Content Replication in Web++

Mehmet Sayal	Peter Scheuermann	Radek Vingralek
Hewlett-Packard Labs	Northwestern University	Oracle Corp.
1501 Page Mill Road	2145 Sheridan Avenue	400 Oracle Parkway
Palo Alto, CA 94034	Evanston, IL 60208	Redwood Shores, CA 94065
sayal@hpl.hp.com	peters@ece.nwu.edu	radek.vingralek@oracle.com

Abstract

Web++ is a prototype system that supports user transparent wide area replication of resources in order to improve the response time and reliability of the HTTP service. Our architecture is based on smart clients, and can be dynamically downloaded as mobile code into a user's application, presents a number of advantages. Clients keep track of the average HTTP latency that they experience from various servers and use that information in order to make to choose the replica of a resource that is expected to deliver the best response time for them. The clients also provide feedback on the observed request latencies to the servers, which allows helps the servers to determine which resources should be replicated and what would be the best locations for the replicas. We describe in this paper a distributed server-initiated approach for resource replication in which all servers can decide autonomously whether to replicate resources and the locations where the replicas should be allocated. In addition to the novel use of smart clients, our algorithm also avoids keeping track of complex network topologies by using the concept of logical segments. We present the results of experiments that show that our algorithm for resource allocation scale well with respect to the number of servers and the number of replicated resources.

Keywords and Phrases

Web resource replication, content distribution networks, smart clients, HTTP.

1 Introduction

1.1 Motivation

The Web makes large quantities of data available for download to end-users. However, the scale of the Web usage is stressing the capacity of the Internet resulting in poor performance and low reliability of Web service. According to Juniper Research, nearly half of the Web users responding to a company survey reported having stopped using a preferred site because of performance difficulties.

Several techniques have been adopted to reduce the response time of the Web service, improve its reliability, and balance load among Web servers. However, the utility of web caches is limited by the relatively low time-locality exhibited by many web access workloads [1, 4, 7]. Server clusters is another possible approach whereby a single dispatcher intercepts all Web requests and redirects them to one of the servers in a cluster. However, since server clusters are typically located within a single LAN, they are suitable only for local load balancing.

Wide-area replication of Web resources is a way to improve the Web response time and reliability that overcomes many of the difficulties of the previous two approaches. In contrast to proxy caches, the replication servers know about each other and can enforce any type of consistency protocol. In addition, the efficiency of replicated servers does not depend on access locality and replication allows for balancing the clients' request among a number of distributed servers. Currently, a number of commercial services, such as Akamai [11], Digital Island, Mirror Image and others, maintain Content Distribution Networks (CDNs) that consist of a collection of replication servers that offload the work from origin servers by delivering content on their behalf. CDNs provide edge delivery of data, i.e., for each request the CDN attempts to locate the CDN server "closest" to the client to serve the request. DNS redirection and URL rewriting are two of the commonly used techniques for directing client requests to a particular server in a CDN [2].

1.2 Resource Replication in Web++

In [13, 14] we introduced a prototype system, termed Web++, that dynamically replicates Web resources on servers distributed on a wide-area network. Our system is similar to CDNs, but it provides some novel features for overcoming some of the shortcomings with content delivery. Most importantly, the architecture of Web++ is based on the concept smart-clients [16], which take an active role in both selecting the best replica and determining which resources should be replicated and what should be the locations of the replicas. In contrast to our architecture, most commercial CDNs deploy passive clients, with all the decisions made by the servers. Although none of the algorithms for replica selection or location used by the commercial CDNs has been published (to our knowledge), we believe that the server-based approach presents several problems:

- It is hard for a server to determine which replica is best for a given client because the server has no information on the response time observed by the client. It has been shown that the commercial CDNs do not make optimal decisions, but rather avoid the worst ones [6]. The Web++ client directly select the replica with the best observed response time.

- It is hard for a server to determine which resources should be replicated and the best locations them because the server does not know which resources are accessed by "far-away" clients. Web++ servers received feedback from the client about the response time they observe and the "closest" server the client knows about. As described in the rest of the paper, the servers base their replication decisions on aggregated feedback they receive from the clients.

In addition, the smart-client architecture allowed us to further reduce the response time perceived by the clients by batching several resources sent from a server to the client into a single HTTP response [14].

