The Distributed Object Consistency Protocol Version 1.0

John Dilley, Martin Arlitt, Stephane Perret, Tai Jin
Internet Systems and Applications Laboratory
HP Laboratories Palo Alto
HPL-1999-109
September, 1999

World Wide Web, cache consistency, cache invalidation, publish/subscribe protocol, proxy cache, HTTP cache control, strong consistency, delta consistency

This report describes a protocol for improving content consistency in web proxy cache servers, which leads to reduced response time and server load. The Distributed Object Consistency Protocol (DOCP) is an extension to HTTP, replacing some of HTTP's current cache control mechanism. It supports incremental evolution: proxy servers that implement the protocol can interoperate with non-DOCP proxy servers and gradually learn about peers as they are deployed in the network.

The DOCP uses a publish/subscribe mechanism with server invalidation when objects change instead of client validation to test for changes. Stronger semantic consistency assures content providers that pages served from cache are the same as are on the master origin server, and assures end users that the information they are getting is fresh, limiting the need to "Reload".

Strong consistency is difficult to achieve in wide area networks, so the DOCP provides "delta consistency", where an object is consistent for all but a short, bounded time after a modification. This allows the DOCP to meet web user expectations for content availability and responsiveness.

Protocol simulation shows that DOCP consumes fewer network and server resources than the current HTTP cache consistency protocol does, while providing greatly improved consistency. DOCP eliminates the need for users and content providers to guess at correct values for HTTP cache control headers by providing a clean framework for wide area web object replication.

The Distributed Object Consistency Protocol
Version 1.0

John Dilley, Martin Arlitt, Stéphane Perret, Tai Jin

Hewlett-Packard Laboratories
Palo Alto, CA

Abstract

This report describes a protocol for improving content consistency in web proxy cache servers, which leads to reduced response time and server load. The Distributed Object Consistency Protocol (DOCP) is an extension to HTTP, replacing some of HTTP's current cache control mechanism. It supports incremental evolution: proxy servers that implement the protocol can interoperate with non-DOCP proxy servers and gradually learn about peers as they are deployed in the network.

The DOCP uses a publish/subscribe mechanism with server invalidation when objects change instead of client validation to test for changes. Stronger semantic consistency assures content providers that pages served from cache are the same as are on the master origin server, and assures end users that the information they are getting is fresh, limiting the need to "Reload".

Strong consistency is difficult to achieve in wide area networks, so the DOCP provides "delta consistency", where an object is consistent for all but a short, bounded time after a modification. This allows the DOCP to meet web user expectations for content availability and responsiveness.

Protocol simulation shows that DOCP consumes fewer network and server resources than the current HTTP cache consistency protocol does, while providing greatly improved consistency. DOCP eliminates the need for users and content providers to guess at correct values for HTTP cache control headers by providing a clean framework for wide area web object replication.

1 Introduction

To improve service to web clients, web proxy cache servers have been deployed throughout the network. These cache servers store copies of objects requested by web users and subsequently serve that object to those users if requested again. By serving cached objects, the proxy reduces network and origin server demand. However, the objects they serve are not necessarily current with the origin server. This *weak consistency* leads content providers to disable caching for some objects and forces users to reload pages when they suspect inconsistency. This increases server load, especially during *flash crowds* when many users visit a site at the same time.

Cache servers can not know whether an object in cache is consistent without querying the origin server. Checking consistency every time assures consistency at the cost of additional connections to origin servers, which adds considerable delay to the servicing of user requests [8]. Use of polling to achieve strong consistency also increases demand on the network and origin server, which reduces the benefit of caching.

Weak consistency is currently accepted for most applications of the web. Client-driven polling is the only current alternative to provide strong consistency in the web. When consistency is needed, origin servers must serve every request, which requires sufficient hardware and network resources. However this scheme does not scale well; implementing strong consistency by polling is too expensive for all but a few content providers. Furthermore these solutions can not improve response time.

Stronger consistency is important for business use of the web such as for interaction with customers, viewing of supplies from vendors, and access to time-dependent information. Greater commercial use of web technology will lead to increased expectations of performance and predictability. The Distributed Object Consistency Protocol is designed to address these needs.

The remainder of the paper is organized as follows. Section 2 presents an overview of cache consistency and some issues with today's web cache implementation. Section 3 discusses requirements for a wide area web consistency protocol. Section 4 presents an overview of the DOCP consistency model and Section 5 describes the protocol in detail. Section 6 discusses the interaction between DOCP and HTTP cache control headers. Section 7 analyzes the protocol's performance and summarizes the results of a simulation of the protocol. Section 8 discusses related work. Section 9 concludes with some observations and suggestions for future work.

2 Web Cache Consistency

Caching enhances system scalability and performance, particularly where remote communication has higher cost (in latency or resource utilization) than local communication with the cache. With caching or replication comes the issue of maintaining consistency or coherence among the copies. Caches can be characterized in terms of the coherence they provide to their users. In general, the weaker the consistency requirement, the easier it is to build a scalable distributed system.

2.1 Strong Consistency Models

Under a strong consistency model a cache guarantees not to serve data that is different from a request for the original data to the origin server. This requirement is frequently seen in operating systems (such as in a memory or disk cache) and transaction-processing systems. Mosberger [22] provides an overview of the following types of strong consistency.

- *Atomic*. Any read on a shared item returns the most recent write of that item. In a distributed system each participant has an independent notion of event order; there is no global "most recent". Atomic action is only defined locally.
- *Sequential*. The result of any set of operations is the same as if the operations were executed in some sequential order [16]. In other words, there is some total ordering for all operations taken together. Any execution must obey this total ordering to be sequential.
- *Causal*. Writes that are potentially causally related must be seen by all processes in the same order, but concurrent writes that are not related may be seen in a different order by different processes [15].
- *Pipelined RAM* (PRAM). Writes by a single process are received by all other processes in the order they were issued, but writes from different processes may be seen in a different order by different processes [18].

Strong consistency in a distributed system can be client-driven or server-driven: either clients query the server to validate the object on each request, or servers invalidate cached copies when an object changes.

In the client-driven case the load on the origin server can be nearly as high as it would be without caching since the server has to handle each request (perhaps not with full object data). If communication latency between the client and server is great, the cache can not reduce response time significantly.

In the server-driven case the server must maintain state about all cached replicas, and notify each replica before a change to the data is committed. In general, read access is not permitted while the data is changing (after invalidations begin to be delivered but before they are acknowledged by all replicas). A write can not complete until all replicas have been notified to flush the previous copy of the object. This will prevent a replica serving inconsistent data relative to its peers, and supports sequential consistency. In wide area distributed systems, communication delay between peers can be quite high, so this protocol may require a long time to complete.

In any strong consistency implementation, network or replica delay or failure will either delay or prevent read and write requests from completing.

2.2 Weak Consistency Models

Relaxing the consistency requirement can improve the availability, fault resilience, and performance of widely distributed systems, such as the Grapevine [5] and LOCUS [25] distributed systems; [28] discusses weak consistency and suggests a mechanism to provide session guarantees to reduce the potential confusion caused by weak consistency on users and applications.

To describe web cache consistency we classify weak consistency models for information access as follows.

- *Delta consistency.* The result of any read operation is consistent with a read on the original copy of an object except for a (short) bounded interval after a modification.
- *Eventual consistency.* The result of any read operation is consistent with the last known update, but the propagation time of updates is not prescribed. Information about updates is distributed to replica sites and if updates cease replicas will eventually become consistent. This is the model used in Grapevine [5] by the Clearinghouse directory service.
- *Loose*. There is no guarantee about consistency of any read with the original copy of an object, but there are systematic (possibly manually invoked) mechanisms to validate and restore consistency.
- *Periodic*. Replicas are periodically synchronized with their original copy, after which they may again diverge. There is no systematic mechanism to validate object consistency on demand.
- *Offline*. The state of the original copy of an object can not be determined on demand.

Most current web cache implementations provide loose consistency. They serve object data for some period of time without contacting (or being contacted by) the origin server. Loose consistency is inherent in the HTTP protocol: there is no protocol mechanism to assure stronger consistency other than to prevent caching of objects. HTTP does provide for validation and most caches implement occasional validation of object consistency. One widely used mechanism is described in the next section.

2.3 The Alex Protocol

Avoiding contact with the origin server improves user response time [8] and reduces network utilization. A cache does this by determining a time to live (TTL) value for each object in the cache. During the TTL period the cache will serve the object locally. The TTL value could be chosen as some constant value, however web objects have widely varying lifetimes and modification rates. The Alex file system protocol [6] developed an *adaptive TTL* mechanism to address this need. The Harvest cache [7] employed this mechanism, which is fairly widely used by Harvest-derived caches, including the Squid open-source proxy cache [27]. We use the Squid implementation to illustrate the protocol.

In the Squid adaptive TTL implementation, the TTL is determined to be the time indicated in the **`Expires`** header, if one is present; if not it uses the time in the **`Last-Modified`** header, computes a percentage of the age of the object in the cache at each new request and uses that value as the TTL ([8] has an illustration and description of this protocol). After the TTL period expires the cache will make an HTTP **`GET`** request with an **`If-Modified-Since`** (IMS) header field indicating the last modification time of the object as previously reported by the origin server. If the object has changed the origin server will respond with a fresh copy of the object and the HTTP **`200 OK`** response status; otherwise it will return the response status **`304 Not Modified`**, indicating that the cached copy is still current.

In most cases these **`GET IMS`** requests are unnecessary, since web objects change much less frequently than they are accessed. In a study of five months of web access from a group of users with cable modems we observed 15% of total requests resulted in **`304 Not Modified`** responses [2]. In another study of the World Cup web site we observed as high as 37% of responses from the server were these "positive validations" [4]. These percentages are of total requests, including first time requests and requests for objects evicted from caches. We can not determine how many requests were **`GET IMS`** requests from the logs.

Eliminating these **`GET IMS`** requests would seriously compromise object consistency. It is not possible to assess how many stale objects were served by caches; that requires global state, which is not available. The rate and distribution of object modification and object accesses both affect the stale hit ratio.

