Web++ Architecture, Design and Performance

Radek Vingralek[*]
Yuri Breitbart[#]

Mehmet Sayal[+‡]
Peter Scheuermann[+‡]

[*] STAR Lab
InterTrust Technologies
4750 Patrick Henry Drive
Santa Clara, CA 95054
rvingral@intertrust.com

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

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

[‡]The work of these authors was supported in part by NSF grant CCR-9902023 and by a grant from Usenix Association

Abstract

We describe the design of a system for fast and reliable HTTP service which we call Web++. Web++ achieves high reliability by dynamically replicating Web data among multiple Web servers. Web++ selects the available server that is expected to provide the fastest response time. Furthermore, Web++ guarantees data delivery given that at least one server containing the requested data is available. After detecting a server failure, Web++ client requests are satisfied transparently to the user by another server. Furthermore, the Web++ architecture is flexible enough for implementing additional performance optimizations. We describe implementation of one such optimization, batch resource transmission, whereby all resources embedded in an HTML page that are not cached by the client are sent to the client in a single response. 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. In particular, Web++ clients are dynamically downloaded to Web browsers as signed Java applets. We implemented a Web++ prototype; performance experiments indicate that the Web++ system with 3 servers improves the response time perceived by clients on average by 36.6%, and in many cases by as much as 59%, when compared with the current Web performance. In addition, we show that batch resource transmission can improve the response time on average by 39% for clients with fast network connections and 21% for the clients with 56 Kb modem connections.

Keywords and Phrases

Web resource replication, smart clients, batch resource transmission, replica selection.

1 Introduction

1.1 Motivation

The success of the Web has proven the value of sharing different types of data in an autonomous manner. In the past five years the number of Web users, servers, and total Internet traffic have been growing exponentially. The scale of Web usage is overwhelming the capacity of the Internet infrastructure, resulting in the poor performance and low reliability of Web service. Multi-second response time for downloading relatively small documents is not unusual [Sayal *et al.* 1998]. Furthermore, recent studies [Long *et al.* 1995] indicate that 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, for example, electronic commerce or 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 the application (*non-transparent caches*) or the network (*transparent caches*) [Danzig and Swartz 1998] protocol layers. Caching improves the response time of subsequent requests which 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 [Farrell 1997]. 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.

HTTP redirect. Each server may redirect a client to another server [Fielding *et al.* 1997]. It is the client's responsibility to follow the redirection. Most browsers transparently support such redirection to the user.

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, routing distance, or TCP/IP probe response time [Delgadillo 1998][1)]

Each of the proposed solutions, however, improves either the 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 for example,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 load among wide-area replicated Web servers since all requests must be routed via the dispatcher. Moreover, the dispatcher creates a single point of failure. While HTTP Redirect improves load balance and potentially request response time, it does not increase reliability because the redirecting server is still a single point of failure. However, even the response time improvement is unclear, since HTTP redirect adds an extra message exchange between the client and servers. DNS aliasing is capable of improving both request response time and service reliability. However, the hostname is bound to an IP address by a DNS server that typically is co-located with one of the servers. DNS policies for redirection requiring a zero or low TTL value [Cardellini *et al.* 1999] are difficult to implement; one reason is the fact that many intermediate name servers will reject low TTL values. DNS protocols based on adaptive TTL values are more flexible, but they also do not consider the client-to-server distance in making scheduling decisions [Colajanni *et al.* 1999]. Finally, DNS aliasing requires that the replication is performed on all the servers constituting the web site. In many cases this is difficult or impractical; consider a CGI script accessing a legacy database system that does not support replication.

For performance reasons it is important that the data be replicated close to the clients. In fact, both proxy server caching (which is by far the most common performance optimization on the Web) and DNS aliasing aim at this goal. However, in the case of proxy caching, the clients can select only among the servers that lie on the network path between the client and a ”primary” server (transparent caches) or manually configured

[1)] The Cisco’s Distributed Director also supports HTTP-based redirection.

set of servers (non-transparent caching). While a DNS server can redirect each client to an arbitrary server, the server choice is problematic since the DNS server cannot directly measure the end-to-end performance between the client and all servers. Although the DNS server can approximate this information with a varying overhead, maintaining such a state for all clients is difficult to scale.

On the other hand, it is more efficient to select the server on the client, since clients directly can measure the end-to-end performance with no extra overhead, and this approach is more scalable, since there are fewer servers than clients. It is important for reliability purpose that the server selection component does not become a single point of failure (as is the case with server cluster dispatchers, and potentially[2)] with DNS servers). Distributing the server selection component among all clients also avoids creation of reliability bottlenecks. Finally, any practical architecture that implements data replication must be evolutionary; in particular it must not require any modifications of the client software or the communication protocol.

A complementary approach to improving Web performance, besides using replication, is to reduce the number or size of messages exchanged between client and server. Examples of such an approach are persistent connections [Padmanabhan and Mogul 1994, Fielding *et al.* 1997] and various forms of message compression. The architecture that we present in this paper supports as a by-product a similar optimization, which we term *batch resource transmission*. The batch resource transmission mechanism allows a server to send to each client not only the requested HTML page, but also all resources embedded in it (such as images) that are not cached by the client.

