A Scalable Web Cache Consistency Architecture*

Haobo Yu
USC/Information Sciences Institute
4676 Admiralty Way Suite 1001
Marina del Rey, CA 90034
haoboy@isi.edu

Lee Breslau
AT&T Labs-Research
75 Willow Road
Menlo Park, CA 94025
breslau@research.att.com

Scott Shenker
International Computer Science Institute
1947 Center Street
Berkeley, CA 94704
shenker@icsi.berkeley.edu

Abstract

The rapid increase in web usage has led to dramatically increased loads on the network infrastructure and on individual web servers. To ameliorate these mounting burdens, there has been much recent interest in web caching architectures and algorithms. Web caching reduces network load, server load, and the latency of responses. However, web caching has the disadvantage that the pages returned to clients by caches may be *stale*, in that they may not be consistent with the version currently on the server. In this paper we describe a scalable web cache consistency architecture that provides fairly tight bounds on the staleness of pages. Our architecture borrows heavily from the literature, and can best be described as an *invalidation* approach made scalable by using a caching hierarchy and application-level multicast routing to convey the invalidations. We evaluate this design with calculations and simulations, and compare it to several other approaches.

1 Introduction

The world-wide-web has become an important component of the global information infrastructure. The rapid increase of web usage has imposed a heavy load on the network and server infrastructure, and significant delays are not uncommon. To mitigate the effects of this increased usage, there has been much recent interest in developing and deploying techniques for web caching (see, for example, [8, 29, 33] and references therein). Web caching has several beneficial effects: it lowers the load on servers, reduces the overall network bandwidth required, and lowers the latency of responses.

However, web caching does have (at least) one serious disadvantage. If a page has been modified after being stored in a cache, the version of the page delivered to the requesting client[1] may be inconsistent with the server's version of that page. We call such inconsistent pages *stale*, and call consistent pages *fresh*; the degree of staleness is the delay between when the page was changed on the server and when the previous version was delivered. To make this precise, consider the version of the page that was delivered to the client. Let $t = M$ be the time the delivered version was first rendered invalid by being modified at the server. Let $t = R$ be the time the cache responds to the client's request for that page. We then define the staleness[2] to be $\max(0, R - M)$.

For many pages, being significantly stale is not a serious problem. For some pages, however, clients may care a great deal if the pages are substantially stale. For instance, it is clear that pages devoted to current news stories (*e.g.*, CNN) should be as fresh as possible. Other examples of pages that are sensitive to being stale – we will call such pages *perishable* – are catalogs, product information, and code distribution pages. Perishable pages need not have zero staleness (*i.e.*, a news page could be a minute or so out of date without serious harm), but they should not be significantly stale.

One could most easily meet the freshness needs of perishable pages by circumventing caching; this can be accomplished by marking pages as uncacheable, or by merely expecting users to manually hit the "reload" button. However, since some perishable pages are likely to be quite popular – news sites in particular – one would like to ensure the relative freshness for these pages while retaining the advantages of caching. Because there is a finite latency between the server and the cache, it is impossible to guarantee absolute freshness (*i.e.*, true consistency between what the cache

*We would like to thank Mike Spreitzer and Marvin Theimer for their collaboration in the early stages of this work. They are responsible for many of the key ideas that inspired the design described in this paper. We would also like to thank Pei Cao for several helpful conversations. This research was supported by the Defense Advanced Research Projects Agency (DARPA) through the VINT project at USC/ISI under DARPA grant DABT63-96-C-0054 and at Xerox PARC under DARPA grant DABT63-96-C-0105. Lee Breslau and Scott Shenker were at Xerox PARC while this work was carried out.

[1] We use the term *client* to refer to a browser or other user process at the end host that generates requests for pages.

[2] Note that even if the staleness is zero by this definition, the page may be out of date when it actually arrives at the client due to changes made at the server while the data was in transit to the client; this, however, is not a problem with the caching infrastructure – since this source of inconsistency occurs even if the request was sent directly from the client to the server rather than being handled by a cache – and so we do not consider it part of being stale.

delivers and the current version at the server) without instituting write-locking on servers.[3] While write-locking is sensible for keeping file systems consistent, it makes less sense for web pages,[4] since write-locking merely masks the underlying reality that the content delivered is different than the content the server thinks is most current. Thus, the most practical goal is to merely limit the degree of staleness – *i.e.*, to achieve *loose consistency* – rather than trying to achieve strict consistency. We believe such loose-consistency guarantees should be sufficient for the vast majority of perishable pages.

In this paper we focus on the design of a scalable web cache consistency architecture that meets this goal. Our design retains the benefits of web caching (as listed above), while providing fairly tight limits on the degree of staleness of delivered pages. Of course, as we review in Section 2, there has been much previous work on techniques to achieve various degrees of consistency for web pages; the architecture we propose combines many of the features of these previous proposals, melding them together in a scalable fashion. Moreover, our proposal can easily be extended to support the *pushing* of data, in which modified pages are sent to caches even before clients have requested them.

Since we envision, at least initially, that a small fraction of pages are perishable, our design can be restricted to those pages that are deemed by the server to be perishable; that is, our proposal does not change how caches handle nonperishable pages and only modifies how caches handle perishable ones. Our design does make use of a caching hierarchy. However, this hierarchy can be replaced by a cache mesh, as we describe in Section 3.2.

We evaluate this design in two ways. We first investigate its behavior analytically in a very simplified setting, and then present simulation results in a somewhat more realistic setting. In both cases we compare our proposed design against several other schemes.

This paper is addressing the question of design, not of deployment. That is, we are asking: can one design such a scalable web consistency architecture? We are most definitely not addressing the question of whether such an architecture, once designed, should be deployed (although we discuss this question briefly in Section 7) since the question of deployment is a complicated cost/benefit tradeoff involving many nontechnical factors, such as the future usage of the web and the economics of the ISP business. However, deployment can only occur if a scalable web consistency architecture exists, and our contribution here is to demonstrate that such a design is indeed possible.

This paper has 7 sections. We begin in Section 2 by reviewing several of the previous approaches to ensuring consistency. We present our approach in Section 3, starting with our basic scheme and then adding in the ability to push pages. We then evaluate this design analytically in Section 4 and through simulations in Section 5. We discuss additional design issues in Section 6, and conclude with a brief discussion of our results in Section 7. We include estimates of cache state and network bandwidth requirements in an appendix.

[3] If the cache receives a request for a page, obtains a fresh version of the page from the server, and then delivers the page to the client, the page would still be stale when delivered if the page was modified on the server between the time the server sent the page to the cache and when it arrived at the cache. The only way to avoid this would be to write-lock the page during the interval while the page was being delivered to the cache.

[4] The crucial distinction between file systems and web pages, in terms of the role of write-locking, is that web pages have a single logical writer (the hosting server) whereas files have many logical writers (they can be written from many hosts). Merging multiple writers requires strict consistency, whereas handling multiple readers does not.

2 Previous Approaches

All web caching proposals attempt to achieve some degree of consistency, but the approach taken to achieve consistency depends greatly on the degree of consistency desired. In this section we briefly review three basic approaches to consistency. These approaches function both as inspirations for our proposed architecture and also as benchmarks against which we evaluate our design in Sections 4 and 5.

2.1 Time-To-Live

The simplest way to achieve some limited form of consistency is to associate a time-to-live with each page. When a request arrives at a cache after the TTL for the requested page has expired, the cache sends an If-Modified-Since (IMS) message to the server (or parent cache) to determine if the version held by the cache is still valid. If the TTL is fixed then the staleness is bounded by this TTL (plus the latency between the server and the cache). Setting small values of the TTL provides fairly tight consistency guarantees, but also mitigates against some of the benefits of web caching, since many IMS requests will be forwarded to the server even though the page is still valid. The limit of TTL=0 generates an IMS for every request, thereby guaranteeing no staleness; we call this scheme *poll-always*.

It has long been known that files exhibit the property that the longer they have gone unmodified, the longer they are likely to go unmodified [3, 4]. In [7] this insight was used to develop an adaptive TTL scheme in which the TTL is set, at the first request after each TTL expiration, to be proportional to the page's age (current time minus the last modification time); the algorithm takes, as a parameter, the constant of proportionality (called the update threshold in [15]) used to update the TTL. However, adaptive TTL schemes do not give an upper bound on the staleness of a page, since the TTL can grow without bound.

