Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web

Venkata C. Duvvuri

University of Massachusetts
June,1999

Masters Thesis
Project Report

Laboratory of Advanced Systems Software
Department of Computer Sciences
University of Massachusetts, Amherst

Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web

Approved by
Supervising Committee:

____

Prof. Prashant Shenoy

____

Prof. Krithi Ramamritham

To my Family

Acknowledgments

I would like to express my gratitude to Dr. Prashant Shenoy for his counsel and support throughout the duration of this work. I thank Dr. Krithi Ramamritham for his guidance and encouragement. Finally, I thank my family and friends for their support and understanding.

Venkata C. Duvvuri
University of Massachusetts
Amherst, MA
June, 1999

ABSTRACT

Currently, the internet supports weak cache consistency mechanisms. We argue that in order to support growing diversity in application requirements, the weak consistency mechanisms must be augmented by strong consistency mechanisms. Existing strong consistency mechanisms are not appealing for the web environments due to large state space or control message overhead. We present the leases approach that balances these tradeoffs and present the policies for determining the optimal lease duration. We present extensions to the http protocol to incorporate the leases and then implement our technique in the Squid Web proxy and Apache Web server. Our experimentation reveals that the overhead of leases implementation is comparable to existing weak consistency mechanisms.

Contents

List of Figures

List of Tables

1 Introduction

The World Wide Web (WWW) has observed an exponential growth in recent times. The growth is in the number of users as well as in the diversity of applications accessing information stored at geographically distributed sites. This growth, however, is not uniform. Certain objects are accessed more than others and create `hot spots` [1]. This leads to overload at the server, congestion in the network and increase in client response times. In addition, newer applications require smaller access latencies and stronger consistency guarantees. Consider the following illustration:

Consider a web server that provides on-line stock trading over the internet. Typically, the online traders require the latest stock quote of the stock which is under consideration. In addition, they want to know the latest news about the company, its quarterly earnings, charts on its performance in the last few months and other statistical information. However, the semantic requirements of the information varies. The traders will be able to tolerate small inconsistency in the statistical information (like number of employees) that is not critical in making a decision. However, the stock quotes, latest news and other critical information need to be consistent all the time. A trader should not be given a inconsistent stock quote and asked to decide whether to buy the stock. This example shows that applications require different consistency guarantees, often requiring these diverse guarantees to coexist.

Proxy caching has been the widely used to meet such demands. In such an architecture, clients requests objects from a proxy, proxy services the requests from local cache or fetches it from the server. This enables caching of objects near clients, thus improving the access latencies. It reduces the hot spots at the server and reduces the server load. It reduces the congestion at the network links to the server.

In order to prevent stale information from being transmitted to clients, a proxy must maintain consistency of cached objects with those on the servers Existing proxies mostly employ weak consistency mechanisms [2] [3]. That is, the proxy does not guarantee that the object served from the cache will always be consistent with the server at all times. This mechanism may be good enough for applications (like reading information about a company) that do not require strong consistency. Until recently, applications did not place stronger consistency requirements. However, with the evolution of the web, application like online trading and shopping are gaining dominance, and these impose strong consistency requirements. The current proxy consistency mechanisms provide no or little support for such applications.

In this report, we propose an efficient way to maintain strong consistency in the WWW. A prototype model of the algorithm has been developed and implemented in the current internet. The work enables coexistence of weak consistency mechanisms with those that provide strong consistency, thereby serving the diverse needs of applications.

The rest of the report is organized as follows - Section 2 looks at the existing proxy consistency mechanisms, Section 3 describes the adaptive lease mechanism, Section 4 presents various lease duration policies that we have developed, Section 5 describes the prototype implementation, Section 6 presents the experimental evaluation and, finally, section 7 presents our conclusion.

2 Cache Consistency in WWW

Cache consistency has been an active research problem in distributed shared memory, distributed databases, distributed file systems and the World Wide Web(WWW). However, the consistency problems in the WWW have some notable characteristics, that specialize the problem and make it different.

