Maintaining Strong Cache Consistency in the World-Wide Web

Pei Cao and Chengjie Liu

Department of Computer Science
University of Wisconsin-Madison
Madison, WI 53706
{cao,chengjie}@cs.wisc.edu

Abstract

As the Web continues to explode in size, caching becomes increasingly important. With caching comes the problem of cache consistency. Conventional wisdom holds that strong cache consistency is too expensive for the Web, and weak consistency methods such as Time-To-Live (TTL) are most appropriate. This study compares three consistency approaches: *adaptive TTL*, *polling-every-time* and *invalidation*, through analysis, implementation and trace replay in a simulated environment. Our analysis shows that weak consistency methods save network bandwidth mostly at the expense of returning stale documents to users. Our experiments show that *invalidation* generates a comparable amount of network traffic and server workload to *adaptive TTL* and has similar average client response times, while *polling-every-time* results in more control messages, higher server workload and longer client response times. We show that, contrary to popular belief, strong cache consistency can be maintained for the Web with little or no extra cost than the current weak consistency approaches, and it should be maintained using an invalidation-based protocol.

Keywords: *World Wide Web*, *Cache Consistency*, *Invalidation Protocols*, *Distributed Systems*, *Performance Analysis and Measurements*.

1 Introduction

The exploding popularity of the World Wide Web (WWW) has led to exponentially increasing traffic on the Internet. Since the network infrastructure does not grow at an exponential rate, the increase of network load has lead to increased latency in accessing Web documents. Fortunately, caching can reduce both network traffic and document access latency. By caching replies to HTTP requests and using the cached replies whenever possible, client-side Web caches reduce the network traffic between clients and Web servers, reduce the load on Web servers, and reduce the average user-perceived latency of document retrieval.

For Web caches to be useful, however, cache consistency must be maintained, that is, cached copies should be updated when the originals change. We define *weak* consistency as the consistency model in which a stale document might be returned to the user, and *strong* consistency as the model in which after a write completes, no stale copy of the modified document will ever be returned to the user. The exact definition of the completion of a write varies by the consistency approaches.

Existing Web caches mostly provide weak consistency. That is, a stale document might be returned to the user, though infrequently. Weak consistency mechanisms include TTL (Time-To-Live), in which a client considers a cached copy up-to-date if its time-to-live has not expired, and client polling, in which a client periodically contacts Web servers to verify the freshness of cached copies. Weak consistency, however, is not always satisfactory. Users have to be aware that the browser may occasionally display a stale page. To make sure that a requested document is up-to-date, a user has to instruct the browser to "reload", which means contacting the Web server to validate the cached copy. Reload not only burdens the user, but also burdens the Web server. Essentially, lack of strong consistency reduces the effectiveness of client caches as a way to improve the scalability of the Web.

In this paper, we investigate the cost and performance of two approaches that provide strong consistency, *invalidation* and *polling-every-time*. In the invalidation approach, the Web server keeps track of all the client sites that cache a document, and when the document is changed, sends invalidation messages to the clients. A write is considered complete when invalidation messages reach all of the relevant clients. In the polling-every-time approach, every time the user requests a document and there is a cached copy, the cache first contacts the Web server to validate the cached copy, then returns the copy to the user. In this approach, a write is complete when the modification is registered in the server's file system.

We compare invalidation and polling-every-time with a widely-used weak consistency approach, *adaptive TTL* [6], through analysis, implementation and trace replay experiments. Adaptive TTL is shown to perform the best among existing weak consistency protocols [11]. Using a simple model that captures the interleave of document requests and modifications, we demonstrate that weak consistency protocols save network bandwidth mostly at the expense of returning stale documents to the user, and the comparison of invalidation and polling-every-time depends on the relative frequency of document requests and modifications.

We implemented the three consistency approaches in the popular Web caching system Harvest [7], and compared their performance by replaying Web server traces through the prototypes running on workstations connected by an Ethernet. Our experiments show that the invalidation approach performs the best among the three consistency approaches, It provides strong consistency at a cost that is similar to that of adaptive TTL. Compared with adaptive TTL, invalidation generates similar (within 6%) number of network messages and imposes similar server load (within 3%). It also has similar client response times, though occasionally a request is stalled for a long time (due to an inefficiency in our current implementation, the server does not accept new requests until it finishes sending all invalidation messages). Polling-every-time, on the other hand, generates significantly more network messages (as high as 50% in some experiments) and imposes higher server CPU load. Thus, strong cache consistency can be maintained on the Web with little or no extra cost than the current weak consistency approaches, and invalidation-based protocols are the appropriate methods.

2 Related Work

Our study is motivated by recent work on Web caching and Web cache consistency. In particular, Gwertzman and Seltzer's paper [11] gave an excellent comparison of cache consistency approaches via simulation, and concluded that a weak-consistency approach such as adaptive TTL would be best for Web caching. The main metric used in [11] is network traffic. The study did not address many other important questions, such as server loads, client response times, and consistency message latency.

Another study that is similar to ours is Worrell's thesis [21]. The study investigates using invalidation as the consistency approach in hierarchical network object caches. It compares invalidation with a *fixed* TTL approach, in which a single time-to-live is assigned to all files. The study concludes that invalidation is a better approach for cache consistency. However, the results in [21] relies on the existence of a hierarchical caching structure, which significantly reduces the overhead for invalidation. Unfortunately, hierarchical caches are not yet widely present in the Internet. Thus, we focus on invalidation in the absence of caching hierarchies.

Another study [20] describes a light-weight caching server that employs both adaptive TTL and invalidation for cache consistency. However, the paper focuses on comparing the performance differences between the light-weight server and the CERN proxy server, and does not compare the consistency approaches.

Though we chose the Harvest [8, 7] system for our implementations, there are many other Web caching software. Popular browsers such as Netscape and the Internet Explorer all offer some form of client caching. CERN proxy [17] offers caching for all its clients, using TTL as the consistency mechanism. We choose Harvest due to its source code availability and good performance [20].

Cache consistency problems exist in any system that uses some form of cache to speed up accesses. In particular, cache consistency protocols have been studied extensively in computer architecture [12], distributed shared memory (DSM) [16, 5], network and distributed file systems [19, 13, 18], and distributed database systems [9, 10]. The consistency problems are slightly different in the four contexts.

In computer architecture and DSM systems, the consistency algorithms must handle multiple writers to a data item, and are subject to the most stringent limits on CPU and storage overheads. The World Wide Web so far provides only a single-writer multiple-reader interface for document retrieval, and processors as well as storage are relatively fast and cheap compared to the Internet bandwidth. On the other hand, some of solutions in the computer architecture and DSM contexts apply to the Web. For example, our initial interest in this work came from discussions with Stefanos Kaxiras and James Goodman at University of Wisconsin at Madison on applying the GLOW scalable hardware shared-memory coherence protocol [14] to cache consistency on the Web. GLOW uses hierarchical caching and hardware multicast network for invalidation messages. Though we did not pursue the idea due to the lack of hierarchy on the Internet, GLOW-style multicast protocols can potentially improve the performance of invalidation significantly by saving network traffic and reducing server load. This remains part of our future work.

