Report Number: WUCS-97-24

1997-01-01

Reducing Web Latencies Using Precomputed Hints

Girish P. Chandranmenon and George Varghese

Current network technology is bandwidth-rich but latency-poor; thus round-trip delays will dominate access latency for web traffic. We describe four new techniques that reduce the round-trips needed for web accesses. The techniques are based on the paradigm of preprocessing a web page to collect information about links and inline data in the page. Stored Address Binding almost always eliminates the DNS lookup (which can cost seconds) at the start of a transaction. In Informed Server Proxying, a server tells its client that it has cached pages referenced in a page the client just retrieved; this allows the client to... **Read complete abstract on page 2.**

Reducing Web Latencies Using Precomputed Hints

Girish P. Chandranmenon and George Varghese

Complete Abstract:

Current network technology is bandwidth-rich but latency-poor; thus round-trip delays will dominate access latency for web traffic. We describe four new techniques that reduce the round-trips needed for web accesses. The techniques are based on the paradigm of preprocessing a web page to collect information about links and inline data in the page. Stored Address Binding almost always eliminates the DNS lookup (which can cost seconds) at the start of a transaction. In Informed Server Proxying, a server tells its client that it has cached pages referenced in a page the client just retrieved; this allows the client to retrieve the pages from its current connection, instead of creating a new connection. In Selective Link Redirect, a server can direct its client to any server that has cached a page referenced in the current document; this allows cooperating servers to balance load. Finally, Auto Inline Download allows a server to send inline data such as images and applets, without additional client requests. We describe implementations and measurements of these techniques.

Reducing Web Latencies Using Precomputed Hints

Girish P. Chandranmenon
George Varghese

WUCS-97-24

February 25, 1997

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

Current network technology is bandwidth-rich but latency-poor; thus round-trip delays will dominate access latency for web traffic. We describe four new techniques that reduce the round-trips needed for web accesses. The techniques are based on the paradigm of preprocessing a web page to collect information about links and inline data in the page. *Stored Address Binding* almost always eliminates the DNS lookup (which can cost seconds) at the start of a transaction. In *Informed Server Proxying*, a server tells its client that it has cached pages referenced in a page the client just retrieved; this allows the client to retrieve the pages from its current connection, instead of creating a new connection. In *Selective Link Redirect*, a server can direct its client to any server that has cached a page referenced in the current document; this allows cooperating servers to balance load. Finally, *Auto Inline Download* allows a server to send inline data such as images and applets, without additional client requests. We describe implementations and measurements of these techniques.

This work was supported by National Science Foundation under grant NCR-9628218

Reducing Web Latencies Using Precomputed Hints

Girish P. Chandranmenon
girish@cs.wustl.edu
+1 314 935 4163

George Varghese
varghese@askew.wustl.edu
+1 314 935 4963

1. Introduction

The World Wide Web is arguably the most successful Internet application. Using a simple interface paradigm of following pointers to data (regardless of geographic location), the web has made the Internet accessible to the public. Information on everything from Babysitting Cooperatives to Immigration Law can be found on the web because of the huge number of content providers building web sites. As Spectrum describes it: "In 1996, everything and everybody — from your favorite soda pop to graduate students looking for jobs or dates — built a Web Site ...the Web has been expanding at a rate more commonly associated with nuclear reactions"[GS97]. Indeed, one study, due to Mathew Gray at MIT, shows 230,000 web sites in June 1996 versus 23,500 web sites in June 1995.

While the Web as a mass market is still in its infancy, users and suppliers clearly see the Web as a means of trading information, entertainment, and other purchases. If the Web is to become part of the nation's infrastructure, it is important that the Web has the performance and reliability of other utilities such as the telephone and power services. Our paper is about improving the performance of the Web by *lowering access latencies.* Other issues such as reliability and security are also critical; we do not, however, address those topics.

Latency is the time that elapses between the user typing in a request and the arrival of useful information on the user's screen. Users care about such delays and tend to complain (or give up) when faced with large retrieval times. Long delays experienced by web accesses can be classified into two categories: delays due to waiting for the start of a response (round-trip times) and the time taken to transfer the data in a response. Currently, round-trip delays can be exacerbated by slow servers and slow lines. As the Web gains popularity, faster web servers and transmission links will be deployed. However, there is no corresponding technological fix for basic speed of light limitations.

Thus we argue that as network technology moves towards a bandwidth-rich regime (e.g.., gigabit data transfers in 1 sec), the number of round-trip times spent in getting a document will be the dominant factor in the time required to retrieve a document. For example, a 1MB file transfer will require at least 60ms (typical observed round trip time, 36ms if we can send data at speed of light) + 8.4ms (transfer time at 1 Gb/s) across the USA; the round trip times are worse (about 130ms) across trans-atlantic links. Since the speed of light imposes a basic lower bound on the round-trip time, the only way to improve access latency is to reduce the number of round-trip times (RTTs) required.

step	time spent
DNS lookup	$\geq$ 1 RTT
Connection setup	1 RTT
Doc. request and response	1 RTT + (response size)/bandwidth
Image request and response	1 RTT + (image size)/bandwidth

Table 1: Typical steps in a web page retrieval and associated latency costs.

Why does a Web access require more than one round-trip time? [HC94] describes the operation of the web in detail. A Web document is identified by a *Uniform Resource Locator (URL)*, which contains a host name and the name of a file on that host. Given a URL, a browser, such as Netscape or Mosaic, queries a domain name server (DNS [Moc87]) for the address of the host, establishes a connection to the host using its address, and finally makes a request for the file identified by the URL. If the file contains images or inline data that are to be displayed with the web page, the browser sends additional requests to retrieve them. Once the file is rendered on screen, the page may contain *hyperlinks* which are references to other web pages. If the user selects a hyperlink, the browser retrieves the new page using the same process described earlier. The client and the server communicate using the Hyper Text Transfer Protocol (HTTP)[FGM+97, BLFN96] and the page is often written in Hyper Text Markup Language (HTML).[BLC95]

Clearly, the number of round-trips spent by two communicating entities is an artifact of the protocols used. Theorem 1 summarizes the steps involved in retrieving a web page. All steps require at least one RTT to complete.[1] In what follows, we describe schemes for reducing some or all of these round trips, thus improving overall access time.

The most common techniques to reduce such round trip delays are *caching* (e.g., at users and in proxies) and *prefetching* documents associated with hyperlinks. These techniques can help but are not sufficient. Caching depends on locality patterns which may not hold as users explore new links; [WAS+96] finds the locality of reference to be only 50% with high miss penalties for proxy cache misses. Also, [AW96] found that about a third of the files and bytes are accessed only once; this implies that caching at the server is tricky. Prefetching documents is problematic because a user may traverse only a small set of the links associated with a page; without knowing the user's tastes, prefetching wastes network bandwidth and dilutes caches. Persistent-HTTP [Mog95a] eliminates extra connection set up for inline data requests but does not remove the other latencies described in Theorem 1.