- Most of the WWW applications involve modifying the information non-concurrently(a web-master modifying the html page, a stock updater releasing the latest stock quote). Whereas, in distributed databases or file systems, it is not uncommon to have concurrent writes on an object.
- The amount of information available in the WWW is orders of magnitude more than traditional file systems or databases.
- The numbers of clients, servers, and proxies accessing this information is very large.
- The diverse WWW applications place different consistency demands on the information being served.
- Information accesses in WWW is non-uniform. Some objects become instantly popular and become so called `hot objects`. After some time,

the number of accesses die down and objects are said to be `cold`. Many objects change frequently and some are updated every minute [2].

The consistency mechanisms designed for distributed file systems and databases do not take into account these special characteristics, and hence are either unsuitable or inefficient in the WWW. The focus of this work is to develop a consistency mechanism that allows these diverse requirements to coexist

Cache consistency mechanisms can be classified to belong to two categories: weak consistency and strong consistency.

2.1 Weak Consistency Mechanisms

A *weak consistency mechanism* provides no guarantee that the object served from the cache will be consistent with that on the server. This means that `stale` objects can be served from the cache. Most existing proxies (like squid [3]) employ this form of consistency. Several weak consistency mechanisms have been proposed:

2.1.1 TTL mechanism

Here the server assigns each object with a timestamp that specifies the lifetime of the object. The time-stamp also called as time-to-live (TTL) is based on a heuristic guess. The proxies cache the object as long as the TTL value has not expired. When the TTL expires the proxy checks the consistency of the object with the server.

In this mechanism, the server makes a calculated guess on the lifetime of the object. However, there is no assurance that the object will not be modified before the TTL expires. The proxy may have the object with valid TTL and the server may modify it. In this case an inconsistent object will be served by the proxy. The main advantage of this method is the server need not maintain any state for an object.

2.1.2 Poll mechanism

In this mechanism, the proxy initiates a check(poll) to see if the object in its cache is consistent with the server [2] [3] [4]. The checks can be initiated periodically or based on some other factor like the age of the object, as in [4]. The proxy is thus said to *poll* the server.

In this mechanism the proxy guesses that the object may have been modified and decides to poll. If the object is modified in between polls, and a request is made after an object has been modified but before the poll, then the proxy serves inconsistent data. The advantage of this mechanisms is that no state is required to be maintained at the server nor any computation is needed at the server.

2.1.3 Hybrid mechanism

This mechanisms combines the TTL mechanisms with the poll. The proxy polls only after the TTL has expired. However, in this mechanisms though the proxy makes an educated guess on when to poll (based on a TTL), it still suffers from the same drawbacks of both mechanisms.

2.2 Strong Consistency Mechanisms

A *strong consistency mechanism* guarantees that the object served from the cache will always be consistent with those on the server. This means that stale objects can `never` be served from the cache. The strong consistency mechanism can be categorized as follows:

2.2.1 Server Invalidation

In this mechanism the server notifies the proxies whenever the object is modified [5]. Proxies are always aware whether the data in their cache is consistent with the server. Thus, the proxy never serves inconsistent objects.

The advantage of this mechanism is that it is optimal in the number of network messages needed to maintain consistency(a message is sent only when the object is being modified).

The disadvantage is that the server needs to maintain state for each object. It has to know the list of all proxies that have accessed the object in order to invalidate them. This state has to maintained indefinitely, since it is not known when the object may be modified. The resulting state space can be very huge and ever growing if we consider the millions of clients accessing the object. The second disadvantage is that the server may have to send many invalidations when the object is modified. This will cause a burst in the network traffic when the object is modified. If the proxies are unreachable due to network failure or a proxy crash, then the server must block on the write or risk violating consistency guarantees.

2.2.2 Client Poll

In this mechanism, the proxy polls the server on `every` object read. If the object has changed then, it is fetched from the server and the cache is updated. Thus, the proxy always serves consistent objects from its cache.

The advantage of this mechanism is that the server need not maintain any state. If the object is modified and the network between the proxy and server is broken, only the proxy needs to block on the read (unlike server invalidation). Thus, the writes at the server are independent of network failures.

The main disadvantage is that every read at the proxy adds a round trip delay. This will negate the benefit of caching data close to clients. In addition, the number of network messages sent to maintain consistency is much more than needed. Thus, there are lot of control messages are generated that waste network bandwidth.

3 Adaptive Leases

As seen in the previous section, server invalidation and client poll mechanism suffer from limitations that prevent efficient deployment of strong consistency mechanisms in the current WWW. Server invalidation has the problem of unbounded state space requirement at the server, while client poll results is a network message and round trip delay on every read.