In distributed databases, the database system must provide transactional guarantees over a set of data accesses in the presence of caches [9, 10]. In comparison, the Web provides a much more primitive interface. Thus, the Web cache consistency problem is much simpler than the database cache consistency problem. However, many of the performance tradeoffs between polling (called "validity check" in [10]) and invalidation (called "change notification" in [10]) are similar in both contexts. In this paper we mainly compare strong cache consistency protocols with the current weak cache consistency approach in the Web. We plan to look more into leveraging the techniques in database consistency algorithms to improve the performance of Web consistency protocols.

Of the four contexts, the cache consistency problem of network and distributed file systems [19, 13, 18] is most similar to the Web consistency problem. The main differences are that the Web is orders of magnitudes bigger than any distributed file system, and the systems participating in the Web are heterogeneous, use different operating systems and belong to different organizations. Despite the differences, there are similarities in the solutions. For example, the TTL approach is very similar to the NFS protocol for cache consistency [19], the polling-every-time approach is similar to what is adopted in the Sprite file systems (clients contact the server on every file open/close) [18], and invalidations are essentially callbacks in AFS [13]. The unique challenge in the Web is to scale these protocols to the size of the Internet. Though many recent studies on distributed file systems followed the trend of letting client workstations assume more responsibilities, including caching, consistency maintanence and failure resilience [4, 1], these techniques do not easily apply to the current Web because most web clients have limited resources.

3 Consistency Approaches

This section discusses in more detail the three cache consistency approaches, the load they put on the network, and the consistency they provide under the current Internet.

3.1 Cache Consistency Maintanence

The current HTTP protocol provides two mechanisms for cache consistency. Each URL (Universal Resource Locator) document has a "time-to-live" (or "expire") field, which is an a priori estimate of how long the document will remain unchanged. The time-to-live field can be used by the cache manager to determine if a cached copy is up to date. In addition, each client can send an "if-modified-since" request, containing the URL of the document and a timestamp, to the Web server. Upon receiving the request, the server checks whether the document has been modified since the timestamp. If so, the server returns the status code "200" and the new data; otherwise, the server returns the code "304", which stands for "document unmodified."

Existing Web caches mostly utilize combinations of two basic approaches for cache consistency. The TTL approach maintains cache consistency using the copy's time-to-live attribute; a cached copy is considered valid until its TTL expires, at which point the next request to it results in an "if-modified-since" message. The client polling approach sends an "if-modified-since" request every time the validity of a cached copy needs to be verified.

The difficulty with the TTL approach is that it is often hard to assign an appropriate time-to-live for a document. If the value is too small, the server will be burdened with many "if-modified-since" messages, even when the document is not changed. If the value is too large, the probability that the user will see a stale copy of the document significantly increases. Similar difficulties exist with the client polling approach in deciding when to send "if-modified-since" requests.

The **adaptive TTL** approach handles the problem by adjusting a document's time-to-live based on observations of its lifetime. The approach, also called the Alex protocol, was first proposed in [6]. Adaptive TTL takes advantage of the fact that file lifetime distributions tend to be bimodal [4, 2]; if a file has not been modified for a long time, it tends to stay unchanged. Thus, in adaptive TTL, the cache manager assigns a time-to-live attribute to a document, and the time-to-live is a percentage of the document's current "age", which is the current time minus the last modified time of the document (this information is provided in the header of a HTTP reply).

Studies [6, 11] have shown that adaptive TTL can keep the probability of stale documents within reasonable bounds ($< 5\%$). The Harvest cache manager [7]

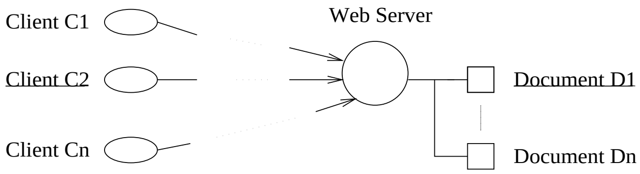

Figure 1: Modeling clients' requests to documents.

mainly uses this approach to maintain cache consistency, with the percentage set to 50% [1].

Though adaptive TTL keeps the frequency of stale documents low, it does not eliminate its occurance. Two other approaches can provide stronger consistency guarantees.

The **polling-every-time** approach simply sends an "if-modified-since" request every time a request for a document hits in the cache. The approach has the advantage that it can be implemented easily in the existing HTTP protocol, and indeed the Netscape browser allows users to select this approach for cache consistency. The problem with this approach, however, is that the user waits a network round-trip latency on every document retrieval, even though the document itself is cached.

The **invalidation** approach relies on the server to send out notifications when a file is modified. It is similar to the cache consistency mechanism used in the wide-area file system AFS. The server keeps track of all the client sites that cache a particular document, and when the document is changed, the server sends out invalidation messages to all the clients. Upon receiving an invalidation message, the cache deletes the cached copy (if there is one), but does *not* retrieve a new copy. The advantage of this approach is that it eliminates the stale copy problem (subject to network connectivity) with low cost. The disadvantage of this approach is that current HTTP protocol does not include invalidation messages.

3.2 Analysis of Network Traffic

We can analyze the network traffic contributed by the three approaches through a simple model. There are many Web servers and clients in the Web. A particular Web server may have many documents, and more than one clients may request the document (Figure 1). If we focus on a particular client C_2's accesses to a particular document D_1, we can calculate the network traffic due to delivering D_1 to C_2. The total network traffic is then the sum of the network traffic incurred by each client-document pair.

During a time period, C_2 may request to view D_1 multiple times, and D_1 may change many times as well. The interleave of requests and file modifications can be expressed as a stream: "r r r m m m r r m r r r m m r m." The stream completely captures the events relevant to C_2's accesses to D_1, for the period from C_2 bringing D_1 into its cache till C_2 evicting D_1 from its cache. That is, the first "r" is the regular GET request, and the last "r" is the request that hits in the cache just before D_1 is replaced. If D_1 is changed before C_2 brings it into cache again, the stream is ended by a file modification "m." Assuming C_2's cache makes the same cache replacement decisions regardless of the consistency protocol, we only need to look at a single stream to compare the network traffic of the consistency approaches.

We distinguish between two types of network traffic: control messages and file transfers. The control messages include regular GET requests, "if-modified-since" requests, invalidation messages, and "304" replies. The reason to distinguish the two is because control messages are usually much smaller than file transfers. The average size of control messages is usually less than two hundred bytes, while file transfers vary from a few thousand to over tens of thousands of bytes.

Let R be the number of times C_2 views D_1, and let RI be the number of intervals during which C_2 repeatedly requests D_1 while D_1 is unchanged. For example, R is 9 in the above sequence, and RI is 4. Assuming the cache at C_2 always has space for D_1, the *minimum* amount of network traffic needed to ensure that C_2 always sees an up-to-date version of D is RI control messages plus RI file transfers.

