Maintaining Cache Consistency in Content Distribution Networks *

Anoop George Ninan

Department of Computer Science,
University of Massachusetts,
Amherst MA 01003.
agn@cs.umass.edu

Abstract

While several mechanisms for maintaining consistency in single proxy caches exist today, not as much research has addressed ways in which such techniques may be extended to a cluster of proxy caches. If such techniques are not developed and deployed carefully, the overheads involved in maintaining cache consistency in large-scale systems such as Content Distribution Networks (CDNs) increases by several orders of magnitude with increase in the number of proxies. The goal of developing such algorithms is therefore to come up with ways by which consistency guarantees can be provided while keeping network and server resource usage low. In this paper, we present efficient ways to maintain cache consistency in CDNs using **leases***. We address the following issues: (i) Selection of a leader proxy, via which updates and/or invalidates may be propagated to other proxies (ii) The effects of co-operation between proxy caches (iii) Eager vs. Lazy renewal of leases (iv) Policies for intelligent dissemination of updates and/or invalidates. (v) Means of adapting to changing server and network loads and yet providing consistency guarantees. (vi) Multi-level hierarchical proxy organization and (vii) The scalability achieved with increasing number of independent proxy clusters.*

1 Introduction

1.1 Motivation

The Internet and the World Wide Web have seen tremendous growth in the last decade. This growth has made it possible for millions of users to gain access to geographically distributed web content. However, due to the magnitude of the increase in the user population and the non- uniformity of content accesses, popular objects (especially those which change frequently), create server and network overload, and thereby significantly increase the latency for content access [40]. *Proxy caching* is a commonly used technique to reduce content access latencies. A proxy server (or web cache) sits in between clients and servers. Requests from clients are directed to the proxy server. If the proxy server receives a request for an object it does not cache (a *cache miss*), it obtains the object requested from the server, caches a copy of this object locally, and then serves the object to the requesting client. On the other hand, if the proxy gets a request for an object that exists in its cache (a *cache hit*), it directly serves the request without contacting the server. As proxy caches are commonly deployed on the edges of networks and closer to clients, proxy

*Master's Thesis - under the guidance of Prof. Prashant Shenoy and Prof. Krithi Ramamritham. Submitted to the Department of Computer Science, UMASS Amherst. This research was supported in part by a NSF Career award CCR-9984030, NSF grants ANI 9977635, CDA-9502639, EIA-0080119, Intel, IBM, EMC, Sprint, and the University of Massachusetts.

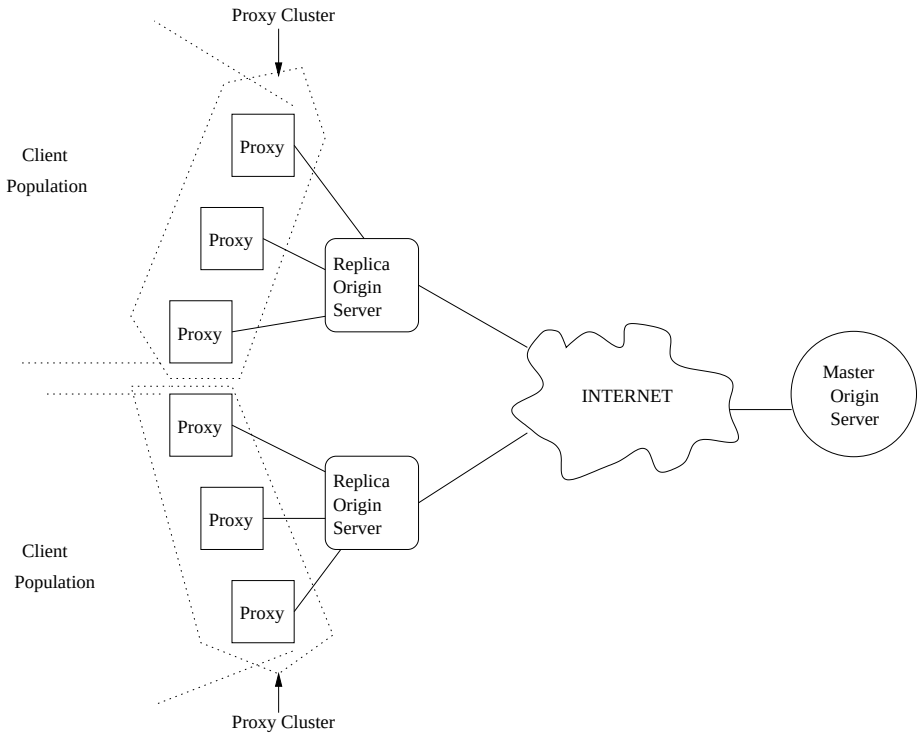


Figure 1: A typical CDN Architecture

caching is also known to reduce network bandwidth usage [40, 35]. Proxy servers are sometimes deployed closer to the servers than the clients (a technique called *reverse proxy caching*)[8, 3]. This technique reduces the load on the server in terms of the number of requests processed. Given the development of such techniques, several organizations have deployed such *proxies* to enable lower latencies or better *user response times*, reduce server load etc. However, with the exponential growth of the World Wide Web, these deployments were not good enough and hence motivated the need for scalable caching solutions. This gave birth to the idea of deploying networks of proxies that provide service to content providers (such as news sites), the service provided being the distribution of information to clients that request the content. Such networks are what are known today as *Content Distribution Networks* (CDNs). Organizations that have deployed such networks include Akamai, Speedera etc [1, 2].

A CDN comprises a few replica origin servers at one level and several intermediary proxies at a lower level, via which content is served to clients requesting objects from origin servers. *Origin servers* are those servers for which a CDN provides service to. *Replica origin servers* belong to a CDN and are those servers across which content from origin servers are replicated. *Intermediary proxies* in CDNs are those entities via which content is served from replica origin servers to requesting clients. See figure 1 for a typical CDN architecture. Tyipcally in a CDN, a client request to an origin servers is re-directed to a replica server, which then transfer the requested content to the client via intermediary proxies.

An important problem with caching of objects is maintaining the *freshness or consistency* of these objects. While proxy caching has its benefits, the corresponding versions of the cached objects, at the origin servers can change (e.g., news items, stock quotes etc). If proxies do not employ cache consistency mechanisms, they stand the risk of

serving *stale content* or content that is *not up-to-date* to requesting clients. Therefore, on a larger scale, if CDNs do not employ cache consistency mechanisms that ensure that proxies are *in-sync* with replica servers, CDNs also stand the same risk. However, maintenance of consistency does not come for free. In addition to processing client requests and caching objects, proxies and/or servers now have to do extra work in order to maintain consistency of objects. Some of the common consistency mechanisms include *proxy polling* [19] (proxies poll servers on behalf of the clients to retrieve changing content if any), or the server-based mechanisms [10, 46] (servers push content as they change to proxies that cache them). Note that hybrid policies also exist. This extra work generates network traffic and also involves keeping state at the server (of proxies that cache its objects) if the mechanism is server-based.

While a lot of work has looked into the development of web cache consistency mechanisms, most of these mechanisms have been studied for deployment in single proxy caches. Several of such mechanisms have indeed been shown to be effective for single proxies, but such mechanisms do not scale with the growth in the number of proxies, as the communication overhead generated in the network and the state space overhead at the proxies and servers are most likely to be prohibitive. Therefore with the increase in user populations and content requested, networks such as CDNs are bound to grow (in terms of the number of proxies for instance) to meet demands. Hence there is a need to develop *scalable* and *efficient* solutions for cache consistency to deal with such growths.

While studies have addressed viable CDN architectures [48, 8, 4, 33, 12, 43, 37, 16], the role of a proxy in a CDN [32], cooperative caching to improve hit ratio and response times [44, 5, 11, 43, 12, 17, 28, 34, 38, 49], load balancing amongst proxies [23, 24, 33], redirection schemes and other performance issues [21, 20, 31], object replication strategies in such networks [22], and how prefetching affects performance in CDNs [42], very few have addressed consistency issues in CDNs. [47, 46, 48, 45, 33] are few of the existing studies that have looked into the area of managing consistency in large-scale systems. While these techniques are possible solutions to the problem, it is not clear whether these solutions are indeed efficient solutions for CDNs, and how they should be deployed. Also, these solutions address the more general class of large-scale systems, and not CDNs in specific. Moreover, as CDNs provide a *service* of content distribution to origin servers, it is not clear whether these solutions can indeed provide *consistency guarantees* to clients requesting content.

We claim that efficient consistency algorithms for such large networks should not only provide consistency guarantees, but should do so efficiently, and should be scalable i.e., keep user response times low by keeping hit ratio high, and also keep network and server resource usage low, even with the large growth of such networks. In this work, we present one such technique based on *leases*.

1.2 Relation to Previous Work

In earlier work, we developed adaptive proxy-based approaches for maintaining individual and mutual consistency of web objects [41]. We also developed a server-based technique called *Adaptive Leases* for maintaining strong consistency on the Web [10]. Before we proceed, let us define what leases are in the context of our work. A **lease** for an object O, is a contract (a tuple $< s, d >$ - where s is the lease start time and d is the duration of the lease) given by a server to a proxy that requests an object, which states that $\forall t, s \leq t \leq s + d$, the server will propagate changes to the object to a proxy. Two of the biggest concerns in providing consistency guarantees are the control message overhead (communication between participating entities in order to maintain consistency - common with

poll-based mechanisms) and the state space overhead at the server for the same purpose (common with server-based mechanisms). We can imagine these overheads forming a spectrum which the *leases* technique can potentially span. If leases of zero duration are issued, the technique boils down to a poll-based technique and if leases of infinite duration is issued, the technique becomes a purely server-based technique. Therefore one can achieve a balance of these trade-offs by the intelligent computation of a lease duration [10]. We extend some of the ideas in our previous work with leases to develop adaptive, efficient and scalable consistency solutions for large networks such as CDNs with proxies grouped into clusters.

1.3 Research Contributions

As the number of proxies grow, it is necessary to take load off the server (in terms of the number of requests processed and the amount of state maintained) and thus avoid *swamping* the server or the creation of *hot spots*. A straight-forward way is to organize proxies into a hierarchy, since hierarchies are synonymous with scalability. However, [40] shows that multi-level hierarchies worsen user response times by a factor of about 2; this is mainly due the fact the requests received by leaf proxies (those at the lowest level) are forwarded to their immediate parents if they do not cache the object and if neither of the proxies in the hierarchy cache the requested object, this request forwarding continues all the way all to the server before a response is finally generated. Hence we borrow their idea of *hint caching*, and organize a cluster of proxies into logical, per-object, one-level hierarchies for efficient dissemination of updates and/or invalidates. Such hierarchies take the burden off the server in terms of the number of proxies it may need to communicate to in order to propagated updates/invalidates (as more than one proxy may cache an object). At the root of these logical hierarchies exist what we refer to in the rest of this report as *leader proxies* or *leaders*. The other proxies caching the same object will henceforth be referred to as *member proxies* or *members*, as they are members of the object's group. Therefore an object's group comprises a leader and potentially one or more members and the server does not have to maintain state for all proxies that cache an object; rather, it only need maintain per-object state state for one proxy - the leader. The leader in turn maintains a list of *proxy ids* or *IP Addresses* representing its members and in addition manages the leases for objects it represents; i.e., in addition to forming a channel via which updates and invalidates may be propagated (as object change at the server) to proxies that cache an object it is the leader for (i.e., its members), it also maintains lease information and makes the decision of whether to *renew* a lease or not. When a lease for an object expires, it is the leader's responsibility to decide whether to renew the lease or not. A *lease renewal* involves a leader sending a request to the server asking for a fresh lease for an object.

Given the above introduction, we study the following issues that are important for scalable consistency solutions for CDNs, using extensive simulations and a prototype implementation of our algorithms.

- **Leader selection schemes:** The decision of which proxy becomes leader for an object is an important one, as we will see later in the report. We study two leader selection schemes, by which a proxy is picked as leader for a per-object group and compare how well these schemes balance load across proxies and the network overhead generated. Note that if the number of proxies increase by large numbers, a one-level proxy organization will not scale as leaders themselves may suffer from overload. In such cases, a multi-level hierarchy organization of proxies and/or the division of the proxies into more physical groups are possible solution that can scale (3).

- **Cooperative caching:** Such techniques when employed among a group of proxies are known to help in improving hit ratios and user response times but at the cost of increased network bandwidth usage. Cooperation between proxies generates network traffic as it necessitates communication between the proxies for locating an object requested. Several such techniques have been developed and studied in detail [43, 12, 17, 28, 34, 38, 49]. While this is orthogonal to the goal of maintaining consistency, the trade-off of potentially achieving better user response times versus higher network bandwidth usage has to be addressed.

- **Propagation of updates and/or invalidates:** When an object changes at the server, for server-based consistency schemes, it is necessary to propagate these changes either in the form of updates or invalidates to proxies holding cached versions of the object. While [25] measures gains involved in piggybacking invalidates along with reply messages in order to lower the number of IMS requests, and [30] investigates *delta-encoding* as a means of update propagation, only [13] proposes a technique for deciding the basis for propagating updates or invalidates. This decision is an important one as the propagation of updates is potentially more expensive that the propagation of invalidates in terms of network bandwidth consumption. [13] addresses ways to propagate either updates or invalidates based on the the number of requests for an object and its update frequency at the server. They propose a protocol where replica origin servers communicate the number of requests received to their parents and so on till the origin server gets the information, aggregates it, and decides whether to propagate updates or invalidates. While employing leases, we propose policies that are approximations of the above mentioned technique, and study how these affect network bandwidth consumption.

- **Eager vs. Lazy Lease Renewal:** The decision of whether to renew a lease for an object in an eager manner or in a lazy manner is also an issue that needs to be understood. Leases are said to be renewed in an *eager* manner, if leaders pro-actively decide on some basis, whether to renew a lease or not. On the other hand, leases are said to be renewed in a *lazy* manner if the leader does not renew a lease on expire. In such a situation, a lease for an object gets renewed only on the arrival of a request for the object. We study these techniques in terms of network overhead, server overhead and response times.

- **Scalability issues:** With the growth of a CDN and increase in popularity of objects, the state space overhead increases in terms of amount of state to be maintained in the system due to growing number of proxies, and the network overhead increases due to increased number of requests processed, increased number of control messages etc, to name a few. One way to reduce network bandwidth consumption is *not* to propagate all the updates at the server. Assuming users can tolerate occasional violations of consistency guarantees, we propose techniques by which the frequency of update or invalidate propagation adapts to varying network and server load. In addition, we also evaluate the scalability of leases in multi-level hierarchical organization of proxies and also with increasing number of groups of proxies.

The rest of the paper is organized as follows: Earlier in this section we introduced some of the related work in this area. Section 2 describes similar work in more detail. Section 3 elucidates our algorithms, techniques and policies. In Section 4, we present results of experiments we conducted using simulations and a prototype implementation. Finally in Section 5 we conclude.

2 Related Work

A lot of techniques (proxy-based, server-based and hybrids of these) for maintaining proxy cache consistency and deployable in individual proxies have been developed. Some of these include:

- **Periodic-polling [19]** : Proxies poll the server periodically and *pull* changes to objects it caches. Such mechanisms result in unnecessary consumption of network bandwidth for polling, especially when objects polled for are static or are changing infrequently.

- **Time-to-live (TTL) values [29]**: Servers associate time values with requested objects. Proxies can serve requests to such objects for a duration equal to the associated time value of the object. Once this time value lapses, a later requests necessitate the proxy to issue an IMS request to the server to check whether the object has changed and if so, get the new object before it responds to the client request. If TTL values are not carefully assigned, such a mechanism may result in a large number of IMS requests, if TTL values are small, and objects change infrequently.

- **Adaptive TTL [6]**: Here proxies provide servers with feedback about an object's popularity and thus enable the server to compute suitable TTL values for objects rather than assign fixed or arbitrary TTL values.

- **Time-to-refresh (TTR) techniques [39, 41]**: These investigate proxy-based techniques for maintaining both individual and mutual consistency for cached web objects. The idea here is to provide user-desired fidelity or consistency guarantees by making proxies track the rate of change of objects they cache, at the server and thus intelligently adapt their polling frequency to the rate at which cached objects change. This way, the mechanism not only provides consistency guarantees, but also optimizes network bandwidth consumption by polling only when required.

- **Server-based invalidation [25]**: The server takes the onus of ensuring consistency of objects cached at the proxies by maintaining state about such proxies and propagating invalidates to proxies as and when objects cached at the proxies change. Such mechanisms involve maintaining state at the server about which proxies cache what objects.

- **Adaptive Leases [10]**: The server employs the leases mechanism, which determines for how long it should propagate invalidates to the proxies. This work also presents policies using which appropriate lease durations are computed so as to balance the trade-offs of state space overhead and control message overhead.

- **Hybrid techniques - PoP, PaP [9]**: This work delves into the details of when proxies should *pull* data from servers and when servers should *push* data to proxies/clients and also thoroughly evaluates schemes.

It should be easy to see that these techniques cannot be deployed directly into large networks comprising several proxies, just because they do not scale. These techniques have been developed keeping individual proxies in mind, and not taking into consideration the existence of a large number of proxies that may need to communicate with a server. However, some of them, if *carefully extended* can serve the purpose.

Cache consistency in CDNs has not received as much attention as for individual proxies. Recently, [13] suggests a scheme for managing consistency in CDNs, and has been mentioned in the previous section. They addresses consistency issues between master origin servers and replica origin servers. Therefore, maintaining consistency between replica origin servers and intermediate proxies is not an area that has received any attention at all. While no other research has gone into this specific area, others have touched upon ways to maintain consistency in large-scale systems.

In [33], they suggest a distributed caching architecture that uses a central controller to keep track of popular objects, based on the assumption that web requests follow Zipf-like distributions and hence try to maintain freshness for such objects only. Client-based consistency solutions can possibly provide consistency guarantees that are close to server-based schemes, but are not as good, since all the information concerning the original copies of the objects cached at the proxies reside at the server. The suggested solution is a centralized one and is not likely to scale with the growth of such networks. Moreover, this mechanism only addresses popular objects and not all objects and hence does not provide strong consistency guarantees. [48] suggest a mechanism which employs leases and an application-level multicast scheme to propagate invalidates. However they assume multi-level hierarchical organization of caches, they employ a lazy lease renewal technique and suggest the use of *heartbeat messages* to communicate interest in objects between child and parent caches. In [46, 45, 47], the authors introduce *Volume Leases* as a consistency solution. This amortizes the lease renewal overhead over volumes of objects. Some other techniques they suggest include *delayed invalidation* as a scheme to reduce network overhead; note that these schemes do not provide *consistency guarantees*. In addition they note that lease renewals may be prefetched to keep clients and servers *synchronized* for longer periods of time either by pull or push-based mechanisms.

While the above mentioned techniques focus on the use of application-level multicast and grouping of objects into volumes to reduce lease renewal overhead, we study several other issues and propose novel scalable techniques to keep resource usage low and yet provide consistency guarantees.

3 Consistency Semantics, Algorithms and Policies

Before we proceed, we shall touch upon existing typical architectures of CDNs. See figure 1. As mentioned earlier, a CDN consists of the following:

- *Servers:* Master origin server and a few replica origin servers.
- *Proxies:* These sit in between clients and the servers.

We assume that *strong consistency* exists between these master and replica origin servers. For more information on this, one may read [13]. Given that the replica servers are consistent with the master origin server, in what follows, we address consistency semantics, algorithms and various policies that may be employed for deploying *leases* as an efficient mechanism for maintaining cache consistency between replica servers and a fixed group of proxies in a CDN.

3.1 Consistency Semantics

Consider a proxy in a CDN which caches frequently accessed objects from a replica server to improve user response times. Consider one such object ***a***. Assume all objects have associated version numbers. Also assume that these version numbers increase monotonically, i.e., every update to an object causes its version number to increase and a proxy cache never replaces an existing version of an object with an older version.

Now let P_t^a be the version of ***a*** at the proxy and S_t^a be the corresponding version at the server. We say that a consistency mechanism is *strongly consistent* when $\forall t, P_t^a = S_t^a$. If we take network delays into account, and assume a delay *d* in transferring a version of an object from the server to the proxy, then the above condition becomes $\forall t, P_t^a = S_{t-d}^a$.

While it is most desirable to provide strong consistency guarantees, some times meeting these guarantees may not be practical due to the very high overhead involved (which we will see later in Section 4). In such cases, assuming clients can tolerate occasional violations of consistency guarantees, we relax the above definition and say that systems need only provide a weaker notion of consistency we call Δ-*consistency*. i.e., a client may not see all updates to an object it is interested in, but is guaranteed to see an update once every Δ time units. More formally, we say that mechanisms that provide Δ-consistency guarantees should satisfy the condition that $\forall t, \exists \tau, 0 \leq \tau \leq \Delta$ such that $P_t^a = S_{t-d-\tau}^a$.

CDNs should employ mechanisms that satisfy the above conditions in order to provide strong or Δ-consistency guarantees thereby ensuring that the population of clients that they serve, do no receive *stale* data.

3.2 Deploying Leases in CDNs

We use *leases* [18] as a mechanism to achieve cache consistency in CDNs. Given a fixed physical group of proxies, which we henceforth refer to as a *cluster*, we now describe means by which leases may be deployed in CDNs. When a proxy receives a request for an object it does not have a valid lease on, it forwards the request to the server, which then grants a lease for the object and sends the object to the requesting proxy. A *lease* is a contract given by the server to the proxy, which says that, for the length of the lease, the server will propagate all changes to the object at the server, to the proxy. [10] presents different policies for intelligent lease duration estimation. At the end of the lease duration, if a proxy is still interested in the object, the lease gets renewed; in addition, proxies ensure that they have valid leases on objects they serve requests for. As a result, the system ensures strong consistency between the server and proxies. Now, since many proxies may request the same object, it is inefficient for the server to maintain leases on a per-proxy basis. Moreover, the overhead for maintaining multiple leases for an object and propagating updates for an object to multiple proxies is not a scalable solution, given that the number of proxies in a CDN may grow to several thousands. Hence an approach that guarantees scalability is to elect a leader proxy via which changes to an object may be propagated to other proxies that cache the object using an application-level multicast scheme. However, use of a fixed hierarchy with a single leader proxy, suffers from the same problems of high state space and control message overhead that the server suffers from. We therefore employ a dynamic grouping mechanism within a cluster. Proxies that cache an object, form a logical multicast group, with one such proxy assuming leadership of

the group. Hence, scalability achieved and in addition, the load gets shared among all proxies in the group. More simply said, there are as many leaders and multicast groups as there are object cached in the cluster.