Server invalidation and client poll form two ends of the spectrum. One is optimal in the number of network messages (equal to number of modifications), while the other is optimal in the amount of state space maintained at the server (no state space). Thus, there is a tradeoff between server state space vis a vis network bandwidth. As one moves from the client poll to server invalidation, the state space increases while the number of network messages and bandwidth requirement decreases.

3.1 Server granted leases

An alternate approach to guarantee strong consistency is server-granted leases. Here, the server assigns a lease on an object to every proxy. A *lease* on an object o is a tuple (s, d), where s is the start time and d is the expiry of the lease. The lease is defined as a two-way agreement between the client and the server:

- **server part** During the time interval of the lease $(s \leq t \leq d)$, the server agrees to notify the proxy of any modification made to the object.
- **client part** After the expiration of the lease $(t > d)$ or before the expiry $(t < s)$, the proxy polls the server on a read request. If the proxy is polling the server for the first time, or after the lease has been invalidated, it will issue a **grant** request. If the lease on o has expired then, it will issue a **renew** request.

This is a hybrid mechanism that employs both server invalidation and client poll to maintain strong consistency. It has the following advantages:

1. The state space maintained at the server is bounded. The server has to maintain state only for those proxies that have valid leases. After the expiration of the lease, the state can safely be discarded, as no invalidation needs to sent.
2. The client is not required to poll the server during the duration of the lease. It polls only when a read request arrives after the lease expires. Thus the number of network messages is reduced.

By appropriately determining the duration of the lease, this mechanism allows for a smooth tradeoff between the server state space and network messages. Smaller the lease duration, lesser the state space and more is the number of network messages. In contrast, larger the duration of the lease, larger the state space at the server, lesser the number of network messages. In fact, an infinite lease duration reduces to server invalidation and zero lease duration reduces to client poll mechanism.

The concept of leases was first introduced in the context of distributed file systems [6]. This mechanism focuses on granting leases of fixed duration. A recent project investigated grating leases to a collection of objects, so as to reduce the lease renewal overhead, and blocking overhead at server due to unreachable proxies[7]. Thus, to date most work has been focused on efficiently granting of fixed duration leases. The problem of determining optimal lease duration so as to balance state space overhead and number of message exchanged, has not received much attention, and is the focus of this work.

3.2 Lease Protocols

A lease is a two way agreement between the server and the proxy. As such, the proxies and server adhere to protocols to fulfill their part of the contract. This contract is specified in three protocols: proxy protocol, server protocol, and write protocol.

3.2.1 Proxy Protocol

This protocol ensures that the client never receives a stale object. Let client request object o from the proxy.

- If the lease on o is valid ($s \leq t \leq d$), then the proxy serves the object from its local cache.
- If the request is the first read on o, then the proxy issues a grant request to the server.
- If the request is the first read following an invalidation, then the proxy issues a grant request.
- If the lease on o has expired ($t > d$), then the proxy issues a renew request. The renew request is sent as an If-Modified-Request (IMS) request. If the response is a new lease with a Not-Modified (304) response, then the proxy can serve o from the local cache. If the response is a Http-Ok response (200), then the proxy stores the latest object piggybacked with the 304 response, and returns this object to the client.

3.2.2 Server Protocol

This protocol ensures that the server responds to grant and renew request from proxies. It also handles write requests by sending invalidation to the proxies.

- If the request is a grant request, then the server assign a new lease (based on lease duration policies). It sends o accompanied by a lease. It records in a state information maintained by it that the proxy has a lease on it.

- If the request is a renew request (IMS), then the server assign a new lease and updates it state. If o has been modified, it sends 200 response with new o and a lease, else, it just sends 304 response accompanied by a lease.

- If the request is an write request, then the server send invalidations to all valid proxies for o. It waits for an acknowledgement from each of the proxies, and then send an acknowledgement back to the writer. If the proxy has not responded it make a few retries until it receives an acknowledgement. A Not-acknowledgement (NOK) is sent if one or more proxies are unreachable.