We present here our algorithm for dynamic resource replication in Web++. Our system uses a distributed server-initiated approach for resource replication in which all servers can decide autonomously whether to replicate resources and the locations where the replicas should be allocated. Unlike other replication algorithms, ours does not require that the routers keep track of replica locations as is the case with the Webwave algorithm [5], neither do we make use of the area tables kept by the routers as Radar [9]. Our algorithm makes use of the novel concept of logical segments in order to avoid keeping track of complex topologies and to reduce the amount of statistics kept by each server.

The rest of the paper is organized as follows: In Section 2 we describe in a more detail the architecture of Web++. In Section 3 we describe the replication algorithm implemented in Web++. In Section 4 we present performance evaluation of the replication algorithm based on a simulation driven by workload traces obtained from NCSA. Section 5 concludes the paper.

2 Web++ Architecture

The Web++ architecture consists of Web++ clients and Web++ servers as shown in Figure 1. Both client-to-server and server-to-server communication is carried on top of the HTTP 1.1 protocol. Users submit their requests to the client, which is a signed applet executing in a browser. The client sends user requests to the server, which is a servlet executing in a web server. The server returns the requested resource that can be either statically or dynamically generated. Users are not required to install any plug-ins or client-side proxy. Furthermore, commercially available browsers are not modified; the Web++ applet can execute in both Netscape Navigator 4.x and Microsoft Explorer 4.x browsers. Web servers that support servlets can be directly extended with Web++ servlets. Other servers are extended with a server-side proxy that supports servlets.

Before sending HTML resources to the clients, servers substitute each hyperlinked URL with a callback to the client applet and a list of locations, where the hyperlinked resource is replicated. After receiving the HTML resource, the client applet selects the replica location that is expected to provide the best response time for the client using the Refresh algorithm described in [13, 14]. After a timeout the applet fails over to the next fastest server. This process continues until the entire list of replicas is exhausted or the resource is successfully retrieved. Additional detail on the client applet implementation can be found in [13, 14].

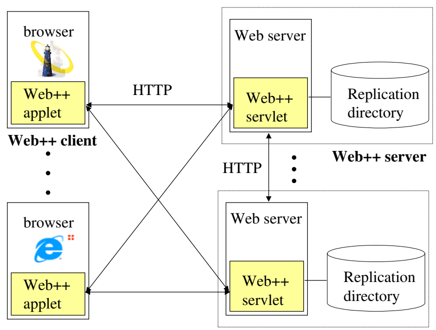

Figure 1: **Web++ architecture.**

The servers create, destroy and update resource replicas. After creation or destruction of a replica each server automatically updates its local *replication directory*, which contains replica locations, and propagates the update to other servers. The servers guarantee eventual consistency of their replication directories in the presence of concurrent updates at different servers. The servers also keep track of the load associated with their resources. The load statistics are used to determine when, where and which resource should be replicated or when a replica should be destroyed.

The server functionality is implemented by extending an existing Web server with a Java servlet. The current implementation of Web++ servlet consists of approximately 24 KB of Java bytecode. we have shown that the overhead of invoking servlets on a Java Web Server does not exceed 18% [14].

3 Resource Replication Algorithm

The main objective of the Web++ resource replication algorithm is to minimize the average waiting time of the Web users by bringing resources closer to the clients who access them frequently. Replication also helps reduce the load of overloaded servers so that their throughput is increased. As a proxy for the waiting time we use the expected *HTTP request latency*, which is defined as the duration between the submission of an HTTP request and the receipt of the first byte of the response [10]. The latency is independent of resource size and still well correlated with the response time [14].

Our algorithm is a distributed, greedy algorithm where servers and clients collaborate in order to collect the necessary statistics and then the servers independently make replication decisions. In principle, each server minimizes the objective function by replicating its local resources that are frequently requested by ”far-away” (i.e. with a high access latency) clients to servers that are ”closer” to the clients.

In order to avoid keeping track of complex network topologies, Web++ uses the novel idea of logical *segments*. A segment corresponds to a server and clients that access the server with a minimal latency among all servers (we call such a server a *home server* of the given client). The introduction of segments substantially simplifies the task of determining the locations of clients and servers in that the servers need to keep track of statistics only on a segment basis. This network segmentation is illustrated in Figure 2.