2.2 Invalidation

In the TTL approach, the cache can only guess as to whether a page is still valid. A very different approach to consistency requires servers to send explicit *invalidation* signals to caches when pages are modified. The invalidation approach is most easily explained, as we do below, when considering only the interaction between a server and a client without caches as intermediaries; later, when presenting our design, we will discuss the role of invalidations in the presence of proxy caches.

In its simplest incarnation, an invalidation scheme works as follows: each server keeps track of all clients who have requested a particular page and then, whenever that page changes, notifies those clients. We say that servers have an *invalidation contract* with the clients so that clients are assured that they will be informed of any changes to pages they have read.

While invalidation schemes are effective in limiting staleness, they incur the cost of requiring the server to keep state on every client of each page. Thus, this approach does not scale well in the limit of many readers per page; both the state required to store the list of readers, and the OS and network burden of having to contact every reader of a page when it changes, grow linearly in the number of readers.[5]

This scaling problem can be overcome by using multicast to transmit the invalidations. By assigning a multicast group to each page, and having clients join the groups associated with the pages they have accessed, the burden on the server is greatly reduced; the server need not keep any readership state, and need only send a single invalidation message to inform the group of any page modifications. Such an approach is described in [28], and the somewhat related idea of *pushing* content (rather than sending invalidations) via multicast is described in [23, 27, 28]. However, while multicast solves the scaling problems at the server, it creates (following the law of conservation of difficulty) another one at the routers. The state required by such schemes in routers is substantial, easily on the order of hundreds of thousands of addresses (judging by the proxy traces in [19]); this is certainly too much for many currently deployed routers. Moreover, the rate at which clients would be joining and leaving multicast groups, as they read and discard pages, will likely create an unscalable overhead on the routing infrastructure [17].

A recent proposal [9] includes information about related pages in responses to page requests; this information may include invalidations and delta-encoded page updates. It can be used to greatly improve consistency on average but it does not provide staleness assurances.

2.3 Lease

The lease approach to consistency combines features of the TTL and invalidation approaches; see [13] for the basic reference on leases in file systems, and see [31] for applications of these ideas to web caching. In the simplest version of this approach, whenever a cache stores a page, it requests a *lease* from the server. Whenever a page changes, the server notifies all caches who hold a valid lease of the page; the invalidation contract applies only while the lease is valid. If a cache receives a request for a page with an expired lease, it renews the page's lease by sending an IMS to the server before responding to the request. While the lease is valid, the approach is exactly like invalidation, but the expiration of leases resembles the TTL approach. One wants to choose the length of the lease so that the number of readers holding valid leases remains reasonably small when writes are made, but most reads occur while the lease is still valid. In distributed file systems, leases are usually short (seconds or minutes) [4], but in the Web context using overly short leases makes the scheme roughly equivalent to TTL.

Yin *et al.* [31] presented two volume lease algorithms aimed at reducing validation traffic of short leases. They assign a long lease to every page, and a short lease to sets of pages called *volumes*. A cache must renew a lease whenever either the page lease or the volume lease expires. The advantage of this approach is that the overhead of renewing the short leases is amortized over the many pages in a volume.

3 Our Approach

Our approach borrows quite freely from these previous approaches. It is based primarily on multicast-based invalidations, but avoids the scalability problem by using a hierarchy of caches.[6] The multicast groups are associated with caches, not pages, and the caches send heartbeats to each other that are the equivalent of cache-to-cache volume leases. In contrast to a previous use of volume leases [31], the unit of our lease is all pages in a cache, instead of a single page or page group. Caches maintain a server table in order to locate where servers are attached to the hierarchy. Invalidation messages for a page, which may be sent both up and down the hierarchy, are filtered so as to limit the scope of distribution. Client requests are forwarded through the caching hierarchy to the server or to the first cache containing a valid copy of the requested page.[7] We first describe the basic protocol and then describe how to add pushing to the architecture.

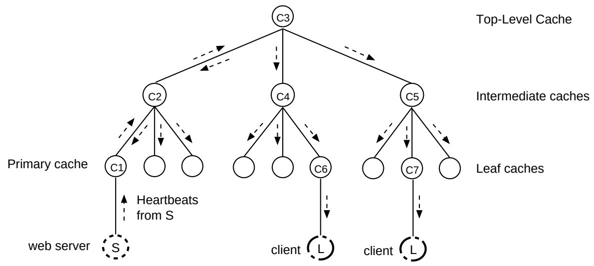

Figure 1: Example of a single multicast caching hierarchy. The arrows indicate the propagation directions of heartbeats.

3.1 Simple Description of Protocol

To describe the algorithm most compactly we first consider the special case where all caches are infinite, all pages are part of this consistency architecture, there is a single stable caching hierarchy with all caches having synchronized clocks, and no caches fail (although we make no assumption about the reliability of communication between caches). Aspects of the design associated with more realistic settings are addressed in Section 6. The descriptions given here (and in Section 6) are rather cursory and informal; a more complete and detailed description of the entire protocol can be found in [32].

Hierarchy The caching hierarchy (Figure 1) is glued together by multicast. Each parent cache *owns* a unique multicast group, in the sense that it is responsible for allocating the group address, and it is the only sender in the group. Each child cache joins the group owned by its parent. Thus, parents need not know who their children are, and children can choose their parents freely as long as cycles are prevented, and that is easily accomplished with a convention on assigning each cache to one of a few levels – *e.g.*, leaf caches, intermediate caches, and top-level caches – and requiring that parents always outrank their children. We do not address the issue of hierarchy establishment and maintenance; see [25] for one approach to these issues. We discuss alternatives to the use of a hierarchy later in this section.

Heartbeats The hierarchy is kept alive by *heartbeats*. Each group owner sends out a periodic heartbeat message to its

[5] Also, in the oversimplified version just described, there are robustness problems when servers lose their state or when network partitions occur. These robustness issues can be addressed, as we shall see in Section 3.

[6] We discuss alternatives to a hierarchy in Section 3.2.

[7] An extension that allows requests to bypass the caching hierarchy, thus reducing response latency, is described in Section 6.

associated multicast group; let τ be the time period between heartbeats. The heartbeat functions as a volume lease of length T to its children; this lease applies to all pages sent by the cache to its children. The time period of the lease starts when the message was generated (reflected in its timestamp), not when it was received. Typically τ will be significantly less than T ($\frac{T}{\tau} = 5$ in our simulations) so that if one or a few consecutive heartbeat are lost – which is a possibility since we are not sending them reliably – the lease won't expire unnecessarily. Each child cache compares the current time to the last heartbeat's timestamp (or, more precisely, the highest timestamp among all received heartbeats). If this time gap ever reaches T then the lease on all pages from that server expires and all such pages are marked as invalid.

Invalidations On top of these heartbeats we piggyback explicit invalidations. We need only invalidate pages that have been requested (by a client or another cache) after they were last rendered invalid; we call these *read pages*. Each heartbeat message contains a list of all read pages that have been rendered invalid at the parent cache within the last time period T. Thus, if a read page is rendered invalid at the parent cache at time $t = 0$ then by time $t = T$ each child cache has either received a heartbeat with an invalidation for that page, or has expired the lease from that parent cache (and thereby rendered the page invalid). A child cache that had a previously valid copy of the page will mark it invalid and propagate the invalidation if and only if the page was previously read; otherwise it ignores the invalidation.

Attaching Servers In addition to heartbeats going down the hierarchy, we also have a set of heartbeats traveling up the hierarchy from servers towards the top-level cache. To describe this, we first define how servers attach to the hierarchy. Each web server is attached to a cache (not necessarily a leaf cache) in the hierarchy, which we call the server's *primary cache*. Upon attaching, each server must reliably unicast a JOIN message to its primary cache. This message is forwarded upwards (by each cache to its parent cache) via reliable unicast until it reaches the top-level cache. We say that the parent cache *sources* a server from a child cache if it receives that server's JOIN message from a child cache (and has not received a LEAVE message for that server; we define LEAVE messages below). Each cache has a listing of those servers it sources (*i.e.*, those servers attached below it); we call this list the *server routing table* (Figure 2). If a cache does not source a server, we say that its server routing table entry for the server points to the parent cache. Note that the top-level cache knows about all servers attached in the hierarchy.

Servers send (via unreliable unicast) periodic heartbeats to their primary cache, also piggybacking invalidations of any read pages as we described above. Similarly, every child cache who sources at least one server must unicast heartbeats to its parent, along with piggybacked invalidations. A cache can ignore invalidations for unread pages (pages that are not in residence in the cache are automatically considered unread pages). Invalidations are thus propagated from the server to every cache from which the page has been read. If a cache C1 is closer to the server than cache C2 along this propagation path, we call C1 an *upstream* cache (compared with C2); otherwise, we call it a *downstream* cache. Each upstream cache is said to maintain an invalidation contract with its immediate downstream cache(s) for any page that has been read by a downstream cache.