The TTL mechanism creates a trade-off between object consistency on the one hand, and network utilization and response time on the other. By choosing a large TTL value, a cache will make fewer external requests and service data more quickly, but it will serve more inconsistent copies of information to end users. By choosing a small value, the cache will make more unnecessary **`GET IMS`** requests but achieve higher consistency. Cache administrators are sometimes able to make this trade-off through cache configuration based upon local system and user requirements. Note that this decision may not reflect content provider or user preferences.

3 DOCP Requirements

The broad goals of the DOCP are to improve the user response time, predictability, accountability, scalability, and fault tolerance of HTTP access to web objects. This section presents some key requirements for a protocol designed to meet these goals.

3.1 Incremental Evolution

Servers in the Internet are not owned by any central organization nor do they obey any common set of rules. Proxy cache servers can come and go at any time; origin servers may be here one day but gone the next. Any system to improve web consistency and performance cannot alter the fundamental independent ownership and operation model inherent in the Internet. The system must work with, not against, Internet philosophy. The protocol must be able to be run between independently owned and operated caches and origin servers.

The web is a large-scale, widely deployed distributed system. Such systems are impossible to update in a synchronized manner due to their size and complexity, not to mention that servers are independently owned and managed. The consistency protocol must be an incremental addition to the system. Implementations of cache servers running the new protocol must work with origin servers and caches that do not support it. They should automatically discover the protocol capability in their peers if possible, and use it where possible.

3.2 Internet Scale

A user may access dozens to hundreds of individual origin servers and hundreds to thousands of objects in a day. Web servers may receive requests from thousands to millions of users in a day. Proxy servers are in between, often serving thousands of users who may access millions of objects on thousands of origin servers throughout the Internet.

The protocol should reduce network traffic and load on origin servers; implementations must be able to handle current web workloads better than the current protocol can; and should demonstrate the potential of handling future workloads better than the current protocol.

3.3 Fault Tolerance

Any large-scale, wide-area distributed system must accommodate component or infrastructure failure, because failures *will* happen. Since any component of the system may fail or disappear from the network, the parties sharing system state must be able to recover from such failure. When failures occur there should be minimal disruption to the operation of the non-failing parts of the network.

3.4 Web Consistency and Performance

The protocol should support stronger consistency than current cache servers can provide and improve user performance. Objects in cache should be more highly available than in today's caches and should be served more quickly to end users. Communication between a cache and origin server that is synchronous to user requests should be eliminated where possible.

The protocol should support high service quality, in particular low delay since the object is close, and low delay variance since the end network is usually more predictable than the wide area Internet.

3.5 Accounting and Content Distribution

Most cache solutions mask user requests served by caches from content providers. When content providers want control for consistency or accounting reasons they

have few options. They can attach HTTP cache control headers, such as the **`Expires`** header, so user requests will be sent to the origin server each time. Attaching such headers is generally referred to as *cache-busting*.

To limit cache-busting, caches should assure content providers that their content is consistent with what is currently published, and should deliver access accounting information to the provider.

Proxies that can serve objects authoritatively at the edge of the network must provide accounting information about user requests to the content provider. They also allow more flexible content distribution and hosting across wide area networks.

4 DOCP Overview

The DOCP improves consistency and accountability through a publish/subscribe mechanism in conjunction with soft state and other optimizations. This helps to accommodate the scale of the Internet and to limit the damage caused by unanticipated failures or changes in network configuration. DOCP is similar to the HTTP cache control mechanism, and allows incremental evolution of that mechanism. The DOCP relies on information push between cooperating proxies to achieve better performance when objects change.

The protocol operates between a publisher (master) and subscriber (slave). Since most web objects are requested only once [2], the slave will only attempt to subscribe to an object that has been requested before. If the slave is granted a subscription it does not have to validate the object's freshness with the master, but can serve that object directly from cache. The master will send a notification to the slave if the object changes. If the slave gets another request for that object it will request a fresh copy of the object (and maybe a renewed subscription) from the master. This provides improved consistency of content; in a simulation of over 40 M user requests not a single object was served inconsistently.

Objects can be subscribed to only after they have been re-referenced without an intervening modification. This limits subscriptions to uninteresting or very rapidly changing objects.

Serving objects directly from cache speeds the delivery of web objects to end users and reduces the demand on origin servers and networks. In addition to the stronger consistency, content is served faster and cheaper.

Strong consistency is difficult to achieve in large, widely distributed systems, due to latency, coordination and object availability requirements. Instead of strong consistency the protocol allows *delta consistency*, where each object returned by the cache is consistent to within some bounded time, delta, since its last modification. The period of potential inconsistency is roughly the propagation time of a notification from the server to the cache after an object is modified. Note that this period is on the same order as the propagation time of a **`GET IMS`** response from a server to a cache, currently used to check object consistency. Broadly speaking this protocol replaces several **`GET IMS`** requests with zero or one invalidation message (one if there was a change; otherwise no communication).

The DOCP defines a new set of HTTP headers to provide object consistency between content origin servers and edge proxy cache servers. The DOCP distributes the ability to serve objects authoritatively on behalf of a content provider throughout the network. The key components and roles in the DOCP architecture are depicted in Figure 1 and described in the sections below.

4.1 Web Objects

Web objects are identified using a Uniform Resource Locator (URL, defined in RFC 1738) and accessed using HTTP (RFC 2616 [9]), usually with a web browser application, which may use a web proxy cache. The DOCP does not change any of these definitions.

4.2 Content Provider

A content provider develops objects to be published on the web. A content provider should be able to use their choice of web publishing and preparation tools, and publish to any supported web server platform.

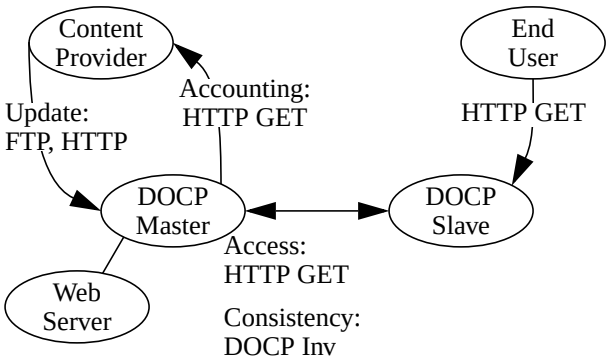
A content provider must provide the DOCP service with object modification notification. This is done via a mechanism external to the DOCP. The DOCP makes access accounting information available to the content provider as described in Section 5.9.

4.3 DOCP Master

A DOCP master serves content for one or more content providers. It behaves like a *reverse proxy server*: it responds to HTTP requests for the content provider's objects. If the content is not currently in its cache the DOCP master accesses it from an origin web server affiliated with the content provider.

The DOCP master serves objects as well as subscription requests for the objects. When a content provider publishes new content a consistency manager component detects object modifications and delivers invalidation messages to the subscribed DOCP slaves, as described in Section 5.7.

FIGURE 1. DOCP Architecture



4.4 Web Server

The DOCP master communicates with content on a web server. This allows a content provider to select the publishing tools they wish, while improving the content service. The DOCP master accesses content on the web server via HTTP, caching a copy of the response. All new information published to the web server must pass through the DOCP master so that it can generate change notifications to subscribed slaves.

4.5 DOCP Slave

A DOCP slave serves requests for one or more clients. It behaves like a regular web proxy cache: it accepts HTTP **`GET`** requests, checks its cache for a current copy of a requested object. If it has a current copy that is returned to the client; otherwise the DOCP slave makes an HTTP **`GET`** or **`GET IMS`** request to retrieve or validate the object. If the object is served by a DOCP master, the slave may also request to subscribe to the object.

DOCP proxies can be arranged in a hierarchy, and may be a master for some slaves and a slave to others. The master and slave terms describe roles in the protocol.

4.6 Client

Clients issue HTTP requests to DOCP proxies as usual. The fact that they are using a DOCP enabled cache is transparent to them except for improved consistency and performance. No modification is required to browsers.

5 DOCP Protocol Operation

This section presents the operation of the Distributed Object Consistency Protocol.

5.1 Protocol Overview

The DOCP uses a publish/subscribe replication model for popular web objects. When an object becomes sufficiently popular in a DOCP slave cache the slave will request to subscribe to that object with the origin server's DOCP subscription manager. If the object changes during the subscription interval assigned by the subscription manager, the DOCP master will send a notification of the change to all current subscribers. Until then a slave can serve subscribed objects authoritatively without contacting the origin server.

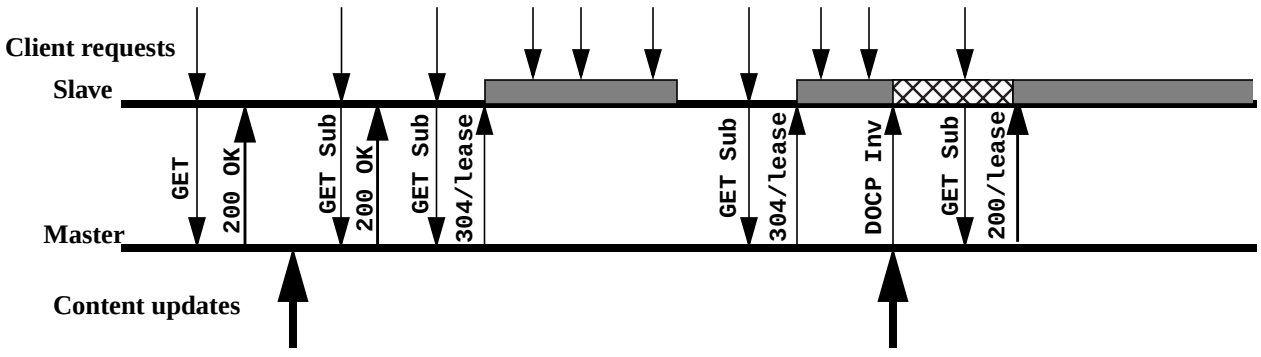
Figure 2illustrates the operation of the protocol. Client requests for an object arrive from the top to be served by a DOCP slave. On the first request for the object the slave makes a **`GET`** request to load the object into its cache. On subsequent client requests the slave makes a **`GET IMS Sub`** request to determine if the object has been modified and request to subscribe to it. The DOCP slave guarantees that every object it serves is consistent with what the content provider has published. The **`GET IMS Sub`** request can return a fresh copy of the object (and HTTP status **`200 OK`**), or a positive validation (HTTP status **`304 Not Modified`**) with no object data.