1.1. Contributions

In this paper, we propose four new ideas to reduce latencies due to round-trip times. All four schemes add more information — *hints* — to the web document to avoid extra round-trip latencies. These hints are added to documents which the user must navigate through anyway to reach the desired data. Thus, the performance gains obtained by exploiting these hints is independent of the typical locality assumptions needed for caching to work well.

The first three schemes compile information about the URLs of links in a web page; the fourth scheme adds information about inline data. These hints are prepended to the web page as MIME [FB96] headers and are expected to be generated by a compiler or preprocessor for the web page. A summary of our ideas, in the context of some earlier schemes, is given in Theorem 2. A more detailed comparison with earlier work is in Section 2.

[1] In this table we have not included the additional waits due to interactions between HTTP and the underlying transport protocol [Hei96].

Idea	Added information	benefit
Stored Address Binding (SAB)	IP address of the host name in the URL	Avoid DNS lookup at clients
Informed Server Proxying (ISP)	A flag for each URL indicating whether the server has cached the referenced document.	Clients can avoid connecting to the original server to retrieve the referenced document.
Selective Link Redirect (SLR)	A list of alternate server addresses for every URL	A set of cooperating servers can distribute the documents among them to balance their load.
Auto Inline Download (AID)	A list of inline images and applets needed to render the web page	Servers can send the inline data immediately following the web page without parsing the entire page.

Table 2: Four new ideas to reduce latency of web accesses. Each idea adds information to web pages. However, this information can be automatically generated by preprocessing the web page when the page is published.

Our first three schemes add information about links in a web page. We propose adding three pieces of information to a web page about each *URL* it contains: (1) the IP address of the host name in the URL, (2) a *locallyCached* flag which indicates whether the server providing the current document has cached the web page pointed to by the link and (3) a list of alternate server addresses where the document identified by the URL can be found. We briefly explain the use of this extra information.

In *Stored Address Binding*, the client uses the stored IP address to avoid a DNS lookup that potentially costs one RTT or longer. Since IP addresses change very rarely, we believe that the cached IP address will almost always be correct. If needed, the Address Bindings can be updated dynamically at intervals comparable to DNS cache expiry times (Section 4).

In *Informed Server Proxying (ISP)*, the server uses the *locallyCached* flag to inform clients that the server has cached pages corresponding to some of the links in the current page. This helps the client avoid additional connections to the original server of the document. By contrast, in a normal server proxy (or a web server accelerator) [CDN+96], the proxy takes over the web server's identity (port 80 in the TCP world). From then on, all requests to the server must go through the proxy. In ISP, the proxying server has its own identity, and can choose to set the locallyCached flag to `false`, which forces the client to go to the original server. ISP increases the scope of caching to apply to *any* site that references a document; it also allows a finer granularity of caching than standard server proxies (Section 5).

Selective Link Redirect (SLR) is a generalization of the *locallyCached* flag, expanding it to a list of possible server names or addresses. Thus a link information header contains the URL, an IP address which is the DNS translation for its host name, and a list of proxy addresses where the same document can be obtained from. This generalization can enable load balancing among multiple machines in a domain that provide web service. ISP and SLR are illustrated in Figure 1. SLR is described in detail in Section 6.

Our fourth idea, *Auto Inline Download*, allows a client to request the server to send inline data (e.g., images) in a document immediately following the web page containing such data. The client request specifies whether the server should send *all* inline images, only those which are *unique* to the page, or *no* inline data. If the web server has to send the inline data using current web pages, it has to parse the contents of the page and identify the inline data to send them. To avoid examining the entire file, we suggest adding a header in a web page, which compiles information about all inline data for the web page (Section 7).

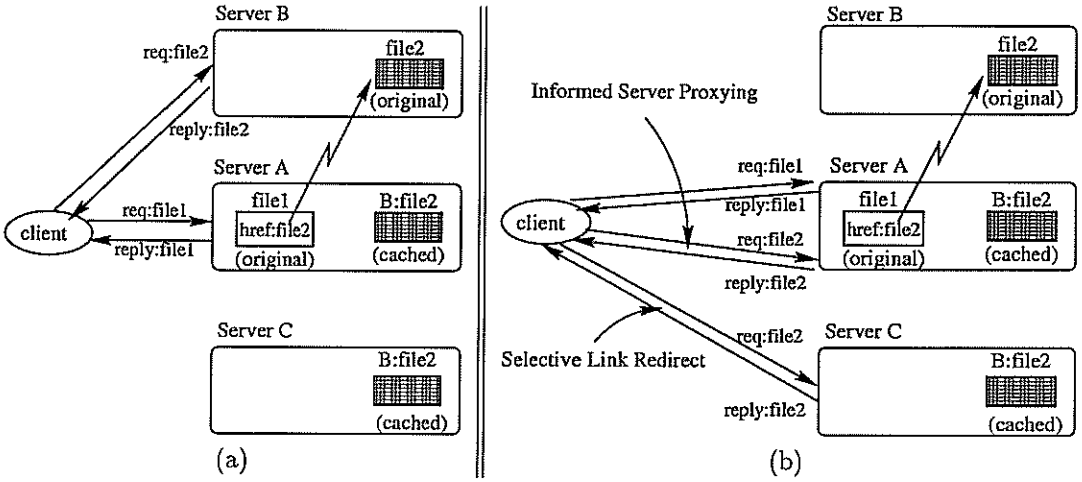

Figure 1: ISP and SLR: The left frame shows normal web access, while the right frame shows a modified access using ISP (or SLR). In ISP, server A announces to the client that it has cached file 2 locally. This information enables the client to retrieve file 2 from server A itself, thus avoiding a connection set up. In SLR, server A annotates the URL for file 2 to indicate that file 2 is available at server C as well. If server B is busy, the client can connect to server C to retrieve the document.

In all four optimizations we add hints to a web page; these hints should be produced by compiling[2] the web page before publishing the page to the outside world. In order to keep the user created version consistent with the compiled version, we include the time of last modification of the user edited page in the compiled version. When the page is accessed, the server compares the stored time in the compiled page against the actual time of last modification of the file. If they are the same, the compiled version is sent to the client, otherwise the original page is sent and recompiled to ensure consistency.

The idea of rewriting an HTML file before publishing it opens a whole new area of optimizations; we discuss some of these other optimizations in Section 9.

2. Related Work

We compare our schemes with existing optimizations in Theorem 3. Assume that a client is navigating web pages in some domain, and each web page (document) in this domain has $n-1$ inline images on the average. The rows represent optimizations and the columns count the number of round trips contributed by each major source of round trips: connections, requests, and DNS lookups. The last column lists specific disadvantages of each scheme.

