Web++: A System For Fast and Reliable Web Service

Radek Vingralek[a]
Yuri Breitbart

Information Science Research Center
Bell Laboratories - Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
{rvingral,yuri}@research.bell-labs.com

Mehmet Sayal
Peter Scheuermann

Northwestern University
ECE Department
2145 Sheridan Avenue
Evanston, IL 60208
{mehmet,peters}@ece.nwu.edu

[a]Current affiliation: STAR Lab, InterTrust Technologies, 460 Oakmead Parkway, Sunnyvale, CA 94086, rvingral@intertrust.com.

Abstract

We describe the design of a system for a fast and reliable HTTP service termed Web++. Web++ achieves high reliability by dynamically replicating Web data among multiple Web servers. Web++ selects a server which is available and that is expected to provide the fastest response time. Furthermore, Web++ guarantees data delivery, provided that at least one server containing the requested data is available. After detecting a server failure, Web++ client requests are satisfied transparently to the client by another server. Web++ is built on top of the standard HTTP protocol and does not require any changes either in existing Web browsers, or the installation of any software on the client side. We implement a Web++ prototype; performance experiments indicate that Web++ improves the client response time on average by 36.6%, and in many cases by as much as 59%, when compared with the current Web performance.

1 Introduction

1.1 Motivation

The success of the Web has proven the value of sharing different types of data in an autonomous manner. The number of Web users, servers, and total Internet traffic have been growing exponentially in the past 5 years [1]. The scale of Web usage is stressing the capacity of the Internet infrastructure and leads to poor performance and low reliability of Web service. Multisecond response times for downloading a 1KB resource are not unusual [35]. Furthermore, recent studies [29] indicate that server mean time to failure (MTTF) is 15 days, thus a client accessing 10 servers may experience a failure every 36.4 hours. Such a failure rate is not acceptable for many important Web applications such as electronic commerce and online stock trading. Recently, several techniques have been adopted to reduce Web response time, improve its reliability, and balance load among Web servers. Among the most popular approaches are:

Proxy server caching. Proxy servers intercept client requests and cache frequently referenced data. Requests are intercepted either at an application protocol level (*non-transparent caches*) [20, 30] or a network protocol level (*transparent caches*) [13]. Caching improves the response time of subsequent requests that can be satisfied directly from the proxy cache.

Server clusters A single dispatcher intercepts all Web requests and redirects them to one of the servers in the cluster. The requests are intercepted at the network protocol level [12, 16]. Since a server cluster typically is located within a single LAN, the server selection is mostly based on server load and availability within the cluster.

DNS aliasing A single host name is associated with multiple IP addresses. A modified DNS server selects one of the IP addresses based either on round-robin scheduling [26], routing distance, or TCP/IP probe response time [14].

Each of the proposed solutions, however, improves request response time, reliability, or load balancing among servers, but does not address all these issues together. Furthermore, many of the proposed solutions often introduce additional problems. Proxy server caching improves request response time, but it introduces potential data inconsistency between the cached data and the same data stored at the server. Non-transparent proxy caches create a single point of failure. Server clusters improve reliability at the server end, but do not address the reliability of the network path between the cluster dispatcher and a client. Server clusters are not suitable for balancing a load among geographically replicated Web servers because all requests must pass through a single dispatcher. Consequently, server clusters only improve the load balance among the back end Web servers. Finally, although DNS aliasing improves both request response time and service reliability, it forces data providers to replicate the entire Web site. This is impractical for two reasons: (1) since most Web servers exhibit skewed access pattern [32], replicating the entire Web server could be an overkill; (2) in some cases it is not possible or desirable to replicate all dynamic services. In addition, the DNS aliasing implementation becomes problematic when client-side DNS agents cache results of DNS queries or submit recursive DNS queries.

1.2 Paper Preview

One way to improve Web performance and reliability is to replicate popular Web resources among different servers. If one of the servers fails, clients satisfy their requests from other servers that contain replicas of the same resource. Client requests can be directed to the "closest" server that contains the requested resource and thereby improve the request response time. Replication also allows the balancing of clients' requests among different servers and enables "cost-conscious scalability" [9, 42] of the Web service whereby a surge in a server load can be handled by dynamically replicating *hot* data on additional servers.

In this paper we present an overview of design of our Web++ system for replication of the HTTP service. Unlike other similar systems reported in literature, Web++ is completely transparent to the browser user and requires no changes to the existing Web infrastructure. Web++ clients are downloaded as cryptographically signed applets to commercially available browsers. There is no need for end-users to install a plug-in or client-side proxy. There is no need for any modification of the browser; 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. All client-to-server and server-to-server communication is carried on top of HTTP 1.1. Other salient features of Web++ are:

Reliability Resources are replicated among multiple Web++ servers. If one of the servers fails, clients transparently fail-over to another server that replicates the requested resource. After a failure repair, the server transparently returns to service without affecting clients. Furthermore, Web++ guarantees data delivery if at least one of the servers holding the requested resource is available.

Fast response time User's requests are directed by Web++ to the server that is expected to provide the fastest response time among all other available servers where the resource is replicated. This is done transparently to the user and the user is not required to know which server has delivered the resource.

Dynamic replication If there is a high demand for a resource, the resource can be dynamically replicated on another server that is lightly loaded or close to the clients that frequently request the resource. Furthermore, when demand for a resource drops, some servers may drop the resource copy. Additional servers may be recruited from a pool of underutilized servers to help sustain a load peak.

Light-Weight Clients The client applets maintain very little state information. In fact, the only information that they maintain is the HTTP latency of the various servers. This allows our system to be run on many hardware configurations with limited resources.

2 Web++ Architecture

The Web++ architecture is shown in Figure 1. It consists of Web++ clients and Web++ servers. Both client-to-server and server-to-server communication is carried on top of the standard HTTP 1.1 protocol [18]. Users submit their requests to Web++ client, which is a *Smart Client* [44], i.e., a standard Web browser (Netscape Navigator 4.x or Microsoft Internet Explorer 4.x) extended by downloading a cryptographically signed applet. The Web++ applet must be signed so that it can execute outside of the security "sandbox". The Web++ client sends user requests to a Web++ server, which is a standard HTTP server extended either with a Web++ servlet or a server-side proxy. The Web++ server returns the requested resource that can be either statically or dynamically generated.

Each Web++ resource contains a set of *logical URLs* to reference other resources. Before resource retrieval, each logical URL is bound to one of the *physical URLs* that corresponds to one of the resource replicas. A *physical URL* is the naming scheme currently used on the Web as specified in [6]. A *logical* URL is similar to Uniform Resource Name (URN) [39] in that it uniquely identifies a single resource independently of its location. Unlike the URN specification, there are no restrictions on syntax of logical URLs. In fact, a physical URL of resource replica can also be considered as a logical URL[1]. The only difference between a logical and a physical URL lies in their interpretation at resource retrieval time. While a physical URL is directly used to retrieve a resource, a logical URL first must be bound to a physical URL. The binding is done by the Web++ applet that executes within the user's browser.

[1] In Section 3.3 we explain why it may be useful to use one of the physical URLs as a logical URL.

