Engineering Web Cache Consistency

JIAN YIN, LORENZO ALVISI, and MIKE DAHLIN
University of Texas at Austin
and
ARUN IYENGAR
IBM T. J. Watson Research Center

Server-driven consistency protocols can reduce read latency and improve data freshness for a given network and server overhead, compared to the traditional consistency protocols that rely on client polling. Server-driven consistency protocols appear particularly attractive for large-scale dynamic Web workloads because dynamically generated data can change rapidly and unpredictably. However, there have been few reports on engineering server-driven consistency for such workloads. This article reports our experience in engineering server-driven consistency for a sporting and event Web site hosted by IBM, one of the most popular sites on the Internet for the duration of the event. We also examine an e-commerce site for a national retail store. Our study focuses on scalability and cachability of dynamic content. To assess scalability, we measure both the amount of state that a server needs to maintain to ensure consistency and the bursts of load in sending out invalidation messages when a popular object is modified. We find that server-driven protocols can cap the size of the server's state to a given amount without significant performance costs, and can smooth the bursts of load with minimal impact on the consistency guarantees. To improve performance, we systematically investigate several design issues for which prior research has suggested widely different solutions, including whether servers should send invalidations to idle clients. Finally, we quantify the performance impact of caching dynamic data with server-driven consistency protocols and the benefits of server-driven consistency protocols for large-scale dynamic Web services. We find that (i) caching dynamically generated data can increase cache hit rates by up to 10%, compared to the systems that do not cache dynamically generated data; and (ii) server-driven consistency protocols can increase cache hit rates by a factor of 1.5-3 for large-scale dynamic Web services, compared to client polling protocols. We have implemented a prototype of a server-driven consistency protocol based on our findings by augmenting the popular Squid cache.

Categories and Subject Descriptors: H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services; Data sharing*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems; Performance evaluation* (*efficiency and effectiveness*); C.4 [**Computer Systems Organization**]: Performance of Systems—*Design studies; Fault tolerance; Performance attributes; Reliability, availability, and serviceability*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/server*; D.4.3 [**Operating Systems**]:

Authors' addresses: J. Yin, L. Alvisi, M. Dahlin, Dept. of Computer Science, Univ. of Texas at Austin, Taylor Hall-2.124, Austin, TX 78712-1188; email: yin,lorenzo,dahlin@cs.utexas.edu; A. Iyengar, IBM T. J. Watson Research Center, POB 704, Yorktown Heights, NY 10598; email: aruni@watson.ibm.com.

File Systems Management—*Distributed file systems*; D.4.8 [**Operating Systems**]: Performance—*Simulation*

General Terms: Design Performance, Reliability

Additional Key Words and Phrases: Cache coherence, cache consistency, scalability, lease, dynamic content, volume

1. INTRODUCTION

Although Web caching and prefetching have the potential to reduce read latency significantly, the inefficiency of the cache consistency protocols in the current version of HTTP prevents this potential from being fully realized. HTTP uses *client polling* in which clients query servers to determine if cached objects are up to date. Thus, clients may need to poll servers before returning cached objects to users even when these objects are valid. For example, in the workloads that we examine, more than 20% of requests to the servers can be client polls to revalidate unmodified cached objects. Thus, client polling not only increases server and network load, but also significantly increases read latency.

Many of the drawbacks of the traditional client polling protocols can be addressed with *server-driven* consistency protocols. In server-driven protocols, servers inform clients of updates [Howard et al. 1988]. Thus, clients can return cached objects without contacting the server if these objects have not been invalidated. A range of server-driven consistency protocols have been proposed and evaluated in both unicast and multicast environments using client Web traces [Yin et al. 1998], synthetic workloads [Yin et al. 1999a], single Web pages [Yu et al. 1999], and proxy workloads [Li and Cheriton 1999].

Server-driven consistency appears particularly attractive for large-scale Web sites containing significant quantities of dynamically generated and frequently changing data. There are two reasons for this. First, in these workloads, data changes often and at unpredictable times. Therefore, client polling is likely to result in obsolete data unless polling is done frequently—in which case the overhead becomes prohibitive. Second, the ability to cache dynamically generated data is critical for improving server performance. Requests for dynamic data can require orders of magnitude more time than requests for static data [Iyengar and Challenger 1997] and can consume most of the CPU cycles at a Web site, even if they only make up a small percentage of the total requests.

However, to deploy server-driven consistency protocols for large-scale dynamic Web services, several design issues critical to scalability and performance must be examined. This article provides one of the first studies of server-driven consistency for Web sites serving large amounts of dynamically generated data. We examine two workloads. Our first workload is generated by a major sporting and event Web site hosted by IBM,[1] which in 1998 served 56.8 million requests on the peak day, 12% of which were dynamically generated data [Iyengar et al. 1999]. Our second workload is taken from the e-commerce site of a national retail store. This site generated more than one million hits a day, 6.4% of which were to dynamically generated data.

[1]The 1998 Olympic Games Web site.

The first issue we address is scalability. In server-driven consistency, scalability can be limited by a number of factors:

—As the number of clients increases, the amount of memory required to keep track of the content of clients' caches may become large.

—Servers may experience bursts of load when they need to send invalidation messages to a large number of clients as a result of a write.

Previous efforts to improve the scalability of server-driven consistency have primarily focused on using multicast and hierarchies to flood invalidation messages [Li and Cheriton 1999; Yin et al. 1999a; Yu et al. 1999]. Although these approaches are effective, relying on them would pose a barrier to deployment. Our primary focus is on engineering techniques to improve scalability that are independent of network layers. These techniques make it feasible to deploy server-driven consistency for services as large as the IBM Sporting and Event Web site on today's infrastructure, and will continue to improve scalability in the future as multicast and hierarchies become widespread.

We show that the maximum amount of state kept by the server to enforce consistency can be limited without incurring a significant performance cost. Furthermore, we show that although server-driven consistency can significantly increase peak server load, it is possible to smooth out this burstiness without significantly increasing the time during which clients may access stale data from their caches.

The second issue we address is assessing the performance implications of the different design decisions made by previous studies in server-driven consistency.

Different studies have made widely different decisions in terms of the length of time during which clients and servers should stay *synchronized*, i.e., the length of time during which servers are required to notify clients whenever an object in the clients' cache becomes stale. Some studies argue that servers should stop notifying idle clients to reduce network, client, and server load [Yin et al. 1998], while others suggest that clients should stay synchronized with servers for days at a time to reduce latency and to amortize the cost of joining a multicast channel when multicast-based systems are used [Li and Cheriton 1999; Yu et al. 1999].

Using a framework that is applicable in both unicast and multicast environments, we quantify the trade-off between the low network, server, and client overhead of maintaining synchronization for short periods of time on the one hand, and the low read latency of maintaining synchronization for long periods of time on the other hand. We find that for both of our workloads, there is little performance cost in guaranteeing that clients will be notified of stale data within a few hundred seconds. We also find that there is little benefit to hit rate in keeping servers and clients synchronized for more that a few thousand seconds.

Previous studies also propose significantly different *resynchronization protocols* to resynchronize servers' and clients' consistency state after recovering from disconnections, which may be caused by choice, by a machine crash, or by a temporary network partition. Proposals include invalidating all objects in

clients' caches [Liu and Cao 1997; Yu et al. 1999], replaying "delayed invalidations" upon resynchronization [Yin et al. 1998], bulk revalidation of cache contents [Baker 1994], and combinations of these techniques. This study systematically compares these alternatives in the context of large-scale services. We find that for desynchronizations that last less than 1000 seconds, delayed invalidations result in significant performance advantages compared to the other alternatives.

The final issue that we address is quantifying the performance implications of caching *dynamically generated data*, data generated by server programs executed in response to HTTP requests, as opposed to *static data*, data contained in static documents in a Web server. Although the high frequency and unpredictability of updates make dynamically generated data virtually uncachable with traditional client-polling consistency, server-driven consistency may allow clients to cache dynamically generated data effectively. Through a simulation study that uses both server-side access traces and update logs, we demonstrate that server-driven consistency allows clients to cache dynamic content with nearly the same effectiveness as static content.

We have implemented the lessons learned from the simulations in a prototype that runs on top of the popular Squid cache architecture (http://www.squid-cache.org). Our implementation addresses the consistency requirements of large-scale dynamic sites by extending basic server-driven consistency to provide consistent updates of multiple related objects and to guarantee fast resynchronization and atomic updates. Preliminary evaluation of the prototype shows that it introduces only a modest overhead.

The rest of the article is organized as follows. Section 2 reviews previous work on WAN consistency, on which this study is built. Section 3 evaluates various scalability and performance issues of server-driven consistency for large-scale dynamic services. Section 4 presents an implementation of server-driven consistency based on the lessons that we learned from our simulation study. Section 5 and Section 6 discuss related work and summarize the contributions of this study.

2. BACKGROUND

The guarantees provided by a consistency protocol can be characterized using two parameters: worst-case staleness and average staleness. We use $\Delta(t)$ consistency to bound worst-case staleness. $\Delta(t)$ consistency ensures that data returned by a read is never stale by more than t units of time. Specifically, suppose the most recent update to an object O happened at time T. To satisfy $\Delta(t)$ consistency, any read after $T+t$ must return the new version of object O. Average staleness is instead expressed in terms of two factors: the fraction of reads that return stale data and the average amount of time for which the returned data has been obsolete. For example, a live news site may want to guarantee that in the worst case it will not supply its clients with any content that has been obsolete for more than five minutes, while attempting to provide good average staleness by delivering most updates within a few seconds.

The optimal Web consistency protocol is *precise expiration*. In precise expiration, servers set the expiration time of each object to be the next modification time. When a cache loads a Web object, the corresponding expiration time is also passed to the cache via the HTTP header "Expires." Because in precise expiration Web objects expire when and only when they are updated, caches only contact servers to load new versions of Web objects. Unfortunately, servers generally cannot predict future modification times, and thus precise expiration is unrealistic. Nevertheless, the performance of precise expiration provides a benchmark for evaluating other consistency protocols.

Consistency protocols other than precise expiration use two mechanisms to meet consistency guarantees. First, worst-case guarantees are provided using some variation of *leases* [Gray and Cheriton 1989], which place an upper bound on how long a client can operate on cached data without communicating with the server. Second, some systems decouple average staleness from the leases' worst-case guarantees by also providing *callbacks* [Howard et al. 1988; Nelson et al. 1988] which allow servers to send invalidation messages to clients when data is modified.

For example, HTTP's traditional client polling associates a *time to live* (TTL) or an expiration time with each cached object [Mogul 1996]. This TTL can be regarded as a per-object lease to read the object; in particular, it places an upper bound on the time that each object may be cached before the client revalidates the cached version. To revalidate an object whose expiration time has passed, a client sends a `Get-if-modified-since` request to the server, and the server replies with "304 not modified" if the cached version is still valid or with "200 OK" and the new version if the object has changed.

The HTTP polling protocol has several limitations. First, because TTL determines both worst-case staleness and average staleness, they can not be decoupled. Second, each object is associated with an individual TTL. After a set of TTLs expire, each object has to be revalidated individually with the server to renew its TTL, thereby increasing server load and read latency.

As a result, several researchers [Li and Cheriton 1999; Liu and Cao 1997; Yin et al. 1998; Yin et al. 1999a; Yu et al. 1999] have proposed server-driven consistency protocols. These protocols can be understood within the general framework of *volume leases* [Yin et al. 1999b]. Volume leases decouple average staleness from worst-case staleness by maintaining leases on objects as well as volumes, which are collections of related objects. Whenever a client caches an object, it requests a lease on the object. The server registers callbacks on object leases and revokes them when the corresponding objects are updated. Typically, the length of an object lease is chosen so that the lease will be valid for as long as the client is interested in the corresponding object (unless, of course, the object is updated by the server). Thus, object leases allow servers to inform clients of updates as soon as possible. Worst-case staleness is enforced through volume leases, which abstract synchronization between clients and servers. A server grants a client a volume lease only if all previous invalidation messages have been received by the client, and a client is allowed to read an object only if it holds both the object lease and the corresponding volume lease. Thus, a volume lease protocol enforces a staleness bound that is equal to the volume lease

length. In addition to decoupling average staleness from worst-case staleness, volume lease protocols can reduce server load and read latency by amortizing volume lease renewal overheads over many objects in a volume. Moreover, understanding the relation between consistency semantics and volume lease mechanisms reveals an opportunity for reducing the number of messages. Since a client with an expired volume lease cannot read any objects in the volume, we can delay sending invalidation messages for these objects until the client renews its volume lease without compromising consistency. This optimization allows the invalidation messages to piggyback on other messages and thus reduces the number of messages. We call this optimization *delayed invalidations*.

Within this general framework, a wide range of implementations can be related. Volume leases are either explicitly renewed [Li and Cheriton 1999; Yin et al. 1998; Yin et al. 1999a] or implicitly renewed via heartbeats [Yu et al. 1999]. Moreover, the implementation details of these protocols differ considerably. Yin et al. [1998] assume a unicast network infrastructure with an optional hierarchy of consistency servers [Yin et al. 1999b] and specify explicit volume lease renewal messages by clients. Li and Cheriton [1999] assume a per-server reliable multicast channel for both invalidation and heartbeat messages. Yu et al. [1999] assume an unreliable multicast channel, but bundle invalidation messages with heartbeat messages and thus tie average staleness to the system's worst-case guarantees. The implications of these design choices are evaluated in the next section.