3.2.3 Write Protocol

This allows a write to proceed after all the proxies have been invalidated. This protocol guarantees that the write happens only after all proxies have been invalidated. The writer issues a write request to the server. The server responds with an acknowledgement. If the acknowledgement is an OK, then the write may proceed. If the acknowledgment is a NOK, then the writer risks writing the object without guaranteeing strong consistency.

4 Lease Computation Policies

The crucial parameter that determines the efficiency of the leases algorithm is the lease duration d. By appropriately determining d, the server can balance the amount of state space overhead it needs to maintain and the number of control messages exchanged.

One of the main differences between traditional applications and Web applications is the access characteristics [8]. We propose various policies that take into account these characteristics and hence to cater to the specialized needs of WWW. The duration is calculated based on various object and system characteristics. These various policies seek to optimize different parameters and allow the server to adapt to changing conditions.

The lease duration policies are based on object lifetime, client access characteristics and server state space.

4.1 Age-Based Leases

This policy is motivated by bimodal nature of object lifetimes [2]. Here, majority of the updates go to a small fraction of (young) objects. Consequently, if we consider granting larger lease on younger objects, this will require invalidating many clients frequently. This will lead to frequent network traffic bursts and increased server load. In contrast, smaller leases to younger objects will result in reduced network traffic bursts with reduced server load.

So this policy assigns a lease duration based directly on the lifetime of the object. But, since an object lifetime cannot be known a-priori, we choose the age of the object as the reasonable predictor of its lifetime. Hence, the lease is computed as

$$d_i = \tau \cdot age_i \tag{1}$$

where age_i denotes the age of object i and τ is a threshold parameter. Since the policy assumes that larger the age of the object, longer is its lifetime, older objects are granted longer leases.

4.2 Renewal Frequency-Based Leases

This policy is motivated by non-uniform nature of accesses over the world wide web [2]. When a server grants a lease to a proxy, it commits its network and storage resources. A server could reduce it overhead by granting longer leases to proxies that have sustained interest in the object. Moreover, granting shorter leases to proxies that have only limited interest in the object reduces the state space overhead.

So this policy assigns the lease in direct proportion to the number of renewal messages sent by the proxy. Hence, the lease is computed as follows:

$$d_i = \tau \cdot renewal_i^p \tag{2}$$

where $renewal_i^p$ denotes the number of renewal messages sent by the proxy p for object i, and τ is a threshold parameter.

4.3 State Space Overhead-Based Leases

This policy is motivated by server space constraints. When a server grants a lease to a proxy, it has to maintain some state information about the proxy. More popular an object larger the state. If longer leases are granted, then state needed to maintain may exceed available limits. By granting shorter leases to popular objects the server can adaptively control the amount of state needed to maintain.

So this policy assigns a lease based on the number of valid leases granted for the object. Hence, the lease is computed as follows:

$$d_i = \frac{\tau}{l_i} \tag{3}$$

where l_i denotes the number of valid leases for object i and τ is a threshold parameter.

5 Prototype Implementation

A commercial prototype of the adaptive leases mechanism has been developed. The algorithm has been implemented in Squid Web Proxy (version 2.2 stable) and Apache Web Server (version 1.3.6). The Apache Web server has been augmented with a lease server that maintains state for all valid leases.

5.1 Lease-Control grammar

The protocols between the proxy and server have been completely implemented within HTTP/1.1. The interaction between the proxies and the server happen via HTTP request and response messages. HTTP/1.1[9] allows for extension of the entity header. A new `Lease-Control` header field has been introduced in the entity header. This header allows for lease information back and forth between the proxies and server. Shown below is the grammar for Lease-control header in BNF notation:

```
entity-header-extension         =  lease-control
lease-control                   = "Lease-Control" ":" lease-directive
lease-directive                 =  lease-request-directive    |
                                   lease-response-directive   |
                                   lease-invalidate-directive |
                                   lease-invalidate-ack-directive
lease-request-directive         = "Grant-Lease" |"Renew-Lease"
lease-response-directive        = "Lease" ":" lease-period | "Deny-Lease"
lease-invalidate-directive      = "Invalidate-Lease"
lease-invalidate-ack-directive  = "Invalidate-ack" ack
ack                             = "OK" | "Failed"
lease-period                    =  lease-start "-" lease-expires
lease-start                     =  HTTP-date
lease-expires                   =  HTTP-date
```

5.2 Implementation Architecture

The architecture for implementation is shown in figure 1. Clients make requests to the squid proxy. The proxy serves the object either from its local cache or may send HTTP/1.1 grant or renew requests (enhanced with `Lease-Control` header) to the server. Apache server recognizes the Lease-Control header and hands over the assignment of leases to lease server. When an object needs to be changed, the writer issues a write request to the server. The server recognizes the write request and hands over control to the lease server. The lease server invalidates all clients and sends an acknowledgment to the writer.

