Volume Leases for Consistency in Large-Scale Systems*

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

Abstract

This article introduces volume leases *as a mechanism for providing server-driven cache consistency for large-scale, geographically distributed networks. Volume leases retain the good performance, fault tolerance, and server scalability of the semantically weaker client-driven protocols that are now used on the web. Volume leases are a variation of object leases, which were originally designed for distributed file systems. However, whereas traditional object leases amortize overheads over long lease periods, volume leases exploit spatial locality to amortize overheads across multiple objects in a volume. This approach allows systems to maintain good write performance even in the presence of failures. Using trace-driven simulation, we compare three volume lease 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

To fulfill the promise of an environment in which essentially all human knowledge is available from a set of servers distributed across wide area networks, the data infrastructure must evolve from protocols optimized for one application—browsers—to protocols that support a range of more demanding applications. In the future, we expect data-intensive applications to extend beyond human-driven browsers to include program-driven agents, robots, distributed databases, and data miners that will place new demands on the data-distribution infrastructure. These new applications will require aggressive caching for acceptable performance, and they will not be as tolerant of cache inconsistencies as a browser. Unfortunately, current cache consistency protocols do not scale to large systems such as the web because of poor performance, weak consistency guarantees, or poor fault tolerance.

*This article is to appear in the *IEEE Transactions on Knowledge and Data Engineering* Special Issue on Web Technologies. 1999. This work was funded in part by a NSF CISE grant (CDA-9624082), gifts from Novell and Sun Microsystems, and DARPA/SPAWAR grant number N66001-98-8911. Dahlin and Alvisi were supported by NSF CAREER awards (CCR-9733842) and (CCR-9734185), respectively.

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.

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 cache 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 browsers account for weak consistency through a human-based error-correction 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 the use of 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 [8], which specify a length of time during which servers notify clients of modifications to cached data. After a lease's timeout expires, a client must renew the lease by sending a message to the server before the client 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 writes 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, 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 article, we show how *volume leases* [22] 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.

We examine three variations of volume leases: volume leases, volume leases with delayed invalidations, and best effort volume leases. In the delayed invalidations algorithm, servers defer sending object invalidation messages to clients whose volume leases have expired. This optimization reduces peaks in server load, and it can reduce overall load by batching invalidation messages and eliminating messages entirely in cases when clients never renew a volume lease. The third variation is motivated by the observation that some workloads do not require strict consistency but do prefer that clients observe fresh data. For example, when an important event occurs, a news service would like to invalidate stale cached copies of their front page quickly, but they may want to begin distributing the new front page immediately rather than wait until they have notified all customers that the old page is invalid. The best effort variation of volume leases uses relaxed consistency to satisfy such applications. We find that this approach can improve performance by allowing servers to utilize longer volume lease timeouts.

This article evaluates the performance of volume leases using trace-based simulation. We compare the volume 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 article 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.

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 algorithm discussed in this paper, including our three new algorithms. We also refer to Figure 1, which defines several parameters of the algorithms.

In Table 1, we summarize 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

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

Table 1: Summary of algorithm performance.

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	Number of active objects per volume
C_{tot}	Number of clients with a copy of object o
C_o	Number of clients with lease on object o
C_v	Number of clients with lease on volume v
C_d	Number of clients whose volume leases expired less than d seconds ago.
$size(x)$	bytes of server state to support x clients

Figure 1: Definition of parameters in Table 1

expects to track for each object.

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 about 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 poor 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 [11].

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 to *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 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 [10] 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 [11, 17], 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* [8]. 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 [14]. 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 [10]. 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*. At the end of this section, we examine *Best Effort Volume Leases* to support applications where timely updates are desired, but not required.

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—one 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 [17]. 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 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

```
Server writes object o
  for all ⟨client, expire⟩ ∈ o.at
    if expire > currentTime ∧ client ∉ o.volume.unreachable
      To_contact ← To_contact ∪ client
  send(INVALIDATE, o.id) to all clients in To_contact
  T_f ← min(o.volume.expire, o.expire)
  if T_f < msgTimeout
    T_f ← msgTimeout
  while (T_f ≥ currentTime) and (To_contact ≠ ∅) do
    receive(ACK_INVALIDATE, o.id) from c ∈ To_contact
    To_contact ← To_contact − { c }
  o.volume.unreachable ← o.volume.unreachable ∪ {To_contact}
  o.at ← ∅
  o.version ← o.version + 1
  write o

Server renews client lease
  receive(RENEW_LEASE_REQ, volId, volEpoch, objId, clientVersion) from c
  let v be the volume such that v.id = volId
  let o be the object such that o.id = objId
  if (c ∈ v.unreachable) or (v.epoch > volEpoch) then
    v.unreachable ← v.unreachable ∪ c
    recoverUnreachableClient(c, v)                  // see below
  if c ∉ v.unreachable
    v.expire ← currentTime + volumeLeaseTimeout
    v.at ← v.at − {⟨client, X⟩}                     // delete old leases for client
    v.at ← v.at ∪ {⟨client, v.expire⟩}
    o.expire ← currentTime + objLeaseTimeout
    o.at ← o.at − {⟨c, X⟩}                          // delete old leases for client
    o.at ← o.at ∪ {⟨c, o.expire⟩}
    if (o.version > clientVersion) then
      send(RENEW_LEASE_RESP, v.id, v.expire, v.epoch, o.id, o.version, o.expire, o.data)
    else if(o.version = clientVersion) then
      send(RENEW_LEASE_RESP, v.id, v.expire, v.epoch, o.id, o.version, o.expire)

recoverUnreachableClient(client c, volume v)
  send(MUST_RENEW_ALL, v.id) to c
  T_f ← msgTimeout
  renewRecvd ← FALSE
  while (T_f ≥ currentTime) and (¬renewRecvd) do
    receive(RENEW_OBJ_LEASES, volId, leaseSet) from c
    rewnewRecvd ← TRUE
  if (¬renewRecvd) then
    return                                          // client still unreachable
  for all ⟨objId, objVersion⟩ ∈ leaseSet do
    let o be the object such that o.id = objId
    if (o.version > objVersion) then
      invalList ← invalList ∪ {objId}
      o.at ← o.at − {⟨c, X⟩}                        // delete old leases for client
    else
      o.expire ← currentTime + objLeaseTimeout
      renewList ← renewList ∪ ⟨o.id, o.version, o.expire⟩
      o.at ← o.at − {⟨c, X⟩}                        // delete old leases for client
      o.at ← o.at ∪ {⟨c, o.expire⟩}
  send(INVALIDATE, invalList, RENEW, renewList)
  T_f = currentTime + msgTimeout
  while (T_f ≥ currentTime) and (c ∈ v.unreachable)
    receive (ACK_INVALIDATE) from c
    v.unreachable ← v.unreachable − {c}
```

