# Using Leases to Support Server-Driven Consistency in Large-Scale Systems[*]

Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin
Computer Sciences Department
University of Texas at Austin

## Abstract

*This paper introduces* volume leases *as a mechanism for providing cache consistency for large-scale, geographically distributed networks. Volume leases are a variation of leases, which were originally designed for distributed file systems. Using trace-driven simulation, we compare two new algorithms against four existing cache consistency algorithms and show that our new algorithms provide strong consistency while maintaining scalability and fault-tolerance. For a trace-based workload of web accesses, we find that volumes can reduce message traffic at servers by 40% compared to a standard lease algorithm, and that volumes can considerably reduce the peak load at servers when popular objects are modified.*

## 1 Introduction

As valuable information become increasingly available through wide area networks, users will seek to use it in more elaborate ways. For example, although the HTTP protocol was initially developed for disseminating slowly changing scholarly and technical information, it is now often used to distribute quickly changing commercial services and news updates. In the future, we expect applications that manipulate distributed data to extend beyond human-driven browsers to include program-driven agents, robots, and data miners that will place new demands on the data-distribution infrastructure. These new applications motivate the use of caching and cache consistency.

Cache consistency can be achieved through either *client-driven* protocols, in which clients send messages to servers to determine if cached objects are current, or *server-driven* protocols, in which servers notify clients when data change. In either case, the challenge is to guarantee that a client read always returns the result of the latest completed write. Protocols that achieve this are said to be strongly consistent.

[*]This work was funded in part by a NSF CISE grant #CDA-9624082 and by gifts from Novell and Sun Microsystems.

Client-driven protocols force caches to make a difficult choice. They must either poll the server on each access to cached data or risk supplying incorrect data. The first option, polling on each read, increases both the load on the server and the latency of each request; both effects can be significant in large scale systems because servers support many clients and polling latencies can be high. The other option, periodic polling, relaxes consistency semantics and allows caches to supply incorrect data. For example, web browers account for weak consistency through a human-based protocol in which users manually press a "reload" button when they detect stale data. Weak consistency semantics may be merely annoying to a human, but they can cause parallel and distributed programs to compute incorrect results, and they complicate building aggressive caching or replication hierarchies because replication is not transparent to the application.

Server-driven protocols introduce three challenges of their own. First, strong consistency is difficult to maintain in the face of network or process failures because before modifying an object, a server using these protocols must contact all clients that cache that object. If there are many cached copies, it is likely that at least one client will be unreachable, in which case the server cannot complete the write without violating its consistency guarantees. Second, a server may require a significant amount of memory to track which clients cache which objects. Third, sending cache invalidation messages may entail large bursts of server activity when popular objects are modified.

In distributed file systems, the problems of server driven protocols were addressed by using leases [5], which specify a length of time during which servers notify clients of modifications to cached data. After a lease's timeout expires, clients must renew the lease by sending a message to the server before they may access the cached object. Leases maintain strong consistency while allowing servers to make progress even if failures occur. If a server cannot contact a client, the server delays the write until the unreachable client's lease expires, at which time it becomes the client's responsibility to contact the server. Furthermore, leases free servers from notifying idle clients before modifying an object; this reduces both the size of the server state and the load sustained by the server when reads and writes are bursty.

Although leases provide significant benefits for file system workloads, there are reasons to believe that they may be less effective in a wide area network (WAN). To amortize the cost of renewing a lease across multiple reads, a lease should be long enough that in the common case the cache can be accessed without a renewal request. Unfortunately, at least for browser workloads, repeated accesses to an object are often spread over minutes or more. When lease lengths are shorter than the time between reads, leases reduce to client polling. On the other hand, longer lease lengths reduce the three original advantages of leases.

In this paper, we show how *volume leases* restore the benefits of leases for WAN workloads. Volume leases combine short leases on groups of files (volumes) with long leases on individual files. Under the volume leases algorithm, a client may access a cached object if it holds valid leases on both the object and the object's volume. This combination provides the fault-tolerance of short leases because when clients become unreachable, a server may modify an object once the short volume lease expires. At the same time, the cost of maintaining the leases is modest because volume leases amortize the cost of lease renewal over a large number of objects.

This paper evaluates the performance of volume leases using trace-based simulation. We examine two variations of volume leases: volume leases, and volume leases with delayed invalidations. In the latter algorithm, servers defer sending object invalidation messages to clients whose volume leases have expired. We compare these algorithms with three traditional consistency algorithms: client polling, server invalidations, and server invalidations with leases. Our simulations demonstrate the benefits of volume leases. For example, volume leases with delayed invalidations can ensure that clients never see stale data and that servers never wait more than 100 seconds to perform a write, all while using about the same number of messages as a standard invalidation protocol that can stall server writes indefinitely. Compared to a standard object lease algorithm that also bounds server write delays at 100 seconds, this volume algorithm reduces message traffic by 40%.

The rest of this paper is organized as follows. Section 2 describes traditional algorithms for providing consistency to cached data, and Section 3 describes our new volume lease algorithms. Section 4 discusses our experimental methodology, and Section 5 presents our experimental results. After discussing related work in Section 6, Section 7 summarizes our conclusions.

| **Variable** | **Meaning** |
|---|---|
| $t$ | timeout for an object |
| $t_v$ | timeout for a volume |
| $d$ | time servers store state for inactive clients |
| $R$ | frequency object $o$ is read |
| $V$ | # active objects per volume |
| $C_{tot}$ | # clients with a copy of object $o$ |
| $C_o$ | # clients with lease on object $o$ |
| $C_v$ | # clients with lease on volume $v$ |
| $C_d$ | # clients whose volume leases expired $< d$ seconds ago. |
| $size(x)$ | bytes of server state to support $x$ clients |

**Figure 1: Definition of parameters in Table 1**

## 2 Traditional consistency algorithms

This section reviews four traditional cache consistency algorithms. The first two—*Poll Each Read* and *Poll*—rely on client polling. The remaining algorithms—*Callback* and *Lease*—are based on server invalidation. In describing each algorithm we refer to Table 1, which summarizes key characteristics of each of the algorithms discussed in this paper, including our two new algorithms. We also refer to Figure 1, which defines several parameters of the algorithms.