The consistency protocols generate different messages:

- In adaptive TTL, the first request is a regular GET request, and subsequently, for each request that hits in the cache but the document's time-to-live has expired, an "if-modified-since" message is sent. Of all the "if-modified-since" requests, only those that are the first "if-modified-since" requests in their access intervals receive a reply carrying the new document, and all others receive a reply "file not changed." If an "if-modified-since" request does not occur in an access interval, all requests in the interval are stale hits, and the interval is called a stale-hit-interval. A file transfer occurs upon the first "if-modified-since" request

[1] Harvest also uses the "time-to-live" field of a document if the Web server provides one. In addition, the percentage and maximum TTL can be adjusted and set for specific types of documents through configuration files.

in an access interval.

- In polling-every-time, except the first request, all requests are cache hits and result in "if-modified-since" requests, of which only $RI-1$ requests are useful (receiving new documents), and the rest only result in "304" responses. The number of file transfers is exactly RI, since the file is delivered once at the beginning of every access interval.

- For invalidation, the number of invalidation messages is the number of access intervals, not the number of file modifications. The reason is that once an invalidation is sent, the client no longer caches the document, so unless the client accesses the document again, there is no need to notify the client about the document's changes. Thus, an invalidation message is only sent upon the first modification after an access interval. The first request in each access interval generates a new GET request, resulting in RI GET requests, and RI file transfers.

Table 1 shows the control messages and file transfers of the three consistency approaches.

We can make the following observations from Table 1:

- The only times when adaptive TTL saves file transfers over the other approaches are when stale documents are returned to user. Since network bandwidth is mostly consumed by file transfers, this means that the bandwidth saving of the TTL approach comes mainly at the cost of stale hits.

- Invalidation incurs at most twice the minimum number of control messages; both polling-every-time and invalidation incur the minimum number of file transfers.

- Though adaptive TTL also incurs fewer control messages than polling-every-time, it may incur more control messages than invalidation if the TTL expires and the document is not changed.

- The comparison of polling-every-time and invalidation depends on the relative frequency of requests and modifications. If modifications happen often but requests happen infrequently, then invalidation incurs an extra invalidation message on every request. On the other hand, if requests happen frequently between modifications, polling-every-time may generate too many validation requests.

Thus, strong consistency mechanisms do not necessarily consume more network bandwidth than weak consistency mechanisms.

3.3 Consistency in the Presence of Failures

Our discussion so far ignores the fact that in today's Internet, networks, servers, and clients may fail at any time. The three approaches provide different consistency guarantees in the presence of failure.

Polling-every-time guarantees that when the user requests a document, the returned document is up-to-date at the time when the server processes the "if-modified-since" message. If a server or network failure happens when the user requests a document, polling-every-time can inform the user of the failure. The user can then choose to view the cached copy, while being aware that the copy maybe stale and server or networks failures are hindering the propagation of updates.

Invalidation guarantees that when the user requests a document, the returned document is up-to-date within the time it takes the server to send an invalidation message to the client. However, if a network partition prevents the flow of information from the server to the client, the user might view stale documents for an extended period of time without knowing it. The only way to guard against it is for the client to periodically contact the server to make sure the network links and the server are up.

Depending on implementation, invalidation can guarantee that the cached files are in a "casually" consistent state even when a network failure occurs. Essentially, if the messages are delivered in order (as in TCP), and the server sends out invalidations for document A before it sends out invalidations for document B, then if a client sees the new version of document B and then requests document A, the client is guaranteed to see the new version of document A or be notified that a network failure occurred. This guarantee can be quite useful in certain situations.

Adaptive TTL can only guarantee that the returned document is up-to-date within the specified time-to-live period. To keep overhead low, time-to-live must be similar to the life-time of the document, which can be hours, days or even months. Thus, it is much larger than the network delays or the invalidation processing delays, which are often well within a second. Thus, strong consistency approaches provide a much better guarantee on the "freshness" of documents.

Messages	Polling-Every-Time	Invalidation	Adaptive TTL
"GET" Requests	1	RI	1
If-Modified-Since	R-1	0	TTL-missed - 1
304 replies	R-RI	0	TTL-missed - TTL-missed-and-new-doc
Invalidation	0	RI	0
Total Control Msg	2R-RI	2RI	2*TTL-missed - TTL-missed-and-new-doc
File transfers	RI	RI	RI - stale-hit-intervals

Table 1: Message counts for the three consistency approaches. R is the total number of requests, and RI is the interval of requests with no intervening modification.

4 Implementation

We implemented all three consistency approaches in the Harvest Web caching system. (Source codes can be found in http://www.cs.wisc.edu/~cao/icache.) We ignore the support for cache hierarchies in Harvest, and all consistency operations happen between the Web server and individual cache sites. For example, invalidation does not depend on higher level caches to send out invalidation messages to lower level caches — the Web server sends invalidation to all cache sites.

4.1 Adaptive TTL

We improved the original adaptive TTL implementation in Harvest. Everytime a cache hit happens, the cache entry's time-to-live is checked. If it expires, the original implementation deletes the entry and sends a regular GET request for the document. Our implementation keeps the cached file and sends an "if-modified-since" request instead.

4.2 Polling-Every-Time

For polling-every-time, we simply send an "if-modified-since" request to the server every time a cache hit happens. If the server replies with status code 304, the cached copy is returned to the user. If the server replies with the document, the cached copy is deleted, the new copy is put in the cache and returned to the user.

4.3 Invalidation

The Harvest cache software provides both a proxy cache for client browsers and an HTTP accelerator for Web servers. The accelerator intercepts HTTP requests by running on port 80 and putting the Web server on port 81. The original purpose of the accelerator is to improve server performance by keeping a main memory cache of URL documents. We implemented the invalidation approach in the accelerator to avoid modifying the server.

The accelerator has to perform three basic operations: keeping track of the remote sites that cache a copy of a document, detecting changes to the document, and sending out invalidation messages.

Keeping track of client sites The accelerator maintains an invalidation table which records, for each URL document, a list of remote sites that accessed the document since the previous invalidation of the document. We do not rely on the client telling the server whether it will cache a document; rather, every time a client accesses the document, we assume that it might cache the document and add its address to the remote site list. (For those who are worried about scalability issues, see Section 6.)

Detecting changes Detecting modifications to a document is surprisingly nontrivial. There are many ways a URL document can be modified, including all the editors and shell commands like "cp." The only reliable way to detect changes to a file is at the file system level. One would like a trigger mechanism in the kernel to dispatch a user-level handler every time a file is modified. Unfortunately, most operating systems do not provide such mechanisms.

We identify two approaches for the accelerator to detect changes to a document. The first, "notify", provides a check-out/check-in mechanism for the user. The document is considered changed when it is checked-in, and the check-in utility automatically informs the accelerator about it. The second approach takes advantage of the fact that users often invoke the browser to see a document when they change it. Thus, when the proxy server sees a request from the browser for a local document, it suggests to the accelerator to check whether the document has been modified. This means that each entry in the invalidation table needs to keep a timestamp of when the document was last seen modified. A third alternative is for the accelerator to check the status of all files periodically. We ruled it out because the overhead is prohibitively high for even moderate numbers of files. We implemented both "notify" and the browser-based approach in Harvest.

