly from node j and vice versa, it would make sense for these nodes to swap these pages - the load would remain the same but the proximity would improve. Yet the load above the low watermark prevents either node to accept any request for page placement.

Under the **regional** workload, popular pages are distributed evenly among all servers, so that the load is spread evenly among all the servers. Therefore, all servers have the load around the low watermark, and not much improvement is possible. In the case of *hot-pages*, the load is not distributed as even (e.g., server 1, which hosts the first page, will be initially loaded the mosts according to Zipf's law). Thus, the protocol is still able to move pages around. The **hot-sites** workload places most of the load initially on just the few hot sites. These sites become severely overloaded while the rest of the servers are underloaded. Therefore, the overloaded sites are able to offload their pages to more beneficial locations among the underloaded servers, not only bringing each server's load below its capacity, but also improving client-server proximity.

Moreover, the improvements for **hot-pages** and **regional** occur more gradually than on Figure 8, which indicates a worse responsiveness to changes in demand. This is because, with recipient nodes being close to the low-watermark, the protocol is unable to relocate multiple objects in one transfer. In the case of *hot-sites*, again, the system is still able to adapt quickly. For the **hot-sites-99** case, the long time it took the latency to drop is not indicative of responsiveness of the system; it is due to a greater backlog of accumulated requests that was being cleared long after the hot spots were eliminated. As shown in Figure 11, the hot spots were completely eliminated by time 1400, when the total rate of processed requests in the system started exceeding the total rate of arriving requests. Then, until time 6000, the number of processed requests remained above the number of arriving requests, which means the system was clearing its backlogged requests. Note that the time when the processed requests number drops to the level of the arriving requests number (time 6000 on Figure 11) coincides with the time of the latency drop on Figure 8b.

[6] The latency reductions are not discernible on Figure 10b due to the logarithmic scale on the y-axis.

In general, we conclude that for our protocol to perform well, the total system capacity, with respect to the request volume, must be such that on average, each host would stay well below its low watermark. In this case, the protocol will successfully eliminate hot spots and also quickly adapt to changing demand patterns by moving objects close to the requests. When average per-host request volumes approach the low watermark, the ability of the protocol to move objects closer towards requests diminishes, while the ability to eliminate hot spots remains.

We are currently exploring ways to improve system behavior under heavy load. There are a few possibilities, one of which is providing a mechanism for detecting cycles of overloaded nodes that would like to send pages to each other, similar to deadlock detection. These cycles could then be broken by either accepting the pages in question regardless of temporary overloading, or by making room for these pages by sending some other pages to random servers whose load is below the watermarks.

8 Conclusion

Manual mirroring of Internet objects is a slow and labor-intensive process. Beside high costs, this results in poor performance while system administrators react to changes in demand for some objects. This paper proposes a protocol for automatic replication and migration of objects in response to demand changes. The goal is to place objects in the proximity of a majority of requests while ensuring that no hosts become overloaded.

The protocol incorporates two main parts - an algorithm for deciding on the number and placement of replicas and an algorithms for distributing requests among replicas. The replica placement algorithm executes autonomously on each node, without the knowledge of other object replicas in the system. The request distribution algorithm uses the same simple mechanism to take into account both the proximity of servers to requests and server load, without actually knowing the latter. Both algorithms only use information that is feasible to obtain from the routing databases maintained by Internet routers.

A simulation study showed that the additional traffic generated by our protocol due to moving objects between hosts is by far offset by the reduction of backbone traffic due to servicing client requests from nearby hosts. It is also effective in reducing response latency experienced by clients and eliminating hot spots among servers. These benefits are achieved at the expense of creating only a small number of extra replicas, which indicates that the heuristics used by the protocol are successful in deciding selectively when and where to perform replication.

