A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service

Michael Rabinovich
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932

Irina Rabinovich
The Dun & Bradstreet Corp.
3 Sylvan Way
Parsippany, NJ 07054

Rajmohan Rajaraman
DIMACS Center
Rutgers University
Piscataway, NJ 08854

Amit Aggarwal
Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

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.

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 (e.g., [23]) 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 (e.g., one of the algorithms in [9]) would create problems when a server is swamped with requests originating from its vicinity: no matter how many additional replicas the server creates, all requests will be sent to it anyway.

The main novelty of our request distribution algorithm is that it allows a hosting server to derive bounds on the number of requests going to potential new replicas based on local knowledge only. This makes the algorithm well-suited for dynamic object replication. Beside allowing a host to decide on object migration and replication autonomously, these bounds enable object relocation to be done *en masse*, without waiting for load observations after each move. Without this, a system of our intended scale would be hopelessly slow in adjusting to demand changes.

Our replica placement algorithm relies on the bounds provided by the request distribution algorithm to avoid vicious cycles of back-and-forth migrations and replications and deletions of objects.

1.1 Past Work

Much work has been done on caching by clients or by proxy servers that provide a shared cache to multiple clients. While clearly useful, caching alone cannot obviate the need for replication within the hosting platform, as evidenced by a wide use of mirrors. First, only static non-personalized objects can be cached. Second, caching is outside control of the service provider, which complicates many issues (such as copyright protection, usage metering, advertisement insertion, update propagation, etc.), giving service providers an incentive to disallow caching even when objects are otherwise cacheable. Overall, numerous studies have shown proxy hit ratios of only around 40%. Caching therefore is complimentary to a service-based solution considered here. In the remainder of this paper, we only consider client requests that missed in their caches.

Many commercial products offer transparent load balancing among Web servers on a local area network. using a front-end IP multiplexing device (see [14] for a now-incomplete survey). Approaches by Katz et al. [23] and Colajanni et al. [11] use DNS servers as request distributors. They also focus on load only and thus target primarily local-area networks.

CISCO DistributedDirector, IBM Network Dispatcher, and WindDance are products that allow request distribution over a wide-area network. They differ in the network point where request distribution is performed (clients, or DNS server, or HTTP redirectors) and server selection algorithms. None of them offer dynamic object replication or migration.

Several research proposals for server selection in a wide-area network have been described (e.g., [15, 32, 8, 33]). Notably, in algorithms by Sayal et al [32] and Fei et al [15], clients choose servers based on previously observed latencies and response times, thereby implicitly accounting for both load and network proximity factors. All these protocols focus on replica selection and do not consider replica placement aspect. Another difference with our approach is that our request distribution algorithm allows hosts to predict effects of potential replica set changes on future request distribution. This makes it well-suited for a system where hosts make dynamic decisions on the number and location of replicas.

Guyton and Schwartz describe various methods that can be used to determine proximity of nodes in the network [17]. Following CISCO Distributed Director, we use routers databases to extract proximity information. Unlike CISCO, we do it asynchronously with client requests, thereby reducing request latency at the expense of potential staleness of the proximity information. One reason for using the routing-based proximity metric is our network-centric view, in which reducing the backbone bandwidth is an overriding concern.

Much of existing work on dynamic replication has concentrated on maintaining system availability during failures [20]. 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 [36] 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 (see, e.g., [26]), 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 [5] and Bestavros and Cunha [6] appear to be the predecessors of WebWave. [5] proposes to reduce network traffic within an intra-net by caching organization's popular objects close to the intra-net'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 [5]. In [6], 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 [4] propose an infrastructure for performing replication on the Web, without describing algorithms for deciding on replica sets. Also, the infrastructure assumes gradual learning 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 [25] for early survey and [3] and references therein 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, Awerbuch, Bartal, and Fiat designed a distributed file allocation protocol and show that it is nearly optimal in terms of total communication cost and storage capacity of the nodes [3]. However, their protocol does not address the load balancing issue.

1.2 Our Contributions

Our main contributions are as follows.

- We present a 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. Our request distribution algorithm allows derivation of bounds on load changes due to any potential object migration or replication. The replica placement algorithm relies on these bounds in making placement decisions autonomously on each node, without even knowing of the existence of other replicas. Moreover, these bounds allow a host to safely decide on replica placement for many objects at once, without waiting to observe actual loads after each move.