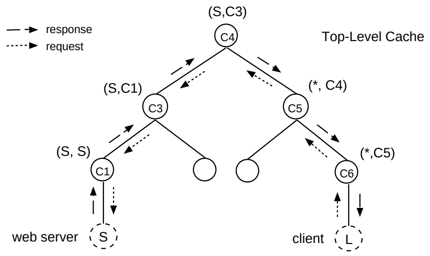

Figure 2: An example of server routing table setup. Routing table entries are shown in parentheses next to each cache. Each entry is in the form (S,C), where C is the next hop cache towards server S. A "*" indicates a default entry. The arrows show how requests flow from a client to the server, and how responses flow in the reverse direction.

When a time period T has passed without cache C1 hearing from cache C2 from whom it sources a server, cache C2 and all the servers sourced from cache C2 are removed from cache C1's server routing table. Cache C1 then sends a LEAVE message to its parents and children, notifying them that those servers are no longer sourced from cache C1, and therefore all of the pages from those servers should be considered invalid. (More details are described in Section 6.) LEAVE messages are a form of invalidation, and are included in the heartbeats (rather than being sent reliably).

Handling Requests We now describe how client requests are handled, as illustrated in Figure 2. Clients can attach to any cache in the hierarchy; we call this the client's primary cache. In particular, a client can attach to its own local cache (*i.e.*, the browser's cache) and then use a nearby proxy cache as a parent cache (as they typically do now). When a client requests a page, it sends the request to its primary cache. The primary cache, and recursively all caches the request visits, first checks to see if the page is resident in the cache. If it is not, then the cache forwards the request to the next cache designated by the server routing table. When the request is fulfilled, by either the originating server or by some intermediary cache, the response takes the reverse path through the caching hierarchy towards the client. The reverse path is automatically set up because every cache has an open HTTP connection to the the requester before it responds.

3.2 Discussion

We list below three important properties of this scheme (proofs can be found in [32]). As stated above we assume a stable hierarchy, synchronized clocks, and the proper functioning of caches, but make no assumptions about the reliability of communication.

Property 1 *If there are no invalidations in transit or waiting to be sent, then if a cache C in the hierarchy has a page P marked as invalid, then no downstream cache considers P valid (i.e., it is either invalid or not in residence).*

Property 2 *When a cache C receives an invalidation for a page P marked as invalid, it may safely discard the invalidation without affecting the resulting state of all downstream caches.*

Property 3 *Assume that each cache uses the same timeout period T. Consider a server S1, a client attached to cache C2 requesting the page, and assume that there are H cache-hops between S1 and C2. Then the maximal staleness of a page hosted on S1 delivered to the client is HT.*

Property 2 follows directly from Property 1. Together they allow us to reduce redundant invalidation traffic. Property 3 sets an upper bound of page staleness for every cache in the hierarchy.

We believe this scheme is a scalable approach to web cache consistency, and is essentially an application-level version of multicast distribution of invalidations. To clarify this analogy, consider a design which has a multicast group per version of a page and in which requesting the page is equivalent to joining the group for that version of the page; when a version of the page is rendered invalid, invalidations are sent to the group associated with that version, and multicast routing makes sure the invalidation ends up at every client and cache that has that version of the page. This is exactly what happens in our design, except that our design has no explicit notion of groups, and all "routing" of invalidations is done by the caches keeping track of the read pages and forwarding invalidations for those read pages.[8] The use of heartbeats facilitates robustness and failure detection.

Before proceeding, we elaborate on the use of caching hierarchies in this design. Our protocol requires application level routing to route messages among clients, servers and caches. Cache hierarchies provide a simple way to do this, but there are other possible cache organizations. The only requirement is that the cache organization provides source-independent and acyclic application-level routing of messages between servers and caches. That is, there must be a single (application-level) path between a cache and a server, and when superimposed, the set of paths to a server from all caches is loop-free. A cache mesh, in addition to a cache hierarchy, can also accomplish this goal.

We see the tradeoff between a mesh and a hierarchy as follows. The hierarchy provides a simple mechanism to reduce the (application-level) routing state in caches. This is particularly true at the leaves of the hierarchy, since a cache only needs explicit information about servers below it in a hierarchy. A mesh, on the other hand, eliminates the bottleneck of a root cache at the expense of increased state at other caches. Since a mesh organization has neither implicit information about cache location, as is provided by the default parent entry in a hierarchy, nor aggregable cache address allocation as is available in IP routing [12], reducing the routing state at caches is difficult. In addition, the lack of aggregation implies increased processing and communication overhead to establish and maintain the routing state. For example, information about changes in server state must be propagated to all caches in the mesh.

Given this tradeoff, we see the choice of a hierarchy as reasonable for the following reasons. First, it places the largest burden on a smaller number of caches (root or other high level caches) that are most easily engineered to meet this load. Engineering all caches to meet the state requirements of a mesh is likely a more difficult problem. Second, estimates of the load on root caches, provided in Appendix A, indicate that the load on the root caches is manageable. Therefore, in this paper we describe our design in the context of a cache hierarchy, nevertheless, it works for both meshes and hierarchies. Moreover, hybrid approaches are possible; for instance, leaf caches could be attached to a general mesh topology, reducing the state requirements on leaves and reducing traffic in the core.

Above we assumed an ideal environment for the sake of discussion, however, our design is capable of handling various issues related to more realistic contexts: *e.g.*, clock skew, finite cache, failure recovery, incremental deployment, etc. We address these issues briefly in Section 6, and refer the interested reader to [32] for additional details.

3.3 Adding Push to the Architecture

There is one aspect of performance that caching cannot improve: the latency suffered by the first request to an unread page. The concept of *pushing* data from the server to caches is of some interest, precisely because it reduces this first access latency so dramatically. While pushing is not directly related to caching, it fits within our architecture and addresses an important web performance issue, so we have included it in our design. We now briefly present a simple proposal for pushing. One only wants to push popular pages that are likely to be read before they are modified again. Servers could identify pages that are sufficiently popular that they should be pushed, or clients could request certain pages be pushed (see [32] for designs of that flavor). Here we present a more adaptive algorithm that chooses which pages to push based on the request and writing pattern. We call this scheme *selective push*.

Rather than pushing the entire page, we push only the delta's from the previous version of the page, which are typically rather small [19]. On the way up the caching hierarchy the updates are sent via reliable unicast. On the way down, we use a single unreliable multicast sent to a cache's multicast group.[9] Pushing the page does not remove the need for sending invalidations for the previous version, since the data could be lost in transit.

We use a heuristic to decide if a page is sufficiently popular to be pushed. We do not make a single global decision about whether or not to push a page; instead, each cache, and the originating server, make their own independent decision about whether or not to push the page. Every cache (and the server) keeps a counter A_P (initialized to 0) and a *push bit* for each of its pages. If the bit is 1, the cache will forward all pushed updates of the page to all of its downstream caches. The heuristic uses three positive constants: θ, γ, and β. Whenever a cache receives an invalidate of page P, it sets $A_P = A_P - \gamma$; whenever it receives a request for P, it sets $A_P = A_P + \beta$. If $A_P > \theta$ for some threshold θ, the cache (or the server) sets the push bit of the page to 1; otherwise the push bit is set to 0. In addition, we let each downstream cache notify its immediate upstream cache when a pushed page is first read; these read notifications are forwarded recursively until they hit a read page. This allows caches who have pushed the page to still get accurate readings on whether the pushed page was read downstream before the page was invalidated.

Recent work has addressed the issue of pushing web pages. Continuous Multimedia Push (CMP) [24] assigns a unique multicast group to every popular page and continuously multicasts pages to their groups. They found that multicast push is preferable to caching only when pages are

[8] Note that our analogy to application-level multicast is completely unrelated to our use of real multicast to communicate between parent and child caches.

[9] Unreliable distribution is sufficient, since pushing affects performance and not correctness of the protocol. However, one could use SRM [11] or other reliable multicast protocol for this distribution; we have not done so in our simulations to reduce complexity, but it is a very natural design choice.

very popular and change very frequently. LSAM [27] assigns one multicast group per "topic"; popular pages of similar topic (*e.g.*, SuperBowl) are multicast to a unique group when they are created or modified. Our scheme is similar in spirit to these approaches, but quite different in implementation. We use application-level "routing" of pushes that is equivalent to multicast, and we adaptively decide which pages are sufficiently popular to push.