The slave requests to subscribe to the object by including a new HTTP **`DOCP-Subscribe`** header with the request on the second and subsequent requests for an object. It will not request to subscribe to an object retrieved only once. In earlier workload characterization [2] we found 60% of objects were requested only once. There is no value to subscribing to these objects.

If the object was modified, as in the first **`GET Sub`** request, a fresh version of the object is returned but no subscription lease. If the object has not been modified since the prior access, as in the second **`GET Sub`** request, the master grants a subscription lease. During the lease interval (shaded intervals in Figure 2). the slave serves client requests directly without contacting the master. The lease can either expire, as in the first lease in the figure, or be terminated through invalidation, as in the second lease in the figure. On the next request after a lease expires or is invalidated, the slave must contact the master. The slave will refresh the object and request a fresh lease for the object. If it was modified only once, as depicted in the figure, the subscription request will be granted. If it had been modified again before the final **`GET Sub`** depicted in the figure the lease would not have been renewed. A client must hold an unmodified object to receive a lease. This limits the need for masters to send invalidations for frequently changing objects.

The following sections describe the subscription decision and some high level details of the protocol. Appendix A contains further detail about the protocol including the syntax of the HTTP protocol extensions and the specific responses from master to slave.

FIGURE 2. DOCP Consistency Protocol Operation



5.2 Subscription Decision - Slave

The request for a subscription is a local decision made by a DOCP slave. The slave does not need to coordinate with the DOCP master or any other client to make a subscription decision.

The slave may use the object's local popularity (reference count), size, last modification time and last access time to make a decision whether to request to subscribe to the object. To request a subscription, the slave makes an HTTP **`GET`** request with an extra header field, **`DOCP-Subscribe`**.

The slave must include an **`If-Modified-Since`** header indicating the slave's last known modification time of the object. A slave should not request validation with an **`IMS`** request without requesting a subscription, since a subscription is more efficient than polling for the slave and the network.

5.3 Subscription Decision - Master

A subscription request must be acknowledged by the DOCP master in its HTTP reply. The master decides whether to grant a subscription based upon local policy, which may include an estimate of the object's global popularity, its size, modification history, and the number of existing subscriptions to that object.

The master's HTTP reply indicates if the subscription was granted, and if so for how long, using a **`DOCP-Lease`** response header. The choices of the lease value are described in Appendix A, in particular in Table 2, "DOCP-Lease Header Values," on page 19.

Upon granting a subscription, the DOCP master must record the subscribing slave's return (notification) address in case the object changes.

5.4 Subscription Decision - Lease Interval

Each subscription is bounded by a lease interval calculated by the master. Each object may have zero or one lease associated with it; all subscriptions to a single object share the same lease value (expiration time) at the DOCP master. This prevents the need for the master to maintain different expiration times for different slaves.

If an object is not modified when its lease expires, the master simply clears the subscriber list. No communication takes place between master and slaves.

The lease provides a simple, robust method for limiting the amount of state that must be kept by the master. It provides a network-efficient mechanism for subscriber list clean-up, since no communication is required to release a lease (no unsubscribe exchange is required between master and slave). The lease also provides a bound on the amount of time a master will attempt to deliver an invalidation to a slave that is unreachable, and therefore the maximum period of object inconsistency. A lease interval will typically be days to weeks long.

The master should set leases to expire at quiet periods in the day if possible. This helps to avoid bursts of requests when object leases expire. At most sites the pre-dawn period between 2-5 AM receives the fewest requests per hour. Frequently changing content may require shorter lease intervals. Lease expiration times should be distributed (for example randomly) during an interval to avoid may object leases expiring together.

5.5 Leases and Clock Skew

Clock synchronization in wide area distributed systems has long been studied [15]. Protocols exist to synchronize time across the Internet [20][26]. However, clocks may still not be synchronized for many reasons. The DOCP must be resilient to unsynchronized clocks.

A DOCP master accommodates this by returning a lease as an absolute time in the slave's time domain as close as possible to the expiration time in the master's domain. To support this the slave must send the master its notion of the current time. The master assigns the lease expiration time for every object. The master uses the expiration time and the slave's current time to compute the lease expiration time at the slave. The slave expiration time is set to expire at or before the master lease expiration time. This prevents slaves serving documents that have expired at the master.

5.6 Modification of Subscribed Objects

If an object is modified during the lease interval the DOCP master must attempt to notify any DOCP slaves currently subscribed to the object that it has changed. At that time the lease is canceled and no further invalidation messages will be sent for that object unless a new subscription is begun. Note that the cost to the network and origin server of delivering *n* invalidation messages for a changed object is approximately the same as the cost of serving one **`IMS`** request from *n* proxies checking the freshness of that object, or *n* **`IMS`** requests from one proxy.

5.7 Change Notification

When a change to a subscribed object is detected the DOCP master will transfer the current subscription list to a notification agent. This transfer resets the subscription list for that object so no further invalidation messages will be sent upon subsequent modifications, unless a DOCP slave re-subscribes to (the new version of) the object.

The notification agent attempts to deliver the change notification to each slave indicated on the subscription list. Each notification must be acknowledged by the slave. If delivery fails (is not acknowledged) the agent will attempt to re-deliver the notification after an initial timeout interval. The interval between retransmissions should employ a backoff mechanism if delivery continues to fail. The notification agent must stop attempting to deliver the notification when the lease expires.

Delivery is accomplished using one of two protocols, under the DOCP notification agent's control (it will attempt both if the first choice fails):

- DOCP/UDP. The DOCP master sends a notification message to the slave's UDP notification port. There is an issue with such UDP traffic traversing firewalls, so other alternatives must also be supported.
- HTTP/TCP. The DOCP master makes an HTTP **`POST`** request to the DOCP slave's HTTP notification port. The HTTP channel should be persistent while the slave and master are actively communicating; the same channel may be used for notifications from master to slave, acknowledgments from slave to master, and subscribe requests from slave to master.
- Remote Procedure Call (RPC). The DOCP master makes a direct RPC call to the slave's notification port (address). An interface has not been defined for any specific RPC mechanism, but the DOCP should support direct client/server communication. RPC can also have issues with traversing firewalls, but has other benefits such as efficient message encoding, well defined security mechanism, development tool support, and procedure call/return semantics.

The slave communicates its notification port (or binding handle) to the master prior to the first subscription request. Each notification message body identifies all modified objects at that master to which the slave is believed to be subscribed. This follows from the observation that many objects can change at or near the same time, as in the case of a tree update operation by a content publisher. The invalidation indicates the former and current modification time of the changed objects and an object digital signature (for example a checksum). Each invalidation message carries a sequence number to allow the slave to detect a missing notification.

The slave must acknowledge each notification; it may negatively acknowledge a notification if it detects a gap in sequence numbers. The slave may send one acknowledgment message for multiple notifications.

When the slave receives an invalidation notification it must annotate the object's metadata such that it will not serve the object again from its cache. After an invalidation the object must not be served to clients, even if communication with the master for a fresh copy fails, since the object may have been removed by the content provider. The slave may remove the object data from disk although removal does not have to be synchronous with the request; it can be done at a quiet time or when disk space is next needed.

The slave could keep object data on disk and use a delta encoding to update the data when it is next requested, as proposed by Mogul et al [21].

5.8 Optimistic Discovery

The web is a large-scale, widely deployed distributed system. Such systems are impossible to update in a synchronized manner due to their size and complexity, not to mention that the origin and proxy servers in the web are independently owned and managed. Therefore DOCP technology must be incrementally added to the system. To accommodate this, DOCP slave servers need to be able to discover DOCP masters. This is accomplished through *optimistic discovery.*

To enable optimistic discovery, a DOCP master must include a **`DOCP-Lease`** header in each response in addition to standard HTTP headers (e.g., **`Expires`**) for the object. Non-DOCP proxies will ignore the extra header field and use standard HTTP cache consistency rules. If desired, this can enforce strong consistency by making proxy cache servers validate consistency every request using **`GET IMS`** requests. DOCP slaves will recognize that the object should be served with the DOCP consistency guarantee, and that they can make subscription requests to that server and receive a valid lease.

This approach provides another potential optimization. Many objects on the web are written only once, but read many times, for example all the little colored dots, company logos, and so on. Since a read-only object will never change (by definition), a DOCP master could always include an "infinite" lease for these objects upon every access. The "infinite" lease can be interpreted at the slave as "a very long time from now", for example the end of time of the local system's clock. It is not necessary that all slaves treat this value as the same absolute time; any time in the distant future will do. A DOCP slave receiving such an unsolicited lease should automatically mark the object as subscribed for that interval and never request to validate the object (unless it is evicted from the cache and re-referenced). The master does not need to maintain a subscriber list for read-only objects since an invalidation will never be sent.

If such an object needed to change, a new object URL must be created and links updated to point to the new object. Using this technique, many more objects could be made to be read-only. There is no mechanism for deleting these read-only objects, so they must be used with care.

5.9 Access Logging

When content is served from caches content providers do not see as many hits at their site. The DOCP addresses this by transmitting access log information from DOCP slaves to the DOCP master for each server for which it served content. These access logs are transmitted periodically and should be transmitted at quiet times (e.g., pre-dawn), as negotiated by the master and the slave. Logs should be made available by the master in one of the accepted standard formats for log reporting, for example the Common Log Format (CLF). A DOCP slave may log all requests by DOCP master instead of using a single log file for all requests and later extracting them.