### 2.1 Poll each read

*Poll Each Read* is the simplest consistency algorithm. Before accessing a cached object, a client asks the object's server if the object is valid. If so, the server responds affirmatively; if not, the server sends the current version.

This algorithm is equivalent to always having clients read data from the server with the optimization that unchanged data is not resent. Thus, clients never see stale data, and writes by the server always proceed immediately. If a network failure occurs, clients unable to contact a server have no guarantees of the validity of cached objects. To cope with network failures, clients take application-dependent actions, such as signaling an error or returning the cached data along with a warning that it may be stale.

The primary disadvantage of this algorithm is read performance, as all reads are delayed by a roundtrip message between the client and the server. In addition, these messages may impose significant load on the servers [8].

### 2.2 Poll

*Poll* is based on *Poll Each Read*, but it assumes that cached objects remain valid for at least a *timeout* period of $t$ seconds after a client validates the data. Hence, when $t = 0$ *Poll* is equivalent *Poll Each Read*. Choosing the appropriate value of $t$ presents a trade-off: On the one hand, long timeouts improve performance by reducing the number of reads that wait for validation. In particular, if a client accesses data at a rate of $R$ reads per second and the timeout

|  | Reads |  |  | Writes |  | State |
|---|---|---|---|---|---|---|
|  | Expected stale time (seconds) | Worst stale time (seconds) | Read cost (messages) | Write cost (messages) | Ack wait delay (seconds) | Server state (bytes) |
| Poll Each Read | 0 | 0 | 1 | 0 | 0 | 0 |
| Poll | $\frac{t}{2}$ | $t$ | $min(\frac{1}{R\cdot t}, 1)$ | 0 | 0 | 0 |
| Callback | 0 | 0 | 0 | $C_{tot}$ | $\infty$ | $size(C_{tot})$ |
| Lease | 0 | 0 | $\frac{1}{R\cdot t}$ | $C_o$ | $t$ | $size(C_o)$ |
| Volume Leases | 0 | 0 | $\frac{1}{\sum_{o\in V}(R_o t_v)} + \frac{1}{R\cdot t}$ | $C_o$ | $min(t, t_v)$ | $size(C_o)$ |
| Vol. Delay Inval($t$, $t_v$, $d$) | 0 | 0 | $\frac{1}{\sum_{o\in V}(R_o t_v)} + \frac{1}{R\cdot t}$ | $C_v$ | $min(t, t_v)$ | $size(C_d)$ |

**Table 1: This table shows the cost of maintaining consistency for an object $o$ using each of the algorithms. Columns correspond to key figures of merit: the** *expected stale time* **indicates how long a client expects to read stale data after $o$ is modified, assuming random reads, random updates, and failures. The** *worst stale time* **indicates how long $o$ can be cached and stale assuming that (1) $o$ was loaded immediately before it was modified and (2) a network failure prevented the server from contacting the client caching $o$. The** *read cost* **shows the expected fraction of cache reads requiring a message to the server. The** *write cost* **indicates how many messages the server expects to send to notify clients of a write. The** *acknowledgment wait delay* **indicates how long the server will wait to write if it cannot invalidate a cache. The** *server state* **column indicates how many clients the server expects to track for each object.**

is long enough to span several reads, then only $\frac{1}{R\cdot t}$ of the client's reads will require network messages (see Table 1). On the other hand, long timeouts increase the likelihood that caches will supply stale data to applications. Gwertzman and Seltzer [7] show that for web browser workloads, even for a timeout of ten days, server load is significantly higher than under the *Callback* algorithm described below. The same study finds that an adaptive timeout scheme works better than static timeouts, but that when the algorithm's parameters are set to make the adaptive timeout algorithm impose the same server load as *Callback*, about 4% of client reads receive stale data.

If servers can predict with certainty when objects will be modified, then *Poll* is ideal. In this case, servers can tell clients to use cached copies of objects until the time of the next modification. For this study, we do not assume that servers have such information about the future.

## 2.3 Callback

In a *Callback* algorithm [8, 12], servers keep track of which clients are caching which objects. Before modifying an object, a server notifies the clients with copies of the object and does not proceed with the modification until it has received an acknowledgment from each client. As shown in Table 1, *Callback*'s read cost is low because a client is guaranteed that a cached object is valid until told otherwise. However, the write cost is high because when an object is modified the server invalidates the cached objects, which may require up to $C_{tot}$ messages. Furthermore, if a client has crashed or if a network partition separates a server from a client, then a write may be delayed indefinitely.

## 2.4 Lease

To address the limitations of *Callback*, Gray and Cheriton proposed *Lease* [5]. To read an object, a client first acquires a *lease* for it with an associated timeout $t$. The client may then read the cached copy until the lease expires. When an object is modified, the object's server invalidates the cached objects of all clients whose leases have not expired. To read the object after the lease expires, a client first contacts the server to renew the lease.