1.2 Our Contribution

In this paper we describe a system, Web++, that dynamically replicates Web resources on servers distributed on a wide-area network. Deployment of Web++ does not require any changes in pre-existing server and client HTTP software. Web++ clients are downloaded as cryptographically signed applets to commercially available browsers. Users are not required to install any plug-ins or client-side proxies. 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

[2)] DNS servers are required to have hot spares for availability.

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 as follows:

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: Clients' requests are directed by Web++ to the server that is expected to provide the fastest response time among all other available servers containing the requested resource. Additional performance gains can be realized by reducing the number of messages exchanged between clients and servers.

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 under-utilized servers to help sustain load peaks.

1.3 Paper Organization

The rest of the paper is organized as follows. Section 2 describes Web++ general architecture. Section 3 provides the details of Web++ client architecture while Section 4 provides a detailed description of Web++ server design. Section 5 discusses our replica selection techniques. Section 6 describes our performance experiments. Section 7 concludes the paper.

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

[Fielding *et al.* 1997]. Users submit their requests to Web++ client, which is a *Smart Client* [Yoshikawa *et al.* 1997], 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++ client applet must be signed so that it can execute outside of the browser's security "sandbox." The Web++ client sends user requests to a Web++ server, which is a standard HTTP server extended either with a servlet or a server-side proxy. The Web++ server returns the requested resource which can be generated either statically or dynamically.

A *resource* is the unit of replication in Web++ is. HTTP 1.1 defines resource as an entity referenced by a URL [Fielding *et al.* 1997]. Typically, a resource corresponds to a single file. A Web++ resource may contain 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 [Berners-Lee *et al.* 1994]. A *logical URL* is similar to a Uniform Resource Name (URN) [Sollins 1995] in that it uniquely identifies a single resource independently of its location. Unlike the URN specification, however, there are no restrictions on the syntax of logical URLs. In fact, a physical URL of resource replica can also be considered as a logical URL[3)]. 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++ client applet that executes within the user's browser.

After receiving a resource, the Web++ client applet intercepts any events that are triggered when either a browser is parsing a user request or a user is 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++ server which performs HTML source modification. We observe there that HTML source modification has been previously used to simulate a virtual server cluster dispatcher [Baker and Moon 1999]. However, our system performs a more complex modification that allows replica selection to be performed on the client rather than on the server.

Using the resource replica selection algorithm, the client selects the physical URL of the resource located

[3)] In Section 3.2 we explain why it may be useful to use one of the physical URLs as a logical URL.

at a server that is expected to provide the best response time for the client. If, after sending a request, the client does not receive a response, the client fails over to the next best server. This process continues until the entire list of physical URLs is exhausted or the resource is successfully retrieved.

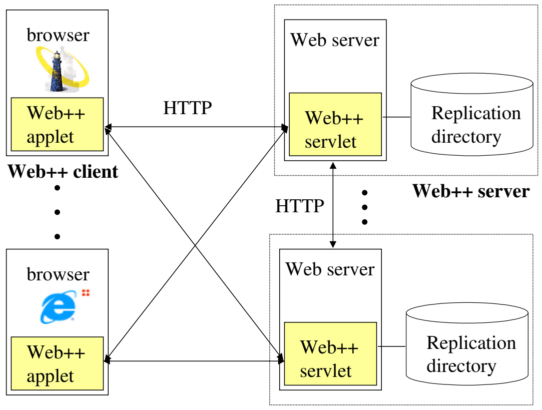

Figure 1: Web++ architecture.

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

```
/today/news.html :
    http://server1.com/files/newsFile.html
    http://server2.com/daily/today.html
```

that indicates that `/today/news.html` 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 can create, destroy and update 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 at different servers. This is achieved by using version numbers maintained with replication directories. 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 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 major design decisions. The 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 so that the client can select the ”closest” replica to be retrieved. The current Web++ client applet implementation consists of approximately 18 KB of Java bytecode.

3.1 Logical to Physical URL Binding

The conversion from a logical URL into a physical URL is done in the Web++ system at the browser side, i.e., by an applet executing within a browser. We discuss in [Vingralek *et al.* 1999b] alternative implementations based on server-side proxies or client-side proxies and explain why a browser-side conversion is preferable.

Browser functionality is extended by dynamically downloading the code that performs the binding. We are aware of two technologies that can perform this binding transparently to the browser: Java applets and ActiveX controls. Since Java applets are supported by both Netscape Navigator and Microsoft Explorer browsers, we have opted for an applet based implementation of Web++ clients. In [Vingralek *et al.* 1999b] we also report on experiments that we performed to measure the overhead of executing an applet within a browser. Specifically, we found that the execution of an applet method that implements the binding of a logical URL to a physical URL on the average took 15 ms on Netscape Navigator 4.002 and 26 ms on Microsoft Internet Explorer 4.0[4)].

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

In order to perform the logical to physical URL binding, a given client finds a list of physical URLs that correspond to replicas of every logical URL found in each HTML resource. The list is generated by the server using its replication directory and piggybacked on responses sent to the client.