The normal HTTP protocol (first row) makes n separate connections for each document (web page plus $n-1$ inline images), n separate requests per document, and one DNS lookup per server. Persistent HTTP[Mog95a] (incorporated into HTTP 1.1) reduces the number of connections to one per server (multiple documents at the same server can be accessed over the same connection). Prefetching (third row) does not reduce the number of connections, requests or DNS lookups. Prefetching can, however, mask the access latency by obtaining the required documents *before* the user wants them. However, prefetching has the disadvantage of potentially prefetching useless documents, thereby wasting network bandwidth and diluting the cache.

[2]Unlike normal compilation of a source language which produces an object code, this compilation produces an output in the same language.

	Scheme	# conn	# req.	# dns	disadvantage
1	HTTP/1.0 ([BLFN96]	1/doc	n/doc	1/server	too many connections
2	HTTP/1.1 ([FGM+97])	1/server	n/doc	1/server	too many requests
3	Prefetching	1/doc	n/doc	1/server	cache dilution
4	Client Proxies ([LA94])	1/doc	n/doc	1/server	single point bottleneck
5	Server Proxies ([CDN+96])	1/domain	n/doc	1/server	single point bottleneck
6	Stored Address Binding (SAB) (Section 4)	1/doc	n/doc	0	page modifications needed ; - consistency problems
7	Informed Server Proxying (ISP) (Section 5)	1/domain	n/doc	1/domain	page modifications needed
8	Selective Link Redirect (SLR) (Section 6)	[1..k]/domain	n/doc	1/domain	page modifications needed
9	Auto Image Download (AID) (Section 7)	1/doc	1/doc	1/server	page modifications needed
10	2,6,7,8 & 9	[1..k]/domain	1/doc	0	modified page consistency

Table 3: Summary of new ideas in the context of some earlier schemes. There are $(n-1)$ images in the document, and the load at the server is distributed to k machines.

The next two rows evaluate server and client proxies. Before we do so, it is important to understand the range of possibilities for proxies illustrated in Figure 2. In scenario 1, (case (a) in figure) the client talks directly to the server. In scenario 2, the client always talks to what is called a *client-proxy*; the proxy talks to servers when the client requests connections to these servers. Client proxy is not transparent to the client; the client must send all requests to the client proxy.

The fourth row of Theorem 3 shows that client proxies do not reduce the metrics with respect to a *single* client. However, client proxies are beneficial when many client machines in a domain request the same data and other clients can be served from the proxy cache. Client proxies, however, can become a bottleneck; they also add an extra connection when data is not in the proxy cache. Further, the effectiveness of client caching depends on good locality of reference. The miss penalty is quite high. Finally, most proxy implementations are in user space, which makes forwarding very slow.

In the third scenario in Figure 2, a client talks to a proxy on the server side, called a *Server Proxy* or a *server accelerator*. Server Proxies can be beneficial because the set of frequently accessed files at any server is small. This proxy is transparent to the clients. The proxy assumes the identity of the server, and the actual server is run at a different port address. Only misses at the proxy and requests for dynamic documents are forwarded to the actual server. It is also possible to combine client and server proxies as in the fourth scenario.

One major benefit of server proxies, not shown in the fifth row of Theorem 3, is that server proxies offload the original servers. A second benefit is that server proxies can help a client to make just one connection per domain (to the server proxy) and hopefully obtain most data from the server proxy cache. A Server Proxy can, however, become a bottleneck because *all* data for that domain must be served out of it. There is no mechanism for a server to selectively serve some (e.g., the most frequently accessed and reasonably sized documents) and have clients directly access any other data from the original servers (instead of using the proxy as an intermediary).

Our schemes are shown in rows 6 through 10 of Theorem 3. Stored Address Bindings can eliminate DNS lookups entirely at the cost of possible consistency problems. Informed Server Proxying

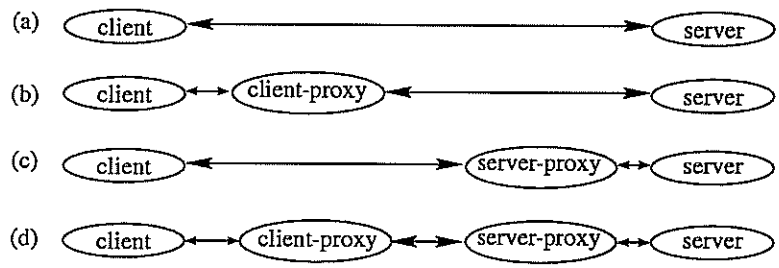

Figure 2: Four possible configurations: (a) client talks directly to the server. (b) client talks only to the client proxy and the proxy talks to all the servers. (c) all clients talk to the server proxy, and the only proxy talks to the server. (d) client talks only to cilent proxy, and only server proxy talks to the server. Proxies talk to each other.

(ISP) can, like Server Proxies, allow just one connection per domain. However, unlike Server proxies, the ISP point need not be a bottleneck because of two reasons. First, there can be multiple ISP points for each domain: any site that contains the first reference to the sites in a domain can serve as an ISP point. Second, the ISP mechanism allows the ISP point to selectively cache certain documents, and to leave the client to *directly* access the remaining documents from the original servers.

Selective Link Redirection allows a form of load balancing (though it does increase the number of connections), thus potentially improving server throughput. Auto Image Download reduces the number of requests to 1 per document by automatically downloading all inline data in a single response. Both these schemes need to be carefully compared to other proposed optimizations for the same purposes: DNS Load Balancing, Cooperating Caching Servers and Pipelining.

DNS Load Balancing [Bri95] has been proposed for use among a set of machines that together implement a server proxy for one site. The scheme uses DNS to provide multiple addresses for the same name; each time a host makes a request, DNS will return a random permutation of the addresses. The idea is to have clients connect to different machines at different times, resulting in balanced load. However, according to Mogul [Mog95b], DNS based load balancing schemes do not work well. Once the client has done one lookup, it tends to use the same address repeatedly. Thus, DNS based load balancing does not react to the actual load at the server. On the other hand, our SLR scheme can *dynamically* redirect the request to one of the cooperative servers. SLR does not depend on DNS to balance load.

Cooperating Caching Servers [MLB95] describes a protocol for load sharing among caching proxies. In this scheme, a client picks one caching server as its master proxy, and if the master does not have the document that the client requested, it multicasts a query to the proxy group. If a member of the group has the document cached, it replies to the master proxy and the master proxy informs the client about the location of the document. Then, the client can make another request to that proxy and retrieve the document. Our load balancing scheme, Selective Link Redirect, is different from [MLB95]. SLR is used by a server to redirect further requests (not the current request) to other lightly loaded servers. Thus SLR does not require additional latency in accessing the currently requested document.

Pipelining of inline Data Requests has also been proposed to reduce the latency of requesting $n-1$ separate images by requesting all of them concurrently. However, pipelining cannot be initiated until the web page itself is received at the client. If the pages are smaller than the delay-bandwidth product (as they often are) round-trip times dominate the transfer time, and the server is kept waiting for the next request. Moreover, if there are applets in the web page, the referenced applet has to arrive at the client in order to figure out which other classes it needs. This makes it impossible to pipeline such requests.