A key problem in caching dynamically generated data is determining how changes to underlying data affects cached objects. For example, dynamic Web pages are often constructed from databases, and the correspondence between the databases and Web pages is not straightforward. Data update propagation (DUP) [Challenger et al. 1999] uses object dependence graphs to maintain precise correspondences between cached objects and underlying data. DUP thereby allows servers to identify the exact set of dynamically generated objects to be invalidated in response to an update of underlying data. Use of data update propagation at IBM sporting and event Web sites has resulted in server side hit rates of close to 100%, compared to about 80% for an earlier version that didn't use data update propagation.

3. EVALUATION

In this section we use trace-based simulation to evaluate the benefit and feasibility of server-driven cache consistency for large-scale Web sites. Section 3.1 describes our workloads and simulator. Section 3.2 compares read latency, overhead, and consistency resulting from server-driven protocols, traditional client-polling protocols, and a theoretically optimal consistency protocol. Section 3.3 considers two optimizations over traditional client-polling protocols and compares them against server-driven consistency protocols. Section 3.4 examines how to use prefetch to further reduce read latency in server-driven consistency protocols. Section 3.5 examines the scalability of server-driven consistency protocols. Section 3.6 examines performance of various mechanisms of server-driven consistency to recover from server, client, and network failures.

3.1 Methodology

In this section, we describe the workloads and simulator used in our simulation study.

3.1.1 *Workload.* We use two workloads with different characteristics to conduct our study; one is from an online news site, the other an e-commerce site. The first workload is taken from a major IBM sporting and event Web site. This site contained about 60,000 objects, and over 60% of the objects were dynamically generated [Iyengar et al. 1999]. The peak request rate for the IBM sporting and event site exceeded 56.8 million requests per day. The sporting and event Web service was hosted on four geographically distributed Web clusters; each of them served about one-fourth of all requests. We took the server access log from one of these clusters on February 19th, 1998, and a modification log of dynamically generated objects for our simulation study. The server access log is in the Common Log Format (http://httpd.apache.org/docs/logs.html), recording the IP address, timestamp, request line, status code, and object size for each access. This log contains about 9 million entries. The modification log for dynamically generated objects contains 20,549 entries. Our second workload is taken from an e-commerce Web site for a national retail store. This trace contains the Web access logs from March 3, 2000 to March 9, 2000. It indicates that 177,978 clients accessed 69,608 URLs during the 7-day period. Overall, 9,504,953 requests are logged.

We distinguish two classes of data: *dynamic data*, data generated by server programs upon users' requests, and static data. We classify URIs in our workloads as static or dynamic, based on whether a URI contains the name of a server program. We can further divide static objects into image objects and nonimage objects by examining the suffix of the URIs. In our sporting and event workload, 12% of the requests were made to pages dynamically generated by server programs. In our e-commerce workload, 6.4% of the requests were to dynamically generated objects.

We detect updates in two ways. First, the IBM sporting and event server logged the updates of the dynamically generated data, and the modification log is available for our simulation study. Second, for the static data, the dynamically generated data in our e-commerce workload, and the subset of dynamic data in the IBM workload, we use changes of object sizes to infer updates from our traces of read requests. This method appears reasonably accurate because two versions of a document are unlikely to have the same size; we can partially confirm this assumption in our workload—we have a modification log of a subset of objects in the first trace from DUP [Challenger et al. 1999], and we find that the object size always changes when there is a write for this subset of the workload. We set the time of such inferred writes to immediately precede the read that detects the change; this approximation of the timing of writes does not affect our results for read latency, number of messages, or average staleness. It may slightly overstate server state, since it maximizes delay on reclaiming server state for callbacks. Finally, it may somewhat affect our measurements of bursts of load, though we do not anticipate any systematic biases.

In our study, all the objects from a Web server form a volume. We expect this method of volume construction to be used in practice because it is simple to implement. Moreover, the bigger the volume, the lower the server load and the read latency, since volume lease renewals are amortized over more objects. However, it is generally not beneficial to group objects from several servers into one volume, since these objects can be disconnected from clients independently. Thus, the ideal volume typically comprises all the objects from a server.

As noted in Section 2, object leases should be long enough to span the period of interest of a client in an object. For simplicity, we use infinite-length object leases in our simulations. By allowing servers to discard some callbacks, more carefully tuned object lease lengths can further reduce message load [Yin et al. 1999b] or server callback state [Duvvuri et al. 1999], while only slightly increasing client response time due to the need to renew expired object leases.

We treat all the reads in our server access logs the same, and we do not care whether these reads are from a browser or a proxy cache. We assume consistency proxies deployed in these proxies as described by Yin et al. [1999a] in our simulation study. Consistency proxies act as servers to clients, act as clients to origin servers, and are essential for invalidation messages to traverse firewalls. Thus, consistency proxies that support volume leases interact with the server in the same way as browsers.

We use the client requests in server access logs to drive our simulation because these logs are the only workloads that contain requests from all clients to large-scale popular servers. All client reads to dynamically generated objects are included in server access logs, since dynamically generated objects are not cached in the current Web cache scheme. However, some reads to static objects are not recorded in server access logs because Web access logs are taken while the clients are running some client-polling protocols. There are two potential inaccuracies. First, a request is filtered out by the local cache when the TTL is valid and the cached object is also valid. This read would turn to a cache hit for both server-driven protocols with long volume leases and client-polling protocols with long TTLs.[2] Thus, it affects both server-driven consistency protocols and client-polling protocols. Second, a client request is filtered out by the local cache when the TTL is valid and the cached object is invalidated. This request could result in a cache miss for server-driven protocols, but a cache hit for long TTLs. Since the role of consistency protocols is to fetch the new data when data changes, it is preferable to suffer a cache miss to load valid data than to avoid a load by reading stale data. Moreover, a cache miss would eventually result in client-polling protocols if the client continues to read the object after the TTL expires. Thus, our results are conservative estimates of the benefits of server-driven protocols over client-polling protocols.

3.1.2 *Simulator.* To study the impact of different design decisions on the performance of server-driven consistency, we built a simulator that reads a Web

[2]When the TTL and the volume lease are short, this hidden read is more likely to trigger a TTL refresh than a volume lease renewal because a volume lease renewal is amortized over many objects. Thus, we underestimate the cache miss rates more for client polling than for volume lease in this case.

access log and a modification log and outputs read latency, server load, and network bandwidth consumption. Our simulator simulates both cache state (whether an object is cached) and consistency state (whether TTLs are valid or expired in client-polling protocols and whether volume leases and object leases are valid in server-driven protocols). Given a workload, our simulator processes the Web access log and the modification log in the order of timestamps, updating cache and consistency state and recording consistency, read load, and server load information. To estimate server and network load, we only track the number of messages and total number of bytes in all messages, and we do not simulate network queuing and round-trip delays. Counting the number of messages and bytes provides a reasonable approximation to network load, independent of many detailed network parameters such as packet sizes and congestion conditions.

3.2 Consistency for Large-Scale Dynamic Workloads

Previous studies have examined the performance characteristics of server-driven consistency for client cache workloads [Yin et al. 1998], synthetic workloads [Yin et al. 1999b], single Web pages [Yu et al. 1999], and proxy workloads [Li and Cheriton 1999]. In this section, we examine the performance characteristics of server-driven consistency protocols for large-scale dynamic server workloads. Our goals are (i) to understand the interaction of server-driven consistency with this important class of workloads; and (ii) to provide a baseline for the more detailed evaluations that we provide later in this article.

Our performance evaluation stresses read latency. To put the read latency results in perspective, we also examine the network costs of different protocols in terms of messages transmitted. Read latency is primarily determined by *local miss rate*, the fraction of reads that a client cannot serve locally. There are two conditions under which the system has to contact the server to satisfy a read. First, the requested object is not cached. We call this a *data miss*. Second, even if the requested object is cached locally, the consistency protocol may need to contact the server to determine whether the cached copy is valid. We call this a *consistency miss*. Read latency for consistency misses may be orders of magnitude higher than that for local hits, especially when the network is congested or the server is busy. Thus, to a fine approximation, reduction in miss rates yields proportional reduction in average read latency.

As described in Section 2, the volume lease algorithm has two advantages over traditional client-polling algorithms. First, it reduces the cost of providing a given worst-case staleness by amortizing lease renewals across multiple objects in a volume. In particular, under a standard TTL algorithm, if a client references a set of objects whose TTLs have expired, each reference must go to the server to validate an object. In contrast, under a volume leases algorithm, the first object reference will trigger a volume lease renewal message to the server, which will suffice to re-validate all of the cached objects. Second, volume leases provide the freedom to separate average case staleness from worst-case staleness by allowing servers to notify clients when objects change.

Figures 1 through 6 illustrate the impact of different consistency parameters for the IBM sporting and event workload and the e-commerce workload.

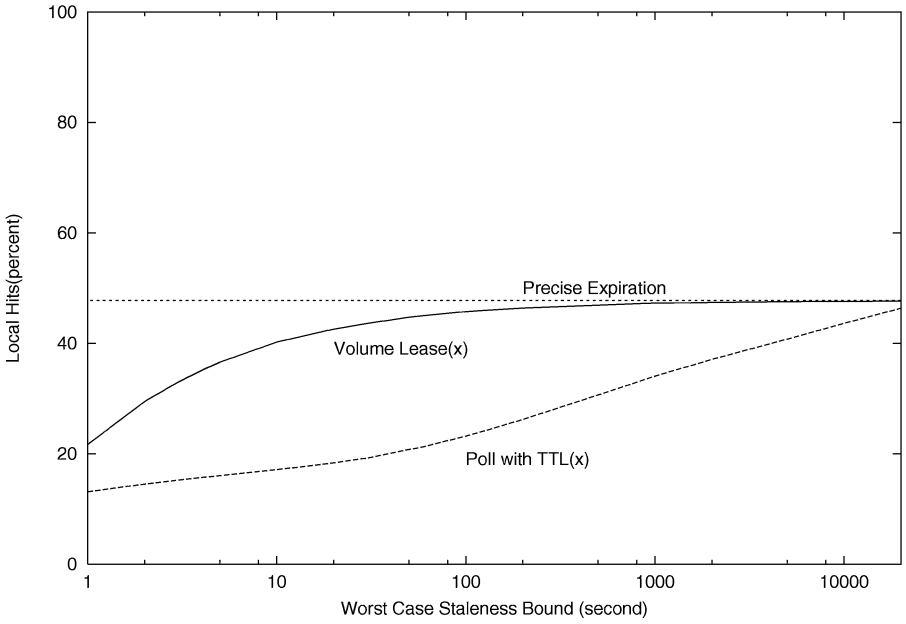

Fig. 1. Local hit rates vs. worst-case staleness bound of volume lease and TTL for the IBM workload. *Volume Lease(x)* represents the volume lease protocol with the volume lease length equal to x seconds; *Poll with TTL(x)* represents a client-polling protocol with the TTL equal to x seconds. Note that in the common case, volume lease caches are invalidated within a few seconds of an update, independent of worst-case staleness bounds.

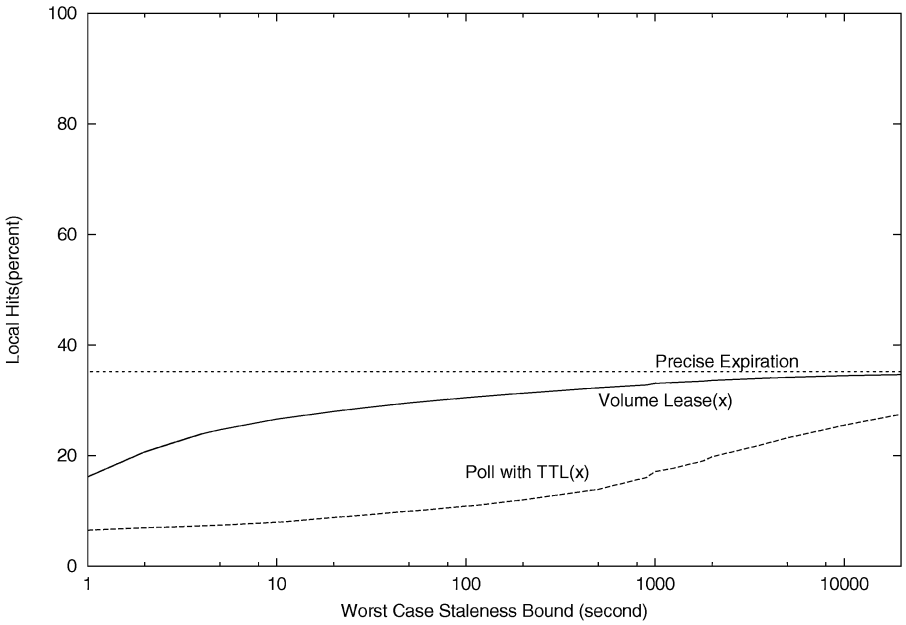

Fig. 2. Local hit rates vs. worst-case staleness bound of volume lease and TTL for the e-commerce workload.