4 Analytical Performance Evaluation

If we assume, as we will throughout this paper, that caches are effectively infinite,[10] then the behavior of our web caching consistency protocol can be analyzed on a per-page basis; if no meta-state or page data is deleted from a cache due to space considerations, then the message generation behavior (*i.e.*, invalidations, etc.) for a given page is independent of what happens for all other pages.[11] We now analytically evaluate the performance of our proposed protocol in a very simple setting. We consider a single client, a single cache, and a single server. The client sends out requests (reads) for a particular page, and the server modifies (writes) that page.

We compare several different web consistency approaches. The first, *omniscient* TTL (OTTL), is not a realistic scheme, but it provides a useful benchmark; in this scheme caches magically know when a page has been modified and only send the IMS request in those cases. The second is *poll-always* (PA) which, as we discussed in Section 2, is just a TTL approach with TTL=0. The other two are variants of our invalidation scheme: our basic invalidation scheme with no page pushing (BINV) and our invalidation scheme with pages always pushed (PINV).[12] To make the modeling easier, we assume there is no delay between when invalidations are generated and their being sent out (*i.e.*, invalidations don't wait for the next heartbeat). Thus, all of the protocols described here provide the same level of strong consistency; if we ignore page modifications made after the server has responded to a request and before the response arrives at the cache, then there are no stale pages delivered by any of these protocols. We do not study the looser policies of adaptive TTL or fixed TTL here because their finite timeout periods makes the analysis intractable; we evaluate them using simulation in Section 5.

Since none of these algorithms depends on absolute time, we care only about the patterns of reads and writes arriving at a cache. We can characterize the behavior of these algorithms by describing which messages get sent upon one of these four events: a read following a write (WR), a read following a read (RR), a write following a write (WW), and a write following a read (RW). Let F_{RR}, F_{RW}, F_{WR}, F_{WW} denote the average rate at which the patterns RR, RW, WR, and WW occur, respectively. We model the reading and writing as Poisson processes of rate r and w, respectively, and so the frequencies of events can be computed as follows: $F_{RR} = \frac{r^2}{r+w}$, $F_{WW} = \frac{w^2}{r+w}$, $F_{RW} = F_{WR} = \frac{rw}{(r+w)}$.

Table 1 summarizes the bandwidth usage, server hit count, and cache response delay of each protocol for these four events. The relative performance in terms of server hit counts and response time holds regardless of the read and write rates. PINV completely eliminates server hits,[13] and BINV and OTTL have the same server hit count, which is less than PA. The same ordering applies to response time: PINV has no delays, OTTL and BINV have an intermediate level of delay, and PA has the most delay. The bandwidth comparison of these algorithms is less clear and, in some cases, depends on the values of the various parameters.

	OTTL	PA	BINV	PINV
RR	delay: 0 bw: 0 hc: 0	delay: 2 d_1 bw: $2b_{IMS}$ hc: 1	delay: 0 bw: 0 hc: 0	delay: 0 bw: 0 hc: 0
RW	bw: 0	bw: 0	bw: b_{inv}	bw: $b_P + b_{inv}$
WR	delay: $d_1 + d_2$ bw: $b_P + b_{IMS}$ hc: 1	delay: $d_1 + d_2$ bw: $b_P + b_{IMS}$ hc: 1	delay: $d_1 + d_2$ bw: $b_P + b_{GET}$ hc: 1	delay: 0 bw: 0 hc: 0
WW	bw: 0	bw: 0	bw: 0	bw: $b_P + b_{inv}$

Table 1: Table of bandwidth, server hit count, and delays for each of the four events: RR, RW, WR, WW. b_{inv} is the cumulative size of a repeated set of invalidation messages. b_P is the average size of a page. b_{GET} is the size of an HTTP GET request. b_{IMS} is the size of an IMS request. b_{ntf} is the size of a read notification message. d_1 is one way delay of IMS, GET, invalidation and responses. d_2 is the one way delay of transmitting a page from server to cache.

For convenience, we assume $b_{IMS} = b_{inv} = b_{ntf} = b_{GET}$, and let b_{ctl} denote this size. Since these are all small packets, we do not introduce significant errors by ignoring the size differences. Notice that OTTL uses less bandwidth than any other scheme. PA uses less bandwidth than BINV if and only if $2r < w$; the tradeoff is between PA sending an IMS and response on reads following reads versus BINV sending an invalidate message on writes following reads. PA uses less bandwidth than PINV if and only if $(\frac{r}{w})^2 < \frac{1+\frac{b_P}{b_{ctl}}}{2}$. Lastly, PINV uses less bandwidth than BINV if and only if $\frac{r}{w} > 1 + \frac{b_P}{b_{ctl}}$.

If one assumes the size of pages dominates the size of the control messages then the limit of $b_{ctl} = 0$ may provide some insight. When $b_{ctl} = 0$ then all the protocols except PINV require the same bandwidth (pages are transmitted whenever a modified page is first read). BINV has the same performance, in terms of server hit counts and response times, as the OTTL, our idealized benchmark. BINV has lower response time and server hit count than PA. This performance gap grows as the reading rate increases, since BINV's advantage is that it need not contact the server (thereby incurring server hit counts and delay) when a valid page is read; when the read rate is much lower than the write rate, few of the requests find a valid page at the cache, but as the read rate increases more of these requests find a valid page at the cache. Thus, if the bandwidth of control messages can be ignored, then the main performance criteria separating BINV from PA are server hit counts and response times, not bandwidth, and these performance gaps become more significant as the reading rate increases. PINV eliminates hit counts and delays but at the cost of increased bandwidth.

In order to make our analysis in this section tractable, we assumed a very idealized environment and did not consider every protocol. In the next section we will use simulations to evaluate all of the consistency protocols in a somewhat

[10] See Appendix A for further discussion of this assumption.

[11] The only degree of interaction is the number of pages over which the overhead of heartbeats is shared.

[12] PINV can be seen as a version of *mirroring* in which updated pages are automatically mirrored at remote sites.

[13] Of course, this reduction in server hits comes at the cost of the server pushing the data; however, we believe that the cost of answering a request may be higher than that of pushing a page update.

more realistic setting.

5 Simulations

In this section we use simulations, performed using the *ns* [2] simulator, to evaluate the performance of our proposal, and to compare it to several other approaches. In particular, we investigate the performance of our basic invalidation protocol (BINV), along with the variants selective push (SINV) and push-always (PINV), and compare them to poll-always (PA), adaptive TTL (ATTL), fixed TTL (FTTL) and omniscient TTL (OTTL).

We evaluate these various web cache consistency protocols using two categories of metrics: user-centric metrics and infrastructure-centric metrics. The user-centric metrics, which quantify the user's level of satisfaction with the service provided, are client response time[14] and staleness. We measure staleness in three ways: the maximum and average staleness taken over all pages, and the percentage of pages which are delivered stale (stale hit rate). Most previous papers on web consistency used stale hit rate as the only metric for staleness; we prefer to emphasize the average staleness, since staleness is not a binary property. That is, how out-of-date a page is, not just whether or not the page is stale, may be important. The infrastructure-centric metrics quantify (aspects of) the burden placed on the network infrastructure by these various protocols; we measure the total network bandwidth (in byte-hops), the bandwidth at the server, and the rate of (GET and IMS) requests at the server.

Recall that several of these algorithms have adjustable parameters that control their performance: the heartbeat rate h for the invalidation-based algorithms, the TTL value for FTTL, and the threshold for ATTL. We are not interested in measuring the tradeoff between staleness and bandwidth achievable by each of these protocols. Rather, we assume low average staleness is a performance requirement and ask how much bandwidth and delay are incurred by the protocols to achieve a particular level of staleness. Therefore, we set the heartbeat rate for BINV to be 10 per minute and then vary the parameters for FTTL and ATTL so that they all have roughly equivalent average staleness.[15] The additional parameters required in SINV are set as follows: $\gamma = 1$ (invalidation constant), $\beta = 2$ (request constant), $\theta = 8$ (push threshold).

We begin our simulations with a very basic scenario, and then later describe several additional scenarios. The results show that our invalidation scheme can achieve the same staleness as the TTL approaches with lower response time and overhead. The advantages are most pronounced for popular pages which do not change often.