- 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

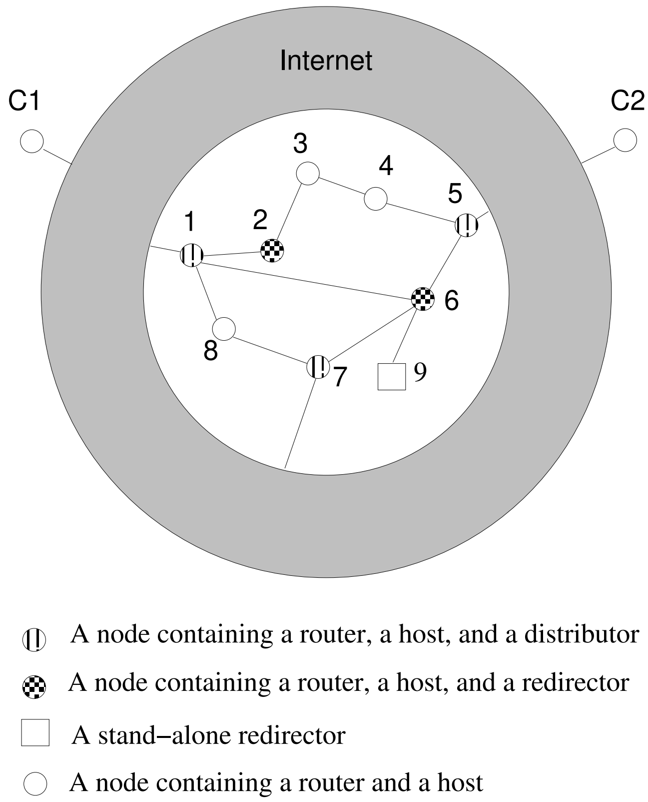


Figure 1: A system model.

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.

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 consider a *Internet hosting system* that maintains and provides access to Web objects. The system contains a set of *nodes*, each consisting of a router and a hosting server connected to each other by a local area network. Hosting servers (hosts for short) maintain Web objects, while routers are connected via wide-area links to form the platform backbone. Some (or perhaps all) of the nodes, called *gateway* *nodes*, have connections to the outside Internet. These are the nodes through which all requests are coming into the system. These nodes contain *distributors*, in addition to routers and hosts. Finally, the system also contains *redirectors*. Distributors and redirectors, whose functions will be described shortly, may be physically co-located on the same machine with hosts.

A request from a client c to any object is directed to the "closest" gateway's distributor, regardless of the requested object, either using CISCO's Distributed Director DNS server [9], or *anycast* [27] when the latter becomes widely available.[1] The notion of "closeness" used in either mechanism is the same as used by Internet routing protocols, so a requested object would be sent to the client through this gateway node anyway. The distributor forwards the request (via a UDP message) to a *redirector*, which chooses a host among those that currently have a replica of the object and forwards the request (again, as a UDP message) to that host. The host sends the object directly to the distributor, which forwards the object to the client. The same redirector is used for all requests to the same object. For scalability, the load is divided among multiple redirectors by hash-partitioning the URL namespace.

Note that, by sending requests for a given object through the same redirector, our protocol does not reduce the distance traveled by requests. Its goal is to reduce the distance for the response message. Since responses carry object data, they are typically much larger than requests and are the determining factor in backbone bandwidth consumption.

For simplicity, we will assume homogeneous hosts. Heterogeneity could be introduced by incorporating into the protocol weights corresponding to relative power of hosts.

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 [29].) Let $s \rightarrow r_1 \ldots \rightarrow r_n \rightarrow c$ is the internal router path from a host s to an external client c. The *preference path* between s and c is a sequence of hosts co-located with the corresponding routers. From now on, we will not distinguish hosts from co-located routers. A message from host s to client c passes by the hosts on the preference path from s to c. 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.

We will assume that a host s knows the preference path from itself to any client c. This information can be statically extracted (and periodically refreshed) from the routing database kept by the platform routers [21]. The routing information also allows comparing the network distance between hosts within the platform.

2.1 Load Metrics

We assume the existence of a uniform load measure that allows load comparison of different servers. 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

[1] The Distributed Director DNS server uses the routes database from the system's routers to resolve the logical name of a Web site (e.g., www.foo.com) into an IP address of a host that is the fewest hops away from the client that issued the query. In the anycast mechanism, a single IP address is used for a set of hosts, but routers advertise routes for this IP that lead to their respective closest hosts.

measure of computational load [24]. In general, the load metric may be represented by a vector reflecting multiple components, notably computational load and storage utilization. Extending our algorithms to allow vector loads is a technical matter.

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.

Finally, load metrics are usually recorded periodically and reflect an average load since the previous measurement. (We will call the interval between consecutive load measurements a *measurement interval.*) So, a load 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 measurement 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. Enabling the derivation of these load bounds is one of the contributions of our protocol.

2.2 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.

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

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

```
ChooseReplica(g_c, x):
/* Executed by the redirector responsible for object x */
  let p be the closest host to gateway g_c that has a replica of x;
  let ratio1 = rcnt(x_p) / aff(x_p)
  let q be the host that has a replica of x with the smallest value of ratio2 = rcnt(x_q) / aff(x_q);
  if ratio1/2 > ratio2
    choose p;
    rcnt(x_p) = rcnt(x_p) + 1;
  else
    choose s;
    rcnt(x_s) = rcnt(x_s) + 1;
  endif
end
```

Figure 2: The algorithm for choosing a replica.

of requests) in the second case. In addition, the algorithm should allow derivation of bounds on load changes due to object relocations. These bounds enable hosts to make replica placement decisions autonomously and for multiple objects at the same time, without waiting to observe the effects of each object relocation. The latter is especially important for the responsiveness of the system.

