Scalable Consistency Maintenance in Content Distribution Networks

Anoop Ninan, Purushottam Kulkarni, Prashant Shenoy, Krithi Ramamritham and Renu Tewari†

Department of Computer Science, University of Massachusetts Amherst
{agn,purukulk,shenoy,krithi}@cs.umass.edu

†IBM Research Division, T J. Watson Research Center
tewarir@us.ibm.com

Abstract— **In this paper, we argue that cache consistency mechanisms designed for stand-alone proxies do not scale to the large number of proxies in a content distribution network and are not flexible enough to allow consistency guarantees to be tailored to object needs. To meet the twin challenges of scalability and flexibility, we introduce the notion of *cooperative consistency* along with a mechanism, called *clustered leases*, to achieve it. By supporting Δ consistency semantics and by using a single lease for multiple proxies, clustered leases allows the notion of leases to be applied in a flexible, scalable manner to CDNs. Further, the approach employs application-level multicast to propagate server notifications to proxies in a scalable manner. We implement our approach in the Apache web server and the Squid proxy cache and demonstrate its efficacy using a detailed experimental evaluation. Our results show a factor of 3.2 reduction in server message overhead and a 20% reduction in server state space overhead when compared to original leases, albeit at an increased inter-proxy communication overhead.**

I. Introduction

A. Motivation

THE past decade has seen a dramatic increase in the popularity and use of the World Wide Web. Numerous studies have shown that web accesses tend to be non-uniform in nature, resulting in (a) hot-spots of server and network load and (b) increases in the latency of web accesses. Content distribution networks have emerged as a possible solution to these problems. A *content distribution network (CDN)* consists of a collection of proxies that act as intermediaries between the origin servers and the end clients. Proxies in a CDN cache frequently accessed data from origin servers and serve requests for these objects from the proxy closest to the end-user. By doing so, a CDN has the potential to reduce the load on origin servers and the network and also improve client response times.

Architectures employed by a CDN can range from tree-like hierarchies [22] to clusters of cooperating proxies that employ content routing to exchange data [11]. From the perspective of endowing proxies with content, proxies within a CDN can either pull web content on-demand, prefetch popular content, or have such content pushed to them [9]. Mechanisms for locating the best proxy to service a user request range from Anycast to DNS-based selection. Regardless of the exact architecture and mechanisms, an important issue that must be addressed by a CDN is that of *consistency maintenance*. Since web pages tend to be modified at origin servers, cached versions of these pages can become inconsistent with their server versions. Using inconsistent (stale) data to service user requests is undesirable, and consequently, a CDN should ensure the consistency of cached data with the server by employing suitable techniques.

The problem of consistency maintenance is well-studied in the context of a single proxy and several techniques such as time-to-live (TTL) values [4], client-polling, server-based invalidation [3], adaptive refresh [18], and leases [21] have been proposed. In the simplest case, a CDN can employ these techniques at each individual proxy—each proxy assumes responsibility for maintaining consistency of data stored in its cache and interacts with the server to do so independently of other proxies in the CDN. Since a typical content distribution network consists of hundreds or thousands proxies, requiring each proxy to maintain consistency independently of other proxies is not scalable from the perspective of the origin servers (since the server will need to individually interact with a large number of proxies). Further, consistency mechanisms designed from the perspective of a single proxy (or a small group of proxies) do not scale well to large CDNs. The leases approach, for instance, requires the origin server to maintain per-proxy state for each cached object. This state space can become excessive if proxies cache a large number of objects or some objects are cached by a large number of proxies within a CDN. These arguments motivate the need for designing novel consistency mechanisms that scale to large CDNs and is the focus of this paper.

B. Research Contributions

In this paper, motivated by the need to reduce the load at origin servers and to scale to a large number of proxies, we (a) argue that delta consistency semantics are appropriate for CDNs because they allow the tailoring of consistency guarantees to the nature of objects and their usage, and (b) introduce the notion of *cooperative consistency* along with a mechanism, called clustered leases, to achieve it. Cooperative consistency enables proxies to cooperate with one another to reduce the overheads of consistency maintenance. By supporting delta consistency semantics and by using a single lease for multiple proxies, our clustered leases mechanism allows the notion of leases to be applied in a scalable manner to CDNs. Another advantage of our approach is that it employs application-level multicast to propagate server notifications of modifications to objects, which reduces server overheads. We address the various design issues that arise in a practical realization of clustered leases and then show how to implement the approach in the Apache web server and the Squid proxy cache using HTTP/1.1. Finally, we experimentally demonstrate the efficacy of our approach using trace-driven simulations and the prototype implementation. Our results show that clustered leases can reduce the number of server messages by a factor of 3.2 and the server state by 20% when compared to original leases, albeit at an increased proxy-proxy communication overhead.

The rest of this paper is structured as follows. Section II defines the problem of consistency maintenance in CDNs and presents our clustered leases approach. We examine various design issues in instantiating clustered leases in Section III. Section IV discusses the details of our prototype implementation. Section V presents our experimental results. Section VI discusses related work, and finally, Section VII presents some concluding remarks.

II. Cache Consistency in CDNs: Semantics, Mechanisms, and Architecture

A. *Delta Consistency: Consistency Semantics for Cached Objects*

Objects cached within a content distribution network need different levels of consistency guarantees depending on their characteristics and user preferences. For instance, users may be willing to receive slightly outdated versions of objects such as news stories and sports scores but are likely to demand the most up-to-date versions of "critical" objects such as financial information and stock prices. Typically, the stronger the desired consistency guarantee for an object, the higher the overheads of consistency maintenance. For reasons of flexibility and efficiency, rather than providing a single consistency semantics to all cached objects, a CDN should allow the consistency semantics to be tailored to each object or a group of related objects.

One possible approach for doing so is to employ Δ-consistency semantics [20]. Δ-consistency requires that a cached version of an object is never out-of-date by by more than Δ time units with its server version. The value of Δ determines the nature of the provided guarantee—the larger the value of Δ, the weaker the consistency guarantee (since the object could be out of date by up to Δ time units at any instant). An advantage of Δ-consistency is that it provides a quantitatively characterizable guarantee by virtue of providing an upper bound on the amount by which a cached object could be stale (unlike certain mechanisms that only provide qualitative guarantees). Another advantage is that it provides the flexibility of choosing a different value of Δ for each object, allowing the guarantee to be tailored on a per-object basis. Finally, strong consistency—a guarantee that a cached object is never out-of-date with the server version—is a special case of Δ-consistency with $\Delta = 0$.[1]

Due to the above advantages, in the rest of this paper, we assume a CDN that provides Δ consistency semantics. Next, we present a consistency mechanism to provide Δ consistency and then discuss its implementation in a CDN.