After a Web++ client picks a home server, it includes a reference to its home server in all of its HTTP requests. Servers use that information to estimate the approximate location of the client. We use the term *segment* to represent a Web++ server and the group of clients who chose that server as their home server

Web++ clients help the servers in resource replication/deletion decisions by piggybacking the HTTP request latencies of previous server responses together with their following requests. Web++ servers use these piggybacked response latencies in order to compute aggregate statistics about the response latencies observed by clients in each segment, for each aggregated resource. Specifically, a server collects the following statistics:

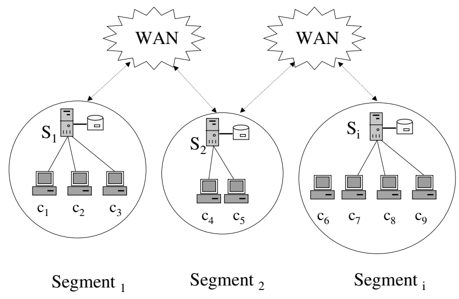

Figure 2: The network segmentation.

- L_{r_c,s_i} - aggregate latency for resource r_c experienced by clients in segment s_i.

- N_{r_c,s_i} - aggregate number of requests for resource r_c received from clients in segment s_i.

Figure 3 shows the interaction between a client and a server in updating these statistics. When a client from segment s_i issues a request for resource r_c it also piggybacks the latency t of its previous response, say to resource r_d that it received from the server. Server uses this information in order to update the two aggregate statistics kept for resource r_d.

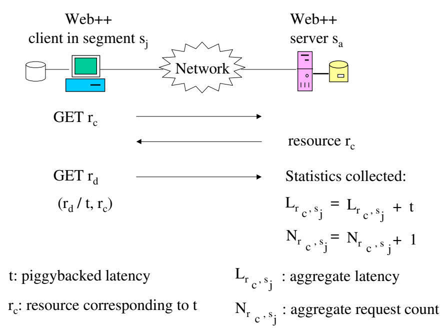

Figure 3: Latency piggybacking that is used to collect the statistics by the servers.

A servers also compute higher level aggregate statistics:

- $L_{r_c} = \sum_{s_i} L_{r_c,s_i}$ - the overall waiting time for resource r_c caused by the replica of resource r_c on the server.

- $N_{r_c} = \sum_s N_{r,s}$ - the total request count for resource r_c on the server.

- U_{r_c} - the number of update requests issued for resource r_c.

In order to adapt to changes in client request patterns, the Web++ servers need a mechanism to diminish the effect of counts and piggybacked latencies that were measured long time ago. We choose to apply an "aging" factor to the computed statistics by periodically dividing the aggregate L_{r_c,s_i}, N_{r_c,s_i} and U_{r_c} values by a constant integer value K. Using the division operation periodically, the contribution of old latencies, request or update counts to the aggregate values can be reduced smoothly over the time. The constant denominator K can be adjusted as function time interval between two consecutive division rounds. We observe here that other aging methods have been used for distributed file systems. The sliding window method used in [15] in which a number of the most recent values are kept for each statistic has proven quite effective. The reason for choosing a single aging factor here instead of a sliding window is due to the requirement to reduce the storage overhead involved with the statistics kept by the servers.

The distributed resource replication algorithm consists of two components, one corresponding to the *source* server and the other to *destination* server. Each server plays dual roles, i.e., of source and destination server. In the source role, a server periodically invokes a daemon in order to choose the optimal resource under its control to be replicated as well as the destination segment. In the destination role, each server determines whether to accept or not the request for replication coming from a source server.

Figure 4 shows the decision making process involved at a source server. It first applies the division operation mentioned above in order to diminish the effect of older measurements of waiting time, request count, and update count. The algorithm chooses a resource r_c and destination segment s_t for which L_{r_c,s_t} is maximal subject to a number of constraints. The first constraint assures that the segment selected as a destination does not have a replica of the resource. The second constraint is that the that the total request count N_{r_c} of the resource is at least an order of magnitude larger than its update count U_{r_c}. Once a candidate resource/segment pair is selected, the source server sends a POST request to the server in the destination segment that includes statistical information such as the average latency for resource r_c experienced by clients in segment s_t. The server in the destination segment then uses these propagated statistics as well as his local statistics in order to decide whether to replicate the resource.

