Engineering Server-Driven Consistency for Large Scale Dynamic Web Services

Jian Yin, Lorenzo Alvisi, Mike Dahlin
Department of Computer Sciences
University of Texas at Austin
yin, lorenzo, dahlin@cs.utexas.edu

Arun Iyengar
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
aruni@watson.ibm.com

ABSTRACT

Many researchers have shown that server-driven consistency protocols can potentially reduce read latency. Server-driven consistency protocols are particularly attractive for large-scale dynamic web workloads because dynamically generated data can change rapidly and unpredictably. However, there have been no reports on engineering server-driven consistency for such a workload. This paper reports our experience in engineering server-driven consistency for a Sporting and Event web site hosted by IBM, one of the most popular web sites on the Internet for the duration of the event. 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 that a server sustains to send out invalidation messages when a popular object is modified. We find that it is possible to limit the size of the server's state without significant performance costs and that bursts of load can be smoothed out 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 how long servers should send invalidations to idle clients. Finally, we quantify the performance impact of caching dynamic data with server-driven consistency protocols and find that it can reduce read latency by more than 10%. We have implemented a prototype of a server-driven consistency protocol based on our findings on top of the popular Squid cache.

Keywords Web cache consistency, performance, scalability, volume lease, dynamic content

1. INTRODUCTION

Although web caching and prefetching have the potential to reduce read latency significantly, the inefficiency of 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. Unfortunately, clients may need to poll servers before returning cached objects to users even when these objects are valid. For example, in the workload that we examine, more than 20% of requests to the server are client polls to revalidate unmodified cached objects. Thus polling not only increases server and network load but also significantly increases read latency.

Many of the drawbacks of traditional client polling can be addressed with *server-driven* consistency protocols. In server-driven protocols, servers inform clients of updates. 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 [23], synthetic workloads [24], single web pages [26], and proxy workloads [14].

Server-driven consistency appears particularly attractive for large-scale workloads containing significant quantities of dynamically generated and frequently changing data. There are two reasons for this. First, in these workloads, data changes often occur at unpredictable times. Therefore, client polling is likely to result in obsolete data unless polling is done quite 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 [11] 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 that are critical to scalability and performance must be examined. This paper provides the first study of server-driven consistency for web sites serving large amounts of dynamically generated data. Our study is based on the workload 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 to dynamically generated data [12].

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 needed to keep track of the content of clients' caches may become large.
- Servers may experience bursts of load whenever they need to send invalidation messages to a large number of clients as a result of a write.

[1]The 1998 Olympic Games web site

Copyright is held by the author/owner.
WWW10, May 1-5, 2001, Hong Kong.
ACM 1-58113-348-0/01/0005.
Best paper award, WWW10.

Previous efforts to improve the scalability of server-driven consistency have primarily focused on using multicast and hierarchies to flood invalidation messages [14, 24, 26]. While 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 also will 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 increase peak server load significantly, 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 the data in the clients' cache becomes stale. Some studies argue that servers should stop notifying idle clients to reduce network, client, and server load [23], while others suggest that clients should stay synchronized with servers for days at a time to reduce latency and amortize the cost of joining a multicast channel when multicast-based systems are used [14, 26].

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 short synchronization on one hand, and the low read latency of long synchronization on the other hand. We find that for the IBM workload, there is little performance cost in guaranteeing that clients will be notified of any 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 have also proposed significantly different *re-synchronization 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 [15, 26], replaying "delayed invalidations" upon re-synchronization [23], bulk revalidation of cache contents [1], and combinations of these techniques. This study systematically compares these alternatives in the context of large-scale services. We find that for de-synchronizations that last less than one thousand seconds, delayed invalidations result in significant performance advantages.

The final issue that we address is quantifying the performance implications of caching dynamically generated data. Although the high frequency and unpredictability of updates make this 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 for the Sporting and Event web site workload.

We have implemented the lessons learned from the simulations in a prototype that runs on top of the popular Squid cache architecture [18]. 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 re-synchronization and atomic updates. Preliminary evaluation of the prototype shows that it introduces only a modest overhead.

The rest of the paper 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 number of seconds for which the returned data has been obsolete. For example, a live news site may want to guarantee that it will not supply its clients with any content that has been obsolete for more than five minutes and also to deliver most of updates within a few seconds.

Consistency algorithms use two mechanisms to meet these guarantees. Worst-case guarantees are provided using some variation of leases [8], which place an upper bound on how long a client can operate on cached data without communicating with the server. Some systems decouple average staleness from the leases' worst-case guarantees by also providing *callbacks* [10, 17] which allow servers to send invalidation messages to clients when data are modified.