B. *Clustered Leases: A Cache Consistency Mechanism for CDNs*

A consistency mechanism employed by a CDN should satisfy two key requirements: (i) *scalability:* the approach should scale to a large number of proxies employed by the CDN and should impose low overheads on the origin servers and proxies, and (ii) *flexibility:* the approach should support different levels of consistency guarantees. We now present a cache consistency mechanism that satisfies these requirements. Our approach is based on a generalization of *leases* [10].

In the original leases approach [10], the server grants a lease to each request from a proxy. The lease denotes the interval of time during which the server agrees to notify the proxy if the object is modified. After the expiration of the lease, the proxy must send a message requesting a lease renewal. More formally, a lease is a tuple $\{O, p, d\}$ maintained by the server, where the server agrees to notify proxy p of all updates to an object O during time interval d.

The leases approach has two drawbacks from the perspective of a CDN. First, leases provide strong consistency semantics by virtue of notifying a proxy of *all* updates to an object. Maintaining strong consistency is expensive, and as argued earlier, not all objects cached within a CDN need such stringent guarantees. Second, leases require the server to maintain state for each proxy caching the object; the resulting state space overhead can be excessive for large CDNs. Thus, leases do not scale well to busy servers and large CDNs.

To alleviate these drawbacks, we generalize leases along two dimensions:

1. We add a *rate parameter* Δ to leases that indicates the rate, $1/\Delta$, at which the server agrees to notify a proxy of updates to an object. This enhancement allows a server to relax the consistency semantics provided by leases from strong consistency to Δ-consistency—a proxy is notified of updates at most once every Δ time units (instead of after every update) and no later than Δ time units after an update. Using $\Delta = 0$ reverts to the original leases approach (i.e., strong consistency), while a non-zero value of Δ allows the server to provide weaker consistency guarantees (and correspondingly reduces the number of notifications sent to a proxy).
2. We allow a server to grant a single lease collectively to a group of proxies, instead of issuing a separate lease to each

[1]Implementing true strong consistency requires the server to first send notifications to all proxies caching the object; the write commits only after receiving acknowledgements from these proxies.

individual proxy.[2] For each cached object, the proxy group designates a distinguished proxy, referred to as the *leader*, that is responsible for all lease-related interactions with the server. The leader of a group manages the lease on behalf of all the proxies in the group. Moreover, the server only notifies the leader upon an update to the object; the leader is then responsible for propagating this notification to other proxies in the group that are caching the object. Such an approach has two significant advantages: (i) it reduces the the amount of state maintained at a server (by using a single lease to represent a proxy group instead of an individual proxy); and (ii) it reduces the number of notifications that need to be sent by the server (by offloading some of notification burden to leader proxies).

We refer to the resulting approach as *clustered leases*. Formally, a clustered lease is a tuple $\{O, G, L, d, \Delta\}$ where the server agrees to notify the leader L representing proxy group G of any updates to the object O once every Δ time units for an interval d. While leases is a pure server-based approach to cache consistency, clustered leases require both the server and the proxy (especially the leader) to participate in consistency maintenance. Hence this approach is more scalable when compared to original leases, and thus, more suited to CDN environments.

C. Clustered Leases in a CDN: System Model

Before discussing the implementation of clustered leases in CDNs, we present the system model assumed in this paper. A content distribution network is defined to be a collection of proxies that cache content stored on origin servers. For the purposes of maintaining consistency, proxies within the CDN are assumed to be partitioned into non-overlapping groups referred to as *clusters* (issues in doing so are beyond the scope of this paper). Proxies within a cluster are assumed to cooperate with one another for maintaining consistency of cached objects. Cooperative consistency *is orthogonal to cooperative caching*—whereas the latter involves sharing of cached data to service user requests, the former involves cooperation solely for maintaining consistency of data cached by proxies within a cluster. Further, the organization of proxies into clusters is limited to consistency maintenance; a different overlay topology can be used for exchanging data and meta-data within the CDN. Each proxy in a cluster is assumed to maintain a directory of all objects cached in the cluster. The directory maintains the leader information for each cached object (and possibly other information required by the CDN). Several directory schemes such as hint caches [19] and bloom filters [6] have been proposed to efficiently maintain such information; any such scheme suffices for our purpose.

D. Operations of Clustered Leases in a CDN

Clustered leases can be instantiated as follows (see Figure 1 and Table I).

[2]In addition, it is also possible for a lease to collectively represent multiple objects. Techniques for doing so are studied in [21].

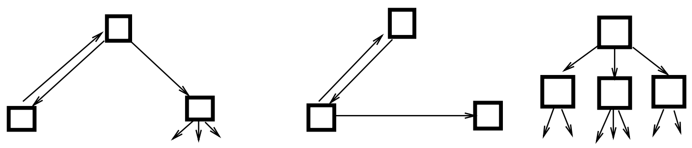


Fig. 1. Interactions between servers (S), leaders (L) and proxies (P) in clustered leases

First-time requests: When an object is requested for the first time within the cluster (i.e., upon a global cache miss), a leader needs to be chosen for the object. The proxy receiving the request runs a leader selection algorithm to pick a leader.[3] Different cached objects can have different leaders—the clustered leases approach attempts to distribute leader responsibilities across proxies in the cluster for load balancing purposes. Specific techniques for leader selection are discussed in Section III-A. After choosing a leader, the proxy issues a HTTP request to the server and piggybacks the leader information with the message; the message can also include optional information such as the desired rate parameter Δ. The requested object is then sent to the proxy and the lease is sent to the leader along with a copy of the object. As will be clear later, the presence of a copy of the object at the leader enables us to perform certain optimizations. The leader proxy then broadcasts a directory update to all proxies in the cluster indicating it is the designated leader for the object. The leader also maintains a *membership list* consisting of all proxies caching the object; the list is initialized to the proxy that requested the object. Figure 1(a) depicts these interactions.

From this point on, the leader is responsible for renewing the lease on behalf of proxies in the cluster and for terminating the lease when proxies are no longer interested in the object. Policies for doing so are discussed in Section III-C.