With this introduction, we now present leader selection policies, consistency algorithms that employ co-operation and how they differ from those which do not employ co-operation, eager versus lazy renewal policies, and two policies based on which, updates and/or invalidates may be propagated. In Section 4, we present experimental evaluations of all these.

3.2.1 Leader Selection Policies

Here we present two policies by which leader proxies may be selected on a per-object basis:

- **First Proxy-based scheme:** Here, we employ a simple strategy. The first proxy in the group to receive a request for an object becomes leader for the object. The proxy forwards the request to the server; the server then marks this proxy as leader and replies to the request with the object requested and sends it a lease for the object. While such a scheme proves to be efficient for a single level hierarchy organization of proxies, we shall present a scheme to scale this technique to multi-level hierarchies later in this section. The experiments that compare the benefits of this scheme, with a hash-based scheme, for multi-level hierarchies are not included in this report. However, results which establish the fact that this method indeed scales will be presented.

- **Hash-based scheme:** Here, the first proxy to get a request for an object in the group hashes the URL of the object requested to a number. This number represents the *id* of a proxy, which is to be the leader proxy for the object. The request along with the leader proxy id gets forwarded to the server; the server notes the leader id for the object and sends the object to both the requesting proxy and the leader proxy (if different from the requesting proxy). The server also sends the lease for the object to the leader. This method may be extended to deal with multiple levels of hierarchy, if the hashing function is made to return a hierarchy of proxy id's given the object's URL and the branching factor desired; i.e., the execution of such a hashing function will determine which proxy communicates with which other proxy in the logical multilevel hierarchy, on behalf of requests for an object.

3.2.2 Co-operation between proxy caches

While co-operative caching techniques have been widely studied and are known to achieve better user-response times, they are also known to significantly increase the communication overhead between co-operating entities. We employ the following directory-based co-operative caching technique [27, 15, 40]. Each proxy in the cluster, maintains a directory of *object-leaderId* mappings. Assuming a first proxy-based scheme of leader selection, if a proxy is the first to receive a request for an object, it looks up its directory and sees that a leader does not exist for the object. It then issues a request to the server for the object. The server replies to this request with the object, a lease for the object and records the *id* of this proxy as leader for the object. This proxy which is now the leader for the object issues a broadcast message to the group claiming itself as leader for the object. All proxies record this information in their directories. From this point on, any other proxy in the cluster that receives a request for this object, need not forward the request to the server, but forwards the request to the existing leader for the object.

Since the lease on the object implies all changes to the object will be propagated to the leader proxy by the server, every other proxy in the cluster is guaranteed to get the latest version of the object from the leader proxy. This is considered cheaper than fetching the object from the server. Hence co-operation between proxies yields better user response time. In the case where there is no co-operation between caches, the differences are as follows: Proxies do not communicate with each other for the retrieval of an object, which may already be cached in a cluster. All requests for objects that are not locally cached, are forwarded to the server. However, logical multicast groups will be formed for update/invalidate propagation, to ensure scalability. Leaders still exist, but on becoming a leader, a proxy does not broadcast this information to the group; however, the server which elects the first proxy to request an object as leader, piggybacks this leader information on all replies to further requests for this object. This is done so that proxies know where to communicate their disinterest in the object. The server also sends a message to the leader asking it to add a distinct requesting proxy as member. This way, a channel is set up for efficient logical group maintenance for update/invalidate propagation, on a per-object basis.

3.2.3 Lease Renewal and Termination

In addition to maintaining a directory, each proxy in the cluster, which potentially becomes a leader for an object at some point, maintains a *membership list* dynamically. Such a list allows a proxy to identify which other proxies are members of a multicast group, for which it is the leader. This allows a leader proxy to multicast updates or invalidates for an object (using an application-level multicast) only to member proxies in the group (i.e., those proxies which cache this object). The membership list defines those proxies that are *interested* in the object. A proxy is said to be interested in an object if it has received a request for the object within some t time units where *t*, is a tunable parameter. If the proxy has not received a request for an object it caches within the last t time units, it issues a *terminate* request to its leader. This is true for the case of no co-operation also. The leader then deletes this member proxy from its group.

Here is a simple way to deploy this *lease termination* policy: We are primarily addressing consistency for static objects and not streaming objects; it is well-known that *LRU* is the most commonly used cache replacement policy for such objects. LRU may be implemented as follows: If an object receives a request, place it at the head of the list. For such an implementation, some fraction of the *tail* of the LRU list, is where candidates for termination exist i.e., this region of the LRU list has objects that have *aged*. Note that one may argue that the whole point about maintaining an LRU list is that future hits for objects are expected. However, [7] shows that Web traffic follows a Zipf-like distribution and that only 25% of the objects contribute to about 70% of the traffic and are therefore *hot*. Given this result, we expect the rest of the 75% of the objects to reside towards the tail of the LRU list. Therefore, every t time units, a proxy may scan the LRU tail for objects that have not received requests for over t time units and issue terminate requests for such objects. This as we see is very easy to implement in today's proxy caches which mostly employ the LRU cache replacement policy, the only additional requirement being the need for a timer that signals a scan of the LRU tail every t time units.

If a leader receives terminate requests from all its members, and it itself is not interested in the object, it terminates the lease and issues a message to all proxies in the cluster and to the server informing all that it is no more leader and that the lease on the object has not been renewed. If on the other hand, at the end of a lease duration, a leader

has at least one interested member, it issues a message to the server requesting *renewal* of the lease. Note that this approach is similar to *pruning* in multicast protocols. After lease termination, the next proxy in the cluster to receive a request for the object assumes leadership and the protocol executes as described.
While it may be expected that the message overhead indeed increases by employing a mechanism that provides consistency guarantees in CDNs, such an optimization should argue in favor of *leases* as a candidate mechanism for achieving strong consistency, in terms of pro-active reduction in the message and state space overhead by discarding state of objects that no one is interested in. Also note that this optimization is not compute-intensive and does not maintain any more state than is usually maintained in proxy caches today.

3.2.4 Lazy Lease Renewal

The renewal policy described above is an *eager* renewal policy, where if at least one interested member for an object exists, the leader for the object issues *renew* requests to the server. Another renewal policy is the *lazy* policy. Here, when a lease for an object expires, irrespective of the existence of interested proxies, the leader proxy does not renew the lease. The leader instead sends messages to all current members for an object informing them that the lease for the object has expired and that they should not serve further requests for the object locally. In addition, the leader discards its membership list. However, the next request for the object, at the existing leader or any other proxy in the cluster, triggers a renewal of the lease, and the protocol executes as described above.
As we shall see in the following section, the trade-offs of the above two policies are that while the control message overhead and the state space overhead are less for the lazy renewal policy, the hit ratio, response time and data transfer overhead are significantly worse. We do expect the performance of both eager and lazy renewal policies to converge with increase in the duration of the leases granted by the server.

3.2.5 Propagation of updates and invalidates

The cost of propagating updates or invalidates differ widely (see Section 4). Since network overhead is something we would like to keep low, the decision of when to propagate updates or invalidates becomes an interesting one. While this problem has received little attention, a recent study [13] addresses this issue to some extent. Some study has measured the gains in piggybacking of updates along with replies to IMS requests reduce the number of IMS messages [25] and another has studied *delta encoding* [30] to make update propagation less expensive. However, in order to compute and propagate 'deltas' of objects, a history of the changes to an object has to be maintained at the server; this is extra state required.

Although it is not common for server to propagate *updates* to proxies these days, the effects of co-operation are realized only when push-based servers are used for update propagation. Therefore our default algorithm requires a server to propagate updates to leaders, while leaders propagate *invalidates* to their members. On receipt of an invalidate, a member proxy evicts the object from its cache, forcing the next request for object to fetch the current version from the leader, if one exists, or from the server otherwise.

We now present the following policies that may be employed for update/invalidate propagation:

- **Popularity of an object:** Intuitively, the propagation of invalidates is good to help alleviate proxies of the work required to update objects they cache due to changes at the server. However, this is useful only when the popularity of an object is low. If an object is popular, and a server propagates only invalidates, then every other request for the object generates GET messages for the objects; this results in prohibitive network overhead for large systems. Hence, if an object is popular, it should be more efficient to propagate updates than invalidates. Now using our algorithm, we can estimate object popularity or object interest, by keeping track of the number of successive lease renewals granted for an object before a lease is terminated. A higher number of successive renewals indicates continuing interest in the object. Let the number of successive renewals be R_{succ}; then a server administrator could associate a threshold number of renewals for objects housed at the server (say R_{thresh}) and assert the following:

  if ($R_{succ} \leq R_{thresh}$)
    propagate INVALIDATEs
  else
    propagate UPDATEs

- **Object size:** Another simple policy that a server may employ is that if the size of an object, S_{obj}, is *small*, propagate the update, since the incremental cost over propagating an invalidate is small. Here again, choosing a threshold size (say S_{thresh}) is the administrator's job. Thus we have:

  if ($S_{obj} \leq S_{thresh}$)
    propagate UPDATE messages
  else
    propagate INVALIDATE messages

- **Hybrid Policy** We could combine the above two policies to work so as to propagate updates not only for popular objects, but if they are small too.

  if ($R_{succ} \leq R_{thresh}$)
    if ($S_{obj} \leq S_{thresh}$)
      propagate UPDATE messages
    else
      propagate INVALIDATE messages
  else
    propagate UPDATEs

Now given that we have a handful of possibilities, the question we try to answer is what threshold one should pick. Say we use an invalidate-only scheme. Let the number of requests that arrived at proxies in the cluster for objects that have been invalidated in their caches be n_{inv}. We suggest that, picking a threshold that allows the propagation of about n_{inv} updates is a good approximation of what is required. This is intuitive because if the number of updates far exceeds n_{inv}, then it is likely that the supply (number of updates propagated) exceeds the demand (number of requests), which is therefore wasted effort. These ideas are evaluated in Section 4.
Note that the above mentioned policies do not require recursive exchange of information in the system to finally accumulate at the server before a decision of whether to propagate updates or invalidates is taken as with techniques proposed in [13].

3.2.6 Adapting to Server and Network Load

It is impossible to predict server and network loads in the Internet today due to unpredictable user-access patterns and large variation in update patterns, depending on how dynamic objects are. If objects at a server become *hot*, (i) the network overhead increases due to increased requests for the object, replies, renewal requests etc (ii) the server overhead involved in maintaining consistency increases (state for leases, lease duration computation, leader proxy information etc). If these loads become *high*, the performance of such networks degrade. It is necessary to employ mechanisms to deal with such bursts of load.
Most applications today can tolerate occasional violations of consistency guarantees. Assuming this, we suggest that a way to deal with such bursts is to relax the strong consistency guarantees and not propagate all updates to an object. Instead, the server need only propagate one change every Δ time units, where Δ is either a user-input to the system or parameter that is computed based on bottleneck resources (the server or the network).
A simple way to implement the idea is to keep track of bottleneck resources for fixed time intervals and compute the frequency at which updates/invalidates should be propagated to the users for the next such time interval, depending on the load recorded in the previous interval. The server should propagate updates at least once every Δ time units in order to guarantee Δ-consistency or better. If the server propagates one update every t time units where $0 \leq t \leq \Delta$, then we say that the frequency $f = \frac{1}{t}$. Thus f lies in the range $[\frac{1}{\Delta}, \infty]$.

Let R_{thresh} denote a threshold that represents either the server load threshold or the network load threshold. Note that we use the number of leases maintained at the server to measure load of the server, and the number of messages received and sent by the server to measure network load. For the $(k-1)^{th}$ fixed interval of length t, let R_{k-1} represent either the server load or the network load.

We present two functions or policies that we use to compute t_k, the time between two update propagations and hence the frequency for the k^{th} interval, f_k.

- **Policy 1:**
  Allow t_k to vary directly with load as follows and guarantee consistency levels between Δ and strong consistency:

$$t_k = \begin{cases} 0 & \text{if } R_{k-1} \leq R_{thresh} \\ \Delta & \text{if } R_{k-1} \geq 2 * R_{thresh} \\ \frac{R_{k-1} - R_{thresh}}{R_{thresh}} * t & \text{otherwise} \end{cases}$$

- **Policy 2:**
  Use the following step function that guarantees strong consistency if the load is below threshold and Δ-consistency when the load goes above the threshold. Note that Δ may be a user input.

$$t_k = \begin{cases} 0 & \text{if } R_{k-1} \leq R_{thresh} \\ \Delta & \text{otherwise} \end{cases}$$

The second policy above, is relatively straight-forward. We therefore evaluate only Policy 1 in Section 4.

3.2.7 Scaling the Leases technique

As CDNs become more and more useful and popular and the demands of clients increase, it is logical to add more proxies to the network to deal with additional load. There are two ways in which this can be done. (i) Proxies may be added to each cluster and (ii) the number of clusters may be increased.

- **Multiple Clusters:** Here, while the consistency algorithm for a cluster as presented above remains the same, additional state has to be maintained at the server on a per-cluster basis. This includes maintaining of leaders and leases on a per-object, per-cluster basis. Therefore, if proxies in multiple clusters are interested in an object, the server propagates updates to multiple leaders and they in turn, propagate invalidates to their members. Also, the maintenance of leases on a per-cluster basis allows the algorithm to address the needs of each cluster independently and therefore adds flexibility to the system. For instance, a server administrator may choose to employ a specific lease duration computation technique [10], for a specific cluster, and/or monitor the load on each cluster independently etc.

- **Multilevel Hierarchies:** If the number of proxies in *a cluster* grow, then the logical one-level hierarchy will not scale, especially when IP multicast is not employed. This is so because the amount of work needed to disseminate data to and manage leases for growing number of proxies, increases. To deal with this scalability problem, we present a generic method of creating logical, $N - ary$, $L - level$ per-object hierarchies of proxies, where N may be an input to the system. This helps reduce the work leader proxies need to do in order to propagate invalidates to their member proxies. As we shall see in Section 4, the *first proxy-based* scheme of leader selection is more efficient in terms of network bandwidth usage than *hash-based* schemes. Moreover, the probability of making proxies which are not interested in an object part of its group is higher with hash-based schemes. In a multi-level hierarchical organization, if a hash-based scheme is used, while each proxy in a cluster will know which other proxy is its parent by execution of some function, it is not that easy in the first proxy-based scheme and additional mechanisms have to be employed. So while it was briefly mentioned, how the hash-based scheme may be extended to multiple levels earlier in this section, the extension of the first proxy-based scheme is not as straight forward. While the extension may be accomplished in many ways, we present one such mechanism that is efficient for our purposes. Some already existing protocols include [14],

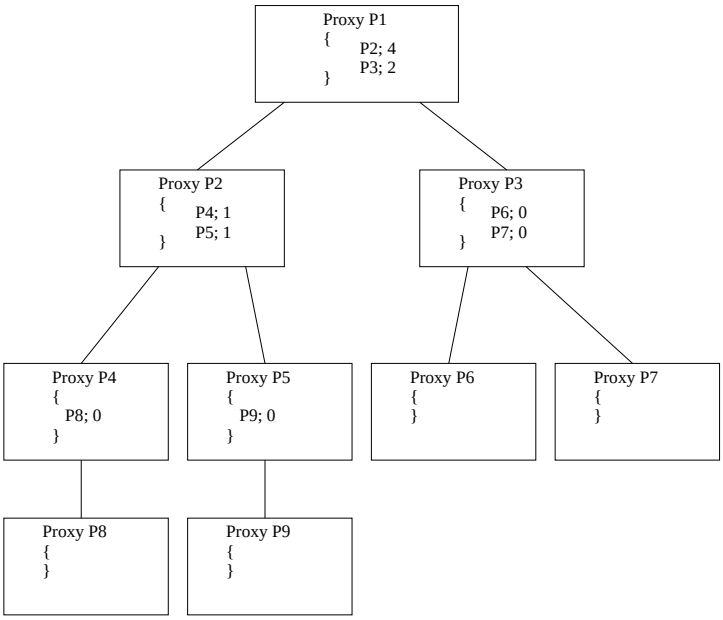


Figure 2: Multilevel hierarchy example with branching factor = 2

[26].

Here, as the name suggests, the first proxy in the cluster that receives a request for an object, becomes leader for the object. This information is broadcast to the cluster. Hence, other proxies that receive requests in the cluster forward the request to the leader. The leader makes such proxies its immediate children as long as the number of such proxies is $\leq N$. To continue with the description of the protocol, we require that a proxy keeps track of the number of children proxies in the subtree rooted at each of its members. If a proxy which does not belong to the hierarchy requests the same object, the following protocol is employed: The leader sends the object to the requesting proxy and forwards the request to any one of its member proxies that has the least number of children; if all members have an equal number of children $< N$, then a member is picked at random. If all the members have N children each, and the leader proxy has N children, it forwards the request (called *Add-Member* request in the Section 4) to one of its children, picked at random. The process recursively continues until at some level of the hierarchy, a proxy with the lowest number of children proxies is located. Such a proxy makes the requesting proxy its child and the new parent sends a message (called *Join-Me* requests in Section 4) to the requesting proxy informing it of its action. The requesting proxy makes such a proxy its parent and thus joins the hierarchy. See figure 2 for an example. The boxes represent proxies in the hierarchy and the fields correspond to *childIds* and the number of children in the subtree rooted at *childId* respectively. Such a scheme as we see, only involves those proxies that are *interested* in an object, unlike the hash-based scheme. However, with time, if members in such hierarchies lose interest in an object, this in effect reduces to a case where proxies in the hierarchy become simple *forwarding proxies*; i.e., although they may not serve requests for an object any more because they do not receive any requests for the object, they

still forward updates/invalidates to children proxies that are interested in the object. Note that when a proxy loses interest in an object, it sends a *terminate* request to its parent. The parent in turn should intimate its parent (this goes all the way to the root) of the same so that state at the leader concerning number of children in all subtrees get updated. These messages are called *UpdateChildrenState* messages in Section 4. Note that this ”join and leave” algorithm involves no multicast or broadcast and is therefore efficient in terms of its use of available network bandwidth. Moreover, since that subtree with the least number of children (or nodes) is eventually picked, the tree remains fairly balanced at all times.

4 Experimental Evaluation

In this section, we demonstrate the efficacy of leases using trace-driven simulations and a prototype implementation. In what follows, we present our experimental methodology and results.

4.1 Simulation Environment

We designed an event-based simulator to evaluate the efficacy of the leases' technique in maintaining cache consistency between a server and many proxies. The simulator simulates one/more clusters of proxy caches (any number) that receive and service requests from several clients. Cache hits are serviced using locally cached data whereas cache misses are simulated by fetching the object either from the server or from another proxy in the cluster (the leader), in case co-operative caching techniques are employed. The proxies are assumed to employ the leases' mechanism for maintaining consistency of cached objects with their versions at the server.
For our experiments, we assume that a proxy employs a disk-based cache to store objects. We assume infinite cache sizes. Data retrievals from disk (cache hits) are modeled using an empirically derived disk model [] with a fixed OS overhead added to each request. We choose the Seagate Barracuda 4LP disk for parameterizing the disk model []. For cache misses, data retrieval time over the network is modeled using the round-trip latency, the network bandwidth and the object size. Since proxies are usually deployed closer to clients, but distant from servers, we choose 3ms and 2MB/s as client-proxy latency and bandwidth; the proxy-proxy latency and bandwidth are chosen to be 75ms and 500KB/s respectively and the proxy-server latency and bandwidth are chosen to be 250ms and 250KB/s respectively (these parameters assume a LAN environment). In reality these parameters vary depending on network conditions, distance from source to destination etc. Since we are more interested in consistency issues, the use of a simple network model suffices.

4.2 Workload characteristics

To generate the workload for our experiments, we use traces from actual proxies, each servicing several thousands of clients over a period of several days. We employ two traces for our experiments (NLANR and DEC traces). The characteristics of these traces are as follows:

- NLANR Trace:
  - Trace Duration: About 15.5 hours

  - Number of Read Requests: 750000
  - Number of Synthetic Writes: 14385

- DEC Trace:
  - Trace Duration: About 11.1 hours
  - Number of Read Requests: 750000
  - Number of Synthetic Writes: 17126

Note: For few of the experiments, we needed the number of proxies used to grow to 15 or even 20. Due to memory constraints, we have not been able to simulate 750000 requests for these cases. In most of such cases we managed 500000 requests (for NLANR: it equals 10.5 hours, for DEC it equals 6.5 hours) and in one such case we used only 490000 requests (only for NLANR: it equals 10 hours).

Each request in the trace provides information such as the time of the request, the requested URL, the size of the object, the client making the request etc. Determining when objects are modified is crucial to cache consistency mechanisms. We employ an empirically derived model to generate synthetic write requests (and hence last-modified times) for our traces. Based on observations in [], we assume 90% of all web objects change very infrequently (i.e., have an average life time of 60 days). We assume that 7% of all objects are mutable (i.e., have an average life time of 20 days) and the remaining 3% of objects are very mutable (i.e., have an average life time of 5 days). We partition all objects in the traces into these 2 categories and generate write requests and last modified times assuming exponentially distributed life times. The number of synthetic writes generated for both traces is as shown above.

If corresponding NLANR and DEC results are compared, subtle differences may exist due to (i) difference in read/write ratios and (ii) difference in the client request distribution across proxies in a cluster.