After receiving a resource, the Web++ applet intercepts any events that are triggered either due to the browser's parsing of a resource or due to a user following logical URLs embedded in a retrieved resource. For each logical URL the applet finds a list of *physical URLs* that correspond to the resource's replicas. The list is embedded into the referencing resource by the Web++ servlet. Using the resource replica selection algorithm, the applet selects the physical URL that corresponds to a resource held by an available server that is expected to deliver the best response time for the client. If, after sending the request, the client does not receive a response, the applet fails over to the next fastest server. This process continues until the entire list of physical URLs is exhausted or the resource is successfully retrieved.

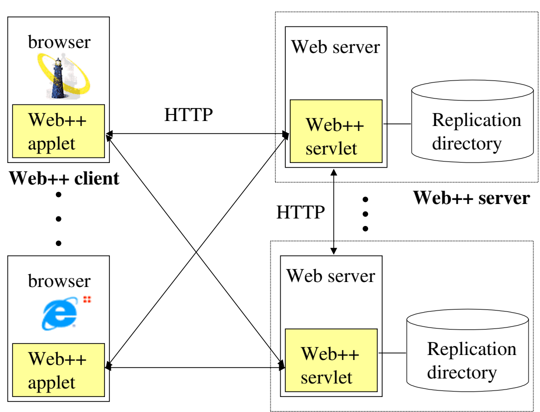

Figure 1: **Web++ architecture.**

Each server maintains a *replication directory* that maps each logical URL into a set of physical URLs that identify the locations of replicas of the resource. For example, the replication directory may contain an entry

```
/misc/file.txt :
http://server1.com/files/miscFile.txt
http://server2.com/misc/file.txt
```

that indicates that `/misc/file.txt` is a logical URL of a resource replicated on two servers, `server1.com` and `server2.com`. To reduce the size of the replication directory, the logical URL suffixes may use a wild card to specify replication locations for a set of resources.

Web++ servers are capable of creating, destroying and updating resource replicas. After creation or destruction of a replica each server automatically updates its local copy of the replication directory and propagates the update to other servers. The servers guarantee eventual consistency of their replication directories in the presence of concurrent updates propagated from different servers. Web++ servers also pre-process each HTML resource sent to a client by embedding physical URLs corresponding to each logical URL that occurs in the resource. The servers also embed a reference to the Web++ client applet within each HTML resource. Finally, Web++ servers also keep track of the load associated with their resources. The load statistics can be used by user-supplied policies that determine when, where and which resource should be replicated or when a replica should be destroyed.

3 Web++ Client Design

In this section we describe our Web++ client design and discuss the major design decisions. Web++ client binds logical URLs received in every HTML resource to one of the physical URLs corresponding to the replicas of the resources referenced by the logical URL. By binding a logical URL to a physical URL, the client selects the "closest" replica of a resource and fails over to another replica if the closest replica is not available. The current Web++ client implementation consists of approximately 18 KB of Java bytecode.

3.1 Location of Web++ client

The conversion from a logical URL into a physical URL can be done at several points on the path between a client and a server:

- Server or server-side proxy
- Client-side proxy
- Browser

A single server has only incomplete information about the network topology and cannot easily predict the kind of end-to-end performance a client would receive from other servers. Similarly, it is difficult for a server to predict whether the client would be able to reach any of the remaining servers. Therefore, the binding of logical URL to a physical URL should be done close to the client. This can be achieved by embedding the binding algorithm in a client-side proxy [4]. We see, however, several problems with the proxy approach. First, it requires making some changes to the existing proxies. Second, the proxy approach is a one-size-fits-all solution. For the same proxy, it is difficult for different content providers to use different algorithms for server selection. Third, the client-side proxy is inflexible for upgrades since it requires large numbers of users to install new versions or patches. Finally, adding a proxy on the path between a browser and a server leads to performance degradation. To quantify the performance degradation, we re-executed the client trace collected in a public lab at Northwestern University in a period of three days [35] and sent each request either directly to the Internet or via an Apache 1.2.6 proxy. To quantify only the proxy-access overhead, the proxy did no caching. The results in Figure 2 show that a commodity proxy may increase the response time by as much as 28% (223.2 ms). Similar results were obtained in [28]. We conjecture that the performance degradation is partly due to the store-and-forward implementation of Apache, i.e., the first byte of the response is not forwarded to the browser until the last byte of the body has been received by the proxy.

connectivity	average response time (ms)
direct	794.6
Apache	1017.8

Figure 2: **Proxy access overhead.**

Ideally, the client (or a client-size proxy) should dynamically download the code that performs the binding. The code can be downloaded together with the data. Such a solution does not require the end-user to install any software or to upgrade the browser. Different data providers may use different algorithms to perform the binding. Finally, upgrades can be quickly distributed among a majority of users. We are aware of two technologies satisfying the above criteria: Java applets and ActiveX controls. Since Java applets are supported by both Netscape Navigator and Microsoft Explorer browsers, we opted for an applet based implementation of Web++ client.

We also measured the overhead of executing an applet within a browser. Both Microsoft Internet Explorer 4.x and Netscape Navigator 4.x incur a relatively high overhead (3 s) to initialize the Java Virtual Machine. However, such initialization is done only once per browser session and could be done asynchronously (unfortunately, both browsers do the initialization synchronously upon parsing the first reference to an applet). We found that the execution of an applet method that implements the binding of a logical URL to a physical URL took on average 15 ms on Netscape Navigator 4.002 and 26 ms on Microsoft Internet Explorer 4.0[2]. In both cases the extra overhead is an order of magnitude smaller than the overhead incurred by using an Apache proxy and less than 4% of the average response time measured in the trace.

We observe that the Web++ applet does not have to be downloaded with every resource. In particular, the applet can be cached by the browser as any other resource. The default browser behavior is that the applet's timestamp is compared to that on the source server (using HTTP conditional `GET`) only once during each browser session.

3.2 Logical to Physical URL Binding

A given Web++ client applet must first find a list of physical URLs that correspond to replicas of every logical URL found in each HTML resource. The list can be found in several ways:

- The client queries a name server to get the list of physical URLs corresponding to a given logical URL. The name service can be either independent of the Web servers or some of the Web servers can also act as name servers. A scheme based on independent name servers similar to the DNS service was proposed for binding of Uniform Resource Names (URNs) to IP addresses [39].

- The server looks up all lists of physical URLs corresponding to every logical URL that occurs in a requested HTML resource. The list is piggybacked on the response sent to the client.

A drawback of the first scheme is that the client may have to incur an additional network round trip to bind a logical URL to a physical URL. The network round trip can be saved by caching the binding information on the client, but such a solution leads to several problems on its own (the interplay of dynamic replication and cache consistency being the most prominent one). Since one of the goals of document replication is to improve the response time perceived by clients, we rejected this option.

The second scheme does not lead to any extra overhead for the client to bind a logical URL. Moreover, since the majority of URL requests can be predicted from the hyperlinks embedded in the HTML text, it makes sense to optimize the binding scheme for this case. The drawback of this scheme is that it is not clear how to resolve logical URLs which are directly supplied by the end-user via e.g. `File` $\rightarrow$ `Open` browser menus.