For example, HTTP's traditional client polling associates a *time to live* (TTL) or an expiration time with each cached object [16]. This TTL corresponds to a per-object lease to read the object and 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 it has only one parameter, TTL, that determines both worst-case staleness and average staleness, there is no way to decouple them. 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 [14, 15, 23, 24, 26] have proposed server-driven consistency protocols. All these protocols can be understood within the general framework of *volume leases.* 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, the client requests an object lease on the object, which is usually long so that they are valid as long as the underlying objects are interested in by clients and haven't been invalidated by servers. The server registers callbacks on objects leases and revokes these object leases when the underlying objects are updated. 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 which is equal to the volume lease length. In addition to decoupling average staleness from worst-case staleness, volume lease protocols reduce server load and read latency by amortizing volume lease renewal overheads over many objects in a volume.

Volume leases are either explicitly renewed [14, 23, 24] or implicitly renewed via heartbeats [26]. Moreover, the implementation details of these protocols differ considerably. Yin et al. [23] assume a unicast network infrastructure with an optional hierarchy of consistency servers [25] and specifies explicit volume lease renewal messages by clients. Li et al. [14] assume a per-server reliable multicast channel for both invalidation and heartbeat messages. Yu et al. [26] assumes an unreliable multicast channel but bundles invalidation messages with heartbeat messages and thus ties 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 affect 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) [4] uses object dependency 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

This section describes the results of our trace-based simulation.

3.1 Methodology

We use simulation to examine various scalability and performance issues of server-driven consistency for large-scale dynamic web services.

3.1.1 *Workload*

The workload for our simulation study 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. The peak request rates for the Sporting and Event site was above 56.8 million hits per day. Overall, 12% of the requests were made to dynamically generated pages. The Sporting and Event web service was hosted on four geographically distributed web clusters; each of them served about one fourth of all requests. The web access trace used in our study contains all requests served by one of these clusters on February 19th, 1998. This trace contains about 9 million entries. Each entry contains an IP address, a time stamp, a URL, a return status, and the size of the HTTP reply. In addition, our workload includes a modification log for more than 99% of the dynamic objects generated by fast CGI scripts. This log contains 20,549 entries. Although write records of static data and dynamic data generated by SSI scripts are not available, we infer writes to these types of objects by observing changes of object sizes in our trace. This analysis generates 45,565 entries.

Our workload has one limitation. Because the trace covers only one day of activity, we can only project the long-term behavior of server-driven consistency.

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 access log and a modification log and outputs local hit rates, server load, and network bandwidth consumption. Given our limited resources, to make our study feasible we make a simplifying assumption: we simulate one cache for all requests sent from one IP address, while in reality the IP address could be a proxy masking a collection of clients or a host running several browsers. Furthermore, we only track the number of messages and total number of bytes in all messages, and network queuing and round-trip delays are not simulated.

3.2 Consistency for large-scale dynamic workloads

Previous studies have examined the performance characteristics of server-driven consistency for client cache workloads [23], synthetic workloads [25], single web pages [26], and proxy workloads [14]. In this subsection, 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 paper.

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 the fraction of reads that a client can not serve locally. There are two conditions under which a cache system has to contact the server to satisfy a read. First, the requested object is not cached. We call this a *cache miss.* Cache misses happen either when the object has not been requested before, or when the cached copy of the object is obsolete. Second, even if the requested object is cached locally, the consistency protocol may need to contact the server to determine

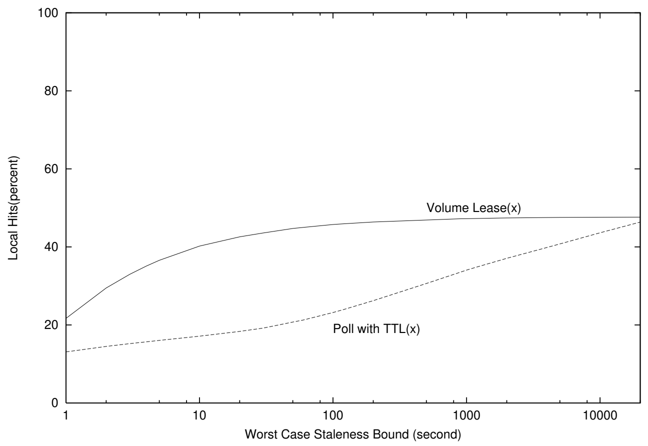

Figure 1: Local hit rates vs. worst case staleness bound for volume lease and TTL. Note that in the common case, volume lease caches are invalidated within a few seconds of an update independent of worst case staleness bounds. Note that in the common case, volume lease caches are invalidated within a few seconds of an update independent of worst case staleness bounds.