5.1 Basic Scenario

In this scenario we consider a single two-level caching hierarchy (5 leaf caches and a top-level cache) embedded in a simple network topology, as shown in Figure 3. As we discussed in Section 4, if we treat the caches as infinite then the behavior attributed to each page is independent of other pages. Consequently, we choose the workload in our basic scenario to have only a single page so that we can focus more narrowly on how the performance of these consistency protocols depends on the reading and writing patterns of a page. This single page, chosen to be 1KB in size, is read and written according to Poisson processes with average rates r (per-client) and w, respectively. We consider two cases: a *write-dominated* (WD) page, where the read rate (per-client) is one per 2.5 hours and the write rate is 1 per 15 minutes ($\frac{w}{r} = 10$); and a *read-dominated* (RD) page, where the read rate (per-client) is 1 per 2 minutes and the write rate is 1 per 10 minutes ($\frac{r}{w} = 5$).

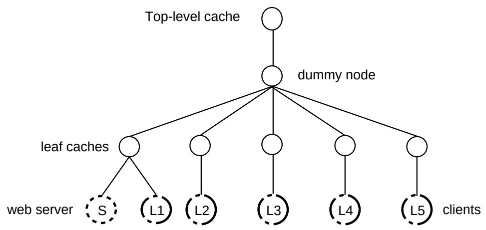

Figure 3: Network topology in the basic scenario. All links between server/clients and leaf caches have 10Mb bandwidth and 2ms delay. All links among caches and the dummy node have 1.5Mb bandwidth and 50ms delay.

We now describe some of the simulation details. The IMS and GET messages are 43 bytes, and each invalidation record adds an additional 32 bytes to a heartbeat. Because in reality the header of a heartbeat is amortized over many pages, we ignore it in these single-page simulations. The RD and WD simulations were run for approximately one day and five days (simulation time), respectively, with the initial 7 and 15 minutes taken to be a warmup period (for the RD and WD simulations, respectively).

Tables 2 and 3 show the results for RD and WD pages, respectively. Because of the sensitivity of the results to the tuning parameters, exactly matching the average staleness across protocols is difficult. When confronted with this, we chose parameter values for ATTL and FTTL that yielded slightly *higher* average staleness than our BINV benchmark (e.g., 8.37 and 11.9 msec versus 8.06 msec in Table 2). This gives us a lower bound on the overhead and delay incurred for the ATTL and FTTL to match the staleness of BINV. We first discuss the RD case, and begin by comparing BINV to the TTL-style protocols. Compared with PA, BINV uses 26% less bandwidth, has 27 times less server hit count and 10 times faster response time. Because FTTL and ATTL are required to maintain the same low staleness as BINV, they both have small TTL values (ATTL threshold equals 0.0105 and FTTL time-to-live equals 9.5 seconds) and therefore behave like PA. Their bandwidth is slightly higher and their response time and server hit count are much higher than those of BINV. BINV's performance is similar to that of OTTL, but it has slightly higher bandwidth consumption due to its invalidation overhead. Comparing BINV to PINV, we find, as expected, that pushing data reduces response time and eliminates server hits while increasing bandwidth by only about 6%. Because the read rate is so much higher than the write rate, updated pages are eventually fetched from the server, so pushing them out immediately for this read-dominated workload does not incur additional bandwidth overhead. SINV's performance is very close to that of PINV.

Turning to the WD case, we see that the problem of matching the average staleness across the tunable protocols is exacerbated. This is due to the fewer number of stale hits (in absolute terms) in a write-dominated work-

[14] The latency between sending a request and complete receipt of the response.

[15] We are not able to accomplish this in all cases. We elaborate on this below.

	BINV	ATTL (0.0105)	FTTL (9.5)	OTTL	PA	SINV	PINV
AS	8.06	8.37	11.9	0.00	0.00	0.34	0.09
MS	4.95	17.00	8.43	0.00	0.00	1.08	0.21
SR	0.38	0.15	0.27	0.00	0.00	0.06	0.05
TB	6.90	8.73	8.42	5.75	9.33	7.35	7.38
CR	0.06	0.53	0.48	0.05	0.61	0.04	0.04
SH	123	2684	2324	124	3300	8	0
SB	158.4	694.1	617.4	147.9	824.2	153.9	153.6

Table 2: Statistics of a read-dominated page in the basic scenario. AS: average staleness (millisecond); MS: maximum staleness (second); SR: stale hit rate (%). TB: total bandwidth (MB-Hop). CR: client response time (second). SH: server hits. SB: server bandwidth (KB).

	BINV	ATTL (0.1)	FTTL (80)	OTTL	PA	SINV	PINV
AS	1.22	544.4	13.1	0.00	0.00	1.22	0.00
MS	0.29	76.19	3.11	0.00	0.00	0.29	0.00
SR	0.42	0.84	0.42	0.00	0.00	0.42	0.00
TB	1.40	1.41	1.41	1.35	1.42	1.40	6.16
CR	0.41	0.60	0.61	0.46	0.62	0.41	0.26
SH	155	224	230	151	236	149	1
SB	184.7	193.7	194.9	179.1	197.2	184.8	275.4

Table 3: Statistics of a write-dominated page in the basic scenario.

load. Nonetheless, the data show that BINV's performance advantage is now reduced. For the time-to-live value shown in the table, FTTL has worse staleness than BINV, nearly the same bandwidth but only about 50% longer response time and 50% higher server hit count. ATTL has worse average staleness while the other metrics are comparable to FTTL. PA has performance very similar to FTTL (reflecting the very small TTL used in FTTL). Again, PINV achieves very low response time and server hit count, but this time at the cost of a factor of 4 in bandwidth consumption. Note that SINV behaves like BINV in this WD case, but behaved more like PINV in the RD case; this was the goal of the adaptive algorithm in SINV, to actively push pages only when they are read-dominated.

These results are completely consistent with the theoretical analysis of Section 4. The major benefits of invalidation schemes (over TTL-based schemes) are savings of response time and server hit count, and these benefits are much more pronounced in the read-dominated case. Adding push increases these advantages further, but at the cost of significantly more bandwidth in the WD case.

In this basic scenario, and in each of the following scenarios, we assume that the heartbeat rate h is greater than the write rate w times the number of cache-hops H. This will likely be true for the vast majority of pages, however there are some pages, such as those containing stock quotes, that will change faster than $\frac{h}{H}$. If such pages are also popular, our invalidation approach will deliver a significant fraction of pages stale (since the invalidations are still in transit from server to leaf cache); see [32] for more details. Such pages are better delivered using multicast techniques, such as Continuous Multicast Push [24].

5.2 More Complex Topology

In the second scenario, to test the effect of having a more complicated network topology, we took a 3-level caching hierarchy (leaf, intermediate, and top-level), with a branching ratio of 2 at each level, and embedded it into a 300 node random transit-stub network topology created by the GT-ITM [5] topology generator. The top-level cache and all intermediate caches are on transit nodes. All leaf caches are in the stub network associated with the transit node where the parent intermediate cache resides, and each intermediate cache is in a different stub network.

	BINV	ATTL (0.01)	FTTL (12)	OTTL	PA	SINV	PINV
AS	15.6	16.2	17.5	0.00	0.00	2.47	0.88
MS	6.44	26.46	10.47	0.00	0.00	1.25	0.94
SR	0.54	0.12	0.34	0.00	0.00	0.26	0.16
TB	18.16	23.04	22.67	15.55	23.83	19.45	19.53
CR	0.18	0.49	0.44	0.12	0.53	0.12	0.12
SH	124	2290	1880	126	2583	8	1
SB	158.6	607.2	560.9	150.3	694.5	154.0	153.6

Table 4: Statistics of a read-dominated page in a more complex topology.

	BINV	ATTL (0.01)	FTTL (95)	OTTL	PA	SINV	PINV
AS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SR	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TB	6.38	6.47	6.45	6.24	6.48	6.38	26.68
CR	0.76	0.91	0.89	0.76	0.91	0.76	0.68
SH	131	192	186	130	193	131	1
SB	156.1	156.8	156.5	154.2	161.0	156.1	274.8

Table 5: Statistics of a write-dominated page in a more complex topology.

Tables 4 and 5 present the results from simulations on this topology with RD and WD pages, respectively. The basic relative trends in the data appear unaffected by introducing a more complicated topology. For the RD page, the TTL approaches have worse response time and server hit counts than BINV, and the push approaches offer reduced response time, server hit counts, and staleness without incurring any additional bandwidth. Compared with the RD case, BINV's advantages are greatly reduced in the WD case.