3.3 Transfer of Control to an Applet

Every event that leads to sending an HTTP request (such as clicking on a hyperlink or parsing a reference to an embedded image) needs to be intercepted by the Web++ applet in order to bind the logical URL to one of the physical URLs embedded in the resource. We found two possible solutions:

- Let the browser itself render every resource along with the necessary graphical controls (e.g. "Back" button). Since the applet itself renders all graphical elements, it is simple to intercept all of the important events.

- Add JavaScript event handlers into the HTML source. The event handler transfers control to the Web++ applet when the events occur.

The first solution leads to a duplication of the browser's functionality within the Web++ applet. We rejected this solution because, in general, duplication of code is a poor software engineering practice. In this specific case, it

[2] Both browsers executed on a PC with 300 MHz Pentium II processor and 64 MB of main memory running Windows NT Workstation 4.0.

would be difficult to keep the applet rendering capabilities in sync with the latest version of HTML implemented by browsers. We therefore adopted the second approach that does not lead to any functionality duplication. However, due to the limitation of the `java.applet` API, not all important events can be intercepted by the applet. We discuss these cases below.

The HTML source modification is performed by the Web++ server. The server expands each reference to a logical URL in the resource (which typically follows <HREF> or <SRC> HTML tags) with an invocation of a JavaScript event handler. The event handler updates the value of the hyperlink reference when the hyperlink is clicked upon. For example, the hyperlink

```
<A HREF="/misc/file.txt">
```

is replaced by the server with

```
<A HREF="/misc/file.txt"
onClick="this.ref =
document.Webpp.getUrl(
http://server1.com/files/miscFile.txt,
http://server2.com/misc/file.txt)">
```

References to embedded resources (following the <SRC> HTML tag) are expanded in a similar manner. On browsers that do not support JavaScript, the `onClick` event handler is ignored and the supplied URL is used instead. Therefore, it is beneficial to select the logical URL to correspond to one of the physical URLs, e.g. the physical URL of a primary copy.

The parameters passed to the applet method directly correspond to an entry in the replication directory of the server. The above method of including the list of physical URLs directly into the HTML source may lead to an increase of size of the resource that must be transmitted to the client. However, it is possible to put all the binding information into a new HTTP header, which is then read by the client applet. Standard compression techniques can be applied on the content of the header to reduce the size of the transmitted data. The compression may be particularly effective if many of the resources are replicated on the same set of servers leading to a high redundancy in the header content. We are currently in the process of implementing the above optimization and evaluating its impact on the performance.

3.4 Batch Resource Transmission

Having a client that can be downloaded directly into the browser creates many additional optimization opportunities of the HTTP protocol. For example, based on the response time perceived by a client, the server may compress not only the content of the header containing the URL binding information, but the entire resource. The resource is decompressed by the receiving client applet. Similarly, since the server substitutes all references to embedded resources (following the <SRC> HTML tag), it can transmit to the client not only the requested resource, but also all resources embedded in it in a single response (the embedded resources are typically located on the same server). The client must be able to exclude from transmission the resources that are already cached in its local cache.

Such a "batch transmission" leads to considerable savings since most commercial Web pages contain 20 to 40 embedded images. Following standard HTTP, the browser parses the containing HTML resource and sends separate GET request for each of the embedded resources. Most browsers reduce the total retrieval time by sending 4 to 5 requests in parallel and reusing the TCP connections [34]. However, even with such optimizations, downloading a typical Web page leads to at least 4 to 5 GET request rounds. The batch transmission method reduces the entire process into a single round with a large response.

Batch transmission is implemented in our Web++ prototype and we are in the process of evaluating its impact on the response time perceived by browser users as well as the number of IP packets transmitted. Our preliminary results indicate that for clients connected to Internet over a fast T3 line, batch loading can reduce the response time by additional 40% to 52% (depending on the number and size of the embedded resources and the distance between the client and server). We also found the saving is much smaller for clients connected over a 56 kbps modem and a phone line (between 13% and 16%), because such clients are mostly limited by the phone line bandwidth, and not by the communication latency.

3.5 Limitations of Java Applets

Our implementation of the Web++ applet revealed also several limitations of the `java.applet` API:

- Applets cannot stream data directly into the browser.
- Applets cannot subscribe to events detected by the browser that are triggered outside of the applet area. For example, applets cannot detect that a user followed a bookmark. Similarly, applets cannot detect that a user typed in a URL that should be followed.

Our implementation of Web++ client applet circumvents the first limitation by writing the received resource into a local file and passing its URL to the browser. Such a mechanism allows us to implement a local browser cache that matches resources based on their logical URL as opposed to physical URL matching used in most browsers. The second limitation could be addressed (although inefficiently) by re-implementing the necessary graphical controls (i.e. bookmark button) directly within the browser area.

Both of the limitations are eliminated in the ActiveX "Pluggable Protocol" interface that is supported by Microsoft Internet Explorer 4.x browser. We plan to explore a Web++ client implementation based on this technology as well as to investigate implementation of a similar interface within the publicly available Netscape Navigator source code.

4 Replica Selection Algorithms

The performance improvement achieved by using a replicated Web service, such as Web++, critically depends on the design of an algorithm that selects one of the replicas of the requested resource. The topic has been recently a subject of intensive study in the context of Internet services [11, 22, 23, 38, 17, 35, 27, 19]. Each of the replica selection algorithms can be described by the goals that should be achieved by replica selection, the metrics that are used for replica selection and finally, the mechanisms used for measuring the metrics. The replica selection algorithms may aim at maximizing network throughput [22, 19], reducing load on "expensive" links or reducing the response time perceived by the user [11, 38, 17, 35, 27]. Most replica selection algorithms aim at selection of "nearby" replicas to either reduce response time or the load on network links. The choice of a metric, which defines what are the "nearby" replicas, is crucial because it determines the effectiveness of achieving the goals of replica selection and also the overhead resulting from measurement of the metric. The metrics include response time[17, 27], latency [35], ping round-trip time[11], network bandwidth [38], number of hops [11, 22] or geographic proximity [23]. Since most of the above metrics are dynamic, replica selection algorithms typically rely on estimating the current value of the metric using samples collected in the past. The selected metric can be measured either actively by polling the servers holding the replicas [19] or passively by collecting information about previously sent requests [38] or a combination of both [17, 35].

4.1 The Extended Refresh Algorithm

The replica selection algorithm used in Web++ is an extension of the *Refresh* algorithm studied in [35]. The Web++ implementation of the Refresh algorithm extends the original algorithm in a number of ways:

- We extend the basic replica selection algorithm with support for fail-over.
- We reduce the size of the state maintained by the algorithm (i.e. the latency table described below) by using recursive formulas.
- We generalize the metric used for replica selection by using *percentiles*.

In addition, we also performed experiments in order to study the accuracy and stability of the estimates maintained by the algorithm. We first describe the basic features of the extended Refresh algorithm and justify their selection.