4.3 Experiments with NLANR Trace

4.3.1 Constants

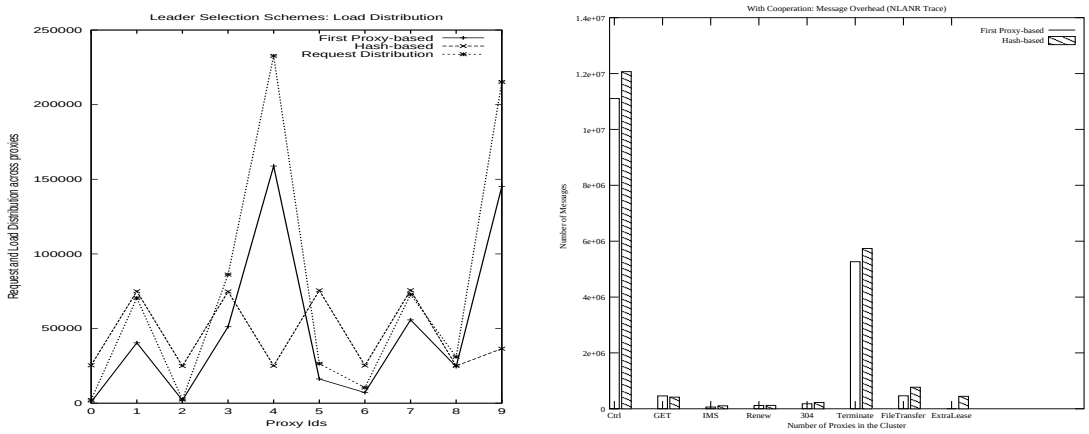
Unless explicitly mentioned, all our experiments have the following:

- **Trace used:** About 15 hours of NLANR Trace (750000 requests).
- **Number of proxies:** 10 Proxies.
- **Cooperation:** Proxies in a cluster co-operate.
- **Lease Duration:** We employ a fixed lease duration of 30 mins for all leases.
- **Renewal Policy:** We employ an eager renewal policy for renewal of leases.
- **Lease Termination Policy:** If a proxy does not receive a request for an object for 30 mins, it concludes it is not interested in the object.
- **Message Counting:** We count a multicast message as n messages.

- **Cache Hit:** Ability of the cluster to serve a request without contacting an external entity.
- **Client-Proxy mapping:** is achieved by use of a simple hashing function (*clientId % NumberOfProxies*).
- **GET Message:** is issued if an object is not locally available OR if an object has been invalidated (counted as 1 message).
- **IMS Message:** is issued only if an object exists in a proxy cache, but there exists no authority within the cluster to serve it. i.e., its lease has expired (counted as 1 message).
- **Renew Messages:** serve as IMS messages also, but are counted separately. This is done to address a real scenario, where an object changes at the server between the time the renew request is issued and the time the request reaches the server. In such a case, the reply from the server has the following (*newLease, newObjectVersion*).
- **304 Messages:** are replies to IMS/Renew requests if the object as the server has not changed (counted as 1 message).
- **Terminate Messages:** are issued from member proxies to leader proxies if a proxy has not received a request for 30mins (counted as 1 message). We choose this duration as 30 mins since we are employing fixed leases for our experiments. Terminate messages are issued from leader proxies to server and all proxies in the cluster if it has no members AND it has not received requests for an object for 30mins. (Counted as $1 + n$ messages: 1 unicast to server, 1 broadcast to other proxies in the cluster).
- **Update/Invalidate Propagation Policy:** *Push-based* mechanisms are not commonly deployed in practice. However, the benefits of cache cooperation are better realized when updates are propagated. We therefore assume that servers propagate *updates* to leader proxies, and these in turn propagate *invalidates* to member proxies. Hence while objects do get replicated across proxies, as more and more proxies get receive requests for an object, if the object changes at the server, there will exist exactly one copy of the new version of the object in the cluster, while all other copies get invalidated. These invalidated copies at member proxies get replaced with the new version of the object, on request.

4.3.2 Leader Selection

We compare the first proxy-based and hash-based leader selection schemes in terms of load balancing across proxies and number of messages generated. While the hash-based scheme achieves better distribution of load across the proxies, for a 10-proxy case, the first proxy-based scheme's load balancing is governed by the request distribution across proxies (see Graph 3(a)). Graph 3(a) plots the number of requests each proxy received and also plots the number of times each proxy in the cluster assumed leadership using both schemes. If the requesting proxy has the same id as returned by the hashing function always, the hash-based scheme becomes as efficient as the first proxy-based scheme in terms of network bandwidth consumption. If this is not the case, the number of messages generated is far more (see graph 3(b) - upto a 10% increase). For instance, there will be at least one more terminate message



(a) Load Distribution (b) Message Overhead

Figure 3: Comparing Leader Selection Schemes in terms of load balancing and message overhead

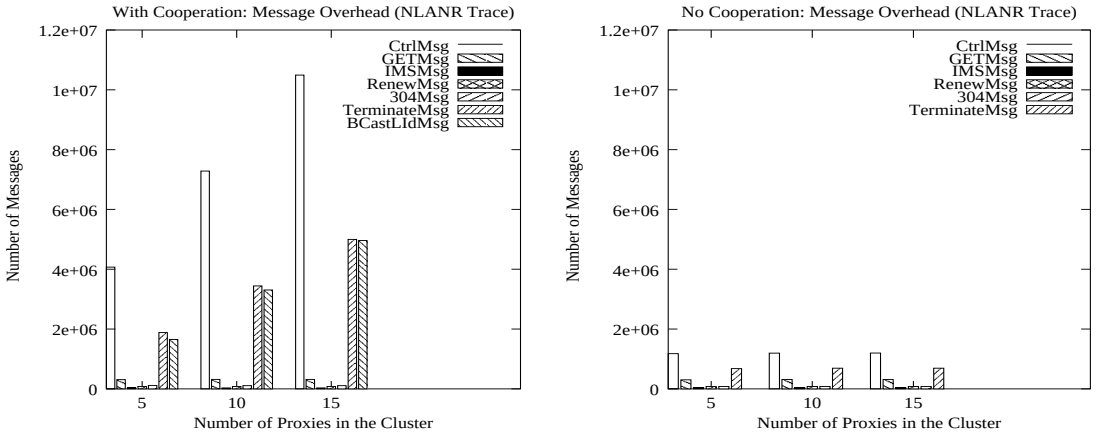
issued from member to leader than in the first proxy-based case. Since the server has to send the lease to the leader, if the leader is not the same as the requesting member, then an extra message is incurred for sending the lease to the leader - called *ExtraLease* messages. Also, since object updates are sent to the leader proxy first, and then later sent to the member proxies on demand, the number of messages required to transfer data are higher in the hash-based scheme (see FileTransfer messages). However, assuming that on several occasions this is the case, what we should note is that replication of the object happens at a rate faster than in the first proxy-based case, and hence there are fewer GET messages, and more IMS messages.

Compare these results with the corresponding DEC result in figure 33. The request distribution is not as skewed as it is in NLANR, hence while the hash-based scheme achieves close-to-uniform load balancing the first proxy-based scheme is not far off. In any case, the first proxy-based scheme consumes less network bandwidth.

4.3.3 Cooperative Caching

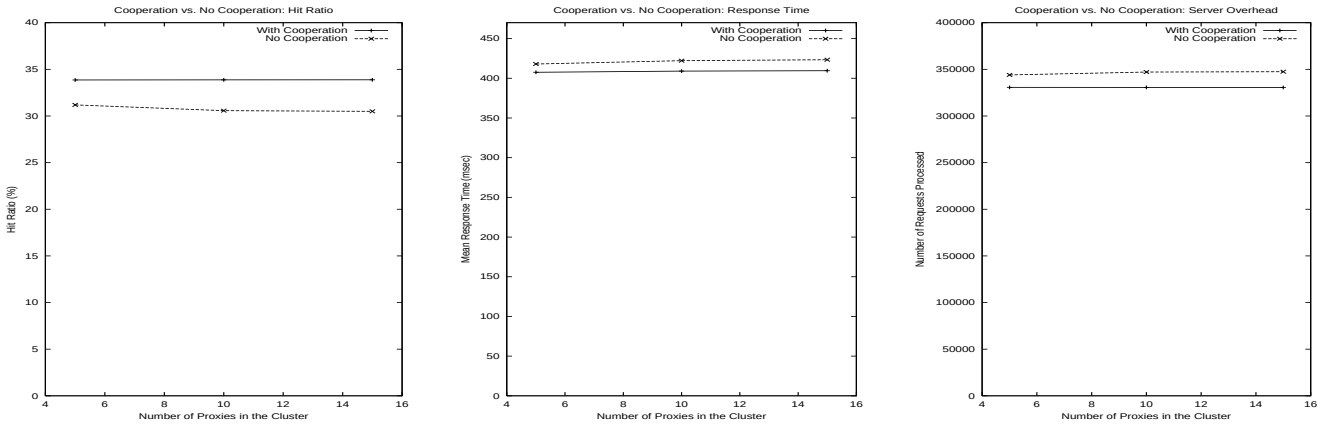
In this experiment, we vary the number of proxies from 5 to 15 and compare results thus got. While there is only a slight increase (about 2%) in the total control message overhead when there is no co-operation employed between proxies, as we go from 5 to 15 proxies in a cluster, the price paid for co-operation is the number of broadcast messages made by a proxy when it becomes leader for an object and when a lease is terminated. This causes the control message overhead to increase by 160% (Graphs 4(a) and (b)). Such a higher number is due to the very high read/write ratio of the NLANR trace used. The positive side of this however, is that there is an improvement of up to 12% in hit ratios (Graph 5(a)) and about 5% in the mean response time (Graph 5(b)). Another clear advantage is that the server performs less work (in terms of the number of requests processed) when co-operation is employed. This is because if an object exists in the cluster and has a valid lease on it, then it is always served from within the cluster, thus taking load off the server. From the graph 5(c), we see that the server overhead reduces by up to about 5%.

From the above results (figures 4 and 5), also note how the metrics vary with increasing number of proxies in the cluster. With cooperation, the hit ratio, mean response time and server overhead remain pretty much the same, as



(a) With Cooperation (b) No Cooperation

Figure 4: Message Overhead comparison for cooperative caching



(a) Hit Ratio (b) Response Time (c) Server Overhead

Figure 5: Effects of cooperative caching on hit ratio, response time and server overhead

we distribute the same workload across increasing number of proxies. This result however, does not hold for the no-cooperation case. While the control message overhead increases with increasing number of proxies, the mean response time and server overhead increase. We see a visible decrease in hit ratio because the number of first-time misses increase with increasing number of proxies.

For more dramatic differences, please see the DEC results in figures 34 and 35.

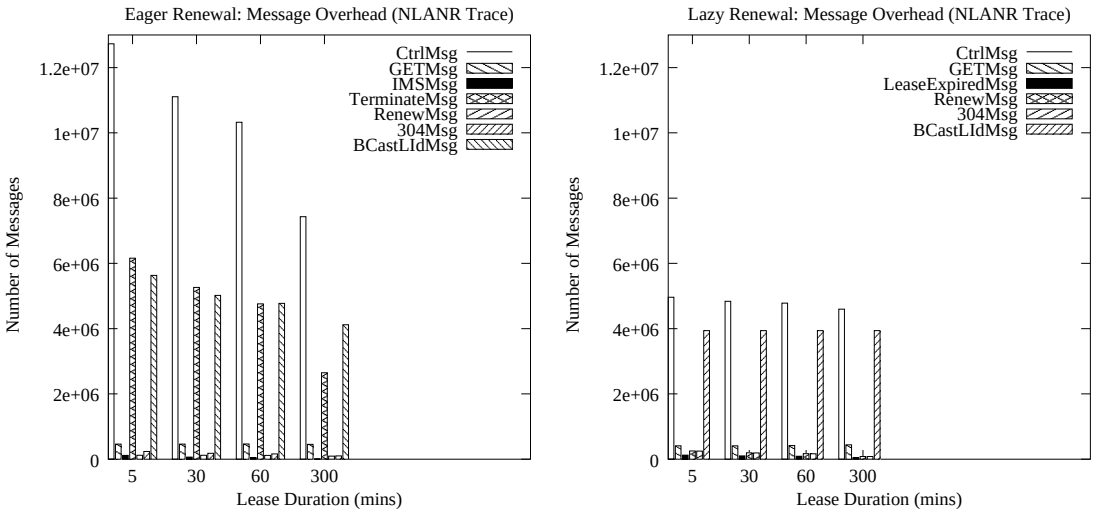
4.3.4 Eager vs. Lazy Renewals

In eager lease renewal policies, while leases are renewed pro-actively based on criteria already stated, in the lazy case, the leases are left to expire, all membership state is discarded and the algorithm is made to re-execute on receipt of the next request. So what we do expect is that at the cost of higher message overhead, data availability is more and hence user response times are lower. This is exactly what we observe in graphs (see figures 6 and 7).

For our experiments we kept most variables fixed and changed on the lease durations of leases granted from 5 mins to 300 mins. The results we observe are similar for short to long lease durations. The number of renewal messages, terminate messages etc decrease will longer leases. However, note how the results tend to converge from shorter to longer leases (see figures 6, 7, 8, and 9). For better convergence, see the DEC results in figures 36 to 39.

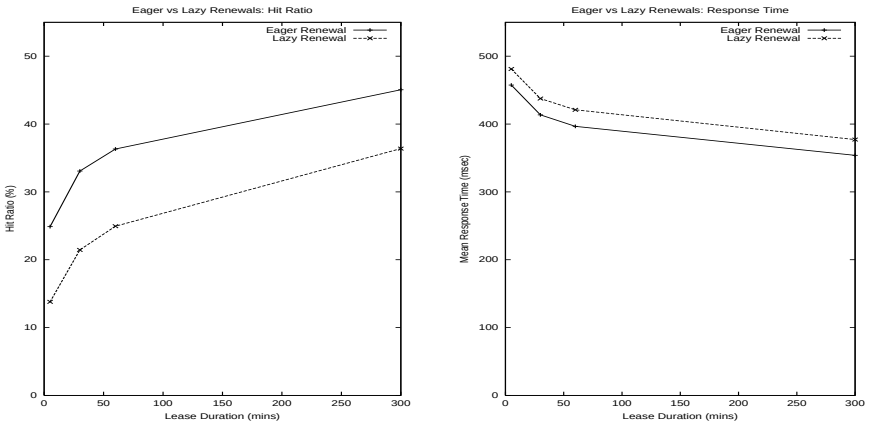
Looking more closely at the graphs we see that while there is a 60% to 150% increase in the number of control messages for eager renewals over lazy renewals depending on lease durations(see figure 6), and up to 16% more state space overhead due to lease maintenance(see figure 8(b)), the pluses of eager renewals include 25% to 80% improvement in hit ratio and up to 7% improvement in mean response time (see figure 7). For shorter leases, the number of file transfer messages is lower because updates get propagated only for those objects that have valid leases. Moreover, eager renewals cause a higher number of file transfers as the chances that they are longer lived than when using the lazy policy are higher (see figure 8(a)). Figure 9 reflect what we expect in that the number of updates propagated are more in the eager renewal case as leases are ”alive” for longer durations of time. However, for the number of invalidates propagated from leaders to members, the numbers are higher for the lazy renewal policy for shorter leases. This is so because, the leader is fixed and hence chances that such messages are generated for other proxies are higher. In the eager case, over long durations of time, several proxies may become leaders and hence chances of members always existing are fewer. Hence the fewer number of invalidates. For longer leases, since leaders remain put for longer durations, with more updates propagated, we see more invalidates.

We also study the effects of varying our termination policy. For the above results we assumed that proxies would issue terminate requests to their leaders if they do not receive requests for 30mins (equivalent to a lease duration). Graphs in figures 10, 11 and 12 show us results of experiments we conducted by varying this duration from 30mins up to 120mins. As we expect, the number of terminate messages and leaderId broadcast messages decrease while number of renew messages increase as proxies retain their group membership for longer periods. If group memberships are retained for longer periods, it implies longer durations for which leases remain valid. Hence the number of updates and invalidates propagated also increase (see figure 11). Also from figure 12 we see that the hit ratio increases by about 18% and the mean response time improves by about 26%. This is because more updates get propagated. However, note that for the above mentioned benefits, the control message overhead and the state space overhead at the server (in terms of number of leases maintained) increase by up to about 9% and 209% respectively (see figure 10).



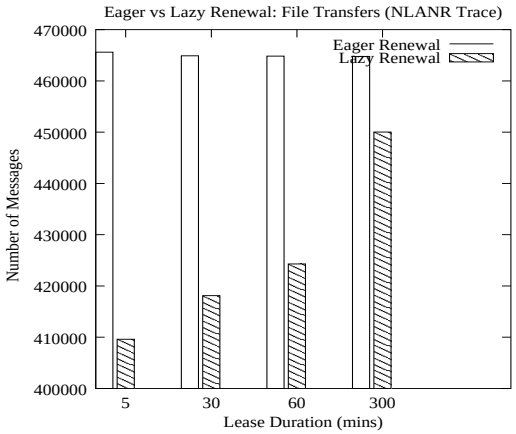
(a) Eager Renewal (b) Lazy Renewal

Figure 6: Message overhead comparison for eager and lazy renewal policies.

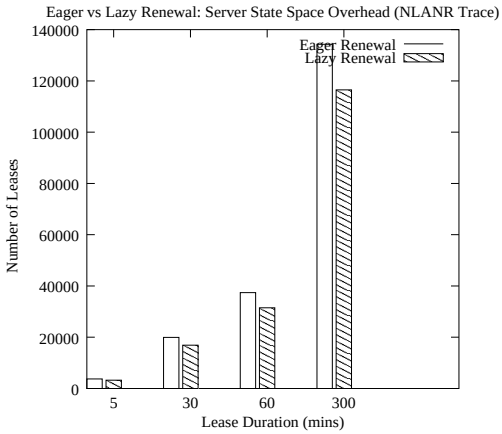


(a) Hit Ratio (b) Response Time

Figure 7: Hit Ratio and Response Time comparison for eager and lazy renewal policies.

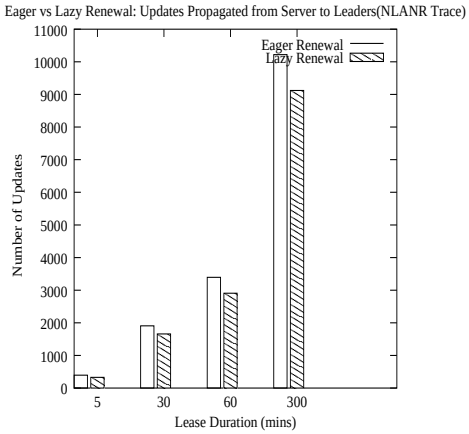


(a) Number of File Transfers

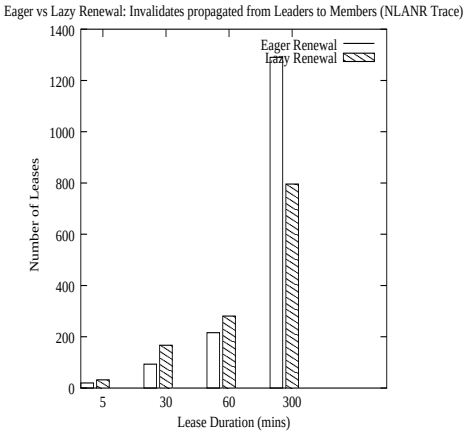


(b) Server State Space Overhead

Figure 8: File Transfers and State Space Overhead comparison for eager and lazy renewal policies.

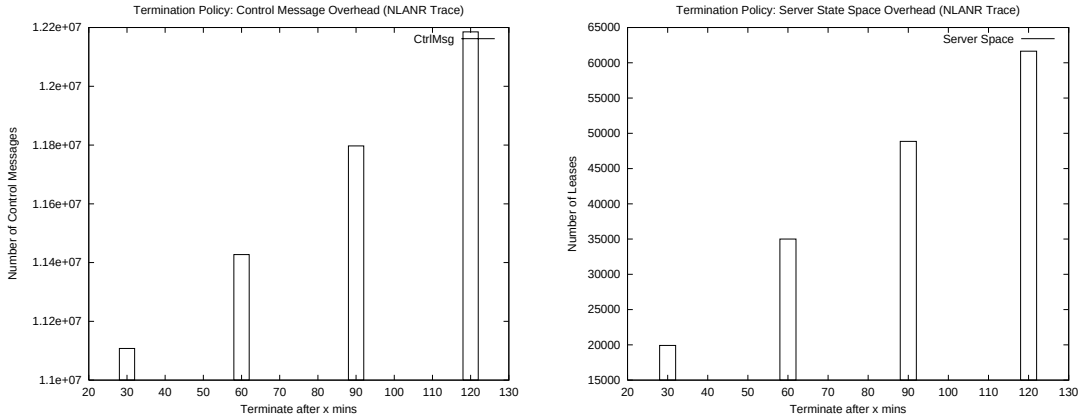


(a) Number of Updates



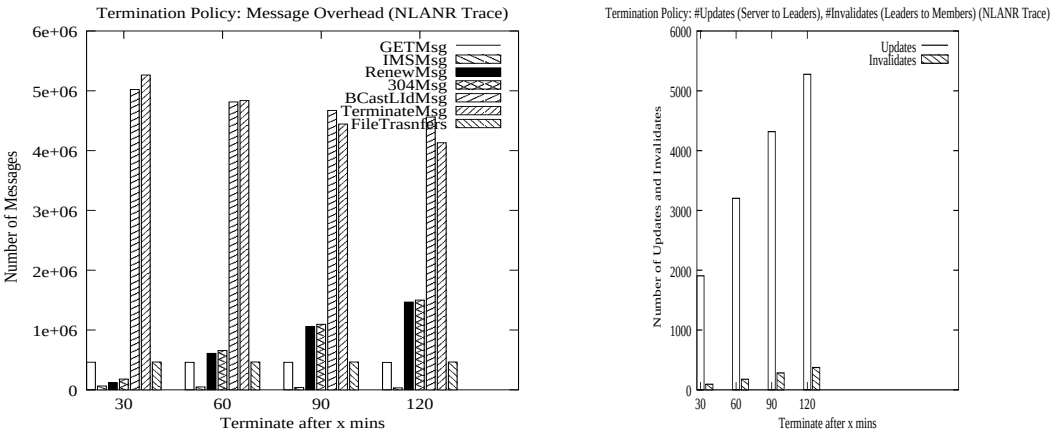
(b) Number of Invalidates

Figure 9: Updates (from server to leaders) and Invalidates (from leaders to members) comparison for eager and lazy renewal policies.