By contrast, in our Auto Inline Download scheme, the server has access to the list of files needed for the web page. Therefore, it can download all the files that are necessary for the client, including all the classes required by an applet, without waiting for further requests from the client.

Finally, Row 10 of Theorem 3 shows that all our schemes can be combined with persistent HTTP to provide the combined benefits of all schemes. Our schemes can also be combined with other optimizations such as client proxies and are, in fact, orthogonal to most other optimizations. A disadvantage of all our schemes is that they require web page modifications. However, we will argue, in the next section, that automatic compilation of web pages can redress this deficiency.

3. Compiling HTMLs

To Modify or not to Modify: We have talked about prepending hints to an HTML file. We can prepend MIME headers to a web page in two ways: first, by rewriting the file, such that the prepended header is in the same file as the web page itself; second, by keeping a shadow file for every web page. All the compiled information is kept in the shadow file. For the rest of the paper, we assume that the compiled information is in the same file as the HTML document. This is not an unreasonable assumption, since most users of the web are likely to use an editor to author their web pages. The editors can be enhanced to compile and store the file.

Maintaining Consistency: In either case, if the original file gets modified, we need to ensure consistency between the original file and its corresponding hints database. We keep the time stamp of the original file in the hints data base. Before the server sends the page to the client, it can detect inconsistent time stamps and take corrective action. A possible action is not to send hints for this transaction (so that the latency of this access is unaffected), and schedule a file recompilation. A second possible action is to to recompile the file before sending a response, which can delay the response to the current transaction.

Note that there are two kinds of hints:

Static hints: These are the hints that are not altered by the server at run-time (e.g., IP addresses of host names in URLs). This information can be kept consistent by doing either periodic compilation or on-demand compilation. In periodic compilation, the server periodically recomputes the hints in each file; in on-demand compilation the server recomputes hints only when requested.

Dynamic hints: In this case the stored hint is just a place holder and its value is recomputed at run-time (e. g., locallyCached flag is best kept as a dynamic hint since the URLs that are cached can change based on access patterns). When the server sends the document to the client, it should lookup its cache and update the dynamic hints.

4. Avoiding DNS Lookup *(Stored Address Binding)*

When a client contacts a server machine for the first time, it has to lookup the server address, since URLs provide only server names. Typically the lookup results in a DNS query; this can take longer than one round trip if the address has not been cached in any of the name servers in the DNS hierarchy. For example, DNS queries of sites outside the U.S. take times of the order of a second. Even when the address is cached in a local name server within the domain, the client might have to send a packet and receive a response, which is extra work.

To avoid DNS lookups for every URL, we suggest including the address in the HTML file itself. Our idea is to preprocess the HTML file to add the addresses of hostnames in the referenced URLs

site	time for one DNS lookup
web1.cims.co.uk	0.24
www.telegraph.co.uk	0.11
www.cnn.com	0.10
www.intellicast.com	0.15
www.nytimes.com	0.13
www.pathfinder.com	0.09
altavista.digital.com	1.52
www.yahoo.com	0.17
city.net	0.30
squid.nlanr.net	0.13
whois.ripe.net	2.63
www.apnic.net	0.32
www.citenet.net	0.14
www.internic.net	0.11
www.nic.de	0.23
www.ripe.net	0.20
www.sfgate.com	0.29

site	time for one DNS lookup
ee.lbl.gov	0.21
www.bellcore.com	0.31
www.inria.fr	1.17
www.parc.xerox.com	0.37
webrunner.neato.org	0.17
www.acm.org	0.17
www.iab.org	0.16
www.ieee.org	3.54
www.ietf.org	0.28
www.usenix.org	0.25
www.w3.org	0.07
vigyan.iisc.ernet.in	24.71
spark.iitm.ernet.in	4.03
www.noao.edu	0.34
kepler.unh.edu	0.21
superior.cs.unh.edu	0.21

Table 4: Time in seconds taken by `gethostbyname()` for various sites taken at early morning hours, by which time most DNS cache information could have expired. This is a very crude estimate of DNS performance. The important thing to note is that lookup times can be in the order of seconds even for sites within the U.S.

(these could be hyperlinks or inline data references). We prepend this information to the HTML file as MIME headers. Since the IP addresses of hosts change very rarely, we expect this information to be correct most of the time. Thus, almost always, the client will be able to avoid a DNS lookup.

When a browser wants to follow a URL to connect to a site, it can try to connect to the IP address stored with the URL. However, if it had already resolved the host name to a different IP address, it should give priority to the address in its local cache over the address supplied with the link. There are two issues to be resolved in storing address bindings.

First, the IP address of the host might change, rendering invalid the address in the file. In this case the client will get an error and can retry after doing a DNS lookup. It could also notify the server of this problem, so that the server can take appropriate corrective action. (see Section 3.) Note that DNS caches today have the same problem. We can always make our scheme have the same degree of consistency as DNS caches by updating our address bindings at intervals comparable to DNS cache timeouts.

A second possible issue arises if the old IP address is reassigned to a different machine. To solve this problem, the HTTP request should contain the name of the host the browser intended to connect.[3] The server should check this name against its own to verify that the client indeed wanted to request the document from that server. If the name does not match, it should report an error to the client and close the connection.

Our very rough estimates on DNS measurements (time taken by `gethostbyname()` to look up various hosts) are shown in Theorem 4 and these indicate that DNS lookup can take anywhere from a few milliseconds to a few seconds. These measurements were done at early morning hours, and that could be one of the reasons for large lookup times on some popular sites like altavista.digital.com.

Lookup on some sites abroad took three to four seconds. By reducing the DNS lookup we save the initial delay in accessing a web page most of the time. Moreover reducing DNS lookups avoids the additional tinygrams on the backbone.

[3] the 'Host:' field in HTTP/1.1 can be used for this purpose.

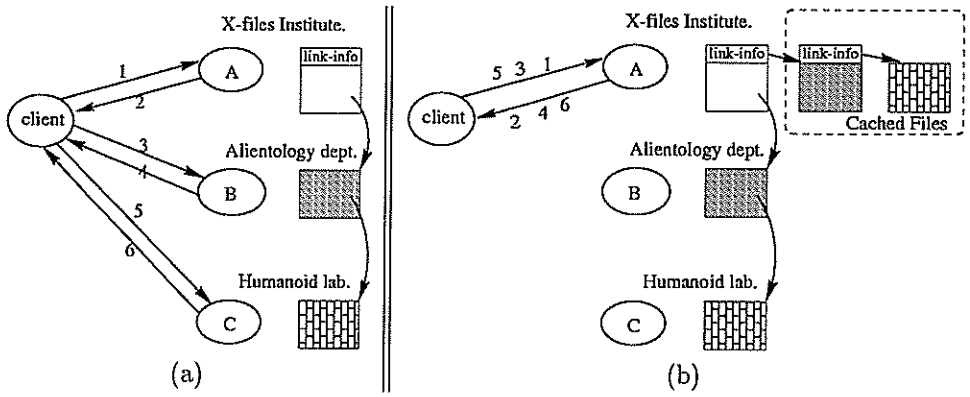

Figure 3: Informed Server Proxying: The left frame shows normal web access and the right frame shows web access using ISP. Notice how all required pages are retrieved from the entry point (X-files institute).