3.2 Transfer of Control to Client Applet

In order to bind the logical URL to one of the physical URLs embedded in the resource, 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++ client applet. There are at least two possible solutions:

- Let the Web++ client applet itself render every resource along with the necessary graphical controls. 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++ client applet when the events occur.

The first solution leads to a duplication of the browser's functionality within the Web++ client applet; therefore, we adopted the second solution.

The HTML source modification is performed by the Web++ server servlet. 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="/today/news.html">
```

is replaced by the server with

```
<A HREF="/today/news.html"
   onClick="this.ref =
   document.Webpp.getUrl(
      http://server1.com/files/newsFile.html,
      http://server2.com/daily/today.html)">
```

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 in size of the resource that is transmitted to the client. We found that the size of a typical HTML page[5)] did not increase by more than 0.9% as a result of the above described modification. Moreover, standard compression techniques can be applied to further reduce the size of the transmitted data. Compression is particularly effective when there is a high redundancy in the replication directory entries at a given server; this occurs when many of the resources are replicated on the same set of servers.

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 which 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 which should be followed.

Our implementation of the 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 which matches resources based on their logical URL as opposed to the physical URL matching used in most browsers. Addressing the second limitation is the subject of our current work.

3.3 Batch Resource Transmission

Our architecture, which is based on smart clients that can be downloaded on demand into a browser, provides a flexibility that exceeds the needs of transparent resource replication. For example, the downloaded

[5)] CNN and Microscape homepages described in Section 6.3.

client may perform additional steps that are not currently supported by the HTTP protocol, such as custom HTTP header and request body compression, custom payment schemes for e-commerce, etc. One important such function that we implemented in Web++ is batch resource transmission. Batch resource transmission is similar to prefetching, in that it reduces HTTP overhead by anticipating future resource requests. But while prefetching tries to bring resources that *may* be referenced by the user to the client's cache, batch resource transmission sends to the client in a single response the requested HTML resource, and the embedded resources that are *required* for parsing of the HTML resource, which have not been already cached by the client. Batch resource transmission has objectives that are similar to the persistent connections [Padmanabhan and Mogul 1994] and request pipelining additions proposed in the HTTP 1.1 protocol [Fielding *et al.* 1997] in that its goal is to minimize the number of network packets exchanged between client and server in order to download an HTML page. However, unlike the other two approaches, batch resource transmission completely eliminates the need for sending more than one HTTP request per page.

Batch resource transmission leads to considerable savings since most commercial Web pages contain 20 to 40 embedded images. Following standard HTTP 1.0, the browser parses the original HTML resource requested by the client and sends separate GET requests 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 [Nielsen *et al.* 1997]. However, even with such optimizations, downloading a typical Web page leads to at least 4 to 5 GET request rounds. The batch transmission mechanism reduces the entire process to a single round with one request and one response.

We illustrate the batch resource transmission in Web++ in the following example. We denote as a *parent resource* the HTML document portion of a web page that contains references to a number of embedded resources. Let us assume two parent resources, `news1.html` and `news2.html`, are accessed by a client. Furthermore, let these parent resources contain the following embedded resources (e.g., inlined images):

`news1.html:`	contains `e1.gif`
`news2.html:`	contains `e1.gif` and `e2.gif`

Figure 2 shows the traditional client-server interaction when a Web browser requests `news1.html` and news2.html, in this order. During each GET round the server responds with at most one resource. The server responds with the requested resource only if the client's cache does not have a copy of the resource or has a stale copy of it.

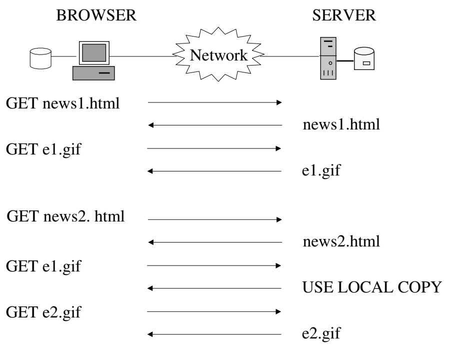

Figure 2: An example of standard client-server communication.

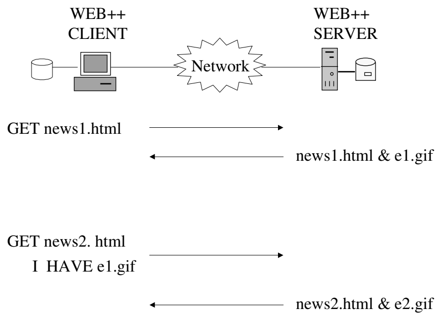

Figure 3: **An example of client-server communication with batch transmission.**

Figure 3 shows the Web++ client-server communication for the same request pattern when batching is used. Batch transmission requires only one GET round for each *parent resource*. Notice that when the client sends a request to the server, it also notifies the server about the resources embedded in the requested parent resource that already are present in the client's cache. The server responds only with those embedded resources whose most recent copies have not been cached by the client.