Acknowledgements

We wish to thank Vasilis Vassalos and Divesh Shrivastava for many discussions during Vasilis'es summer internship at AT&T Labs. Thanks also go to Thimios Panagos for the simulator toolkit used in our simulation study. Jim Reeds provided a convenient approximation for Zipf's law. We would like to thank many other people for stimulating discussions and comments, including Alex Biliris, Fred Douglis, H. V. Jagadish, Donald Towsley, and Dennis Shasha.

References

[1] D. Agrawal and A. El Abbadi. Resilient Logical Structures for Efficient Management of Replicated Data. In *Proc. of the 18th Int. Conf. on Very Large Data Bases*, pp. 151-162, 1992.

[2] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proc. of the IEEE Conf. on Parallel and Distr. Information Sys*, 1996.

[3] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive Distributed File Allocation. In *Proc. of the 25th ACM Symposium on Theory of Computing*, pp. 164-173, May 1993.

[4] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed Paging for General Networks. In *Proc. of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pp. 574-583, January 1996.

[5] Y. Bartal, A. Fiat, and Y. Rabani. Competitive Algorithms for Distributed Data Management. In *Proc. of the 24th ACM symposium on Theory of Computing*, pp. 39-50, 1992.

[6] M. Baentsch, L. Baum, G. Molter. S. Rothkugel, and P. Sturm. Enhancing the Web's Infrastructure: From Caching to Replication. *IEEE Internet Computing*, Vol 1, No. 2, pp. 18-27, March/April, 1997.

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

[8] A. Bestavros and C. Cunha. Server-initiated document dissemination for the WWW. *Bulletin of the Computer Society technical Committee on Data Engineering*, pp. 3-11. Vol. 19, No. 3, September 1996.

[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] V. Cate. Alex – A global file system. In *Proc. 1992 USENIX File System Workshop*, pp. 1–12, 1992.

[11] CISCO DistributedDirector. While paper. Available at http://www.cisco.com/warp/public/734/distdir/dd_wp.htm

[12] CISCO Cache Engine. White paper. Available at http://www.cisco.com/warp/public/751/cache/cds_wp.htm

[13] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Sciences*, 23:493-509, 1952.

[14] 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.

[15] C. A. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW client-based traces. Technical Report TR-95-010, Boston University Department of Computer Science, April 1995.

[16] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of the 6th Symp. on Principles of Distr. Computing*, pp. 1-12, 1987.

[17] S. D. Gribble and E. A. Brewer. System design issues for Internet middleware services: deductions from a large client trace. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pp. 207–218. December 1997.

[18] J. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proc. of the HotOS Workshop*, 1994. Also available as ftp://das-ftp.harvard.edu/techreports/tr-34-94.ps.gz.

[19] 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.

[20] C. Huitema. *Routing in the Internet*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[21] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. on Database Sys.* 15(2), pp. 230-280, June 1990.

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

[23] E. Katz, M. Butler, and R. McGrath. A scalable Web server: the NCSA Prototype. *Computer Networks and ISDN Systems*, 27, pp. 155-164, September 1994. May 1994.

[24] O. Kremlin and J. Kramer. Methodical Analysis of adaptive load sharing algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 6(3), pp.747-760, November 1992.

[25] B. Krishnamurthy and C. E. Wills. Study of piggyback cache validation for proxy caches in the World Wide Web. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pp. $1-12$, December 1997.

[26] Q. Kure. Optimization of File Migration in Distributed Systems. PhD Dissertation, University of California (Berkeley), 1988. Also available as Technical Report UCB/CSD 88/413, Computer Science Division (EECS), University of California (Berkeley), April 1988.

[27] D. D. E. Long and J.-F. Pâris. Regeneration protocols for replicated objects. In *Proc. of the 5th IEEE Int. Conf. on Data Eng.*, pp. 538-545, 1989.

[28] S. Manly and M. Seltzer. Web facts and fantasy. In *USENIX Symp. on Internet Technologies and Systems*, pp. 125–134, 1997.