Sending invalidations We added a new HTTP message type: INVALIDATE. An INVALIDATE message can carry either a URL or the Web server address. In the former case, a proxy cache that receives the message checks to see if the URL is cached. If so, it deletes the cached copy; if not, it ignores the message. In the latter case, the proxy cache checks to see if it has copies of documents from the Web server, and marks those copies as questionable. A questionable copy needs an "if-modified-since" message before it can be returned to the user. This form of message is used when the Web server site fails.

Upon detecting changes to a document, the accelerator sends out INVALIDATE messages carrying the URL to all the client sites. Once a client receives the invalidation message, the accelerator deletes it from the site list of the document. Thus, if the client does not access the document again, it will not receive future invalidations.

Handling Failures: There are three failure scenarios. The first is when a proxy is down and misses an invalidation message. Our solution is simply to let the proxy mark all its cache entries as questionable when it recovers. When the user requests a questionable entry, the proxy checks with the server using "if-modified-since."

The second scenario is when the server site fails (i.e. both the accelerator and the Web server die). When the server site recovers, it must send out invalidation messages for the documents that were changed during the failed interval. This requires that site lists for all documents survive the failure. We could implement it by logging every HTTP request to disk before servicing it, but the overhead would be too high.

Our solution is to store on disk a list of all the sites that ever received a document from the server. When the accelerator recovers, it sends an INVALIDATE message carrying the Web server address to all sites in its list. The solution incur low overhead: a disk access is only necessary when a new client site which has never been seen before contacts the server. The accelerator keeps an in-memory table of all the site addresses it has seen, and updates the list on disk when a new site enters.

The third scenario is when network partitions occur between the server site and the client site. This is the hardest failure case to handle. It is difficult to maintain strong consistency in the event of network partition. Our current solution is to use TCP to send invalidation messages, and when the TCP message fails, use periodic retry.

5 Performance Comparison

We compare the performance of the three approaches by replaying Web server traces in a simulated environments. Below we first discuss the performance metrics we are interested in, then the Web traces and the simulation scheme, and finally our results.

5.1 Performance Metrics

Our performance comparison of the approaches includes the following criteria.

- **total number and bytes of messages:** the total number and byte count of messages sent by the clients and the server, including the messages needed to service HTTP requests and to maintain cache consistency.

- **client response time:** the latency from sending an HTTP request till receiving the document, measured from the browser; it reflects the delay experienced by the user.

- **server load:** the average load experienced by the server to satisfy the incoming requests and maintain cache consistency, measured as CPU and disk utilizations.

- **stale hits:** since adaptive TTL does not maintain strong consistency, a stale copy may be returned to clients. We count all such occurances.

- **invalidation cost:** one of the main concerns for invalidation is that the server may have to maintain a large list for each document to record all sites that have accessed the document (since its last invalidation), and the cost of sending invalidate messages to all the sites maybe high. Thus, we keep track of the total storage required by the site lists, and the time it takes the server to send all the invalidation messages for each modification.

We instrumented our Harvest implementation to keep appropriate counters for these measurements. The server load is measured by running an "iostat" on the server machine; the "iostat" process reports the disk and CPU utilizations for each minute. We report the average of the utilization ratios for the duration of each trace replay.

5.2 Web Access Traces

We used five Web server traces from the Internet Traffic Archive (http://town.hall.org/Archives/pub/ITA/). The servers are:

- *ClarkNet*: a commercial Internet provider for the Metro Baltimore-Washington DC area;
- *EPA*: the EPA WWW server located at Research Triangle Park, NC;
- *NASA*: the NASA Kennedy Space Center WWW server in Florida;
- *SASK*: the Web server at the University of Saskatchewan, Saskatoon, Canada;
- *SDSC*: the WWW server for the San Diego Supercomputer Center.

Since replaying the traces through the prototypes is quite time-consuming. we used only part of the available traces for ClarkNet, NASA and SASK. A summary of the traces appears in table 2.

We did not use the trace files as is, but applied several preprocessing steps. Below we describe each of the preprocessing steps and discuss their effect on the results.

First, we replay only the requests that are cachable and have a status code of either 200 or 304. In other words, POST requests, requests that have responses with status code 302 (temporarily moved), 401 (unauthorized), 403 (forbidden), etc. are filtered out. The total number of trace records and the number of requests we replay are listed in Table 2. The effect of this filtering is that it magnifies the performance differences between the consistency approaches, because the consistency protocols have no effect on such requests, and all incur the same overhead for them.

Second, we assume that file modifications do not change file sizes. Since a small number of documents are modified during the trace period, the trace may record different sizes for the same file. We assign a fixed size to every document, taking an average of its reported sizes in the trace. If a document only appears in the trace with "304" replies (that is, no size information is available), the document size is set to be the average size of the document type in the trace file. The effect of this assumption is that the experiments do not reflect accurately the bandwidth savings from stale hits, but rather reflect only the expected average bandwidth reduction by stale hits.

Third, we assign pseudo IP addresses to requests that come with client host names rather than IP addresses. The only reason we need the IP addresses is to determine which pseudo client emulates which real client (see the next section). Some traces provide the client IP address of each request. For traces that do not provide the client IP address, we assign a pseudo IP address by splitting the host name into a network name and a hostid, and assigning numbers to the network name and hostid based on their first appearances in the trace. This is done to facilitate the setup of the experiments.

Finally, since the traces do not come with complete file modification history, our simulation uses synthetic modification patterns. As explained below, our synthetic modification pattern reflects a bimodal file life time distribution where 90% of the files are changed infrequently, and 10% of the files are changed frequently. We choose the bimodal distribution because it has been observed in the file life time distribution in many real systems [3, 11], and adaptive TTL works particularly well under such distribution.

5.3 Simulation by Trace Replay

We measure the performance of adaptive TTL, polling-every-time, and invalidation by running the traces through the implemented prototypes in an environment emulating the Internet. We pick five workstations, connected with a fast Ethernet (100Mb/s). One of the workstations is designated as the pseudo-server, and the others are pseudo-clients. The workstations are SPARC-20s with 64MB of main memory running Solaris 2.4.1.

Pseudo-Server The pseudo-server emulates the Web server in each of the traces. The workstation runs the NCSA HTTPD 1.5.1 web server and the Harvest server accelerator. On its disk, it also has scaled copies of all the URL documents in the trace. Due to disk space limitations, we scale down the size of each document by a factor of 100, and create a file of the scaled-down size with the same pathname in the HTTP document directory. Our calculation of the bytes of network messages scales back the file transfer sizes by 100 to reflect the actual amount of traffic in

Item	EPA	SDSC	ClarkNet	NASA	SASK
Trace Duration	1 day(9/29/96)	1 day(8/22/95)	10 hours(8/28/95)	1 day(7/1/95)	8 days(7/1-8/95)
Trace Records	47,748	28,338	64,078	64,715	52,963
Requests Replayed	40,658	25,430	61,703	61,823	51,471
Number of Files	3787	1661	6574	1667	2129
Avg. File Size	21 KB	14 KB	13 KB	44 KB	12 KB
File Popularity	1642 (8.2)	1020 (12)	680 (8)	3138 (31)	1155 (14)

Table 2: Summary of the traces used in our experiments. The last row, file popularity, measures how popular the files at different servers are. For each trace, it shows the maximum number of different client sites that requested the same document (the average is show in parathesis). For example, files at NASA are apparently more popular than files at other servers.

the network. Though the scaling reduces the client response time, the effect is the same on all three approaches.