To avoid transmission of embedded resources that already have been cached by the client, the client must notify the server about such resources. Consequently, the client must be able to find which resources are embedded in a parent resource it is about to request. This information is inserted by the server into the hyperlink referencing the requested parent resource. Thus, in addition to the modifications discussed in Section 3.2, the server also adds a list of resources embedded in the referenced parent resource. Each embedded resource is identified by its checksum (MD5 in our current implementation). In addition, checksums also are used to validate the content of client's cache. There are several benefits to using checksums instead of full URLs: First, in most cases a checksum is shorter than the actual filename of an embedded resource (e.g., 128 bits for MD5 checksum). This reduces the network transmission costs. Second, a single resource might be available under different URLs. Checksums eliminate duplication of such resources in the client's cache and their re-transmission by the server.

4 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 a 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 a 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 can execute servlets. If the existing server does not support servlets, it still can be extended with a server-side proxy server that supports servlets (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 4 we compare the average service time of direct access to static resources with access via Web++ servlet. The servlet access also includes 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 4.1. We found that the servlet-based access to resources leads to a 13.6% service time increase on average, and 5% and 17.6% increases in the best and worst cases. On average, the increase in service time is 3.9 ms, which is more than two orders of

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

magnitude smaller that the response time for access to resources over the Internet (see Figures 7 and 9). We therefore conjecture that even with no caching the Web++ servlet overhead is minimal on a *lightly* loaded server.

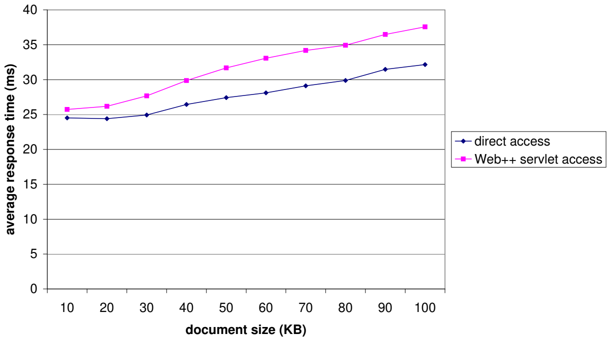

Figure 4: Web++ servlet overhead.

4.1 Resource Pre-processing

After receiving a 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 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 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 main memory. Caching of dynamically generated resources is a complex issue studied elsewhere [Iyengar and Challenger 199 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 in their replication directories to inform other servers about newly created or destroyed resource replicas [Vingralek *et al.* 1999a].

4.2 Resource replica management

Web++ servers can create, delete and update resource replicas. The replica management actions are carried on top of HTTP POST, DELETE and PUT operations using the formats shown in Figure 5.

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

Figure 5: Format of replication operations.

After receiving POST, DELETE or PUT requests, the server either creates a new copy of the resource, or deletes 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 server also updates its local replication directory to reflect either creation or destruction of its replica. The server also propagates the update to other servers using the algorithm described in [Vingralek *et al.* 1999a] that guarantees eventual consistency of the replication directories using version numbers. 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++ server provides the basic operations for creation, destruction and update of replicated resources. Such operations can be used as basic building blocks for algorithms that decide whether 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. Most of these algorithms can be implemented as suitable pre or post filters.

Algorithms that dynamically determine near optimal placement of replicated resources within a network have been studied in [Wolfson *et al.* 1997]. Finally, algorithms for replica consistency maintenance have been described in [Helal *et al.* 1986, Terry *et al.* 1995]. However, in order to apply them to the Web, these algorithms need to be extended with new performance metrics and they must take into account the hierarchical structure of the Internet. Study of such algorithms exceeds the scope of this paper.

5 Replica Selection

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 recently has been a subject of intensive study in the context of Internet services [Crovella and Carter 1995, Fei *et al.* 1998, Guyton and Schwartz 1995, Gwertzman and Seltzer 1995, Klemm 1999, Sayal *et al.* 1998, Seshan *et al.* 1997]. Each of the replica selection algorithms can be described by its goals and metrics that are used for replica selection and by the mechanisms used for measuring the metrics. The replica selection algorithms may aim at maximizing network throughput [Guyton and Schwartz 1995], reducing load on "expensive" links or reducing the response time experienced by the user [Crovella and Carter 1995, Fei *et al.* 1998, Klemm 1999, Sayal *et al.* 1998, Seshan *et al.* 1997]. Most replica selection algorithms aim at selection of "nearby" replicas either to reduce response time or the load on network links. The choice of the metric used to choose the "nearby" replicas is crucial because it determines the effectiveness of replica selection and also the overhead resulting from measurement of the metric. The metrics include response time [Fei *et al.* 1998, Klemm 1999], latency [Sayal *et al.* 1998], ping round-trip time [Crovella and Carter 1995], network bandwidth [Seshan *et al.* 1997], number of hops [Crovella and Carter 1995, Guyton and Schwartz 1995] and geographic proximity [Gwertzman and Seltzer 1995]. Since most of these 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 actively by polling the servers holding the replicas, passively by collecting information about previously sent requests [Seshan *et al.* 1997], or a combination of both [Fei *et al.* 1998, Sayal *et al.* 1998].