5.3 More Complex Workload

The Poisson workload used so far is not intended to be an accurate model of reality; rather, it is merely a simple test case. We have augmented the simulations presented here with simulations on a wide variety of other workloads. We have considered compound pages, where the page contains multiple objects (such as embedded graphics). We have also considered reading and writing processes that are heavy-tailed and processes that are uniformly distributed. The results from these simulations are presented in [32]. Those results were qualitatively similar to these presented here, and space limitations prevent us from including them. However, we do want to present data from one additional workload.

Our previous data was generated using artificial read and write processes. To get a sense of a more realistic scenario, we now consider a trace-driven workload consisting of the read sequence of a single page extracted from a real trace. We pick two pages, one popular and one unpopular, from a 5-day segment of the UCB Home-IP trace [14], and apply the consistency algorithms to the two pages. The popular page has 62,582 requests, and the unpopular page has 21. No page modification data is available for these traces, so we used a Poisson model with an average of one modification

	BINV	ATTL (0.0015)	FTTL (8)	OTTL	PA	SINV	PINV
AS	1.32	1.36	1.65	0.00	0.00	0.05	0.01
MS	4.69	10.90	8.76	0.00	0.00	2.11	0.18
SR	0.07	0.04	0.06	0.00	0.00	0.01	0.01
TB	27.16	75.15	62.07	22.03	91.75	27.07	27.07
CR	0.01	0.45	0.33	0.01	0.60	0.01	0.01
SH	119	39087	25182	119	58124	2	1
SB	72.6	8342	5380	41.8	12381	64.8	64.1

Table 6: Statistics of a popular page in the UCB Home-IP trace.

	BINV	ATTL (0.2)	FTTL (2800)	OTTL	PA	SINV	PINV
AS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MS	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SR	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TB	279.8	281.8	279.2	279.2	283.0	279.8	2781.5
CR	0.61	0.80	0.80	0.77	0.83	0.61	0.55
SH	18	19	21	17	21	18	0
SB	42.5	43.1	43.5	42.6	43.5	42.5	217.5

Table 7: Statistics of an unpopular page in the UCB Home-IP trace.

per hour (based on data in [10]). With this modification rate, the popular page is read-dominated, and the unpopular page is write-dominated. Tables 6 and 7 present the results from simulations for these two pages.

These results are consistent with our previous results. The only novelty here is the fact that for the popular page the IMS overhead of the TTL approaches is more evident. In order to maintain the same staleness as BINV, ATTL required 3 times as much bandwidth as BINV, and FTTL more than doubled bandwidth.

5.4 The Effect of Packet Losses

Up to this point, our simulations do not include any packet losses. We now return to our basic scenario and introduce per-link packet loss rates in order to evaluate the effect of packet losses on the consistency protocols.

In our protocol, both invalidations and pushed updates are sent out via unreliable multicast. When packet loss is present, we expect that performance will degrade. Because invalidations are piggybacked in several consecutive heartbeats, but pushes are sent only once, we expect that invalidations are less vulnerable to packet loss than pushes. In order to test these expectations, we introduced 3% per-link losses into our basic scenario. For the network shown in Figure 3, 3% per-link loss rate corresponds to end-to-end loss rates between 3% and 6% (which is intended to match the loss rates of between 2.65% and 5.28% found in [22]). Results are shown in Tables 8 and 9; the data presented are averages over 9 runs.

Packet loss increases the bandwidth and response time for all the protocols. BINV's stale hit rate and average staleness increase slightly, and the maximum staleness increases significantly, because the lost invalidations need at least another heartbeat interval to reach leaf caches. SINV behaves similarly to BINV but, as expected, PINV, is more significantly affected by packet loss; its average staleness and maximum staleness are increased substantially.

When loss rate grows even bigger, some caches will time out due to consecutively lost heartbeats, and our failure recovery mechanism will be triggered (see Section 6). This will

	BINV	ATTL (0.013)	FTTL (12)	OTTL	PA	SINV	PINV
AS	9.88	12.5	12.2	0.00	0.00	0.73	0.43
MS	7.15	20.77	11.42	0.00	0.00	1.81	1.38
SR	0.41	0.27	0.30	0.00	0.00	0.09	0.06
TB	7.16	9.19	8.88	6.01	9.92	7.56	7.58
CR	0.09	0.74	0.67	0.09	0.93	0.05	0.05
SH	128	2543	2154	128	3260	8	1
SB	198.7	754.5	669.8	180.5	936.3	194.8	192.3

Table 8: Statistics of a read-dominated page in the basic scenario with 3% per-link loss rate.

	BINV	ATTL (0.03)	FTTL (135)	OTTL	PA	SINV	PINV
AS	34.9	544.4	333.1	0.00	0.00	34.9	1.22
MS	4.40	76.19	76.19	0.00	0.00	4.40	0.29
SR	0.84	0.84	0.84	0.00	0.00	0.84	0.42
TB	1.59	1.57	1.54	1.50	1.59	1.59	6.27
CR	0.53	0.93	0.95	0.75	0.89	0.53	0.36
SH	155	239	222	151	239	149	1
SB	224.3	236.5	219.1	223.0	238.8	212.1	316.4

Table 9: Statistics of a write-dominated page in the basic scenario with 3% per-link loss rate.

impose a transient increase in response latency (because all affected cached pages are invalidated, and an IMS will be generated by the next request).

5.5 Related Work

There have been several recent papers comparing the effectiveness of TTL and invalidation approaches: Worrell [30], Gwertzman and Seltzer [15], and Cao and Liu [6]. Worrell claimed that when FTTL has similar bandwidth consumption as unicast invalidation, it has 20% stale hits, and therefore concluded that unicast invalidation is preferable for strong consistency. Gwertzman and Seltzer argued that bimodal lifetime of web pages makes ATTL the preferred choice; their trace-driven simulation showed that ATTL had few stale hits (<5%) and took much less bandwidth than unicast invalidation. Using real systems in trace-driven experiments, Cao and Liu confirmed that ATTL had few stale hits, but they found that ATTL and unicast invalidation had similar bandwidth usage. Moreover, they found that unicast invalidation at times led to increased latency because of the message processing overhead at the server.

Our results differ from those in previous work for a couple of reasons. Compared to simple unicast invalidation, our invalidation protocol can avoid much of the redundant invalidation traffic. Thus, in most cases, it takes less or the same bandwidth as ATTL while achieving the same level of page staleness and resulting in much less server load and client response time. At the same time, our work is somewhat complementary to the previous investigations. Because we focus on single-page workloads when evaluating this protocol, we are able to identify more precisely the effect of different reading and writing processes on the results. In addition, we focus on average staleness, rather than the stale hit rate, as the crucial staleness metric. Finally, because we assume that perishable pages require very low staleness, we focus our simulations on operating regimes with much lower staleness measures than previous studies.

6 Additional Design Issues

We have presented the basic design of our protocol in an ideal environment with infinite caches that never fail, a single stable hierarchy with synchronized clocks, and with all pages included in the architecture. In this appendix we discuss additional aspects of the design to cope with more realistic settings.

Clock Skew In Section 3 we assumed that the clocks in the caching hierarchy were perfectly synchronized. However, if the maximal clock skew between a cache and its upstream and downstream neighbors is bounded by ϵ then the cache timeout period should be $T-\epsilon$ instead of T. We assume that in typical cases $T \gg \epsilon$ so this modification in the protocol will have little impact.

Finite Cache Caches are, in reality, finite. While we argue in Appendix A that our design does not require unrealistically large amounts of state in caches, it is important that the design can cope with situations where the cache has exceeded its capacity. First, to keep the invalidation contract in force, a cache need only remember the *meta-data* (the URL and the last-modification time) about the page, and can freely discard the actual contents of the page. Second, if the cache is forced to discard the meta-data itself, then it must send an invalidation for that page to its children and/or its parent depending on whether the page has been read from those directions. While this may impact performance, the correctness of the protocol is unaffected.

Failure Recovery The algorithm as described deals with the case where a cache fail-stops. However, it does not describe how a cache can recover from a failure. We require that caches recover in a *naive* state; that is, they invalidate all pages in the cache and send a LEAVE message to their parent and child caches. This allows all affected invalidation contracts to be broken before the cache reattaches. We have the following property:

Property 4 *As long as caches that have failed recover in a* naive *state then the three properties in Section 3.1 hold even in the presence of failures and recoveries.*

One remaining problem is how to recover the server routing entries that were evicted during a partition or lost during a failure. There are two cases. First, if a parent cache C1 times out a child cache C2 from whom it sourced servers, it needs to send a JOIN_QUERY after hearing from C2 again. C1 can piggyback the JOIN_QUERY in a heartbeat, just as it does with invalidations. Second, if C2 times out C1, C2 needs to send C1 a JOIN which contains its server routing table, *i.e.*, all of the servers from which it has heard JOINs. In both cases, when C1 recovers its routing table, it needs to notify its parent of its current routing table.