We chose to minimize the response time perceived by the end-user because this is the metric perceived by the end-user. Consequently, the HTTP request response time would be an ideal metric for selection of a "nearby" server. However, the response time depends also on resource size, which is unknown at the time of a request submission. Therefore, the HTTP request response time needs to be estimated using some other metric. We chose the HTTP request latency, i.e., the time to receive the first byte of the request, because we found that it is well correlated with the HTTP request response time as shown in Figure 3. The results in Figure 3 are based on client-side proxy traces collected in the computer lab of Northwestern University and further described in [35].

metric	correlation
#hops	0.16
ping RTT	0.51
HTTP latency	0.76

Figure 3: **Correlation with HTTP request response time.**

We chose a combination of active and passive measurement of HTTP request latency. Namely, most of the time clients passively reuse the statistics they collected from previously sent requests. However, periodically, clients actively poll some of the servers that have not been used for a long time. Each Web++ client applet collects statistics about the latencies observed for each server and keeps them in a *latency table*, which is persistently stored on a local disk. To increase the sampling frequency perceived by any individual client, the latency table is shared by multiple clients. In particular, the latency table is stored in a shared file system and is accessible to all clients using the file system[3]. We have implemented a rudimentary concurrency control mechanism to provide access to the shared latency table. Namely, the table is locked when clients synchronize their memory based copy with the disk based shared latency table. The concurrency control guarantees internal consistency of the table, but does not prevent lost updates. We believe that such a permissive concurrency control is adequate given that the latency table content is interpreted only as a statistical hint. The importance of sharing statistical data for clients using passive measurements has been pointed out in [38].

[3] If a shared file system is not available, each client uses its local version of latency table.

The estimate of the latency average, which is kept in the latency table, is used to predict the response time of a new request sent to a server. However, should two servers have similar average latencies, the latency variance should be used to break the tie, because it estimates the quality of service provided by a given server. There are several ways to combine the average and variance into a single metric. We chose a *percentile* because unlike e.g. statistical hypothesis testing it always provides an ordering among the alternatives.
An S-percentile is recursively estimated as

$$\textit{S-percentile} = avg_{new} + \frac{c_S \cdot \sqrt{var_{new}}}{\sqrt{n}} \qquad (1)$$

where S is the parameter that determines the percentile (such as 30, 50 or 85), avg_{new} is the current estimate of average, var_{new} is the current estimate of variance, c_S is an S-percentile of normal distribution (which is a constant) and n is the number of samples used for calculation of average and variance.
The average avg_{new} is estimated using a recursive formula

$$avg_{new} = (1-r) \cdot avg_{old} + r \cdot sample \qquad (2)$$

where avg_{new} and avg_{old} are new and old estimates of average, $sample$ is the current value of latency and r is a fine-tuning parameter. Similarly, the variance is estimated using [31]

$$var_{new} = (1-r) \cdot var_{old} + r \cdot (sample - avg_{new})^2 \qquad (3)$$

where var_{new} and var_{old} are new and old estimates of variance.
The number of samples that affect the estimates in (2) and (3) continuously grows. Consequently, the importance of variance in (1) would decrease in time. However, the samples in (2) and (3) are exponentially weighted, so only a small fixed number of most recent samples affects the current estimates. Namely, the recursive formula for average (2) can be expanded as

$$avg_{new} = \sum_{k=1}^{N} r \cdot (1-r)^{N-k} sample_k + (1-r)^N sample_0 \qquad (4)$$

where N is the total number of all samples and $sample_0$ is an initial estimate of the average. It is straightforward to derive from (4) that only the m most recent samples contribute to $100 \cdot p\%$ of the total weight where

$$m \geq \frac{\ln(1-p)}{\ln(1-r)} - 1 \qquad (5)$$

Our extended Refresh algorithm selects the server with the minimum S-percentile of latency. Unfortunately, a straightforward implementation of a replica selection algorithm that selects resource replicas solely based on latencies of requests previously sent to the server holding the selected replica leads to a form of starvation. In particular, the replica selection is based on progressively more and more stale information about the replicas on servers that are not selected for serving requests. In fact, it has been shown that in many cases a random decision is better than a decision based on too old information [33]. There are several possibilities for implementing a mechanism for "refreshing" the latency information for the servers that have not been contacted in a long time. One possibility is to make the selection probabilistic, where the probability that a replica is selected is inversely proportional to HTTP request latency estimate for its server. An advantage of such a mechanism is that it does not generate any extra requests. However, the probabilistic selection leads also to performance degradation as shown in [35] because some requests are satisfied from servers that are known to be sub-optimal. We, therefore, chose a different approach where the client applet refreshes its latency information for each server with the most recent sample that is older than *time-to-live* (TTL) minutes. The refreshment is done by sending an *asynchronous* `HEAD` request to the server. Therefore, the latency estimate refreshment does not impact the response time perceived by the user. On the other hand, the asynchronous samples lead to an extra network traffic. However, the volume of such traffic can be explicitly controlled by setting the parameter TTL.
Upon sending a request to a server, the client applet sets a timeout. If the timeout expires, the applet considers the original server as failed and selects the resource on the best server among the remaining servers that replicate the resource. The timeout should reflect the response time of the server. For example, a server located overseas should have a higher timeout than a server located within the same WAN. We chose to set the timeout to a T-percentile of request latency in order to reuse the statistical information collected for other purposes. T is a system parameter and typically should be set relatively high (e.g. 99) in order to base the timeout on a pessimistic estimate.
After the timeout for a request sent to a server expires, the applet marks the server entry in the latency table as "failed". For every "failed" server, the applet keeps polling the server by *asynchronously* sending a HEAD request for a randomly chosen resource every F seconds until a response is received. Initially, F is set to a default value that can be for example the mean time-to-repair (MTTR) of Internet servers measured in [29]. Subsequently, the value of F is doubled each time an asynchronous probe is sent to the server. After receiving a response, the value of F is reset to its default value. The default value of F is a system parameter. The pseudocode of the replica selection algorithm can be found in Figure 4.

4.2 Experimental Evaluation

We compared the efficiency of the original Refresh algorithm with several other algorithms used for HTTP re-

```
Input: d - requested resource
       R - available servers replicating resource d
       L - latency table
Output: s - server selected to satisfy request on d
while (R nonempty) do
  if (all servers in R have expired entries in L) then
    s := randomly selected server from R;
  else
    s := server from R with minimal S-percentile of latency;
  fi
  send a request to server s;
  timeout := T-percentile of latency for s;
  if (timeout expires) then
    mark entry of server s in L as "failed";
    remove server s from R;
    send asynchronous request to s after F seconds until s responds
      and double F each time a request is sent;
    if (response received) then
      mark entry of server s in L as "available";
      include server s in R if no response received;
      reset F to its default value;
    fi
  fi
  if (response received) then
    update estimates of latency average and variance in L for server s;
  fi
  if (any server in R has expired entry in L) then
    s' := server with the oldest expired entry in L;
    send asynchronous request to s';
    depending on response either update L or mark as "failed";
  fi
od
```

Figure 4: **Pseudo-code of replica selection algorithm.**

source replica selection in [35]. We simulated replicated resources by measuring the latencies of HTTP requests sent for resources residing on 5 or 50 most popular servers in a client trace we collected at Northwestern University. In summary, we found that the Refresh algorithm improved the HTTP request latency compared with the other algorithms described in the literature on average by 55%. The Refresh algorithm improved latency on average by 69% compared with a system using only a single server (i.e. no resource replication). More details on the experimental evaluation can be found in [35].