Web++ client applets select resource replicas based on the algorithm described below, which we named *Refresh* algorithm. In [Sayal *et al.* 1998] we experimentally compare performance of the Refresh algorithm against other design alternatives.

In the design of the Refresh algorithm, we chose to minimize the HTTP request response time because this is the metric most relevant to the end-user. Consequently, the HTTP request response time would be an ideal metric for selection of a "nearby" server. However, the response time also depends 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 Section [Sayal *et al.* 1998].

The Refresh algorithm uses a combination of active and passive measurement of HTTP request latency. Most of the time clients passively reuse the statistics they collected from previously sent requests; however, periodically, clients actively poll the servers that have not been used for a long time to obtain reasonably recent statistics about the HTTP latency of such servers. 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 of the client machine.

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 and select servers with more predictable performance. 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 a total 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 the latency average, var_{new} is the current estimate of the latency 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 the average and the variance.

The average avg_{new} is estimated using a recursive formula

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

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. The optimal setting of parameter r is experimentally analyzed in [Vingralek *et al.* 1999b]. Similarly, the variance is estimated using [MacGregor and Harris 1993]

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

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 over 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

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

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

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

The Refresh algorithm selects the server with the minimal 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 [Mitzenmacher 1998]. 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 the 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 also leads to performance degradation as shown in [Sayal *et al.* 1998] because some requests are satisfied from servers that are known to deliver sub-optimal performance. Therefore, we chose a different approach where the client 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. The latency estimate refreshment does not impact the response time perceived by the user. On the other hand, the asynchronous samples lead to extra network traffic. However, the volume of such traffic can be explicitly controlled by setting the parameter TTL. The feasibility of such an approach critically depends on the stability of HTTP request latency. Unstable HTTP latency would lead to a small value of TTL to keep the latency table statistics current and thus a large number of extra requests generated only to keep the latency table up-to-date. The experiments in [Vingralek *et al.* 1999b] show that the HTTP request latency is indeed relatively stable.

Upon sending a request to a server, the client sets a timeout. If the timeout expires, the client 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 client marks the server entry in the latency table as "failed." For every "failed" server, the client 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 [Long *et al.* 1995]. 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.

6 Experimental Evaluation

In Section 1 we identified two main reasons for data replication on the Web: better reliability and better performance. The reliability improvement is straightforward: if a single server has a mean time to failure $MTTF_1$ and a 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 [Gray and Reuter 1993]. Clearly, the mean time to failure improves with the number of replica servers. Since the performance gains of resource replication are more difficult to quantify analytically, we demonstrate the performance potential of resource replication in a series of live experiments with an implementation of the Web++ system (Section 6.2). In addition, we analyzed the performance benefits of batch resource transmission described in Section 3.3.

6.1 Experimental Setup

Unless stated otherwise, all experiments are based on a configuration that consists of three geographically distributed clients and servers. The servers were located at Stanford University (US west coast), the University of Kentucky (US east coast) and the University of Saarbruecken (Germany). On each site we installed a Web++ server consisting of Sun's Java Web Server 1.1 extended with the Web++ servlet. In all experiments we used the default settings for the Java Web Server, except that we set the number of requests satisfied over a single network connection to 100. The clients were located at the University of California at Santa Barbara (US west coast), Lucent Bell Labs (US east coast) and Charles University (Czech Republic). In order automatically to run experiments with large numbers of requests, we converted the Web++ client applet into an application and ran it under the JDK 1.0 Java interpreter [7)].