Direct Request Using a hierarchy (or cache mesh) to forward requests to servers can introduce significant delay [1]. Because requests in our hierarchy might travel both up and down the hierarchy, this risk of delay is higher. However, we can extend our design so that the client's primary cache can, upon a cache miss, go directly to the server to get the data. When the cache receives the data, it then, after handing the data to the client, establishes the invalidation contract by sending a *pro forma* request up the hierarchy. The pro-forma request is used merely to establish the required correct state in the hierarchy, and does not elicit a reply of data from the caches or the server. The pro-forma carries with it the Last-Modified time of the page returned by the server. It stops being forwarded when it hits a cache which has that version of the page, or meta-data for it, in residence. If the pro-forma hits a cache (or server) that has a more recent version of the page in residence, an invalidate is generated and sent back down the path. If the pro-forma hits a cache with a valid older version of the page, no action need be taken since an invalidate is on the way. In this manner, the caching hierarchy provides invalidations while the delivery of actual web pages bypasses this hierarchy. This alleviates some of the disadvantages of a web caching hierarchy, such as parent cache overloading and increased response time [26].

Multiple Hierarchies and Multi-Homing There will obviously be multiple caching hierarchies in the Internet, although we expect the number to be relatively limited (less than, say, 100). Our design can easily be extended to handle these multiple hierarchies by having the Top-level cache of one hierarchy contact caches in other hierarchies. This can be accomplished using a single multicast group comprised of the members of all Top-level caches. Each top-level cache multicasts its heartbeats to this group, as well as to its multicast group in its own hierarchy. Whenever a top-level cache, call it TLC1, gets a request for an unknown web server, it queries the server about its top-level cache, call it TLC2, and then forwards the request to TLC2 as if TLC2 were a parent cache.

While our design requires that a server only attaches to a single cache in a given hierarchy, we allow it to attach to multiple hierarchies; we call this a *multi-homed* server. The design works without significant modification.

Supplying Service to a Subset of Pages We do not expect that all pages will need the level of consistency provided by our architecture. In order to provide invalidations on a subset of all web pages, we propose a new HTTP header field that describes whether or not the page should be subject to this consistency architecture. The simplest approach is to have the server set this field. There are some situations where it might be appropriate to allow a client to set this field, thereby requesting invalidation service for the page. Of course, the server must be willing to support this service by participating in the sending out of heartbeats and invalidations. There are some subtle issues in both of these approaches which are too detailed to discuss here but are covered in [32].

Deploying in Existing Cache Hierarchies In order to implement our protocol in existing cache hierarchies, we can enhance ICP [29], the *de facto* inter-cache communication protocol, to support our consistency protocol. Four new types of ICP messages are needed: *heartbeat*, *JOIN*, *LEAVE*, *request notification*, and *PUSH*. If direct request is desired, another message type, *pro forma* is needed. Because these messages do not interact with existing ICP messages, adding them to ICP is straightforward.

7 Conclusion

In this paper we have presented and evaluated a web cache consistency protocol based on invalidation. Our proposal builds on previous work in the literature, combining the ideas of multicast invalidations with volume leases and incorporating them within a caching hierarchy to make the design more scalable. Our performance evaluation suggests that when the heartbeat rate h is larger than the writing rate times the number of hops (wH), then the invalidation approach is very effective in keeping pages relatively fresh. When pages are write-dominated, then the invalidation approach offers few advantages since all the protocols, if they are to ensure freshness, must go back to the server to get a valid page. However, when pages are read-dominated, which we think will be the common case for perishable pages (*e.g.*, CNN and other news pages), then the invalidation approach offers significant reductions in server hit counts and client response time. In both cases, our invalidation scheme requires similar or less bandwidth than the TTL-style protocols.

Our analysis focused exclusively on the technical aspects of the protocol. However, the remaining questions, and the barriers to deployment, may be more economic and institutional in nature. Our design uses a set of relatively stable and well-managed caching hierarchies (though it can work with other cache organizations). Currently this does not describe the current state of web caching, and so assuming the existence of caching hierarchies may seem like a dubious foundation on which to build our architecture. However, the institutional trends in ISPs appear to be one of consolidation, and in the future these large ISPs may very well provide such a caching hierarchy as part of their service (and the mirroring service provided by @Home is some evidence in this direction). Moreover, the hierarchy we envision does not require central management (since parents need not know the list of their children explicitly) nor must it be deployed ubiquitously to be useful, so the barriers to its realization are somewhat reduced.

In addition, the deployment of any such a web cache consistency protocol would only be undertaken if ISPs determine that there is sufficient demand for relatively fresh versions of perishable pages. It seems clear that perishable pages comprise only a small fraction of current web usage. On this basis one might be tempted to dismiss the consistency problem as unimportant. However, if the web is to serve as the foundation on which much of the information infrastructure is built, then perhaps it should be augmented to meet the needs of this class of pages.

Clearly the whole issue of deployment, depending as it does on such unknowables as the future usage and economics of the web, and the nature of the ISP business, is far beyond our ken. We only caution that the growth path of the web caught many of us by surprise, and we should be humble in our confidence to predict, based on its current usage and existing institutional arrangements (where we expect the case for its deployment is weak) whether the future of the web would be significantly aided by deploying such a consistency architecture, and whether it is organizationally feasible. Our goal here was merely to demonstrate that it is indeed technically feasible.

References

[1] Baentsch, M., Baum, L., Molter, G., Rothkugel, S., and Sturm, P. World-Wide Web caching – the application level view of the Internet. *IEEE Communications Magazine 35*, 6 (June 1997). http://www.uni-kl.de/AG-Nehmer/Projekte/GeneSys/Papers/communic.ps.

[2] Bajaj, S., Breslau, L., Estrin, D., Fall, K., Floyd, S., Haldar, P., Handley, M., Helmy, A., Heidemann, J., Huang, P., Kumar, S., McCanne, S., Rejaie, R., Sharma, P., Shenker, S., Varadhan, K., Yu, H., Xu, Y., and Zappala, D. Virtual Inter-Network Testbed: Status and research agenda. Tech. Rep. 98-678, University of Southern California, July 1998. *ns* web site: http://mash.cs.berkeley.edu/ns.

[3] Baker, M., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. Measurements of a distributed file system. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Oct. 1991), pp. 198–221.

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

[5] Calvert, K., Doar, M., and Zegura, E. Modelling Internet topology. *IEEE Communications Magazine* (June 1997).

[6] Cao, P., and Liu, C. Maintaining strong cache consistency in the World-Wide Web. In *Proceedings of the International Conference on Distributed Computing Systems* (May 1997), pp. 12–21.

[7] Cate, V. Alex - a global filesystem. In *Proceedings of the 1992 USENIX File System Workshop* (Ann Arbor, MI, May 1992).

[8] Chankhunthod, A., Danzig, P., Neerdaels, C., Schwartz, M., and Worrell, K. A hierarchical Internet object cache. In *USENIX Conference Proceedings* (1996), pp. 153–63.

[9] Cohen, E., Krishnamurthy, B., and Rexford, J. Improving end-to-end performance of the Web using server volumes and proxy filters. In *Proceedings of the ACM SIGCOMM* (1998).

[10] Douglis, F., Feldmann, A., Krishnamurthy, B., and Mogul, J. Rate of change and other metrics: a live study of the world wide web. Tech. Rep. 97.24.2, AT&T Labs, Dec. 1997. A shorter version appeared in Proc. of the 1st USENIX Symposium on Internet Technologies and Systems.

[11] Floyd, S., Jacobson, V., Liu, C., McCanne, S., and Zhang, L. A reliable multicast framework for light-weight sessions and application level framing. *ACM/IEEE Transactions on Networking 5*, 6 (Dec. 1997), 784–843. ftp://ftp.ee.lbl.gov/papers/srm_ton.ps.Z.

[12] Ford, P. S., Rekhter, Y., and Braun, H.-W. Improving the routing and addressing of IP. *IEEE Network Magazine 7*, 3 (May 1993), 10–15.