Rather than repeating the experiments from [35], we concentrate here on the accuracy of the estimates used by the above described extension of the Refresh algorithm. The experiments also reveal surprising characteristics of the behavior of HTTP request latency[4]. In the first experiment, we compare the accuracy of HTTP request latency prediction based on an average calculated using the recursive formula (2). To collect the performance data for the comparison, we measured the HTTP request latencies of the fifty most popular servers outside of Northwestern University campus that were referenced in client traces from [35]. Each server was polled with a 1 minute period[5] from a single client at Northwestern University for a period of approximately three days. All together, we collected 228,194 samples of HTTP request latencies. At each step of the experiment, we estimated the next latency sample using the recursive formula (2). The estimate depends on a factor r that determines the weight given to the most recent sample. Figure 5 shows the mean of relative prediction error for various values of r. First, the results show that the HTTP request latency can be predicted relatively accurately from its past samples. For example, even when the sampling interval is increased from 1 minute to 10 and 100 minutes, the mean of relative prediction error is relatively low as shown in Figure 5. Second, the experiment also shows that the smaller the weight given to older samples, the better the accuracy of prediction. Such a behavior can be partly explained by existence of "peaks" on the HTTP request latency curve. The larger the value of r, the faster can the average estimate "forget" the value of the "peak". However, even after filtering out the peaks (all values 5 or 3 times the magnitude of average), we still observed qualitatively similar behavior to that shown in Figure 5. Consequently, we *conjecture* that the "memoryless" behavior is an intrinsic property of the distribution of HTTP request latency (and response time). Finally, we also verified that the accuracy of HTTP request latency prediction based on the recursive formula (5) is as good as the accuracy of prediction based on sliding window used in [35] that lead to a higher storage space overhead.

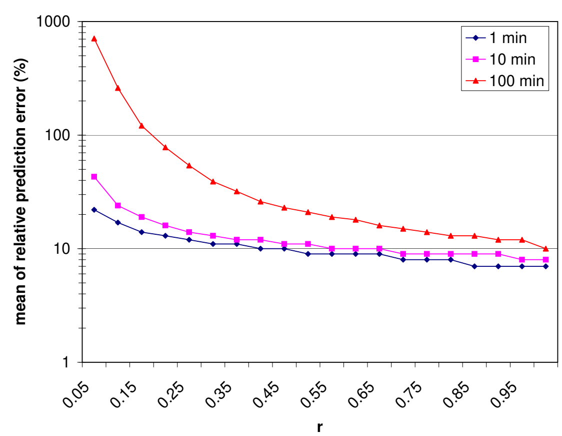

Figure 5: **Latency prediction based on recursive formula.**

The feasibility of our approach for latency estimate refreshment depends on the stability of HTTP request latency. If the latency is unstable, then a small value of TTL must be selected to keep the estimates reasonably close to their current values. Consequently, a large num-

[4] In all experiments we measured also HTTP response time and found its behavior fairly close to that of HTTP request latency.

[5] We did not use a higher polling rate as it could be interpreted as a denial-of-service attack.

ber of extra requests is sent only to keep the latency table up-to-date. Therefore, we used the experimental data described above to evaluate the stability of HTTP request latency and gain an insight to selection of the TTL parameter. For each HTTP request latency sample s_0, we define a (p, q)-*stable period* as the maximal number of samples immediately following s_0 such that at least $p\%$ of samples are within a relative error of $q\%$ of the value of s_0 (this property must hold also for all prefixes of the interval). The stability of a HTTP request latency series can be characterized by the mean length of (p, q)-stable periods over all the samples. Figure 6 shows the means of length of (p, q)-stable periods for various settings of parameters p and q. The results indicate that the HTTP request latency is relative stable. For example, the mean length of the $(90, 10)$-stable period is 41 minutes and the mean length of the $(90, 30)$-stable period is 483 minutes.

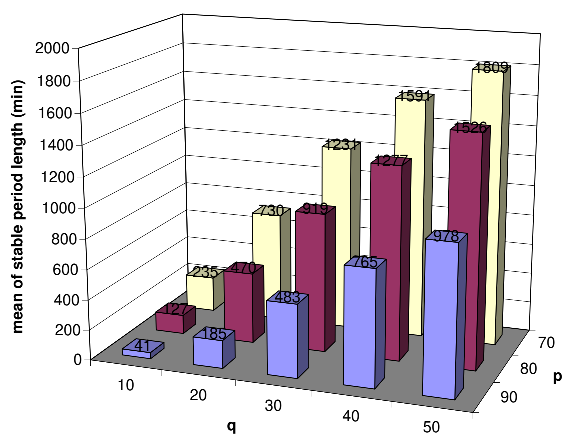

Figure 6: **Latency stability.**

5 Web++ Server Design

The Web++ server is responsible for

- Pre-processing of resources sent to the client.
- Creation and destruction of resource replicas.
- Maintaining consistency of the Replication Directory and replicated resources.
- Tracking the server load.

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. The Web++ servlet can be configured as either *server* or *proxy*. In the server configuration, the requested resources are satisfied locally either from disk or by invoking another servlet or CGI script. In the proxy configuration, each request is forwarded as a HTTP request to another server. The server configuration can be used if the server to be extended supports servlet API. If the existing server does not support servlet API, it can be still extended with a server-side proxy server that supports the servlet API (such as the W3C Jigsaw server which is freely available).

The viability of the Web++ design depends on the overhead of Web++ servlet invocation. In Figure 7 we compare the average service time of direct access to static resources and access via Web++ servlet. The servlet access includes also the overhead of resource pre-processing. The client and server executed on the same machine[6] to keep the impact of network overhead minimal. For the purpose of the experiment, we disabled caching of pre-processed resources described in Section 5.1. We found that the servlet-based access to resources leads to a 13.6% service time increase on average, and 5% and 17.6% increase in the best and worst cases. On average, the increase in service time is 3.9 ms, which is more than two orders of magnitude smaller that the response time for access to resources over the Internet (see Figures 11 and 13). We therefore conclude that even with no caching the Web++ servlet overhead is minimal.

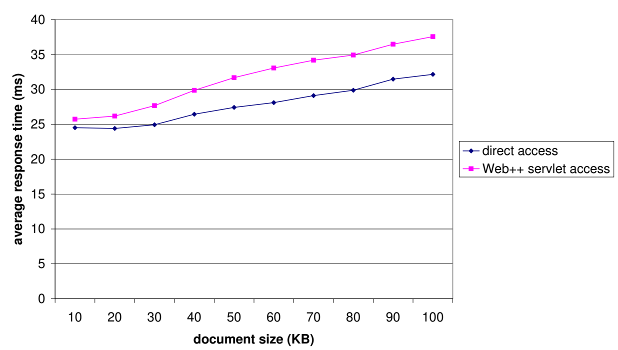

Figure 7: **Web++ servlet overhead.**

5.1 Resource Pre-processing

After receiving an HTTP `GET` request, the servlet first obtains the resource corresponding to the requested URL. The exact method of obtaining the resource depends on whether the resource is static or dynamic and whether the servlet is configured as a proxy or a server. If the resource is a HTML text, the Web++ servlet pre-processes the resource by including a reference to the Web++ client applet and expanding references to all logical URLs with JavaScript event handlers as described in Section 3. The logical URLs are found by a regular expression match for

[6] Sun SparcStation with 128 MB of RAM running Solaris 2.5 and Java Web Server 1.1.