Resource Replication Algorithm at source server :
let r_c := candidate resource for replication ;
let s_t := candidate destination segment for replication ;
let K := division constant for aging of old measurements;
let L_{max} := 0;
forall replicas r **do**
  divide U_r by K;
  if ($N_r < U_r$) **then**
    mark the local replica of resource r as "removable";
  fi
  forall segments s **do**
    divide $L_{r,s}$ by K;
    divide $N_{r,s}$ by K;
    if ($L_{r,s} > L_{max}$ and $s_a vailable$ and $N_r >> U_r$ **then**
      $L_{max} := L_{r,s}$; $r_c := r$; $s_t := s$;
    fi
  od
od
if ($L_{max} > 0$) **then**
  send POST request to the server of segment s_t for
  replicating resource r_c and include the values of
  L_{r_c,s_t} , N_{r_c,s_t} , U_{r_c} and $size(r_c)$;
fi

Figure 4: Web++ resource replication algorithm at source server

Figure 5 shows the steps of the replication algorithm at the destination server s_t. The replication decision that is made by the source server needs to be confirmed by the destination server, because each server collects statistics about the request traffic individually; and in order to minimize the communication overhead, servers do not exchange data about the request traffic until a replication decision is initiated by a source server. Therefore, the traffic seen by the source server may be different from the traffic seen by the destination server. The destination server saves the statistics sent by the source server so that it can have some initial statistics about the new replica r_c until it starts itself accumulating more data about it. In order to decide whether to accept the resource for replication, s_t first checks if it has enough storage space to accommodate the new replica r_c. If this is the case, then the replication request is accepted. Otherwise, it compares the expected gain of accepting the new replica with the expected loss of dropping any one of the local replicas. The gain of accepting a new replica is measured as the estimated reduction in the overall waiting time of Web users. Similarly, the loss from dropping an existing replica is measured as the estimated increase in the overall waiting time of Web users. If the server determines that it is beneficial to accept the new replica r_c it removes as many local replicas r_d as needed in order to create room r_c. The local replicas are chosen in increasing order of estimated loss and the new replica is accepted if the gain of replicating it exceeds the total loss due to dropping the designated local replicas.

Resource replication algorithm at destination s_t:
Let r_c be the candidate resource for replication store sent by the source server.
If s_t storage space is not sufficient **then**
compute $gain(r_c)$, the estimated gain caused by accepting the resource:

$$gain(r_c) = \bar{L}_{r_c,s_t} - \bar{N}_{r_c,s_t} \cdot \frac{\sum_d L_{r_d,s_t}}{\sum_d N_{r_d,s_t}}$$

for all local replicas r_d
Subsequently, **compute** $loss(r_d)$, the estimated loss from droping a local replica r_d:
$loss(r_d) = N_{r,s_t} \cdot D_r$
Let D = list of local replicas r_d for which $gain(r_c) > loss(r_d)$;
If D is not empty then
Sort D in increasing order of $loss(r_d)$;
Let C = prefix of D such that $size(C) >= size(r_c)$;
If $gain(r_c) > loss(C)$ **then**
drop local replicas from C;
download r_c and update replication directory;
fi
Else reject r_c;
fi
fi

Figure 5: Web++ resource replication algorithm at the target server

The gain of accepting a new replica is calculated as follows. The destination server receives in the replication request the following statistical data which was measured by the source server: total waiting time $\bar{L}_{r_c,s_t}$, the request count $\bar{N}_{r_c,s_t}$, update count $\bar{U}_{r_c}$, and the size Z of the resource r_c. The total waiting time for the new replica at s_t is estimated as the product of the request count $\bar{N}_{r_c,s_t}$ and the average response latency provided by s_t to its local clients. Note that the average latency provided by server s_t to its local clients for the existing replicas is used as a proxy for the average latency for the new replica r_c. Thus, in order to compute the gain server s_t uses a combination of local statistics and those gathered by the source server.