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

Unless noted otherwise, the Web++ system parameters (shown in Figure 6) were set based on the sensitivity study described in [Vingralek *et al.* 1999b]. In particular we set parameter r to 0.95 since values close to 1 lead to the best prediction accuracy as shown in [Vingralek *et al.* 1999b]. Time-to-live (TTL) was set to 41 minutes that corresponds to the average length of (90,10)-stable periods as shown in [Vingralek *et al.* 1999b]. The default value of F was set to 28.8 hours, corresponding to the mean time-to-repair for Internet servers as measured in [Long *et al.* 1995]. We set the percentile S to 55 to keep the impact of the variance on server selection minimal (the purpose of variance is only to 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 all experiments we define the response time as the time between sending the first byte of client request and receiving the last byte of server response. Unless noted otherwise, all experiments were repeated a sufficient number of times so that the 95% confidence intervals for the reported averages lie within 5% of the reported values.

6.2 System Performance Experiments

We demonstrated the performance potential of wide-area resource replication by conducting a series of live experiments using an implementation of Web++ which was configured as described in Section 6.1. We compared the performance of Web++ against two systems. The first system (*Single*) makes no use of resource replication and all clients always satisfy their request from a single, a priori fixed server. The Single system models performance of the current web clients and servers. The second system (*Optimal*) makes an aggressive use of resource replication by sending each request to all servers that replicate the resource.

The clients in each system sent requests for resources of sizes 0.5 KB, 5 KB, 50 KB and 500 KB. Each resource was fully replicated among all three servers. The resources did not contain any embedded resources. Therefore, batch resource transmission was not used and the reported performance gains are solely due to selection of "nearby" resource replicas by each client. In each experiment, all clients generated

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

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

Figure 6: Parameter settings for Web++ system.

HTTP GET requests only for a single size resource. The advantage of such a workload is that it allowed 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). Each client generated a new request only after receiving a response to a previously sent request. Therefore, all servers were relatively lightly loaded and the difference in server capacity of the Single system (1 server) and Web++ (3 servers) was insignificant. We executed the entire workload both during peak office hours (noon EST) and during evening hours (6pm EST). In each experiment we report the average of response time measured across all requests sent by all clients.

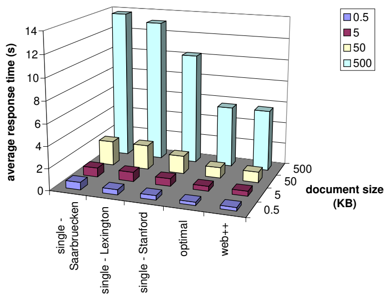

Figure 7: Absolute response time - noon.

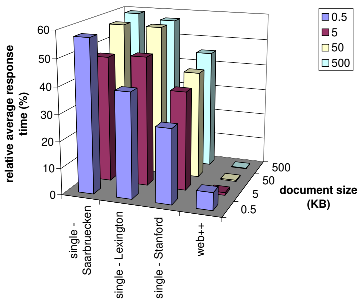

Figure 8: Relative response time - noon.

We found that Web++ improves the response time during the peak hours on average by 47.8%, at least by 27.8% and at most by 59.1% relative to the Single system using any of the three servers. 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 surprisingly, we found that the performance benefits of the Web++ system are weaker during evening hours. In particular, we found that the Web++ system improves response time on average by 25.5%, at most by 58.9% and in the worst case it may degrade response by 1.4% relative to the Single system using any of the three servers. We also found that the Web++ system degrades response time during evening hours 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.

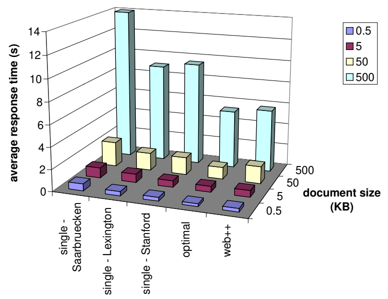

Figure 9: Absolute response time - evening.

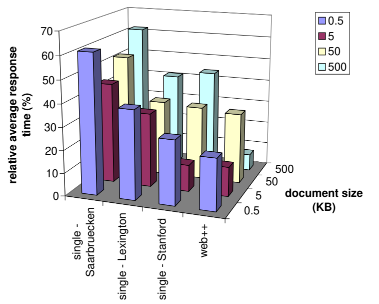

Figure 10: Relative response time - evening.

The experimental results can be found in Figures 7 - 10. Figures 7 and 9 show the average of the response time during peak and evening hours. Figures 8 and 10 show the relative average response time with respect to the average response time of the Optimal system.

6.3 Batch Resource Transmission

We conducted a series of experiments to analyze the performance benefits of batch resource transmission. We compared the performance of a Web++ client applet that used batch transmission and executed within a browser with the performance of a stand alone browser. We measured the response time and the number of network packets sent to download a typical web page.

We used two clients located at the University of Kentucky. The first client was connected to the Internet via a T1 line (*fast net* experiments). The second client was connected to Internet via a dial-up line with a 56 Kb modem (*slow net* experiments). Both clients executed the Web++ applet within either a Netscape Navigator 4.51 browser (*Navigator* experiments) or a Microsoft Internet Explorer 4.0 browser (*Explorer* experiments). Both browsers implement HTTP 1.1 persistent connections. The clients sent requests against servers located either at the University of Kentucky (*local* experiments) or the University of the Saarland (*transcontinental* experiments).