Content providers can use the access information to see greater detail of user access patterns on their site. A server log today may contain only a single hit from a proxy for the first request of an object. Subsequent user requests to the proxy result in a cached response and are therefore not seen by the origin server until a validation

request (**`If-Modified-Since`**) is made. Furthermore, access by all users of a proxy appears to the origin server as coming from the same source (the IP address of the proxy server). Therefore log analysis on servers provides only an approximation of user activity.

By having full proxy logs at the origin server the content provider is able to see all user requests that arrived at the proxy and use that information to better understand traversal patterns and optimize their site's layout. A web hosting provider can supply detailed analysis of the access patterns at all DOCP slaves as a value added service to a content provider.

5.10 Predictive Subscription Renewal

If an object is deemed to be extremely popular, a DOCP slave may request a subscription prior to the next user request. In general it is hard to predict both the time of the next request and the time of the next modification. We have observed that modifications to our own web pages tend to occur in clusters where a single page will receive several updates in a relatively short time, sometimes without intervening reads.

For this reasons, the DOCP uses a passive model that achieves the best use of the network while providing better average retrieval latency than the TTL-based model. However, renewal of a subscription after expiration is allowed by the protocol. This is particularly useful to support high availability of content at the slave or assurance of fast response time.

Pushing copies of modified objects is also being considered, for example to support dynamic mirror maintenance. This is not currently defined by the protocol pending definition of an economic model for push caching in the DOCP.

5.11 Content Provider Updates

The mechanism for content provider updates is external to the DOCP protocol. Various methods may be used to communicate changes to the DOCP master.

- A content publishing system can transmit information about published objects as they are changed.
- An application can scan the content for changes.
- Manual notification can be sent to a DOCP master.

The most effective and efficient mechanism is to integrate with the content publishing system. In any case a consistency manager must check each published object. It computes the object's modification date and checksum, compares these with the previous state of the object, and notifies the subscription manager if the object differs. The subscription manager determines if any of the changed objects had current subscriptions and generates change notifications for them.

5.12 Hierarchy for Scalability

In order to scale to very large networks DOCP proxies can be configured into a hierarchy. The hierarchy should reflect network topology with parent proxies serving a set of child proxies logically farther from the origin servers they are attempting to contact; for example proxies within an ISP should route their requests through a regional parent proxy for that ISP.

Many organizations implement security firewalls, creating few places in the network where the organization is connected to the external network. Caching at this location is natural, and satisfies the goal of increasing scalability through hierarchy. Each organization should have one or a few DOCP proxies connected to the external network, and possibly internal proxies connected to those "parent" proxies.

A DOCP parent will serve as a master for some set of DOCP child proxies, and as a slave to other DOCP masters. The intermediate parent proxy's role is

- to aggregate HTTP **`GET`** and DOCP subscribe messages for the child slave proxies it serves
- to manage subscription lists on behalf of these slaves and the DOCP masters to which they subscribe
- to aggregate access information from slaves and forward logs to parent DOCP masters
- to redistribute DOCP invalidation messages from external masters to the slaves when content changes.

If there are many DOCP child proxies interested in an object, only one subscription needs to be made to the origin's DOCP master; the others can be served by the DOCP parent. All subscribed slaves share the same lease value granted by the master.

Not all DOCP proxies need to maintain a copy of the object data. A DOCP parent may maintain only a subscription list for its child proxies. The children would manage object data and receive notification when an object changes, purging it from their caches.

A data-less parent will have knowledge of the slaves that are subscribed to popular objects but not object data. To serve a request for a subscribed object the parent may request a copy of the data from one of its children; if no child has a copy of the subscribed object the parent must obtain a copy from the origin. A slave can subscribe to an object with a DOCP parent without further communication with the origin. When an **`IMS`** request arrives for a subscribed object (whether the parent has the data or not), the parent may respond with an HTTP **`304 Not Modified`** (if it has not been invalidated) and include a **`DOCP-Lease`** header with the lease expiration time for the object. The request is handled locally, while providing the consistency guarantee.

A DOCP parent also aggregates access accounting information from its children. It periodically receives updates on subscribed objects it has served to its children, aggregates them with data from direct access and other children, separates the access information according to DOCP master, and forwards the appropriate data to each of the DOCP masters that have granted subscriptions to the DOCP parent.

6 HTTP Cache Control

The DOCP makes several HTTP cache control headers unnecessary. In a DOCP slave the HTTP cache control headers are subordinate to DOCP headers. Note that a DOCP slave may communicate both with DOCP masters and ordinary web servers and other non-DOCP proxies. A DOCP slave must still implement HTTP cache control headers.

6.1 Pragma: No-Cache

This HTTP/1.0 header can be attached to a client **`GET`** request or a server response. It indicates that the proxy should not serve a copy from the cache but rather a current copy from the origin server. In practice, some cache implementations modify this into a **`GET IMS`** request so they do not have to retrieve the whole object if it has not changed since the last modification time. This is the mechanism by which users can check object consistency if they believe their cache to be out of date.

With DOCP, subscribed objects are served authoritatively from the DOCP slave cache. Therefore the slave may respond with a **`304 Not Modified`** response without communicating with the DOCP master. If an object is modified the DOCP slave will be informed and will serve a consistent object on the next request.

6.2 Expires

The **`Expires`** header on a returned object includes a timestamp that identifies how long an object may remain in cache before it is validated. Prior to the object's expiration a cache may serve the object without checking its validity. After expiration the cache must request a new version from the origin server. Some cache servers interpret this as allowing a validation with the origin server.

It is difficult for content providers to use this feature effectively because it is hard to predict when an object will be modified. Even for objects with regular publication schedules, there are cases where a retraction is necessary to correct factual or other errors. If the content provider uses the **`Expires`** header to prevent unnecessary **`IMS`** requests, the incorrect object will remain in caches until it expires. An HTTP **`PURGE`** method has been proposed to address this (as an IETF work in progress report; no published version exists yet).

This header is also used to prevent a cache from holding an object too long before checking its consistency. This is the de-facto mechanism to assure consistency in the web today. However, this polling mechanism leads to increased request traffic at origin servers and response time for end users, with little benefit since most objects rarely change.

With DOCP, this header becomes unnecessary. Instead of expiring, objects are either modified or deleted, in which case a DOCP Inv message is sent to all DOCP slave servers holding a copy of the object. Upon subsequent requests for the object the slave will retrieve a fresh copy or get an error if the object was deleted. Until that time the cache may serve the object directly.

6.3 Cache-Control: Headers

The HTTP/1.1 specification defines explicit cache control headers, which are handled as follows.

6.3.1 max-stale, min-fresh, min-stale,...

It is not clear whether or how web browsers will support these headers, nor how end users will choose the values. There is little reason for end users to set anything but "always give me the most current version" since the latency trade-off is difficult to quantify to end users. Practical experience also indicates end users will do as little as possible, and that they have minimal understanding of the operation of the web service. Content providers and server operators also improperly use the existing headers.

DOCP will ignore these headers the same as it does for **`Pragma: No-Cache`**: it will serve objects authoritatively, and await notification of change from the DOCP master.

6.3.2 Do-Not-Cache, Private,...

These headers are intended to prevent a cache from holding a copy of an object. The reason for preventing caching of objects is often for accounting or consistency reasons, which DOCP addresses. However there are other reasons to prevent caching of an object including for intellectual property protection. DOCP caches must honor these HTTP/1.1 headers and not hold a copy of objects so marked. Content providers should not use these headers just to obtain hit accounting or to assure consistency since DOCP provides that function already.

7 Protocol Analysis and Simulation

During the design of DOCP we explored several consistency models from per-server to per-object (and per-subtree in between). A per-server model limits the amount of state that must be maintained by proxy and origin servers, however it is imprecise: the server does not know specifically which of its objects are subscribed. A per-object model carries the most precise state information about subscriptions, but requires the most state on the origin and proxy servers. In the end we chose to maintain consistency on a per-object basis after evaluating the object popularity distribution within a server, and therefore the number of imprecise invalidation messages that would have been sent. To address concerns about servers being able to maintain all that state, we observed that currently web proxy cache servers maintain per-object state; in fact they require more state for the TTL value and to perform positive validations than DOCP requires to manage subscriptions and perform invalidations.

Next we wanted to validate the assumptions about scalability and efficiency (that this protocol was at least no worse than the current weak consistency mechanism), as

well as explore various trade-offs in protocol design, such as the computation of the lease interval and the determination of when an object was sufficiently popular. In order to accomplish this we extended a simulator previously used to measure the performance of various replacement policies [3] such that it could also simulate the consistency protocol.

The method we used to analyze the characteristics and performance of the new protocol is trace-driven simulation. This method consists of replaying logs (traces) from real users through a simulator that is as similar to the protocol and underlying network as is feasible. We chose trace-driven simulation because Internet traffic has been shown to be difficult to model accurately. By using a trace of actual activity we are assured of having a valid model of the activity, although only at a single location and for a limited time. Fortunately, we had access to a large data set from a busy proxy server used earlier to perform a workload characterization [2]. The data were gathered by logging every request made by a population of thousands of home users connected to the web via cable modem technology over a five-month period (a total of 117 million requests). Using this log, we were able to simulate protocol behavior for a set of web objects accessed through a cache by a large group of users representative of other high-speed home users.

We also built a statistical model of object updates, since that information is not present in the logs. We did this by studying busy web server logs and the existing literature. Based upon this we used a power-law function to distribute updates among objects within a site, and an exponential distribution (Poisson process) to distribute the updates for an object within the simulation period. We simulated two modification levels: *regular* and *heavy*. The regular workload corresponds to our best estimate of object modification profiles for web objects. We are aware that the modification rate varies greatly among sites, so we also simulated a heavy modification workload, in which objects change much more frequently. This provides a worst-case scenario for the DOCP. We used the same access workload (and popularity distribution) for both modification rates.

The simulation examined the following characteristics of the protocols under study.

- Network demand. We measured the number of requests by type: misses (cold and capacity), slow hits (where the data was in cache but validation with the origin server was required), fast hits (the data was in cache but no communication with the origin server was required), and invalidations.
- Server demand. We measured the number of inbound object requests and outbound DOCP invalidation requests between origin servers and proxies that were all either DOCP or non-DOCP.
- Protocol performance. We examined performance of the DOCP under varying modification levels and lease intervals to optimize the protocol.