Subsequent requests: For each subsequent request to the object within the cluster, a proxy first examines its local cache. In the event of a cache hit, the proxy services the request using locally cached data. In the event of a local cache miss, the proxy can pursue one of several possible alternatives. It can either fetch the object from the server or consult its directory for a list of proxies caching the object and fetch the object from one such proxy (the exact proxy that is chosen may depend on the information in the directory and metrics such as proximity). Since the focus of our work is on consistency maintenance, the clustered leases approach does not mandate the use of cooperative caching or require a particular policy for cooperative caching—the proxy is free to fetch the object from from any entity that

[3]It is also possible to do a DNS lookup and have the DNS server pick a leader. We also note that leader selection in CDNs is distinct from leader election algorithms developed by the distributed systems community.

TABLE I
DESIGN CONSIDERATIONS

Event	*Design Decision*	*Discussed in*
Issue a new lease	Choose a leader Choose d and Δ	Sec III-A Sec III-B
Lease expiry	Lease renewal policy	Sec III-C
Object changes	Send update or invalidate	Sec III-D
Global cache miss	see "issue a new lease" event	–
Local cache miss	Update membership list	Sec II-D

has the object, including the server. The only requirement imposed by clustered leases is that the proxy notify the leader of its interest in the object. The leader then updates the membership list for the object and starts forwarding any subsequent notifications from the server to this proxy. Figure 1(b) depicts these interactions.

Observe that a proxy can optimize the overheads of the above operations by just fetching the object from the leader. Since the leader always caches the most recent version of the object (recall that a copy of the object was sent to the leader), this eliminates the need to send two different messages, one to fetch the object and the other to notify the leader of this fetch.

Updates to the Object: In the event the object is modified at the server, each proxy caching the object needs to be notified of the update. To do so, the origin server first notifies the leader of each cluster caching the object, subject to the rate parameter Δ. The notification consists of either a cache invalidate or a new version of the object (see Section III-D for details). Each leader in turn propagates this notification to every proxy in the cluster caching the object (i.e., to all proxies in the membership list). Depending on the type of notification, proxies then either invalidate the object in the cache or replace it with the newer version. Our approach is equivalent to using *application-level multicast* for propagating notifications; the membership list and the leader constitute the "multicast group". Figure 1(c) depicts these interactions.

For simplicity, this paper assumes that the application-level multicast tree within a cluster is only two levels deep, spanning from servers to leaders and from leaders to proxies. Whereas a two level hierarchy suffices for small clusters (likely to be the common case), a multi-level tree is needed for large clusters. The clustered leases algorithm can be recursively extended to multi-level hierarchies as well. Due to space constraints, the generalized approach is discussed briefly in Section VII; the complete algorithm for multi-level clusters can be found in [17].

III. DESIGN CONSIDERATIONS FOR CLUSTERED LEASES

In this section, we discuss various design issues that arise when implementing clustered leases in a CDN. These include leader selection, selecting the lease duration and notification rate, policies for lease renewal and sending invalidations versus updates (see Table I).

A. Leader Selection

We consider two different policies for choosing a leader when an object is accessed for the first time within the cluster. In the simplest case, the proxy that receives this request can become the leader for the object. Since many web objects tend to be accessed by only one user [2], an advantage of this approach is that only one proxy is involved in consistency maintenance for such objects (since the proxy caching the object is also the leader). This results in lower communication overheads. A drawback, however, is that the approach has poor load balancing properties—leader responsibilities can become unevenly distributed if a small subset of proxies receive a disproportionate number of first-time requests. Additionally, if several proxies receive simultaneous first-time requests to an object, it is possible for multiple proxies to declare themselves the leader. Such duplication can be prevented using tie-breaking rules or by having the server perform additional error checks before issuing a new lease to a cluster.

An alternate approach is to employ a hashing function to determine the leader for an object. To illustrate, the leader could be determined based on the MD5 hash of the object URL (i.e., $L = MD5(URL) \bmod N$, where N is the number of proxies in the cluster). More complex hashing functions can take other factors, such as the current load on proxies, into account in addition to the URL [13]. An advantage of the hash-based approach is that it has good load balancing properties and results in a more uniform distribution of leader responsibilities across proxies. A limitation though is that it can impose a larger communication overhead than our first approach. Since the leader can be potentially different from proxies caching the object, additional directory updates, server notifications and lease management messages need to be exchanged between these proxies, which increases communication overheads. Section V quantitatively evaluates the tradeoffs of these two policies.

B. Choosing the Lease Duration and Notification Rate

Two key factors that influence the performance of clustered leases are the lease duration d and the rate parameter Δ. In a recent work, we investigated techniques for determining the lease duration for the original leases approach and proposed policies for computing d based on parameters such as object popularity, write frequency, and server/network load [5]. Since similar policies can be employed for computing the lease duration d in CDNs, we do not consider this issue any further.

The notification rate can either be specified by the user (or proxy), or computed by the server. In the former approach, the end-user or the proxy specifies a tolerance Δ based on the desired consistency guarantee The server then grants a lease with this Δ if it has sufficient resources to meet the desired tolerance. In the latter approach, the server computes an appropriate notification rate based on various system parameters while issuing a new lease. For instance, the server could compute Δ based on the server or network load. Rather than rejecting a request for a lease during periods of heavy load, the server could continue to grant leases but provide weaker guarantees (i.e., use a larger Δ). To illustrate,

$$\Delta = \begin{cases} 0 & \text{load} < \text{LWM} \\ c \cdot load & \text{LWM} \leq \text{load} < \text{HWM} \\ d & \text{load} \geq \text{HWM} \end{cases} \qquad (1)$$

where c is a constant and LWM and HWM denote low and high watermarks (thresholds), respectively. Here the server notifies leaders of all updates at low loads. Δ is increased linearly with the load at moderate utilizations and is finally set to the lease duration at high loads (d is the least possible notification rate, since at least one update should be sent in each lease duration).

C. Eager versus Lazy Lease Renewals

Another important issue in clustered leases is the policy for lease renewals. Since the leader manages the lease on behalf of all proxies in the cluster, it needs to decide whether and when to renew a lease. Two different renewal policies are possible:

• *Eager renewals:* In this policy, the leader continuously renews the lease upon each expiration until it is explicitly notified by proxies not to do so. This approach requires each proxy to track its interests in locally cached objects and send a "terminate lease" message to the leader when it is no longer interested in an object. For instance, a proxy can send such a message if it hasn't received a request for an the object for a long time period. Upon receiving such a message, the leader removes that proxy from its membership list and stops forwarding server notifications to the proxy. Consequently, a "terminate lease" message is equivalent to a "leave" message from the application-level multicast group. When the membership list becomes empty (i.e., all proxies caching the object send terminate messages), the leader stops renewing the lease. It then broadcasts a directory update to all proxies indicating that it has relinquished leader responsibilities for the object.