The algorithm is run by the redirector responsible for the requested object x, and is shown in Figure 2. For each replica x_s, the redirector keeps a count of the number of times it chooses this replica, the *request count* $rcnt(x_s)$. It also maintains the *replica affinity*, $aff_r(x_s)$. 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, instead of creating another replica. (The subscript in aff_r indicates that this variable is kept by the redirector.) We refer to the ratio $\frac{rcnt(x_s)}{aff(x_s)}$ as a *unit 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 unit 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 unit request count of the least-requested replica with the unit request count divided by 2 of the closest replica. (A different constant can be chosen, with corresponding changes to load bounds in the replica placement algorithm below. We explore tradeoffs in choosing this constant in [1]. In this paper, we only show that a somewhat arbitrary constant that makes intuitive sense works well.)

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 inter-spacing (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 redirector resets all request counts to 1 whenever it is notified of any changes to the replica set for the object.

The main strength of this protocol is that is allows derivation of bounds on load effects resulting from object replication and migration. These bounds are given by the following theorems. The proofs and the formal definition of steady demand are given in [30].

Theorem 1 *Let host i replicate object x on host j, i.e., create a new replica of x on j with affinity 1 or, if j already has it, increment its affinity by 1. Then, under steady demand and in the absence of other replications and migrations, the load on i may decrease by at most $\frac{3}{4}\ell$, where ℓ is the load on x_i before replication.*

Theorem 2 *Let host i replicate object x on host j and ℓ be the load on x_i before replication. Then, under steady demand and in the absence of other replications, migrations and deletions, the load on j after replication may increase by at most $4\frac{\ell}{aff(x_i)}$.*

Theorem 3 *Let host i migrate object x on host j, i.e., (1) create a new replica of x on j with affinity 1 or, if j already has it, increment its affinity by 1, and (2) reduce the affinity of x_i by 1 and drop x_i if its affinity becomes 0. Then, under steady demand and in the absence of other replications and migrations, the load on i after the migration may decrease by at most $\frac{\ell}{aff(x_i)} + \frac{3}{4}\ell\frac{aff(x_i)-1}{aff(x_i)}$, where ℓ is the load and $aff(x_i)$ is the affinity of x_i before migration.*

Theorem 4 *Let host i migrate object x on host j and ℓ and $aff(x_i)$ be the load and affinity of x_i before migration. Then, under steady demand and in the absence of replications, migrations, and deletions, the load on j after the migration may increase by at most $4\frac{\ell}{aff(x_i)}$.*

4 Replica Placement Algorithm

A trivial solution to replica placement would replicate every object on every server. However, requiring every server to have enough storage capacity would be too expensive. We will see in Section 6 that significant reduction in network traffic can be achieved at the expense of only few extra replicas. Thus, even if it were feasible to have servers holding the entire Web repository, it is better to spend money on a greater number of inexpensive hosts.

In fact, our request distribution algorithm makes having needless replicas harmful. Indeed, since the algorithm is oblivious to server loads, it distributes requests to all available replicas. Thus, excessive replicas would cause more requests to be sent to distant hosts. Our 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

A host s maintains the following state for every object x_s it keeps.

For each host p that has appeared on preference paths of some requests to x_s since the last execution of the replica placement algorithm, host s keeps the count of the number of these appearances, $cnt(p, x_s)$, referred to as the *access count* of p. In particular, $cnt(s, x_s) = cnt(x_s)$ gives the total access count for x_s.

Host s also maintains replica *affinity*, $aff(x_s)$. We will call a ratio $\frac{cnt(p,x_s)}{aff(x_s)}$ a *unit access count* of candidate p, and a ratio $\frac{load(x_s)}{aff(x_s)}$ a *unit load* of replica x_s.

4.2 The Algorithm

Periodically, a host s runs the algorithm illustrated in figure 3 to decide on replica placement for its objects. There are two pairs of tunable parameters in the protocol: low and high load watermarks for hosts, lw and hw, and deletion and replication thresholds for objects, u and m.

Water-marking is a standard technique to add stability to the system. The high watermark reflects host capacity. Exceeding this load would degrade the performance of a host to an undesirable level. The low watermark must be chosen to be below the high watermark.

Deletion and replication thresholds determine options for replica placement. 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.

To provide stability, these parameters must be chosen subject to a theoretical constraint $4u < m$. The reason for this constraint is provided by the following theorem:

Theorem 5 *If host i replicates object x on host j only when its unit access count on i exceeds m, then, under steady demand, $m/4$ is the lower bound on the unit access count of any replica of x after replication, even if other nodes independently replicate or migrate x.*

Thus, when object replication(s) occur, every replica after replication will have the access count exceeding u, so no replicas will be dropped. Knowing this allows a host to 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. In practice, the m/u ratio should be higher, to prevent boundary effects. We use $m/u = 6$ in our experiments.