The subsections below present a summary of our results. Details about the simulation can be found in [24].

We have also explored cache response time based upon cache consistency [8]. Proxy logs indicate an order of magnitude improvement in response time when an object is served directly from cache (as a fast hit) as compared with a positive validation. Positive validations are again much faster than misses. This establishes the benefit of cache object consistency on response time.

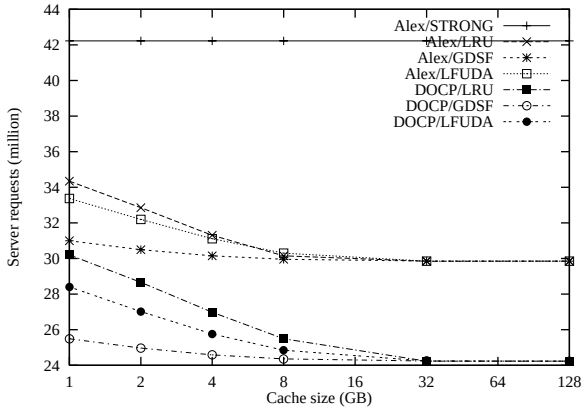
7.1 Network and Server Demand

External network demand includes the total number of external requests made by a proxy to all origin servers and the total number of bytes transferred.

Figure 3 presents the request demand a proxy running the DOCP, Alex, or strong consistency protocol, and for a range of cache replacement policies. Strong consistency is achieved through polling. This simulation used the *normal* modification workload, as explained in [24], and includes DOCP invalidations and subscription messages, and Alex validations. The simulation examined a range of cache sizes (along the x axis) from 1 GB to 128 GB. A 128 GB cache was sufficient to hold the entire working set for the simulation interval (104 GB of unique content was requested), and therefore simulates an “infinite cache”.

The simulation of the Alex consistency protocol used a 20% age factor and a max stay in cache of three weeks. These are the default consistency parameters used by the popular Squid proxy cache [27].

FIGURE 3. Request Demand



As cache size grows the external request demand decreases because objects in cache do not need to be transferred again. For smaller cache sizes the replacement policy plays a significant role in the total request demand. The cache replacement policy is the algorithm that determines which objects remain in cache and which are evicted to make room for new objects. We explored the following replacement policies [1]:

- The classical Least Recently Used (LRU) policy.

- The Greedy Dual-Size with Frequency (GDSF) policy, which is designed to maximize object hit rate by keeping more of the popular objects in cache.
- The Least Frequently Used with Dynamic Aging (LFUDA) policy, which is designed to maximize byte hit rate by keeping more popular bytes in cache.

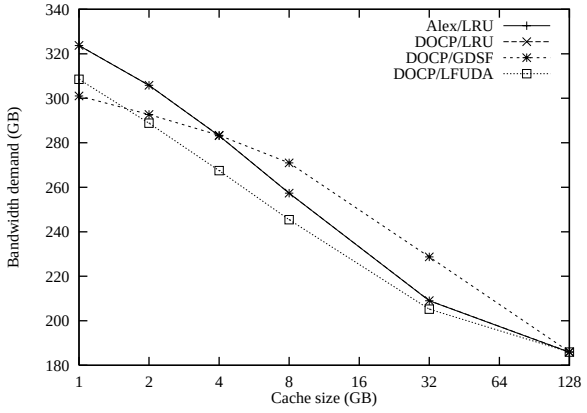
LFUDA ignores object size in makin its replacement decision. GDSF keeps more objects in cache by evicting larger objects before smaller ones (assuming they have the same popularity or reference count).

As cache size grows to the size of the working set (effectively an “infinite” cache), the replacement policy does not affect the request demand. In an infinite cache all objects that are cachable will be in cache after the first request. At this point only the consistency policy plays a role in determining request demand. The DOCP reduces the number of external requests by approximately 19% as compared with the Alex protocol; and by 42% when compared with strong consistency. It does this while delivering object consistency equivalent to the strong policy (polling every time).

At smaller cache sizes, the consistency policy reduces external demand for each of the replacement policies by approximately the same ratio as at larger cache sizes.

Figure 4 presents the total bandwidth demand under the same configuration of proxies.

FIGURE 4. Bandwidth Demand



The consistency protocol does not play a significant role in the reduction of bandwidth demand between proxies and origin servers. Bandwidth demand is dominated by the cost of transferring object data across the network. The consistency protocol reduces the number of consistency validation messages required between proxy and origin servers, but these are small messages, as are the invalidation messages that take their place.

In Figure 4 the lines for the Alex protocol with the GDSF and LFUDA policies have been left off for clarity. They fall exactly on the DOCP/GDSF and DOCP/LFUDA lines, just as Alex/LRU and DOCP/LRU are overlaid.

7.2 Protocol Performance

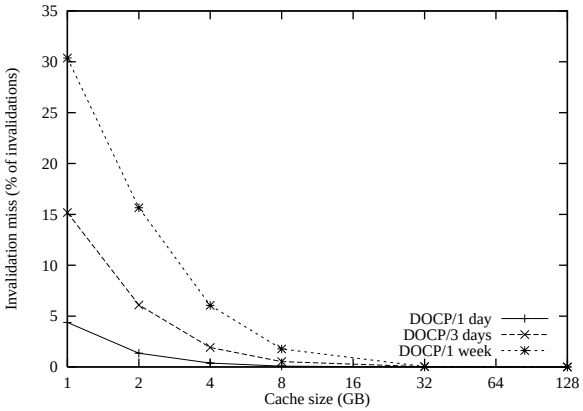
During the simulation we explored the performance of the DOCP with various cache replacement policies and choices of lease interval.

To quantify the performance of the DOCP subscription mechanism under varying lease durations we added two sensors to our simulator. The sensors measure the *invalidation miss rate* and *subscription miss rate*. The invalidation miss rate is the ratio of object invalidation requests for an object that the slave’s cache replacement policy had already removed to total invalidations. The subscription miss rate is the ratio of subscription requests at a master for an object the master considers the slave already subscribed to total subscriptions. Both of these types of misses are inefficient and should be minimized.

During protocol development we explored various lease intervals including static intervals of one day, three days, and one week; and a lease interval based on the last modification time of an object, which attempted to expire at or near the next predicted modification time. Predicting next modification time proved to be difficult: the decision was often wrong, resulting in inappropriate lease durations. In the end we chose not to use the more complex dynamic lease computation mechanism.

Figure 5 shows the invalidation miss rate for the three lease intervals we explored using the normal workload, a 4 GB cache, and the GDSF replacement policy.

FIGURE 5. Invalidation Miss Rate by Lease



Extending the lease interval does not affect network bandwidth demand because renewals and validations do not carry much content relative to object data transfer. It only alters the length of time a popular object will be served during a subscription. Increasing the lease allows a slave to deliver more fast hits, and also increases the chance of eviction of a subscribed object. Sending invalidations increases DOCP master workload, but since they occur asynchronously to user requests they do not affect response time. A long lease also increases the size of the master’s subscriber list. We did not study the protocol’s effect on master sate in this simulation. The impact of the lease interval on fast hit rate, request

demand, and bandwidth demand is summarized in Table 1.

Table 1 Lease Impact - 4 GB cache, DOCP/GDSF

Lease	1 day	3 days	1 week
Fast hits	36.78%	41.78%	44.79%
Bandwidth (GB)	283.337	283.354	283.719
Requests (million)	26.696	24.582	23.312
Invalidation miss (% invalidations)	0.38%	1.92%	6.05%

7.3 Hit and Miss Rates

The next two figures present the fast hit rate and total hit rate of the two consistency protocols, three replacement policies, and six cache sizes we studied. These are all based upon the normal workload, default Alex consistency parameters, and a static three day DOCP lease.

FIGURE 6. Fast Hit Rate

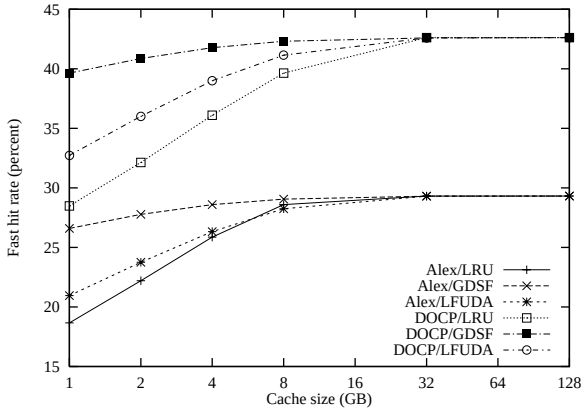
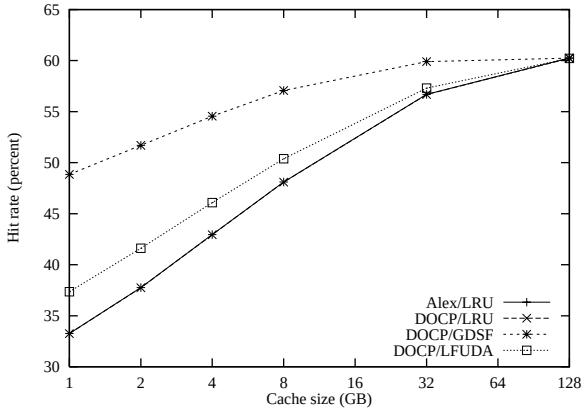


Figure 6 shows an improvement in fast hit rate when using the DOCP protocol as compared with the Alex protocol regardless of cache size. In large caches the replacement policy is not a factor; the consistency protocol alone determines the fast hit rate.

FIGURE 7. Hit Rate



The consistency protocol does not significantly affect the overall cache hit rate nor the byte hit rate. These are determined by the cache replacement policy. The consistency protocol improves the fast hit rate by allowing the cache to serve objects directly from cache rather than wait for a consistency validation.

We also explored the impact of the cache replacement policy on the subscription and invalidation miss rate.