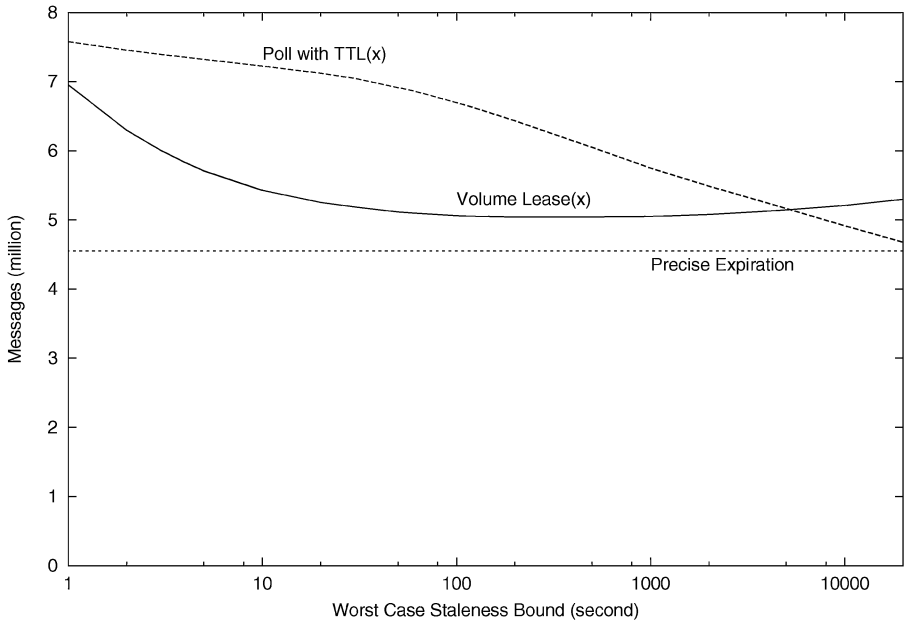

Fig. 3. Number of messages vs. staleness bound of Volume Lease and TTL for the IBM workload.

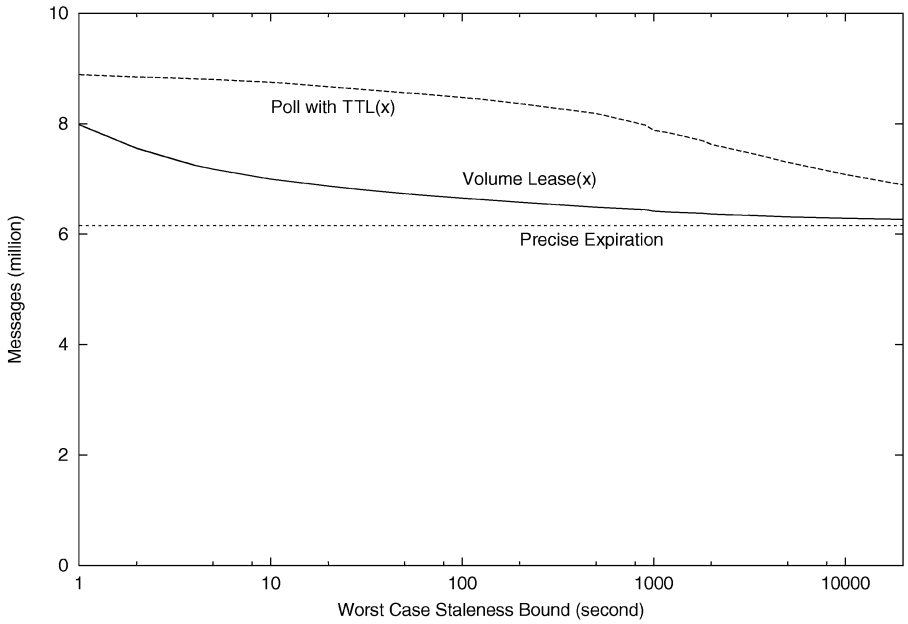

Fig. 4. Number of messages vs. staleness bound of Volume Lease and TTL for the e-commerce workload.

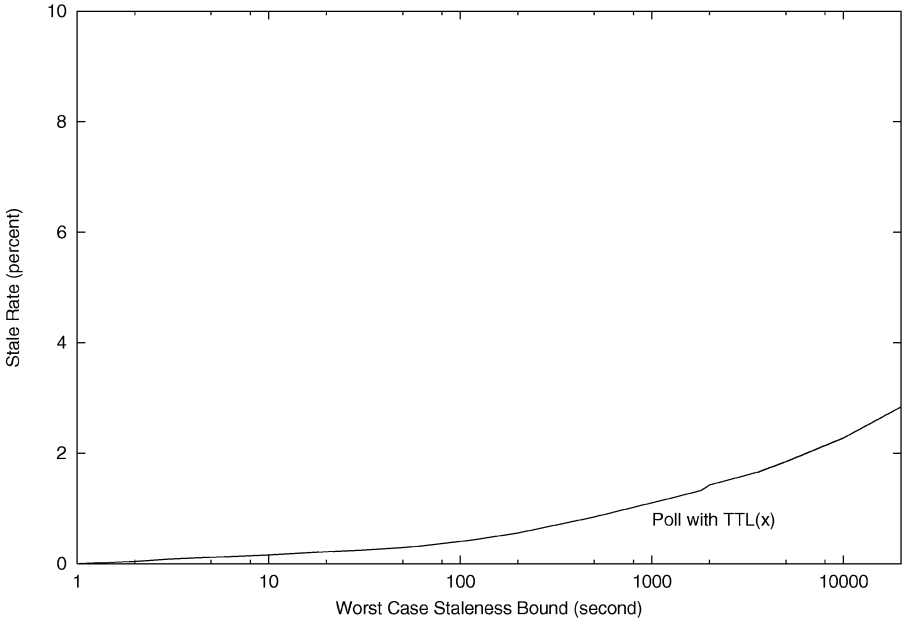

Fig. 5. Stale rate vs. staleness bound of TTL for the IBM workload.

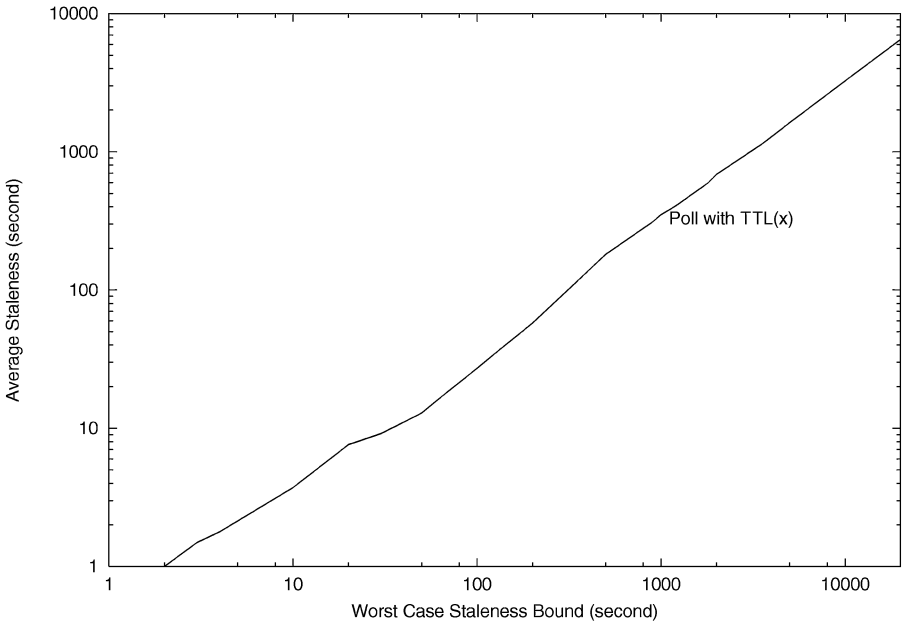

Fig. 6. Average staleness vs. staleness bound of TTL for the IBM workload.

In these figures, the x axis represents the worst-case staleness bounds for the volume lease algorithm; these bounds correspond to the volume lease length for volume lease algorithms and the TTL for TTL algorithms. The y axes in these figures show the fraction of local hits, network traffic, stale rate, and average staleness. The access patterns, such as read frequencies and numbers of repeated accesses of Web objects, are quite different for the IBM Web site and the e-commerce Web site. Thus, hit rates achieved by a given consistency protocol are different for these workloads. We can see that volume leases consistently provide larger advantages for both workloads, due to the impact of amortizing lease renewal overheads across volumes. In particular, for short worst-case staleness bounds, volume lease algorithms achieve significantly higher hit rates and incur lower server overheads compared to TTL algorithms.

As indicated in Figures 1 and 3 (and 2 and 4), providing worst-case staleness bounds of 100 seconds by volume lease is cheaper than providing 10,000-second worst-case staleness bounds by polling in terms of higher local hit-rate and fewer network messages. And, as Figures 5 and 6 indicate, this comparison actually understates the advantages of volume leases because, for polling algorithms, the number of stale reads and their average staleness increase rapidly as the worst-case bound increases. In contrast, volume lease schemes allow servers to notify active clients of updates quickly, regardless of the worst-case staleness guarantees.

Also, notice in Figures 3 and 4 that the server load decreases when volume lease lengths increase from several seconds to several thousand seconds. In the IBM workload, load then increases slightly. This increase arises because as volume lease lengths increase, the number of volume lease renewals decreases, but the number of invalidation messages increases, since if a client holds a volume lease, the server sends invalidation messages to the client immediately, instead of piggy-backing invalidation messages on later volume renewals. When the volume lease length exceeds several thousand seconds, the effect of increasing invalidation messages overtakes the effect of reducing volume lease renewals in the IBM workload. However, server overhead with very long volume leases is not much higher than that with the volume lease length of several thousand seconds.

Overall, volume leases increase hit rates by a factor of 1.5-3 compared to client-polling protocols when the staleness bound is between 10 and 100 seconds. Furthermore when the volume lease length exceeds 1000 seconds, volume lease achieves the local hit rates that are within 5% of Precise Expiration, the theoretical optimal protocol for maintaining cache consistency.

In Figures 7 through 9, we examine two key subsets of the requests in the workloads. We examine the response time and average staleness for the dynamically generated pages and the nonimage objects fetched in the workload. Table I shows that for the IBM workload, the nonimage objects account for 67.6% of all objects, and requests to nonimage objects account for 29.3% of all requests—while the dynamic objects account for 60.8% of all objects, and requests to dynamic objects account for 12% of all requests. The fraction of requests to dynamic data rises to 40.9% when we exclude requests to image

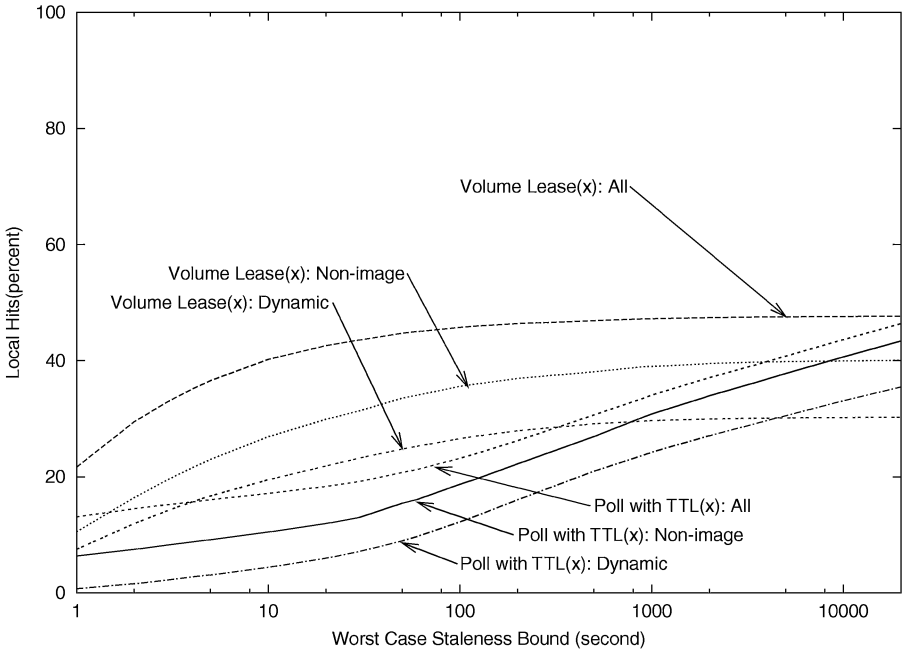

Fig. 7. Local hit rates vs. staleness bound for TTL for the IBM workload. Note that in the common case, volume lease caches are invalidated within a few seconds of an update, independent of worst-case staleness bounds.