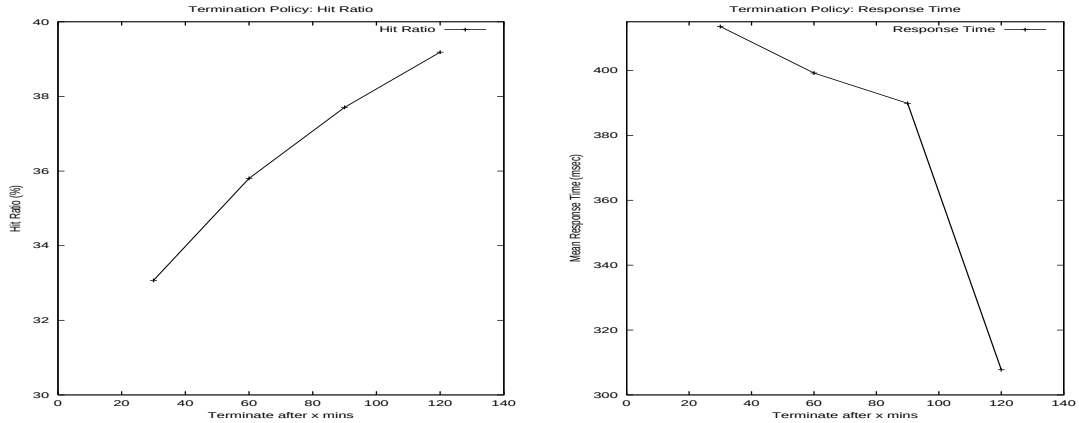
(a) Control Messages (b) Server State Space Overhead

Figure 10: The Control Message and Server State Space Overheads



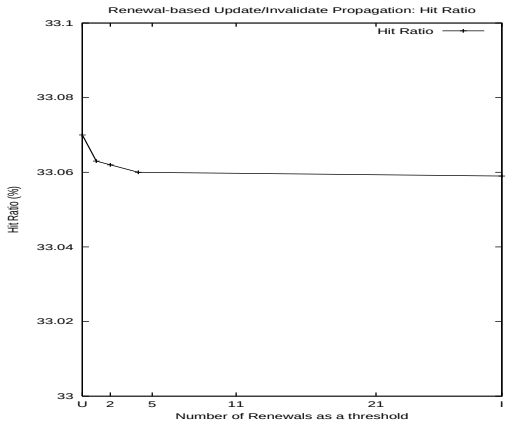
(a) Message Types (b) Updates and Invalidates

Figure 11: Message Overhead Types and Number of updates and invalidates propagated

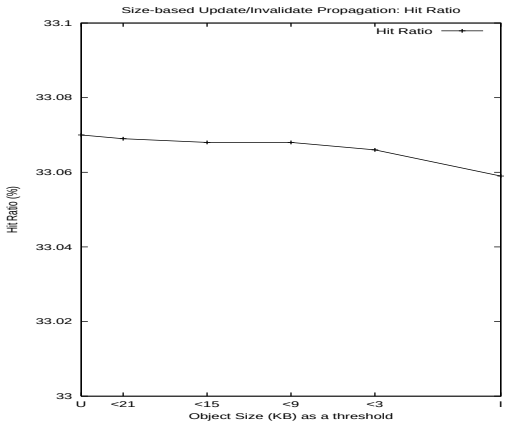


(a) Hit Ratio (b) Response Time

Figure 12: The Hit Ratio and Response time with varying Termination durations

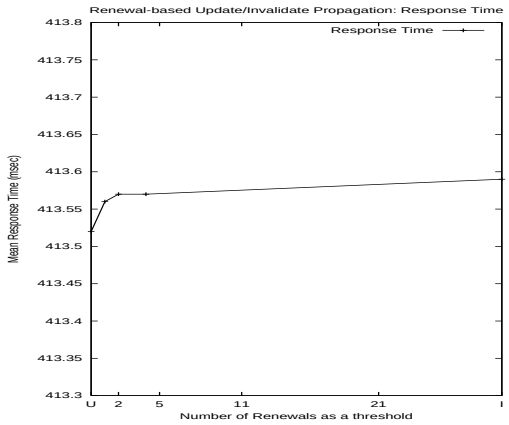


(a) Renewal-based: Hit Ratio

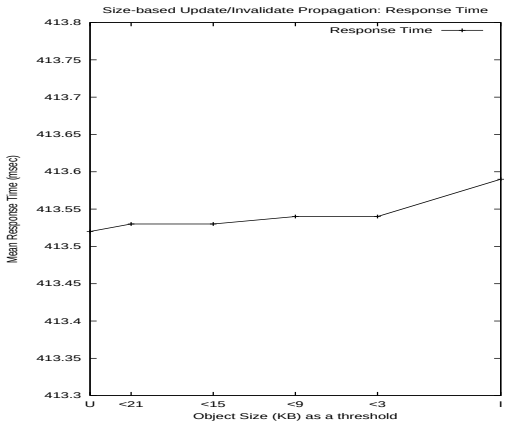


(b) Object Size-based: Hit Ratio

Figure 13: Hit Ratio with varying thresholds



(a) Renewal-based: Response Time



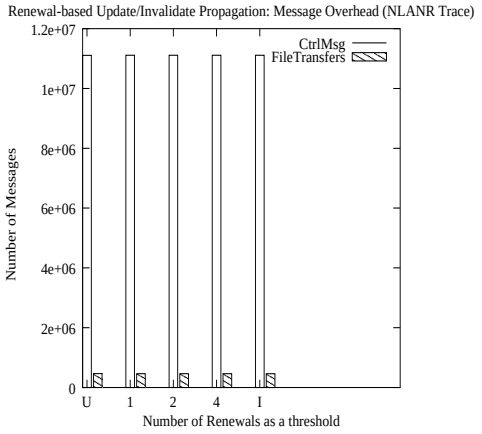
(b) Object Size-based: Response Time

Figure 14: Response Time with varying thresholds

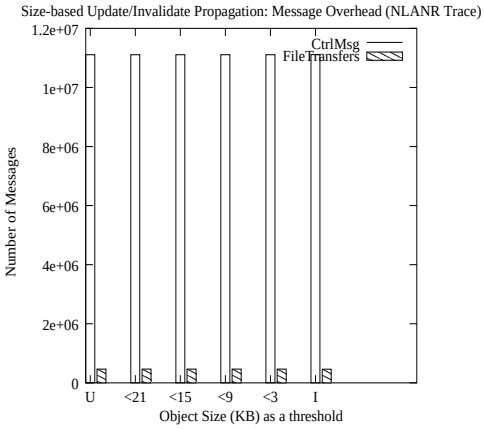
4.3.5 Update vs. Invalidate Propagation

Here we present the evaluation of the schemes we proposed earlier for update/invalidate propagation. What we expect is that irrespective of the scheme used, the results achieved by varying the thresholds that determine whether to propagate an update or not will pronounce ways to bridge the gaps between the overheads of update-only and invalidate-only propagation schemes. The graphs 13(a) and (b) show the variation of hit ratio as we move from updates-only to invalidates-only schemes. As is expected, if fewer and fewer updates get propagated, the hit ratio is bound to decrease. Similarly the mean response time (see graphs 14(a) and (b)) is bound to increase when more invalidates are propagated as they cause further requests for the object to ”get” the object from leader proxies. This changes are small, which is a direct cause of the high read/write ratio of the trace used.

Graphs 15(a) and (b) show how the total number of control messages and file transfers vary. While there is an increase in the number of control messages, the number of file transfer messages decrease. This is so, because the more the number of invalidates propagated, the more a scheme tends towards to ”serve-request-on-demand” scheme,

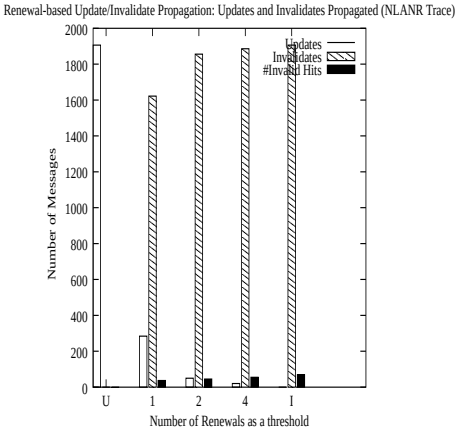


(a) Renewal-based: Message Overhead

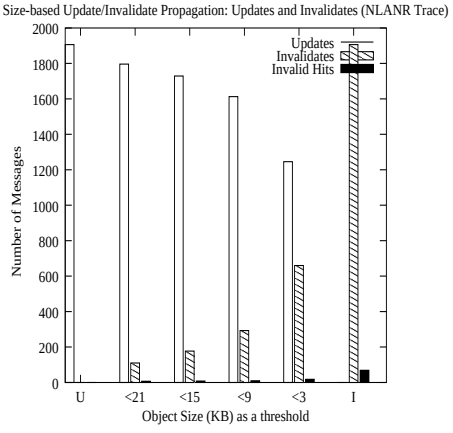


(b) Object Size-based: Message Overhead

Figure 15: Message overhead with varying thresholds

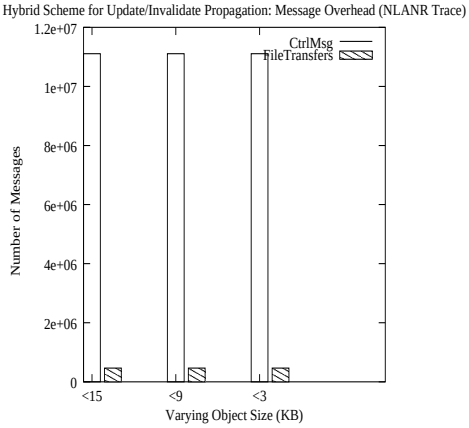


(a) Renewal-based: Updates and Invalidates

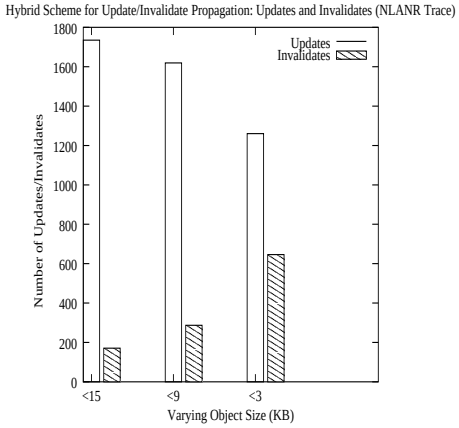


(b) Object Size-based: Updates and Invalidates

Figure 16: Number of updates and invalidates propagated from Server to Leaders, with varying thresholds

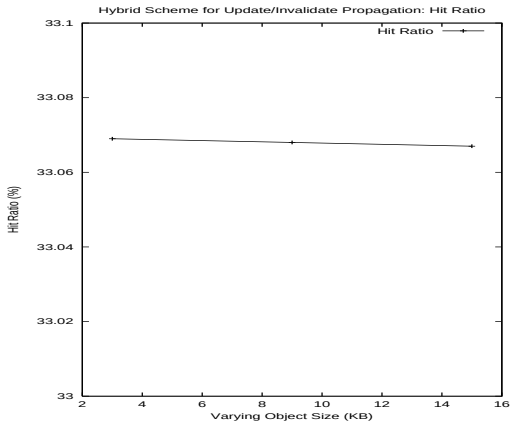


(a) Hybrid scheme: Message Overhead

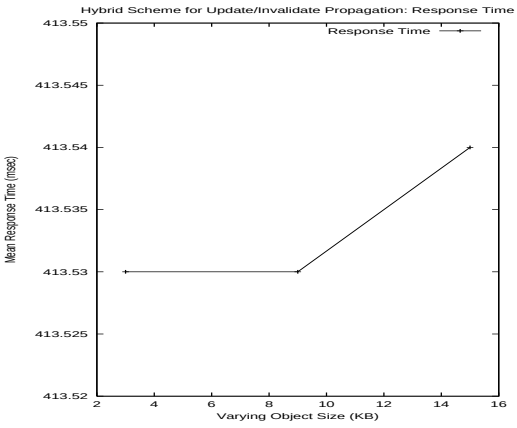


(b) Hybrid scheme: Updates and Invalidates

Figure 17: Message overhead and updates/invalidates propagated from Server to Leaders, with varying thresholds using the hybrid scheme



(a) Hybrid scheme: Hit Ratio



(b) Hybrid scheme: Response Time

Figure 18: Hit Ratio and Response Time, with varying thresholds using the hybrid scheme

and hence the number of data-carrying-messages decrease. On the other hand, as mentioned above, requests for invalidated objects generate more and more GET requests; this causes an overall increase in the number of control messages. Graphs 16(a) and (b) show how the total number of update and invalidate messages vary. As is expected, with increase in the threshold, the number of update messages decrease and the number of invalidate messages increase. These graphs plot the number of requests that arrive for objects that are invalidated also (69 such requests for the invalidate-only case). As we can see, our idea of choosing a threshold based on this number for the invalidates-ony case, is a good predictor for what threshold one should use, when employing the renewal-based scheme (in the result shown, a renewal-based threshold of 2 suffices to bring the number of such requests down to 45). This does not work as well with the size-based scheme. In the hybrid scheme, we fix the the number of successive renewals that indicate popularity to 2 and vary the size threshold. We see from graph 17(b) that such a scheme allows one to propagate upto 1700 more updates if the size thresold is set to $< 15KB$; this scheme is based on the assumption that it is not expensive to propagate "small" updates. Graphs in figure 18 show how the hit ratio and mean response time vary for the hybrid scheme.

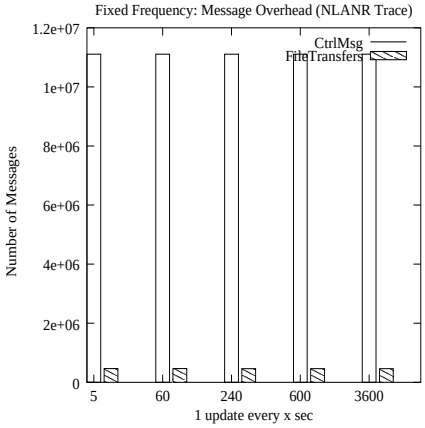
4.3.6 The *Frequency* Parameter

In the previous section we introduced the notion of "frequency" as a means to deal with increasing server or network load. The idea is to avoid the propagation of all the changes to objects happening at the server, in order to conserve network bandwidth, assuming that applications accessing these objects from the proxies can tolerate occasional violations or *stale hits*.

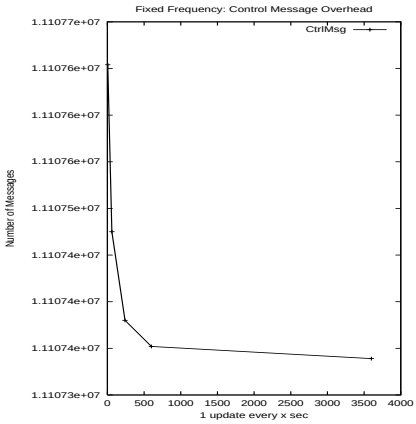
Frequency can be incorporated in two ways - we can either accept it as an input to the system, and provide user-desired consistency or we could dynamically track resources (server and network overhead) and decide the frequency of update propagation from the server to the leaders. We first present results by assuming that the time interval between two updates propagated is fixed and later present results by computing frequency dynamically based on server overhead (in terms of the number of active leases) and network overhead (in terms of number of control messages).

In the *fixed frequency* experiments, we vary the period between the propagation of any two updates from values as small as 5 seconds to values as large as 1 hour. If every update is not propagated from the server to the leaders, we expect the following: (i) violations of strong consistency, though we are meeting Δ-consistency guarantees, Δ being the inter-update propagation period. (ii) As the frequency decreases, the control message overhead will reduce due to decreased number of updates propagated. (iii) The number of number of times requests arrive for objects that are invalidated will be fewer, with fewer invalidates are propagated from leaders to members. Hence the hit ratio increases with decreasing frequency.

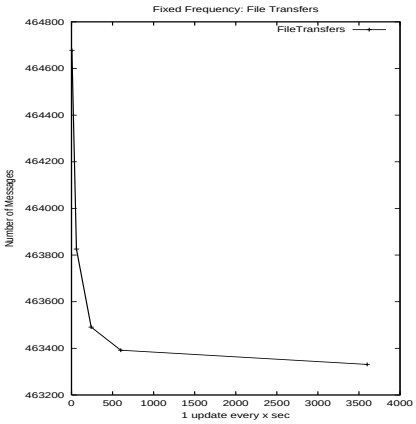
Given the above intuitions, we now examine results in figures 19, 20 and 21. Figure 19(a) shows how the number of control messages and file transfer messages vary with decreasing frequency. While the decrease in network bandwidth consumption is not clear from this graph, figures 19(b) and (c) accenuate the decrease. The control message overhead is seen to decrease by about 300 messages and the number of file transfers decreases by about 1200. The decrease does not seem to be significant because the update frequency of objects at the server is not high; about 2000 updates are propagated if the frequency parameter is not introduced - therefore the total decrease in the



(a) Control Messages and File Transfers

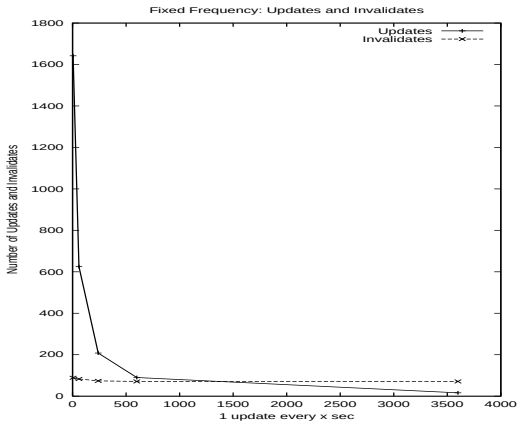


(b) Control Messages

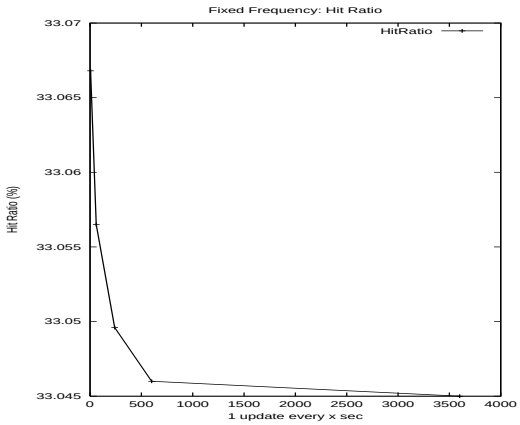


(c) File Transfers

Figure 19: Control Message Overhead and File Transfers with varying frequency

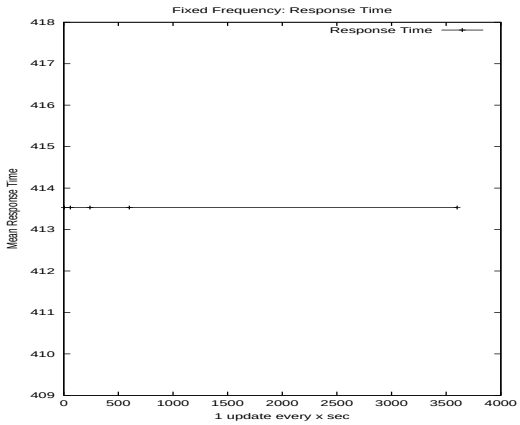


(a) Updates and Invalidates

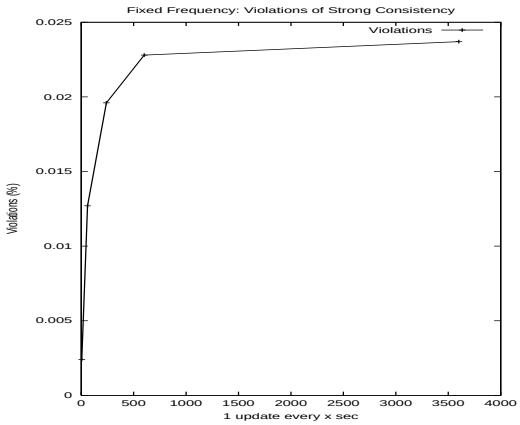


(b) Hit Ratio

Figure 20: Number of updates/invalidates and hit ratio with varying frequency

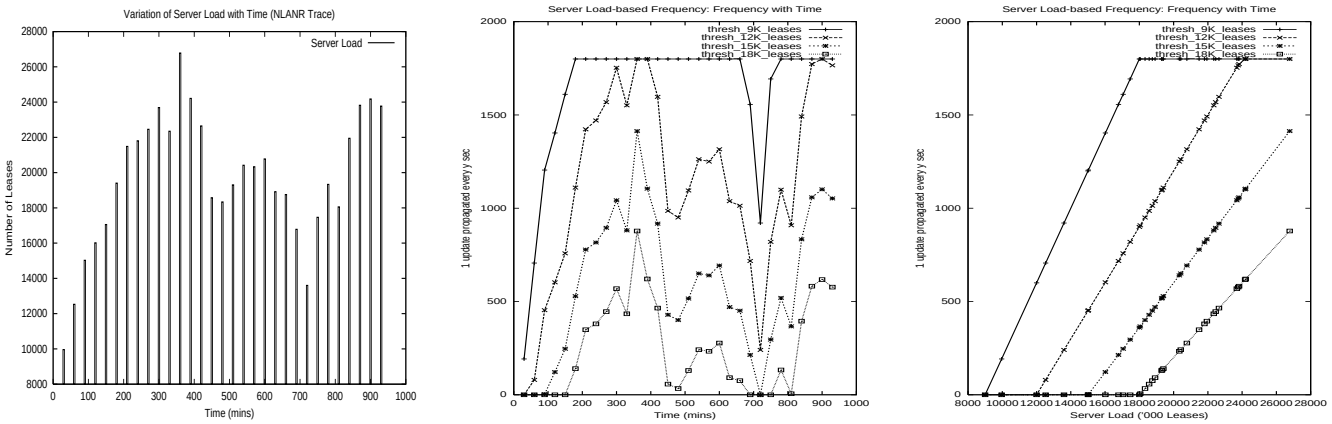


(a) Response Time



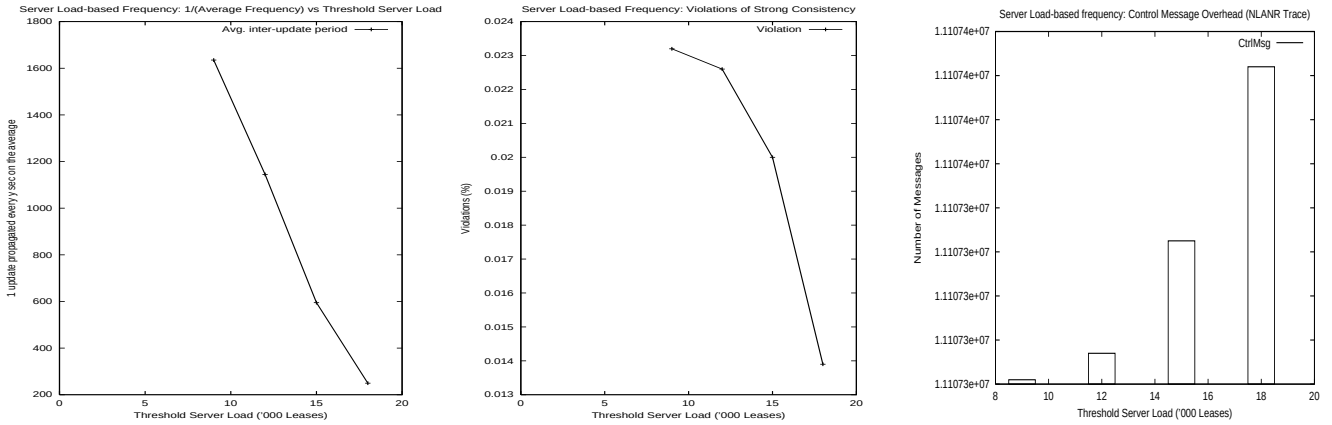
(b) Violations

Figure 21: Mean Response Time and Percentage of Violations varying with frequency