• *Lazy renewals:* Here, the leader does not renew a lease upon expiration. Instead it sends a "lease expired" message to all proxies caching the object; proxies in turn flag the object as "potentially stale". Upon receiving a subsequent request for this object, a proxy sends an if-modified-since (IMS) request to the server. The server then issues a new lease for the object, if one hasn't already been issued, and responds to the IMS request by sending a new version of the object if the object was modified in the interim. The lease, if one is issued, is sent to the leader.

In the lazy approach, proxies do not need to track their interest in each cached object. Moreover, since leases are renewed lazily and only when an object is accessed, the approach is efficient for less popular objects (e.g., "one-timers"). The drawback though is that each request received after a lease expiration involves an additional interaction with the server (in the form of an IMS request). In contrast, the eager approach only involves leader-server interactions after lease expiry; individual proxies do not need to interact with the server, which reduces server load.

D. Propagating Invalidates versus Updates

Upon modification to an object, the server notifies each leader proxy with an active lease (subject to the rate parameter Δ). As explained earlier, this notification consists of either a cache invalidate or an updated (new) version of the object. Sending a cache invalidate causes a proxy to delete the object from its cache; a subsequent request requires the proxy to fetch the object from the server (or from another proxy in the cluster if that proxy has already fetched the updated object). Thus, each request after a cache invalidate incurs an additional delay due to this remote fetch. No such delay is incurred if the server sends out the new version of the object upon a modification.[4] In such a scenario, subsequent requests can be serviced using locally cached data. A drawback, however, is that sending updates incurs a large network overhead (especially for large objects). This extra effort is wasted if the object is never subsequently requested at the proxy. Consequently, cache invalidates are better suited for less popular objects, while updates can yield better performance for frequently requested objects. Observe that sending invalidates is equivalent to a *lazy update* policy at proxies, while sending new versions of objects amounts to *eager updates*.

A server can dynamically decide between invalidates and updates based on the characteristics of an object. One policy is to send updates for objects whose popularity exceeds a threshold and to send invalidates for all other objects. Although a server does not have access to the actual request stream at proxies to compute object popularities, it can estimate the popularity based on lease renewals. A continuously renewed lease is an indication that the object is popular within a cluster. Hence, the server can send updates for objects whose leases have been renewed at least τ consecutive times (τ is a threshold). Using $\tau = 0$ causes only updates to be sent, whereas $\tau = \infty$ causes only invalidates to be sent; an intermediate value of τ allows the server to dynamically choose between the two based on the object popularity. A more complex policy is to take both popularity and object size into account. Since large objects impose a larger network transfer overhead, the server can use progressively larger thresholds for such objects (the larger a object, the more popular it needs to be before the server starts sending updates).

IV. Implementation Considerations

We have implemented the clustered leases algorithm in the Squid proxy cache and the Apache web server. Our

[4]Security and authentication issues in doing so are beyond the scope of this paper.

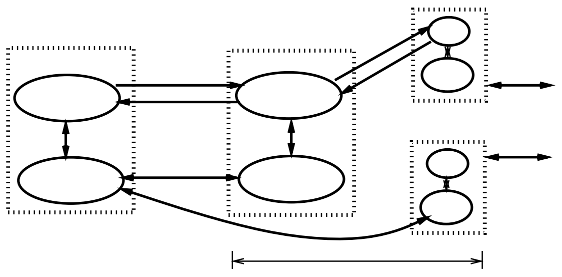


Fig. 2. Implementation architecture. The figure depicts the architecture of a single cluster. Each CDN will have a number of such clusters.

implementation is based on HTTP/1.1, which allows user defined extensions as part of the request/response header. We use these header extensions to enable proxies to request and renew leases from a server. To do so, lease requests and responses are piggybacked onto normal http requests and responses. Lease renewals and invalidation requests are also sent as request header extensions. The exact HTTP grammar for lease requests, renewals and invalidations is described in [17].

For simplicity and modularity, our implementation separates lease management functionality from the serving of web requests. Lease management at the server is handled by a separate lease server (`leased`). Such an architecture results in a clean separation of functionality between the Apache server, which handles normal http processing, and the lease server which handles lease processing and maintains all the state information (see Figure 2). Whenever the Apache server receives a lease grant/renewal request piggybacked on a http request, it forwards the former to the lease server for further processing. The lease duration d and the rate parameter Δ are computed using policies listed in [5] and Section III-B. The http response is then sent back to the client (proxy), while the lease is sent to the leader. Invalidation requests are handled similarly—the web server forwards the request to the lease server, which then sends invalidations to all leaders with active leases. Leaders forward the invalidations to all proxies caching the object as described below.

Analogous to the web server architecture, our implementation in Squid consists of two components–the proxy cache and the lease handler—that separate the caching functionality from lease management. The lease handler (LH) can either act as a leader or as a client. In the former case, the lease handler maintains a membership list of all proxies caching the object and forwards notifications from the server to this list. The lease handlers at member proxies are responsible for tracking object popularities and sending lease terminate messages to the leader for cold objects. Server failures and/or network partitions can be handled at the leader by exchanging heartbeat messages [15] or by maintaining a persistent TCP connection with the server—a broken connection indicates a failure and requires cached objects to be invalidated within Δ time units.

V. Experimental Evaluation

In this section, we demonstrate the efficacy of clustered leases by (i) comparing the approach with the original leases from the perspective of scalability, (ii) evaluating the tradeoffs of various policies described in section III and (iii) quantifying the implementation overheads of clustered leases. We employ a combination of trace-driven simulation and prototype evaluation for our experiments. We use simulations to explore the parameter space along various dimensions and use our prototype to measure implementation overheads (an aspect that simulations don't reveal). In what follows, we first present our experimental methodology and then our experimental results.

A. *Experimental Methodology*

A.1 Simulation Environment

We have designed an event-based simulator to evaluate the efficacy of clustered leases. The simulator simulates one or more proxy clusters within a CDN. Each proxy is assumed to receive requests from a large number of clients. Cache hits are serviced using locally cached data. Cache misses involve a remote fetch and are serviced by fetching the object from the leader (if one exists) or from the server. The directory maintained by the proxy is used to make this decision. Our simulator supports all policies discussed in Section III for leader selection, server notifications, lease renewals and rate computations.