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

```
DecidePlacement():
/* Executed by host s */
  if load(s) > hw offloading = Yes;
  if load(s) < lw offloading = No;
  for each x_s
    if cnt(s,x_s)/aff(x_s) < u
      ReduceAffinity(x_s);
    else
      Loop through nodes with non-zero access counts to x_s, in the decreasing order of distance form s.
      For each node p such that cnt(p,x_s)/cnt(s,x_s) > MIGR_RATIO
        send CreateObj("MIGRATE",x_s, load(x_s)/aff(x_s)) to p;
        if p responded with "OK"
          ReduceAffinity(x_s);
          break from loop;
        endif
      endloop
    endif
    if x_s has not been dropped or migrated AND cnt(s,x_s)/aff(x_s) > m
      Loop through nodes with non-zero access counts to x_s, in the decreasing order of distance from s.
      For each node p such that cnt(p,x_s)/cnt(s,x_s) > REPL_RATIO
        send CreateObj("REPLICATE",x_s,load(x_s)/aff(x_s)) to p
        if p responded with "OK"
          break from loop;
        endif
      endloop
    endif
  endfor
  if offloading = Yes AND no objects were dropped, migrated or replicated
    Offload();
  endif

ReduceAffinity(x_s):
  if aff(x_s) > 1
    decrement aff(x_s) and send the new value of aff(x_s) to x's redirector;
  else
    inform x's redirector of the intention to drop x_s;
    if the redirector responds with "OK", drop x_s;
  endif
end
```

Figure 3: Replica placement algorithm.

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. 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.

An affinity unit of an object can be dropped if its unit access count is below a deletion threshold u. (The redirector ensures that the last replica of an object be not deleted, by arbitrating among competing replica deletions and disallowing the last one in the ReduceAffinity procedure.)

An object x_s is chosen for migration if there is a host p that appears in preference paths in $MIGR\_RATIO$ fraction of requests for this object. To prevent an object from migrating back and forth between nodes, $MIGR\_RATIO$ must be over 0.5. We choose 60%, to provide extra stability in the presence of load fluctuations around the 50% boundary.

An object x_s is chosen for replication if it has not been migrated and if its total unit access count exceeds m and there is a candidate that was closer than s to some minimum number of requests, relative to the total number of requests received by s. $REPL\_RATIO$ must be chosen to be less than $MIGR\_RATIO$, for replication to ever take place. We chose $REPL\_RATIO = 1/6$, to ensure that replication is beneficial at least when the object to be replicated is represented by the sole copy, as illustrated below.

Assume s has the sole replica of object x, and replicates x on host p that appeared in 1/6 of its requests. Assuming requests from any given client are evenly spaced in time and neglecting some boundary effects right after the replication, the request distribution algorithm will direct 1/3 of all requests to host p, including all requests that are closer to r. That constitutes 50% of requests serviced by p. If p's access count on s prior to replication exceeded 1/6 of the total, the number of requests closer to p would exceed 50% of all requests p will service after the replication, and therefore the replication would be beneficial.

```
CreateObj(method,x_s,load(x_s)):
/* Executed by candidate p. */
  if load(E) > lw
    send "Refuse" to invoker and terminate;
  endif
  if method = "MIGRATE" and load(E) + 4load(x_s) > hw
    send "Refuse" to invoker and terminate;
  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
  if a new copy x_p has been created or the value of aff(x_p) has changed
    notify x_s's redirector
  endif
  load(r) = load(r) + 4load(x_s);
  send "OK" to invoker;
end
```

Figure 4: Algorithm for creating an object replica.

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 or migrate object x_s on a candidate E, s sends a request to E, which includes the ID of the object to be replicated or migrated and the load on host s generated due to x_s. The candidate accepts a replication request if its load is below low watermark. To accept a migration request, the candidate checks, in addition, if its upper-bound load estimate *after* the proposed migration would be below the high watermark. This prevents migration to a host in cases when a single object migration would bring the recipient's load from

```
Offload():
/* Executed by the offloading host s */
  find a host r with load(r) < lw;
  recipient_load := load(r);
  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 cnt(E,x_s)/cnt(s,x_s) for some E
    if cnt(x_s)/aff(x_s) <= m
      send CreateObj("MIGRATE",x_s, load(x_s)/aff(x_s)) to q;
      if q responded "OK"
        load(s) = load(s) - load(x_s)/aff(x_s) - (3/4) * load(x_s) * (aff(x_s)-1)/aff(x_i);
        recipient_load = recipient_load + 4 load(x_s)/aff(x_s);
        ReduceAffinity(aff(x_s))
      else exit.
    else
      send CreateObj("REPLICATE",x_s, load(x_s)/aff(x_s)) to q;
      if q responded "OK"
        load(s) = load(s) - (3/4) * load(x_s);
        recipient_load = recipient_load + 4 load(x_s)/aff(x_s);
      else exit.
    endif;
  endwhile;
end
```

Figure 5: The protocol for host offloading.