*Lease* allows servers to make progress while maintaining strong consistency despite failures. If a client or network failure prevents a server from invalidating a client's cache, the server need only wait until the lease expires before performing the write. By contrast, *Callback* may force the write to wait indefinitely.

Leases also improve scalability of writes. Rather than contacting all clients that have ever read an object, a server need only contact recently active clients that hold leases on that object. Leases can thus reduce the amount of state that the server maintains to track clients, as well as the cost of sending invalidation messages [10]. Servers may also choose to invalidate caches by simply waiting for all outstanding leases to expire rather than by sending messages to a large number of clients; we do not explore this option in this study. *Lease* presents a tradeoff similar to the one offered by *Poll*. Long leases reduce the cost of reads by amortizing each lease renewal over $R \cdot t$ reads. On the other hand, short leases reduce the delay on writes when failures occur.

As with polling, a client that is unable to contact a server to renew a lease knows that it holds potentially stale data. The client may then take application-specific actions, such as signaling an error or returning the suspect data along with a warning. However, unlike *Poll*, *Lease* never lets clients believe that stale objects are valid.

## 3 Volume leases

Traditional leases provide good performance when the cost of renewing leases is amortized over many reads. Unfortunately, for many WAN workloads, reads of an object may be spread over seconds or minutes, requiring long leases in order to amortize the cost of renewals [7]. To make leases practical for these workloads, our algorithms use a combination of *object leases*, which are associated with individual data objects, and *volume leases*, which are associated with a collection of related objects on the same server. In our scheme a client reads data from its cache only if both its object and volume leases for that data are valid, and a server can modify data as soon as either lease has expired. By making object leases long and volume short, we overcome the limitations of traditional leases: long object leases have low overhead, while short volume leases allow servers to modify data without long delays. Furthermore, if there is spatial locality within a volume, the overhead of renewing short leases on volumes is amortized across many objects. This section first describes the *Volume Leases* algorithm and then examines a variation called *Volume Leases with Delayed Invalidations*.

### 3.1 The basic algorithm

Figures 2, 3, and 4 show the data structures used by the *Volume Leases* algorithm, the server side of the algorithm, and the client side of the algorithm, respectively. The basic algorithm is simple.

**Reading Data.** Clients read cached data only if they hold valid object and volume leases on the corresponding objects. Expired leases are renewed by contacting the appropriate servers. When granting a lease for an object $o$ to a client $c$, if $o$ has been modified since the last time $c$ held a valid lease on $o$ then the server piggybacks the current data on the lease renewal.

**Writing Data.** Before modifying an object, a server sends invalidation messages to all clients that hold valid leases on the object. The server delays the write until it receives acknowledgments from all clients, or until the volume or object leases expire. After modifying the object, the server increments the object's version number.

#### 3.1.1 Handling unreachable clients

Client crashes or network partitions can make some clients temporarily unreachable, which may cause problems. Consider the case of an unreachable client whose volume lease has expired but that still holds a valid lease on an object. When the client becomes reachable and attempts to renew its volume lease, the server must invalidate any modified objects for which the client holds a valid object lease. Our algorithm thus maintains at each server an *Unreachable* set that records the clients that have not acknowledged, within some timeout period, some of the server's invalidation messages.

After receiving a read request or a lease renewal request from a client in its Unreachable set, a server removes the client from its Unreachable set, renews the client's volume lease, and notifies the client to renew its leases on any currently cached objects belonging to that volume. The client then responds by sending a list of objects along with their version numbers, and the server replies with a message that contains a vector of object identifiers. This message (1) renews the leases of any objects not modified while the client was unreachable and (2) invalidates the leases of any objects whose version number changed while the client was unreachable.

Data Structures

| | |
|---|---|
| **Volume** | A volume v has the following attributes |
| id | = unique identifier |
| objects | = set of objects in v |
| epoch | = volume epoch number (incremented on server reboot) |
| expire | = time by which all current leases on v will have expired |
| at | = set of $\langle client, expire\rangle$ of valid leases on v |
| unreachable | = set of clients whose volume leases have expired and who may have missed object invalidation messages |
| | |
| **Object** | An object o has the following attributes |
| id | = unique identifier |
| data | = the object's data |
| version | = version number |
| expire | = time by which all current leases on o will have expired |
| at | = set of $\langle client, expire\rangle$ of valid leases on o |
| volume | = volume |

**Figure 2: Data Structures for Volume Lease algorithm.**

#### 3.1.2 Handling server failures

When a server fails we assume that the state used to maintain cache consistency is lost. In LAN systems, servers often reconstruct this state by polling their clients [12]. This approach is impractical in a WAN, so our protocol allows a server to incrementally construct a valid view of the object lease state, while relying on volume lease expiration to prevent clients from using leases that were granted by a failed server. To recover from a crash, a server first invalidates all volume leases by waiting for them to expire. This invalidation can be done in two ways. A server can save on stable storage the latest expiration time of any volume lease. Then, upon recovery, it reads this timestamp and delays all writes until after this expiration time. Alternatively, the server can save on stable storage the duration of the longest possible volume lease. Upon recovery, the server then delays any writes until this duration has passed.