The following two figures illustrate that for a small cache the choice of replacement policy has a dramatic effect on the subscription and invalidation miss rates. When the cache is a large enough to hold all modified objects the miss rate drops to zero.

FIGURE 8. Subscription Miss Rate

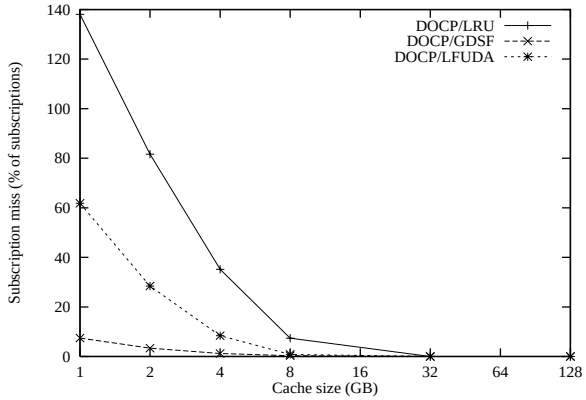
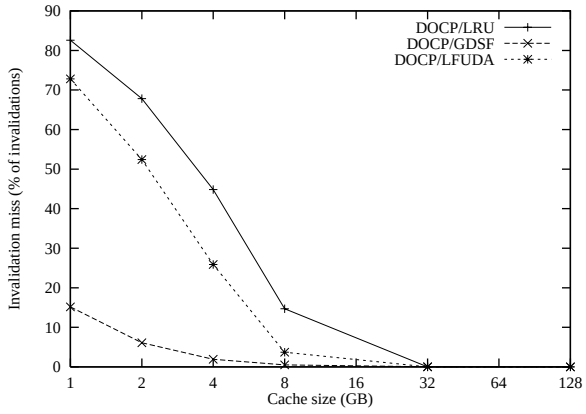


FIGURE 9. Invalidation Miss Rate



Note the synergy between the GDSF replacement policy and the DOCP consistency protocol. The GDSF policy tries to keep more popular objects in cache. The DOCP subscribes based upon object popularity.

By keeping more popular objects in cache, a 1 GB slave using the GDSF policy generates 95% fewer subscription misses masters than when using the LRU policy.

The LFUDA policy also keeps popular objects, but it ignores size and therefore keeps fewer, larger objects. This increased the overall miss rate and therefore also the subscription and invalidation miss rate.

7.4 Simulation Summary

In summary, the simulation of the DOCP in a wide area network showed an overall reduction in connection demand while serving zero stale hits. The eliminated requests were positive validations for cached objects: instead of a slow hit, where a cache must make a wide area round trip, the DOCP slave achieves a fast hit by serving subscribed objects immediately from its cache. This reduces client response time and server demand.

We simulated the three replacement policies defined in [1] using the DOCP and Alex adaptive TTL consistency protocols. The replacement policy was the dominant factor in bandwidth reduction; the consistency protocol had little effect on bandwidth demand since the connections eliminated are validations, which only exchange header information. The new replacement policies show a significant improvement in bandwidth demand and total in hit rate, as reported in the earlier work.

The choice of replacement policy affects the performance of the DOCP for smaller cache sizes. Frequency based policies show better synergy with the frequency based subscription mechanism, and cause less wasted effort on the part of DOCP masters and slaves.

8 Related Work

The DOCP is based upon a significant body of work, most notably in the area of scalability, caching and replication in wide area distributed systems. The focus of the work presented here is to bring together a number of previous results in order to construct an efficient protocol for object consistency in the world-wide web.

Neuman [23] presents an overview of scale in distributed systems, motivating the need for scaling on the basis of improved reliability and performance, and observing the complexity of multiple administrative domains in large-scale widely distributed systems. Neuman discusses distributed naming systems, which are often used to locate system components. Neuman also discusses replication, caching, and consistency, provides a set of guidelines for designing scalable systems, and examples of several such systems. Another useful set of guidelines can be found in Lampson [17].

Kermarrec et al [11] have defined a framework for consistent, replicated web objects. Their framework is part of the Globe wide-area distributed systems research project. In their paper they define a hierarchy of coherence models and a framework to allow clients and stores to negotiate the coherence they wish according to a set of implementation parameters. Their framework is more general than what we propose. No HTTP implementation was proposed as part of this work.

Liu and Cao [19] demonstrated that strong consistency could be achieved at about the same cost as the current weak consistency approach implemented by proxy caches. In their paper, every object that changes caused an invalidation message to be sent to the caches that accessed that object. They demonstrated that with this model they could support consistency equivalent to polling every time without increased network demand. In the proxy workloads we have studied we have observed that most objects are never re-referenced even over significant durations [2], so sending notifications for these objects should be avoided. Their report encouraged us to explore ways to provide stronger consistency in the web more efficiently and with better scalability.

Krishnamurthy and Wills [13] demonstrated that clever use of existing HTTP requests can improve the consistency of web objects. However, no guarantees are made of the degree of object consistency; the proposed mechanism also may cause a significant number of unnecessary invalidation messages to be exchanged.

Yin et al [29] propose using volume leases to support consistency in large-scale systems. This builds on the earlier notion of leases from Gray and Cheriton [10]. Volume leases allow a server to make progress updating objects even if clients (proxies) fail. The server need only wait until a relatively short volume lease expires before modifying an object. Each proxy must have both a short volume lease and a longer object lease for each consistent object. This approach supports true strong consistency since no object will be served without an active lease, and no object will be modified without revoking or waiting for termination of a lease. There is still the possibility of a delayed read or write response if a server can not be contacted. This is fundamental to strong consistency. Their algorithm also supports a “best effort” consistency level similar to our notion of delta consistency, which allows some content to be served with a strong consistency guarantee and other content with best effort (with corresponding update semantics).

Version 1.0 of the DOCP uses only object leases and has a relaxed consistency requirement, with a delta between modification and eventual consistency. This delta is similar to the normal lag between when an object is created and when it is made available through a hosting service. We believe there is an opportunity to integrate our approach with theirs to take advantage of volume leases to provide improved consistency.

Sandpiper and Akamai have developed services that distribute content over the Internet from content providers into the geographic locality of clients, to reduce client response time and server load. Their work and ours share the same goals; our approach is to improve the core HTTP protocol while they layer a service above it.

WebSpective and others also have products to consistently replicate content. They focus on a data center under common administrative control. We allow ownership of proxies and content providers to differ and propose a standard protocol for wide area consistency.

There is some other related work worth a brief mention:

- The IETF hit metering RFC proposed a mechanism for proxy caches to inform origin servers how many hits they received for cached objects. This has not been widely adopted, possibly because the proxy

caches cannot be audited and therefore the information is suspect. Our approach is auditable and accountable, and furthermore log aggregation provides the opportunity to see specific user browsing patterns in proxy caches.

- The SkyCache service populates its subscribing caches with data that has been found to be popular elsewhere. The service uses an out-of-band satellite path. Cache misses from subscribing caches are relayed to SkyCache Central where a set of rules determine if an object is sufficiently popular to be sent "up to the bird." If so the object is sent up once and broadcast to all participating caches.
- Multicast and other push technologies have been suggested, although the asynchronous web model has been shown not to match well with the synchronous multicast model. Multicast and push may be viable for simultaneous popular media or objects, but the infrastructure has yet to mature.

There are several proposals and much work in progress aimed at improving web performance. Some, like ours, require modification to core infrastructure. This type of modification is an expensive, long-term effort. We believe that when such modifications are being made, they should yield the broadest possible improvement to the overall system.

9 Conclusions

We have designed and simulated a distributed consistency protocol for web objects in the current Internet environment. We have confirmed earlier research findings that stronger consistency can be achieved in the web at lower cost than weak consistency. Through protocol simulation we have demonstrated a reasonable reduction in origin server load and improvement in page load times as a result of the elimination of unnecessary consistency validation requests in the current system.

Providing consistency along with access accounting can eliminate the need for cache-busting techniques on the part of content providers. Since objects are served consistently with what the provider intended, and since access information is returned to the provider by the DOCP service, there should be no need for cache-busting. Furthermore, inappropriate use of headers such as **`Expires`** and **`Last-Modified`** (accidentally, through ignorance, or intentionally) will not cause the DOCP system to behave improperly, since only actual modification will cause object invalidation and subsequent network demand.

We believe this service can enable accelerated growth of caching and provide for new service opportunities. For example, improved consistency can increase confidence in web information and lead to greater business use of the web.

As bandwidth continues to grow and bandwidth cost falls we believe the primary benefits proxy cache servers will provide are to reduce response time and server demand. This protocol is focused on those two goals.

In summary, we believe the benefits of the protocol presented here are as follows.

- **Better** consistency for content providers and users.
- **Faster** response times for popular objects.
- **Cheaper** for the origin server to serve fewer hits.
- **Simpler** for everyone by replacing confusing cache control headers with a cleaner, simpler mechanism.

9.1 Future Work

Work is currently in progress to integrate this protocol definition with the volume lease and consistency work done by Dahlin, Alvisi, and their colleagues at UT Austin. We believe our approaches are complimentary. Version 1.1 of the DOCP will incorporate volume leases.

Further research work needs to be done to define the security trust model and a combination of security protocols and services identified and incorporated into the DOCP architecture. We believe this work will focus on protecting the integrity of content provider objects, protecting the DOCP infrastructure from intentional abuse of cached objects, and on providing auditable access accounting information back to content providers from the edge of the network. In order to accomplish this it may be necessary to define a DOCP certification authority in order for DOCP masters to trust DOCP slaves without expensive manual configuration. We must also formally define and describe the mechanism by which access accounting information will be conveyed back to the DOCP master from DOCP slaves.

In addition to providing access to hot static objects we are looking at how to enable consistent access to dynamic and personalized data. Many dynamic requests are data-driven, however the underlying data and the applications that access it to create the dynamic response do not change with every request. Therefore, some of these requests (such as searches on popular keywords, or price or availability quotes) can be cached and even subscribed, provided an invalidation can be sent when the data (or the relevant portion of the data) changes. Distributed databases can also address this issue, however some of those mechanisms are heavyweight for a web workload.