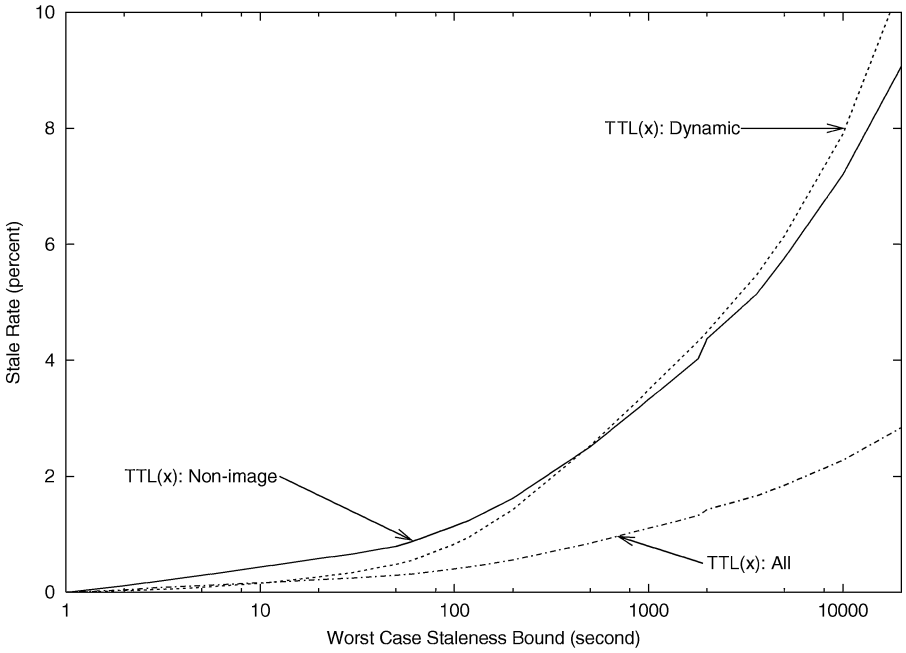

Fig. 8. Stale rate vs. staleness bound for TTL for the IBM workload.

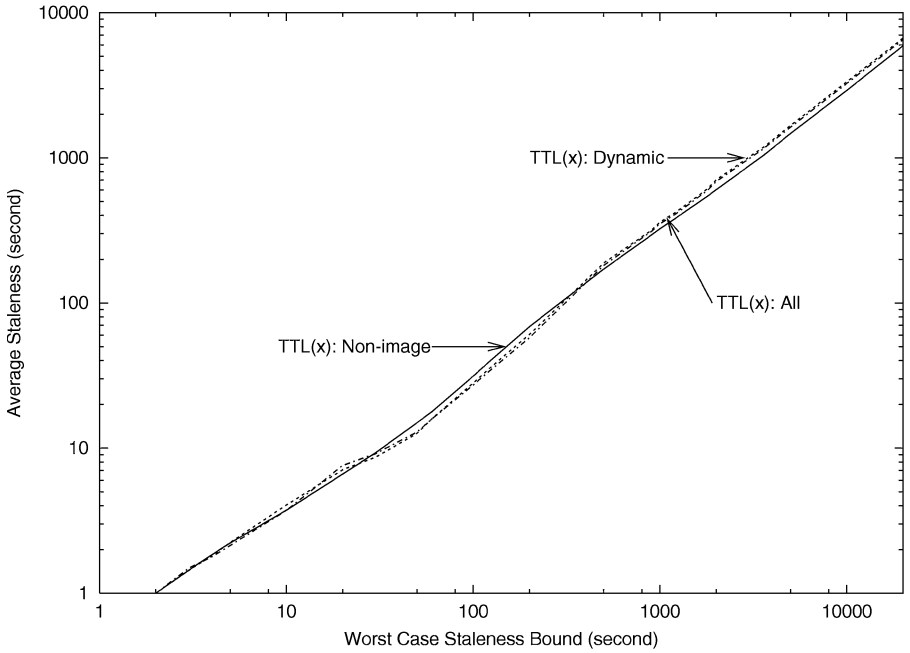

Fig. 9. Average staleness vs. staleness bound for TTL for the IBM workload.

Table I. Classifying Objects and Requests in the IBM Trace According to URL Types

	Object		Request	
	Number	Percent	Number	Percent
image	9027	32.4	6165803	70.7
non-image	18857	67.6	2553543	29.3
dynamic	16960	60.8	1044712	12.0
other non-image	1897	6.8	1508831	17.3
total	27884	100	8719346	100

Table II. Classifying Objects and Requests in the E-Commerce Trace According to URL Types

	Object		Request	
	Number	Percent	Number	Percent
image	9255	13.3	8004528	84.2
non-image	60353	86.7	1500425	15.8
dynamic	59083	84.9	606073	6.4
other non-image	1270	1.8	894352	9.4
total	69608	100	9504953	100

objects. Table II shows that there are also a significant number of requests to dynamically generated objects in our e-commerce workload.

The dynamic and other nonimage data are of interest for two reasons. First, few current systems allow dynamically generated content to be cached. Our system provides a framework for doing so, and no studies to date have examined the impact of server-driven consistency on the cachability of dynamic

data. Several studies have suggested that uncachable data significantly limits achievable cache performance [Wolman et al. 1999a; 1999b], so reducing uncachable data is a key problem. Second, the cache behavior of these subsets of data may disproportionately affect end-user response time. This is because dynamically generated pages and nonimage objects may form the bottleneck in response time, since they must often be fetched before images may be fetched or rendered. In other words, the overall hit rate data shown in Figure 1 may not directly bear on end-user response time if a high hit rate to static images masks a poor hit rate to the HTML pages.

In current systems, a number of factors limit the cachability of dynamically generated data, including (1) the need to determine which objects must be invalidated or updated when underlying data (e.g., databases) change [Challenger et al. 1999]; (2) the need for an efficient cache consistency protocol; (3) the inherent limits to caching that arise when data changes rapidly; and (4) privacy requirements that prevent some dynamic data from being cached at proxy caches. As a result, most current systems use cache control metadata to disable caching for dynamically generated data. Two mechanisms allow our system to cache dynamically generated data effectively. First, as detailed in Section 2, our system provides an efficient method for identifying Web pages that must be invalidated when underlying data changes. Second, as Figures 7 through 9 indicate, volume lease strategies can significantly increase the hit rate for both dynamic pages and for the "bottleneck" nonimage pages. Simulations with our e-commerce workload yield similar results.

Finally, the figures quantify the third limitation. One concern about caching dynamic objects is that dynamic objects may change so quickly that caching them would be ineffective. This concern appears justified, at least for client-polling protocols. To reduce stale read rates for dynamic objects in client polling, clients have to frequently resynchronize with servers by using short TTLs, which leads to low cache hit rates. As shown in Figures 8 and 9, client polling leads to high stale hit rates and high average staleness when the TTL exceeds 1000 seconds. However, the cache hit rate is low when the TTL is small. Fortunately, volume leases allow us to eliminate stale reads in the common case and to achieve high cache hit rates. Hit rates for dynamic objects are lower than for all objects. However, as many as 25% of reads to dynamically generated data can be returned locally with long leases, which increases the local hit rate for nonimage data by up to 10%. Since the local hit rate of nonimage data may determine the actual response time experienced by users, caching dynamic data with server-driven consistency can improve cache performance by as much as 10%. Further performance improvements can be made by prefetching up-to-date versions of dynamically generated objects after the cached versions have been invalidated.

Notice that Figure 7 shows that dynamic pages and nonimage pages are significantly more sensitive to short volume lease lengths than average pages. This sensitivity supports the hypothesis that these pages represent "bottlenecks" to displaying other images; dynamic pages and nonimage pages are particularly likely to cause a miss due to volume lease renewal because they are often the

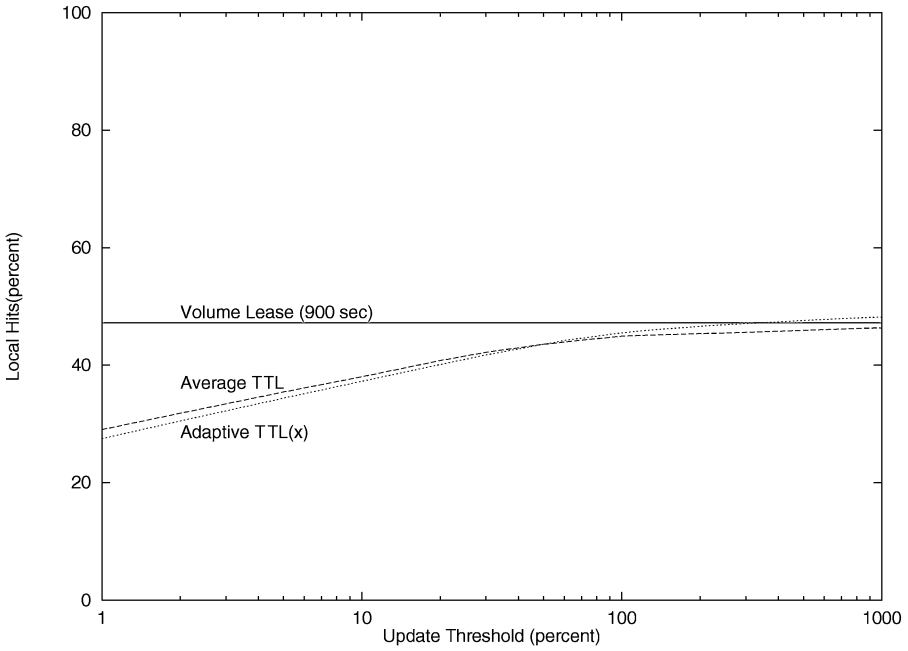

Fig. 10. Local hit rates of Adaptive TTL, Average TTL, and Volume Lease (900) for the IBM workload.

first elements fetched when a burst of associated objects is fetched in a group. In Section 3.4, we examine techniques for reducing the hit rate impact of short worst-case guarantees.

3.3 Adaptive TTL and Average TTL

In this section, we evaluate the set of client-polling protocols that only intend to provide low stale read rates. Since these protocols do not provide worst-case staleness bounds, they are not compared against volume lease in the previous sections. One such TTL algorithm is adaptive TTL [Cate 1992]. Adaptive TTL is designed to reduce stale reads by adjusting client-polling frequencies to modification frequencies of Web objects. In adaptive TTL, the TTL is set to be proportional to an object's age, the amount of time since the last modification. We call this proportion the *update threshold* of the adaptive TTL. The second algorithm is Average TTL. In Average TTL, the TTL of an object is set to be the product of its average life time over the entire trace and the update threshold. Here an object's life is defined as the amount of time between two adjacent updates. Average TTL can be a reasonable approach because researchers have observed that update patterns of some Web objects can be closely approximated by a Poisson distribution [Brewington and Cybenko 2000; Cho and Garcia-Molina 2000] with fixed modification frequencies.

As shown in Figures 10 through 12, volume lease with reasonable volume length can achieve higher local hit rates than average TTL, and adaptive TTL with higher than 100% of update threshold, while only introducing insignificant amounts of additional network overhead. While adaptive TTL with higher than

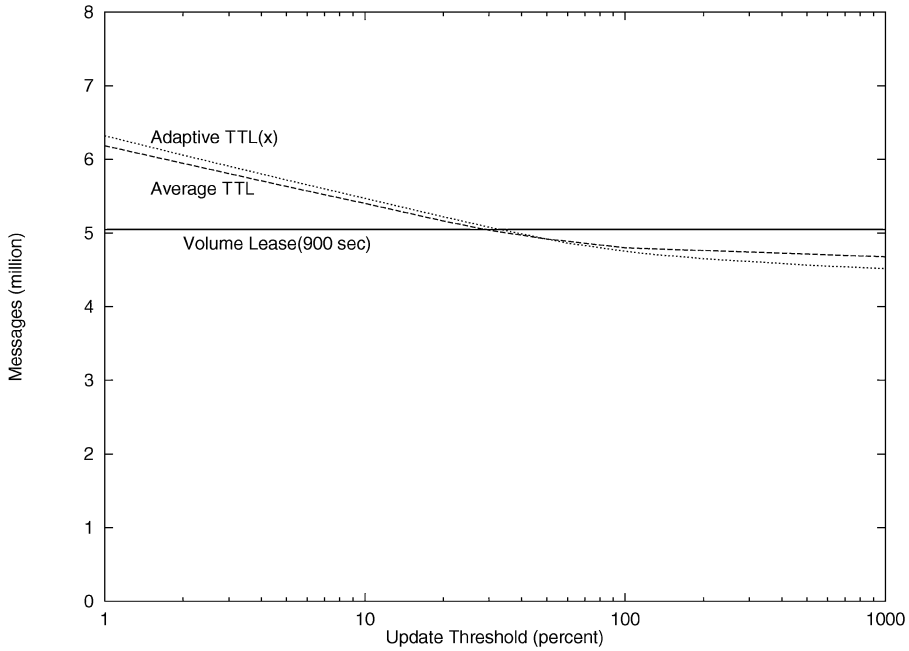

Fig. 11. Number of network messages generated by Adaptive TTL, Average TTL, and Volume Lease (900) for the IBM workload.