Our experiments assume that each proxy maintains a disk-based cache to store objects. We assume each proxy cache is infinitely large—a practical assumption, since disk capacities today are in tens of gigabytes and a typical proxy can employ multiple disks. Data retrievals from disk (i.e., cache hits) are modeled using an empirically derived disk model with a fixed OS overhead added to each request. For cache misses, data retrieval over the network are modeled using the round trip time, available network bandwidth and the object size. The network latency and bandwidth between proxies and leaders is assumed to be 75ms and 500KB/s, while that between proxies and origin servers is 250ms and 250 KB/s. Although actual network latencies and bandwidths vary with network conditions, the use of this simple network model suffices for our purpose (due to our focus on consistency maintenance rather than end-user performance). Due to space constraints, we only present results for a single cluster; we performed experiments with multiple clusters to verify that each cluster behaves similarly to other clusters from the perspective of consistency maintenance (see [17]). Unless noted otherwise, our experiments assume a default cluster size of 10 proxies and a lease duration of 30 minutes. We also assume that a leader always caches a copy of the object and this copy is updated upon a modification.

TABLE II
TRACE CHARACTERISTICS

Trace	Num Requests	Duration (secs)	Unique Objects	Num Writes
DEC	750000	42031	276914	17126
NLANR	750000	56518	393853	14385

A.2 Workload Characteristics

The workload for our experiments is generated using traces from actual proxies, each containing several hundred thousand requests. We use two different traces for our study; the characteristics of these traces are shown in Table II. The same set of traces are used for our simulations as well as our prototype evaluation (which employs trace replay). Each request in the trace provides information such as the time of the request, the requested URL, the size of the object, the client ID, etc. We use the client ID to map each request in the trace to a proxy in the cluster—all requests from a client are mapped to the same proxy. To determine when objects were modified, we considered using the last modified times as reported in the trace. However, these values were not always available. Since the modification times are crucial for evaluating cache consistency mechanisms, we employ an empirically derived model to generate modification times. Based on observations in [1], [12], we assume that 90% of all web objects change very infrequently (i.e., have an average lifetime of 60 days). We assume that 7% of all objects are mutable (i.e., have an average lifetime of 20 days) and the remaining 3% objects are very mutable (i.e., have a lifetime of 5 days). We partition all objects in the trace into these three categories and generate write requests and last modified times using exponentially distributed lifetimes. The number of synthetic writes generated for each trace is shown in Table II.

Next, we describe our experimental results.

B. Impact of Leader Selection Policies

To evaluate leader selection policies, we simulated a cluster of ten proxies that employed two different policies—the hash based policy and the “first proxy is leader” policy. Our experiment assumed eager lease renewals and notifications in the form of invalidations (leaders were sent updates, leaders forwarded invalidations). For each policy, we measured how evenly leader responsibilities were distributed across proxies in the cluster as well as the total control message overhead imposed. Figure 3(b) and (c) depict our results, while Figure 3(a) shows the number of requests processed by each proxy in the cluster (we only plot results for one of the traces due to space constraints. See [17] for complete results). As expected, the “first proxy is leader” scheme suffers from load imbalances since some proxies service a larger number of requests (and assume leader responsibilities for a correspondingly larger number of first-time requests). The figure also shows that there is a factor of 1.5 difference in load between the most heavily-loaded and the least-loaded proxy. In contrast, the hash-based policy shows better load balancing properties but imposes a larger communication overhead (since leaders can be different from proxies caching the object, requiring additional message exchanges). As shown in Figure 3(c)), the total increase in control message overhead is about 10% and the increase is primarily due to the lease terminate messages sent from proxies to leaders. Since a small (10%) increase in message overhead is tolerable to correct a potentially large imbalance (factor of 1.5), our results indicate that the hash-based leader selection is a better policy than the “first proxy is leader” approach.

C. Eager versus Lazy Renewals

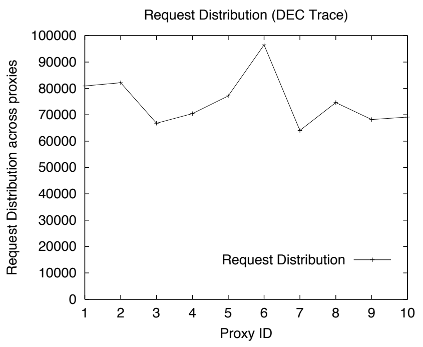
Next, we evaluate the impact of eager and lazy lease renewals on performance. Like in the previous experiment, we assume a cluster of ten proxies, each with an infinite cache. We vary the lease duration from 5 minutes to 5 hours and measure its impact on lazy and eager renewals. Figure 4 depicts our results. As shown on Fig 4(a), depending on the lease duration, eager renewals result in a 15-63% improvement in cache hit ratios; the hit ratio is lower for lazy renewals since requests arriving after a lease expiry trigger an IMS request to the server. The higher hit ratios for eager renewals are at the expense of an increased control message overhead (see Figure 4(b)). The message overhead is 33-175% higher and is primarily due to extra lease renew and terminate messages. The overhead for both policies decreases with increasing lease durations (since longer leases require fewer renewals). Finally, Figure 4(c) plots the state space overhead of the two policies; as expected, eager renewals result in a larger number of active leases at any instant, causing a 3-9% increase in state space overhead.

An important factor governing the performance of the eager renewals is the lease termination policy—the policy employed by member proxies to notify the leader that they are no longer interested in the object. As shown in Figure 5, the larger the period of inactivity before which a “terminate lease” message is sent, the larger the state space overhead at the server and the larger the control message overhead (since the leader continuously renews leases until such a message is received).

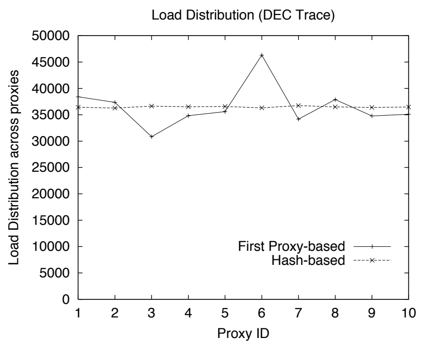
Thus, the two policies show a clear tradeoff—eager renews yield better hit ratios and response times at the expense of a larger control message overhead and a slightly larger state space overhead. Depending on whether user performance or network/server overheads are the primary factors of interest, one policy can be chosen over the other.