The loss from dropping a replica is a little harder to estimate. In order to estimate the loss for a particular replica r_d, the destination server needs to estimate the total waiting time that will be provided by the next best replica for the local clients. Figure 6 explains what happens when a replica is dropped by the destination server. Local clients of the destination server start requesting the dropped resource from the closest server to them that has the replica, which would most likely be the closest replica location to the destination server. Straight lines show user requests before dropping a local replica, and the dotted lines show user requests after dropping it. Some clients might choose different replica locations, but most of them are going to choose the closest server to the destination server since the destination server is their home server. The calculation of $loss$ from removing a replica d assumes that the distance of a local client to the closest alternative replica location for resource r_d is equal to the sum of the distance of the clients' home server to the alternative replica location and the clients' distances to the home server. Thus, the loss is computed by adding to the latency of every single request from local clients for r_d the latency between the destination server and the next best replica location. Since all servers measure their distances to each other and it is possible to find all replica locations for a given resource, the destination server can easily determine the latency of the closest replica location for a particular resource r_d. Let us denote this distance D_r.

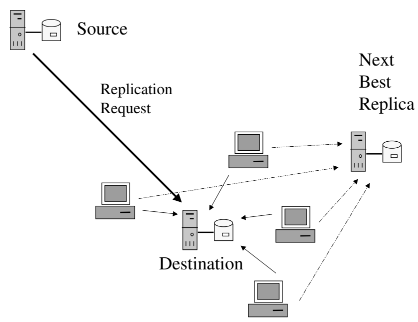

Figure 6: An example scenario of dropping a local replica for a new one.

3.1 Replica Consistency Maintenance

The servers also propagate updates among resource replicas. We have chosen to propagate updates lazily and guarantee a weaker consistency criteria, *eventual consistency*, which requires that all replicas converge towards the same state. Since only the primary replica can be updated, the implementation of eventual consistency in Web++ is greatly simplified. We feel that eventual consistency is sufficient for web pages, because the users are already trained to cope with data inconsistency. However, we felt that Web++ should not make the matters any worse. Therefore, Web++ also guarantees monotonic reads [12] for each client session, which assures that the client sees only progressively newer versions of a resource. Both eventual consistency and monotonic reads session guarantee are implemented using version numbers, which are generated each time the primary replica is updated. Eventual consistency is enforced by requiring that servers never accept updates that contain lower version numbers than their local replicas. The monotonic reads session guarantee is enforced by piggybacking with each client request the highest version of the resource it has seen. If a server has a lower version of the resource, the client is redirected to the primary replica, which is by definition current.

4 Experimental Evaluation

We have evaluated the performance of the replication algorithms described in Section 3. Although Web++ is implemented as a standalone system consisting of Java applets and servlets, we were unable to set up a test framework consisting of a sufficient number of sites to test the scalability of the replication algorithms. Therefore, we used a simulator driven by web server traces collected at NCSA (National Center for Supercomputing Applications).

4.1 Experiment Setup

The simulator is based on the INSANE (Internet Simulated ATM Networking Environment) network simulator developed at the University of California, Berkeley [8]. INSANE was designed to test various IP-over-ATM algorithms with realistic traffic loads. It simulates network traffic in detail; even individual ATM cell transmissions and their queueing delays at routers are simulated. We have modeled the global network topology using the methodology described in [3]. The complete virtual network space is divided into domains and sub-domains, which yields a hierarchical domain structure very similar to that of the current Internet hierarchy. The sub-domains at the leaf level were modeled as fast Ethernet LANs, the interconnects at higher levels of the hierarchy were modeled as ATM OC3 links[1]. The domains and sub-domains are connected to each other through gateways and routers. The topology of the network structure has the internet backbones at the highest level, which are connected to domains through gateways. The domains and sub-domains have various topologies implemented, such as star networks, ring networks, and fully and partially connected point-to-point networks. The clients and servers are located inside sub-domains, which are connected to each other through routers in order to construct the domains. The topology of each domain and sub-domain is selected randomly.

We inserted servers and clients into the sub-domains after the whole domain structure was developed, as it is described in [3]. The servers and clients are independently assigned to sub-domains. That means, a sub-domain may contain only clients, only servers, or both. We have performed tests with zero to ten servers per sub-domain, which are the typical values for most sub-networks (LANs) in the Internet. The number of sub-domains in the network architecture was chosen as 10, 100 and 1000 for separate runs of the simulation tests.