In our architecture, the lease server is an independent entity. This leads to a clean separation of functionality between the http server and the lease server. Moreover, the design allows for the lease server to be on a different machine. The apache server handles normal http processing and lease server handles the lease control functions. Specifically, it assigns new leases, renews leases, invalidates proxies, maintains state and updates state information.

5.3 Squid Web Proxy

Squid is a widely used proxy for web caching. The latest implementation Squid/2.2.STABLE2 [3] supports a weak consistency mechanism similar to

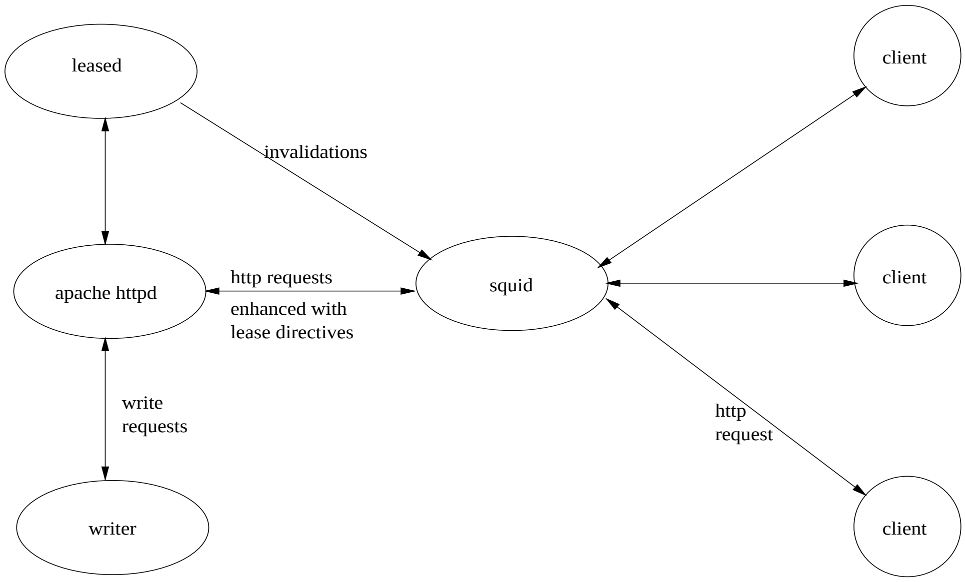

Figure 1: Implementation Architecture

TTL mechanism. In our **Squid/2.2.Lease** implementation, we augment this mechanism with the lease algorithm for strong consistency. Specifically, Squid/2.2.Lease supports following functionality:

1. Coexistence of stronger (Adaptive Leases) and weaker consistency (TTL based) mechanism. The mechanism can be configured in a configuration file. The configuration can be changed dynamically, thus allowing interchangeability of the stronger mechanism with the weaker one.

2. Issuing grant requests and storing lease expiry information in primary memory.

3. Issuing renew(IMS) request and updating lease expiry information.

4. Responding to invalidation requests by negating the object and the lease in the cache.

5.4 Apache Web Server

The apache server is the most common web server used throughout the WWW. It is a stateless server. It does not support any stronger consistency mechanism. The latest implementation, Apache/1.3.6 [10], assigns a TTL that may be used by a proxy to support weaker consistency.

Our implementation, **Apache/1.3.6.Lease**, supports the adaptive lease mechanism. To support huge magnitude of WWW clients, the architecture of the web server is divided into two sub-servers. The apache server handles normal http requests. The lease server handles lease assignment and state handling functionality. This allows for the lease server to be isolated and improve the performance of the web server on a whole.