URL strings following the <HREF> and <SRC> HTML tags. The matched strings are then compared against the local replication directory. If a match is found, the server emits the modified text into the HTML resource, otherwise the matched URL is left unmodified.

To amortize the post-processing overhead, the Web++ servlet caches the post-processed resources in its main memory. Caching of dynamically generated resources is a complex issue studied elsewhere [25] and its full exposition exceeds the scope of this paper. In the current implementation of the Web++ servlet, we limit cache consistency enforcement to testing of the `Last-Modified` HTTP header of the cached resource and the most recent modification UNIX timestamp for static resources corresponding to local files.

Web++ servlets exchange entries of their replication directories in order to inform other servers about newly created or destroyed resource replicas [41].

5.2 Resource replica management

The Web++ servlet provides a support for resource replica creation, deletion and update. The replica management actions are carried on top of HTTP `POST`, `DELETE` and `PUT` operations with the formats shown in Figure 8.

creation:	`POST`	logical URL	*<resource>*
deletion:	`DELETE`	logical URL	
update:	`PUT`	logical URL	*<resource>*

Figure 8: **Format of replication operations.**

After receiving `POST`, `DELETE` or `PUT` requests, the servlet creates a new copy of the resource, deletes the resource or updates the resource specified by the logical URL depending on the type of operation. In addition, if the operation is either `POST` or `DELETE`, the servlet also updates its local replication directory to reflect either creation or destruction of its replica. The servlet also propagates the update to other servers using the algorithm described in [41] that guarantees eventual consistency of the replication directories. We assume that such an exchange will occur only among the servers within the same administrative domain (for scalability and security reasons). Servers in separate administrative domains can still help each other to resolve the logical URL references by exporting a name service that can be queried by servers outside of the local administrative domain. The name service can be queried by sending an HTTP `GET` request to a well known URL. The logical URL to be resolved is passed as an argument in the URL.

The Web++ servlet provides the basic operations for creation, destruction and update of replicated resource. Such operations can be used as basic building blocks for algorithms that decide if a new replica of a resource should be created, on which server it should be created or how the replicas should be kept consistent in the presence of updates. Web++ provides a framework within which such algorithms can be implemented. In particular, each of the servlet handlers for `POST`, `DELETE` and `PUT` operations can invoke user-supplied methods (termed *pre-filters* and *post-filters*) either before or after the handler is executed. Any of the operations mentioned above can be implemented as a pre or post filter. Algorithms that dynamically decide whether the system should be expanded with an additional server have been described in [9, 42]. Algorithms that dynamically determine near optimal placement of replicated resources within a network have been studied in [5, 43]. Finally, algorithms for replica consistency maintenance have been described in [24, 40, 8]. However, in order to apply them to the Web these algorithms need to be extended with new performance metrics as well as take into account the hierarchical structure of the Internet. Study of such algorithms exceeds the scope of this paper.

6 Performance Analysis

In Section 1 we identified two main reasons for data replication on the Web: better reliability and better performance. The reliability improvement is obvious: If a single server has a mean time to failure $MTTF_1$ and mean time to repair $MTTR_1$, then a system with n fully replicated servers has a mean time to failure given by

$$MTTF_n \approx \frac{MTTF_1^n}{n \cdot MTTR_1^{n-1}}$$

assuming that the server failures are statistically independent [21]. Clearly, the mean time to failure improves with the number of replica servers. In order to ascertain the performance gains of resource replication, we conducted a live experiment with the Web++ system.

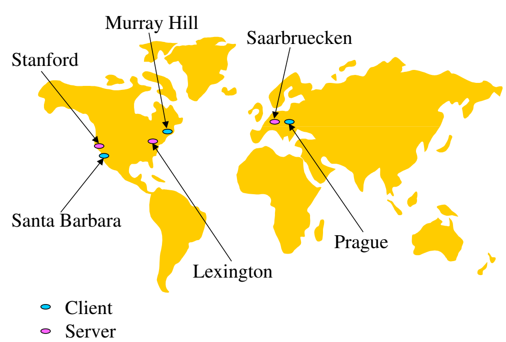

Figure 9: **Experimental configuration.**

6.1 Experimental Setup

The experimental configuration consists of three geographically distributed clients and servers. The servers were located at Stanford University (Palo Alto, US west coast), University of Kentucky (Lexington, US east coast) and University of Saarbruecken (Saarbruecken, Germany). On each site we installed a Web++ server consisting of Sun's Java Web Server 1.1 extended with the Web++ servlet. The clients were located at University of California at Santa Barbara (Santa Barbara, US west coast), Bell Labs (Murray Hill, US east coast) and Charles University (Prague, Czech Republic). For the purpose of the experiment, we converted the Web++ client applet into an application and ran it under the JDK 1.0 Java interpreter [7]. The parameter settings of our experimental configuration are shown in Figure 9.

parameter	value (unit)
r	0.95
TTL	41 (min)
default F	28.8 (hours)
S	55
T	99.9

Figure 10: **Parameter settings for Web++ system.**

The workload on each client consisted of 500 `GET` requests for resources of a fixed size. We considered 0.5KB, 5KB, 50KB and 500KB files fully replicated on all three servers. The advantage of such a workload is that it allows us to study the benefits of replication in isolation for each resource size. It is also relatively straightforward to estimate the performance gains for a given workload mix (e.g. SPECweb96 [2]). Each client generated a new request only after receiving a response to a previously sent request. We executed the entire workload both during peak office hours (noon EST) and during evening hours (6pm EST). In each experiment we report the mean of response time measured across all requests sent by all clients.

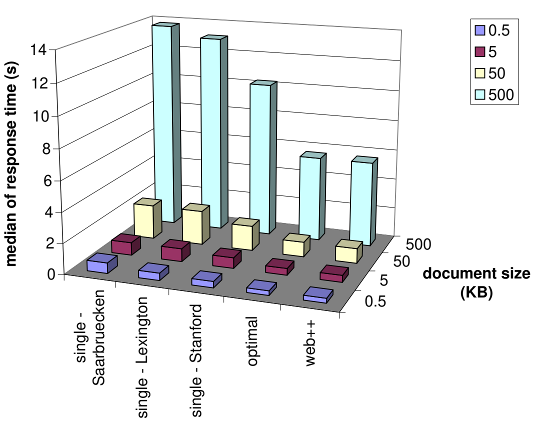

Figure 11: **Absolute response time - noon.**

We compared the performance of our Web++ system with a system that uses only a *single* server (i.e. no resource replication) and an *optimal* system that sends three requests in parallel and waits for the first response. The parameters of our Web++ system (shown in Figure 10) were set based on the sensitivity study conducted in Section 4. In particular we set parameter r to 0.95 since values close to 1 lead to best prediction accuracy as shown in Figure 5. Time-to-live (TTL) was set to 41 minutes that corresponds to the average length of a (90,10)-stable periods as shown in Figure 6. The default value of F was set to 28.8 hours that corresponds to the mean time-to-repair for Internet servers as measured in [29]. We set the percentile S to 55 to keep the impact of the variance on server selection minimal (the purpose of variance is to only break ties for servers with similar average latency). Finally, we set the percentile T to 99.9 to minimize the number of false timeouts [8].