[29] J. Mogul. An alternative to explicit revocation? January, 1996. Available at http://weeble.lut.ac.uk/lists/http-caching/0045.html

[30] Network Appliances' NetCache. White paper. Available at http://www.netapp.com/products/level3/netcache/webcache.html

[31] V. Paxson. Measurements and Analysis of End-to-End Internet Dynamics. PhD dissertation. Lawrence Berkeley National Laboratory, University of California. 1997.

[32] M. Rabinovich and I. Rabinovich. RaDaR: A Scalable Architecture for Global WWW Hosting. Draft.

[33] M. Rabinovich and E. D. Lazowska. Improving fault-tolerance and supporting partial writes in structured coterie protocols for replicated objects. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 226-235, June 1992.

[34] J. Reeds. Personal communication.

[35] http://www.sandpiper.com/

[36] D. Sleator and R. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202-208, 1995.

[37] W. R. Stevens. TCP/IP Illustrated. Volume 1, Addison-Wesley, Reading Massachusetts, 1994.

[38] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Beyond hierarchies: design considerations for distributed caching on the Internet. Technical Report TR98-04, Dept of Computer Sciences, University of Texas – Austin, 1998.

[39] UUNET Global Backbone. Available at http://www.uu.net/lang.en/network/

[40] Web Challenger. White paper, WindDance Network Corporation. Available at http://www.winddancenet.com/newhitepaper.html. 1997.

[41] O. Wolfson and S. Jajodia. Distributed algorithms for dynamic replication of data. In *Proc. of Symp. on Principles of Database Systems*, pp. 149-163, 1992.

[42] O. Wolfson, S. Jajodia, and Y. Huang. An Adaptive Data Replication Algorithm. *ACM Transactions on Database Systems (TODS)*, Vol. 22(4), June 1997, pp. 255-314.

Appendix A: Analysis of the Protocol

In this section, we formally analyze the behavior of our protocol. In obtaining some of the claims (especially regarding migrations), we make certain simplifying assumptions in our analysis. We note, however, that these assumptions are only needed for the analysis, and our protocol does not require these assumptions to hold. In particular, our simulation study of Section 7 does not rely on these assumptions. More work is needed to see if removing or relaxing these assumptions would affect any of the claims.

In terms of replica placement and request distribution, our protocol is equivalent to the protocol in which a host obtaining an object copy always creates a new copy, even if it already has one, instead of incrementing the affinity of the existing copy. In this protocol, all replicas have affinity 1, but a host may have multiple replicas of the same object. While the latter protocol is impractically wasteful, it is simpler to reason about. We will consider the latter protocol in this section, with the understanding that replica deletion in this protocol may translate into affinity decrement and replica creation may correspond to affinity increment in a real system. In addition, we will assume that the interval between successive executions of replica placement algorithm (referred to as a *round* below) is the same on every host.

We assume that the access pattern for each object is stable. Formally, (i) for each object x, the total number of requests for x in any round is r_x, where r_x is a nonnegative integer, and (ii) there exists a probability distribution $\mathcal{D}_x$ on the set of clients such that each new request for x originates from a client drawn according to $\mathcal{D}_x$.

We remark that the requirement of a fixed number of requests for an object in any round is used only to highlight main ideas in the analysis. It is a technical matter to extend the analysis to the case where the number of requests for an object x in any round is allowed to vary between $(1-\varepsilon)r_x$ and $(1+\varepsilon)r_x$ for a sufficiently small ε. We restate without proofs our claims, when needed, to cover this case. The main change is that, for the stability results to hold, the constraint between the deletion and replication thresholds becomes $m \geq 4(1+3\varepsilon)u$ (which makes sense intuitively - when we allow a variation in demand we need some extra padding in the difference between u and m). While the load bounds also change, the old bounds may still provide good heuristics for the protocol. Similarly, with minor modifications, our analysis also applies for a model that considers stochastic variations in the demand as long as the probability that the number of requests for an object in any round deviates significantly from the mean is sufficiently small.