Since object lease information is lost when a server crashes, the server effectively invalidates all object leases

```
**Server writes object** $o$
  **for all** $\langle client, expire \rangle \in o.at$
    **if** $expire > currentTime \wedge client \notin o.volume.unreachable$
        $To\_contact \leftarrow To\_contact \cup client$
  **send**($INVALIDATE, o.id$) **to** all clients in $To\_contact$
  $T_f \leftarrow$ min($o.volume.expire$, $o.expire$)
  **if** $T_f < msgTimeout$
    $T_f \leftarrow msgTimeout$
  **while** ($T_f \geq currentTime$) and ($To\_contact \neq \emptyset$) **do**
    **receive**($ACK\_INVALIDATE, o.id$) **from** $c \in To\_contact$
    $To\_contact \leftarrow To\_contact - \{$ c $\}$
  $o.volume.unreachable \leftarrow o.volume.unreachable \cup \{To\_contact\}$
  $o.at \leftarrow \emptyset$
  $o.version \leftarrow o.version + 1$
  **write** $o$

**Server grants lease for object** $o$ **with** $o.id = objId$
  **receive**($REQ\_OBJ\_LEASE, objId, version$) **from** $c$
  let $o$ be the object such that $o.id = objId$
  $o.expire \leftarrow currentTime + objLeaseTimeout$
  $o.at \leftarrow o.at - \{\langle client, X \rangle\}$ // delete old leases for client
  $o.at \leftarrow o.at \cup \{\langle client, o.expire \rangle\}$
  **if** ($o.version > clientVersion$) **then**
    **send**($OBJ\_LEASE, o.id, o.version, o.expire, o.data$)
  **else if**($o.version = clientVersion$) **then**
    **send**($OBJ\_LEASE, o.id, o.version, o.expire$)

**Server grants lease for volume** $v$ **with** $v.id = volId$
  **receive**($REQ\_VOL\_LEASE, volId, volEpoch$) **from** $c$
  let $v$ be the volume such that $v.id = volId$
  **if** ($c \in v.unreachable$) or ($v.epoch > volEpoch$) **then**
    recoverUnreachableClient($c$, $v$) // see below
  **if** $c \notin v.unreachable$
    $v.expire \leftarrow currentTime + volumeLeaseTimeout$
    $v.at \leftarrow v.at - \{\langle client, X \rangle\}$ // delete old leases for client
    $v.at \leftarrow v.at \cup \{\langle client, v.expire \rangle\}$
    **send**($VOL\_LEASE, v.id, v.expire, v.epoch$)

**Server re-establishes contact with unreachable client** $c$ **for volume** $v$
recoverUnreachableClient($c$, $v$)
  **send**($MUST\_RENEW\_ALL, v.id$) **to** $c$
  **receive**($RENEW\_OBJ\_LEASES, volId, leaseSet$) **from** $c$
  **for all** $\langle objId, objVersion \rangle \in leaseSet$ **do**
    let $o$ be the object such that $o.id = objId$
    **if** ($o.version > objVersion$) **then**
        $invalList \leftarrow invalList \cup \{objId\}$
        $o.at \leftarrow o.at - \{\langle c, X \rangle\}$ // delete old leases for client
    **else**
        $o.expire \leftarrow currentTime + objLeaseTimeout$
        $renewList \leftarrow renewList \cup \langle o.id, o.version, o.expire \rangle$
        $o.at \leftarrow o.at - \{\langle c, X \rangle\}$ // delete old leases for client
        $o.at \leftarrow o.at \cup \{\langle c, o.expire \rangle\}$
  **send**($INVALIDATE, invalList, RENEW, renewList$)
  $T_f = currentTime + msgTimeout$
  **while** ($T_f \geq currentTime$) and ($c \in v.unreachable$)
    **receive** ($ACK\_INVALIDATE$) from $c$
    $v.unreachable \leftarrow v.unreachable - \{c\}$
```

**Figure 3: The Volume Leases Protocol (Server Side).**

```
**Client reads object** $o$
  **if** $validLease(o.volume) \wedge validLease(o.id)$ **then**
    **read** local copy of $o$
  **if** $validLease(o.volume) \wedge \neg validLease(o.id)$ **then**
    request lease for object $o$
    **read** local copy of $o$
  **if** $\neg validLease(o.volume) \wedge validLease(o.id)$ **then**
    request lease for volume $v$
    **read** local copy of $o$
  **if** $\neg validLease(o.volume) \wedge \neg validLease(o.id)$ **then**
    request lease for volume $v$ and object $o$
    **read** local copy of $o$

**Client** $c$ **requests lease for object** $o$
  $vnum \leftarrow \max(o.version, -1)$
  **send**($REQ\_OBJ\_LEASE, o.id, vnum$) **to** server
  **receive**($OBJ\_LEASE, o.version, o.expire[, o.data]$) from server

**Client** $c$ **requests lease for volume** $v$
  $epoch \leftarrow \max(v.epoch, -1)$
  **send**($reqVolLease, v.id, epoch$) **to** server
  **if receive**($MUST\_RENEW\_ALL, v.id$) **from** server **then**
    $leaseSet \leftarrow \emptyset$
    **for all** objects $o$ for which $((o.volume = v) \wedge (o.expires < currentTime))$
        $leaseSet \leftarrow leaseSet \cup \langle o.id, o.version \rangle$
    **send**($RENEW\_OBJ\_LEASES, v.id, leaseSet$) to server
    **receive** ($INVALIDATE, invalList, RENEW, renewList$) from server
    **for all** $objId \in invalList$
        let $o$ be the object for which $o.id = objId$
        $o.expire = -1$; delete $o.data$; $o.data \leftarrow NULL$
    **for all** $\langle objId, version, expire \rangle \in renewList$
        let $o$ be the object for which $o.id = objId$
        assert($o.version = version$)
        $o.expire \leftarrow expire$
    **send**($ACK\_INVALIDATE, v.id$) to server
    **receive**($VOL\_LEASE, v.id, v.expire, v.epoch$) from server

**Client receives object invalidation message for object** $o$
  **receive**($INVALIDATE, objId$) from server
  let $o$ be the object for which $o.id = objId$
  $o.expire = -1$; delete $o.data$; $o.data \leftarrow NULL$
  **send**($ACK\_INVALIDATE, o.id$) **to server**

**validLease(lease** $l$**)**
  **if** $l.expire > currentTime$
    return TRUE
  **else**
    return FALSE
```

**Figure 4: The Volume Leases Protocol (Client Side).**

by treating all clients as if they were in the Unreachable set. It does this by maintaining a volume epoch number that is incremented with each reboot. Thus, all client requests to renew a volume must also indicate the last epoch number known to the client. If the epoch number is current, then volume lease renewal proceeds normally. If the epoch number is old, then the server treats the client as if the client were in the volume's Unreachable set.