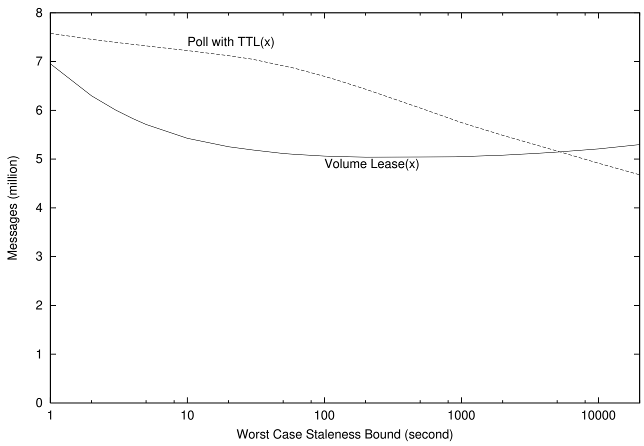

Figure 2: Number of messages vs. staleness bound for Volume Lease and TTL.

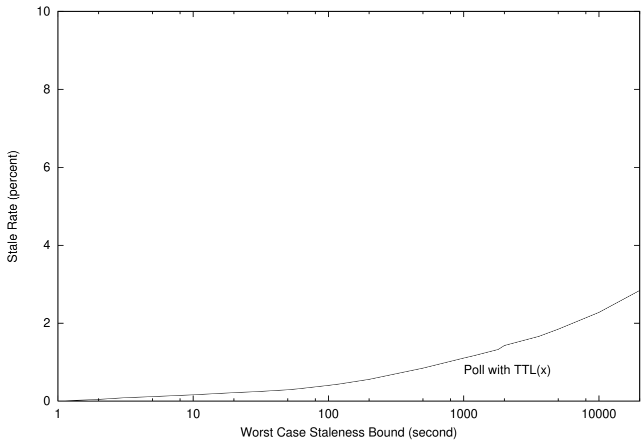

Figure 3: Stale rate vs. staleness bound for TTL.

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.

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 leases 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 4 illustrate the impact of different consistency parameters for the IBM Sporting and Event workload. In these figures, the x axis represents the worst-case staleness bounds for the volume lease algorithm; this bound corresponds 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. Considering the impact of amortizing lease renewal overheads across volumes, we see that volume leases provide larger advantages for systems that provide stronger consistency guarantees. 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 2, volume leases can provide worst-case staleness bounds of 100 seconds for about the same hit-rate and network message cost that traditional polling has for 10,000-second worst-case staleness bounds. And, as Figures 3 and 4 indicate, this comparison actually understates the advantages of volume leases because for traditional polling algorithms the number of stale reads and their average staleness increase rapidly as the worst case bound increases. In contrast, as we detail in Section 2, in the common case of no network failures, volume lease schemes can notify clients of updates within a few seconds of the update regardless of the worst-case staleness guarantees.

In Figures 6 through 8, 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 non-image objects fetched in the workload. Figure 5 shows that the non-image objects account for 67.6% of all objects and requests to non-image 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 raises to 40.9% when we exclude requests to image objects.

The dynamic and other non-image 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

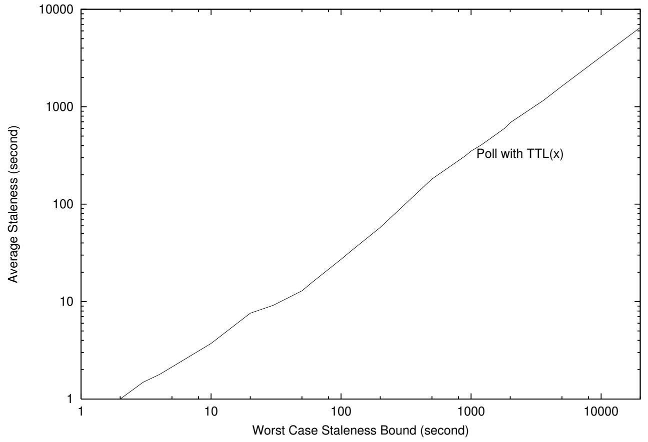

Figure 4: Average staleness vs. staleness bound for TTL.

	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

Figure 5: Classifying objects and requests according to URL types.