Pseudo-Clients Each pseudo-client generates the requests for approximately one fourth of the real clients in the trace. Each real client has a clientID, which is a 32-byte integer concatenating the four bytes in its IP address. Pseudo-client i handles real clients whose clientID mod 4 is i. A caching proxy (Harvest "cached") runs on each pseudo-client. A separate program reads every record from the trace file, and if the real client in the record is handled by the pseudo-client, generates a corresponding HTTP request and sends it to the proxy process, then waits for the reply. The proxy checks if the request URL is in cache, and if not, forwards the request to the Web server.

Since in reality client sites do not share caches, we simulate separate caches for individual clients. The proxy concatenates the real client's clientID to the URL before putting the document in cache. That is, if client x requests document $url0$, and the proxy fetches $url0$ from the server, the proxy puts the reply as $url0@x$ in its cache. If client y asks from $url0$ next, the request would be a cache miss. However, if client x asks for $url0$ again, the request would be a cache hit. In running the invalidation prototype, the proxy sends the real clientID along with the GET request to the server, so that the accelerator can register the clientID in the site list. The proxy also handles the invalidation messages to individual real clients.

Timing Coordination in Trace Replay To coordinate the replay of timestamped requests, a time coordinator is introduced to run the simulations in lock step for every five minutes in the trace file. The coordinator first broadcasts the current simulated time, then all the pseudo-clients send requests whose timestamps in the trace file fall in the five minute interval after the current simulated time. After a pseudo-client finishes its requests, it sends a reply back to the time coordinator. After collecting all replies from the pseudo-clients, the time coordinator broadcasts a new simulated time which is five minutes after the previous one. The time coordinator also coordinates the modifier process, as described below.

Document Modification Since the traces do not contain enough information on the modification history of the documents, we generate the modifications ourselves. We use a hot/cold pattern, where 10% of the documents change frequently, and the rest of the document have a very long life time. The experiments treat the cold files as if they do not change during the simulated period, which is a reasonable approximation given the relatively short durations of our traces.

In selecting the hot files in the hot/cold pattern, we make sure that documents of varying degree of popularity are equally sampled. We sort the documents by their degrees of popularity (i.e. the number of different sites requesting them), and pick one out of every ten files on the list. This is to make sure that the simulations include the cases of popular files changing frequently, which we feel is the case for many commercial servers.

A modifier process is run on the pseudo-server to generate the file changes. Based on the timing information sent by the coordinator, the modifier chooses a random file from the hot files to modify every N seconds. This modification pattern leads to a geometric life time distribution for the hot files; N is set so that the average life time of the hot files is a particular value (for example, 5 days). For each selected file, the modifier performs a "touch", which updates the last modified time of the file, then a "check-in" of the file, which notifies the accelerator that the file has been modified. After the modifier finishes its work for the five minute interval, it sends a reply back to the time coordinator.

5.4 Results

Tables 3 through 5 show the performance of the consistency approaches during the replay of the traces. Shown in the table are the replays of the EPA trace with average hot file life time of 5 days, the SASK trace with hot file life time of 1.4 days, the ClarkNet trace with hot file life time of 5 days, the NASA trace with hot file life time of 0.7 days, and the SDSC with hot file life time of .25 days and of 2.5 days. The average hot file life times are chosen to sample the life time range from less than a day to over 5 days, which roughly corresponds to the life time range for hot files found in other studies [3, 11].

Data are presented in rows. "Hits" lists the number of client cache hits. "GET Requests" through "Invalidations" list the number of messages of different types. "Reply 200" means the number of replies with status 200 ("document follows"); these are file transfers in response to a GET request or an "if-modified-since." "Reply 304" means the number of replies with status 304 ("file not modified"), in response to an "if-modified-since." "Total Messages" reports the total number of messages. The next three rows list bytes of file transfer messages ("file xfer bytes"), bytes of control messages ("Ctrl msg bytes"), and total bytes of messages ("Message Bytes").

"Stale Hits" reports the number of stale hits in adaptive TTL. The three rows following it list the average, minimum and maximum of client response times. Finally, "Server CPU" and "disk RW/s" are the average CPU utilization and the average number of disk reads and writes per second at the pseudo-server workstation, as reported by iostat (the server load numbers are only meaningful for comparison purposes). Below, we look at each performance metrics in more detail.

Trace	EPA, 40658 requests		
Modification	72 files modified		
Approach	TTL	Polling	Invalidation
Hits	8530	8533	8532
GET Requests	32128	32125	32126
If-Modified-Since	205	8533	0
Reply 200	32128	32136	32126
Reply 304	205	8522	0
Invalidations	0	0	96
Total Messages	64666	81316	64348
File xfer bytes	238MB	238MB	238MB
Ctrl msg bytes	3.38MB	5.06MB	3.35MB
Message Bytes	241MB	243MB	241MB
Stale Hits	< 11	0	0
Avg. Latency	0.166	0.175	0.158
Min Latency	0.010	0.039	0.010
Max Latency	12.2	12.2	20.1
Server CPU	37.6%	41.6%	38.6%
Server Disk	2.7;3.1	2.6;3.2	2.8;3.3

Trace	SASK, 51471 requests		
Modification	1148 files modified		
Approach	TTL	Polling	Invalidation
Hits	16456	16565	16268
GET Requests	35015	34906	35203
If-Modified-Since	922	16565	0
Reply 200	35388	35689	35203
Reply 304	549	15782	0
Invalidations	0	0	6028
Total Messages	71874	102942	76434
File xfer bytes	185MB	187MB	183MB
Ctrl msg bytes	3.91MB	7.09MB	4.29MB
Messages Bytes	189MB	194MB	187MB
Stale Hits	< 410	0	0
Avg. Latency	0.124	0.138	0.134
Min Latency	0.010	0.039	0.010
Max Latency	32.1	12.2	107.0
Server CPU	26.0%	30.2%	27.6%
Disk RW/s	.37;2.2	.41;2.3	.41;2.5

Table 3: Results for EPA and SASK.

Cache hits Looking at the number of cache hits , we see that the three approaches have fairly similar numbers in most experiments, except for SASK. To understand the difference in SASK, we have to look at the impacts the three approaches have on cache hits.

Harvest's implementation of adaptive TTL replaces expired documents first. Coupled with adaptive TTL's conservative estimate of the file's lifetime, this policy can lead to undesirable effects. For example, if a document has just been modified at the server and then requested by a client, adaptive TTL assigns

Trace	EPA	SASK	ClarkNet	NASA	SDSC(57)	SDSC(576)
Storage	1.0 MB	621 KB	1.6 MB	742 KB	489 KB	474 KB
Avg. SiteList	4	13	26	6.7	8.3	12.0
Max. SiteList	23	666	103	70	81	503
Avg. Invalidation Time	.180	1.15	2.18	.295	.437	.813
Max. Invalidation Time	1.82	106	8.88	7.45	6.78	50.9

Table 6: Invalidation costs. SDSC(57) is the SDSC replay with average file life time of 2.5 days, and SDSC(576) is the replay with the .25-day life time. "Storage" is the amount of memory consumed by site lists at the end of each trace replay simulation. The average and maximum site list lengths are taken among the site lists of files that have been modified. Invalidation time is the time it takes the accelerator to send all invalidation messages for one document modification.

a short life-time to the document. The document is then among the first to be replaced when cache space is needed, despite that it was recently brought into cache and maybe accessed again soon. This effect shows up in the replay of SASK; examination of simulation results shows that there are many documents replaced due to TTL expiration, accounting for the lower hit ratio compared to polling-every-time. Clearly, if adaptive TTL is used as the cache consistency protocol, TTL expiration should not be a factor in determining whether a document should be replaced.