In contrast to the experiments in [35], in the experiments reported here we tested a complete system (Web++ clients and servers). Since we had a control over the server side, we were able to compare the HTTP request response times for resources of different pre-determined sizes. Finally, we also used three geographically distributed clients as opposed to a single client in [35]. On the other hand, all resources were replicated only on three servers (as opposed to five and fifty in [35]). This limitation was imposed on us by the number of accounts we could obtain for the purpose of the experiment.

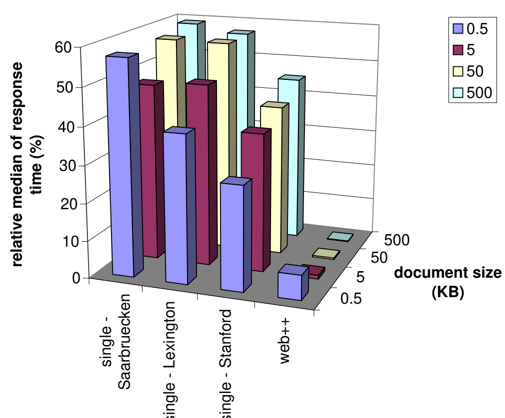

Figure 12: **Relative response time - noon.**

6.2 Experimental Results

We found that Web++ improves the response time during the peak hours on the average by 47.8%, at least by 27.8% and at most by 59.1% when compared to the single any single server system. At the same time, it degrades the response time relative to the optimal system on average by 2.2%, at least by 0.2% and at most by 7.1%. Not

[7] Some of the clients ran on platforms that did not support JDK1.1 at the time of experiment.

[8] In most cases the calculated timeout was larger than the timeout of the underlying `java.net.URLConnection` implementation

surprisingly, we found that the performance benefits of Web++ are weaker during the evening hours. In particular, we found that Web++ improves the response time on the average by 25.5%, at most by 58.9% and in the worst case it may degrade performance by 1.4%. We also found that Web++ degrades the response time with respect to the optimal system on average by 25.5%, at least by 7.8% and at most by 31%. Throughout all the experiments we found that Web++ did not send more than 6 extra requests to refresh its latency table (compared with three times as many requests sent by the optimal system!).

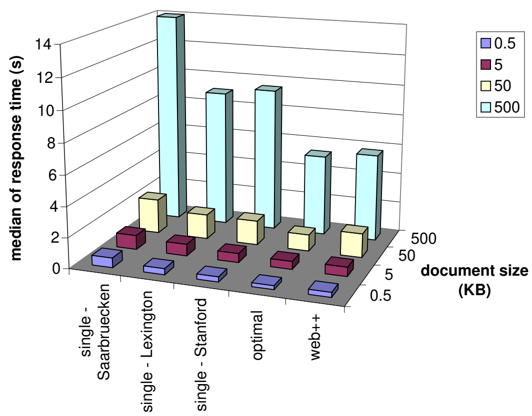

Figure 13: **Absolute response time - evening.**

The experimental results can be found in Figures 11 - 14. Figures 11 and 13 show the median of the response time during office and evening hours. Figures 12 and 14 show the relative median of the response time with respect to the median response time of the optimal system. Compared with the experimental results reported in [35] (an average 69% improvement in HTTP request *latency*), the results reported here (an average 37% improvement in HTTP request *response time*) indicate a weaker improvement in performance. We believe that the difference is due to a smaller number of replicas for each resource in the experiments reported here (3 compared with 5 and 50 in [35]). The bigger the number of replicas the higher the probability that a client finds a replica "close" to it.

7 Related Work

The use of replication to improve system performance and reliability is not new. For example, process groups have been successfully incorporated into the design of some transaction monitors [21]. The performance benefits of Web server replication were first observed in [7, 23]. The authors also pointed out that resource replication may eliminate the consistency problems introduced by proxy server caching.

The architecture of the Web++ client is closely related to Smart Clients [44]. In fact, the Web++ client is a specific instance of a Smart Client. While Yoshikawa et. al.

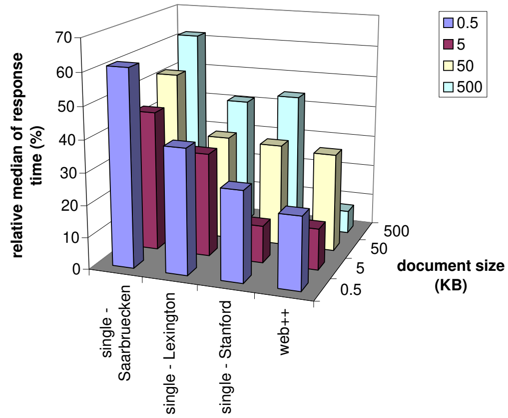

Figure 14: **Relative response time - evening.**

describe smart clients implementing FTP, TELNET and chat services, we concentrate on the HTTP service. We provide a detailed description how the client applet can be integrated with the browser environment. In addition, we describe a specific algorithm for selection of replicated HTTP servers and provide its detailed performance analysis. Finally, we describe the design and implementation of the server end.

Cisco DistributedDirector is a product that provides DNS-level replication [14]. DistributedDirector requires full server replication, because the redirection to a specific server is done at the network level. As argued in the Introduction, this may be impractical for several reasons. The DNS-level replication also leads to several problems with recursive DNS queries and DNS entry caching on the client. DistributedDirector relies on a modified DNS server that queries server-side agent to resolve a given hostname to an IP address of a server which is closest to the querying DNS client[9]. DistributedDirector supports several metrics including various modification of routing distance (#hops), random selection and round-trip-time (RTT). However, it is not clear from [14] how the RTT delay is measured and how it is used to select a server.

The Caching goes Replication (CgR) is a prototype of replicated web service [4]. A fundamental difference between the designs of Web++ and CgR is that CgR relies on a client-side proxy to intercept all client requests and redirect them to one of the replicated servers. The client-side proxy keeps track of all the servers, however no algorithms are given to maintain the client state in presence of dynamic addition or removal of servers from the system. Our work does not assume full server replication and provides a detailed analysis of resource replica selection algorithm.