There are some security issues. Consider the scenario where the IP address stored in the web page is taken up by a different server now, and the server decides to lie to the clients that it is the actual server. The client has no way of figuring out that the server is lying; but the current web also has the same problem. The right solution in both cases is to authenticate the server.

5. Avoiding Multiple Connections *(Informed Server Proxying)*

In a typical browsing session, a client is likely to access a web page at at some entry point server (such as a server at a university or a company), and follow links from that page to other servers (such as that of departments within a university or projects within a company) that are within the same domain. Informed Server Proxying tries to reduce the number of connections to be set up to the domain.

Based on some policy (we describe possible choices later in this section), a server S locally caches as many pages (corresponding to URLs it references) as possible. From then on, when a client accesses a page from S, as part of the hint headers, the server annotates each URL in the file with a *locallyCached* flag. This information enables the client to retrieve a cached URL directly from S, instead of establishing a new connection to its original server.

Consider the example in Figure 3. A client is trying to locate the web page of the 'Humanoid laboratory' in 'Alientology department' in the 'X-files Institute'. Using current web protocols, the client first makes a connection to server A to get the page of the institute, connects to server B to get the page of the department, and finally connects to server C to get the lab page. In this sequence of steps, many round trips are spent in setting up connections. Instead, if server A cached the top level home pages of all departments and labs in the institute, and could tell the client that it has cached them in the link-info header, the client could request server A for those pages, without making additional requests.

The client does not, however, have to set up a connection to server B to get to a technical report from the 'Humanoid Laboratory' (as it would in a server proxy). After the user has found the document in the Humanoid Laboratory's page, the browser can connect to server C and fetch the document. The advantage of ISP is that the client does not waste time setting up connections to several servers just to navigate through them.

Benefits of ISP:

1. **Flexible Caching:** Clients can hop through web pages until they get to the "leaves" of the web

graph; at that point, the actual files (e.g., papers by a professor, or a technical report on a project) can be fetched from its original server. Such files may be too large to be cached at the reference points.

2. Client Flexibility: If the flag indicates that the current server has the document cached, the clients still have the option of retrieving the link from the current server, or ignoring the flag and retrieving the link from its original server.

3. Multiple Server Caches: No single server has to assume responsibility for a collection of servers, and any server can act as a proxy for any other server. Unlike Server Proxying (or server accelerators), ISP is not transparent to the clients.

Annotating a URL:. When a client accesses a web page, if the server has to go through it to annotate each URL, the server has to parse the entire page. The page can be preprocessed to compile a list of URLs at the start of the page, as in the case of *SAB*. For each URL, we keep a header which contains a flag indicating if the URL is locally cached. The value of the flag should be recomputed whenever the set of cached pages changes. If changes are rare (see below), the value can be written into the file and the server does not have to compute it when a client accesses the page. If the set of cached pages changes frequently, the flag should be assigned a value only when a client accesses the page. Note that the hint headers should be updated for both the local and cached pages in order to gain the full benefit of ISP.

Which Pages Should Be Cached?. The answer to this question is entirely policy dependent. If we have information about all future accesses, we can design an optimal caching policy. In the absence of that information, we have some pragmatic choices.

First, we can statically decide which pages a server is going to cache. For example, a server in a university might decide to cache the home pages of all departments. Another possible static policy is to cache every page upto k levels from any local page, where k is some parameter.

The problem with static policy is that sometimes we might end up caching a page that is never accessed. Thus we can allow the server to dynamically change the cached set of pages. A possible policy is to store the Most Frequently Followed URLs, or Most Recently Followed URLs. Detailed evaluation and analysis of these policies is left for future work. A link hit counter mentioned in Section 9 will aid such dynamic policies.

Our performance measurements for ISP are given in Section 8.

6. Load Balancing *(Selective Link Redirect)*

In Informed Server Proxying, a server informed the client about those URLs that were cached locally at the server. This information caused the client to redirect its request to the same server instead of the original server for the document. *Selective Link Redirect* extends this idea to enable a server to redirect client requests to servers other than the original server of the URL and itself. In SLR, a server annotates each URL with a list of IP addresses, all of which can supply the document. This scheme helps in balancing load among a set of co-operating servers that provide web service to a set of organizations.

Consider the example in Figure 4. Servers A, B and C are configured to have knowledge of the documents in each other's disk. The original server of x and y is C, that of z is A and that of p is B. The links shown are the actual URL references in the files. i.e., x has a URL that is `http://C/y`

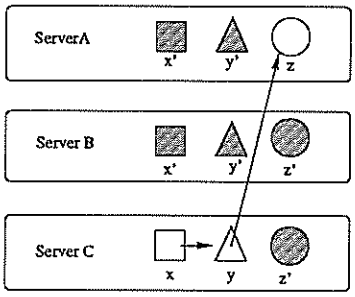

Figure 4: Load Balancing using Selective Link Redirect. Server A, B and C have copies of x, y and z and all of them know that. (The shaded files are copies.) When a client connects to C and requests for x, if it is overloaded, it can annotate the URL for y with the address A or B or both. This annotation can redirect the client to go to establish a connection to A or B and request for y. The replication and distribution of files can be driven by access profiles.

and y has the URL `http://A/z`. The shaded files are the copies, and all the servers know that the others have copies of the same files. When a request comes to Server C for file x, if it is overloaded, it sends the file x, but it annotates the URL for y to indicate that y is available at servers A and B. If the client is not getting fast enough response from C, it can connect to server A or server B and retrieve file y. Similarly, the server delivering y can annotate the URL for z if it is busy,

7. Avoiding Requests for Inline Data *(Auto Inline Download)*

Using current web protocols (HTTP/1.0, HTTP/1.1) to retrieve pages with inline images, a client has to send one request for the document and one request for each inline image. Since the client does not know the names of images in a document until it receives the first image, multiple requests result in waiting periods of more than one RTT for fetching a document with inline images. In case of applets the scenario is even worse: it is only after some applet A referenced in a page is fetched that the class loader can know that A depends on other applets B, C, and D that also need to be fetched. This process can repeat since B could depend on P and Q, and so on.

With newer network technology, the delay bandwidth product of the network pipe has been increasing, making round trip times more and more expensive. Since most web pages are small to medium size, and most requests are small units of data, a typical web transaction between a browser and a web server can consists of long periods of waiting interspersed with small periods of data transfer.

Auto Inline Download (AID) reduces these periods of wait significantly by allowing the server to send the inline images as soon as it has finished sending the page, without an explicit request from the browser. Under normal conditions this would mean that the server has to understand the contents of the web page, parse it to find the references to inline images, and send them. It is expensive to have a server parse every web page it serves. We suggest adding headers to web pages (HTML files) compiling information about all the inline images in the page.