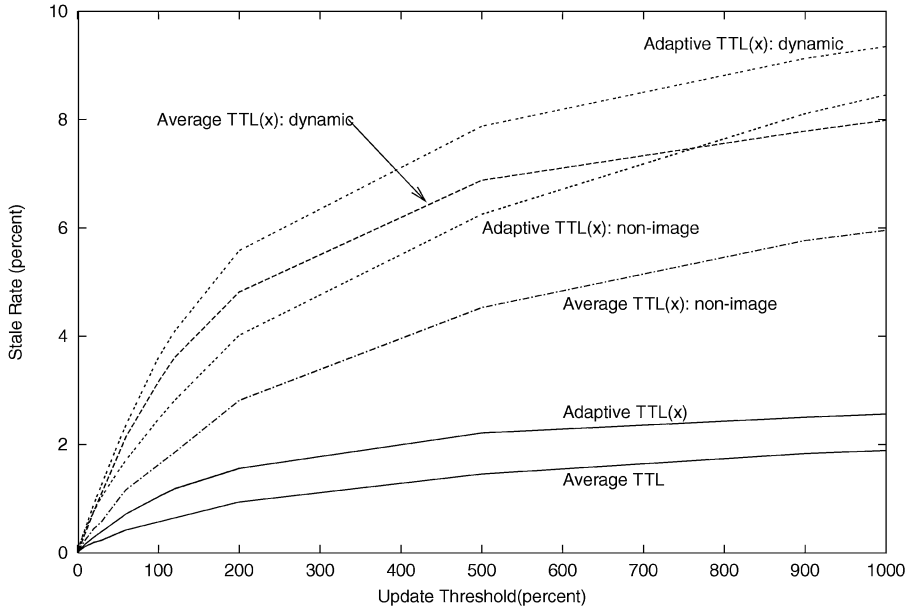

Fig. 12. Stale hit rates of all the objects, the nonimage objects, and the dynamic objects running Adaptive TTL, Average TTL for the IBM workload.

100% threshold and Average TTL introduce significant stale read rates, volume lease always returns fresh data in the absence of failures. The results from simulations with our e-commerce workload are similiar. Thus, volume lease is attractive even for applications that do not need worst-case staleness bound guarantees.

3.4 Prefetching/Pushing Lease Renewals

The above experiments assume that when a volume lease expires, the next request must go to the server to renew it. A potential optimization is to prefetch or push volume lease renewals to clients before their leases expire. For example, a client whose volume lease is about to expire might piggy-back a volume lease renewal request on its next message to the server [Yin et al. 1998], or it might send an additional volume lease renewal prefetch request even if no requests for the server are pending. Alternately, servers might periodically push volume lease renewals to clients via unicast or multicast heartbeat [Yu et al. 1999].

Regardless of whether renewals are prefetched or pushed and whether they are unicast or multicast, the same fundamental trade-offs apply. More aggressive prefetching keeps clients and servers synchronized for longer periods of time, increasing cache hit rates, but also increasing network costs, server load, and client load.

Previous studies assumed extreme positions regarding prefetching volume lease renewals. Yin et al. [1999b] assumed that volume lease renewals are piggy-backed on each demand request, but that no additional prefetching is done; soon after a client becomes idle with respect to a server, its volume lease expires, and the client has to renew the volume lease in the next request to the server's data. Conversely, Li and Cheriton [1999] suggest that to amortize the cost of joining multicast hierarchies, clients should stay connected to the multicast heartbeat and invalidation channel from a server for hours or days at a time.

In Figure 13 and Figure 14, we examine the relationship between pushing or prefetching renewals, read latency, and network overhead. In interpreting these graphs, consider that in order to improve read latency by a given amount, we could increase the volume lease length by a factor of K. Alternatively, we could get the same improvement in read latency by prefetching the lease K times as it expires. We would expect that most services would choose the worst-case staleness guarantee they desire and then add volume lease prefetching if the improvement in read latency justifies the increase in network overhead.

As illustrated in Figure 13, volume lease pull or push can achieve higher local hit rates than basic volume leases for the same freshness bound. In a *push-K* algorithm, if a client is idle when a demand-fetched volume lease expires, the client prefetches or the server pushes to the client up to $K-1$ successive volume lease renewals. Thus, if each volume renewal is for length V, the volume lease remains valid for $K \cdot V$ units of time after a client becomes idle. If a client's accesses to the server resume during that period, they are not delayed by the need for an initial volume lease renewal request.

Both *push-2* and *push-10* shift the basic volume lease curve upward for short volume leases, and larger values of K increase these shifts. Also note that the

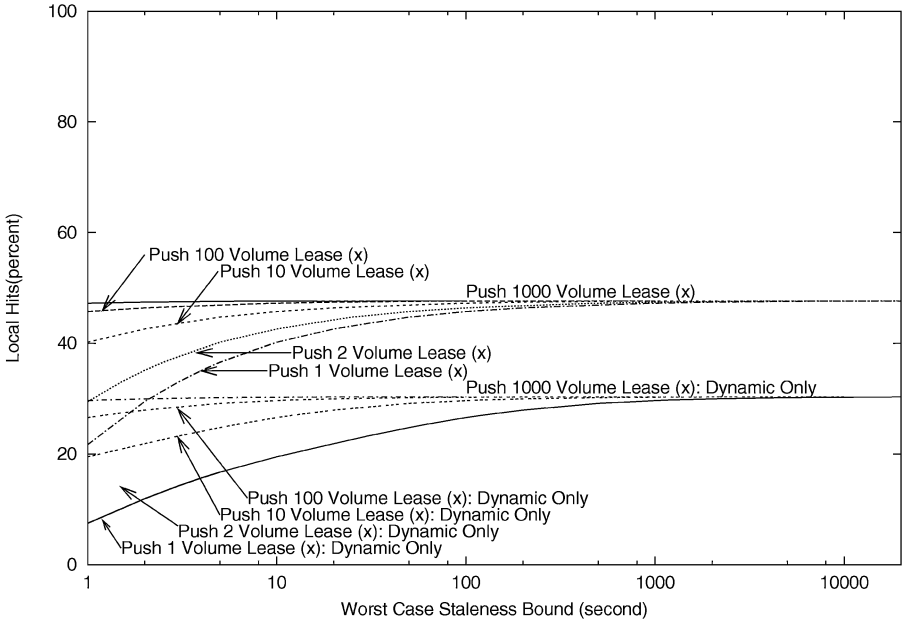

Fig. 13. Prefetching/pushing volume leases to reduce read latency for the IBM workload. Push y volume leases (x) represents a volume lease protocol with volume lease length equal to x which pushes or prefetches volume leases $y - 1$ times after a volume lease expires.

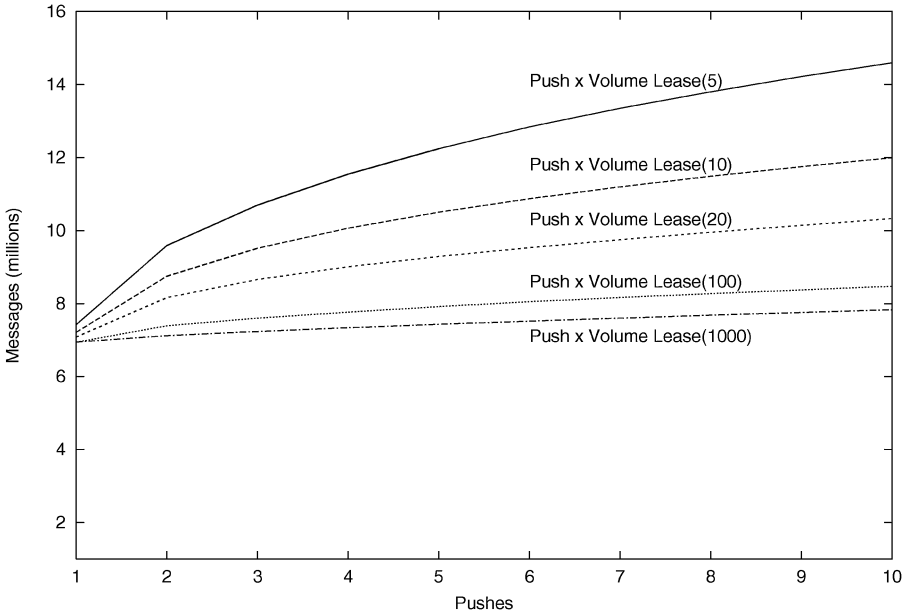

Fig. 14. Cost of pushing volume leases for the IBM workload.

benefits are larger for the dynamic elements in the workload, suggesting that prefetching may improve access to the bottleneck elements of a page.

However, pulling or pushing extra volume lease renewals does increase client load, server load, and network overhead. This overhead increases with the number of renewals prefetched after a client's accesses to a volume cease. For a given number of renewals, this overhead is lower for long volume leases than for short ones.

Systems may use multicast or consistency hierarchies to reduce the overhead of pushing or prefetching renewals. Note that although these architectures may effectively eliminate the volume renewal load on the server and may significantly reduce volume lease renewal overhead in server areas of the network, they do not affect the volume renewal overhead at clients. Although client renewal overhead should not generally be an issue, widespread aggressive volume lease prefetching or pushing could impose significant client overheads in some cases. For example, in the traces of the Squid regional proxies taken during July 2000, these busy caches access tens of thousands of different servers per day [Chandra et al. 2001].

In general, we conclude that although previous studies have examined extreme assumptions for prefetching [Yu et al. 1999; Li and Cheriton 1999], it appears that for this workload, modest amounts of prefetching are desirable for minimizing response time when short volume leases are used, and little prefetching is needed at all for long volume leases. This is because after a few hundred seconds of a client not accessing a service, maintaining valid volume leases at that client has little impact on latency.

3.5 Scalability

Large-scale Web services present several potential challenges to scalability. First, callback-based systems typically store state that is proportional to the total number of objects cached by clients. In the worst case, this state could grow to be proportional to the total number of clients multiplied by the number of objects. Second, when a set of popular objects is modified, servers in callback-based systems send callbacks to the clients caching those objects. In the worst case, such a burst of load could enqueue a number of messages equal to the number of clients using the service multiplied by the number of objects simultaneously modified. For both memory consumption and bursts of load, if uncontrolled, this worst-case behavior could prevent deployment for the IBM Sporting and Event or the e-commerce service we examined.

A wide range of techniques for reducing memory capacity demands or bursts of load are possible. Some have been evaluated in isolation, while others have not been explored. There has been no previous direct comparison of these techniques to one another.

—***Hierarchy or precise multicast.*** Using a hierarchy of consistency servers to flood invalidation messages to caches [Yin et al. 1999b] can reduce bursts of load at the server. *Precise multicast*, which distributes invalidations to clients via multicast and ensures that clients receive invalidations only for objects they are caching, can accomplish the same thing. Precise multicast

can be implemented by having separate multicast channels per object or by filtering multicast distribution on a per-object basis in the network or using a hierarchy of consistency servers [Yin et al. 1999a]. Note that although a hierarchy or precise multicast reduces the amount of state at the central server, the total amount of callback state, and thus the global system cost, is not directly reduced by a hierarchy.

—***Imprecise multicast invalidates.*** Imprecise multicast invalidation [Li and Cheriton 1999] combines two ideas. It uses a multicast hierarchy to flood invalidation messages and *imprecise invalidations* to reduce state. Imprecision of invalidations stems from the use of a single unfiltered multicast channel to transmit invalidations for all objects in a volume. The advantage of imprecise invalidations is reduced state; state at the server and multicast hierarchy is proportional to the number of clients subscribed to the volume, rather than to the number of objects cached across all clients. The disadvantage of imprecise invalidations is increased invalidation message load at the clients and in the network near the clients.

—***Delayed invalidation messages.*** Rather than sending invalidation messages immediately, systems may delay when invalidation messages are sent to reduce bursts of load. There are two variations. *Delayed invalidations* [Yin et al. 1998] enqueue invalidation messages to clients whose volume leases have expired and send the enqueued messages in a group when a client renews its volume lease. *Background invalidations* place invalidation messages in a separate send queue from replies to client requests and send invalidations only when spare capacity is available. Note that background invalidations may increase the average staleness of data observed by clients, while delayed invalidations have no impact on average staleness of data reads. At the same time, while both techniques reduce bursts of load, background invalidations also have the ability to impose a hard upper bound on the maximum load from invalidations.

—***Forget idle clients.*** Two techniques allow servers to drop callbacks on objects cached by idle clients, and thereby reduce server memory requirements. First, by issuing short object leases, servers can discard callback state when a client's lease on an object expires. Duvvuri et al. [2000] examine techniques for optimizing the lease lengths of individual objects. Second, servers can mark clients whose volume leases have expired some amount of time in the past as "unreachable," and drop all callback state for unreachable clients [Yin et al. 1998]. When an unreachable client renews its volume lease, the client and server must execute a reconnection protocol to synchronize server callback state with client cache contents. The next section discusses reconnection protocols. The fundamental trade-offs for both approaches are the same: shorter leases reduce memory consumption but also increase consistency misses and synchronization overhead. Either algorithm can enforce a hard limit on memory capacity consumed by adaptively shortening leases as space consumption increases [Duvvuri et al. 1999].