A subset of the simulation parameters is shown in Figure 7. The tests were carried out with the requests generated from the traces obtained from NCSA's six Web servers located at University of Illinois, Urbana-Champaign. The load of NCSA's Web site was approximately 400,000 requests per day during the collection of our traces. We have observed that the same traffic pattern repeats itself everyday, except that there is less traffic during the weekends. Moreover, the same pattern repeats itself every week. Based on those observations, we have re-played the traces multiple times during each run of the simulation experiments in order to both increase the number of requests and provide a simulation of a longer period of time than the actual trace length.

Workload parameters	
resource sizes	2-200 KB
min. request count	500-10,000
replication interval	1-180 min.
#resources	100-10,000
Network parameters	
#servers	10-1000
#clients	100,000
#sub-domains	10-1000
LAN link bandwidth	10-100 Mbits/sec
LAN topology	star, ring, p-to-p
ATM link bandwidth	1.536 Mbits/sec
ATM signaling- admission delay	5 ms
ATM signaling- commit delay	2 ms
ATM signaling- release delay	2 ms
#data ports / ATM switch	8
Max #virtual circuits / ATM switch	1024
# ATM switch priority levels	4
port connection delay at routers	1 ms

Figure 7: Simulation parameters.

We first analyzed scalability, in terms of the effect of the number of servers and clients on the Web++ performance. Figure 8 shows the average response time after Web++ service reaches a stable state for different number of servers. A stable state means that no more replications are needed under a given request traffic. The tests were repeated for 10, 50, 200 and 1000 servers. Because of virtual memory limitation, we could not run the tests with more than 1000 servers and 10,000 clients. However, we simulated a larger client population (100,000

[1] Although modern backbones have much higher capacities than OC3, we have modeled a closed system with all the bandwidth available to carry the simulated workload

clients) by repeating the request patterns of clients multiple times using the limited number of clients we could fit into virtual memory. The number of clients in the real Internet environment is virtually infinite. The tests summarized in Figure 8 had 100,000 clients, 10,000 documents, 10,000 requests as the lower limit LL, and a periodic replication interval of 600 seconds. Although the ratio of clients to servers seems to be small, the clients we used in the tests generated requests very frequently in order to imitate the request load of a larger population. In this experiment, we wanted to observe the decrease in the average response time as the number of servers increase, where the number of clients and sub-domains stay the same. This experiment tries to show, given the network topology of the global network (or at least a reasonably large subset of it), it is possible to reduce the average response time observed by the clients using Web++ architecture. It also tries to observe the affect of increasing number of servers on the average response time. The average response time without Web++ resource replication was 14.5 seconds. The reduction in the response time is remarkable up to 50 servers. The reduction speed decreases as the number of servers increase because the average client-server distance in the network does not decrease much after a certain number of servers. This experiment shows one important observation, in addition to the response time reduction, that the addition of a small number of servers can reduce the average response time significantly. Addition of extra servers does not affect the average response time much after a certain point. Therefore, Web++ can achieve a reasonably satisfactory response time reduction with a relatively small number of servers, compared to the number of clients.

We have performed various experiments in order to observe the effect of different parameters on average response time. For example, we ran experiments to see how the number of clients, sub-domains, or resources individually influence the average response time, while other parameters are fixed. We cannot provide all of the results due to space limitation, but we would like to show one interesting example here. The experiments that observe the effect of changing number of resources with a fixed number of servers, clients, and sub-domains, showed a somewhat unexpected result. We used 10 servers and 1000 clients, and kept the remainder of the parameters the same as those of the first experiment above. Our expectation in this experiment was to observe more than linear (at least quadratic) increase in response time when the number of resources that needs to be hosted increases linearly. Figure 9 shows that the average response time increases approximately linearly with the number of resources. We believe that one reason for linear increase comes from an important aspect of web traffic: most client requests received by a web site concentrate on a small subset of the resources hosted by the site. This observation is also the main reason why web proxy caching and resource replication algorithms can be efficiently used for response time reduction. Therefore, an increase in the actual number of resources that are hosted by a web site implies a much smaller increase in the number of very hot documents that need to be cached or replicated. Another reason for linear increase comes from the fact that Web++ clients can automatically select the best server for requesting resources on the fly. Therefore, when one of the servers start getting overloaded with too many requests and providing poor response time, the Web++ clients automatically start using other servers that respond faster. This capability of Web++ clients avoids rapid increase in load of individual servers, which helps keep the increase in response time at a reasonable level.