Invalidation has a beneficial effect on cache hits. In invalidation, the proxy cache deletes stale documents upon receiving invalidation messages. This frees up cache space for fresh documents. Thus, invalidation tends to have less *file transfers* than polling-every-time, though its cache hit count appears lower (because our cache hit counts include "hits" on stale documents in the case of polling-every-time).

Number of messages Rows "GET Requests" through "Invalidations" list the number of network messages according to their types. Numbers in the "invalidations" row include both invalidation messages and their acknowledgements from the clients. All messages are small control messages except "Reply 200" messages, which are responses from the server carrying the document (that is, file transfers).

The tables show that the number of control messages incurred by adaptive TTL to maintain cache consistency (that is, if-modified-since messages and "304" replies) is similar to the number of control messages incurred by invalidation. The reason is that invalidation messages are not sent to a client every time the file is modified, but rather when the file is modified after the client has requested a copy. Thus, the number of invalidation messages is proportional to the times when the client needs to get a new copy of the file. Since adaptive TTL also sends if-modified-since requests when it guesses that the client needs to get a new copy of the file, the number of control messages in the two cache consistency protocols are similar.

Polling-every-time has significantly more control messages. This is because every cache hit generates an if-modified-since request, and the majority of such requests result in "304" replies because the frequency of file changes are far less than the frequency of cache hits.

Thus, in terms of total message counts, adaptive TTL and invalidation are similar, while polling-every-time has 10% to 50% more messages.

Network bandwidth consumption The tables show that adaptive TTL and invalidation have very similar total message bytes, given that their file transfer counts and control message counts are very similar. As the stale hit count shows, adaptive TTL is very successful at keeping stale hit ratio low, and thus has file transfers similar to the strong consistency protocols.

The tables also show that polling-every-time has 50-100% more control message bytes, though their contribution to the total message bytes is less than 3%. In general, control messages are much shorter than file transfers. However, control message overhead in polling-every-time increases as the cache hit ratio increases, and can become significant when the cache hit ratio is high.

Stale hits Though our experiments did not count them exactly, the number of stale hits in adaptive TTL is bounded by the difference between the number of useful "if-modified-since" requests in polling-every-time and that in adaptive TTL (a useful "if-modified-since" request is the one that results in a file transfer). Assuming that all cache hits in adaptive TTL are cache hits in polling-every-time, the difference is an upper bound on the number of stale hits (it is not exactly the number of stale hits because cache

Trace	ClarkNet, 61703 requests		
Modification	40 files modified		
Approach	TTL	Polling	Invalidation
Hits	6102	6119	6094
GET Requests	55601	55584	55609
If-Modified-Since	13	6119	0
Reply 200	55601	55611	55609
Reply 304	13	6092	0
Invalidations	0	0	424
Total Messages	111228	123406	111642
File xfer bytes	503MB	503MB	503MB
Ctrl msg bytes	5.95MB	7.20MB	5.99MB
Messages Bytes	509MB	510MB	509MB
Stale Hits	< 27	0	0
Avg. Latency	0.204	0.210	0.202
Min Latency	0.010	0.039	0.010
Max Latency	15.0	13.8	12.2
Server CPU	38.3%	40.4%	38.1%
Disk RW/s	3.6;3.4	3.8;3.5	3.8;3.5

Trace	NASA, 61823 requests		
Modification	144 files modified		
Approach	TTL	Polling	Invalidation
Hits	9740	9758	9754
GET Requests	52087	52069	52073
If-Modified-Since	113	9758	0
Reply 200	52098	52110	52073
Reply 304	102	9717	0
Invalidations	0	0	466
Total Messages	104400	123654	104612
File xfer bytes	1.31GB	1.31GB	1.31GB
Ctrl msg bytes	5.96MB	8.01MB	5.98MB
Messages Bytes	1.32GB	1.32GB	1.32GB
Stale Hits	< 30	0	0
Avg. Latency	.188	.195	.184
Min Latency	0.010	0.039	0.010
Max Latency	13.9	12.2	20.4
Server CPU	32.6%	36.1%	34.4%
Disk RW/s	.25;2.5	.24;2.6	.25;2.8

Table 4: Results for ClarkNet and NASA.

Trace	SDSC, 25430 requests		
Modification	57 files modified		
Approach	TTL	Polling	Invalidation
Hits	4907	4907	4905
GET Requests	20523	20523	20525
If-Modified-Since	239	4907	0
Reply 200	20535	20549	20525
Reply 304	227	4881	0
Invalidations	0	0	248
Total Messages	41524	50860	41298
File xfer bytes	263MB	263MB	263MB
Ctrl msg bytes	2.39MB	3.38MB	2.36MB
Messages Bytes	265MB	266MB	265MB
Stale Hits	< 14	0	0
Avg. Latency	0.160	0.173	0.165
Min Latency	0.010	0.038	0.010
Max Latency	12.2	12.2	12.2
Server CPU	34.1%	35.6%	32.7%
Disk RW/s	.94;2.3	1.4;2.0	1.0;2.2

Trace	SDSC, 25430 requests		
Modification	576 files modified		
Approach	TTL	Polling	Invalidation
Hits	4904	4907	4847
GET Requests	20526	20523	20583
If-Modified-Since	572	4907	0
Reply 200	20595	20614	20583
Reply 304	503	4816	0
Invalidations	0	0	2190
Total Messages	42196	50860	43356
File xfer bytes	264MB	264MB	264MB
Ctrl msg bytes	2.47MB	3.38MB	2.57MB
Messages Bytes	266MB	267MB	266MB
Stale Hits	< 22	0	0
Avg. Latency	0.172	0.177	0.186
Min Latency	0.010	0.038	0.010
Max Latency	8.7	8.6	51.6
Server CPU	33.6%	36.7%	34.7%
Disk RW/s	.92;2.5	.72;2.2	.65;2.3

Table 5: Results for SDSC with two average file life times.

hits under polling-every-time maybe cache misses in adaptive TTL, because of adaptive TTL's eviction of TTL-expired documents). This upper bound on stale hits is listed in the tables.

The results show clearly that adaptive TTL is very successful at keeping stale hit ratio low, most of the time under .1% and always less than 1%. In other words, adaptive TTL does not save much network traffic compared to invalidation, but it also keeps the stale hit ratio very low. Invalidation still has the advantage of providing the guarantee that if the propogration time for invalidation messages is bounded by t seconds, clients never see documents more than t seconds stale. Because adaptive TTL adjusts the time-to-live period according to the age of the file copy, it is difficult to provide such guarantee.