In evaluating this range of options, two factors must be considered. First, given the potential worst case memory and load behavior of callback

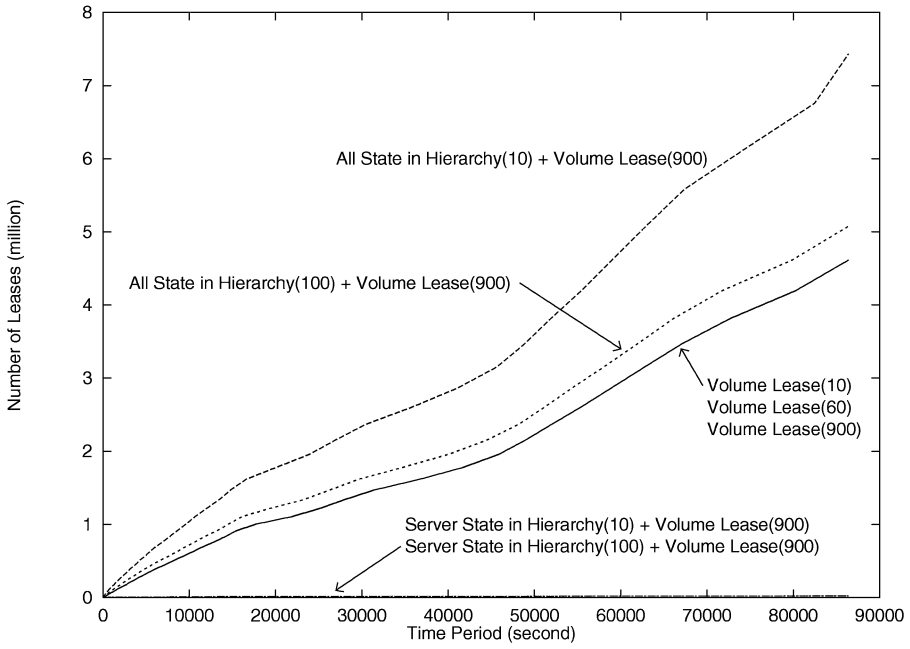

Fig. 15. Callback state increases with elapsed time for the IBM workload. In this graph, *Hierarchy (m) + Volume Lease (l)* represents the hierarchical volume lease algorithm with the volume lease length equal to l units of time running over a hierarchy in which each node in the hierarchy can have up to m children. Note that the *Server State in Hierarchy* lines overlap the x-axis.

consistency, a system should enforce a hard worst-case limit on state consumption and bursts of load regardless of the workload. Second, systems should select techniques that minimize damage to hit rates and overhead for typical loads.

3.5.1 *Server Callback State.* Figure 15 shows the number of object leases stored as a function of elapsed time in the trace for a nonhierarchical system and for the consistency hierarchies (or precise multicast systems) with fan-out of 10 or 100 children per invalidation server. Note that whether the server holds an object lease on an object for a client is determined only by whether the client caches the object, and is independent of volume lease length. For the time period covered in our trace, server memory consumption increases linearly. Although for a longer trace, higher hit rates might reduce the rate of growth, for the Zipf workload distributions common on the Web [Breslau et al. 1998; Wolman 1999a; 1999b], hit rates improve only slowly with increasing trace length, and a nearly constant fraction of requests will be compulsory cache misses. Nearly linear growth in state therefore may be expected even over long time scales for many systems.

Although the near linear growth in state illustrates the need to bound worst-case consumption, the rate of increase for this workload is modest. After 24 hours, fewer than 5 million leases exist in one of the four Sporting and Event server clusters even with infinite object leases. Our prototype consumes 62 bytes per object lease, so this workload consumes 310 million bytes per day under the baseline algorithm for the whole system. This corresponds to 0.4%

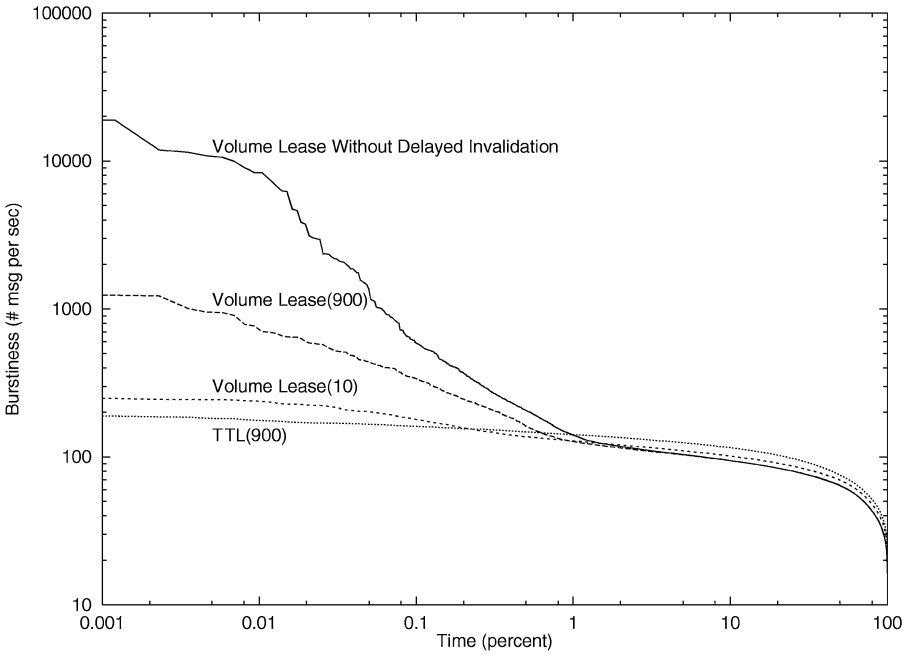

Fig. 16. Distribution of invalidation burstiness for the IBM workload. A point of (x, y) means that the server generates at least y messages per second during x percentage of time.

of the memory capacity of the 143-processor system that actually served this workload in a production environment. In other words, this busy server could keep complete callback information for 10 days and increase its memory requirements by less than 4%.

These results suggest that either of the "forget idle clients" approaches can limit maximum memory state without significantly hurting hit rates or increasing lease renewal overhead, and that performance will be relatively insensitive to the detailed parameters of these algorithms. Because systems can keep several days of callback state at little cost, further evaluation of these detailed parameters will require longer traces than we have available to us.

The same conclusion applies to consistency hierarchies or precise multicast. As Figure 15 indicates, although a consistency hierarchy can reduce server callback state by orders of magnitude, the total amount of callback state maintained by all the machines in a hierarchy increases compared to that of a nonhierarchical central server.

3.5.2 *Bursts of Load.* Figure 16 shows the cumulative distribution of server load, approximated by the number of messages sent and received by a server with no hierarchy. As we can see from the right edge of this graph, volume leases with callbacks reduce average server load compared to TTL. However, as can be seen from the left side of the graph, the peak server load increases by a factor of 100 for volume leases without delayed invalidations.

This figure shows that delayed invalidations can reduce peak load by a factor of 76 for short volume lease periods and 15 for long volume lease

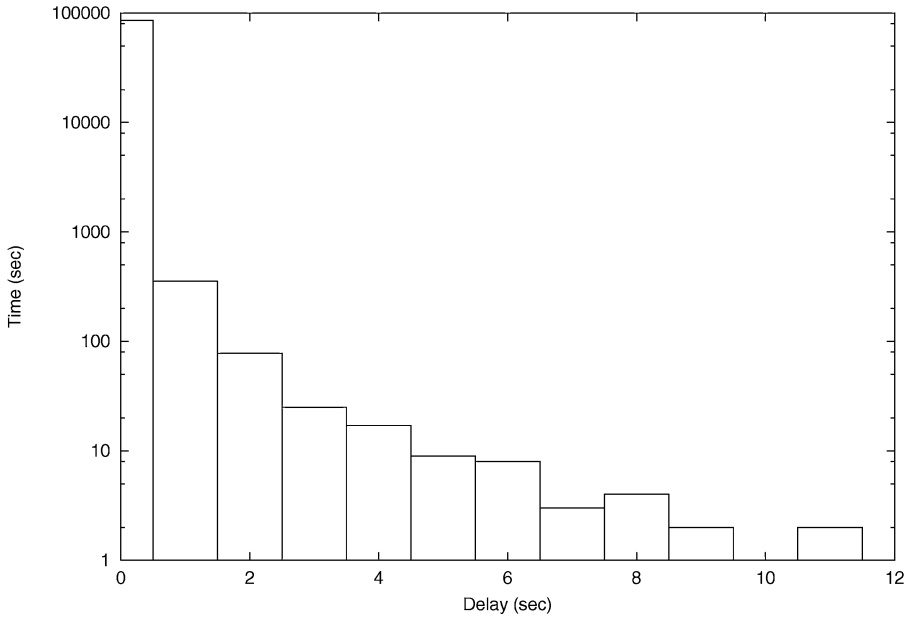

Fig. 17. Distribution of delay of invalidation messages under background invalidations for Volume Lease (900).

periods, but even with delayed invalidations, peak load is increased by a factor of 6 for 15 minute volume leases. This increase is smaller for short volume leases and larger for long volume leases, since delayed invalidations' advantage stems from delaying messages to clients whose volume leases have expired.

Further improvements can be gained by also using background invalidations. In Figure 17, we limit the server message rate to 200 messages per second, which is approximately the average load of TTL, and we send invalidation messages as soon as possible, but only using the spare capacity. Well over 99.9% of invalidation messages are transmitted during the same second they are created, and no messages are delayed more than 11 seconds. Thus, background invalidation allows the server to place a hard bound on load burstiness without significantly hurting average staleness.

Figure 5 shows the average staleness for the traditional TTL polling protocol. The data in Figure 17 allows us to understand the average staleness that can be delivered by invalidation with volume leases. Clients may observe stale data if they read objects between when the objects are updated at the server and when the updates appear at the client. There are two primary cases to consider. First, the network connection between the client and server fails. In that case, the client may not see the invalidation message, and data staleness will be determined by the worst-case staleness bound from leases. Fortunately, failures are relatively uncommon, and this case will have little effect on average staleness. Second, server queuing and message propagation time will leave a window when clients can observe stale data. The data in Figure 17 suggests

that this window will likely be at most a few seconds, plus whatever propagation delays are introduced by the network.

We conclude that delaying invalidation messages makes unicast invalidation feasible with respect to server load. This is encouraging because it simplifies deployment: systems do not need to rely on hierarchies or multicast to limit server load. In the long run, hierarchies or multicast are still attractive strategies for further reducing latency and average load [Yin et al. 1999b; Yu et al. 1999].

3.5.3 *Client Issues.* Server-driven consistency protocols appear to introduce several challenges for clients. First, a client can interact with a large number of servers over a long period of time. Thus, a client may need to process invalidation messages from a large number of servers. However, our analysis suggests that volume lease can manage client overhead effectively. Volume lease effectively caps the total number of invalidation messages that a client can receive and the burstiness of the invalidation messages. In volume lease, servers maintain the callback state of client caches so that invalidation messages are only sent to the cached objects that have not been invalidated. Thus, the number of invalidation messages can be no larger than the number of reads by clients. Moreover, our previous simulation results suggest that volume lease generally leads to a smaller number of messages between servers and clients compared to client polling.

Second, in naive server-driven consistency protocols, a client may suffer bursts of invalidation messages if the cached objects from many servers are updated at the same time. This issue can be effectively addressed by delayed invalidation as discussed in the background section. With delayed invalidation, a server sends invalidation messages to a client only if the client holds the volume lease from that server. The amount of time that a volume lease is valid after the time that the client reads an object from the volume is equal to the volume lease length. Thus at any point in time, a client only receives messages from the small number of servers that it has accessed within the period of time that equals the volume lease length.

Third, a server may not always be able to contact a client when it needs to send invalidation messages. For example, a mobile client can be disconnected from networks occasionally, or it may choose to power down to save battery power. We call these scenarios *disconnections*. Volume lease provides efficient mechanisms to recover after a disconnection. These mechanisms are discussed in the next section.

3.6 Resynchronization

In server-driven consistency, servers' consistency state must be synchronized with client cache contents to bound staleness for client reads. This synchronization can be lost due to failures, which include server crashes, client crashes, and network partitions. Additionally, to achieve scalability, servers may drop callbacks of idle clients to limit server state, as discussed in Section 3.5.

These disconnections can be roughly divided into two groups based on whether the consistency state before a disconnection survives the disconnection. *State-preserving disconnections* caused by network partitions preserve

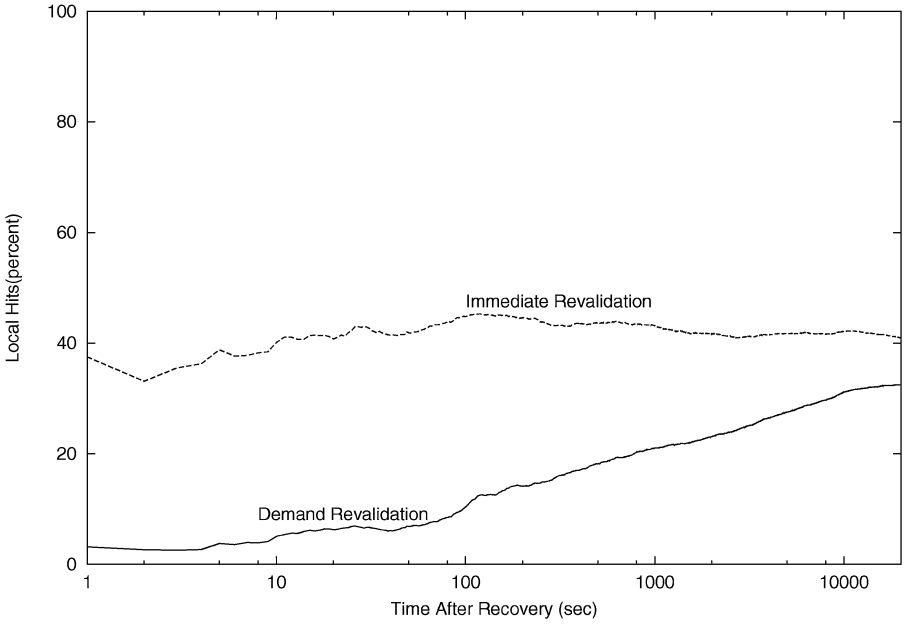

Fig. 18. Hit rates after recovery for a disconnection of 1 second for the IBM workload.

the consistency state prior to the disconnections. *State-losing disconnections* caused by server crashes, client crashes, and deliberate protocol disconnections result in loss of consistency state. In this section, we systematically study the design space of resynchronization to recover from all these disconnections.