The remainder of this section is organized as follows. In Section 8.1, we derive bounds on the change in the load of a host due to the deletion, replication, or migration of a replica. In Section 8.2, we prove that the total number of deletions and replications is bounded. In Section 8.3, we prove under certain assumptions that the system reaches a state in which, with probability approaching one, there is no migration in a given round, provided the number of requests for objects in a round is large enough for probabilistic trends to take hold.

8.1 Load Bounds

The following claim is immediate from the definition of the request distribution algorithm.

Lemma 8.1 *Under steady demand, for any hosts i, j and any object x, access counts $cnt(x_i)$ and $cnt(x_j)$ are such that $0.5 \leq \frac{cnt(x_i)}{cnt(x_j)} \leq 2$.* ■

Lemma 8.2 *Consider the requests for object x in a given round. If the total number of requests issued for x is L, the number of requests directed to replica x_i is ℓ, and the number of replicas of x is n, then we have $\frac{\ell(n+1)}{2} \leq L \leq \ell(2n-1)$.*

Proof: The load on x_i is ℓ. By Lemma 8.1, the load on each of the $n-1$ other replicas of x is at least $\ell/2$ and at most 2ℓ. Thus, L is at least $\ell + (n-1)\ell/2 = \ell(n+1)/2$ and at most $\ell + (n-1)2\ell = \ell(2n-1)$. ■

Corollary 8.2.1 *Consider the requests for object x in a given round. If there are n replicas of x, and the total number of requests for x issued in the round is L, then the load ℓ on an individual replica x_i satisfies the following inequality: $\frac{L}{2n-1} \leq \ell \leq \frac{2L}{n+1}$.* ■

The following lemma is used in Section 8.2 to place a bound on the number of replications and deletions. (In the model where the number of requests for x in any round is between $(1-\varepsilon)r_x$ and $(1+\varepsilon)r_x$, the same lemma holds if m is set to be at least $4(1+3\varepsilon)u$.)

Lemma 8.3 *If host i replicates object x on host j, then u is the lower bound on load of any replica of x after replication, even if other nodes independently replicate or migrate x.*

Proof: Let there be N replicas of x before replication, and n hosts autonomously decide to replicate. Let ℓ be the smallest per-replica load among these n hosts. By the replica placement algorithm, $\ell \geq 4u$. By Lemma 8.1, the load on the remaining $N-n$ replicas must be at least $\ell/2$. Therefore, the total load on x in the round just before replication, L, is at least $n\ell + (N-n)\ell/2$. After replication, n additional replicas are created to obtain a total of $N+n$ replicas.

Since the demand is stable, the total load in any round is L. By Lemma 8.2, the lower bound on replica load is $\frac{L}{2(N+n)-1} = \frac{N+n}{4(N+n)-2}\ell > \ell/4 > u$. ■

Lemmas 8.4 through 8.7 form the basis for the load bounds used in the algorithms for object replication, migration, and offloading (see Sections 4.2.1 and 4.2.2, and Figures 5 and 7). While Lemmas 8.4 and 8.5 establish bounds on the change in the load of a replica due to replications, Lemmas 8.6 and 8.7 establish bounds on the change in the load of a replica due to migrations. (For a model that allows variations in demand by a factor of at most ε, the bound $3\ell/4$ in Lemmas 8.4 and 8.6 should be replaced by $(3+5\varepsilon)\ell/(4+4\varepsilon)$ and the bound 4ℓ in Lemmas 8.5 and 8.7 should be replaced by $4(1+3\varepsilon)\ell$.)