D. Server Notifications: Invalidate versus Updates

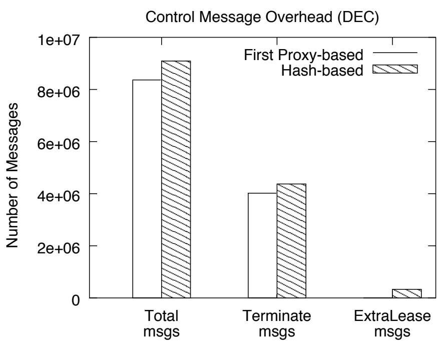
To understand the implications of sending invalidates versus updates, we considered a policy where the server sent updates for objects whose leases were renewed at least τ times in succession; invalidates were sent for the remaining objects. We varied τ from 0 to ∞ and measured its



(a) Request distribution

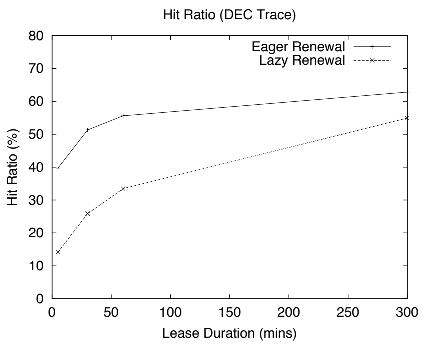


(b) Leader distribution

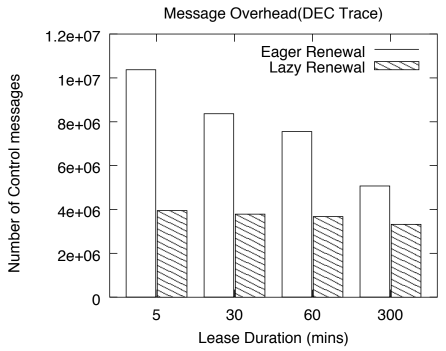


(c) Control message overhead

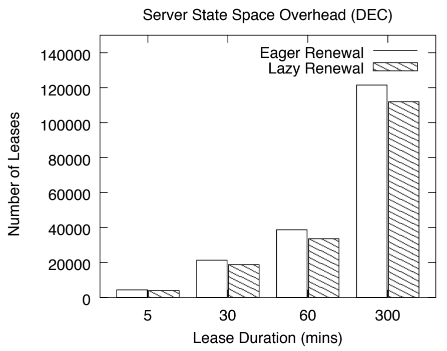
Fig. 3. Comparison of leader selection schemes



(a) Cache hit ratio

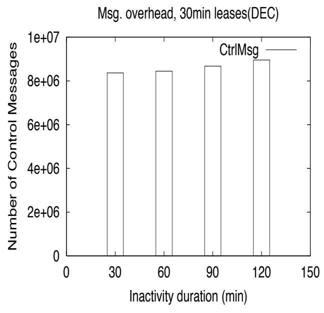


(b) Control message overhead

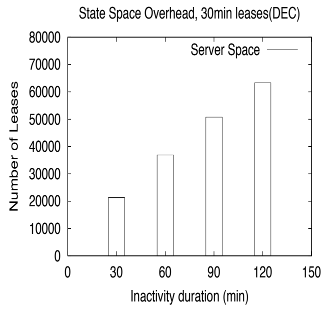


(c) State space overhead

Fig. 4. Comparison of lease renewal policies.



(a) Control message overhead



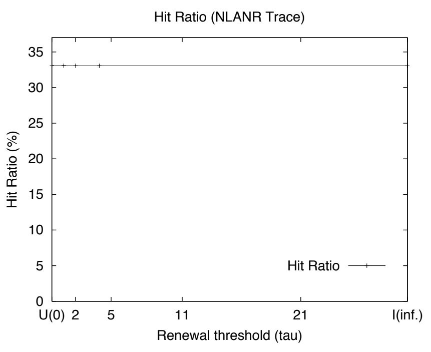
(b) State space overhead

Fig. 5. Evaluation of lease termination policies.

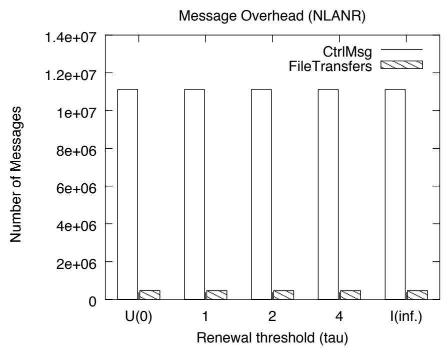
impact on the cache hit ratio and the control message overhead. Figure 6 shows that the notification policy has a negligible impact on the cache hit ratio ($<$ 1% reduction as τ increases from 0 to ∞). The control message overhead increases slightly (by about 1%) with increasing τ. This small increase is due to an increase in the number of invalidates, each of which triggers an HTTP request upon a subsequent user request. To better understand this behavior, Figure 6(c) plots the percentage of updates and invalidates sent for different τs; the percentage of objects accessed subsequent to a server notification is also shown. As shown, when $\tau = \infty$ (i.e., the invalidate-only scenario), only 5% of the invalidated objects are accessed subsequently. This indicates that a vast majority of the objects are never accessed after a server notification and sending updates for such objects results in wasted network transfers. Thus, our results show that updates should be sent only for very popular objects, which can be achieved using a large τ. More generally, our analysis of read and write frequencies has shown that updates are advantageous when the write frequency is (i) less than 3 times the read frequency for small objects and (ii) less than the read frequency for large objects [17].

E. Impact of the Notification Rate

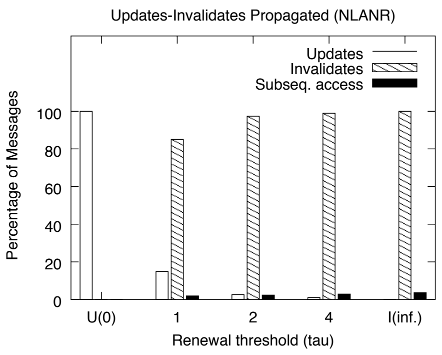
To understand the impact of the notification rate, we varied Δ from 5 seconds to 30 minutes and measured the impact on the number of notifications (invalidates) sent by the server (the leases duration was fixed at 30 minutes). As shown in Figure 7(a), the number of notifications drops by an order of magnitude with increasing Δs. This indicates that an appropriate choice of Δ can result in substantial savings at the server, albeit at the expense of weaker consistency guarantees. Next we considered a policy where the server computes Δ based on the load as explained in Equation 1; the server state space overhead is used as an indicator of the load. Note that Δ is computed based on the server load only at the beginning of a lease; once picked, Δ does *not* change for that lease until lease expiry. We var-



(a) Cache hit ratio



(b) Message overhead



(c) Num. of notifications

Fig. 6. Comparison of server notification policies.

ied the high and low watermarks in Eq. 1 and measured its impact on Δ. Figure 7(b) shows the variation in the server load over a 15 hour period, while Figure 7(c) plots the corresponding value of Δ used for new leases and renewals. The figure shows that the value of Δ closely matches the variation in the server load. Further, depending on the low and high watermarks used, the server uses $\Delta = 0$ during periods of low load and increases Δ to its maximum value (i.e.the lease duration) during periods of heavy load. Thus, an intelligent choice of Δ helps provide the desired level of consistency guarantee while lowering server overheads.