[13] Gray, C., and Cheriton, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the ACM Symposium on Operating Systems Principles* (1989), pp. 202–210.

[14] Gribble, S. D., and Brewer, E. A. System design issues for Internet middleware services: Deductions from a large client trace. In *Proceedings of The USENIX Symposium on Internet Technologies and Systems* (Dec. 1997).

[15] Gwertzman, J., and Seltzer, M. World-Wide Web cache consistency. In *Proceedings of the USENIX Conference Proceedings* (Copper Mountain Resort, CO, USA, Dec. 1996), pp. 141–51.

[16] Inktomi Inc. Inktomi Traffic Server, 1998. http://www.inktomi.com/products/traffic/product.html.

[17] Kumar, K., Radoslavov, P., Thaler, D., Alaettinoglu, C., Estrin, D., and Handley, M. The MASC/BGMP architecture for inter-domain multicast routing". In *Proceedings of the ACM SIGCOMM* (Vancouver, Canada, Sept. 1998). http://catarina.usc.edu/estrin/papers/masc-bgmp-arch.ps.

[18] Lawrence, S., and Giles, C. L. Searching the Web: General and scientific information access. *IEEE Communications Magazine 37*, 1 (Jan. 1999), 116–121.

[19] Mogul, J., Douglis, F., and Feldmann, A. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM* (Sept. 1997), pp. 181–194. http://www.acm.org/sigcomm/sigcomm97/papers/p156.ps.

[20] Netcraft. The Netcraft Web server survey. http://www.netcraft.com/Survey/.

[21] Nua Inc. Nua Internet surveys. http://www.nua.ie/surveys/how_many_online/index.html.

[22] Paxson, V. End-to-end Internet packet dynamics. In *Proceedings of the ACM SIGCOMM* (1997).

[23] Rodriguez, P., and Biersack, E. W. Continuous multicast distribution of Web documents over the Internet. *IEEE Network Magazine* (March-April 1998).

[24] Rodriguez, P., Biersack, E. W., and Ross, K. W. Improving the WWW: Caching or multicast. In *Proceedings of The Third International WWW Caching Workshop* (June 1998).

[25] Rosenstein, A., Li, J., and Tong, S. Y. MASH: The multicasting archie server hierarchy. *SIGCOMM Computer Communication Review 27*, 3 (July 1997).

[26] Tewari, R., Dahlin, M., Vin, H., and Kay, J. Beyond hierarchies: Design considerations for distributed caching on the internet. Tech. Rep. CS98-04, Department of Computer Sciences, UT Austin, May 1998.

[27] Touch, J. The LSAM proxy cache - a multicast distributed virtual cache. In *Proceedings of The Third International WWW Caching Workshop* (June 1998).

[28] Vahdat, A., Eastham, P., and Anderson, T. WebFS: A global cache coherent filesystem. Tech. rep., Dept of EECS, UC Berkeley, Dec. 1996. http://www.cs.berkeley.edu/~vahdat/webfs/webfs.html.

[29] Wessels, D., and Claffy, K. ICP and the Squid web cache. *IEEE Journal of Selected Areas in Communication 16*, 3 (Apr. 1998).

[30] Worrell, K. J. Invalidation in large scale network object caches. Master's thesis, Department of Computer Science, University of Colorado, 1994.

[31] Yin, J., Alvisi, L., Dahlin, M., and Lin, C. Using leases to support server-driven consistency in large-scale systems. In *Proceedings of the 18th International Conference on Distributed Computing System* (May 1998).

[32] Yu, H., Breslau, L., and Shenker, S. A scalable web cache consistency architecture. Tech. Rep. 99-708, Dept. of Comp. Sci., Univ. of Southern Calif., June 1999.

[33] Zhang, L., Floyd, S., and Jacobson, V. Adaptive web caching. Project proposal, Feb. 1997. http://irl.cs.ucla.edu/AWC/proposal.ps.

A Estimation of State and Bandwidth Requirements

Our architecture requires cache state and inter-cache communication in order to provide loose consistency. In this section we provide some very crude estimates on the cache state and inter-cache bandwidth required by our scheme. These estimates, which should not be taken as a definitive quantitative statement about the overhead of the protocol, indicate that the scheme is indeed feasible.

A.1 State Requirements

Our protocol introduces two additional items into the cache state: page metadata and the server routing table. We first estimate the amount of metadata that might be stored in a cache. If we assume that that a top-level cache holds no more than 320 million pages (the estimate of all publicly indexable web pages [18]), and one meta-data record contains 80 bytes (which is enough for a URL, last-modification time, push counter and several flags), this results in about 25.6GB of metadata. This is quite small compared to modern large caches [16], and is dwarfed by the storage requirements needed to store the actual pages.

Next, we estimate the size of the server routing table. The top-level cache, if there is only a single hierarchy, has a list of every server. We assume there are roughly 4 million web servers (Netcraft's web server survey [20]). The resulting size of the server routing table is on the order of 32MB, assuming 4 bytes to store each server address and 4 bytes for each child cache address. This again poses no challenge to well-equipped caches. Thus, for the purposes of analyzing our design, we can reasonably assume that caches are effectively infinite (at least as far as meta-data is concerned).

A.2 Invalidation traffic

Our design generates an invalidation every time a read page is written, and we now seek to estimate how much traffic this produces. Let's characterize every page P by a reading rate r_P and a writing rate w_P. The number of invalidations generated by a page is bounded above by $\max[r_P, w_P]$; we will call a page *write-dominated* if $r_P < w_P$ and *read-dominated* if $r_P \geq w_P$. A bound on the invalidation rate for a given cache is $\sum_{P\ in\ cache} \max[r_P, w_P]$ where the sum is over all valid pages in the cache.

We first estimate the traffic seen at a top-level cache. If there is significant logical locality to requests, so that pages tend to be more frequently requested by clients close to them in the hierarchy, then there will be many pages that are never cached at the top-level cache. However, we have no way of estimating the extent of this effect, and so will assume the worst case that all pages are indeed cached at the top-level cache. We estimate that the entire Web has 1 billion pages, which is three times the size of publicly indexable pages [18]. To estimate r_P and w_P, we use numbers from the DEC proxy traces cited in [10]. This trace covers a large population (7400 distinct clients), and contains 505,000 requests of 204,000 distinct pages over a period of 2 days. Most pages, roughly 80%, have only one access in the trace, and we consider these to be write-dominated pages. It is difficult to estimate r_P from the trace due to its limited duration. Instead, we use the average number of such pages read by each user, then extend that rate to the web population. In the DEC trace, about 50% of the requests went to these write-dominated pages. We can compute the read rate of such pages by each user: $0.5 * \frac{505000}{7400} * \frac{1}{2*24*3600} = 0.0002$ (request/user/second). We now extend this to the entire web user population. It is estimated that the web has 151 million users as of December, 1998 [21], and we assume that 1% of these users are as active as those in the DEC trace, and the rest are 100 times less active. This yields a total sum of r_P over all write-dominated pages of $0.0002 * (1.51 + 149.49 * 0.01) * 10^6 = 601$ (invalidation/second).

We consider the other 20% of pages read-dominated. From figures in [10], we conservatively estimate their average change rate as once every 1 hour ($w_P = 0.00028$ per second). Without any evidence on which to base a more educated guess, we conservatively assume that 0.1% of all Web pages are sufficiently popular to be read-dominated. Recall we estimate there are 1 billion web pages, so the sum of w_P over all read-dominated pages is $0.00028 * 0.001 * 10^9 = 280$ (invalidations/second).

If we assume that each invalidation is repeated 5 times, and 32 bytes per invalidation, this yields a total traffic level of $(280 + 601) * 1280 = 1.1$Mbps. Repeating this calculation for the AT&T trace in [10] yields an estimate of 1.7Mbps.

We next estimate the traffic at an intermediate-level cache. We assume that the DEC and AT&T traces are reasonable representatives of intermediate-level caches; using their estimates of the number of readers and the number of pages in residence (rather than the global numbers used in the top-level estimates), we arrive at estimates of 75Kbps and 90Kbps for the DEC and AT&T traces, respectively.

The above estimates assume all pages are included in the consistency architecture; we do not expect that most pages will be considered perishable, and so the consistency architecture will be carrying all web pages only a small fraction of the total web traffic. Moreover, we completely neglected any locality of reference, and made rather generous assumptions about the number of popular pages (.1% of the web!). Nonetheless, in spite of these rather pessimistic assumptions, the overall bandwidth levels are rather reasonable.