In each experiment a client submitted several requests to download a typical web page. We used a synthetic "Microscape" page created by combining home pages of Netscape and Microsoft [Nielsen *et al.* 1997] and the CNN home page of July 23, 1998. The total size of the Microscape home page (with all inlined images) was 172 KB. The size of the HTML resource itself was 43 KB. The Microscape home page contained 42 inlined GIF images with sizes ranging between 44 B and 40 KB. The sizes of 18 images ranged between 0 and 1 KB, 18 images ranged between 1 and 5 KB, 4 images ranged between 5 and 15 KB and one image had size 40 KB. The total size of the CNN home page was 95 KB. The size of the HTML resource itself was 46 KB. The CNN home page contained 21 inlined GIF images with sizes ranging between 35 B and 7 KB. The sizes of 7 images ranged between 0 and 1 KB, 10 images ranged between 1 and 5 KB and 3 images had sizes between 5 and 7 KB. The exact benefit of batch resource transmission depends on the number of resources found in the local browser cache. We measured two extreme cases: when all resources must be requested from a remote server (*cold cache* experiments) and when all resources are found in the local cache (*warm cache* experiments). In the latter case, we configured the browser to validate each resource before accessing it from the local cache so that it provided the same degree of consistency as the batch transmission algorithm.

We define the response time as the time between a user click on a hyperlink referencing the page and the completion of the download of the entire page (including all embedded resources). Consequently, the reported response times also included the browser overhead. This overhead is also experienced by the users when browsing web pages and thus should be included in the reported response times. The response times of batch resource transmission also include the delays due to downloading the Web++ applet. Since all requests had to be manually submitted via the browser interface, each experiment was repeated only five times, resulting in a 95% confidence interval for the reported averages to lie within 30% of the reported values.

We found that batch transmission improved the response time experienced by the user by 15% - 31% in cold cache experiments (as shown in Figures 11 and 12) and 87% - 93% in warm cache experiments (as shown in Figures 13 and 14). Surprisingly, the improvement is stronger in warm cache experiments because

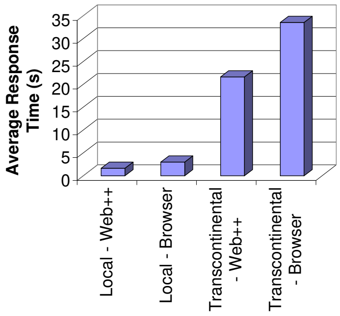

Figure 11: Response time for the fast net, cold cache experiments.

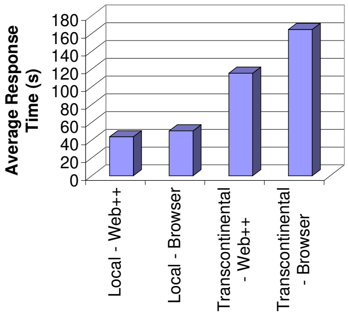

Figure 12: Response time for the slow net, cold cache experiments.

browsers, unlike batch transmission, generate an extra validation request for every cache hit irrespective of whether the resource was updated. The batch transmission response contains checksums of all embedded resources. Consequently, a validation request is sent only if the resource has been updated (which was the case in all experiments).

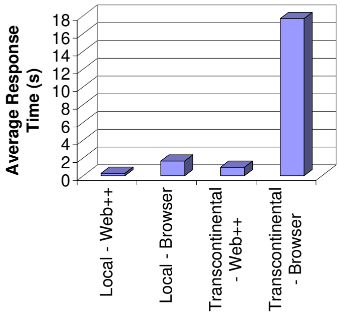

Figure 13: Response time for the fast net, warm cache experiments.

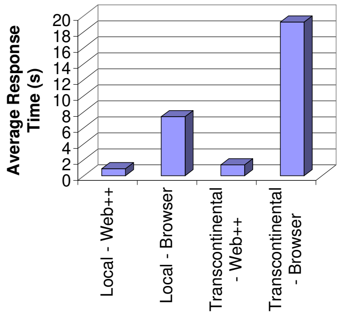

Figure 14: Response time for the slow net, warm cache experiments.

As expected, the benefits of batch transmission are stronger in fast net and transcontinental experiments. The overall response time improvement in fast net experiments was 39% compared with 21% in slow net experiments because the response time in slow net experiments is dominated by the dial-up link bandwidth, not by network latency (which batch resource transmission is designed to reduce). Similarly, the response time improvement in transcontinental experiments with cold cache was 31% compared with 15% improvement in local experiments because the local experiments also are less dominated by network latency. We did not observe significant differences between response time in Navigator and Explorer experiments. Therefore, we report only averages over both types of experiments in Figures 11, 12, 13, and 14 under "Browser".

Although the end-user response time was our primary performance metric, we also measured the reduction in the number of transmitted network packets. Overall, we found that batch resource transmission decreased the number of network packets on average by 33% in cold cache experiments and on average by 75% in warm cache experiments. Figure 15 summarizes the average number of packets transmitted by Web++ batch transmission and normal browser operation for retrieving CNN and Microscape documents with cold and warm client caches.

Cache	Document	Web++	Browser
Cold	CNN	303	411
Warm	CNN	108	395
Cold	Microscape	312	514
Warm	Microscape	110	459

Figure 15: Number of transmitted packets.

7 Conclusions

Web++ is a system that improves the response time and the reliability of Web service by wide-area 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 site mirroring 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 a Web++ client based on Microsoft ActiveX technology.

We demonstrated the efficiency of the entire system in a live Internet experiment on six geographically distributed clients and servers. The experimental results indicate that Web++ improves response time on average by 47.8% during peak hours and 25.5% during night hours, relative 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.