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 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.

If the conditions for migration or replication are satisfied, p creates a copy of x, notifies the corresponding redirector about the new copy, and updates its upper-bound load estimate. The load estimates are based on the fact that, when a host acquires a new replica of an object, extra load imposed on this host is bounded by four times the load of an existing replica of the object [30]. In the case of migration, once the source host s learns that the recipient p created a copy, it removes its copy by executing ReduceAffinity.

Whenever a replica set for an object changes, the corresponding redirector is notified. Note that the redirector is notified of copy creation *after* the fact and of deletion *before* the fact. This sequence of actions preserves the invariant that the replica set recorded by the redirector is always a subset of object replicas that actually exist. This ensures that the redirector never assigns requests to non-existing replicas, without resorting to expensive distributed transactions for updating its 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 is shown on Figure 5. The offloading host s first finds a recipient node. We refer the reader to [30] for details on finding or choosing the recipient. (For the purpose of this paper, one can assume that hosts periodically exchange load reports, so that each host knows a few probable candidates.)

The recipient accepts the offloading request if its load is below low watermark, in which case it responds to the requesting host with its load value. The offloading host then goes through all its objects, starting with those that have a higher rate of "foreign" requests, and attempts to migrate or replicate them to the recipient. Choosing between migration and replication is done similarly to the *DecidePlacement* algorithm. The only difference is that the offloading host does not try to migrate heavily loaded objects - objects with the unit 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. 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 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 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 [13]. These objects can be replicated or migrated freely, provided the location of the primary copy is tracked by the object's redirector. Multiple studies (e.g., [26, 28]) have shown that an overwhelming majority (80–95%) 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

Parameter	Value
Number of objects	10000
Size of object	12KB
Placement decision frequency	Every 100 seconds
Node request rate	40 requests per sec
Server capacity	200 requests per sec
Network delay	10ms per hop
Link bandwidth	350 KBps
High watermark	50 and 90 requests/sec
Low watermark	40 and 80 requests/sec
Deletion threshold u	0.03 requests/sec
Replication threshold m	$6u$, or 0.18 requests/sec

Table 1: Simulation parameters.

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 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, it may still be beneficial to create a limited number of replicas. The protocol itself remains the same, with the additional restriction that the total number of replicas remain within the limit.

6 Performance

We have performed an event-driven simulation of our algorithm to study its performance. We use synthetic workloads for our simulation. [1] presents the results of a trace driven simulation and explores in depth the tradeoffs involved in adjusting the various parameters in the algorithm.

6.1 Simulation Model

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 [34] for the map) contains 53 nodes in North America, Europe, Pacific Rim, and Australia. We assume that all the backbone nodes serve as gateways. In the simulation, each backbone node generates client requests at a constant rate that enter the platform through it. Further, each node in the backbone can service client requests. There are $10K$ documents in the system which are initially distributed among the nodes in the round-robin fashion, so that object i is assigned to node $i \bmod 53$. 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. The redirector is co-located with a node whose average distance in hops to other nodes is minimum. In future, we plan to explore the problem of optimally placing redirectors for different objects in order to minimize the added latency due to them. We assume that all pages are of equal size, and the request size is negligible compared to the page size.

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

- **Zipf.** Previous studies [12, 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 the popularity of the page with rank i is proportional to $1/i$. Thus, in this workload, clients choose pages according to Zipf's law, where the page number corresponds to its popularity rank.[3]

- **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 choose $p = 0.9$. This workload models the situation when entire Web sites vary in popularity.

- **Hot-pages.** In the hot pages workload, all the pages are divided into hot and cold buckets in the ratio $1:9$. A page from the hot bucket is requested with a high probability (0.9). The hot-pages workload models the situation when some objects are uniformly more popular than others.

- **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 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 first-come-first-serve order. A node's load is measured as the rate of serviced requests and is averaged over a period called the *load measurement interval*, which is $20sec$ for our simulation. A request incurs latency due to queuing and processing at the server and network delays.

Table 1 lists simulation parameters for our experiments. There are a number of tradeoffs involved in choosing the parameters which are highlighted and further explored in [1]. We present some of the tradeoffs here and the rationale for choosing particular values:

- The time between successive placement decisions determines the responsiveness of the system. A low inter-placement time leads to high responsiveness; however it also makes the algorithm sensitive to

[3] Precise modeling of Zipf's law requires a large table of points dividing a (0,1) interval into fragments with length equal to each object's popularity, and then determining a fragment to which a randomly chosen point on this interval belongs. Instead, we use a closed-form approximation of Zipf's distribution due to Jim Reeds, where the requested page number is calculated as the value of $e^{u(0,1)*\ln(n)}$ 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 [31]. It gives page popularities within 15% of the actual Zipf's law.

temporary bursts. For our experiments, the inter-placement time is not very relevant since the request rate remains constant. However, in the real world, inter-placement time should be chosen so as to have an acceptable responsiveness and at the same time mask burstiness at small time scales. [16] shows that Internet traffic is bursty at fine time scales (of the order of seconds) but smoothes out for larger time-scales (of the order of minutes or larger). Therefore, we choose an inter placement time of 100 seconds for our simulations.