It is also possible to store the cache consistency information on stable storage [3, 6]. This approach reduces recovery time at the cost increased overhead on normal lease renewals. We do not investigate this approach in this paper.

#### 3.1.3 The cost of volume leases.

To analyze *Volume Leases*, we assume that servers grant leases of length $t_v$ on volumes and of length $t$ on objects. Typically, the volume lease is much shorter than the object leases, but when a client accesses multiple objects from the same volume in a short amount of time, the volume lease is likely to be valid for all of these accesses. As the read cost column of Table 1 indicates, the cost of a typical read, measured in messages per read, is $\frac{1}{\sum_{o \in V}(R_o t_v)} + \frac{1}{R \cdot t}$. The first term reflects the fact that the volume lease must be renewed every $t_v$ seconds but that the renewal is amortized over all objects in the volume, assuming that object $o$ is read $R_o$ times per second. The second term is the standard cost of renewing an object lease. As the *ack wait delay* column indicates, if a client or network failure prevents a server from contacting a client, a write to an object must be delayed for $min(t, t_v)$, *i.e.*, until either lease expires. As the *write cost* and *server state* columns indicate, servers track all clients that hold valid object leases and notify them all when objects are modified. Finally, as the *stale time* columns indicate, *Volume Leases* never supplies stale data to clients.

### 3.2 Volume leases with delayed invalidations

The performance of *Volume Leases* can be improved by recognizing that once a volume lease expires, a client cannot use object leases from that volume without first contacting the server. Thus, rather than invalidating object leases immediately for clients whose volume leases have expired, the server can send invalidation messages when (and if) the client renews the volume lease. In particular, the *Volume Leases with Delayed Invalidations* algorithm modifies *Volume Leases* as follows. If the server modifies an object for which a client holds a valid object lease but an expired volume lease, the server moves the client to a per-volume *Inactive* set, and the server appends any object invalidations for inactive clients to a per-inactive-client *Pending Message* list. When an inactive client renews a volume, the server sends all pending messages to that client and waits for the client's acknowledgment before renewing the volume. After a client has been inactive for $d$ seconds, the server moves the client from the Inactive set to the Unreachable set and discards the client's Pending Message list. Thus, $d$ limits the amount of state stored at the server. Small values for $d$ reduce server state but increase the cost of re-establishing volume leases when unreachable clients become reconnected.

As Table 1 indicates, when a write occurs, the server must contact the $C_v$ clients that hold valid volume leases rather than the $C_o$ clients that hold valid object leases. Delayed invalidations provide three advantages over *Volume Leases*. First, server writes can proceed faster because many invalidation messages are delayed or omitted. Second, the server can batch several object invalidation messages to a client into a single network message when the client renews its volume lease, thereby reducing network overhead. Third, if a client does not renew a volume for a long period of time, the server can avoid sending the object invalidation messages by moving the client to the Unreachable set and using the reconnection protocol if the client ever returns.

## 4 Methodology

To examine the algorithms' performance, we simulated the algorithms discussed in Table 1 under a workload based on web trace data.

### 4.1 Simulator

We simulate a set of servers that modify files and provide files to clients, and a set of clients that read files. The simulator accepts timestamped read and modify events from input files and updates the cache state. The simulator records the size and number of messages sent by each server and each client, as well as the size of the cache consistency state maintained at each server.

We validated the simulator in two ways. First, we obtained Gwertzman and Seltzer's simulator [7] and one of their traces, and compared our simulator's results to theirs for the algorithms that are common between the two studies. Second, we used our simulator to examine our algorithms under simple synthetic workloads for which we could analytically compute the expected results. In both cases, our simulator's results match the expected results.

**Limitations of the simulator.** Our simulator makes several simplifying assumptions. First, it does not simulate concurrency—it completely processes each trace event before processing the next one. This simplification allows us to ignore details such as mutual exclusion on internal data structures, race conditions, and deadlocks. Although this could change the messages that are sent (if, for instance, a file is read at about the same time it is written), we do not believe that simulating these details would significantly affect our performance results.

Second, we assume infinitely large caches. Thus, clients experience no capacity cache misses, and we do not simulate server disk accesses. Both of these effects reduce potentially significant sources of work that are the same across algorithms. Thus, our results will magnify the differences among the algorithms. Infinite client caches might also reduce an advantage of short leases and polling: a server may send an invalidation to a client for an object the client has already discarded. Short leases and client polling may reduce these unnecessary messages.

Finally, we assume that the system maintains cache consistency on entire files rather than on some finer granularity. We chose to examine whole-file consistency because this is currently the most common approach for WAN workloads [1]. Fine-grained consistency may reduce the amount of data traffic, but it also increases the number of control messages required by the consistency algorithm. Thus, fine-grained cache consistency would likely increase the relative differences among the algorithms.