The Apache/1.3.6.Lease server supports the following functionality:

1. Coexistence of normal http requests with Lease-Control enhanced requests that support strong consistency.
2. Identifying grant request and contacting the lease server to obtain a lease. The lease is then appended in the response headers.
3. Identifying renew request and contacting the lease server to renew the lease. The lease is then appended in the response headers.
4. Identifying write requests and delegating the invalidations to the lease server.

Apache/1.3.6.Lease allows for the address of lease server to be configured in a configuration file.

5.5 Lease server

The lease server is an independent server that handles all state information of the web server. Our implementation **Lease/1.1** supports the following functionality:

1. Creation and maintainence of state information for all objects on the server. This state information is stored in the primary memory to facilitate faster grants, renews and invalidations.

2. Granting leases based on lease duration policies. The following three adaptive lease duration policies are supported:

   - age-based leases
   - renewal frequency-based leases
   - fixed leases

   The policies can be defined in a configuration file. The configuration also allows for specification of the threshold parameter associated with individual policies.

3. Renewing leases based on lease duration policies.

4. Invalidating all proxies of an object. Invalidations are sent is parallel (to a max of five concurrent invalidation) to facilitate faster invalidations. The lease server associates each invalidation with a timeout that can be configured. A failed invalidation (timeout) is re-tried a configurable number of times. A successful invalidation is acknowledged by sending an OK response to the writer. A failure results in a FALIED response to the writer.

6 Experimentation

We have evaluated the efficacy of the prototype implementation. First, we measured the overheads incurred during grant, renew and invalidation request. Next, we have replayed a 2 month trace (BU traces) and measured the overheads again. Our results demonstrate that the overheads in grants, renewals and invalidations are comparable with existing mechanisms.

The testbed for our micro-benchmarks consisted of the web server (httpd and leased), the Squid web proxy cache and the client running on a cluster of PC-based workstations. Each PC used in our experiments is a 350 MHz Pentium II with 64 MB RAM and runs RedHat Linux 5.1. All the machines were interconnected by a 10Mb/s Ethernet.

6.1 Overhead measurement

This experiment consisted of a client that requested a 1KB file first from an unmodified Squid/Apache combination and then from our prototype imple-

	Client Response Time	Lease Server Overhead
Unmodified	76.7 ms	-
Grant Lease	86.6 ms	3.77 ms
Renew Lease	112.3 ms	4.31 ms
TTL (weak)	112.0 ms	-

Table 1: Overhead for Granting and Renewing Lease

mentation. The experiment was repeated 2000 times. Table 1 lists the client response time and the overhead of the lease server.

6.1.1 Grant/Renew Overhead

The results show that the overhead of granting and renewing leases in the lease server is only 4 ms in Lease/1.1. Granting leases results in a modest increase of 12% in the client response time. Renewing leases result in a negligible increase in client response time when compared to a TTL refresh based weak consistency algorithm. The increase in client response time for renew/TTL as to the unmodified Squid/Apache is due to the extra refresh check computation that Squid perform when a lease/TTL already exists on an object.

6.1.2 Invalidation Overhead

This experiment measures the overhead for invalidations, at the client and the lease server. The graph 2 plots variations in invalidation overhead with increasing number of leases held for an object. Since, the lease must send invalidations to each proxy holding the lease the overhead increases slowly with the number of active leases. This represents a worst case invalidation time as the lease server does no parallelization to invalidate.

6.2 Real Trace experiment

This experiment replays a 42-day-Boston-University trace. The original was preprocessed and then used for the experiment.