The example in Figure 5 illustrates this idea. In the normal case, Figure 5(a), the client has to make a number of requests to get all inline images. Even if we can send all the requests for inline images at once, the overall time spent in retrieving the images may be more than the time it should take to transfer the images, due to the round trip times involved. The round trip times can be avoided altogether, if the server is able to send all the images the page needs immediately following the page, as shown in Figure 5(b).

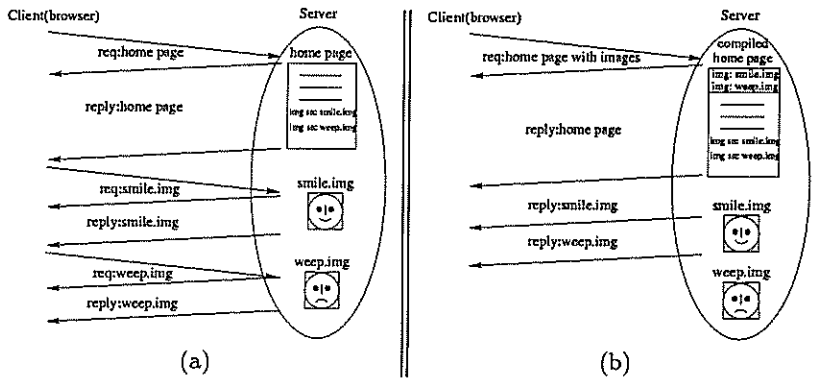

Figure 5: Auto Inline Download:

Figure 5(a) shows a typical web transaction without AID, and Figure 5(b) shows a transaction with AID. The round trip times saved become more significant when the images themselves are small, as is the case with the current web. Most images are icons of various types, many of size less than a few hundred bytes.

It could be wasteful to always dump all the inline data in a web page to the client. Consider the case of a client that is browsing through pages at a company web site. Most of the pages have the same logo image; after the first access, the client might have cached the logo locally. By sending all the images with each web page, the server is loading the network unnecessarily.

To solve this problem, we suggest using a client request that has three levels of granularity. In the original request for the web page, the client could ask the server to send *all* the images, only the *unique* ones, or *no* images at all. The compiled image information can keep track of a counter per image, which indicates how many other files include the same inlined image. We believe that there are likely to be just two categories: the ones that are unique to that file, and those that appear in a lot of files. Thus a request with three levels of granularity appears sufficient.

With this modification, the protocol becomes:

1. The client sends a request to the server for a web page; in the request header, the client indicates that it needs to receive *all*, only *unique*, or *no* images. If it is contacting this server for the first time, it should ask for all images; from then on, it should ask for only unique ones. The *no* flag should be used only when it does not want any images.

2. When the server receives the request, it sends the web page, and the correct set of images.

3. When the client receives the reply, it can examine the image header and see which files the server will send, and request the remaining files that it does not have. In the best case, the client will not have to ask for any images.

Some images could be on a different machine. Although this is typically not the case, the protocol should correctly handle this case. Obviously the server cannot automatically send any image that it does not have; the client has to retrieve it by establishing a connection to the other machine. But it can help the client retrieve it faster by adding information about the IP address of the machine where the image resides so that the client should not have to do a DNS lookup.

AID reduces the number of round-trips that the client has to wait to receive all the pieces it needs to display the page. Our performance measurements for AID are given in Section 8.

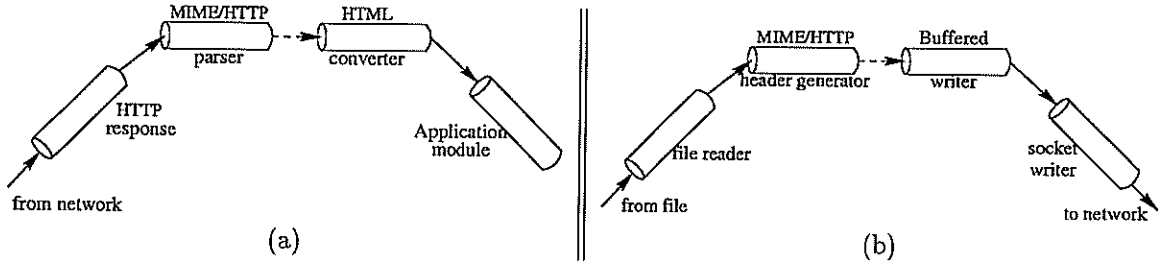

Figure 6: LibWWW control flow for client and server: Once the stream modules are set up, data flows through the pipes asynchronously.

8. Prototype Implementation Using `libwww`

We used `libwww5.0a`, the reference implementation released by the World Wide Web Consortium (W^3C). `Libwww` is a library, in C, that can be used to build web applications. The w^3c release has a client which is HTTP/1.1 compliant, but the server was not fully functional. We modified it to work as a simple server that supported the basic function of receiving a request and sending replies over persistent connections.

`Libwww` uses the abstraction of stream modules.[4] To process each request, the application (a client or a server) creates a chain of streams, that has a *reader* at one end, and a *writer* at the other. Both the *reader* and the *writer* use Unix style file descriptors, allowing them to work uniformly on the network as well as on local files. Thus we can have any of the four combinations: transfers from file/network to file/network. Each request is processed asynchronously (using non-blocking I/O) as its input and output descriptors become ready for data transfer. An event loop coordinates various events (network reads and writes, file reads and writes etc.), and invokes the appropriate stream chain for the event.

Figure 6 illustrates[5] a typical web transaction. At the client side, the client establishes a connection to the server and sends a request. When it receives a reply, the socket reader directs it to the HTTP-Response stream (see Figure 6(a)), which checks for the success/failure of the request; if the request succeeds, the reply is passed to the MIME/HTTP header parser. After parsing the reply format, it uses the length derived from the header to locate the end of the message in the stream and directs the data part to the appropriate converter. For an HTML file, the data is passed through an HTML parser and then to the application module.

Upon receiving the connection request, the server program creates an instance of a server object; this server object is responsible for all the transactions over that connection. When the server object receives a request, it sets up a chain of stream modules to read the corresponding file, and generates all necessary headers, based on the file type and length. The stream chain directs the data to the network, headers first, via a buffered writer stream.

The following subsection describes our implementation strategy. We have implemented a version of Auto Inline Download on `libwww` and we are in the process of implementing Informed Server Proxying. The following subsection describes the suggested modifications for our schemes, and the next subsection discusses the measured performance of the modified and unmodified versions.

[4] These are not streams in the system V sense; they are just objects implemented in C.
[5] This figure is adapted from the online manual pages at w^3c.