Client response times The results show that contacting the server at every cache hit costs polling-every-time a high minimum latency and a higher average latency. Invalidation's average latency is similar to that of adaptive TTL, except when the number of invalidation messages is very high.

The numbers show that invalidation has a significantly larger worst-case latency, that is, a request from the browser can be stalled for a long time. Comparing the data with those in Table 6 shows that this is mainly due to the fact that, in our current implementation, the accelerator does not accept new requests until all invalidation messages for a document have been sent via TCP. A more fine-tuned implementation would have a separate process sending the invalidation messages, thus avoiding the maximum latency problem. On the other hand, the numbers do show that sending invalidation messages via TCP takes time, and the invalidation protocol needs to either limit the number of invalidation messages for each document (see Section 6), or use reliable multicast schemes.

Since the experiments are done in the local area network, how would the relative comparison of the response times change in the real Internet? We can estimate the trend by looking at the number of client-server interactions in the three approaches. Polling-every-time incurs a server interaction upon every request, adaptive TTL incurs fewer "if-modified-since" interactions, and invalidation incurs "invalidate" interactions. Thus, polling-every-time will have a much worse average response time in the Internet. However, "invalidate" interactions do not directly impact client response times. The proxy's action upon receiving an invalidation message is simple: evicting the stale document. As long as sending invalidations is decoupled from handling regular HTTP requests at the server site, the only impact sending invalidations has on client response times is that it increases server CPU load and consumes kernel resources for networking (such as socket descriptors), and this may in turn increase response times when the server is overloaded. However, the number of "if-modified-since" requests incurred by adaptive TTL is similar to the number of invalidation messages in invalidation, and the "if-modified-since" requests burden the server in similar fashion. Thus, we believe that in the real Internet, invalidation would still have similar client response times as adaptive TTL.

Server loads The last two rows in tables 3 to 5 show the average server CPU utilization and disk read/write per second during the trace replay. Looking at the numbers, we see that polling-every-time generally has a higher server CPU utilization, especially when the proxy cache hit ratio is high. This reflects the CPU cost of handling "if-modified-since" requests at the server. Invalidation tends to have a slightly higher CPU utilization than adaptive TTL, but the differences are mostly within 3%. The disk loads are similar since all three approaches log incoming requests.

Table 6 shows the invalidation costs. The numbers show that the storage consumed by site lists is actually quite small (on the order of 20 to 30 bytes per request), and the main concern is the time for sending invalidation messages. Comparing the two runs of SDSC, we see that when more files are modified, the chance that a file with a very long site list is modified increases, and thus the maximum and average invalidation times increase.

How would the results change in the Internet? Clearly, the delay for sending invalidations using TCP is longer, which means the invalidation approach occupies longer memory resources used for network protocol handling and incurs more CPU overhead. However, the delays for receiving "if-modified-since" requests and sending "304" replies are also longer, with the same effect on the server load of adaptive TTL and polling-every-time. Thus, we expect the relative comparison of CPU and disk utilization to stay mostly unchanged.

Summary To summarize, our experiments show:

- *Adaptive TTL works very well at keeping stale hit ratio low, but strong consistency protocols such as invalidation do not cost more in comparison.* Whether it is in terms of cache hits, network traffic, response time, or server loads, invalidation performs quite similarly to adaptive TTL, while providing strong cache consistency.

- *Invalidation is a preferred method for maintaining strong consistency than polling-every-time.* Except in the extreme case of file lifetimes on the order of minutes, cache hits occur much more often than file modifications. Thus, invalidation incurs much fewer network transactions than polling-every-time. It also improves cache utilization by deleting stale copies, and incurs less server CPU overhead.

- *Sending a large number of invalidation messages via TCP can lead to long delays.* If not implemented carefully, the delay can cause a client request to stall for a long time.

The results suggest that invalidation should be used to maintain strong cache consistency in the Web. However, a simple invalidation scheme has a scalability problem, as evidenced by the long invalidation time. In section 6, we introduce a two-tier lease-augmented invalidation scheme that addresses the problem.

5.5 Limitations in the Experiments

There are many limitations in our experiments. First, the trace replay is performed in a local area network instead of the Internet. The effect of this limitation on client response times and server loads has been discussed above. The number of cache hits, network messages, and stale hits are not affected by the trace replay environments. Second, we use server traces in the experiments, instead of client or proxy traces. Since requests seen by the server are partially filtered by client caches, server traces show a lower hit ratio at the client sites. This means that, in reality, polling-every-time performs even worse than the results shown here. However, we expect the relative comparison between invalidation and adaptive TTL to stay the same, since the two approaches are less affected by client hit ratios. Finally, the traces are for short periods and relatively old. However, we believe the experiments still give us valid insights on the comparison of the three consistency approaches.

6 Scalability Issues

There are a number of scalability concerns in the invalidation approach, including the storage needed to keep track of client sites for each document, the CPU overhead to search and update the site lists, and the time to send invalidation messages. Results in Section 5.4 show that for short periods (from 10 hours to 8 days), memory consumption and CPU overhead do not pose serious problems to invalidation. However, site lists grow linearly with the number of requests seen by the server, which in turn increases linearly with time. Thus, unless some measure is taken, the size of site lists can grow to be unmanageable.

The solution is to augment the simple invalidation scheme with *leases*. Every document shipped from the server to a client carries a lease. The server promises to notify the client via invalidation if the document changes before the lease expires. The client promises to send a "if-modified-since" request to the server when the lease expires, to validate the freshness of its copy. Thus, the server only needs to remember clients whose leases have not expired. For example, if the lease is three days, the total size of site lists is bounded by the total number of requests seen by the server for the last three days. The performance of the invalidation approach reported in the last section can be interpreted as the performance of a lease-augmented invalidation scheme with the lease equal to the duration of each trace.

A second optimization further reduces the size of site lists and the number of invalidation messages. From the server's point of view, the benefit of site lists and invalidation messages is reduced "if-modified-since" request traffic. Thus, the server would like to remember only those clients that are truly interested in its documents and view the documents multiple times. To achieve this goal, the server assigns a very short lease (possibly zero) to regular "GET" requests, and assigns the regular lease to "if-modified-since" requests. The scheme trades extra "if-modified-since" requests for filtered site lists; only those clients that ask to view the document for the second time are remembered by the server. We call this scheme "two-tier lease-augmented invalidation."

The two-tier approach is quite effective at reducing site list sizes. For example, at the end of the 8-day SASK trace, the site lists have only 5021 entries, compared to 29106 entries under the simple invalidation scheme. The maximum length of the site list of a document is reduced from 1155 entries to 521 entries. The reduction is achieved with 5021 extra "if-modified-since" requests, much less than the 16565 requests generated by polling-every-time (Table 3). Other traces have similar results.