Lemma 8.4 *Let host i replicate object x on host j and ℓ be the load per replica of x at i in the round before replication. Then, in the absence of other replications or migrations, the load on i may decrease by at most $\frac{3}{4}\ell$ per replica of x at i.*

Proof: Let N be the total number of replicas before the replication and n be the number of replicas on i. By Lemma 8.1, the total load on x before replication is $L \geq \ell n + (N-n)\frac{\ell}{2} = \frac{N+n}{2}\ell$. Let ℓ' be the average load per replica of x at i after replication. By Lemma 8.1, we have $L \leq \ell' n + 2\ell'(N-n+1)$. We thus have: $1 \geq \ell(n+(N-n)/2)/(\ell'(n+2(N-n+1)))$. For all nonnegative n and N, it follows that $\ell'/\ell \geq 1/4$, which implies that $\ell - \ell' \leq (3/4)\ell$. ■

Lemma 8.5 *Let host i replicate object x on host j and ℓ be the load on x_i before replication. Then, the load on j after replication may increase by at most 4ℓ.*

Proof: By Lemma 8.2, the total load on x in the round before replication, L, is bounded by $\ell(2n-1)$. The load on j after replication increases by the load due to accesses to x. Since no replicas of x can be deleted by Lemma 8.3, there will be at least $n+1$ replicas of x after the replication (there could be more replicas if other nodes replicate x at the same time). By Corollary 8.2.1, the load on a single replica is bounded by $\frac{2L}{n+2}$. Thus, the load on replica x_j after replication is at most $\frac{2L}{n+2} \leq \frac{2\ell(2n-1)}{n+2} < 4\ell$. ■

The following two lemmas have proofs similar to Lemmas 8.4 and 8.5.

Lemma 8.6 *Let host i migrate object x on host j and ℓ be the average load on i due to replicas of x before migration. Then, in the absence of other migrations and replications, the load on i after the migration may decrease by at most $\frac{3}{4}\ell$ per remaining replica of x at host i.* ■

Lemma 8.7 *Let host i migrate object x on host j and ℓ be the load on i due to one replica of x before migration. Then, in the absence of deletions, the load on j after the migration may increase by at most 4ℓ.* ■

8.2 Stabilization of Deletions and Replications

The following lemma follows directly from Lemma 8.3.

Lemma 8.8 *Under steady demand, after a replication of object x, no replica of x will ever be deleted.* ■

Theorem 1 *Under steady demand, the total number of replications and deletions is bounded.*

Proof: By Lemma 8.8, under stable demand, once replications of an object begin, no deletions may occur. Therefore, due to the upper bound for load on an individual replica given by Corollary 8.2.1, after a bounded number of replications, no replica load will be above $4u$, which is a necessary condition for replication. ■

8.3 Stabilization of Migrations

Our protocol performs two kinds of migrations: (i) geomigrations, and (ii) migrations that occur in the offloading process. In this section, we will prove the stability of the migration process in a probabilistic sense for a slightly more restrictive variant of the protocol presented earlier. In this variant, an object x_s is chosen for geomigration if there is a candidate E such that there exist over a fraction $f > 1/2$ of the requests to x_s for which (i) E is on the preference path, and (ii) x_s is the closest replica. (We note that (ii) is the added restriction.) Moreover, a host accepts a replica for geo- or load-migration only if the upper bound on its load after the migration is below the low watermark (as opposed to high watermark in the protocol presented earlier).

We also assume that for any object x, r_x is $\Omega(R\log(MN))$, where R is the maximum number of replicas created for any object, M is the total number of objects and N is the total number of nodes in the system. Since the object x will be fixed for the following arguments, we omit the subscript x in r_x for convenience. Let n be the number of replicas of x. We say that an event occurs *with high probability*, if the probability of the occurrence of the event is at least $1 - e^{-\Omega(r/R)}$. (Note that this probability may be made arbitrarily close to 1 by choosing r to be sufficiently large.)