Multicast may help to reduce the burstiness of traffic from a DOCP master. Multicast may be well suited to synchronous distribution of invalidations to slaves. Future work will explore using (reliable) multicast to distribute update notifications. Creating a multicast tree is an expensive task relative to retrieval of a web page or subscription, so the protocol must minimize the number of multicast trees. We envision a multicast tree could be set up by a DOCP master for the busiest slaves it is communicating with.

Other future work may include optimization of the subscription and lease mechanism to account for the fact

that objects are distributed together. For example a web page and all of its inline images can be thought of as a single “package”. Additional web pages may be logically contained within a single package, such as for a home page, its images, and the most popular destinations off the home page (many of which share a set of images). By distributing packages instead of individual objects, efficiency could be improved. This also would help reduce the burstiness of change notifications.

In addition to researching multicast, packaging, and other techniques, we are working to characterize server demand under a standard, unicast model.

Finally, we would like to pursue an open standards-based approach to the formal definition and adoption of this technology. Only through broad adoption will this mechanism deliver its full value in improving web response time and reducing network and server load.

10 Acknowledgments

This work was motivated by the desire to improve the currently inefficient object replication in the web. It is based upon a substantial body of research and development in wide area distributed systems, in particular distributed file systems like xfs and afs. Some of the related work is referenced, but other is part of our “collective consciousness.” We would like to be the first to acknowledge that we did not invent consistency, invalidations, leases, or the Internet. To paraphrase Newton, we are standing on the shoulders of giants...

The authors are specifically grateful to Rich Friedrich who has provided guidance and feedback on the technology and the business aspects of this work, to Godfrey Tan who assisted in early protocol design discussions, and to John Barton, Nina Bhatti, Mark Nottingham, and Craig Wills, for their helpful review comments.

This version of the protocol was also significantly improved through discussions with Michael Dahlin and his research group at UT Austin. Their team contributed the method to handle clock skew in particular, and a wide variety of other useful comments - some of which have been addressed, others will be addressed in a future version of this technical report.

11 References

[1] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, T. Jin, “Evaluating Content Management Techniques for Web Proxy Caches”, in Proceedings of the 2nd Workshop on Internet Server Performance, Atlanta GA, May 1999.

[2] M. Arlitt, R. Friedrich, T. Jin, “Workload Characterization of a Web Proxy in a Cable Modem Environment”, in ACM SIGMETRICS Performance Evaluation Review, vol 27 no 2, pp25-36, August 1998.

[3] M. Arlitt, R. Friedrich, T. Jin, “Performance Evaluation of Web Proxy Cache Replacement Policies”, to appear in Performance Evaluation, 1999.

[4] M. Arlitt, T. Jin, “Workload Characterization of the 1998 World Cup Web Site”, Technical Report HPL-1999-35, February 1999.

[5] A. Birrell, R. Levin, R. Needham, M. Schroeder. “Grapevine: An exercise in distributed computing.” Communications of the ACM, April 1982.

[6] V. Cate. “Alex -- A Global Filesystem”. In Proceedings of the USENIX File System Workshop, pages 1--12, May 1992. USENIX Association.

[7] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, K. Worrell. “A Hierarchical Internet Object Cache”. In Proceedings of the 1996 USENIX Annual Technical Conference, January 1996.

[8] J. Dilley, “The Effect of Consistency on Cache Response Latency”, Technical Report HPL-1999-107, HP Laboratories, September 1999.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1”, IETF RFC 2616, June 1999. The Internet Society.

[10] C. Gray, D. Cheriton, “Leases: An efficient fault-tolerant mechanism for distributed file cache consistency”. In Proceedings of the 12th ACM Symposium on Operating Systems Principles, pp 202-210, Dec 1989.

[11] A.M. Kermarrec, I. Kuz, M. van Steen, A. S. Tanenbaum, “A Framework for Consistent, Replicated Web Objects”. In Proceedings of the 18th International Conference on Distributed Computing Systems, May 1998.

[12] M. Korupolu, M. Dahlin. “Coordinated Placement and Replacement for Large-Scale Distributed Caches”, IEEE Workshop on Internet Applications, pp 62-71, July 1999.

[13] B. Krishnamurthy, C. Wills. “Study of piggyback cache validation for proxy caches in the world wide web”, USENIX Symposium on Internet Technology and Systems, pp 1-12, Monterey CA, December 1997.

[14] B. Krishnamurthy, C. Wills. “Proxy Cache Coherency and Replacement - Towards a More Complete Picture”. In Proceedings of the 19th International Conference on Distributed Systems, Austin, TX, June 1999.

[15] L. Lamport, “Time, Clocks, and the ordering of events in a distributed system.” Communications of the ACM, v 21 no 7, pp 558-565, July 1978.

[16] L. Lamport. “How to make a multiprocessor computer that correctly executes multiprocess programs”. IEEE Transactions on Computers, September 1979.

[17] B. Lampson, “Hints for Computer System Design”, In Operating Systems Review, v 15 nr 5, pp 33-48, Oct 1983.

[18] R. J. Lipton, J. S. Sandberg. “PRAM: A Scalable Shared Memory.” Technical Report CS-TR-180-88, Princeton University, September 1988.

[19] C. Liu, P. Cao, “Maintaining Strong Cache Consistency in the World-Wide Web”, Proceedings of the 17th IEEE International Conference on Distributed Computing Systems, May 1997.

[20] D. Mills, “Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI”, Internet RFC 2030, October 1996.

[21] J. Mogul, F. Douglis, A. Feldmann, B. Krishnamurthy. “Potential benefits of delta encoding and data compression for HTTP”. In Proc. ACM SIGCOMM, pages 181-194, September 1997. AT&T Labs Research TR 97.22.1.

[22] D. Mosberger. “Memory Consistency Models”. Operating Systems Reviews, 27(1):18--26, Jan. 1993.

[23] B. C. Neuman, “Scale in Distributed Systems”, In Readings in Distributed Computing Systems, IEEE Computer Society Press, 1994.

[24] S. Perret, J. Dilley, M. Arlitt, “Performance Evaluation of the Distributed Object Consistency Protocol In a Web Proxy Cache”. Technical Report HPL-99-108, Hewlett-Packard Laboratories, September 1999.

[25] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel. “LOCUS: A Network Transparent, High Reliability Distributed System”. In Proceedings of the Eights Symposium on Operating Systems Principles (SOSP), pp 169-177, December 1981.

[26] J. Postel, K. Harrenstien, “Time Protocol”, Internet RFC 868, May 1983, Internet Society.

[27] Squid Internet Object Cache, http://squid.nlanr.net/

[28] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, B. B. Welsh. “Session Guarantees for Weakly Consistent Replicated Data”. In Proceedings of the 1994 Symposium on Parallel and Distributed Information Systems, pp. 140-149, Austin, TX, September 1994.

[29] J. Yin, L. Alvisi, M. Dahlin, C. Lin. “Using Leases to Support Server-Driven Consistency in Large-Scale Systems”. IEEE Transactions on Knowledge and Data

Appendix A: Protocol Details

This section presents the protocol interactions between a DOCP slave and master using DOCP atop HTTP. The syntax of HTTP requests is described in RFC 1945, “Hypertext Transfer Protocol -- HTTP/1.0” and updated in RFC 2616, “Hypertext Transfer Protocol -- HTTP/1.1”. We do not describe the core HTTP protocol here but rather its use by and interaction with DOCP

HTTP GET

An HTTP **`GET`** request is used to retrieve an object when it is not in cache or when its **`Last-Modified`** date is unknown. It always retrieves object data and should also retrieve the modification date for later object validation.

The response to an HTTP **`GET`** request is either HTTP status **`200 OK`** and an object body, a redirection, or an HTTP error. A redirection is a response that refers the requestor to another URL. Most browsers will then fetch the new object.

DOCP compliant slaves and masters must implement these requests according to the specification. A DOCP master will include an additional header in its response:

`DOCP-Lease: Granted 0`

This supports optimistic discovery. See section 5.8, Optimistic Discovery on page 7 for a discussion of this mechanism and Table 2, “DOCP-Lease Header Values,” on page 19 for a complete discussion of lease values.

HTTP GET IMS

An HTTP **`GET If-Modified-Since`** request is used to validate an object and retrieve a fresh copy if it has been modified since its **`Last-Modified`** date. A slave can only use this request type if the modification date is known.

The response to an HTTP **`GET IMS`** request is either an HTTP **`304 Not Modified`** indication with no object data or an HTTP **`200 OK`** response with a new copy of the object, which had been modified since the **`Last-Modified`** date supplied by the requestor. The **`200 OK`** response should also include a **`Last-Modified`** time for future validation.

A DOCP master should include the **`DOCP-Lease: Granted 0`** response the same as with a **`GET`** response.

DOCP-Inv

A DOCP Invalidation message is sent from a master to invalidate one or more subscribed objects on a slave. The **`DOCP-Inv`** message carried over HTTP contains the following headers.

`DOCP-Master:` *`Master-Ident`*
`DOCP-Host:` *`Master-Host TxnId`*
`DOCP-Inv:` *`URI Last-mod Mod-time`*

Master-Ident is the identity of the DOCP master as described below. *Master-Host* is the host whose object has changed (the host part of the URL). *URI* is the Uniform Resource Identifier for the changed object.

Last-mod is the previous modification time of the object at the master. When received by the slave this should be the last modified time of the object. If this time is earlier than the slave’s notion of the modification time the **`DOCP-Inv`** message is ignored; this is a late invalidation message arrival, after the client has already refreshed the object. If the *Last-mod* time is greater than the slave’s notion of the object modification time the object is invalidated as usual and the new value marked as the object’s more recent known modification time. Any get or subscribe response from the master must include a *Mod-time* at or after this time or it will not be honored. This is the case if a **`DOCP-Inv`** arrives early.

Mod-time is the new modification time of the object at the DOCP master. It is used in the lease calculation as described in the DOCP-Lease section below.