## 4.2 Workload

We use a workload based on traces of HTTP accesses at Boston University [4]. These traces span four months during January 1995 through May 1995 and include all HTTP accesses by Mosaic browsers—including local cache hits—for 33 SPARCstations.

Although these traces contain detailed information about client reads, they do not indicate when files are modified. We therefore synthesize writes to the objects using a simple model based on two studies of write patterns for web pages. Bestavros [2] examined traces of the Boston University web server, and Gwertzman and Seltzer [7] examined the write patterns of three university web servers. Both studies concluded that few files change rapidly, and that globally popular files are less likely to change than other files. For example, Gwertzman and Seltzer's study found that 2%–23% of all files were *mutable* (each file had a greater than 5% chance of changing on any given day) and 0%–5% of the files were *very mutable* (had greater than 20% chance of changing during a 24-hour period).

Based on these studies, our synthetic write workload divides the files in the trace into four groups. We give the 10% most referenced files a low average number of random writes per day (we use a Poisson distribution with an expected number of writes per day of 0.005). We then randomly place the remaining 90% of the files into three sets. The first set, which includes 3% of all files in the trace, are *very mutable* and have an expected number of writes per day of 0.2. The second set, 10% of all files in the trace, are *mutable* and have an expected number of writes per day of 0.05. The remaining 77% of the files have an expected number of writes per day of 0.02.

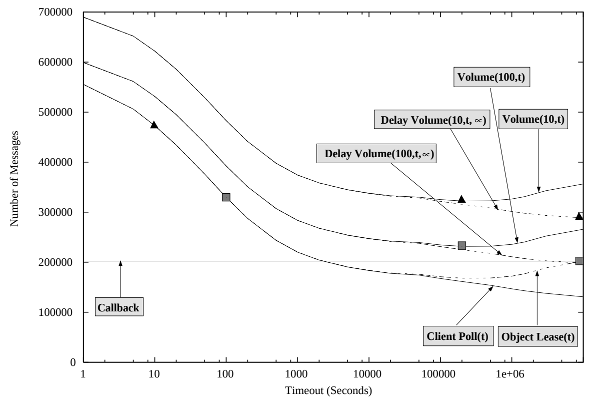

**Figure 5: Number of messages vs. timeout length.**

We simulate the 1000 most frequently accessed servers; this subset of the servers accounts for more than 90% of all accesses in the trace. Our workload consists of 1,034,077 reads of 68,665 different files plus 209,461 artificially generated writes to those files. The files in the workload are grouped into 1000 volumes corresponding to the 1000 servers. We leave more sophisticated grouping as future work.

## 5 Simulation results

This section presents simulation results that compare the volume algorithms with other consistency schemes. In interpreting these results, remember that the trace workload tracks the activities of a relatively small number of clients. In reality, servers would be accessed by many other clients, so the absolute values we report for server and network load will are lower than the servers would actually experience. Instead of focusing on the absolute numbers in these experiments, we focus on the relative performance of the algorithms under this workload.

### 5.1 Server/network load

Figure 5 shows the performance of the algorithms. The x-axis, which uses a logarithmic scale, gives the timeout length, $t$, in seconds, while the y-axis gives the numbers of messages sent between the client and servers. For *Volume Lease*, $t$ refers to the object lease timeout and not the volume lease timeout; we show different volume lease timeouts with different lines. The line for *Callback* is flat because *Callback* invalidates all cached copies regardless of $t$. The *Lease*, and basic *Volume Lease* lines decline until $t$ reaches about 100,000 seconds and then rise slightly.

This shape comes from the competing influence of two factors. As $t$ rises, the number of lease renewals by clients declines, but the number of invalidations sent to clients holding valid leases increases. For this workload, once a client has held an object for 100,000 seconds, it is more likely that the server will modify the object than that the client will read it, so leases shorter than this reduce system load. *Delayed Invalidation* and *Client Poll* algorithm send strictly fewer messages as $t$ increases because *Delayed Invalidation* avoids sending invalidations to clients that are no longer accessing a volume even if the clients hold valid object leases and because *Client Poll* never sends invalidation messages. Note that for timeouts of 100,000 seconds, *Client Poll* results in clients accessing stale data on about 1% of all reads, and for timeout values of 1,000,000 seconds, the algorithm results in clients accessing stale copies on about 5% of all reads.

The separation of the *Lease*, *Volume(*$10, t$*)*, and *Volume(*$100, t$*)* lines shows the additional overhead of maintaining volume leases. Shorter volume timeouts increase this overhead. *Lease* can be thought of as the limiting case of infinite-length volume leases.

Although *Volume Leases* imposes a significant overhead compared to *Leases* for a given value of $t$, applications that care about fault tolerance can achieve better performance with *Volume Leases* than without. For example, the triangles in the figure highlight the best performance achievable by a system that does not allow writes to be delayed for more than 10 seconds for *Lease*, *Volume(*$10, t$*)*, and *Delayed Invalidations(*$10, t, \infty$*)*. *Volume(10, 100000)* sends 32% fewer messages than *Lease(10)*, and *Delayed Invalidations(10, $10^7$, $\infty$)* sends 39% fewer messages than the basic object lease algorithm. Similarly, for applications that can delay writes at most 100 seconds, *Volume Lease* outperforms *Lease* by 30% and *Delayed Invalidations* outperforms the lease algorithm by 40% as indicated by the squares in the figure.