Proxy server caching is similar to server replication in that it also aims at minimizing the response time by placing resources "nearby" the client [20, 30, 3, 10, 28, 36, 15, 37,

[9] The DNS client may be in a completely different location than the Web client if a recursive DNS query is used

13]. The fundamental difference between proxy caches and replicated Web servers is that the replicated servers know about each other. Consequently, the servers can enforce any type of resource consistency unlike the proxy caches, which must rely on the expiration-based consistency supported by the HTTP protocol. Secondly, since the replicated servers are known to content providers, they provide an opportunity for replication of an active content. Finally, the efficiency of replicated servers does not depend on access locality, which is typically low for Web clients (most client trace studies show hit rates below 50% [3, 10, 15, 28, 36]).

Several algorithms for replicated resource selection have been studied in [11, 22, 23, 38, 17, 35, 27, 19]. A detailed discussion of the subject can be found in Section 4.

8 Conclusions

Web++ is a system that aims at improving the response time and the reliability of Web service. The improvement is achieved by geographic replication of Web resources. Clients reduce their response time by satisfying requests from "nearby" servers and improve their reliability by failing over to one of the remaining servers if the "closest" server is not available. Unlike the replication currently deployed by most Web sites, the Web++ replication is completely transparent to the browser user.

As the major achievement of our work, we demonstrate that it is possible to provide user-transparent Web resource replication without any modification on the client-side and using the existing Web infrastructure. Consequently, such a system for Web resource replication can be deployed relatively quickly and on a large scale. We implemented a Web++ applet that runs within Microsoft Explorer 4.x and Netscape Navigator 4.x browsers. The Web++ servlet runs within Sun Java Web Server. We currently explore an implementation of Web++ client based on Microsoft ActiveX technology.

We demonstrated the efficiency of the entire system on a live Internet experiment on six geographically distributed clients and servers. The experimental results indicate that Web++ improves the response time on average by 47.8% during peak hours and 25.5% during night hours, when compared to the performance of a single server system. At the same time, Web++ generates only a small extra message overhead that did not exceed 2% in our experiments.

Web++ provides a framework for implementing various algorithms that decide how many replicas should be created, on which servers the replicas should be placed and how the replicas should be kept consistent. We believe that all of these issues are extremely important and are a subject of our future work.

9 Acknowledgments

We would like to thank Reinhard Klemm for numerous extremely helpful discussions on the subject, sharing his performance results with us and proving us with a code of WebCompanion. Our understanding of the subject was also helped by discussions with Yair Bartal. Finally, our thanks belong to Amr El Abbadi, Divy Agrawal, Tomáš Doležal, Hector Garcia-Molina, Jim Griffioen, Jaroslav Pokorný, Markus Sinnwell and Gerhard Weikum who helped us with opening accounts on their systems, installing necessary software and keeping the systems running. Without their help, the study of Web++ performance would have been impossible.

References

[1] Internet weather report (IWR). available at `http://www.mids.org`.

[2] The workload for the SPECweb96 benchmark. available at `http://ftp.specbench.org/osg/web96/workload.html`, 1998.

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

[4] M. Baentsch, G. Molter, and P. Sturm. Introducing application-level replication and naming into today's web. *Computer Networks and ISDN Systems*, 28, 1996.

[5] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proceeding of the 24th Annual ACM Symposium on Theory and Computing*, 1992.

[6] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). IETF Network Working Group, RFC 1738, 1994.

[7] A. Bestavros. Demand-based reource allocation to reduce traffic and balance load in distributed information systems. In *Proceeding of the 7th IEEE Symposium on Parallel and Distributed Processing*, 1995.

[8] Y. Breitbart and H. F. Korth. Replication and consistency: Being lazy helps sometimes. In *Proceeding of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1997.

[9] Y. Breitbart, R. Vingralek, and G. Weikum. Load control in scalable distributed file structures. *Distributed and Parallel Databases*, 4(4), 1996.

[10] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[11] M. Crovella and R. Carter. Dynamic server selection in the internet. In *Proceeding of IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, 1995.

[12] O. Damani, P. Chung, Y. Huang, C. Kintala, and Y. Wang. ONE-IP: techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29, 1997.

[13] P. Danzig and K. Swartz. Transparent, scalable, fail-safe web caching. Technical Report TR-3033, Network Appliance, 1998.

[14] K. Delgadillo. Cisco DistributedDirector. available at http://www.cisco.com/warp/public/751/distdir/dd_wp.html, 1998.

[15] 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.

[16] R. Farrell. Review: Distributing the web load. *Network World*, 1997.

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

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol - HTTP/1.1. IETF Network Working Group, RFC 2068, 1997.

[19] P. Francis. A call for an internet-wide host proximity service (HOPS). available at `http://www.ingrid.org/hops/wp.html`.

[20] S. Glassman. A caching relay for the world wide web. *Computer Networks and ISDN Systems*, 27, 1994.

[21] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[22] J. Guyton and M. Schwartz. Locating nearby copies of replicated internet servers. In *Proceeding of ACM SIGCOMM'95*, 1995.

[23] J. Gwertzman and M. Seltzer. The case for geographical push cashing. In *Proceeding of the 5th Workshop on Hot ZTopic in Operating Systems*, 1995.

[24] A. Helal, A. Heddaya, and B. Bhargava. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1986.

[25] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[26] E. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27, 1994.

[27] R. Klemm. WebCompanion: A friendly client-side web prefetching agent. *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[28] 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.

[29] D. Long, A. Muir, and R. Golding. A longitudinal survey of internet host reliability. In *Proceeding of the 14th Symposium on Reliable Distributed Systems*, 1995.

[30] A. Luotonen and K. Altis. World-wide web proxies. *Computer Networks and ISDN Systems*, 27, 1994.

[31] J. MacGregor and T. Harris. The exponentially weighted moving variance. *Journal of Quality Technology*, 25(2), 1993.

[32] S. Manley and M. Seltzer. Web facts and fantasy. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1998.

[33] M. Mitzenmacher. How useful is old information? Technical Report TR-1998-002, Systems Research Center, Digital Equipment Corporation, 1998.

[34] H. Frystyk Nielsen, J. Gettys, A. Baird-Smith, Eric Prud'hommeaux, H. Wium Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceeding of ACM SIGCOMM'97*, 1997.

[35] 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`.

[36] P. Scheuermann, J. Shim, and R. Vingralek. A case for delay-conscious caching of web documents. *Computer Networks and ISDN Systems*, 29, 1997.

[37] P. Scheuermann, J. Shim, and R. Vingralek. An unified algorithm for cache replacement and consistency in web proxy servers. In *Proceeding of the Workshop on Data Bases and Web*, 1998.

[38] S. Seshan, M. Stemm, and R. Katz. SPAND: shared passive network performance discovery. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[39] K. Sollins. Architectural principles for uniform resource name resolution. IETF Network Working Group, RFC 2276, 1995.

[40] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceeding of the ACM SIGOPS Symposium on Principles of Operating Systems*, 1995.

[41] R. Vingralek, Y. Breitbart, M. Sayal, and P. Scheuermann. Architecture, design and analysis of web++. Technical report, CPDC-TR-9902-001, Northwestern University, Department of Electrical and Computer Engineering, 1999. available at `http://www.ece.nwu.edu/cpdc/HTML/techreports.html`.

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

[43] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2), 1997.

[44] 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.