(a) Server Load (b) Tracking Frequency (c) Frequency vs. Load

Figure 22: Variation of server load and frequency with time; also frequency vs. server load

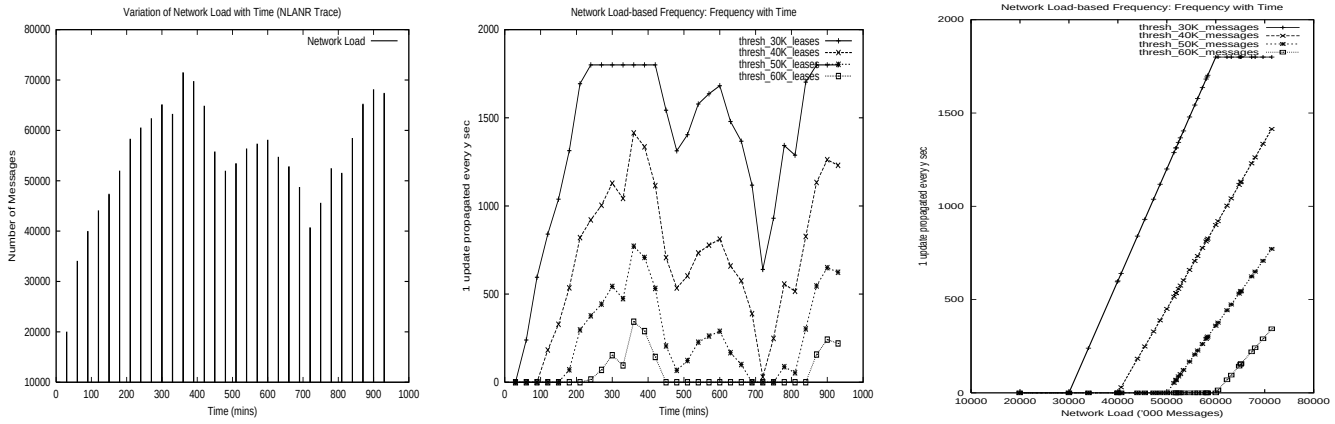


(a) Average Frequency (b) Violations (c) Control Message Overhead

Figure 23: Average frequency, Violations and Control Messages varying with the threshold server load

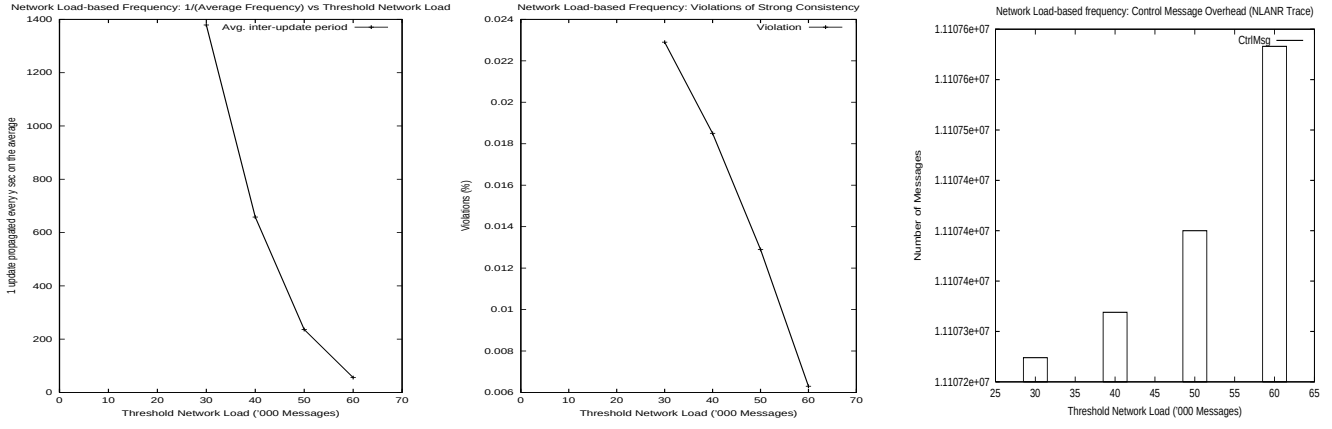
control message and file transfer overheads is small. Due to fewer updates propagated (see figure 20(a)), as stated above, the hit ratio decreases by small amount. The mean response time (see figure 21(a)) is not seen to change here. However expected results are better seen in the experiments conducted with the DEC Trace due to a higher number of updates propagated. Figure 21(b) shows violations of strong consistency guarantees to be about 0.0025% (if we propagate updates once every 5 seconds) to about 0.025% (if we propagate updates once every hour).

Now we show how we can dynamically adapt frequency to a resource one would like to conserve. Figures 22, 23, 24 and 25 show results of experiments we conducted by setting different thresholds representing the point beyond which the server is considered to be overloaded. So for instance say we want to provide consistency guarantees of at least 30 minutes (i.e., proxies in the cluser and the server are never out-of-sync by more than 30 minutes) and let our server load threshold be 12000 leases. All updates are propagated if the number of active leases at the server is less than this threshold. As described in the earlier section, if the number of active leases goes above 12000, our algorithm computes an appropriate frequency at which updates are propagated. Also if the number of active leases



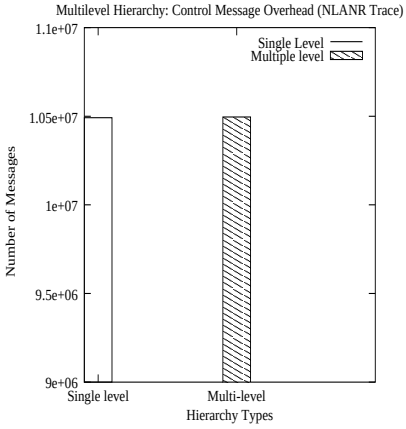
(a) Network Load (b) Tracking Frequency (c) Frequency vs. Load

Figure 24: Variation of network load and frequency with time; also frequency vs. network load

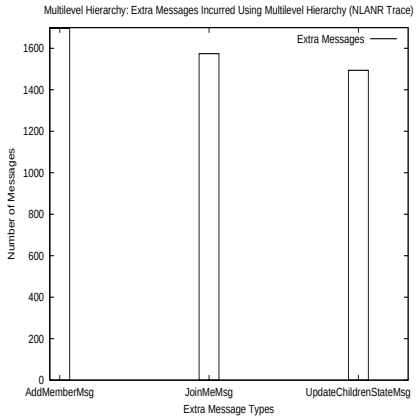


(a) Average Frequency (b) Violations (c) Control Message Overhead

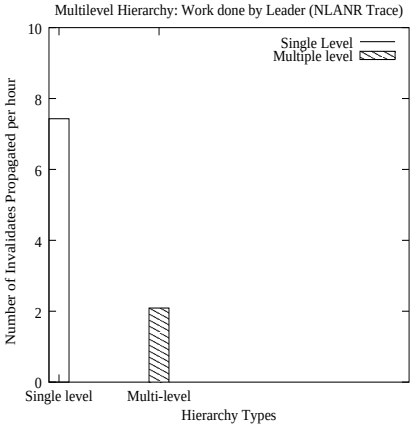
Figure 25: Average frequency, Violations and Control Messages varying with the threshold network load



(a) Control Message Overhead



(b) The Extra Messages



(c) Invalidate Propagation Rate at Leaders

Figure 26: Comparing single and multiple level hierarchical proxy organizations in a cluster

is ≥ 24000, updates get propagated at a frequency of 1 update every 30 mins, in accordance with guarantees meant to be provided by the system.

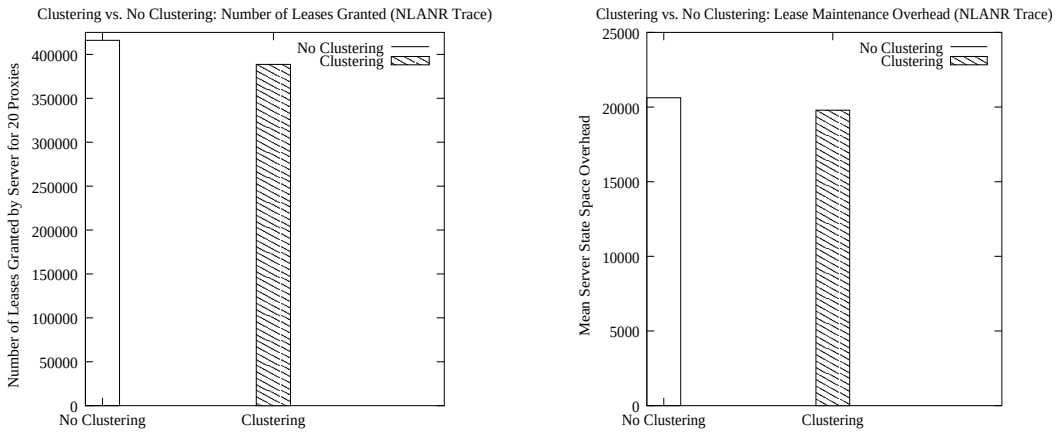
Figure 22 shows us the following: how the load at the server varies with time (graph 22(a)), how the frequency with which updates are propagated vary with time corresponding to the loads (graph 22(b)) and how frequency varies with server load (graph 22(c)). For the different thresholds, we see that all updates get propagated until the threshold is reached. The frequency then varies linearly with load until the peak load is reached after which updates are sent once every 30 minutes. Figure 23(a) shows how the mean inter-update period varies with increasing server load thresholds. The higher the threshold, the more the updates propagated hence the mean inter-update period decreases; the number of times strong consistency violations occur decreases with increasing threshold (graph 23(b)) and the number of control messages increases (graph 23(c)).

The results for frequency adaptation based on network load (figures 24 and 25) are similar. Network load is measured in terms of the number of messages received and sent by the server and thresholds selected for the experiment varied from 30000 to 70000 messages every half hour.

4.3.7 Multi-level Hierarchies

Here we show results of experiments we conducted using 15 proxies in a cluster. The branching factor input to the system is 2. i.e., if all proxies got requests for the same object, the per-object hierarchy would be a 3-level binary tree, with the leader proxy at root. We simulate 500000 requests of the NLANR trace here.

The graphs in figure 26 reflect the important results we get of our evaluation. As we move from one-level to multi-level hierarchies in a cluster, we expect the number of control messages to increase due to the extra messages involved in maintaining a hierarchy. However, the work required to be done by the leaders (in form of invalidate propagation to members, per object) is expected to decrease, depending on the number of proxies in the cluster and the input branching factor to the system.



(a) Number of Leases Granted (b) Mean Server State Space Overhead

Figure 27: Clustering vs. No Clustering - Number of Leases granted and Server State

Graph 26(a) shows an increase of about 4000 control messages in all as we adopt the logical multi-level hierarchical organization. The extra messages generated in the multi-level case (*AddMember*, *JoinMe* and *UpdateChildrenState* - also refer Section 3) are plotted in graph 26(b). At the cost of the above increases, we see the benefits of a multi-level hierarchy in graph 26(c) where there is a decrease of about 78% in the work done by the leader for propagating invalidates. While invalidates are propagated at the rate of about 7.43 invalidates per hour in the case of single level hierarchies, leaders in the multi-level case propagate invalidates at a rate of only 2.09 per hour.

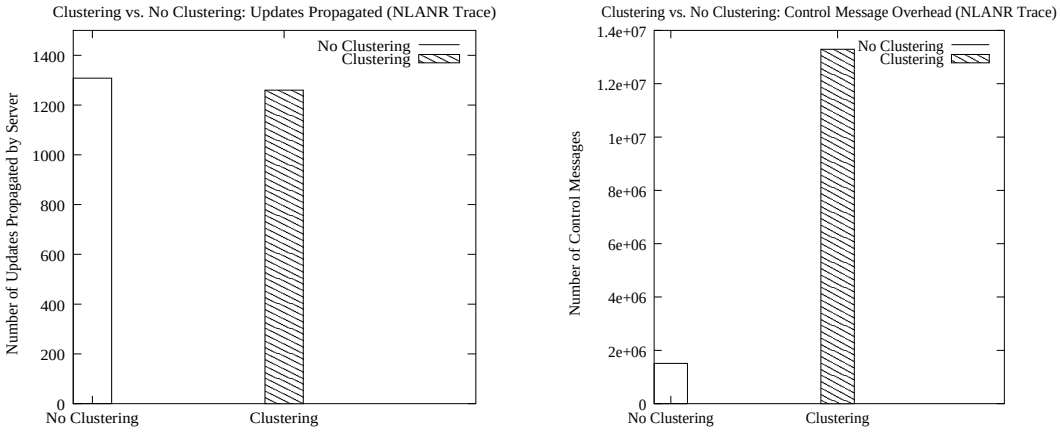
4.3.8 Scalability achieved with Clustering

In earlier work [], we presented and evaluated how leases may be deployed individual proxies. In this section we study the scalability achieved with *clustering* of proxies in terms of the amount of work done by the server for granting leases, maintaining leases and update propagation to leader proxies.

We keep the total number of proxies fixed at 20 and simulate 490000 requests.

If proxies are not organized as groups, then the per-object state maintained at the server would be high as more than one proxy may request an object causing he server to maintain per-proxy state as well. This in turn will cause the server to propagate multiple updates per object to the proxies interested in the object (i.e., those which have leases on the object). As the number of proxies increase, this could cause high overhead at the server in terms of the number of leases granted, the number of leases maintained and therefore the number of updates propagated. Therefore deployment of leases on a per-proxy basis is indeed not scalable. However, if we group proxies into a cluster, there is a need to maintain only per-object state for one proxy (the leader proxy for the object). Hence the server need propagate updates only to the leader (which in turn propagates invalidates to its members, if any). However, the cost of such benefits will reflect in increased number of control messages due to broadcasts required for co-operation.

In Graph 27(a), we see the decrease in the number of leases granted when we employ clustering. This decrease, though just 4% for this trace, is so because with clustering, once a proxy in the cluster requests an object, becomes leader for the object and gets the lease for the object from the server, all further requests for the object are handled



(a) Number of Updates Propagated (b) Control Message Overhead

Figure 28: Clustering vs. No Clustering - Number of Updates Propagated and Control Message Overhead

by the leader proxy (as long as one exists) and do not go to the server and thus do not result in further lease grants by the server. Requests for an object go to the server if and only if there does not exist a leader proxy for the object. Graph 27(b) reflects the average state space overhead required to maintain leases. We see a 4.5% decrease in the amount of state required to be maintained by the server. With clustering, since the server now need communicate with fewer proxies per object, the number of updates propagated by the server is seen to decrease by 3.6% (graph 28(a)). However, for these benefits, we see an eight-fold increase in the number of control messages in graph 28(b). Possible ways to deal with such increases are mentioned in the following sub section.

4.3.9 Multiple Clusters

Here we try to understand how the number of proxies in a cluster (in effect the number of clusters given a group of proxies) affect the relevant overheads.

First we address possible ways to deal with the increase in control messages as we move from no clustering to clustering. The large increases are due to the broadcast messages required for co-operative caching in clusters (*broadcastLeaderId* and *terminate* messages). A possible and well-known way to get over this problem is to deploy IP multicast. Since IP multicast is not widely deployed, we have not assumed it for all the experiments we have performed. However, if we assume the deployment of IP multicast and also assume that the proxies in a cluster can form multicast groups, the following figure depict the effect of multicast on control message overhead. Note that studies have compared caching and multicast for improved WWW performance [36]. Here we suggest the use of both for scalable consistency.

Contrast the above graph with graph 28(b). Instead of an eight-fold increase in the total number of control messages, we see only a 25% increase when we employ IP multicast (the 20-cluster plot for a group of 20 proxies reflects *no clustering* in effect; compare this with the 1-cluster plot). Note that with decreasing number of clusters (from 10 down to 1), the number of leaders decrease causing a decrease in the number of *broadcastLeaderId* messages. This is one of the causes for an overall decrease in the control message overhead. The other reason is, as proxies get more and more distributed across clusters, chances that terminates occur are higher (hence lower number of

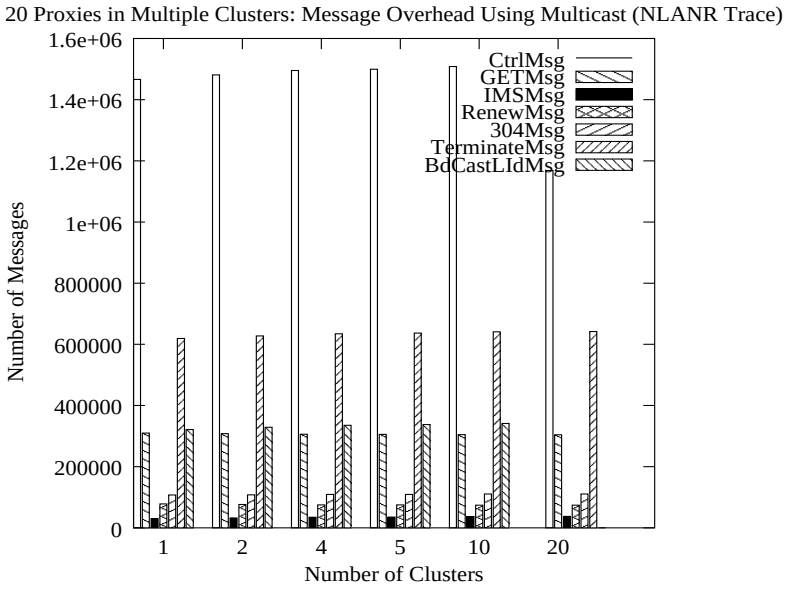


Figure 29: Overcoming the message overhead using clustering : Multicast

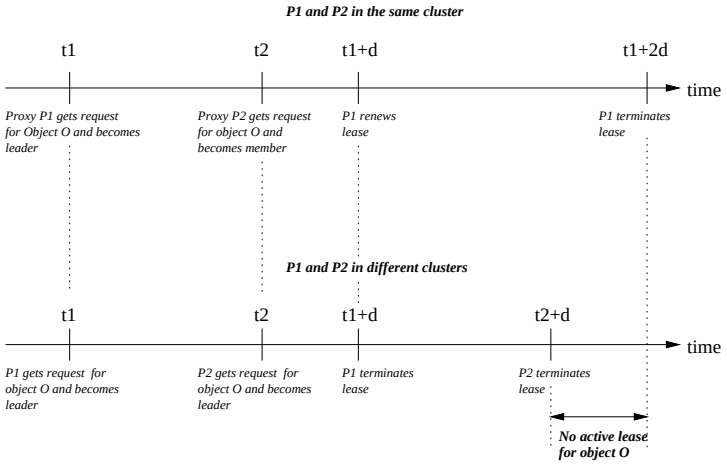
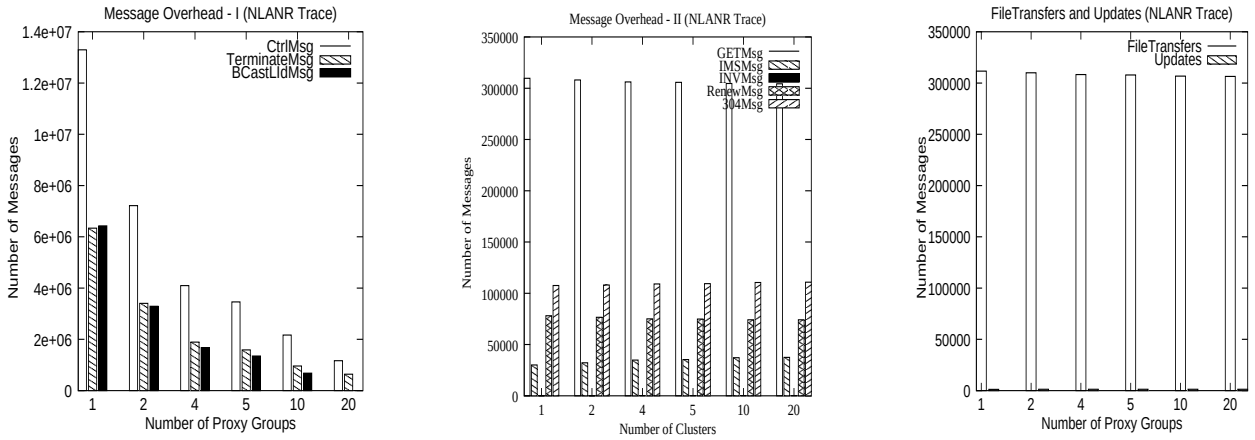


Figure 30: Possible reason as to why leases are shorter lived when we distribute a fixed number of proxies over an increasing number of clusters.