There are three potential policies to control how aggressively clients resynchronize with servers. At one extreme, *demand revalidation* marks all cached objects as potentially stale after reconnection and revalidates each object individually as it is referenced. At the other extreme, *immediate revalidation* revalidates all cached objects immediately after reconnections to reduce the read latency associated with revalidating each object individually. When the overhead of revalidating all cached objects is high, immediate revalidation may delay clients' access to servers immediately after reconnections. To address this problem, *background revalidation* allows bulk revalidation to be processed in the background. Some previous studies have assumed that demand revalidation is sufficient [Liu and Cao 1997], while others have assumed that immediate revalidation is justified [Yin et al. 1999b]. In this study, we quantitatively evaluate these two options and the middle ground, background revalidation.

Figures 18 and 19 show that immediate revalidation achieves higher average local hit rates than demand revalidation. The performance disparity between immediate revalidation is larger immediately after failures and decreases over time as more cached objects are accessed and validated in demand revalidation. The local hit rates of background revalidation would range between these two lines in the graph—they equal those of demand revalidation immediately after reconnection, and would increase to those of immediate revalidation as

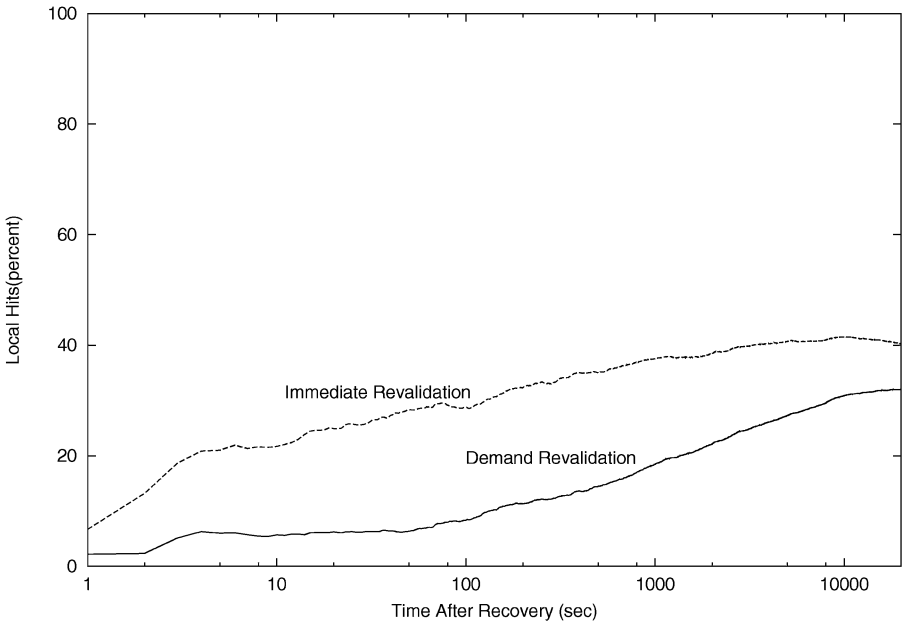

Fig. 19. Hit rates after recovery for a disconnection of 1000 seconds for the IBM workload.

background revalidation completes. The benefit of immediate and background revalidation is also affected by disconnection duration. When disconnection duration is short, the number of cached objects that are invalidated during disconnections is small. Moreover, because of read locality, the chance of reading these cached objects after recovery is high. Hence, as shown by Figures 18 and 19, the benefit of immediate and background revalidation is significant for short disconnections. Conversely, when a disconnection duration is long, demand revalidation may be sufficient.

To implement demand revalidation, systems only need to detect reconnections and mark all cached objects as potentially stale by dropping all object leases. Revalidating cached objects in demand revalidation is the same as validating cached objects after the object leases expire. Two additional mechanisms can be added to support immediate or background revalidation. First, in *bulk revalidation*, a client simply sends a revalidation message containing requests to revalidate a collection of objects. The server processes each included request as it would have if it had been sent separately and on demand, except that the server replies with a bulk revalidation message containing object leases for all unchanged objects and invalidations for objects that have changed. Second, in *delayed invalidation*, the server buffers the invalidations that should be sent to a client when a network partition makes a client unreachable from the server or when the server decides to delay sending invalidation messages to an idle client to reduce server load. When the server receives a volume lease request message from the client, the server piggy-backs the buffered invalidations on the reply message granting the client a volume lease. The client applies these buffered invalidations to resynchronize with the

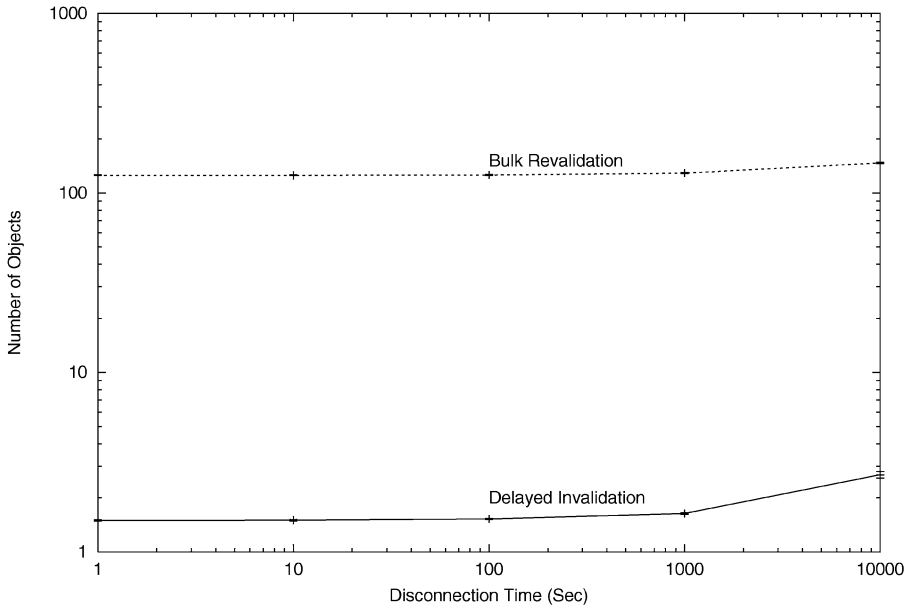

Fig. 20. Resynchronization cost for bulk revalidation and delayed invalidation for the IBM workload.

server. Note that delayed invalidation may only be used after state-preserving disconnections.

The overhead of bulk revalidation and delayed invalidation primarily depends on the number of cached objects and on the number of objects invalidated during the disconnection. In the case of bulk revalidation, server load and network bandwidth are proportional to the number of cached objects; in delayed invalidation, they are instead determined by the number of invalidated objects. As Figure 20 shows, bulk revalidation must examine an average of more than 100 objects. For some recovered clients, several thousand objects must be compared during bulk revalidation. Delayed invalidation can be used to reduce the cost of immediate revalidation for state-preserving disconnections, since the number of cached objects is two orders of magnitude less than the number of invalidated objects for disconnections shorter than 1000 seconds. Unfortunately, for state-losing disconnections, delayed invalidation is not an option. Because bulk revalidation may have to revalidate hundreds or thousands of objects, the system should support background revalidation, rather than relying solely on immediate revalidation. These conclusions also apply to the results from simulations with our e-commerce workload.

In conclusion, server-driven consistency protocols must implement some resynchronization mechanisms for fault tolerance and scalability. Demand resynchronization is a good default choice, since it handles all disconnections and is simple to implement. Background bulk revalidation may be needed to reduce read latency when recovering from short disconnections, and delayed invalidation may be desirable to reduce resynchronization overheads for short state-preserving disconnections.

4. PROTOTYPE

We have implemented server-driven consistency based on volume leases with Squid cache version 2.2.5. The consistency module includes a server part, which sends invalidation messages in response to writes and issues volume and object leases, and it includes a client part, which manages consistency information on locally cached objects to satisfy reads. Servers maintain a per-object version number as well as per-volume and per-object share lists. These lists contain the set of clients that may hold valid leases on the volumes and objects, respectively, along with the expiration time of each lease. Clients maintain a volume expiration time as well as per-object version numbers and expiration times. Objects and volumes are identified by URLs, and object version numbers are implemented by HTTP Etag or modification times [Fielding et al. 1999].

On a client request for data, a callback-enabled client includes a `VLease-Request` field in the header of its request. This field indicates a network port at the client that may be used to deliver callbacks. A callback-enabled server includes volume lease and object leases as `Volume-Lease-For` and `Object-Lease-For` headers of replies to client requests. `Invalidation` and `Invalidation-Ack` headers are used to send invalidation messages to clients and to acknowledge receiving invalidations by clients.

The mechanisms provided by the protocol support either client-pull or server-push volume lease renewal. At present, we implement the simple policy of client-pull volume lease renewal. Volume lease requests and replies can use the same channels used to transfer data or can be exchanged along dedicated channels.

4.1 Hierarchy

We construct the system to support a hierarchy in which each level grants leases to the level below and requests leases from the level above [Yin et al. 1999b]. The top-level cache is a reverse proxy that intercepts all requests to the origin server and caches all replies, including dynamically generated data. The top-level cache is configured to hold infinite object and volume leases on all objects and to pass shorter leases to its children. Invalidations are sent to the top-level cache using the standard invalidation interface used for communication between parent and child caches. These invalidations can be generated by systems such as the trigger monitor used at the major Sporting and Event Web sites hosted by IBM [Challenger et al. 1999]. The trigger monitor maintains correspondences between underlying data (e.g., databases) affecting Web page content and the Web pages themselves. In response to changes to underlying data, the trigger monitor determines which cached pages are affected and propagates invalidation or update messages to the appropriate caches.

The hierarchy provides three benefits. First, it simplifies our prototype by allowing us to use a single implementation for servers, proxies, and clients. Second, hierarchies can considerably improve the scalability of lease systems by forming a distribution tree for invalidations and by serving renewal requests from lower-level caches [Yin et al. 1999b]. Third, reverse-proxy caching of dynamically generated data at the server can achieve high hit rates and can

dramatically reduce server load [Challenger et al. 1998]. By implementing our system as a hierarchy, we make it easy to gain these advantages. Further, if a multilevel hierarchy is used (such as the Squid regional proxies (http://www.squid-cache.org/) or a cache mesh [Tewari et al. 1999]), we speculate that nodes higher in the hierarchy will achieve hit rates between the per-client cache hit rates and the at-server cache hit rates illustrated in Section 3.

4.2 Reliable Delivery of Invalidations

In order to maintain an upper bound on worst-case staleness, volume lease systems must maintain the following invariant: a client may not receive a volume lease renewal unless all of its cached objects that were modified before the transmission of the volume renewal have been invalidated. If this invariant is violated, an object may be modified at time T_1, and the client may then receive a volume lease renewal valid until time $T_2 > T_1 + T_v$. If a network partition then occurs, the client could access stale cached data longer than T_v seconds after it was modified.

The system must therefore reliably deliver invalidations to clients. It does so in two ways. First, it uses a delayed invalidation buffer to maintain reliable invalidation delivery across different transport-level connections. Second, it maintains *epoch numbers* and an *unreachable list* to allow servers to resynchronize after servers discard or lose client state [Yin et al. 1999b].

We use TCP as our transport layer for transmitting invalidations, but, unfortunately, this does not provide the reliability guarantees we require. In particular, although TCP provides reliable delivery within a connection, it cannot provide guarantees across connections: If an invalidation is sent on one connection and a volume renewal on another, the volume renewal may be received and the invalidation may be lost if the first connection breaks. Unfortunately, a pair of HTTP nodes will use multiple connections to communicate in at least three circumstances. First, HTTP 1.1 allows a client to open as many as two simultaneous persistent connections to a given server [Mogul 1996]. Second, HTTP 1.1 allows a server or client to close a persistent connection after any message; many modern implementations close connections after short periods of idleness to save server resources. Third, a network, client, or server failure may cause a connection to close and a new one to be opened. In addition to these fundamental limitations of TCP, most implementations of persistent connection HTTP are designed as performance optimizations, and they do not provide APIs that make it easy for applications to determine which messages were sent on which channels.

We therefore implement reliable invalidation delivery that is independent of transport-layer guarantees. Clients send explicit acknowledgments to invalidation messages, and servers maintain lists of unacknowledged invalidation messages to each client. When a server transmits a volume lease renewal to a client, it piggy-backs the list of the client's unacknowledged invalidations using a Group-Object-Version header field. Clients receiving such a message must process all invalidations in it before processing the volume lease renewal.

Three aspects of this protocol are worth noting.

First, the invalidation delivery requirement in volume leases is weaker than strict reliable in-order delivery, and the system can take advantage of that. In particular, a system need only retransmit invalidations when it transmits a volume lease renewal. At one extreme, the system can mark all packets as volume lease renewals to keep the client's volume lease fresh, but at the cost of potentially retransmitting more invalidations than necessary. At the other extreme, the system can only send periodic heartbeat messages and handle all retransmission at the end of each volume lease interval [Yu et al. 1999].