F. Scalability Issues: Comparison with Original Leases

To compare clustered leases with original leases, we consider a cluster of 20 proxies. To permit a fair comparison, other than the cache consistency mechanism, all simulation parameters are kept identical across our two experiments, the first involving clustered leases and the second employing the original leases approach. The lease duration is set to 30 minutes and $\Delta = 0$. Figure 8 and Table III depict our results. As expected, the number of leases managed by the server decreases when clustered leases is used (since each lease represents multiple proxies, fewer leases are needed). The reduction in state space overhead is 20% (see Table III); the reduction is smaller than expected since a large number of objects in the workload are requested by only one proxy and clustered lease do not provide any benefits in such scenarios. The number of server notifications, however, is smaller by a factor of 3.2 indicating that clustered leases successfully offload the burden of sending notifications to leader proxies, thereby improving scalability. These reductions come at the expense of having to maintain a directory of cached objects and an increased control message overhead due to directory updates. This results in a factor of 6.6 increase in message overhead for a 20 proxy cluster—the directory update overhead is proportional to the number of proxies in the cluster when application-level multicast (i.e., unicast) is used (see Figure 8). The use of IP-multicast, instead of application-level multicast, to send directory updates can help lower this overhead (since IP-multicast is more efficient than unicast). Also note that

TABLE III
COMPARISON WITH ORIGINAL LEASES (DEC)

	Clustered Leases	Leases
Total no. of leases	311600	387210
Num active leases	23038	28653
Server notifications	2267	7319

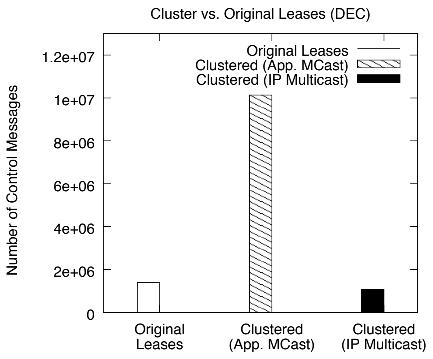
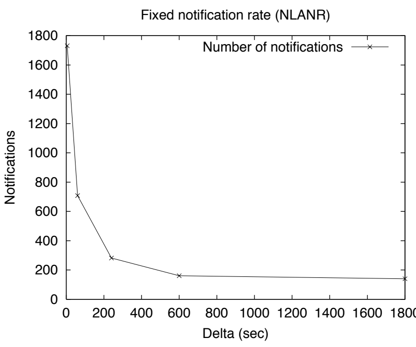


Fig. 8. Control message overhead for clustered and original leases.

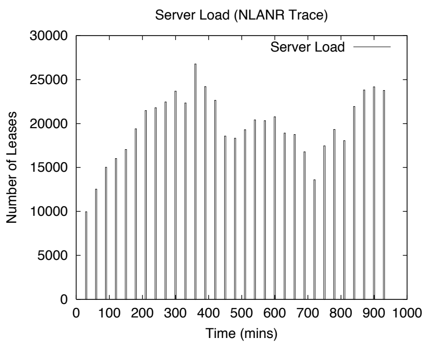
each unique Δ value associated with an object needs its own application level multicast group; a server can reduce the number of multicast groups by restricting itself to a small set of Δs. Thus, we conclude that clustered leases do indeed enhance scalability from the perspective of the server (in terms of the state space and server message overhead), albeit at the expense of increased leader-proxy communication overhead.

G. Implementation Overheads

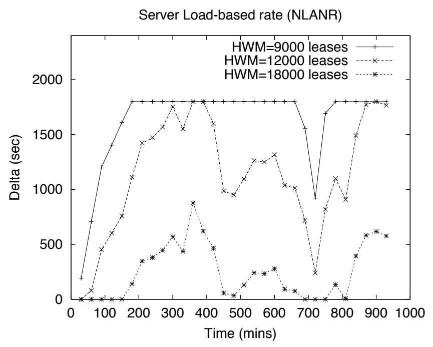
Whereas the preceding sections examined the efficacy of clustered leases using simulations, in this section we study the overheads of various operations needed for consistency maintenance. The testbed for our experiments consists of the lease-enhanced Apache web server, a cluster of four Squid proxy caches and a client workload generator, all of which run on a cluster of Linux PCs. Each PC in our exper-



(a) Effect of Δ



(b) Server load



(c) Computed Δ

Fig. 7. Impact of the notification rate.

TABLE IV
IMPLEMENTATION OVERHEADS (NLANR)

Proxy Overheads		Server Overheads	
Event	Time (ms)	Event	Time (ms)
Dir. lookup	0.052	Grant lease	0.64
Dir. update	0.056	Renew lease	0.28
Dir. broadcast	2.7	Inv. to Leader	3.36
Renew lease	2.65		
Inv. to Proxy	0.565		

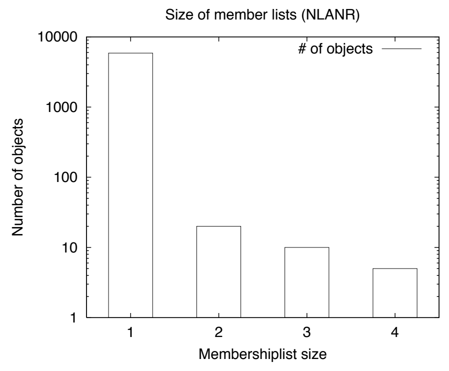


Fig. 9. Distribution of membership list sizes.

iment is a 700MHz Pentium III with 512MB RAM, interconnected by 100 Mb/s switched ethernet. The client workload generator employs trace replay and uses the traces described in Table II. To do so, it maps each URL in the trace to a unique object stored on the server of approximately the same size. Further, like in our simulations, each end-host in the trace is bound to a fixed proxy cache using a hashing function. The proxy and the server maintain consistency using clustered leases as described in Section IV. We measured the overhead of various lease management operations at the server and the proxies over the duration of the trace. Table IV and Figure 9 list our results. As shown in the table, the overhead of granting and renewing leases is very small (order of milliseconds). Similarly directory updates and server notifications (invalidates) can be propagated efficiently to proxies in the cluster (clearly these overheads depend on the number of proxies in the cluster and number of proxies that caching an object, respectively). Figure 9 plots the distribution of membership list sizes at leader proxies. As shown, the number of active leases in the cluster at any instant is in the order of few thousands. Moreover most leases have only one proxy in the membership list; the membership list has multiple proxies only for a small number of (popular) objects. Together, these results indicate that clustered leases can be implemented efficiently in web servers and CDN proxies.