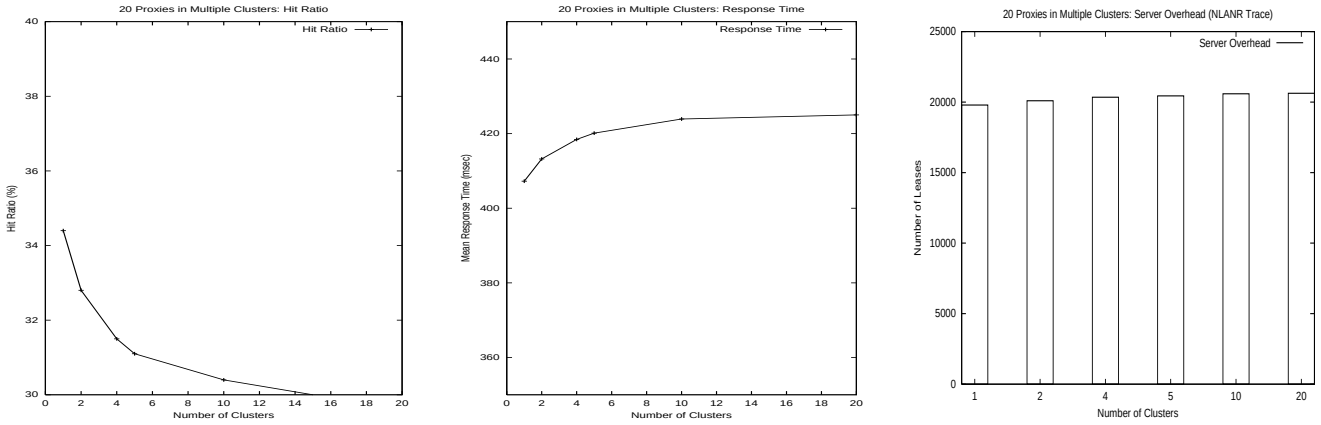
(a) Message Overhead - I (b) Message Overhead - II (c) File Transfers and Updates

Figure 31: Message overhead with increasing number of clusters

terminate messages with a lower number of clusters) and therefore object leases are shorter-lived. Similarly, as the sizes of clusters decrease, the chances that a lease gets renewed because of interested proxies, is less. We thus see an increase in the number of renewals when we go from 10 to 1 cluster. This increase is small (see below) and hence the overall control message overhead decreases with a decrease in the number of clusters. To explain all this, let us consider a simple 2-proxy cluster (also refer figure 30). Say at time $t1$, $P1$ becomes leader for object O. At a later time $t2 \leq t1 + d$, $P2$ gets a request for object O. At time $t1 + d$, $P2$ does not satisfy the termination condition and therefore the lease gets renewed. On the other hand, if $P1$ and $P2$ were in different clusters, $P1$ would terminate at time $t1 + d$ and $P2$ would terminate its lease at time $t2 + d$, of course assuming no further requests. As a direct consequence, the number of renewals increases with a decrease in the number of clusters across which a fixed number of proxies get divided. But as this increase is not high, we see an overall decrease in the number of control messages (about 4%) with decrease in number of clusters.

Therefore, deploying IP multicast is indeed a possible way to deal with large control message overhead involved when trying to provide scalable consistency solutions. It not only avoids the large increase in network bandwidth consumption as we go from no clustering to 1 cluster, there increase there on (from 1 to 10 clusters) is only about 5%.

Now we study the effects of varying the number of clusters across which a fixed number of proxies get divided. For this set of experiments, we assume a the number of proxies to be 20 in all and simulate 490000 requests. We then vary the number of clusters from 1 to 20 comprising 20 to 1 prox(y)ies each. With an increasing number of clusters, note that the number of leader proxies per object increases; however the degree of cooperation decreases with decreasing number of proxies per cluster. This causes a decrease in the number of broadcast messages (*broadcastLeaderId* and *terminate* messages). These two components result in a decrease in the total number of control messages (about 89%) as we grow from one to ten clusters (see graph 31(a)). Graph 31(c) plots the number of file transfer messages and the number of updates propagated from the server to the leader. While the total number of file transfer messages remains about the same, the number of update messages increases by about 4.5% (1 to 10 clusters). There are more updates because the number of leaders per object are likely to increase with increasing number of independent



(a) Hit Ratio (b) Response Time (c) Server Overhead

Figure 32: Hit Ratio, Mean Response Time and Server State Space with increasing number of clusters

clusters, hence the number of updates a server propagates is likely to increase. Graph 32(c) plots the overhead at the server. Note that as the number of clusters increase, the number of leaders increase, causing the server to maintain more state for an object. We see a 10% increase in state space overhead at the server as we move from one to ten clusters. Graphs 32(a) and (b) examine how the overall hit ratio and mean response time vary with the number of clusters. With increase in the number of clusters, the rate at which objects get replicated reduce due to fewer number of proxies per cluster; hence we see a 10% reduction in hit ratio and a 6% increase in mean response time.

5 Concluding Remarks

From what we have seen in all previous sections, we should take home the following messages:

- *A note on Leases*: Leases is a mechanism that can provide strong consistency guarantees on the Web. However, deploying leases on a single-proxy basis is not a scalable solution for the following reasons: (i) The server has to maintain per-object and per-proxy state. (ii) The server will have to compute and grant leases on a per-object, per-proxy basis. These problems pave the way for swamping of web servers. (iii) The mechanism requires updates and/or invalidates to be propagated as and when changes occur at the server. If the write frequency of objects is high, or if the number of proxies grow and the read frequency is high, it results in high network overhead.
  We presented possible ways of allowing the leases mechanism to scale.

- *Frequency:* The introduction of the frequency parameter allows a server to dynamically adapt update propagation based on server and/or network loads. This is of course, based on the assumption that applications today can tolerate occasional violations of consistency guarantees. We showed that in fact, with the introduction of the frequency parameter, our mechanism could guarantee Δ-consistency.

- *Clustering:* By creating subgroups of proxies in a CDN that we call *clusters*, the state space overhead at the server is greatly reduced as the server now only need maintain per-object, per-cluster state. This can be achieved by election of a leader proxy per-cluster and making the leader take the responsibility of lease renewals and invalidate propagation to other member proxies in the cluster. We studied different ways by which leaders could be elected and studied their trade-offs. We also studied the effects the clustering to show reduced number of leases granted, leases maintained and hence updates propagated. However, clustering causes a significant increase in the control message overhead. We suggested the employment of IP multicast as a possible scalable solution to the problem.

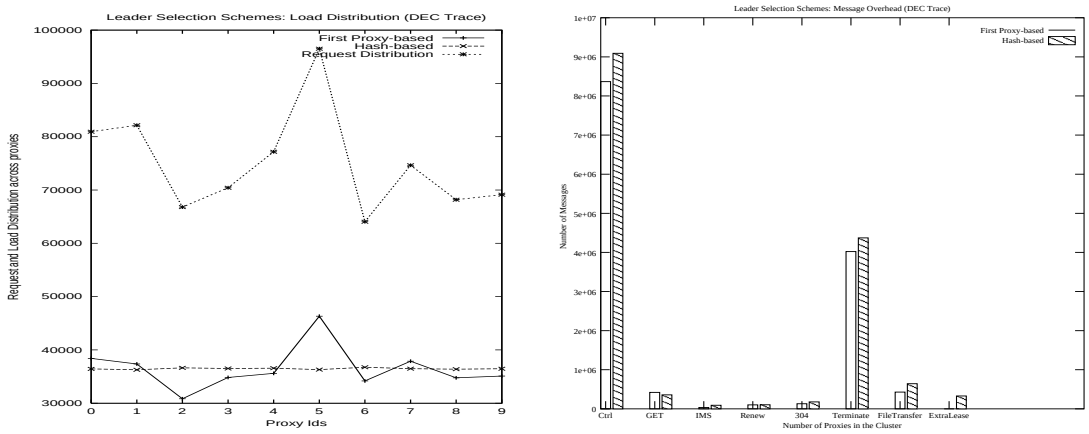
- *Size and Number of Clusters:* We studied the effect of dividing a fixed number of proxies across varying number of clusters. We saw that while there is significant decrease in the control message overhead, the hit ratios decrease and the mean response times increase. In addition, the number of leases maintained by the server increases because it maintains multiple leases per-object, with multiple clusters.

- *Renewal Policies:* We studied two different renewal policies (the eager and lazy policies). We saw that eager renewals provide better hit ratio and response times at the cost of higher network bandwidth consumption and higher server state space overhead. We also saw that with increase in the lease duration of leases granted, the results of the experimental evaluation of the eager and lazy policies tend to converge. If a CDN can provide even more network bandwidth, we see that hit ratios and response times can be improved further by varying the lease termination policy.

- *Updates vs. Invalidates:* Since the propagation of updates alone is potentially more expensive than the propagation of invalidates alone, we suggested two schemes by which the decision of whether to propagate an update or an invalidate could be dynamically taken and thus span the update-invalidate spectrum. We saw that

the renewal-based policy did better than the size-based policy in propagating those updates that were required (i.e., for those objects that were accessed/requested more) since the number of successive renewals is indeed an indicator of object popularity.

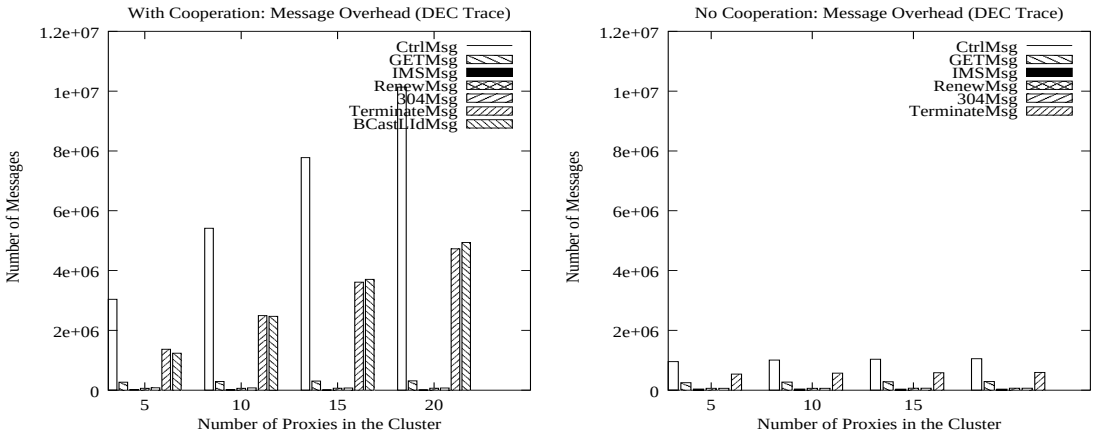
6 Appendix A - Results with the DEC Trace



(a) Load Distribution (b) Message Overhead

Figure 33: Comparing Leader Selection Schemes in terms of load balancing and message overhead

Figure 33 corresponds to figure 3 in Section 4.



(a) With Cooperation (b) No Cooperation

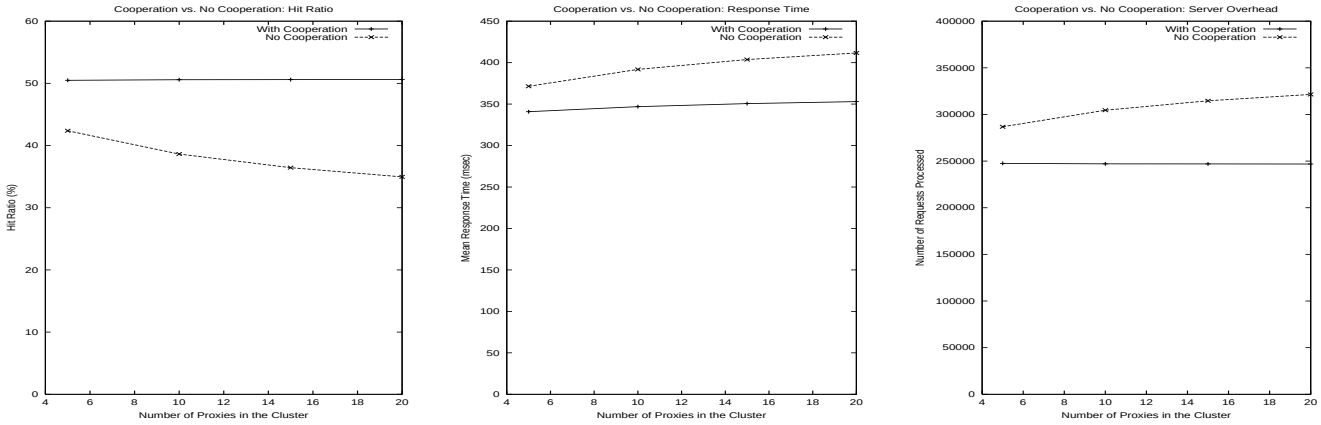
Figure 34: Message Overhead comparison for cooperative caching

Figure 34 corresponds to figure 4 in Section 4.

Figure 35 corresponds to figure 5 in Section 4.

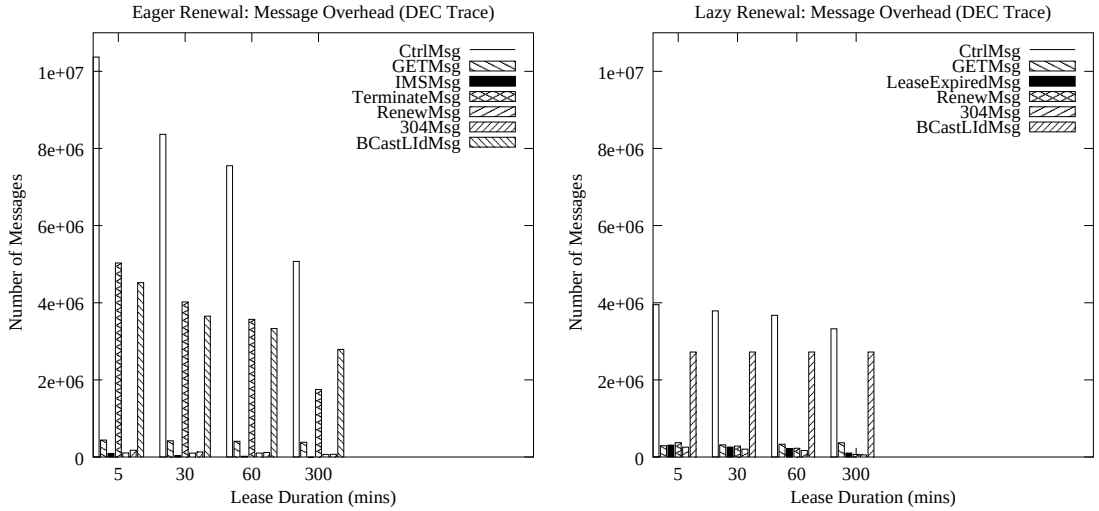
Figure 36 corresponds to figure 6 in Section 4.

Figure 37 corresponds to figure 7 in Section 4.



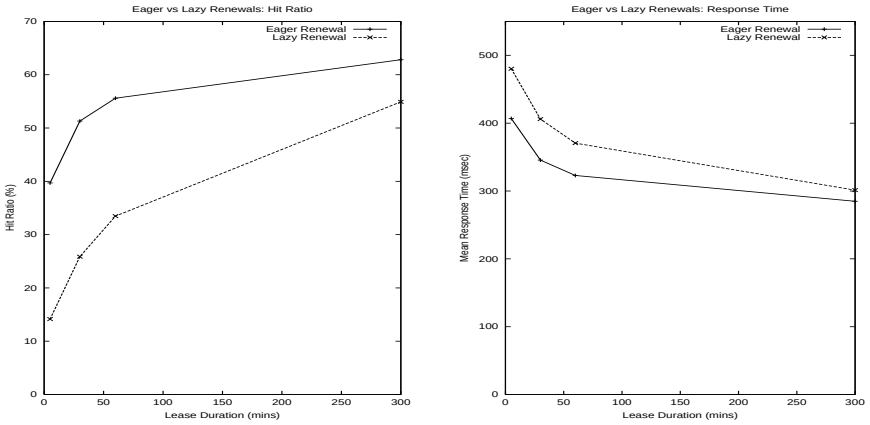
(a) Hit Ratio (b) Response Time (c) Server Overhead

Figure 35: Effects of cooperative caching on hit ratio, response time and server overhead



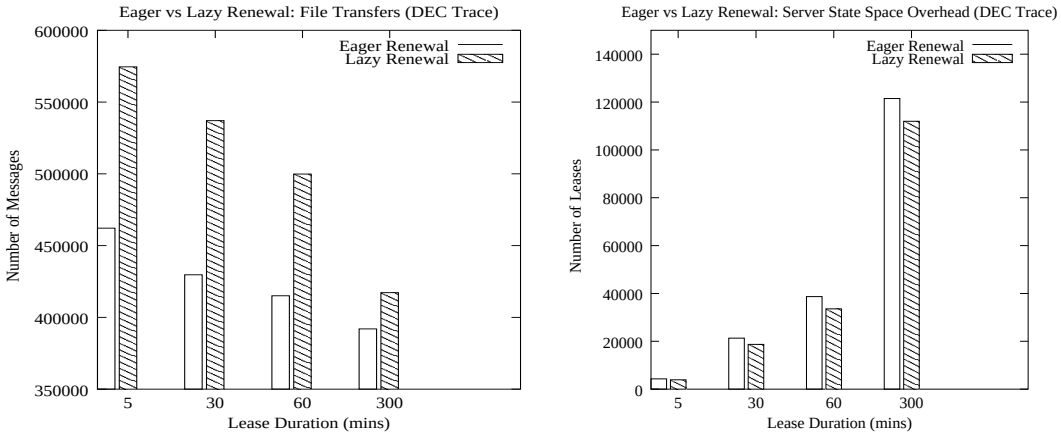
(a) Eager Renewal (b) Lazy Renewal

Figure 36: Message overhead comparison for eager and lazy renewal policies.



(a) Hit Ratio (b) Response Time

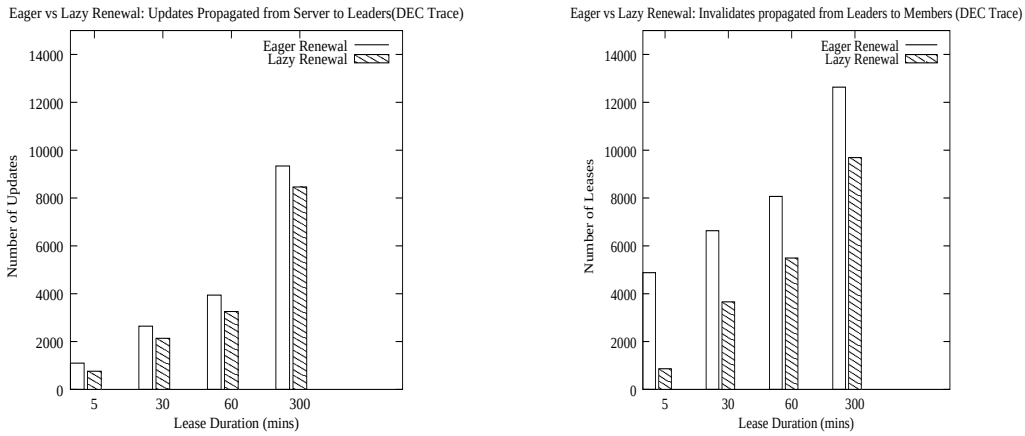
Figure 37: Hit Ratio and Response Time comparison for eager and lazy renewal policies.



(a) Number of File Transfers (b) Server State Space Overhead

Figure 38: File Transfers and State Space Overhead comparison for eager and lazy renewal policies.

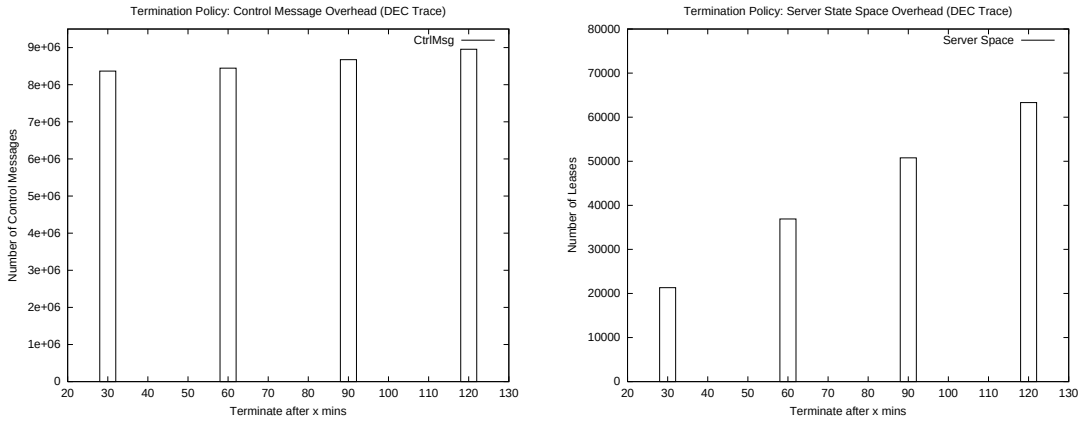
Figure 38 corresponds to figure 8 in Section 4.



(a) Number of Updates (b) Number of Invalidates

Figure 39: Updates (from server to leaders) and Invalidates (from leaders to members) comparison for eager and lazy renewal policies.

Figure 39 corresponds to figure 9 in Section 4.



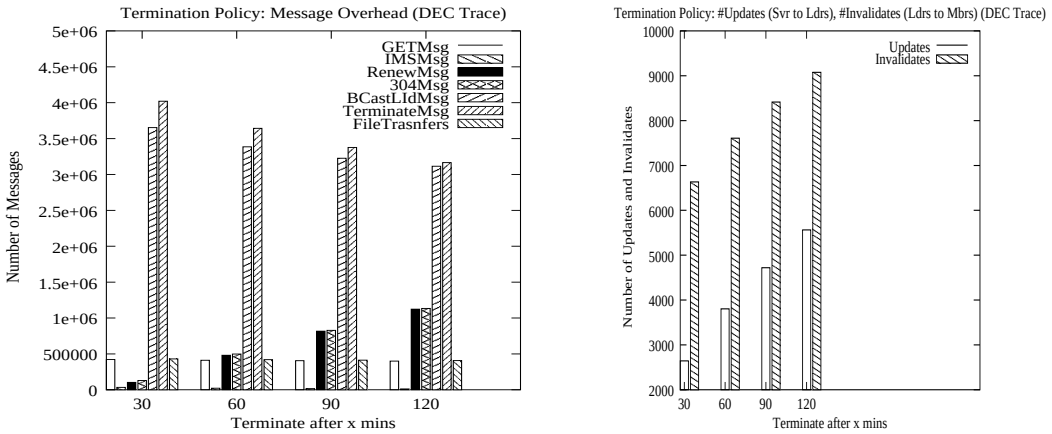
(a) Control Messages (b) Server State Space Overhead

Figure 40: The Control Message and Server State Space Overheads

Figure 40 corresponds to figure 10 in Section 4.