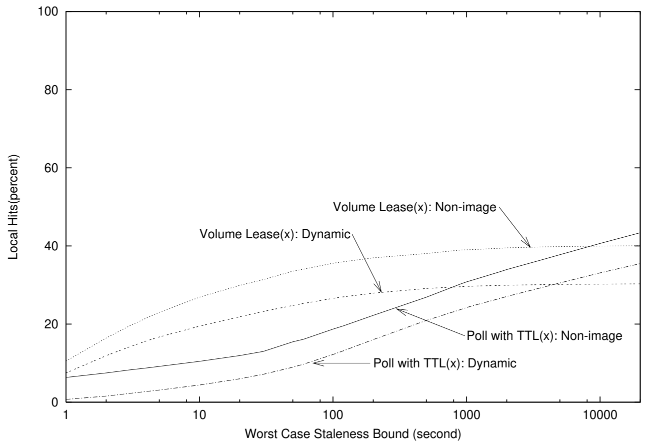

Figure 6: Local hit rates vs. staleness bound for TTL. Note that in the common case, volume lease caches are invalidated within a few seconds of an update independent of worst case staleness bounds.

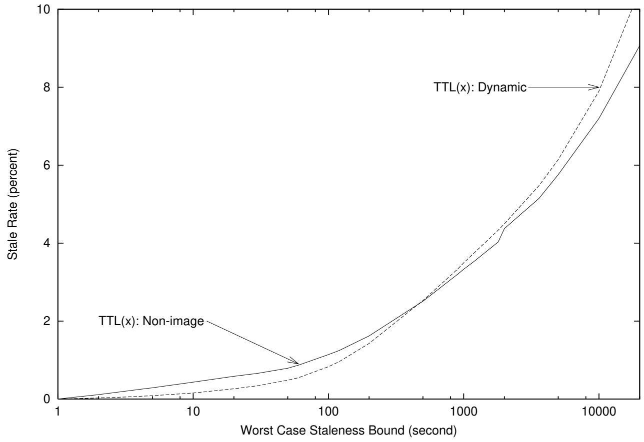

Figure 7: Stale rate vs. staleness bound for TTL.

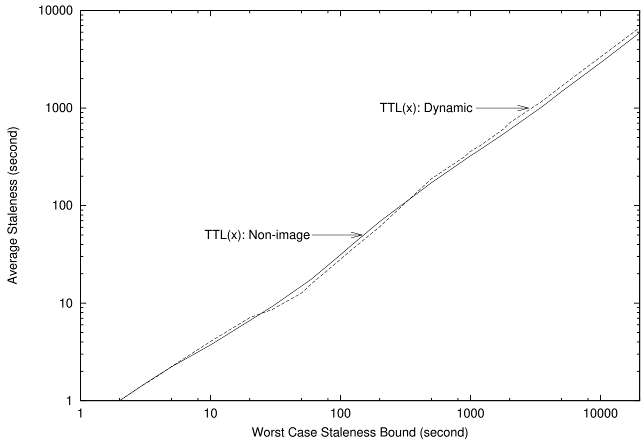

Figure 8: Average staleness vs. staleness bound for TTL.

uncachable data significantly limits achievable cache performance [21, 22], 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 non-image objects may form the bottleneck in response time since they must often be fetched before the images and static elements may be 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, three factors limit the cachability of dynamically generated data: (1) the need to determine which objects must be invalidated or updated when underlying data (e.g. databases) change [4], (2) the need for an efficient cache consistency protocol, and (3) the inherent limits to caching that arise when data change rapidly. As detailed in Section 2, our system provides an efficient method for identifying web pages that must be invalidated when underlying data change. And, as Figure 6 through Figure 8 indicate, volume lease strategies can significantly increase the hit rate for both dynamic pages and for the "bottleneck" non-image pages.

Finally, the figures quantify the third limitation. Although one might worry that dynamic objects change so quickly that caching them would be ineffective, the hit rate difference is relatively small. For long leases, hit rates for dynamic objects are slightly lower than for all objects. As many as 25% of reads to dynamically-generated data can be returned locally, which increases the local hit rate for non-image data by 10%. Since the local hit rate of non-image 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 updated.

Notice that Figure 6 shows that dynamic pages and non-image 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 non-image pages are particularly likely to cause a miss due to volume lease renewal because they are often the first elements fetched when a burst of associated objects are fetched in a group. In the next subsection, we examine techniques for reducing the hit rate impact of short worst-case guarantees.