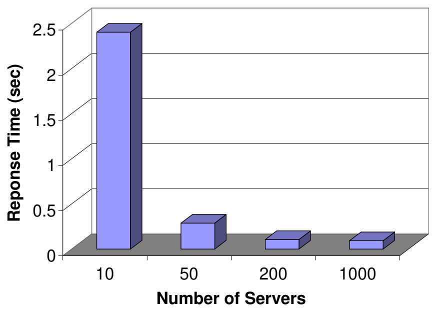

Figure 8: The average stable state response time with respect to different number of servers.

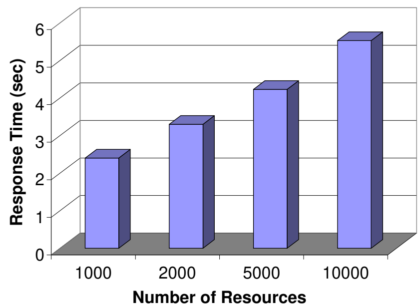

Figure 9: The average stable state response time with respect to different number of resources.

5 Conclusion

We have presented a distributed algorithm for resource replication on the Web where servers and clients collaborate in order to collect the necessary statistics and the servers make independent replication decisions. Each server acting as a replication source periodically invokes a daemon and finds the resource which has the highest aggregate latency with respect to a given logical segment. The server in that segment is then selected as the replication destination. The destination server accepts the resource only if the expected waiting time of its clients is reduced.

Our current implementation of the algorithm considers only the addition of resources and not the addition of servers. In practice, if a server exceeds a given load it is not beneficial to keep storing new resources on it. Instead, we need to consider the addition of servers to our available pool as well as their removal based on the user request pattern. In addition, we plan to experiment with a variant of the algorithm that uses clusters of resources as the units of replication. This can reduce further the communication overhead since clients can request multiple resources from the same server through the same connection. Furthermore, this also allows us to reduce even more the storage overhead for the request statistics kept by each server.

6 Acknowledgements

This research was supported in part by NSF grant CCR-9902023 and by a grant from Usenix Association.

References

[1] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox. Caching proxies: Limitatons and potentials. *Computer Networks and ISDN Systems*, 28, 1996.

[2] B.Krishnamurthy, C. Wills, and Y. Zhang. On the use and performance of content distribution networks. In *Proceedings ACM SIGCOMM Measurement Workshop*, 2001.

[3] K. Calvert, M. Doar, and E. W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, June 1997.

[4] B. Duska, D. Marwood, and M. Feeley. The measured access characteristics of world-wide-web client proxy caches. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[5] A. Heddaya and S. Mirdad. Webvawe: Globally load balanced fully distributed caching of hot published documents. In *Proceedings 17th IEEE Intern. Conference on Distributed Computing Systems*, 1997.

[6] K. Johnson, J. Carr, M. Day, and M. Kaashoek. The measured performance of content distribution networks. In *Proceedings the 5th International Web Caching and Content Delivery Workshop*, 2000.

[7] T. Kroeger, D. Long, and J. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[8] B. Mah. *Quality of Service and Asynchronous Transfer Mode in IP Internetworks*. PhD thesis, University of California, Berkeley, CA, December 1996.

[9] M. Rabinovich and A. Aggarwal. Radar: A scalable architecture for a global web hosting service. In *Proceedings 8th World Wide Web Conference*, 1999.

[10] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek. Selection algorithms for replicated web servers. In *Proceeding of the Workshop on Internet Server Performance*, 1998. available at `http:// lilac.ece.nwu.edu:1024/ publications/ WISP98/ final/ SelectWeb1.html`.

[11] Akamai Technologies. Internet bottlenecks: the case for edge delivery services. available at `http://www.caching.com`.

[12] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, 1994.

[13] R. Vingralek, Y. Breitbart, M. Sayal, and P. Scheuermann. Web++: A system for fast and reliable web service. In *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999.

[14] R. Vingralek, Y. Breitbart, M. Sayal, and P. Scheuermann. Web++ architecture, design and performance. *World Wide Web Journal*, 3, 2000.

[15] R. Vingralek, Y. Breitbart, and G. Weikum. SNOWBALL: Scalable storage on networks of workstations. *Distributed and Parallel Databases*, 6(2), 1998.

[16] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using smart clients to build scalable services. In *Proceedings of USENIX'97*, 1997.