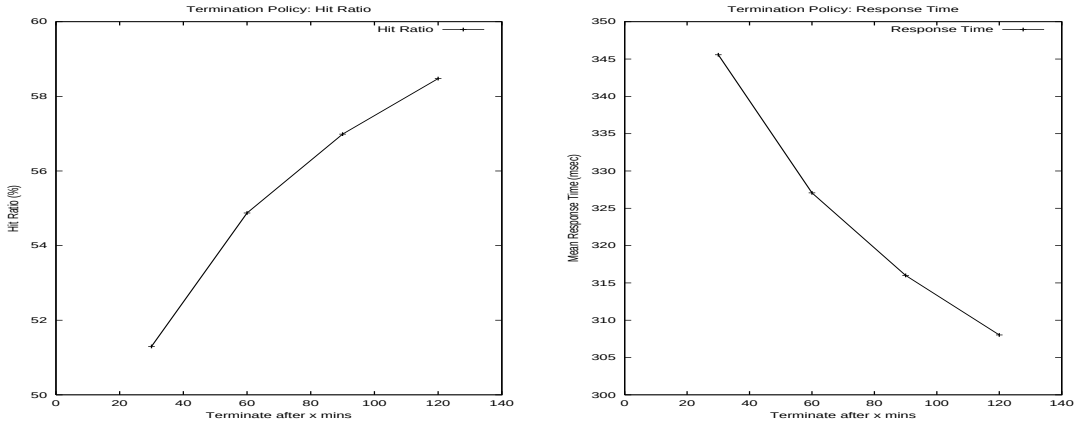
Figure 41 corresponds to figure 11 in Section 4.

Figure 42 corresponds to figure 12 in Section 4.



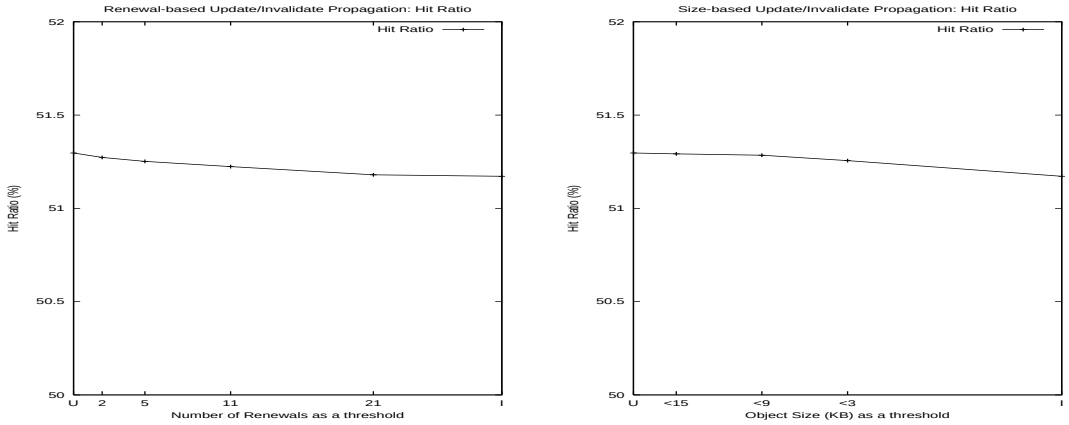
(a) Message Types (b) Updates and Invalidates

Figure 41: Message Overhead Types and Number of updates and invalidates propagated



(a) Hit Ratio (b) Response Time

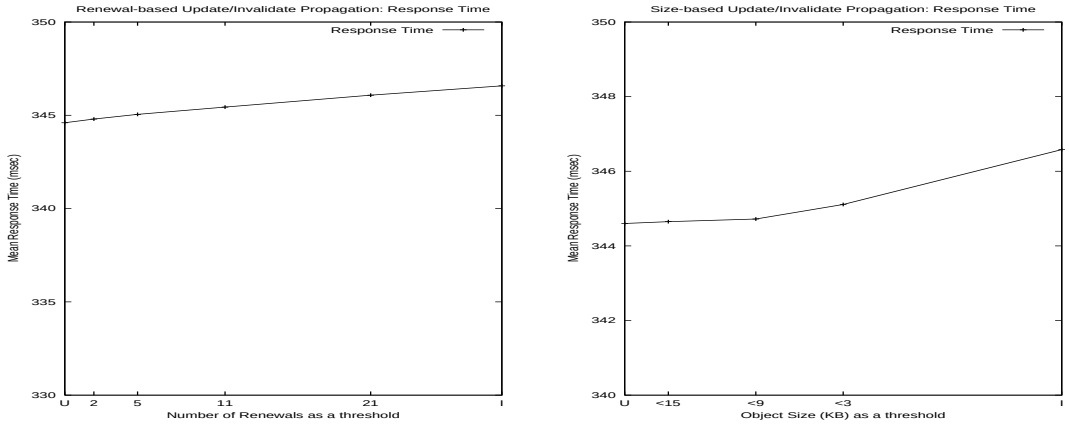
Figure 42: The Hit Ratio and Response time with varying Termination durations



(a) Renewal-based: Hit Ratio (b) Object Size-based: Hit Ratio

Figure 43: Hit Ratio with varying thresholds

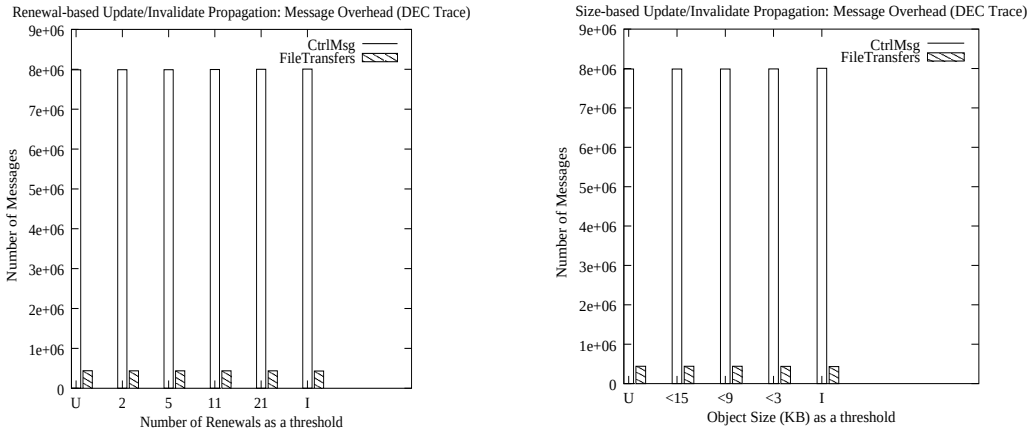
Figure 43 corresponds to figure 13 in Section 4.



(a) Renewal-based: Response Time (b) Object Size-based: Response Time

Figure 44: Response Time with varying thresholds

Figure 44 corresponds to figure 14 in Section 4.



(a) Renewal-based: Message Overhead (b) Object Size-based: Message Overhead

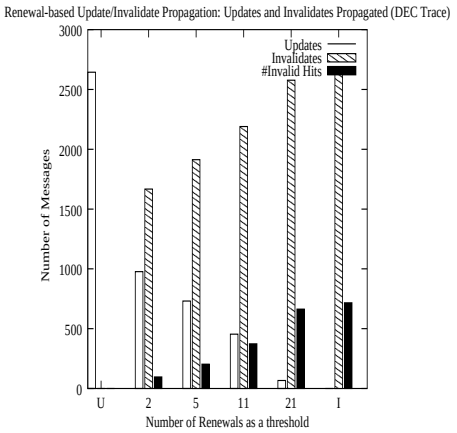
Figure 45: Message overhead with varying thresholds

Figure 45 corresponds to figure 15 in Section 4.

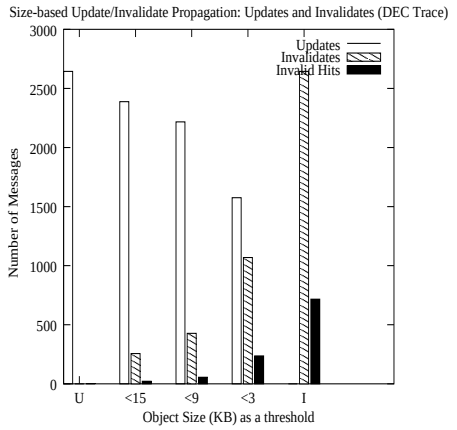
Figure 46 corresponds to figure 16 in Section 4.

Figure 47 corresponds to figure 17 in Section 4.

Figure 48 corresponds to figure 18 in Section 4.

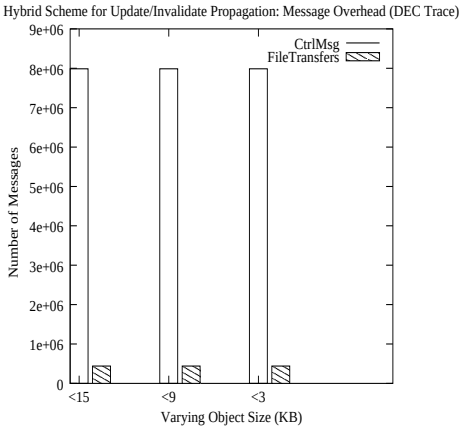


(a) Renewal-based: Updates and Invalidates

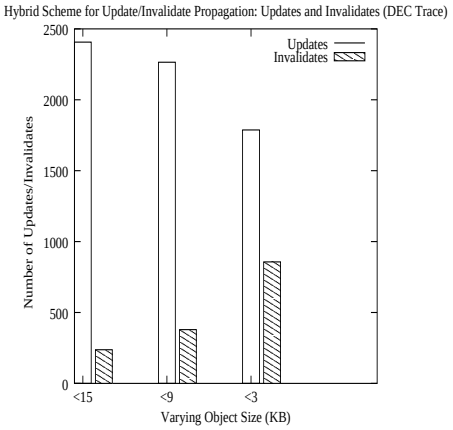


(b) Object Size-based: Updates and Invalidates

Figure 46: Number of updates and invalidates propagated from Server to Leaders, with varying thresholds

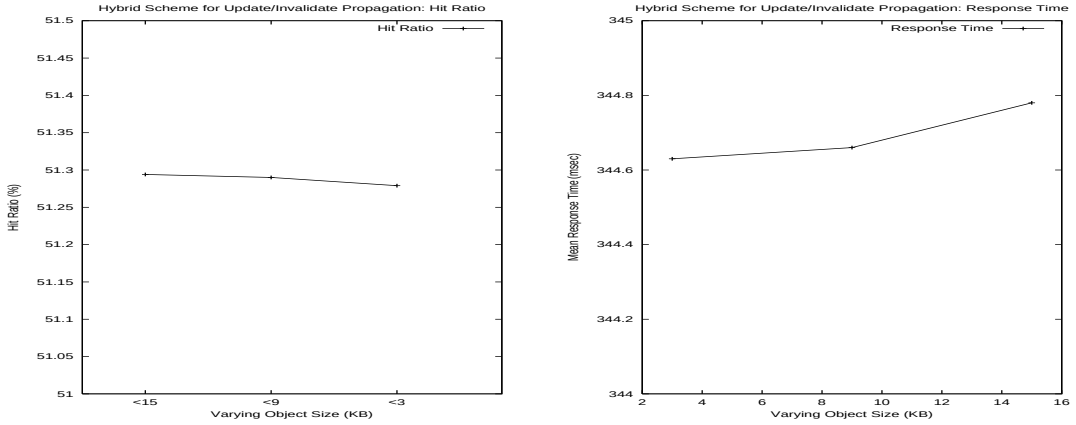


(a) Hybrid scheme: Message Overhead



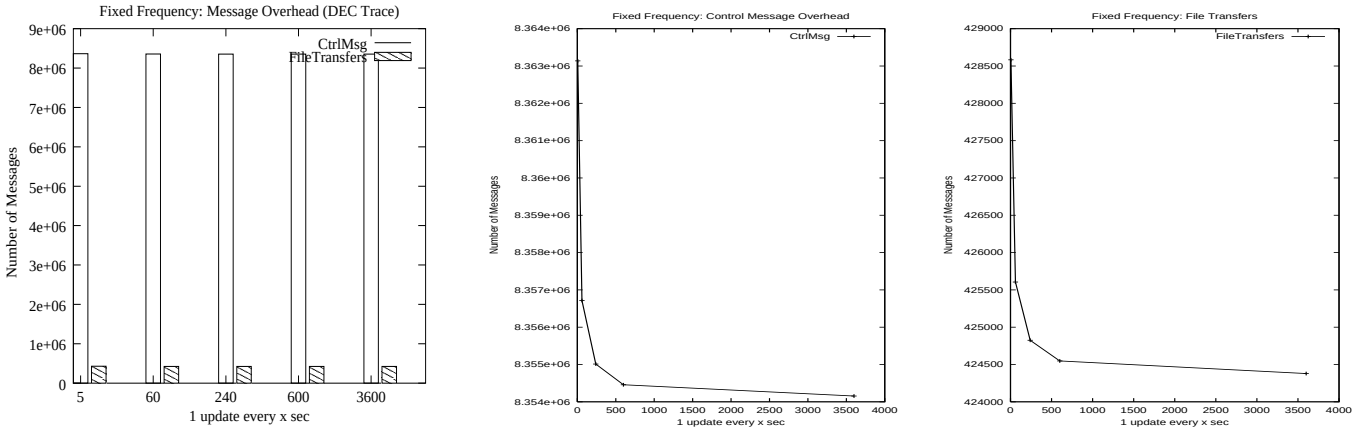
(b) Hybrid scheme: Updates and Invalidates

Figure 47: Message overhead and updates/invalidates propagated from Server to Leaders, with varying thresholds using the hybrid scheme



(a) Hybrid scheme: Hit Ratio (b) Hybrid scheme: Response Time

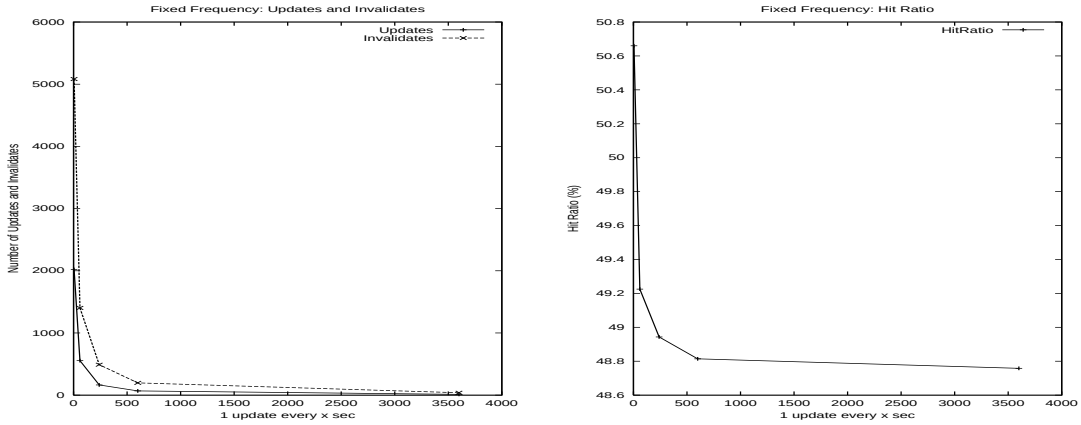
Figure 48: Hit Ratio and Response Time, with varying thresholds using the hybrid scheme



(a) Control Messages and File Transfers (b) Control Messages (c) File Transfers

Figure 49: Control Message Overhead and File Transfers with varying frequency

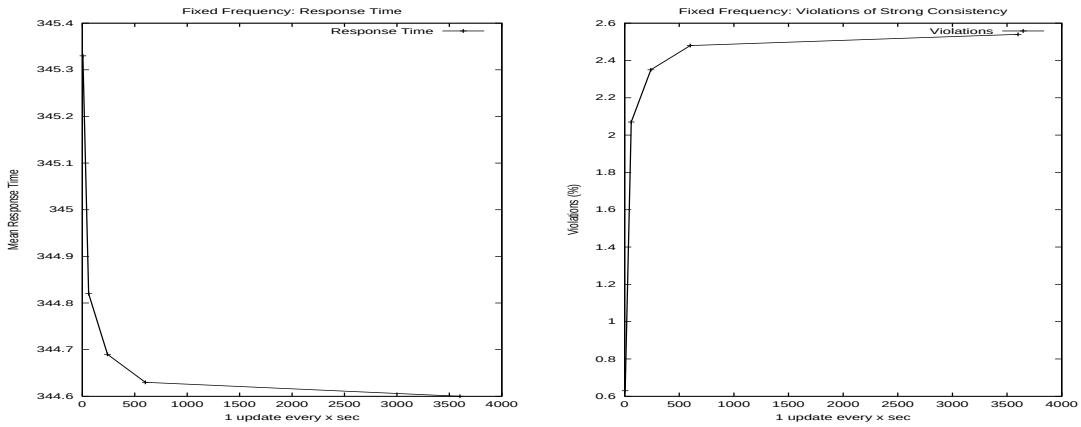
Figure 49 corresponds to figure 19 in Section 4.



(a) Updates and Invalidates (b) Hit Ratio

Figure 50: Number of updates/invalidates and hit ratio with varying frequency

Figure 50 corresponds to figure 20 in Section 4.



(a) Response Time (b) Violations

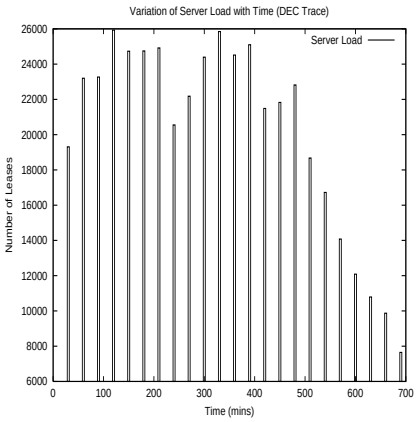
Figure 51: Mean Response Time and Percentage of Violations varying with frequency

Figure 51 corresponds to figure 21 in Section 4.

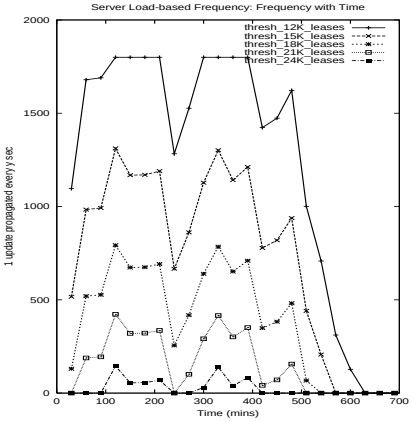
Figure 52 corresponds to figure 22 in Section 4.

Figure 53 corresponds to figure 23 in Section 4.

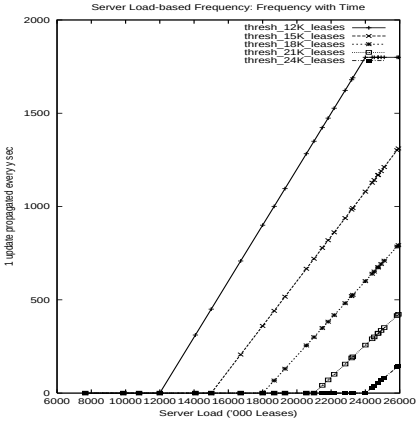
Figure 54 corresponds to figure 24 in Section 4.