Given a set S of replicas for object x and a set C of clients, we define a potential function $\phi_x(S, C)$ as the sum over all clients in C of the distance (in terms of the number of hops) to their closest replica. We note that $\phi(S, C)$ is completely defined by S and C and is independent of the particular request distribution algorithm used. We refer to the time interval between successive invocations of the replica placement algorithm as a *round*.

Lemma 8.9 *Consider the system in the absence of deletions, replications, and offloading. In any round, if there is a geomigration, then the potential of the system decreases with high probability.*

Proof: Let the set of replicas at the start of a round be S. Let the set of clients be C. We evaluate the probability and the effect of a geomigration that moves a replica of object x from host i to j. Let the set of replicas after the supposed geomigration be S'. We divide our analysis into two cases. In one case, we show that a geomigration of x from i to j will not occur with high probability. In the other case, we show that if a geomigration occurs, then the potential of the system decreases.

Let C_1 be the set of clients for which the replica at i is the closest one. Let C_2 be the set of clients c in C_1 for which the preference path from i to c passes through j. Consider any sequence σ of r requests for x. We represent σ by the corresponding sequence of clients $c_1, c_2, \ldots, c_r$, where c_i is the client issuing the ith request. (Note that a client may occur multiple times in σ.)

We now define the notion of a *fair* and a *good* sequence based on the assignment of requests in the sequence to the replicas of x. We say that a sequence σ is a *fair* (resp., *good*) sequence if the number of times in σ that a client in C_1 (resp., C_2) is assigned to x_i is at least an f fraction of the total number of requests assigned to x_i. By Lemma 8.1, the number of requests assigned to x_i is at least $r/2n$. Therefore, the number of requests that are issued from clients in C_1 (resp., C_2) and are assigned to x_i in a fair (resp., good) sequence is at least $fr/2n$.

We consider two cases: $|C_2| \leq |C_1|/2$ and $|C_2| > |C_1|/2$. We first obtain an upper bound on the probability that the geomigration from i to j would occur in the case when $|C_2|$ is at most $|C_1|/2$. In the variant of the protocol we are considering, geomigration from i to j may occur only if σ is good. The probability that σ is good is at least the probability that σ is good given that σ is fair. Given that σ is fair the probability that σ is good is at most the probability that among all the requests that are issued from clients in C_1 and assigned to x_i, an f fraction is issued from clients in C_2. Using standard Chernoff bounds [13] , we obtain that this probability is at most $e^{-(2f-1)^2 r/(6n)}$, which is exponentially small in r/n for any constant $f > 1/2$. Therefore, under the first case, the probability that a sequence is good is at most $e^{-(2f-1)^2 r/(6n)}$. Since n is at most the maximum number R of replicas, we conclude that, with high probability, the geomigration of x from i to j does not take place.

We now consider the case in which $|C_2| > |C_1|/2$. In this case, we show that the potential of the system decreases. Let d_{ij} denote the distance between i and j. For each client in C_2, the distance to the closest replica decreases by at least d_{ij}. For each client in $C_1 \setminus C_2$, the distance to the closest replica increases by at most d_{ij}. Moreover, for every client not in C_1, the distance to the closest replica does not increase. Since $|C_2| > |C_1|/2$, $\phi_x(S', C)$ is at most $\phi_x(S, C) - 1$. This completes the desired claim. ■

We now show that the number of migrations performed during offloading is bounded. We prove this under the assumption that the high and low watermarks are sufficiently apart.

Lemma 8.10 *If $hw > 4lw$, then under stable demand the number of migrations performed during offloading operations is bounded.*

Proof: By Theorem 1, the number of deletions and replications is bounded. Thus, after a bounded amount of time, the only source of increased load on a server is due to migrations. If $hw > 4lw$, it follows from the prerequisites for accepting migrations in the protocol variant being considered and from Lemma 8.1 that the load on any host that accepts a migration can never exceed hw. Therefore, with every migration performed during offloading, the total number of replicas in hosts that may offload decreases. Moreover, this number does not increase with a geomigration, for the same reason . Since the total number of replicas in the system is bounded by a polynomial in the number of objects and nodes in the system (given the bounded number of replications), the desired claim follows. ■

The following theorem follows from Lemmas 8.9 and 8.10.