3.3 Prefetching 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 piggyback a volume lease renewal request on its next message to the server [23], 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 [26].

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, increases cache hit rates, but increases network costs, server load, and client load.

Previous studies have assumed extreme positions regarding prefetching volume lease renewals. Yin et al. [25] assumed that volume lease renewals are piggybacked 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 [14] 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 once.

In Figure 9 and Figure 10, 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, one could increase the volume lease length by a factor of K. Alternatively, one 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 9, 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

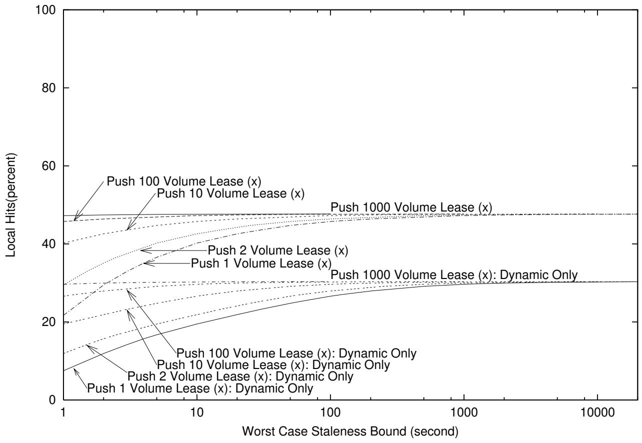

Figure 9: Prefetching volume leases to reduce read latency.

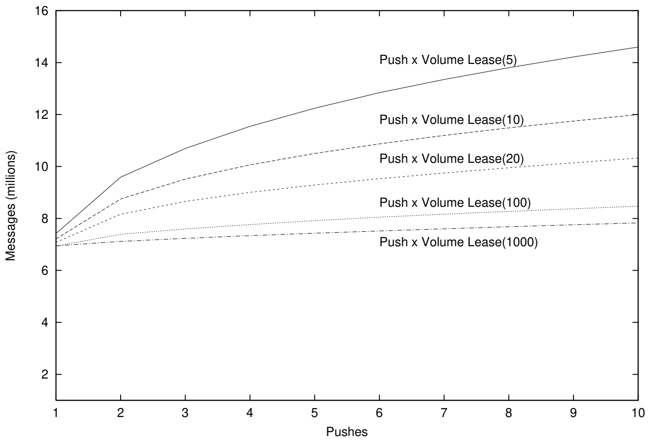

Figure 10: Cost of pushing volume leases.

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 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 linearly 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 [5].

In general, we conclude that although previous studies have examined extreme assumptions for prefetching [26, 14], 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.

3.4 Scalability

Workloads like the IBM Sporting and Event workload 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 times 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 times the number of objects simultaneously modified. For both memory consumption and bursts of load, if uncontrolled, this worst case behavior could prevent deployment for large web services such as the IBM Sporting and Event trace.

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 [25] 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 [26] on a per-object basis [24]. 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 [14] 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* [23] 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 invalidation* places invalidation messages in a separate send queue from replies to client requests and sends 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 clients 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. Tewari et al. [6] 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 [23]. When a 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 subsection 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 [6].

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

3.4.1 Server callback state

Figure 11 shows the number of object leases stored as a function of elapsed time in the trace. 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 [2, 21, 22], 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 proto-

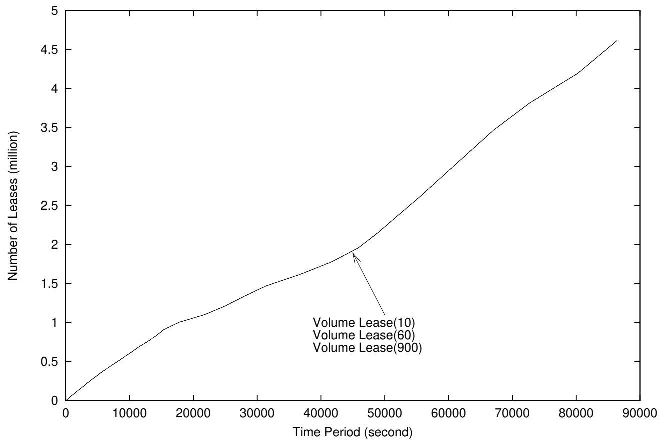

Figure 11: Callback state increases with elapsed time.