Second, the queue of unacknowledged invalidations provides the basis for an important performance optimization: delayed invalidation [Yin et al. 1998]. Servers can significantly reduce their average and peak load by not transmitting invalidation messages to idle clients whose volume leases have expired. Instead, servers place these invalidation messages into the idle clients' unacknowledged invalidation buffer (also called the delayed invalidation buffer) and do not transmit these messages across the network. If a client becomes active again, it first asks the server to renew its volume lease, and the server transmits these invalidations with the volume lease renewal message. The unacknowledged invalidation list thus provides a simple, fast reconnection protocol.

Third, the mechanism for transmitting multiple invalidations in a single message is also useful for atomically invalidating a collection of related objects. Our protocol for caching dynamic data supports documents that are constructed of multiple fragments, and atomic invalidation of multiple objects is a key building block [Challenger et al. 2000].

The system also implements a protocol for resynchronizing client or server state when a server discards callback state about a client. This occurs after a server crash or when a server deliberately discards state for idle clients. The system includes *epoch numbers* [Yin et al. 1998] in messages to detect loss of synchronization due to crashes. The servers maintain unreachable lists, lists of clients whose state has been discarded to detect when such clients reconnect. If a reply to a client request includes an unexpected epoch number or a header indicating that the client is on the unreachable list, the client invalidates all object leases for the volume and renews them on demand. A subject of future work is to implement a bulk revalidation protocol.

4.3 Evaluation

We evaluate our implementation with a standard benchmark of the Web caching industry: the first semi-annual web caching bake-off workload (http://cacheoff.ircache.net/N01/). Our testbed includes four computers. Two of them run the workload. The consistency server and the consistency client, which are the Squid proxies augmented with server-driven consistency, run on two other machines. The consistency server is placed in front of the workload server, which delivers data requested by clients after retrieving it from the workload server. The consistency server also issues leases and sends invalidation messages.

Our initial evaluation shows that our implementation of server-driven consistency, compared to the standard Squid cache, increases the load on the consistency server by less than 3% and increases read latency by less than 5%, while sustaining a throughput of 70 requests per second. In the future, we plan to implement an architecture that decouples the consistency module from the other parts of the Web or proxy server that deliver data, and to quantify the computing resources (CPU, memory) needed to maintain server-driven consistency for the traces examined here.

5. RELATED WORK

Many researchers have observed that Web cache consistency is critical to Web performance. For example, Padmanabhan and Qiu [2000] examined a large-scale enterprise Web trace. They concluded that traditional client polling is unsatisfactory for their workload.

Owing to its significant impact on end-to-end Web performance, Web cache consistency has generated much interest in the research community. Based on their simulation study with write traces collected from Web servers and synthetic read traces, Gwertzman and Seltzer [1996] concluded that adaptive TTL can provide good performance for applications that can tolerate 4% stale reads. Evaluating a prototype with real-world traces, Liu and Cao [1997] found that much of the bandwidth-saving benefit of adaptive TTL is derived from reading stale data by clients and that server-driven consistency protocols can improve freshness of client reading without introducing significant overhead. However, they discovered that there are two challenges in deploying server-driven consistency protocols: scalability and fault tolerance. In particular, they point to (i) the bursts of server load caused by invalidations sent when popular objects are written; (ii) the growth of the state that the server maintains to track the caches of its clients; and (iii) if partial failures prevent servers from contacting clients, then clients may continue to return stale data. These three challenges are addressed in this article.

Several researchers studied how to use multicast to scale server-driven consistency protocols. With multicast, invalidation messages can be delivered precisely, in which only the set of invalidation messages required by a client are delivered to the client, or imprecisely, in which extra invalidations are delivered to a client. Yu et al. [1999] proposed a scalable cache consistency architecture that integrates the ideas of invalidation, volume lease, and unreliable imprecise multicast. They used synthetic workloads of single pages and focused their evaluation on network performance of server-driven consistency. Li and Cheriton [1999] proposed using reliable, precise multicast to deliver invalidations and updates for frequently modified objects. The workloads in their study included client traces, proxy traces, and synthetic traces.

Other studies examined specific design optimizations to reduce overhead or read latency. Mogul [1996] independently proposed a notion of grouping files into volumes to reduce the overhead of client polling. Duvvuri et al. [1999] examined adapting object leases to reduce server state and messages. These techniques can also be employed in our protocol to improve scalability. Cohen

et al. [1998] studied the use of volumes for prefetching and consistency. The consistency algorithms they examined are best-effort algorithms based on client polling. Krishnamurthy and Wills [1998] examined ways to improve polling-based consistency by piggy-backing optional invalidation messages on other traffic between a client and server. While their study doesn't provide the worst-case staleness bounds, several techniques in their study can also be exploited to improve performance and scalability of server-driven consistency. For example, our protocol allows servers to send delayed invalidations to clients by piggy-backing them on top of other traffic between servers and clients. In the same paper, they also proposed to group related objects into volumes and to send the invalidations on all objects contained in volumes, instead of just invalidations on the objects that a client caches. This imprecise unicast idea obviates the need for the server to track the callback state related to each client, resulting in a scheme that may be used in some extreme cases by server-driven consistency protocols to limit server state. Cohen and Kaplan [2001] observed that polling cached objects when TTLs expire, while the cached objects are still valid, can significantly increase read latency. Thus they proposed to proactively refresh TTLs. This technique is similar to the volume lease prefetching we discuss in Section 3.4, but it effectively prefetches individual object leases rather than resynchronizing an entire volume with one lease renewal.

Finally, we observe that cache consistency protocols have long been studied for distributed file systems [Howard et al. 1988; Mogul 1994; Nelson et al. 1988; Sandberg et al. 1985]. In particular, Howard et al. [1988] compared NFS's polling to AFS's callback-based strategy and concluded that polling incurs higher server load and stale hit rates. The notion of invalidation and that of leases for fault tolerance have been examined in this context [Fray and Cheriton 1989]. Baker studied methods for fast resynchronization of callback state [Baker 1994] and showed that server-driven recovery can be used to avoid "recovery storm," the overwhelming recovery load at a server immediately after a server reboot. Background revalidation in our framework allows the server to control the load during revalidation, which has the same spirit as server-driven recovery.

6. CONCLUSIONS

Although server-driven consistency can provide significant performance advantages over traditional client-polling systems, the feasibility of deploying such a system depends on the scalability and performance of these server-driven consistency algorithms over a wide range of applications. Large-scale services delivering both static and dynamically generated data are an important class of applications to be considered, since objects served by such applications change unpredictably and frequently, and the scale of such a service presents many challenges. In this study we find that server-driven consistency can meet the scalability, performance, and consistency requirements of these services. First, we find that we can put a limit on callback state growth with little performance penalty and that we can smooth out server burstiness introduced by invalidations without significantly increasing average staleness. Second, we find that

how long servers and clients are kept synchronized can greatly influence performance and overhead. However, for these workloads there is little performance benefit in keeping servers and clients synchronized longer than 1000 seconds after a read. Third, we find that delayed invalidation is the most efficient fault-recovery protocol for the most common failures in today's Internet. Overall, server-driven consistency can offer excellent performance for both static and dynamically generated data in large-scale Web services.

ACKNOWLEDGMENTS

The authors would like to thank Jim Challenger and Paul Dantzig for their help in acquiring and interpreting the IBM Sporting and Event web logs. The authors would also like to thank Jeffrey Mogul, Craig Wills, and the anonymous reviewers for their suggestions which helped in enhancing the quality of this article.

This work was supported in part by DARPA/SPAWAR grant N66001-98-8911, NSF CISE grant CDA-9624082, the Texas Advanced Technology Program, an IBM Faculty Partnership Award, and Tivoli. Alvisi was also supported by an NSF CAREER award (CCR-9734185) and an Alfred P. Sloan Fellowship. Dahlin was also supported by an NSF CAREER award (CCR-9733842) and an Alfred P. Sloan Fellowship.

REFERENCES

BAKER, M. 1994. Fast crash recovery in distributed file systems. PhD thesis, University of California at Berkeley.

BRESLAU, L., CAO, P., FAN, L., PHILLIPS, G., AND SHENKER, S. 1998. On the implications of Zipf's law for web caching. Technical Report 1371, University of Wisconsin (Apr. 1998).

BREWINGTON, B. AND CYBENKO, G. 2000. How dynamic is the web? In *World Wide Web* (May 2000). http://cacheoff.ircache.net/N01/.

CATE, V. 1992. Alex—a global filesystem. In *Proceedings of the 1992 USENIX File System Workshop* (May 1992).

CHALLENGER, J., DANTZIG, P., AND IYENGAR, A. 1998. A scalable and highly available system for serving dynamic data at frequently accessed web sites. In *Proceedings of ACM/IEEE SC98* (Nov. 1998).

CHALLENGER, J., IYENGAR, A., AND DANTZIG, P. 1999. A scalable system for consistently caching dynamic web data. In *Proceedings of IEEE INFOCOM'99* (Mar. 1999).

CHALLENGER, J., IYENGAR, A., WITTING, K., FERSTAT, C., AND REED, P. 2000. A publishing system for efficiently creating dynamic web content. In *Proceedings of IEEE INFOCOM 2000* (Mar. 2000).

CHANDRA, B., DAHLIN, M., GAO, L., AND NAYATE, A. 2001. End-to-end wan service availability. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (USITS01).

CHO, J. AND GARCIA-MOLINA, H. 2000. Synchronizing a database to improve freshnes. In *VLDB*, 2000.

COHEN, E. AND KAPLAN, H. 2001. Refreshment policies for web content caches. In *INFOCOM 2001*.

COHEN, E., KRISHNAMURTHY, B., AND REXFORD, J. 1998. Improving end-to-end performance of the web using server volumes and proxy filters. In *SIGCOMM '98*.

http://httpd.apache.org/docs/logs.html.

DUVVURI, V., SHENOY, P., AND TEWARI, R. 1999. Adaptive leases: A strong consistency mechanism for the World Wide Web. In *INFOCOM 2000*.

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. 1999. Hypertext Transfer Protocol—HTTP/1.1. Request for Comments 2616, Network Working Group (June 1999).

GRAY, C. AND CHERITON, D. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 202–210.

GWERTZMAN, J. AND SELTZER, M. 1996. World-Wide Web cache consistency. In *Proceedings of the 1996 USENIX Technical Conference* (Jan. 1996).

HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988), 51–81.

IYENGAR, A. AND CHALLENGER, J. 1997. Improving Web server performance by caching dynamic data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (Dec. 1997).

IYENGAR, A., SQUILLANTE, M., AND ZHANG, L. 1999. Analysis and characterization of large-scale web server access patterns and performance. In *World Wide Web* (June 1999).

KRISHNAMURTHY, B. AND WILLS, C. 1998. Piggyback Server Invalidation for proxy cache coherency. In *Proceedings of the Seventh International World Wide Web Conference* 185–193.

LI, D. AND CHERITON, D. R. 1999. Scalable Web caching of frequently updated objects using reliable multicast. In *Proceeding of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS'99)* (Oct. 1999).

LIU, C. AND CAO, P. 1997. Maintaining strong cache consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems* (May 1997).

MOGUL, J. 1996. http://www.roads.lut.ac.uk/lists/http-caching/1996/01/0002.html.

MOGUL, J. 1994. Recovery in Spritely NFS. *Comput. Syst. 7*, 2 (Spring 1994), 201–262.

MOGUL, J. 1996. A design for caching in HTTP 1.1 preliminary draft. Tech. Rep., Internet Engineering Task Force (IETF), Jan. 1996. Work in progress.

NELSON, M., WELCH, B., AND OUSTERHOUT, J. 1988. Caching in the sprite network file system. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988).

PADMANABHAN, V. AND QIU, L. 2000. The content and access dynamics of a busy web site: Findings and implications. In *SIGCOMM '2000*.

SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. 1985. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference* (June 1985), 119–130.

TEWARI, R., DAHLIN, M., VIN, H., AND KAY, J. 1999. Design considerations for distributed caching on the Internet. In *Proceedings of the Nineteenth International Conference on Distributed Computing Systems* (May 1999).

WOLMAN, A., VOELKER, G., SHARMA, N., CARDWELL, N., BROWN, M., LANDRAY, T., PINNEL, D., KARLIN, A., AND LEVY, H. 1999a. Organization-based analysis of Web-object sharing and caching. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems* (Oct. 1999).

WOLMAN, A., VOELKER, G., SHARMA, N., CARDWELL, N., KARLIN, A., AND LEVY, H. 1999b. On the scale and performance of cooperative Web proxy caching. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Dec. 1999).

YIN, J., ALVISI, L., DAHLIN, M., AND LIN, C. 1998. Using leases to support server-driven consistency in large-scale systems. In *Proceedings of the Eighteenth International Conference on Distributed Computing Systems* (May 1998).

YIN, J., ALVISI, L., DAHLIN, M., AND LIN, C. 1999a. Hierarchical cache consistency in a WAN. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS'99)*.

YIN, J., ALVISI, L., DAHLIN, M., AND LIN, C. 1999b. Volume leases to support consistency in large-scale systems. *IEEE Trans. Knowl. Data Eng.* (Feb. 1999).

YU, H., BRESLAU, L., AND SCHENKER, S. 1999. A scalable Web cache consistency architecture. In *SIGCOMM '99* (Sept. 1999).

Received September 2001; revised April 2002; accepted May 2002