Although providing strong consistency is more expensive than the *Poll* algorithm, the cost appears tolerable for many applications. For example, *Poll(100000)* uses about 15% fewer messages than *Delayed Invalidations(*$100, 10^7$, $\infty$*)*, but it supplies stale data to clients on about 1% of all reads. Even in the extreme case of *Poll(*$10^7$*)* (in which clients see stale data on over 35% of reads), *Delayed Invalidations* uses less than twice as many messages as the polling algorithm.

Although space limitations do not allow us to include the graphs here, we also examined the network bytes sent by these algorithms and the server CPU load imposed by these algorithms. By both of these metrics, the difference in cost of providing strong consistency compared to *Poll* was smaller than by the metric of network messages. The relative differences among the lease algorithms was also smaller for these metrics than for the network messages metric for the same reasons.

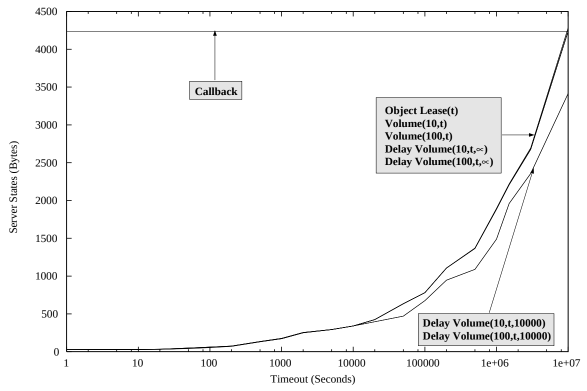

**Figure 6: State at the most popular server vs. timeout.**

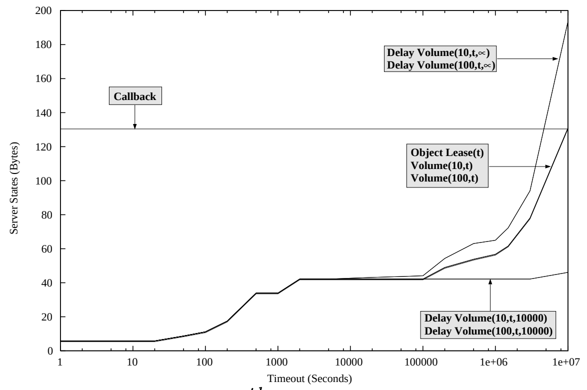

**Figure 7: State at the 10$^{th}$ most popular server vs. timeout.**

## 5.2 Server state

Figures 6 and 7 show the amount of server memory required to implement the algorithms. The first shows the requirements at the trace's most heavily loaded server, and the second shows the demand at the trace's tenth most heavily loaded server. The x-axis shows the timeout in seconds using a log scale. The y-axis is given in bytes and represents the average number of bytes of memory used by the server to maintain consistency state. We charge the servers 16 bytes to store an object or volume lease or callback record. For messages queued by the Delay algorithm, we also charge 16 bytes.

For short timeouts, the lease algorithms use less memory than the callback algorithm because the lease algorithms discard callbacks for inactive clients. Compared to standard leases, *Volume Leases* increase the amount of state needed at servers, but this increase is small because volume leases are short, so servers generally maintain few active volume leases. If the *Delay* algorithm never moves clients to the Unreachable set it may store messages destined for inactive clients for a long time and use more memory than the other algorithms. Conversely, if *Delay* uses a short $d$

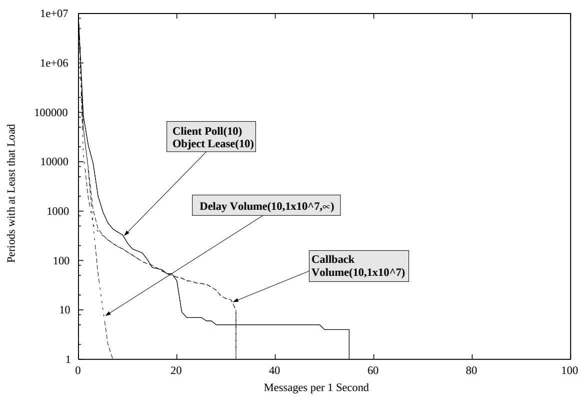

**Figure 8: Periods of heavy server load under default workload for the most heavily loaded server.**

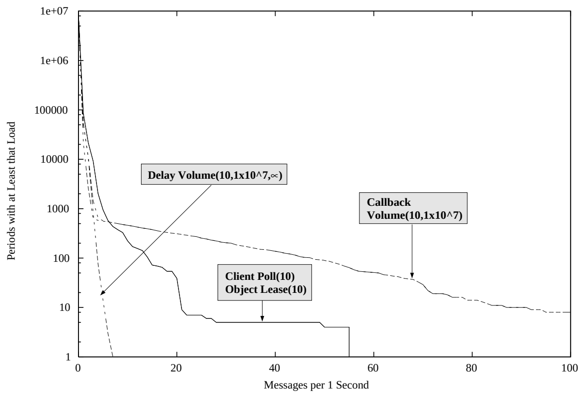

**Figure 9: Periods of heavy server load under "bursty write" workload for the most heavily loaded server.**

parameter so that it can move clients from the Inactive set to the Unreachable set and discard their pending messages and callbacks, *Delay* can use less state than the other lease or callback algorithms. Note that running *Delay* with short discard times will increase server load and the number of consistency messages. We have not yet quantified this effect because it will depend on implementation details of the reconnection protocol.