- Deletion threshold determines how aggressively we migrate or replicate. There is again a tradeoff involved - a low deletion threshold implies aggressive replication and migration; therefore, more replicas are created leading to lower latency for requests. However, it may also increase the overhead traffic and the number of extra replicas. For our simulations, we choose, rather arbitrarily, the deletion threshold to be approximately the rate of requests sent per client per hot document in the hot-pages workload. The replication threshold m should be greater than $4 \times u$ (theorem 5). We choose the threshold to be 6 times the deletion threshold. [1] tries to explore the sensitivity of the algorithm to the replication and deletion thresholds.

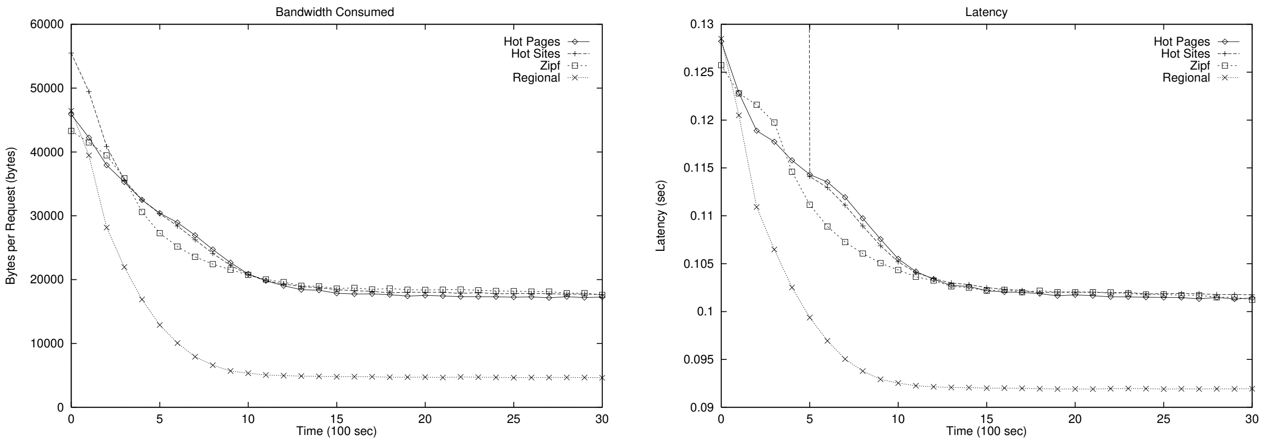
- The constant 2 used in the request distribution algorithm is another parameter that can be varied. [1] explores the tradeoffs involved. In this paper, we choose a value of 2 arbitrarily.

- The watermarks (lw and hw) determine the maximum load on the host. The watermarks along with the inter request time determine the load on the system. We choose the server capacity to be much higher than the watermarks in order to avoid a backlog of messages. A backlog of messages is not representative of the real world since servers normally drop messages or client timeout before queues build up.

- The document size is chosen to be $12KB$, the link delay is assumed to be $10ms$ while the link bandwidth is assumed to be $350KB/sec$. These values are based on preliminary experiments on traces of accesses to pages hosted by AT&T's EasyWWW hosting service and on the Worldnet topology [1]. We kept the number of documents low in order to make the simulation faster.

6.2 Results

Figure 6 depicts the bandwidth consumed and the average response latency for the various workloads. The bandwidth is determined by summing the number of bytes transmitted on each hop. Therefore, the path taken by the response has a greater impact on the bandwidth consumed by a request since the data size is larger than the request size. The bandwidth consumption reduces by 62.9% for hot-pages, 68.3% for hot-sites, 60.1% for Zipf workload and by as much as 90.1% for the regional workload. In the hot-pages, hot-sites and Zipf workloads, every document is requested with the same probability by all the hosts *ie.* a popular document is globally popular. On the other hand, in the regional workload, the probability with which a document is requested by a host depends on the region the host belongs to. Therefore, there is some locality in document access patterns. A document is popular only in a particular region, which allows



(a) Network traffic.

(b) Average response latency.

Figure 6: Performance of dynamic replication.