TxnId is a transaction identifier (sequence number) assigned by the DOCP master. Each notification message sent from a master to a slave has a monotonically increasing sequence number. This allows a DOCP slave to detect that it has missed a **`DOCP-Inv`** message and request a retransmission.

There may be multiple **`DOCP-Inv`** lines for each **`DOCP-Host`** line. There may be multiple **`DOCP-Host`** lines for each **`DOCP-Master`**. There may be multiple **`DOCP-Master`** header lines sent on a single connection.

A master notification agent must keep invalidations for a slave until the *TxnId* (or a later *TxnId*) is acknowledged by the slave. A slave may request any *TxnId* after the last acknowledged *TxnId*

DOCP-Inv-Ack

The slave must respond with an acknowledgment that it has received and processed each DOCP-Inv request. The slave should not acknowledge a *TxnId* until it has received and acknowledged all previous *TxnId*s. The slave’s response may acknowledge multiple *TxnId*s. It must apply each invalidation immediately upon receipt, even if waiting for a missing invalidation.

`DOCP-Inv-Ack: TxnId mInv nInv`

TxnId is the identifier from the previous **`DOCP-Inv`**.

mInv is the number of invalidations made by the slave.

nInv is the number of invalidations requested in the Invalidation message. This allows the master to confirm that all invalidations were processed, and also to compute the number of invalidation messages for which the cache no longer had data.

A cache may replace a subscribed object at any time to make room for other objects as per local policy. The invalidation miss rate helps to tune the lease duration.

DOCP-Inv-Nack

The slave may indicate negative acknowledgment for a range of missing *TxnId* sequence numbers.

`DOCP-Inv-Nack: TxnMin TxnMax`

TxnMin and *TxnMax* are the minimum and maximum sequence numbers that have been seen by the slave. The sequence numbers between were missed (and not acknowledged) by the slave.

Upon receipt of a **`DOCP-Inv-Nack`** the master must retransmit the notifications between *TxnMin* and *TxnMax* to the slave in a **`DOCP-Inv`** message. The master notification agent should cache these messages until they have been acknowledged.

DOCP-Subscribe Request

If a slave learns that a remote master supports the DOCP protocol through optimistic discovery (**`DOCP-Lease: Granted 0`** in a response), it may request to subscribe to objects from that server. To subscribe a slave includes the following additional headers in a subsequent HTTP **`GET IMS Sub`** request.

`GET` *URI* **`HTTP/1.1`**
`If-Modified-Since:` *IMS-time*
`DOCP-Subscribe:` *Slave-Ident Slave-time [Mod-time]*

URI is the Uniform Resource Identifier for the request.

IMS-time is the value of the most recent **`Last-Modified`** header for that object, as usually used with an **`If-Modified-Since`** request.

Slave-Ident is the identity of the slave.

Slave-time is the current clock time at the slave in seconds and microseconds, expressed as **`sec.usec`**. This allows the master to estimate the slave's current time and to assign a lease expiration time in the client's time frame.

Mod-time is the value of the most recent modify time returned by the master in a **`DOCP-Inv`** message. It is optional and can only be included if the object has been previously subscribed and invalidated. *Mod-time* must be less than or equal to IMS-time.

DOCP-Lease Response

A lease header has the following syntax.

`DOCP-Lease:` *Resp-code Slave-time Lease-time*

Resp-code indicates whether the response was granted or if not why. The response code and lease time fields can take on the values proposed in Table 2.

Slave-time is the slave's time reference from the subscribe request. If desired this may be used by the slave to identify the request. If so used the slave must assure each request occurs on a unique microsecond boundary.

Lease-time is the expiration time of the lease in the client's local time frame. It is an absolute time and occurs on a second boundary.

The lease expiration time is computed by the master as described below and depicted in Figure 10.

- When the slave receives a request for an object it checks whether that object can be served locally. If not it records the current time, *Slave-time* (t_s), makes a subscribe (**`GET Sub`**) request to the master.
- When the master receives the subscribe request it records *Slave-time* (t_s) and records its value of the current time, *Master-time* (t_m).
- If no lease is currently active for the object, the master calculates the local lease expiration time and saves that value in object metadata. This is called the *Master-expiration-time*, (t_x).
- The lease expiration at the slave, *Lease-time* (t_l), is computed according to the following equations and returned to the slave.

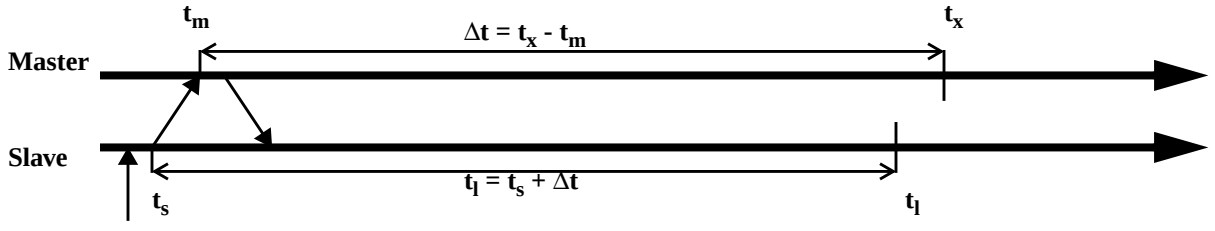
$$\Delta t = t_x - t_m$$

$$t_l = t_s + \Delta t$$

Note that the lease at the slave will expire at or before the lease expires at the master, by an amount equal to propagation time of the request from the slave to the master. Note that the propagation time, $t_m - t_s$, can not be accurately determined unless clocks are synchronized, but this does not matter.

The skew in expiration times between the master and slave may cause a master to send a notification for an object the slave believes is already expired. The slave must acknowledge this invalidation request with **`mInv`** equal to zero for that object (i.e., record this as an invalidation miss by the master).

FIGURE 10. Clock Skew and Lease Interval



The skew may also cause a slave to renew a lease prior to the master considering the lease to be expired. To accommodate this the master must start a second subscriber list prior to the expiration of the object. Requests for subscription to an object by slaves when there is less than a minimum lease interval remaining should be put on the new subscriber list.

Note that the lease response is orthogonal to the HTTP response code. A **`DOCP-Lease`** header will not accompany an HTTP error response unless that error response should be cacheable. For example, HTTP error code **`404 Not Found`** may be cached and even be subscribed to if it is a sufficiently popular response from a server.

HTTP/1.1 cache control headers may also be included in a DOCP response. These cache control headers are superseded by the **`DOCP-Lease`**, but may be passed along to subordinate (non-DOCP) proxies.

Response Calculation and Modifications

The master responds to the request based upon the *IMS-time* and *Mod-time* communicated by the slave and the object's true modification time, *True-modtime* at the master using the following rules. Note that *IMS-time* must be less than or equal to *Mod-time* at the slave if *Mod-time* is supplied in the request headers.

IMS = Mod-time AND IMS = True-modtime

This object was not modified at the master since the last time it was retrieved by the slave. The object lease has expired or there was no lease. The subscription request should be granted.

The master will respond with the following response.

`HTTP: 304 Not Modified`
`DOCP-Lease: Granted` *timestamp*

`<no object body>`

IMS < Mod-time AND Mod-time = True-modtime

This object was previously subscribed at the slave and an invalidation was sent when the object was modified at *Mod-time*. The slave does not have a copy of the new object (*IMS-time* < *Mod-time*). The object has not changed at the master since its first modification. The subscription request should be granted, but object data needs to be sent to the slave.

The master will respond with a new copy of the object and a granted subscription.

`HTTP: 200 OK`
`DOCP-Lease: Granted` *timestamp*

`<object body included in response>`

Mod-time < True-modtime

The relationship between *IMS-time* and *Mod-time* is irrelevant (although it should always be the case that *IMS-time* < *Mod-time*).

This object was previously subscribed at the slave and an invalidation was sent when the object was modified at *Mod-time*. The object was modified again between that first modification and the current request, at *True-modtime*. The slave does not have a copy of the modified object nor does it know the most recent modification. This is a "rapidly changing object" relative to the slave's requests; the subscription should not be granted.

Table 2 DOCP-Lease Header Values

Resp-code	Lease-time	Explanation of Use
Granted	0	No subscription was requested and none is granted. This (unsolicited) DOCP master response supports optimistic discovery. When a slave sees a lease value of zero it knows it may request subscriptions from that master. Until it requests a subscription the slave must use polling to validate its freshness (as if an Expires: 0 header was used).
Granted	*Lease-time*	A lease was granted for this object and expires at this clock time at the slave. The master will deliver a notification if the object changes within that time.
Granted	(unsigned 32)-1 or MAXINT	Whether a subscription was requested or not it has been granted and never expires. The slave may serve it forever without validating freshness but should still report accesses to this object to the master as with any other subscribed object. The master does not need to maintain a subscription list for such read-only objects.
Was-Modified	*Mod-time*	The object was modified since the last-modified time supplied by the slave, specifically at *Mod-time*. The master returns the new object and HTTP code **`200 OK`**. This similar to the behavior of a traditional proxy to an **`If-Modified-Since`** request.
Was-Modified	0	The object is dynamically generated and is considered modified each request. Do not request a subscription for this object.
Use-Parent	*Parent-addr*	Used by a master to indicate that a slave should subscribe through a parent, serving as a DOCP master in the geography of the requestor. This supports peer discovery without requiring an explicit location service.

The master will respond with a new copy of the object and a denied subscription.

HTTP: 200 OK
DOCP-Lease: Was-Modified *Mod-time*

<object body included in response>

The master returns the new *Mod-time* value for the object. On a subsequent request the slave may be granted a subscription if the object has not been modified again (*Mod-time = True-Modtime*).

In the case of an object that is modified more frequently than it is requested, the slave will make a subscribe request each time it validates the object (an HTTP **GET IMS Sub** request) and the master will respond with a fresh copy of the object and a lease denied with a Was-modified indication. This polling assures consistency of rapidly changing objects.

A *Mod-time* value of zero may be used by the master to indicate that an object is dynamically generated. The slave may use this information to avoid requesting a subscription on future requests.