### 5.3 Bursts of load

Figure 5.3 shows a cumulative histogram in which the y value, shown in log scale, counts the number of 1-second periods in which the load at the server was at least x messages sent or received per second. There are three groups of lines. *Client Poll* and *Object Lease* both use short timeouts, so when clients read groups of objects from a server, these algorithms send groups of object renewal messages to the server. *Callback* and *Volume* use long object lease periods, so read traffic puts less load on the server, but writes result in bursts of load when popular objects are modified. For this workload, peak loads correspond to bursts of about one message per client. Finally, *Delay* uses long object leases to reduce bursts of read traffic from clients accessing groups of objects, and it delays sending invalidation messages to reduce bursts of traffic when writes occur. This combination reduces the peak load on the server for this workload.

For the experiment described in the previous paragraph, *Client Poll* and *Object Lease* have periods of higher load than *Callback* and *Volume* for two reasons. First, the system shows performance for a modest number of clients. Larger numbers of clients would increase the peak invalidate load for *Callback* and *Volume*. For *Client Poll* and *Object Lease*, increasing the number of clients would increase peak server load less dramatically because read requests from additional clients would be more spread out in time. The second reason for *Callback* and *Volume*'s advantage in this experiment is that clients in the trace read data from servers in bursts, but writes to volumes are not bursty in that a write to one object in a volume does not make it more likely that another object from the same volume will soon be modified. Conversely, Figure 9 shows a "bursty write" workload in which when one object is modified, we select $k$ other objects from the same volume to modify at the same time. For this graph, we compute $k$ as a random exponential variable with a mean of 10. This workload significantly increases the bursts of invalidation traffic for *Volume* and *Callback*.

## 6 Related work

Our study builds on efforts to assess the cost of strong consistency in wide area networks. Gwertzman and Seltzer [7] compare cache consistency approaches through simulation, and conclude that protocols that provide weak consistency are the most suitable to a Web-like environment. In particular, they find that an adaptive version of *Poll(t)* exerts a lower server load than an invalidation protocol if the polling algorithm is allowed to return stale data 4% of the time. We arrive at different conclusions. In particular, we observe that much of the apparent advantage of weak consistency over strong consistency in terms of network traffic comes from clients reading stale data [10]. Also, we use volume leases to address many of the challenges to strong consistency.

We also build on the work of Liu and Cao [10], who use a prototype server invalidation system to evaluate the overhead of maintaining consistency at the servers compared to client polling. They also study ways to reduce server state via per-object leases. As with our study, their workload is based on a trace of read requests and synthetically-generated write requests. Our work differs primarily in our treatment of fault tolerance issues. In particular, after a server recovers our algorithm uses volume timeouts to "notify" clients that they must contact the server to renew leases; Liu and Cao's algorithm requires the server to send messages to all clients that might be caching objects from the server. Also, our volume leases provide a graceful way to handle network partitions; when a network failure occurs, Liu and Cao's algorithm must periodically retransmit invalidation messages, and it does not guarantee strong consistency in that case.

Cache consistency protocols have long been studied for distributed file systems [8, 12, 13]. Several aspects of Coda's [9] consistency protocol are reflected in our algorithms. In particular, our notion a volume is similar to that used in Coda [11]. However, ours differsin two key respects. First, Coda does not associate volumes with leases, and relies instead on other methods to determine when servers and clients become disconnected. The combination of short volume leases and long object leases is one of our main contributions. Second, because Coda was designed for different workloads, its design trade-offs are different. For example, because Coda expects clients to communicate with a small number of servers and it regards disconnection as a common occurrence, Coda aggressively attempts to set up volume callbacks to all servers on each hoard walk (every 10 minutes).

## 7 Conclusions

We have taken three cache consistency algorithms that have been previously applied to file systems and quantitatively evaluated them in the context of Web workloads. In particular, we compared a Poll algorithm with a timeout, the Callback algorithm in which a server invalidates before each write, and Gray and Cheriton's Lease algorithm. The Lease algorithm presents a tradeoff similar to the one offered by the Poll algorithm. On the one hand, long leases reduce the cost of reads by amortizing each lease renewal over many reads. On the other hand, short leases reduce the delay on writes when a failure occurs. To solve this problem, we have introduced the Volume Lease, Volume Lease with Delayed Invalidation, and Best Effort Lease algorithms that allow servers to perform writes with minimal delay, while minimizing the number of messages necessary to maintain consistency. Our simulations confirm the benefits of these algorithm.

## Acknowledgments

We thank James Gwertzman and Margo Seltzer for making their simulator available for us to use to validate our simulator. We thank Carlos Cunha, Azer Bestavros and Mark Crovella for making the BU web traces available to us. And we thank the program committee and the anonymous reviewers for their valuable feedback.

## References

[1] T. Berners-Lee, R. Fielding, and H. Frystyk Nielsen. Hypertext Transfer Protocol – HTTP/1.0. Internet Draft draft-ietf-http-v10-spec-00, Internet Engineering Task Force, March 1995.

[2] A. Bestavros. Speculative Data Disseminatino and Service to Reduce Server Load, Network Traffic, and Service Time in Distributed Information Systems. In *International Conference on Data Engineering*, March 1996.

[3] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.

[4] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Traces. Technical Report TR-95-010, Boston University Department of Computer Science, April 1995.

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

[6] James N. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, and G. Seegmueller, editors, *Operating Systems: An Advanced Course*, pages 393–481. 1977. Lecture Notes on Computer Science 60.

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

[8] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[9] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

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

[11] L. Mummert and M. Satyanarayanan. Large Granularity Cache Coherence for Intermittent Connectivity. In *Proceedings of the Summer 1994 USENIX Conference*, June 1994.

[12] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[13] V. Srinivasan and J. Mogul. Spritely NFS: Experiments with Cache Consistency Protocols. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 45–57, December 1989.