Theorem 2 *Under stable demand, in a bounded amount of time, the system reaches a state in which in any given round, there is no migration with high probability.*

Proof: The decrease in the potential with every geomigration is at least 1. For any object x, the initial potential is at most the number of clients times the largest distance in the network. Thus, the total initial potential is bounded by a polynomial in N and M.

Consider a given placement of replicas of object x. Under this placement, let p be the probability that a geomigration of a replica of x occurs in a given round. Fix a constant $\alpha > 0$. If $p \leq e^{-\alpha(r/R)}$, then there is no geomigration of x with high probability. Otherwise, within $re^{\alpha(r/R)}/R$ rounds, a geomigration occurs with probability at least $1 - (1 - e^{-\alpha(r/R)})^{re^{\alpha(r/R)}/R} \geq 1 - e^{-r/R}$. Thus, a geomigration occurs with high probability within $re^{\alpha(r/R)}/R$ rounds. It follows from Lemma 8.9 that within $re^{\alpha(r/R)}/R$ rounds, the potential associated with x decreases with high probability. Since the total potential of the system is bounded by a polynomial in N and M and r is $\Omega(R \log MN)$, the total potential is at most $e^{\beta(r/R)}$ (for an appropriately chosen constant $\beta > 0$). It thus follows that after at most $e^{\beta(r/R)} \cdot re^{\alpha(r/R)}/R = e^{O(r/R)}$ rounds the system reaches a state in which in any given round, there is no geomigration with high probability. Since the number of offloading operations is also bounded by a polynomial in N and M, the desired claim now follows. ■

Finally, the following theorem states that the system will never stabilize to a state in which it would have both overloaded and underloaded nodes.

Theorem 3 *Under stable demand, if every node in the system performed* DecidePlacement *without deleting, migrating, or creating any replicas, then either all nodes have a load below hw, or all nodes have a load above lw.* ■

Appendix B: Architecture of the Request Indirection Module

This section briefly outlines the architecture of the request indirection module. Its purpose is to show a feasible implementation. The design rationale, and tradeoffs among possible implementations are detailed in [32] .

Consider an ISP that also provides hosting services. Our description also covers purely hosting systems, since, for our purposes, such a system is equivalent to an ISP system with no customers. The architecture is shown in Figure 12. Users that are the customers of the ISP connect to the Internet through ISP gateways called *points of presence*, or POPs.[7] Packets from users that are not customers of this ISP may in principle enter via any border router, and there is no guarantee that any single point in the network would see all their packets to all destinations.

The request indirection module maps URLs of objects to their replica sets. It also assigns a request for an object to one of the currently available replicas of the object. Its functionality is therefore quite similar to a *name service*. The functionality of the name service is split between two components. One component, a *distributor*, keeps only non-volatile data and can therefore be highly replicated. The other, a *redirection service* keeps the *mapping database* that maps URLs to replica sets. Since this data is volatile, scalability is achieved through partitioning of the database among multiple *redirectors*. The number of replicas of each partition is dictated mostly by availability requirements,[8] and can be low (up to three or four replicas of each segment of the database).

[7] Corporate customers sometimes bypass POPs, but it is always possible to configure internal ISP network in such a way that, for any given customer, there is a point in the network that sees *all* packets from this customer. For our purposes, one can consider this point as the point of presence for this customer.

[8] While volatile, the mapping database is still more frequently read than updated. Thus, a modest number of replicas would still be justified for performance. We envision that every partition will be represented by 3–4 replicas, one in each continent.