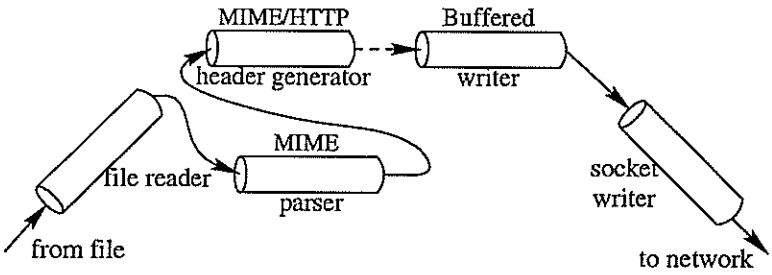

Figure 7: LibWWW control flow for modified server: A MIME parser module is inserted in order to understand the added hint headers in the file.

8.1. Modifications

The object oriented design of the `libwww` was very helpful in adding a few pieces of code to achieve the modifications we wanted.

The main modification allows the server to parse the new MIME headers prepended by the preprocessing step. Since there was already a MIME parser stream, which collected all the MIME headers into a list and dispatched appropriate call back functions, we decided to reuse the code. First, we augmented the MIME parser so that it could parse the new headers. Then, we inserted a MIME parser (see Figure 7) in the server's chain of streams that serve the request. Thus, the reader reads the file into a MIME Parser, and then it passes it to the header generators. All the inline data files are collected in a list created by the MIME parsers, and this list is used to send the files to the client, immediately following original HTML file. The client uses the list of inline data, created by the client side parser, to wait for the inline data without sending additional requests.

In order to support *Stored Address Binding*, the MIME parser adds an auxiliary cache of name to address translations, and inserts all the host names in URLs (and the associated addresses) into this cache. When the client tries to access a URL, it first looks up its local DNS cache; if the translation is not found there, it looks up the auxiliary cache created from the retrieved file. If the address is found in the auxiliary cache, a connection is attempted to that address. In most cases, this address should be correct; if it is incorrect, or the address is not found in the auxiliary cache, the client calls `gethostbyname()` to resolve the host name.

Since our implementation is only a prototype for verifying our ideas, we have not made additional efforts to do extensive error handling; nor do we ensure that the server and client are completely compliant to the specifications. However, our experience suggests that the modifications are quite simple and can easily be integrated into `libwww`.

8.2. Results

We conducted experiments to evaluate the benefits of AID and ISP. In both experiments we used a client running on a SPARCstation 20. In the experiment for AID, we used a server running on a SPARCstation 5 and in the experiment for ISP, we used 8 other Sun machines. (Some are sparcstations, others are older Suns). All the machines are on a 10Mbps Ethernet. All of the servers ran SunOS 4.1.3, and we used `gcc` with `-O` for compiling the applications and the library.

Evaluation of AID: In this experiment the client retrieved a file which contained k inline images all of the same size, where k is a parameter to the experiment. We captured the packet trace using `tcpdump` to count the number of packets used, and the number of bytes sent. `Tcpdump` also proved to be a useful tool in debugging the client and server programs.

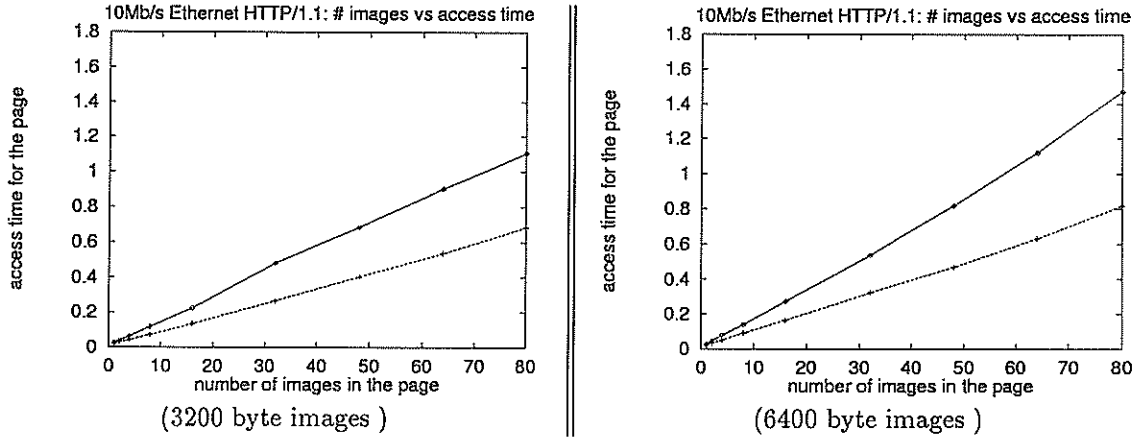

Figure 8: Comparing HTTP/1.1 and HTTP/1.1 with AID: the benefit of AID (reduced access time) increases with the number of images in the file.

	HTTP/1.1 (persistent)		HTTP/1.1 persistent w/ AID	
	nd=1	nd=0	nd=1	nd=0
Packets (cli→srv)	102	102	48	49
Request bytes	9275	9275	215	215
Packets (srv→cli)	153	150	151	144
Reply bytes	163561	163561	163561	163561
Total packets	255	252	199	193
Elapsed time	0.68s	9.88s	0.41s	0.44s

Table 5: Analysis of accessing a web page with 48 images each of size 3200 bytes. The access is over a LAN (10Mbps Ethernet) between two sparcstations. AID reduces the number of requests from the client to the server. (nd = TCP_NODELAY flag.)

Figure 8 shows the comparison between normal HTTP/1.1, and HTTP/1.1 with AID support. In these experiments, the requests from the client are not pipelined. Therefore, the reduction in access time (40% for files with 48 images) is almost entirely due to the number of round-trips saved. This experiment is conducted on a 10Mbps LAN where the round-trip times are less than 1ms; over a WAN the gains are expected to be much higher.

Theorem 5 shows the details of accessing an HTML file with 48 images, each of size 3200 bytes. As expected, the number of packets from the client to the server has reduced. In fact, almost all the remaining packets from the client to the server are TCP control packets (SYN, FIN, and ack), except the original request. With AID, the web transaction looks like one small request followed by one large reply, instead of many short requests and replies. This helps TCP significantly, since TCP is optimized for one way transfer of data.

This table also shows the data for the same experiment with TCP_NODELAY set to 0. (This is the default behavior.) One would expect more reduction in the total number of packets, when TCP_NODELAY is set to zero, because Nagle's algorithm is enabled; this in turn allows the sender TCP to buffer and combine many user writes into a single TCP packet. But most of the writes in our experiment were not small; if we transferred 100 byte images, there might have been significant reduction in the number of packets when we use AID with TCP_NODELAY set to 0. Since Nagle's algorithm can introduce delays, [NGSP97] advocates disabling it. With AID, we may not have to disable Nagle's algorithm.

	HTTP/1.1 (persistent)		HTTP/1.1 persistent w/ ISP	
	nd=1	nd=0	nd=1	nd=0
Packets (cli→srv)	49	49	13	13
Packets (srv→cli)	38	39	12	12
Total packets	87	88	25	25
Elapsed time	0.41s	0.53s	0.13s	0.13s

Table 6: Accessing a chain of 9 documents using HTTP/1.1 without and with ISP. Note that the minimum number of packes from client to server for 9 documents is 9 request response pairs and the connection setup and tear down packets. (nd = TCP_NODELAY flag.)