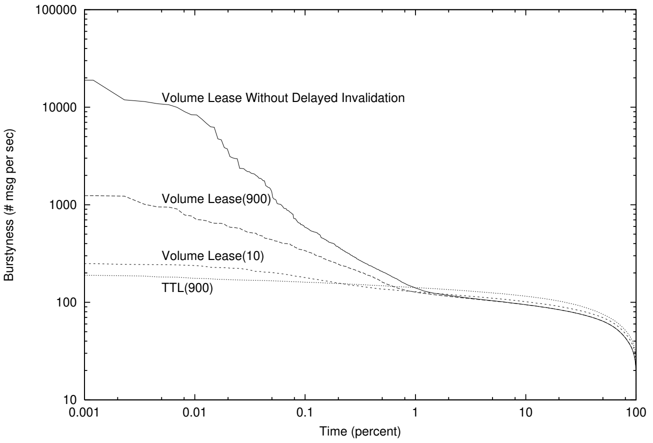

Figure 12: Distribution of invalidation burstiness. A point of (x, y) means that the server generates at least y messages per second during x percentage of time.

type 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% 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 overheads, 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.

3.4.2 Bursts of load

Figure 12 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

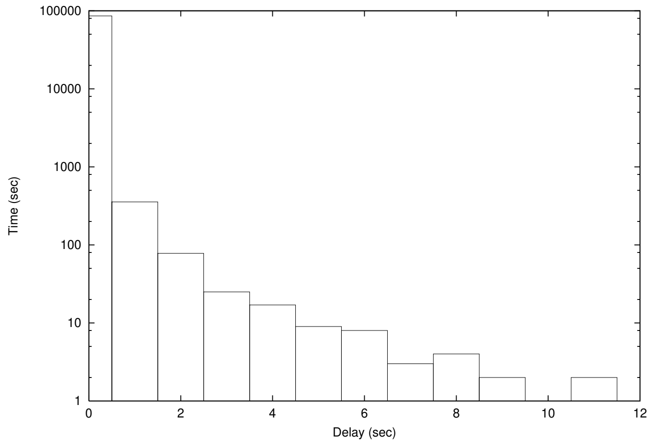

Figure 13: Delaying invalidation to smooth out server load burstiness for Volume Lease (900).

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 periods; but even with delayed invalidations, peak load is increased by a factor of 6 for 900 second 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 13, we limit the server message rate to 200 messages per second, which is approximately the average load of TTL, and 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 3 showed the average staleness for the traditional TTL polling protocol. The data in Figure 13 allow 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 13 suggest 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

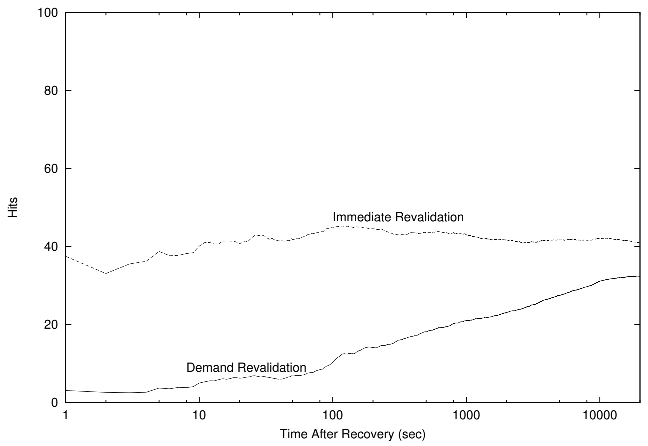

Figure 14: Hit rates after recovery for a disconnection of 1 second.

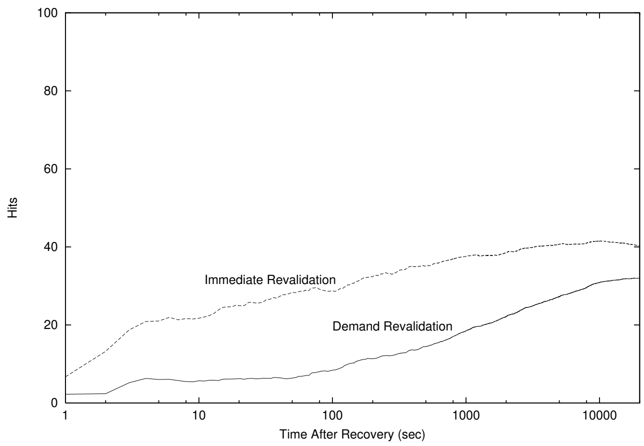

Figure 15: Hit rates after recovery for a disconnection of 1000 seconds.

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 [26, 25].

3.5 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 because of 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 3.4.