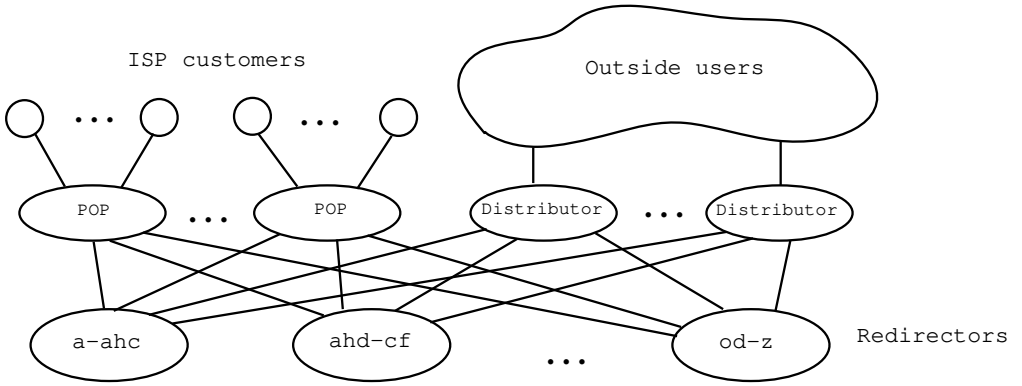


Figure 12: The request distribution service.

The URL name space is partitioned based on some hash function. (A simplest way is to base partitioning on the lexicographical ordering of names, as shown in the figure.) This hash function is known to distributors. Note that while the mapping database is updated frequently, the hash function that partitions the name space among redirectors is very stable.

When an outside user (non-customers) wants to access a page, its Web browser first sends a query to the ISP's DNS servers to resolve the symbolic host name from the requested URL into an IP address. It can be arranged that this DNS server will map all host names into the same set of IP addresses of distributors. In response to the query, the DNS server will send the IP address of a closest distributor. (See [9] for an implementation of DNS name resolution based on the origin of the requester.) Note that the query result can be cached by the client for a long time since the set of distributors usually does not change. Thus, existing caching mechanisms within DNS infrastructure can be used to keep the load on DNS service manageable.

Next, the browser sends the URL request to the IP address received. It arrives at the closest distributor, which sends a query to the appropriate redirector by applying the hash function to the object name. Since there are only a few redirectors for any given URL, the redirector may be far away in the network from the distributor. However, the query and the response are both short control messages, which, unlike transfers of data pages, can be sent as single UDP datagrams. Even when both points are far apart, this causes only a small delay and traffic (an echo UDP packet takes on the order of 100 ms for a coast-to-coast and 150 ms for a cross-Atlantic round-trip on the public Interenet; moreover, an ISP can place redirector replicas so that inter-continental round-trips do not occur in the absence of failures and provide that distributors and redirectors be well-connected within a continent).

Redirectors implement the request distribution algorithm of Figure 2 for their portion of the URL namespace. In response to the above query, the redirector chooses a physical replica for the requested object and sends its ID (a "physical URL") back to the distributor. The distributor now obtains the page from the corresponding host and forwards it to the customer.

Note that a distributor itself can become overloaded. However, since each distributor processes requests for *all* objects from the users in its area, the (aggregate) request arrival rate for distributors is more stable and easier to predict than the request rate for individual objects. Thus, the distributors are easier to provision. Moreover, given that the set of distributors is quite static, DNS-based techniques (e.g., [14, 11]) can be used to temporarily relieve an overloaded distributor until an extra distributor is deployed.

For the customers of the ISP, the role of distributors can be played by POPs directly, using a technique similar to the transparent proxy cache technology [12, 30] . Since the same POP sees all

packets from a given user, this POP can intercept packets for well-known ports (e.g., port 80 for Web pages, 70 for Gopher objects, etc.) and interpret requests for Internet objects. They can then act as distributors, querying redirectors and obtaining and forwarding the objects. Finally, distributors and POPs can cache especially hot objects and service them without contacting redirectors or hosts, using the technique of [7] .