Evaluation of ISP: In order to evaluate ISP, we conducted the following experiment. We created a chain of 9 documents. In other words, the first document has a hyperlink to the second, the second document has a hyper link to the third, and so on. We distributed these 9 documents to 9 different machines, all on the local network, and ran servers on each of them. From a 10th machine, the client browsed through the chain, by first connecting to the appropriate server, retrieving the document and following the hyper link to the next document. We cache all documents on all servers, and with ISP, if a client connects to any one of the machines, it can retrieve all documents without any additional connections.

Theorem 6 show the details of transactions with and without ISP. Without ISP, the client has to make 9 connections, since these documents are on different servers, and it takes 87 packets. But with ISP, the client has to make only one connection and it can retrieve all 9 documents from the same connection. Therefore, it takes only 25 packets. ISP saves time by avoiding connection setup.

9. Future Work

The possibility of *automatically* rewriting an HTML file opens new avenues for optimizing web accesses. Some possibilities are listed below:

1. One can compile a page for faster rendering on specific platforms. For example, a very popular page could be compiled into many browser specific files, where each file can be rendered fast on one of the browsers. Large HTML files can be kept compressed for those clients which are accessing it across a low bandwidth link; clients could ask for compressed files in their request header.

2. Each link can be annotated with a counter which is incremented when a client requests the link. This counter counts the number of times the referenced page has been accessed through this link. The count can be used as a measure of popularity of the link, and can be helpful in making caching or prefetching decisions.

3. Each link can be annotated with the size of the document to be fetched; this value can be used by the clients and proxies to overlap memory management with communication.

4. Each link can be annotated with the server's public key. This can possibly eliminate one round-trip in obtaining the public key from the server, in case an authentication is required. The stored key must only be used as a hint by a client, just as with stored IP address.

5. Each link can be annotated with a cookie or a hint. If the client presents this hint to the server of the link, the server can use this hint for faster lookup.

10. Conclusions

Complex distributed systems have a *space* and *time* dimension: the Web is no exception. One can consider adding performance optimizations at either the various nodes in the distributed system (*space*), or at the different time scales (*time*) at which the system is instantiated. We propose taking advantage of the space dimension by placing hints about Web access points at other points which *refer* to these access points. Since the client has to obtain the reference point page anyway, adding a small amount of hint information costs very little in terms of network bandwidth or latency. We also propose taking advantage of the time dimension in a Web system by automatically compiling user pages when the user first publishes a page. Since publication time typically precedes the time at which a page is read by a large delay, the compilation delay is effectively masked.[6]

All four of our optimizations are based on this paradigm of automatically compiling in hints (about the page itself and other referenced pages) to a web page. AID allows the server to download inline data directly without parsing the HTML file or waiting for separate client requests. The hints required for AID must specify the inline data associated with the file. The other three mechanisms (SAB, ISP, SLR) rely on hints associated with a referenced page. All three are essentially based on adding a mapping from the referenced URL to an IP address. If the client must go to the original server, the IP address is that of the original server (SAB); if the client can obtain the page from the reference point, the IP address is implicitly (using the *locallyCached* flag) that of the reference point (ISP); finally, if the client is redirected to a completely different server, the IP address is that of the new server.

AID removes the round trip delays associated with requesting inline data (though pipelining helps, it cannot avoid at least one extra round trip, and possibly more for applets). SAB often removes the round trip delay for DNS lookup. ISP can remove the round trip delay required to set up a fresh connection to the server of a referenced page. Finally, SLR can actually *increase* round trips but adds great flexibility in allowing dynamic load balancing. All four techniques are based on hints; the client must be prepared to deal with incorrect hints. All four techniques can be gradually added to newer HTTP implementations; hints are never sent to older client implementations, and are never expected from older server implementations.

We have described preliminary implementation details and some measurements that indicate the performance benefits of AID, SAB, and ISP. We are currently completing the implementation to incorporate ISP and SLR more fully. Based on our preliminary results, we are optimistic that our *simple* ideas can *easily* be added to current implementations to improve the ease with which ordinary users can surf the sea of Information that is found in the Web today.

References

[AW96] M. Arlit and C. Williamson. Web server workload characterization: The search for invariants. In *In proceedings of SIGMETRICS'96*, May 1996.

[BLC95] T. Berners-Lee and D. Connolly. Hypertext markup language - 2.0. *RFC 1866*, November 1995.

[BLFN96] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. Hypertext transfer protocol – http/1.0. *Informational RFC 1945*, May 1996.

[Bri95] T. Brisco. Dns support for load balancing. *RFC 1794*, April 1995.

[6] Our compilation requirements are also quite modest, involving parsing of the HTML file, potentially doing DNS requests, and resolving all the inline data associated with the page.

[CDN+96] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical internet object cache. In *USENIX 1996 Annual Technical Conference*, pages 153–163, January 1996.

[FB96] N. Freed and N. Borenstein. Multipurpose internet mail extensions. *RFC 2045, Network Working Group*, November 1996.

[FGM+97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T Bernes-Lee. Hypertext transfer protocol – http/1.1. *RFC 2068*, January 1997.

[GS97] Linda Geppert and William Sweet. Introduction. *IEEE Spectrum*, 34(1):23–26, January 1997.

[HC94] Mark Handley and Jon Crowcroft. *The World Wide Web – Beneath the Surf.* UCL Press, 1994.

[Hei96] J. Heidemann. Performance interactions between p-http and tcp implementation. *http://www.isi.edu/lsam/publications/phttp_tcp_interactions/*, November 1996.

[LA94] A. Luotonen and K Altis. World wide web proxies. *Computer Networks and ISDN Systems*, 27(2), 1994.

[MLB95] R. Malpani, J. Lorch, and D. Berger. Making world wide web caching servers cooperate. In *In 4th International World Wide Web Conference*, pages 107–117, December 1995.

[Moc87] P. Mockapetris. Domain names–concepts and facilities. *RFC 1034*, April 1987.

[Mog95a] J. C. Mogul. The case for persistent connection http. In *Proceedings of the ACM SIGCOMM '95 Symposium*, pages 299–313, September 1995.

[Mog95b] J. C. Mogul. Operating systems support for busy internet servers. *WRL Technical Note TN-49*, May 1995.

[NGSP97] Henrik Frystyk Nielsen, Jim Gettys, Anselm Baird Smith, and Eric Prud'hommeaux. Initial http/1.1 performance tests. *http://www.w3.org/pub/WWW/Protocols/HTTP/Performance/Pipeline.html*, January 1997.

[WAS+96] S. Williams, M. Abrams, C. R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal policies in network caches for world wide web documents. In *Proceedings of the ACM SIGCOMM '96 Symposium*, pages 293–305, October 1996.