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 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 [15], while others have assumed that immediate revalidation is justified [25]. In this study, we quantitatively evaluate these two options and the middle-ground, background revalidation, for this workload.

Figures 14 and 15 show that immediate revalidation achieves higher average local hit rates than demand revalidation.

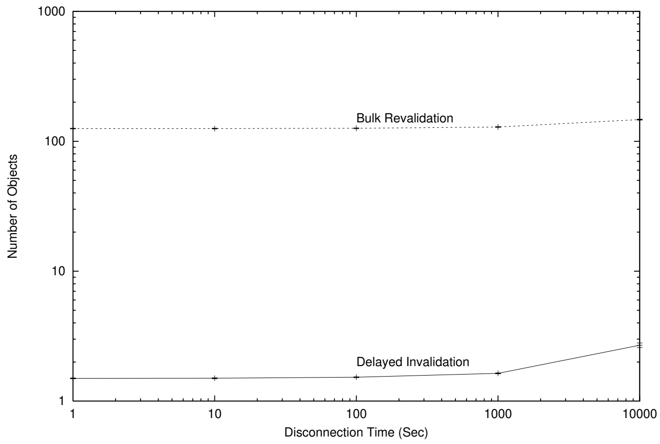

Figure 16: Resynchronization cost for bulk revalidation and delay invalidation.

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 (not shown) would equal those of demand revalidation immediately after reconnection and would increase to those of immediate revalidation as 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 14 and 15, 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 messages 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 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 piggybacks the buffered invalidations on the reply message granting the client a volume lease. The client applies these buffered invalidations to resynchronize with the 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 16 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 disconnection, 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, system should support background revalidation rather than relying solely on immediate revalidation.

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 a client part which manages consistency information on locally cached objects to satisfy reads. Servers maintain a per-object version number as well as a 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 [7].

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 [25]. 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 [4]. 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 [25]. Third, reverse-proxy caching of dynamically generated data at the server can achieve nearly 100% hit rates and can dramatically reduce server load [3]. By implementing our system as a hierarchy, we make it easy to gain these advantages. Further, if a multi-level hierarchy is used (such as the Squid regional proxies [18] or a cache mesh [20]), 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 thereby 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 re-synchronize after servers discard or lose client state.

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 can not 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 [16]. 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 [26].

Second, the queue of unacknowledged invalidations provides the basis for an important performance optimization: delayed invalidation [23]. 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 [4].

The system also implements a protocol for re-synchronizing 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* [23] 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 web caching industry: the first semi-annual web caching bake-off workload. Our testbed includes four computers. Two of them are running the workload. The consistency server and the consistency client, which are the Squid proxies augmented with server-driven consistency, are running 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 web or proxy server that deliver data, and to quantify the computing resources (CPU, memory) needed to maintain server-driven consistency for the major Sporting and Event Web Sites hosted by IBM.

5. RELATED WORK

There have been several studies on cache consistency in wide-area networks. Gwertzman and Seltzer [9] simulate various client-polling and callback-based consistency protocols and conclude that adaptive TTL can provide good performance for applications in which it is acceptable to return about 4% stale reads. Liu and Cao [15] find that, although it is possible to implement server-driven consistency with an overhead comparable with that of client-driven consistency, scalability is an issue. In particular, they point to (i) the bursts of server load caused by invalidations sent in response to writes to popular objects, and (ii) the growth of the state that the server maintains to track the status of its clients' caches.

To address these concerns, several recent studies have explored how to improve scalability by using multicast. Yu et al. [26] propose a scalable cache consistency architecture that integrates the ideas of invalidation, volume lease and unreliable imprecise multicast. They use synthetic workloads of single pages and focus their evaluation on network performance of server-driven consistency. Li and Cheriton [14] propose to use reliable precise multicast to deliver invalidations and updates for frequently modified objects. The workloads in their study include client traces, proxy traces, and synthetic traces. What is unique to our study is to examine server-driven consistency from the perspective of large scale dynamic web services, and to address the challenge of scalability through techniques that can be deployed in both multicast and unicast environments.

Other studies examine specific design optimizations for consistency protocols. Duvvuri et al. [6] examine adapting object leases to reduce server state and messages. These techniques can also be employed in our protocol to improve scalability. Krishnamurthy and Wills [13] examine ways to improve polling-based consistency by piggybacking optional invalidation messages on other traffic between a client and server. While their study doesn't provide the worst case staleness bounds required by our workload, several techniques used 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 piggybacking them on top of other traffic between servers and clients. In the same paper, they also propose 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.