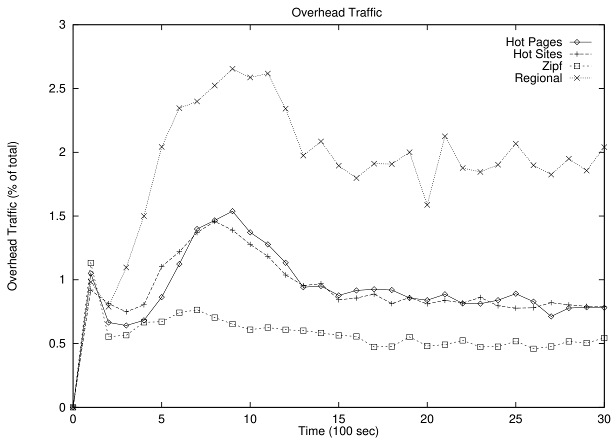
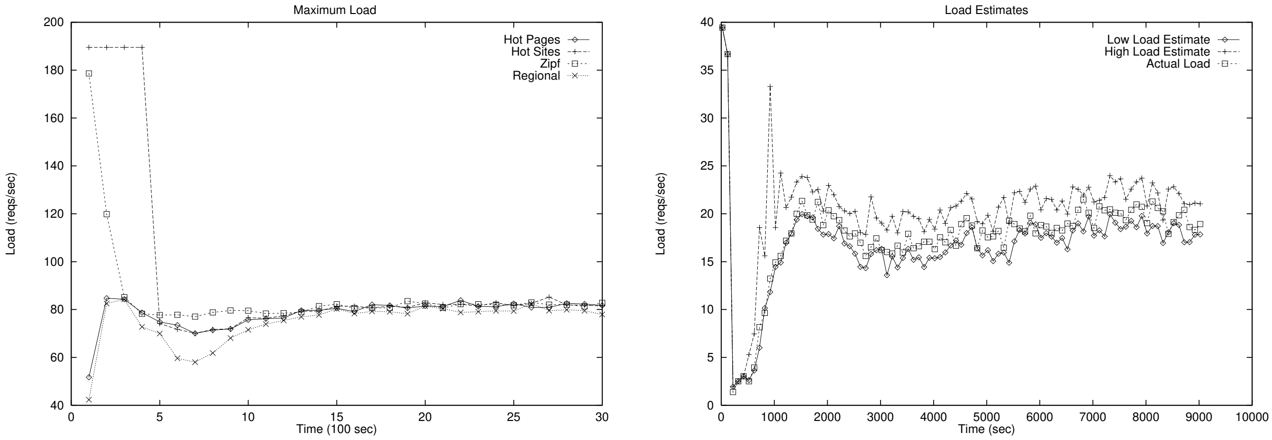


Figure 7: Network overhead.

all the replicas of the document to be concentrated in that region. For the other workloads, the replicas have to be spread out because of the lack of locality in access pattern. This explains the lower bandwidth consumption for regional workloads. Another noticeable feature is that the access pattern for both hot-sites and hot-pages is similar, only the initial configuration of documents differs (in hot-sites, all the hot documents are concentrated at a few sites, while in hot-pages they are well distributed). The equilibrium bandwidth consumption for both the cases is the same, which indicates that the algorithm tries to optimally place replicas based on the access patterns, irrespective of the initial configuration.

The latency reductions also show a similar trend. Average response latency reduces by around 20% for the Zipf and hot-pages workloads and by 28% for the regional workload. The less spectacular improvement in latency as compared to bandwidth is due to the fact that every request goes through the global redirector. For the hot-sites workload, the latency initially is extremely high (in the order of tens of seconds), as all



(a) Maximum Load in the System.

(b) Load Estimates.

Figure 8: Maximum Load and Load Estimates.

Workload	Adjustment Time (min)	Average Number of Replicas
Hot-sites	20	2.62
Hot-pages	22	2.59
Regional	20	1.49
Zipf	23	1.86

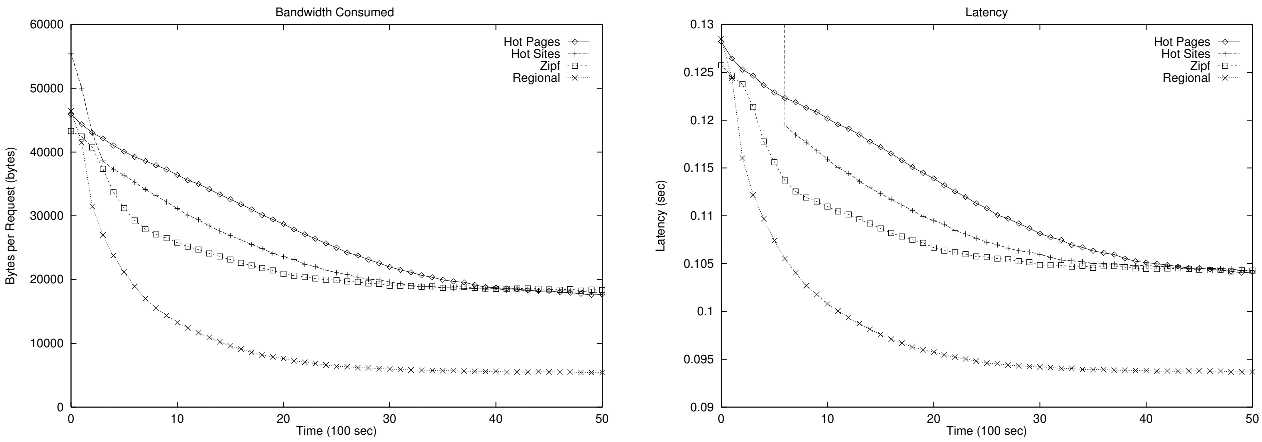
Table 2: Adjustment time and average number of replicas.

requests are queued up at the popular sites. However, it quickly drops to the levels similar to the Zipf and hop-pages workloads, indicating that the algorithm successfully removes these hot-spots.