Taking advantage of the flexible client-server architecture of Web++, we also have added batch resource transmission capability to our prototype implementation. Our experiments showed that batch transmission by itself reduces the response time on the average by 39% for clients with fast network connections and by 21% for the clients with 56 Kb modem connections. When the requested resources are already in the client's cache, batch transmission reduces the response time even more dramatically since the bandwidth is no longer a problem when all that needs to be done is to check whether the resources in the client cache are up to date. Response time reduction was between 87% to 93% in such cases.

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.

8 Acknowledgments

We would like to thank Reinhard Klemm for numerous extremely helpful discussions on the subject, sharing his performance results with us and providing us with a code of WebCompanion. Our understanding of the subject was also helped by discussions with Yair Bartal. We thank Jim Gettys for sharing the source of the Microscape home page with us. Finally, our thanks go 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

[Baker and Moon 1999] Baker, S. and B. Moon (1999), "Distributed Cooperative Web Servers," In *Proceedings of the 8th International World Wide Web Conference*.

[Berners-Lee *et al.* 1994] Berners-Lee, T., L. Masinter, and M. McCahill (1994), "Uniform Resource Locators (URL)," IETF Network Working Group, RFC 1738.

[Cardellini *et al.* 1999] Cardellini, V., M. Colajanni, and P. S. Yu (1999), "Dynamic Load Balancing on Web-Server Systems," *IEEE Internet Computing 3*, 3, 28–39.

[Colajanni *et al.* 1999] Colajanni, M., P. S. Yu, and V. Cardellini (1999), "Dynamic Load Balancing in Geographically Distributed Heterogeneous Web Servers," In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, pp. 295–302.

[Crovella and Carter 1995] Crovella, M. and R. Carter (1995), "Dynamic Server Selection in the Internet," In *Proceeding of IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*.

[Danzig and Swartz 1998] Danzig, P. and K. Swartz (1998), "Transparent, Scalable, Fail-Safe Web Caching," Technical Report TR-3033, Network Appliance.

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

[Farrell 1997] Farrell, R. (1997), "Review: Distributing the Web load," *Network World* .

[Fei *et al.* 1998] Fei, Z., S. Bhattacharjee, E. Zegura, and M. Ammar (1998), "A Novel Server Selection Technique for Improving the Response Time of a Replicated Service," In *Proceeding of IEEE INFOCOM'98*.

[Fielding *et al.* 1997] Fielding, R., J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee (1997), "Hypertext Transfer Protocol - HTTP/1.1," IETF Network Working Group, RFC 2068.

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

[Guyton and Schwartz 1995] Guyton, J. and M. Schwartz (1995), "Locating Nearby Copies of Replicated Internet Servers," In *Proceeding of ACM SIGCOMM'95*.

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

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

[Iyengar and Challenger 1997] Iyengar, A. and J. Challenger (1997), "Improving Web Server Performance by Caching Dynamic Data," In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*.

[Klemm 1999] Klemm, R. (1999), "WebCompanion: A Friendly Client-Side Web Prefetching Agent," *IEEE Transactions on Knowledge and Data Engineering* .

[Long *et al.* 1995] Long, D., A. Muir, and R. Golding (1995), "A Longitudinal Survey of Internet Host Reliability," In *Proceeding of the 14th Symposium on Reliable Distributed Systems*.

[MacGregor and Harris 1993] MacGregor, J. and T. Harris (1993), "The Exponentially Weighted Moving Variance," *Journal of Quality Technology 25*, 2.

[Mitzenmacher 1998] Mitzenmacher, M. (1998), "How Useful Is Old Information?" Technical Report TR-1998-002, Systems Research Center, Digital Equipment Corporation.

[Nielsen *et al.* 1997] Nielsen, H. F., J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley (1997), "Network Performance Effects of HTTP/1.1, CSS1, and PNG," In *Proceeding of ACM SIGCOMM'97*.

[Padmanabhan and Mogul 1994] Padmanabhan, V. and J. Mogul (1994), "Improving HTTP Latency," In *Proceedings of the Second International WWW Conference*.

[Sayal *et al.* 1998] Sayal, M., Y. Breitbart, P. Scheuermann, and R. Vingralek (1998), "Selection Algorithms for Replicated Web Servers," In *Proceeding of the Workshop on Internet Server Performance*, available at `http://lilac.ece.nwu.edu:1024/publications/ WISP98/final/SelectWeb1.html`.

[Seshan *et al.* 1997] Seshan, S., M. Stemm, and R. Katz (1997), "SPAND: Shared Passive Network Performance Discovery," In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*.

[Sollins 1995] Sollins, K. (1995), "Architectural Principles for Uniform Resource Name Resolution," IETF Network Working Group, RFC 2276.

[Terry *et al.* 1995] Terry, D., M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser (1995), "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," In *Proceeding of the ACM SIGOPS Symposium on Principles of Operating Systems*.

[Vingralek *et al.* 1999a] Vingralek, R., Y. Breitbart, M. Sayal, and P. Scheuermann (1999a), "Architecture, Design and Analysis of Web++," Technical report, CPDC-TR-9902-001, Northwestern University, Department of Electrical and Computer Engineering, available at `http://www.ece.nwu.edu/cpdc/HTML/techreports.html`.

[Vingralek *et al.* 1999b] Vingralek, R., Y. Breitbart, M. Sayal, and P. Scheuermann (1999b), "Web++: A System for Fast and Reliable Web Service," In *Proceedings of the 1999 USENIX Annual Technical Conference*.

[Wolfson *et al.* 1997] Wolfson, O., S. Jajodia, and Y. Huang (1997), "An Adaptive Data Replication Algorithm," *ACM Transactions on Database Systems 22*, 2.

[Yoshikawa *et al.* 1997] Yoshikawa, C., B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler (1997), "Using Smart Clients to Build Scalable Services," In *Proceedings of USENIX'97*.