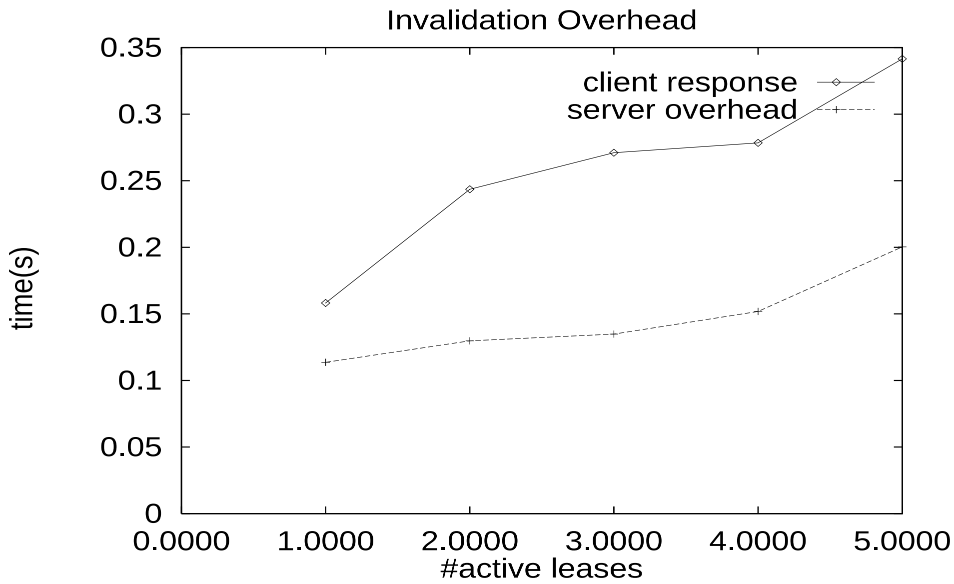

Figure 2: Overhead of Invalidations

6.2.1 Preprocessing

Dynamically generated objects (such as cgi,?) need not be cached and hence have been removed from the trace. Then the requests corresponding to the most popular server were extracted.

The original trace included last modified information only when available. Since, last modified values are important for evaluating cache consistency model, an empirically derived model to generate synthetic writes requests was employed. Based on observation in [2], we assumed that 90% of all objects change very infrequently (i.e. have average lifetime of 60 days). We assume that 7% of the objects are mutable (i.e. have average lifetime of 20days) and the remaining 3% are very mutable (i.e. have average lifetime of 5 days). Using this distribution synthetic write trace for the objects were obtained. The pre-processed trace characteristics is shown in Table 2.

The traces timestamps were scaled by a factor of 100 to allow for completion of the experiment within suitable time. The traces were divided into 3 sets, and fed to three clients on independent machines. The write traces were fed to an invalidator client program that ran on a different machine. The apache web server and the lease server ran on the same machine.

Request	Duration(s)	Unique URL's	Number of Synthetic writes
129561	3656817	1363	817

Table 2: BU trace characteristics

The lease server was configured to adopt the fixed based lease policy with leases of 1) 15 minutes 2) 1 day. A lease duration of 1 day would be equivalent to server invalidation. Hence, this experiment compares the performance of leases algorithm with that of server invalidation.

6.2.2 Control Message Overhead

The results for the number of control messages exchanged between the proxy and the server are shown in Table 3. Server invalidation requires approximately 6% more grant requests. Grant requests are dependent on the number of invalidations, which in turn depend on the lease duration. More the number of invalidations, more will be the number of grants. For leases of a day the are no renews(lease never expires).

The number of active invalidations represents those invalidations for which the lease was valid and hence results in a network message. This depends on the lease duration. Server invalidation requires the maximum number of active invalidations. As the lease duration decreases the number of active invalidations falls.

The total number of control messages for leases is more than server invalidation. This is traded off with the server state space requirements as will be evident in the following section.

6.2.3 Response Times

The response times are shown in Table 4. The client response time has increased from one in Table 1 because of the total size of objects in this experiment is approximately 5 MB as compared to 1 MB in the previous experiment. The other other response times are similar to those in previous experiment.

	15 minutes lease	1 day lease
No. of grants	1426	1488
No. of renews	3304	0
No of active Invalid.	72	125
Total Control Msgs.	4802	1613

Table 3: Control Message Overhead

	15 minutes lease	1 day lease
Avg. Client response	107.5 ms	107.8 ms
Server grant time	4.75 ms	4.8 ms
Server renew time	4.95 ms	0
Client inv. response	173.5 ms	142.97

Table 4: Response Time

6.2.4 Server State Space Overhead

The server space measurements have been made by periodically (900 s) updating the server state and freeing state for proxies whose lease has expired. The plot in Figure 3 shown the variation in the state space (measured by the he number of valid leases) for the whole duration of the experiment. The plot for lease 1 day slowly rises to the the maximum space (1363) as state information is retained indefinitely. But, in case of leases the plot depends on the lease duration as well as the request pattern. The maximum state information for a 15 minute lease that the lease server has to store is 281 (one fifth of server invalidation). This shows that a 15 minute lease tradeoffs server space with number of control messages.