VI. RELATED WORK

Recently several cache consistency mechanisms have been developed for single proxies [3], [4], [5]; as argued earlier, these mechanisms do not scale well to proxies in a CDN. Three recent efforts have focused on the issue of scalability [15], [21], [22]. We discuss each in turn.

A cache consistency mechanism for hierarchical proxy caches was discussed in [22]. The approach does not propose a new consistency mechanism, rather it examines issues in instantiating existing approaches into a hierarchical proxy cache using mechanisms such as multicast. They argue for a fixed hierarchy (i.e., a fixed parent-child relationship between proxies), whereas we allow different proxies to be leaders for different objects. In addition to consistency, they also consider pushing of content from servers to proxies.

Mechanisms for scaling leases are studied in [21]. The approach assumes volume leases, where each lease represents *multiple objects* cached by a stand-alone proxy. In contrast, we focus on CDNs and employ clustered leases where a lease can represent *multiple proxies*. They examine issues such as delaying invalidations until lease renewals, whereas we employ a formal model—Δ consistency—for propagating invalidations. Δ consistency allows a separation of the notification frequency from the lease duration, providing additional flexibility to the server. They also discuss prefetching and pushing of lease renewals. Our renewal policies are more complex, since leaders need to interact with member proxies to decide on renewals.

The web cache invalidation protocol (WCIP) is an attempt to standardize the propagation of server invalidations using application-level multicast [15]. The focus of this effort is on a protocol for propagating invalidations; the approach is agnostic of the actual cache consistency mechanism employed by proxies. In contrast, our work focuses on cache consistency mechanisms and semantics, and we are less concerned about the protocol (i.e., message formats) used for sending invalidations. Indeed, our prototype implementation could have employed WCIP instead of HTTP for sending invalidations.

Finally, numerous studies have focused on specific aspects of cache consistency or content distribution. For instance, piggybacking of invalidations [14], the use of deltas for sending updates [16], an application-level multicast framework for internet distribution [8] and the efficacy of sending updates versus invalidates [7] have all been studied. These efforts complement our work and can coexist with our approach.

VII. Concluding Remarks

In this paper, we argued that existing consistency techniques are not suitable for CDN environments. To alleviate this drawback, we proposed the notion of cooperative consistency and a mechanism called clustered leases to achieve it. Clustered leases meets the twin goals of flexibility and scalability by (i) employing Δ consistency semantics, (ii) using a single lease to represent multiple proxies and (iii) using application-level multicast to propagate server notifications. We implemented our approach into a prototype web server and proxy cache and demonstrated its efficacy via an experimental evaluation. Although our experiments assumed a single cluster with a two-level hierarchy, neither our approach nor our experiments are limited by these assumptions. The generalized clustered leases approach and experimental results for multiple proxy clusters and multi-level hierarchies are presented in [17]. Briefly, multi-level hierarchies require multiple proxies (not just the leader) to participate in propagating server notifications. The exact benefits for origin servers and the overheads of inter-proxy communication depend upon the span-out of a node, the depth of the hierarchy and the size of the cluster [17].

References

[1] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of Infocom'99, New York, NY*, March 1999.

[2] M. Busari and C. Williamson. On the Sensitivity of Web Proxy Cache Performance to Workload Characteristics. In *Proceedings of IEEE Infocom'01, Anchorage, Alaska*, April 2001.

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

[4] V. Cate. Alex: A Global File System. In *Proceedings of the 1992 USENIX File System Workshop*, pages 1–12, May 1992.

[5] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of the IEEE Infocom'00, Tel Aviv, Israel*, March 2000.

[6] L. Fan, P. Cao, J. Almeida, and A Z.Broder. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. In *Proceedings ACM SIGCOMM'98, Vancouver, BC*, pages 254 – 265, September 1998.

[7] Z. Fei. A Novel Approach to Managing Consistency in Content Distribution Networks. In *Proceedings of the 6th Workshop on Web Caching and Content Distribution, Boston, MA*, June 2001.

[8] P. Francis. Yoid: Extending the Internet Multicast Architecture. Technical report, AT&T Center for Internet Research at ICSI (ACIRI), April 2000.

[9] S. Gadde, J. Chase, and M Rabinovich. Web Caching and Content Distribution: A View From the Interior. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, 2000.

[10] 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.

[11] M. Gritter and D R. Cheriton. An Architecture for Content Routing Support in the Internet. In *Proceedings of the USENIX Symposium on Internet Technologies,San Francisco, CA*, March 2001.

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

[13] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth ACM Symposium on Theory of Computing*, 1997.

[14] B. Krishnamurthy and C. Wills. Study of Piggyback Cache Validation for Proxy Caches in the WWW. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems, Monterey, CA*, pages 1–12, December 1997.

[15] D. Li, P. Cao, and M. Dahlin. WCIP: Web Cache Invalidation Protocol. IETF Internet Draft, November 2000.

[16] J C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proceedings of ACM SIGCOMM Conference*, 1997.

[17] A. Ninan. Maintaining Cache Consistency in Content Distribution Networks. Master's thesis, Department of Computer Science, Univ. of Massachusetts, June 2001.

[18] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining Temporal Coherency of Virtual Warehouses. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS98), Madrid, Spain*, December 1998.

[19] R. Tewari, M. Dahlin, H M. Vin, and J. Kay. Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, June 1999.

[20] B. Urgaonkar, A. Ninan, M. Raunak, P. Shenoy, and K. Ramamritham. Maintaining Mutual Consistency for Cached Web Objects. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), Phoenix, AZ*, pages 371–380, April 2001.

[21] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering Server-driven Consistency for Large-scale Dynamic Web Services. In *Proceedings of the 10th World Wide Web Conference, Hong Kong*, May 2001.

[22] H. Yu, L. Breslau, and S. Shenker. A Scalable Web Cache Consistency Architecture. In *Proceedings of the ACM SIGCOMM'99, Boston, MA*, September 1999.