Figure 7 shows the traffic overhead as a percentage of the total traffic. The overhead, which occurs because of the replication and migration of documents, is always below 2.5% of (already reduced) total traffic. Figure 8a shows the maximum load in the system. The maximum load always remains below the high-watermark of $80 reqs/sec$ which shows that the algorithm successfully distributes load among the servers. Initially, the maximum load for the hot-sites and Zipf workloads is high because a few sites/documents are excessively popular. The algorithm manages to remove these hot spots through replications and migrations. Table 2 shows the average number of replicas created and the adjustment time for the various workloads. We compute the adjustment time as the time it takes to reach a bandwidth consumption that is 10% above the average equilibrium bandwidth consumption. Depending on the workload, the adjustment time is 20 – 23 minutes. Note that significant traffic reductions occur much quicker than that. The number of replicas for all the cases is very low, considering that the number of hosts in the topology is 53 with all the nodes requesting documents at the same rate.

Figure 8b plots the load estimates for one of the hosts in the system along with the actual load. The actual load lies between the high load estimate and the low load estimate which shows that the algorithm is successful in predicting the load.

Figure 9 shows the performance of dynamic replication when the load on the system is high. High load



(a) Bandwidth Consumption.

(b) Average Response Latency.

Figure 9: Performance of Dynamic Replication (High Load)

is simulated by decreasing the high and low watermark to 50 and 40 respectively. This, on average, places the low watermark load on every server. There are two effects of the high load. First, the responsiveness of the system decreases. This is because, with recipient nodes being close to the low watermark, the protocol is unable to relocate multiple objects in one transfer. Second, the performance gains diminish. Although not perceptible in the graphs, the bandwidth consumption increases by about 2% (for hot-sites) to 17% (for regional workload) and so does the average latency as compared to the low load case. 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 has a page accessed mostly from node j and vice versa, it would make sense for these nodes to swap pages - the load would remain the same but proximity would improve. Yet the high load prevents either node from accepting any request for page placement.

In summary, our algorithm decreases bandwidth consumption and average response latency considerably without imposing a high network overhead. It is also successful in removing hot-spots from the network and distributing load among the servers. Our experiments show that the algorithm correctly predicts higher and lower bounds on the load of a host, which is one of the main contributions of our scheme. The performance of the algorithm diminishes as the load increases. [1] further explores the effect of load on the performance as well as the sensitivity of the algorithm to other simulation parameters.

7 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, and Dennis Shasha.

References

[1] A. Aggarwal and M. Rabinovich. Performance of replication schemes on the Internet. Submitted for publication.

[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. Distributed Paging for General Networks. In *Proc. of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pp. 574-583, January 1996.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] R. Carter and M. Crovella. Server selection using dynamic path characterization in Wide-Area Networks. in *IEEE Infocom'97*, 1997.

[9] CISCO DistributedDirector. White paper. Available at http://www.cisco.com/warp/public/751/distdir/

[10] CISCO LocalDirector. http://www.cisco.com/warp/public/751/lodir/

[11] 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.

[12] 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.

[13] 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.

[14] R. Farrell. Distributing the Web load. *Network World*, September 22, 1997.

[15] Z. Fei, S. Bhattacharjee, E. Zegura, and M. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proceedings of IEEE INFOCOM'98*, 1998.

[16] 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.

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

[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] A. Helal, A. Heddaya, B. Bhargava *Replication Techniques in Distributed Systems* Kluwer Academic Publishers, 1996.

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

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

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

[27] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. Request for Comments 1546, November 1993.

[28] J. Mogul, F. Douglis, A. Feldmann and B. Krishnamurthy. Potential benefits of Delta-encoding and data compression for HTTP. In *Proc. of ACM SIGCOMM'97 Conference*, pp. 181–194, 1997.

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

[30] M. Rabinovich, I. Rabinovich, and R. Rajaraman. Dynamic Replication on the Internet. AT&T Labs Technical Report HA6177000-980305-01-TM, March 1998. Available at http://www.research.att.com/m̃isha/radar/tm.ps.gz.

[31] J. Reeds. Personal communication.

[32] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek. Selection algorithms for replicated web servers. Workshop on Internet Server Performance, June 1998. Available at http://www.cs.wisc.edu/c̃ao/WISP98/html-versions/mehmet/SelectWeb1.html.

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

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

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

[36] 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.