These experiments ascertain the hypothesis that the leases algorithm represent a tradeoff in the number of control messages with the server space.

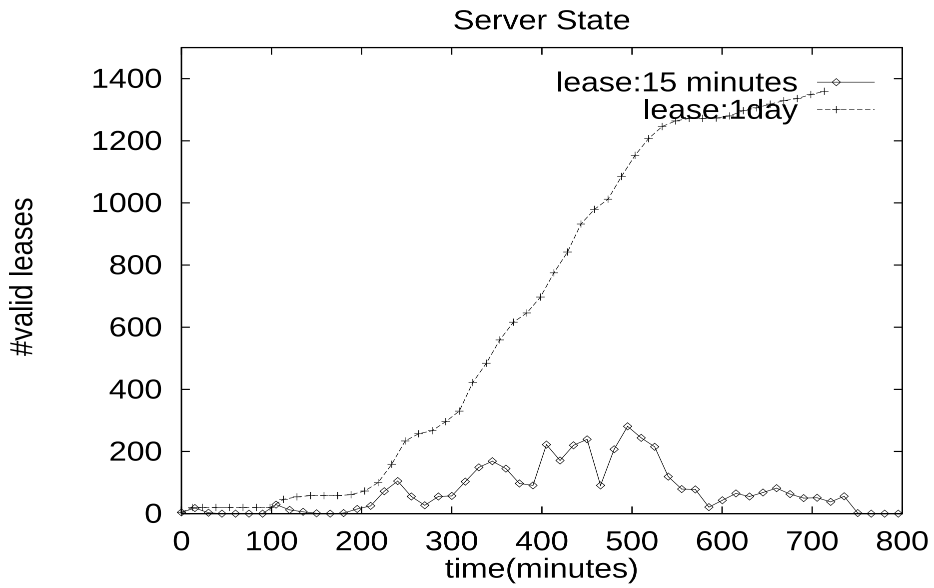

Figure 3: Server State space overhead

7 Conclusion

WWW applications place diverse consistency requirements on proxies. This demands for the existence of stronger consistency mechanisms with weaker ones. Currently, the proxies in the WWW offer only weaker consistency mechanisms. Existing strong consistency mechanism are not appealing for the web environment due to large state space and control message overhead.

We have developed a new approach for guaranteeing strong consistency based on leases, that balances these tradeoffs. We present adaptive policies for determining optimal lease duration. A prototype has been implemented that allows the coexistence of strong and weak consistency mechanisms. The Squid Web proxy and the Apache Web server have been modified, tested and deployed with the leases and TTL mechanisms. Our experimental results ascertain our hypothesis, that the leases algorithm represents a suitable tradeoff between server state space and network messages.

References

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

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

[3] *Squid Internet Object Cache User Guide*. Available on-line at http://squid.nlanr.net, 1997.

[4] V. Cate. Alex: A global File Systems. In *Proceedings of 1992 USENIX File System Workshop*, pages 1-12, May 1992.

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

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

[7] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases for Consistency in Large-Scale Systems. In *IEEE Transactions on Knowledge and Data Engineering*, Jan 1999.

[8] P. Bartford, A. Bestavros, A. Bradley, and M.E.Crovella. Changes in Web Client Access patterns: Characteristics and Caching implications. In *World Wide Web Journal(to appear)*, 1999

[9] JR. Fieldings, J. Gettyps. J.C. Mogul. H. Frystyk, and T. Berners-Lee, In *Hypertext Transfer Protocol- HTTP/1.1*, Internet Draft. Sept. 1998

[10] *Apache web server*, available on-line at http://www.apache.org

VITA

Venkata Duvvuri was born in Hyderabad, India on Nov 12, 1975, the son of Ramani and Krishnamurthy Duvvuri. After completing high school at D. G. Ruparel college, Bombay in 1991 he entered Victoria Jubilee Technical Institute(V.J.T.I, University of Bombay), Bombay from where he received a Bachelors of Engineering(Distinction) in Electronics Engineering in May 1997. In August 1997, he entered the Graduate School at the University of Massachusetts, Amherst. He will be joining Information Technology department of Goldman Sachs & Co. from August 1999.

Permanent Address:

307/A, Anand Sagar Society,
M.D.Marg,
Thane, 400602
India