Finally, we observe that cache consistency protocols have long been studied for distributed file systems [19]. Both the notion of invalidation and that of leases for fault tolerance have been examined in this context [8], as well as methods for fast re-synchronization of callback state [1].

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 because objects served by such applications change unpredictably and frequently, and because 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 this workload there is little performance benefit of 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.

Vitae

Jian Yin is currently a Ph.D. student in Computer Science at University of Texas, Austin. His research interesting includes distributed computing, operating systems and computer networks.

Lorenzo Alvisi is an Assistant Professor and Faculty Fellow in the Department of Computer Sciences at the University of Texas at Austin. His research interests are mainly in reliable distributed computing. He holds M.S. (1994) and Ph.D. (1996) degrees in Computer Science from Cornell University and a Laurea *summa cum laude* in Physics (1987) from the University of Bologna, Italy. Dr. Alvisi is the recipient of an Alfred P. Sloan Research Fellowship and of an NSF CAREER award.

Mike Dahlin is an Assistant Professor and Faculty Fellow in the Department of Computer Sciences at the University of Texas at Austin as well as an Alfred P. Sloan Research Fellow. He received the BS degree *summa cum laude* in electrical engineering from Rice University in 1991 and the MS and PhD degrees in computer science from the University of California at Berkeley in 1993 and 1995. Dr. Dahlin's primary research interests are in distributed I/O systems and operating systems.

Arun Iyengar does research into Web performance, electronic commerce, and parallel processing at IBM's T.J. Watson Research Center. He has a PhD in Computer Science from the Massachusetts Institute of Technology. He is the National Delegate representing the IEEE Computer Society to IFIP Technical Committee 6 on Communication Systems, the Chair of IFIP Working Group 6.4 on Internet Applications Engineering, and the Vice Chair for the IEEE Computer Society's Technical Committee on the Internet. He will be Program Co-Chair for the 2002 International World Wide Web Conference.

Acknowledgements

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.

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.

7. REFERENCES

[1] M. Baker. *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California at Berkeley, 1994.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. On the Implications of Zipf's Law for Web Caching. Technical Report 1371, University of Wisconsin, April 1998.

[3] J. Challenger, P. Dantzig, and A. Iyengar. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *Proceedings of ACM/IEEE SC98*, November 1998.

[4] J. Challenger, A. Iyengar, and P. Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE INFOCOM'99*, March 1999.

[5] B. Chandra, M. Dahlin, L. Gao, A. Khoja, A. Khoja, A. Nayate, and A. Sewani. Resource management for scalable disconnected access to web services. In *WWW10*, 2001.

[6] V. Duvvuri, P. Shenoy, and R. Tewariy. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Infocomm*, 1999.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Internet-Draft draft-ietf-http-v11-spec-07, HTTP Working Group, August 1996.

[8] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[9] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[10] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

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

[12] A. Iyengar, M. Squillante, and L. Zhang. Analysis and characterization of large-scale web server access patterns and performance. In *World Wide Web*, June 1999.

[13] Balachander Krishnamurthy and Craig E. Wills. Piggyback Server Invalidation for Proxy Cache Coherency. In *Seventh International World Wide Web Conference*, pages 185–193, 1998.

[14] D. Li and D. R. Cheriton. Scalable Web Caching of Frequently Updated Objects using Reliable Multicast. In *2nd USENIX Symposium on Internet Technologies and Systems (USITS'99)*, October 1999.

[15] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.

[16] J. Mogul. A Design for Caching in HTTP 1.1 Preliminary Draft. Technical report, Internet Engineering Task Force (IETF), January 1996. Work in Progress.

[17] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[18] http://www.squid-cache.org/.

[19] V. Srinivasan and J. Mogul. Spritely NFS: Experiments with Cache Consistency Protocols. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 45–57, December 1989.

[20] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proceedings of the Nineteenth International Conference on Distributed Computing Systems*, May 1999.

[21] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-Based Analysis of Web-Object Sharing and Caching. In *Proc. of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[22] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative web proxy caching. In *Proc. of the Seventeenth ACM Symposium on Operating Systems Principles*, December 1999.

[23] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of the Eighteenth International Conference on Distributed Computing Systems*, May 1998.

[24] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *2nd USENIX Symposium on Internet Technologies and Systems (USITS'99)*, 1999.

[25] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases to Support Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, February 1999.

[26] H. Yu, L. Breslau, and S. Schenker. A Scalable Web Cache Consistency Architecture. In *SIGCOMM '99*, September 1999.