Variable Leases The two-tier approach is only a special case of a general technique in which the server varies the lease attached with each reply to suit its own needs. Essentially, a lease allows the server to trade the frequency of "if-modified-since" requests with the overhead of remembering the client and sending invalidation messages. The frequency of "if-modified-since" requests from a client cache depends on the frequency of cache hits to the document at the client. Thus, there are three factors that affect the choice of an optimal lease for each reply: *the expected frequency of cache hits at the client site, the expected life-time of the documents (i.e., time till next modification), and the current length of the site list of the document.*

Various adaptations can be designed based on each of the factors. For example, the above two-tier approach changes the leases based on the first factor. Another similar approach is for the server to give out replies with non-zero leases only to Web proxies (which act as gateways between many users and the server). Considering the second factor, the server can monitor the modification history of the document, shorten the leases when the document becomes active (i.e., modified often), and increase the leases when the document becomes dormant. Considering the third factor, the server can reduce the leases proportionally as its available storage runs out, and increase the leases as more storage becomes available. The server can also revoke a lease given to a client (thereby deleting all information it keeps about the client) by sending the client an invalidation of the document. Based on the protocol, the client will contact the server upon next access.

Clearly, there is a rich design space for variable-lease algorithms. We have only characterized it based on the factors that determine the optimal leases. A full investigation requires a large collection of Web access traces unfiltered by browser caches and proxy caches. We are still in the process of collecting the traces, and plan to study the best combination of the above adaptation techniques through simulation and implementation.

Finally, proxy caches and browser caches should use different consistency protocols. Browser processes are started and stopped by the user. WWW proxy processes are expected to be always up and running. Since the invalidation approach requires client processes to be able to receive invalidation messages at any time, it is more appropriate for maintaining strong cache consistency between proxy caches and Web servers. If the browser wants to keep its cached copies strongly consistent, it should communicate with its proxy. That is, the proxy acts as a receiver for invalidation messages to contents in browser caches, and retains the messages to be queried by browsers. The proxy retains all invalidation messages received in the last N days, and the browser is asked to check the freshness of all its cached documents at least once every N days. Since many companies and Internet Service Providers already use or are considering to use Web proxies, the majority of browser processes will be behind Web proxies. For browsers that directly connect to the Web server, polling-every-time can be used to maintain strong cache consistency if desired.

7 Conclusion and Future Work

We have analyzed and compared the performance of three consistency maintanence approaches: adaptive TTL, polling-every-time, and invalidation. Our results show that invalidation performs similarly to adaptive TTL (within 3% most of time), in terms of network traffic, average client response times, and server CPU loads. Polling-every-time, on the other hand, leads to significantly more network messages and higher response times than adaptive TTL. Thus, it is feasible to maintain strong cache consistency for the Web, and invalidation is the right approach for it. We have also described a two-tier lease-augmented invalidation scheme that addresses the scalability issues in invalidation.

There are many limitations in this study. In addition to what we listed before, we have not considered many other concerns that may dictate the consistency approach taken by a Web server. Documents are not all equally important to the user. Not all Web servers want to keep its copies consistent across the Internet. What we have shown is that, for those that do care, invalidation is feasible, and it is a better approach than TTL or client polling. For those commercial Web sites that want to control the accesses to its contents, invalidation should be merged with other hit-metering protocols [15] to provide both the benefits of caching and the availability of access control. Finally, for clients that are behind a firewall, invalidation should operate between the Web server and the proxy running at the firewall machine (many companies run Web proxies for the exact reason of handling Web traffic through the firewall); the proxy then relays the invalidation messages to the clients inside the firewall.

Many future work remains. We plan to improve the implementation of the invalidation protocol, looking into UDP-based or multi-casting schemes that reduce the messaging overhead. We also plan to implement the two-tier lease-augmented invalidation scheme, and improve failure handling in the consistency approach. We hope to provide a highly-efficient and high-performance implementation of invalidation protocols that can be widely deployed in the Internet.

Acknowledgments

C. Y. Chan and Rahul Kapoor participated in the implementation of the initial prototype of invalidation in Harvest. The study benefited much from the discussions with Jeff Mogul, Stefanos Kaxiras, James Goodman, David Wood and Mark Hill. We are indebted to the Wisconsin Wind Tunnel group for letting us to run the simulations on the Wisconsin Clusters of Workstations. Gideon Glass, Doug Burger, Dan Boneh and John Edwards provided helpful feedbacks on the early drafts of the paper. Finally, we would like to thank the anonymous referees for their comments.

References

[1] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*, December 1995.

[2] Mary Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–211, October 1991.

[3] A. Bestavros. Demand-based resource allocation to reduce traffic and balance load in distributed information systems. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*, October 1995.

[4] Matthew A. Blaze. *Caching in Large-Scale Distributed File Systems*. PhD thesis, Princeton University, January 1993.

[5] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[6] V. Cate. Alex - a global file system. In *Proceedings of the 1992 USENIX File System Workshop*, pages 1–12, May 1992.

[7] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *Proceedings of the 1996 USENIX Technical Conference, San Diego, CA*, January 1996.

[8] P. B. Danzig, R. S. Hall, and M. F. Schwartz. A case for caching file objects inside internetworks. In *Proceedings of SIGCOMM '93*, pages 239–248, 1993.

[9] Michael Franklin. *Client Data Caching: A Foundation for High Performance Object Database Systems*. Kluwer Academic Publishers, 1996.

[10] Michael J. Franklin, Michael J. Carey, and Miron Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM Transaction on Database Systems*, 1997.

[11] James Gwertzman and Margo Seltzer. World-wide web cache consistency. In *Proceedings of the 1996 USENIX Technical Conference, San Diego, CA*, January 1996.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[13] John H. Howard, Michael Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, pages 6(1):51–81, February 1988.

[14] Stefanos Kaxiras and James Goodman. Implementation and performance of the GLOW kiloprocessor extensions to sci on the wisconsin wind tunnel. In *Proceedings of the 2nd International Workshop on SCI-Based High-Performance Low-Cost Computing*, March 1995.

[15] Paul Leach and Jeff Mogul. The HTTP hit-metering protocol. *Internet draft*, November 1996. URL ftp://ieft.org/internet-draft/draft-mogul-http-hit-metering-00.txt.

[16] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[17] A. Luotonen, H. Frystyk, and T. Berners-Lee. CERN HTTPD public domain full-featured hypertext/proxy server

with caching. Technical report, Available from http://www.w3.org/hypertext/WWW/ Daemon/Status.html, 1994.

[18] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite file system. *ACM Transactions on Computer Systems*, pages 6(1):134–154, February 1988.

[19] R. Sandberg, D. Boldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Summer Usenix Conference Proceedings*, pages 119–130, June 1985.

[20] D. Wessels. Intelligent caching for the world-wide web objects. In *Proceedings of INET-95*, 1995.

[21] K. Worrell. Invalidation in large scale network object caches. Technical report, Master's Thesis, University of Colorado, Boulder, 1994, 1994.

Author Information

Pei Cao received her Ph.D. degree from Princeton University in 1995 and is currently Assistant Professor of Computer Science at University of Wisconsin-Madison. Her research interests are in operating systems, distributed systems including World-Wide Web, and computer architecture. She is currently a member of the IEEE Computer Society Task Force on Internetworking.

Chengjie Liu received his MS degree in Computer Science from University of Wisconsin-Madison in 1997. He is currently a technical staff member at Sun Microsystems, Inc. His research interests include distributed systems and performance measurements.