(a) Server Load

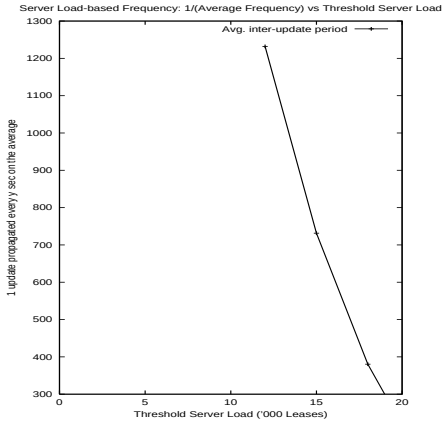


(b) Tracking Frequency

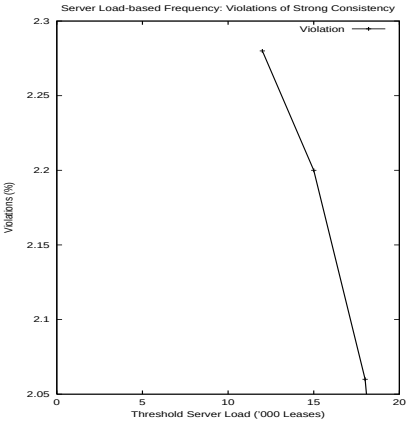


(c) Frequency vs. Load

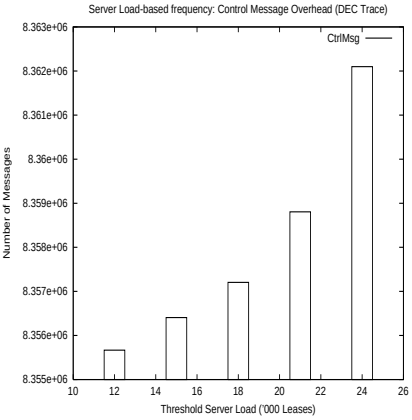
Figure 52: Variation of server load and frequency with time; also frequency vs. server load



(a) Average Frequency

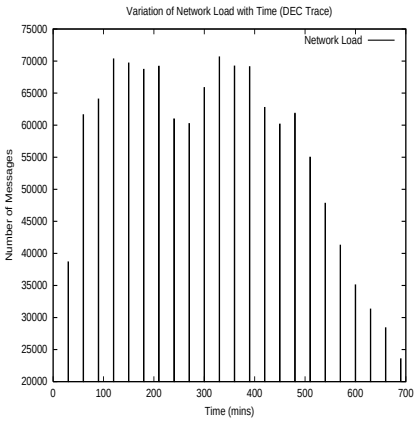


(b) Violations

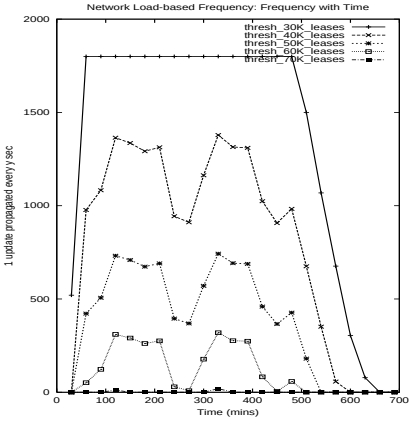


(c) Control Message Overhead

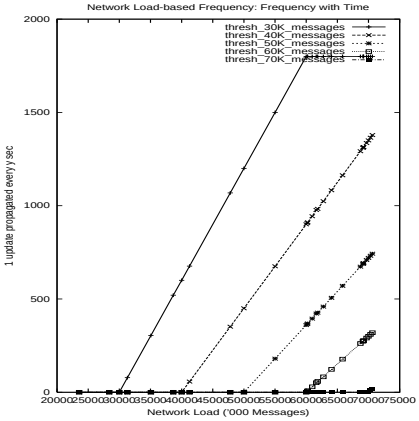
Figure 53: Average frequency, Violations and Control Messages varying with the threshold server load



(a) Network Load

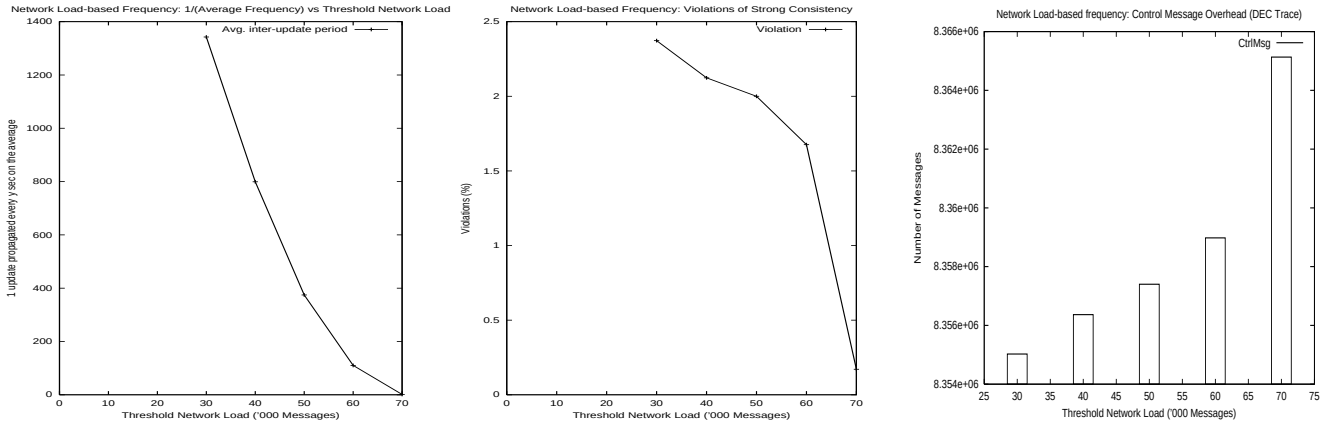


(b) Tracking Frequency



(c) Frequency vs. Load

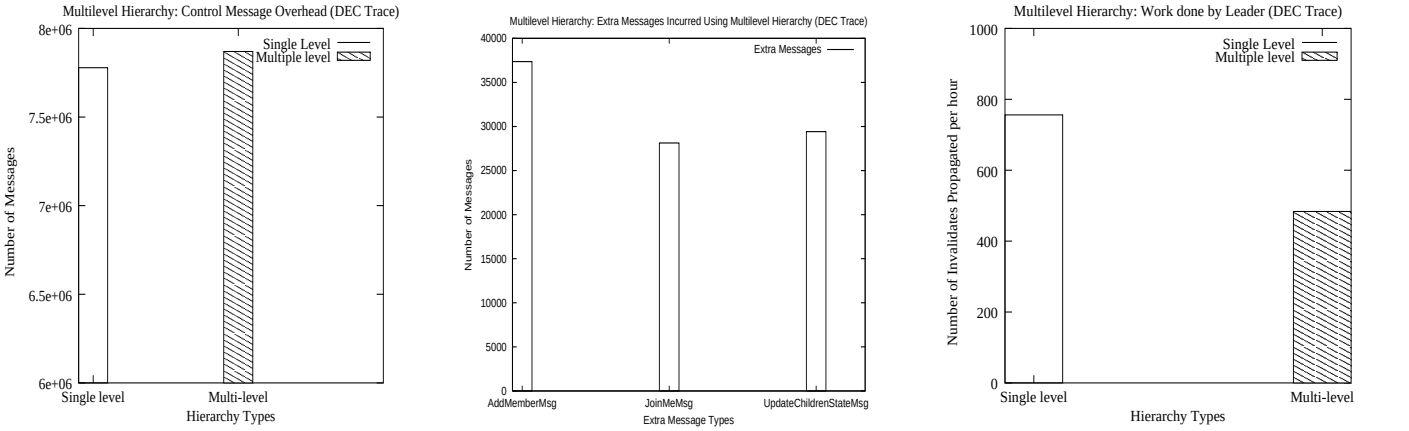
Figure 54: Variation of network load and frequency with time; also frequency vs. network load



(a) Average Frequency (b) Violations (c) Control Message Overhead

Figure 55: Average frequency, Violations and Control Messages varying with the threshold network load

Figure 55 corresponds to figure 25 in Section 4.



(a) Control Message Overhead (b) The Extra Messages (c) Invalidate Propagation Rate at Leaders

Figure 56: Comparing single and multiple level hierarchical proxy organizations in a cluster

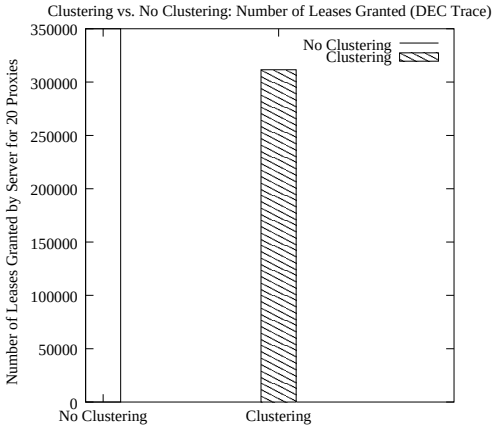
Figure 56 corresponds to figure 26 in Section 4.

Figure 57 corresponds to figure 27 in Section 4.

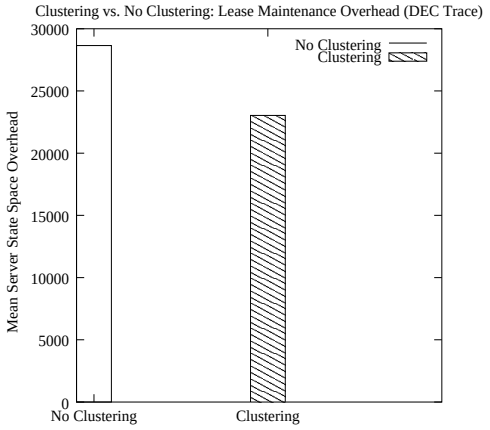
Figure 58 corresponds to figure 28 in Section 4.

Figure 59 corresponds to figure 29 in Section 4.

Figure 60 corresponds to figure 31 in Section 4.

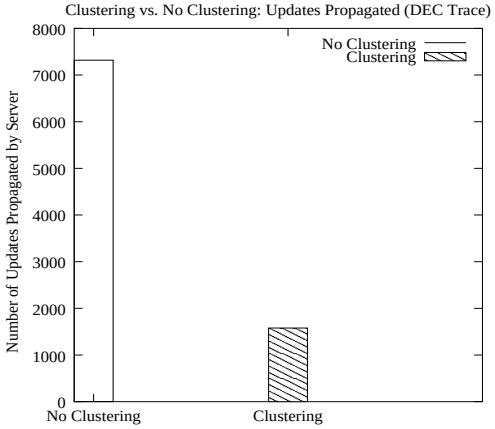


(a) Number of Leases Granted

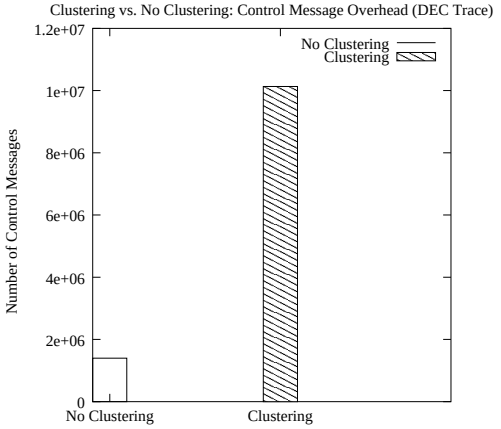


(b) Mean Server State Space Overhead

Figure 57: Clustering vs. No Clustering - Number of Leases granted and Server State



(a) Number of Updates Propagated



(b) Control Message Overhead

Figure 58: Clustering vs. No Clustering - Number of Updates Propagated and Control Message Overhead

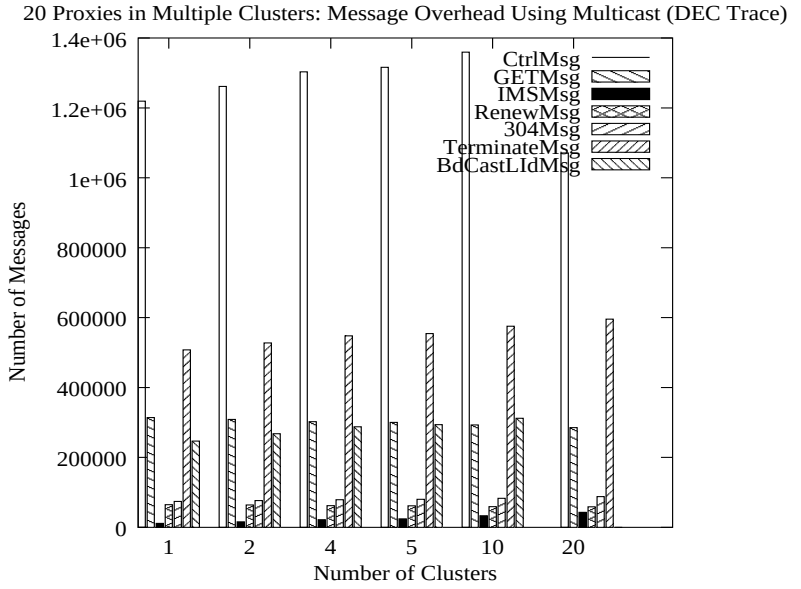
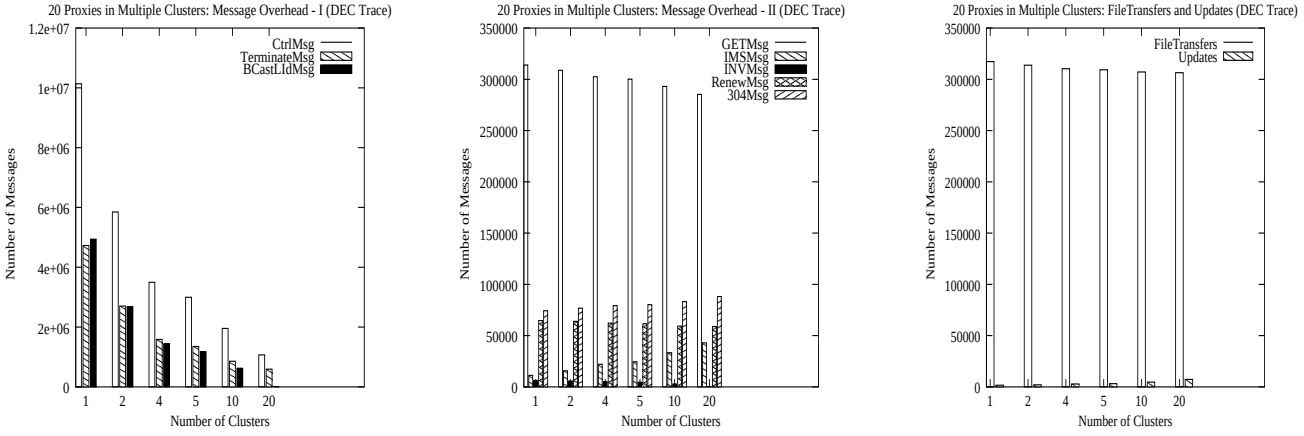
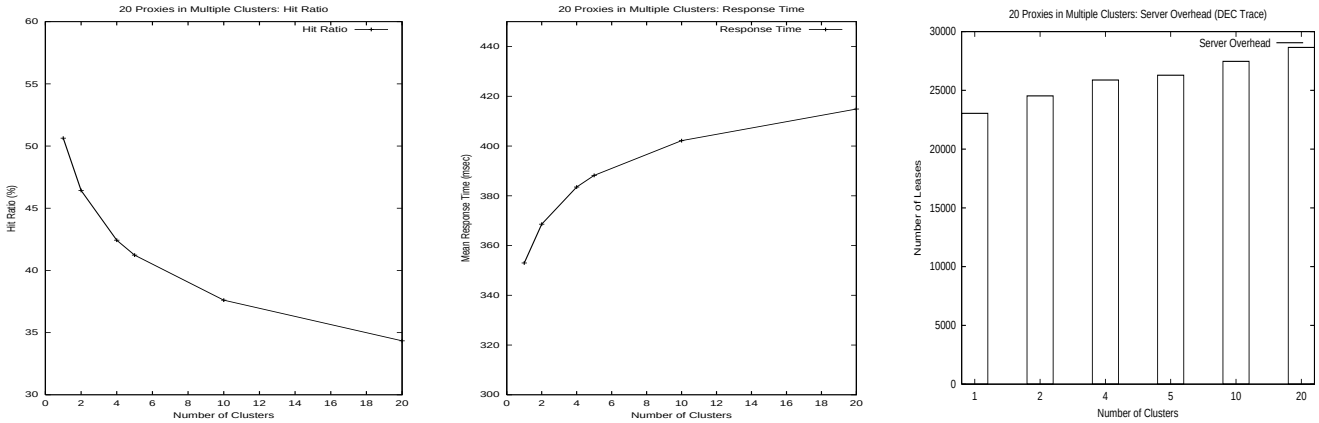


Figure 59: Overcoming the message overhead using clustering : Multicast



(a) Message Overhead - I (b) Message Overhead - II (c) File Transfers and Updates

Figure 60: Message overhead with increasing number of clusters



(a) Hit Ratio (b) Response Time (c) Server Overhead

Figure 61: Hit Ratio, Mean Response Time and Server State Space with increasing number of clusters

Figure 61 corresponds to figure 32 in Section 4.

7 Appendix B - Grammar for lease related requests

entity-header-extension	=	lease-control | object-control CRLF lease-control
lease-control	=	”Lease-Control””:” lease-directive
object-control	=	”Object””:” HTTP-URL
lease-directive	=	lease-request-directive | lease-response-directive | lease-invalidate-directive | lease-update-directive | lease-info-directive | leaseend-time | Imleader-broadcast | Addto-group | lookup-directive | lease-invalidate-ack | lease-upate-ack | lease-terminate-ack | lookup-ack | Addto-group-ack | ImLeader-ack | lease-ack
lease-request-directive	=	”Grant-Lease” | ”Renew-Lease” — ”Terminate-Lease”
lease-response-directive	=	”Lease””:” lease-period | ”Deny-Lease” — ”Granted-Lease”
lease-period	=	lease-start ”-” lease-end
lease-start	=	HTTP-date
lease-end	=	HTTP-date
lease-invalidate-directve	=	”Object-Invalidated”
lease-update-directive	=	”Object-Updated”
lease-info-directive	=	”Lease-Info””:” lease-period
leaseend-time	=	”GetLastref-Time”
Imleader-broadcast	=	”Leader-for-Object”
Addto-group	=	addproxy | addproxy HTTP-ID
addproxy	=	”AddProxy-to-Group”
lookup-directive	=	”Lookup”
lease-invalidate-ack	=	”Invalidate-Ack” ack
lease-update-ack	=	”Update-Ack” ack
lease-terminate-ack	=	”Terminate-Ack” ack
lookup-ack	=	”Lookup” ack HTTP-ID
Addto-group-ack	=	”AddProxy-to-Group” ack
Imleader-ack	=	”Leader-for-Object” ack
lease-ack	=	”Lease-Info” ack
ack	=	”OK” | ”FAILED”

Figure 62: User-defined extensions to HTTP/1.1 to incorporate leases.

References

[1] Akamai: http://www.akamai.com.

[2] Speedera: http://www.speedera.com.

[3] Squid: http://www.squid-cache.org.

[4] G. Agarwal, R. Shah, and J. Walrand. Content distribution architecture using network layer anycast. In *IEEE Workshop on Internet Applications*, 2001.

[5] Scott M. Baker and Bongki Moon. Distributed cooperative web servers. *WWW8 / Computer Networks*, 31(11-16):1215–1229, 1999.

[6] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching, and zipf-like distributions: Evidence, and implications, 1999.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching, and zipf-like distributions: Evidence, and implications, 1999.

[8] A. Chankhunthod, P. Danzig, C. Neerdaels, and M. Schwartz. Worrell: A hierarchical internet object cache, 1995.

[9] P. Deolasse, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull of dynamic web data: Better resiliency, scalability and coherency. In *10th International World Wide Web Conference*, 2001.

[10] V. Duvvri, P. Shenoy, and R. Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. In *INFOCOM*, 2000.

[11] S. Dykes and K. Robbins. A viability analysis of cooperative proxy caching.

[12] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[13] Z. Fei. A novel approach to managing consistency in content distribution networks. In *6th International Web Caching Workshop and Content Delivery Workshop*, Boston, MA, 2001.

[14] P. Francis. Yoid: extending the multicast internet architecture, 1999.

[15] S. Gadde, J. Chase, and M. Rabinovich. Directory structures for scalable internet caches, 1997.

[16] S. Gadde, J. Chase, and M. Rabinovich. Web caching and content distribution: A view from the interior, 2000.

[17] Syam Gadde, Michael Rabinovich, and Jeff Chase. Reduce, reuse, recycle: An approach to building large internet caches. In *6th Workshop on Hot Topics in Operating Systems*, Cape Cod, Massachusetts, USA, 1997.

[18] C. G. Gray and D. R. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, volume 23-5, pages 202–210, 1989.

[19] James Gwertzman and Margo Seltzer. World-Wide Web cache consistency. In *Proceedings of the 1996 Usenix Technical Conference*, 1996.

[20] K. Johnson, J. Carr, and M. Day M Kaashoek. The measured performance of content distribution networks. In *5th International Web Caching Workshop and Content Delivery Workshop*, 2000.

[21] J. Kangasharju, J. Roberts, and K. Ross. Performance evaluation of redirection schemes in content distribution networks, 1999.

[22] K. Kangasharju, J. Roberts, and K. Ross. Object replication strategies in content distribution networks. In *6th International Web Caching Workshop and Content Delivery Workshop*, Boston, MA, 2001.

[23] D. Karger, E. Lehman, F. Leighton, M. Levin, D. Lewin, and R. Panigraphy. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web, 1997.

[24] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1203–1213, 1999.

[25] Balachander Krishnamurthy and Craig E. Wills. Study of piggyback cache validation for proxy caches in the world wide web. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[26] Dan Li and David R. Cheriton. Scalable web caching of frequently updated objects using reliable multicast. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[27] Mesaac Makpangou, Guillaume Pierre, Christian Khoury, and Neilze Dorta. Replicated directory service for weakly consistent distributed caches. In *International Conference on Distributed Computing Systems*, pages 92–100, 1999.

[28] Radhika Malpani, Jacob Lorch, and David Berger. Making World Wide Web caching servers cooperate. In *4th international WWW conference*, Boston, MA, 1995.

[29] J. Moghul. Squeezing more bits out of http caches, 2000.

[30] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *SIGCOMM*, pages 181–194, 1997.

[31] E. Nahum, M. Rosu, S. Seshan, and J. Almeida. The effects of wide-area conditions on www server performance. In *ACM SIGMETRICS 2001 Conference*, 2001.

[32] M. Nottingham. On defining a role for demand-driven surrogate origin servers. In *5th International Web Caching Workshop and Content Delivery Workshop*, 2000.

[33] S. Paul and Z. Fei. Distributed caching and centralized control. In *5th International Web Caching Workshop and Content Delivery Workshop*, 2000.

[34] Michael Rabinovich, Jeff Chase, and Syam Gadde. Not all hits are created equal: cooperative proxy caching over a wide-area network. *Computer Networks and ISDN Systems*, 30(22–23):2253–2259, 1998.

[35] Mohammad S. Raunak, Prashant J. Shenoy, Pawan Goyal, and Krithi Ramamritham. Implications of proxy caching for provisioning networks and servers. In *Measurement and Modeling of Computer Systems*, pages 66–77, 2000.

[36] P. Rodriguez, E. Biersack, and K. Ross. Improving the www: Caching or multicast.

[37] P. Rodriguez, C. Spanner, and E. Biersack. Web caching architectures: Hierarchical and distributed caching, 1999.

[38] Alex Rousskov and Duane Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22–23):2155–2168, 1998.

[39] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining temporal coherency of virtual data warehouses, 1998.

[40] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet, 1999.

[41] B. Urgaonkar, A. Ninan, M. Raunak, P. Shenoy, and K. Ramamritham. Maintaining mutual consistency for cached web objects. In *ICDCS*, 2001.

[42] A. Venkatarammani, P. Yalagandula, R. Kokky, S. Sharif, and M. Dahlin. The potential costs and benefits of long-term prefetching for content distribution. In *6th International Web Caching Workshop and Content Delivery Workshop*, 2001.

[43] Z. Wang. Cachemesh: A distributed cache system for world wide web, 1997.

[44] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. the scale and performance of cooperative web proxy caching, 1999.

[45] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical cache consistency in a wan, 1999.

[46] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume leases to support consistency in large-scale systems, 1999.

[47] J. Yin, L. Alvisi, M. Dahlin, C. Lin, and A. Iyengar. Engineering server driven consistency for large scale dynamic web services. In *Tenth International World Wide Web Conference*, 2001.

[48] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture. In *SIGCOMM*, pages 163–174, 1999.

[49] Philip S. Yu and Edward A. MacNair. Performance study of a collaborative method for hierarchical caching in proxy servers. *Computer Networks and ISDN Systems*, 30(1–7):215–224, 1998.