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

Client reads object o
  if $\neg validLease(o.volume) \vee \neg validLease(o.id)$ **then**
    $renewLease(o.volume, o)$
  read local copy of o

renewLease(volume v, object o)
  $epoch \leftarrow \max(v.epoch, -1)$
  $vnum \leftarrow \max(o.version, -1)$
  send$(RENEW\_LEASE\_REQ, v.id, epoch, o.id, vnum)$
  // Note: if any recieve times out, abort the read.
  if receive$(MUST\_RENEW\_ALL, v.id)$ **from** server **then**
    $renewAll(v)$
  // Note: if any recieve times out, abort the read.
  receive$(RENEW\_LEASE\_RESP, v.id, v.expire, v.epoch, o.version, o.expire[, o.data])$ **from** server

renewAll(volume v)
    $leaseSet \leftarrow \emptyset$
    for all objects o for which $((o.volume = v) \wedge (validLease(\mathrm{o}))$
      $leaseSet \leftarrow leaseSet \cup \langle o.id, o.version\rangle$
    send$(RENEW\_OBJ\_LEASES, v.id, leaseSet)$ **to** server
    // Note: if any recieve times out, abort the read.
    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

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

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**

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

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 [5, 9]. This approach reduces recovery time at the cost of 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.

3.1.4 Protocol verification

To verify the correctness of the consistency algorithm, we implemented a variation of the volume leases algorithm described in Figures 3 and 4 using the Teapot system [4]. The Teapot version of the algorithm differs from the one described in the figures in two ways. First, the Teapot version uses a simplified reconnection protocol for Unreachable clients. Rather than restore a client's set of object leases, the Teapot version clears all of the client's object leases when an Unreachable client reconnects. The second difference is that in the Teapot version every network request includes a sequence number that is repeated in the corresponding reply. These sequence numbers allow the protocol to match replies to requests.

Teapot allows us to describe the consistency state machines in a convenient syntax and then to generate Murphi [7] code for mechanical verification. The Murphi system searches the protocol's state space for deadlocks or cases where the system's correcness invariants are violated. Although Murphi's exhaustive search of the state space is an exponential algorithm that only allows us to verify small models of the system, in practice this approach finds many bugs that are difficult to locate by hand and gives us confidence in the correctness of our algorithm [3].

Murphi verifies that the following two invariants hold: (1) when the server writes an object, no client has both a valid object lease and a valid volume lease for that object and (2) when a client reads an object, it has the current version of the object. The system we verified contains one volume with two objects in it, and it includes one client and one server that communicate over a network. Clients and servers can crash at any time, and the network layer can lose messages at any time but cannot deliver messages out of order; the network layer can also report messages lost when they are, in fact, delivered. We have tested portions of the state space for some larger models, but larger models exhaust our test machine's 1 GB of memory before the entire state space is examined.

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.

3.3 Best-effort volume leases

Some applications do not require strong consistency but do want to deliver timely updates to clients. For example, when an important event occurs, a news service would like to invalidate stale copies of their front page quickly rather than wait until all customers know that the old page is invalid. Thus, it is interesting to consider *best-effort* algorithms. A best effort algorithm should always allow writes to proceed immediately, and it should notify clients of writes when doing so does not delay writes.

Any of the volume algorithms may be converted to best effort algorithms by sending invalidations *in parallel* with writes. Table 1 summarizes the characteristics of the best effort version of the *Delayed Invalidations* algorithm. By sending invalidations in parallel with writes, the algorithm limits the expected stale read time to *notify*(C_v)—the time it takes for the server to send the messages—without delaying writes.

Note that in the best effort algorithms, volume leases serve a different purpose than in the original volume algorithms: they limit the time during which clients can see stale data. Whereas strong consistency algorithms generally set the volume lease time (t_v) to be the longest period they are willing to delay a write, this is no longer a factor for best effort algorithms. Instead, these algorithms set t_v to the longest time they will allow disconnected clients to unknowingly see stale data. Since only the disconnected clients are affected by long t_v values, this may allow larger values for t_v than before. For example, a news service using strong consistency might not want to block dissemination of a news update for more than a few seconds, but it may be willing to allow a few disconnected clients to see the old news for several minutes. Thus, such a system might use $t_v = 10\ seconds$ under strong consistency, but it might use $t_v = 10\ minutes$ under a best effort algorithm. As with the original volume algorithms, combining short volume leases with long object leases allows leases to be short while amortizing renewal costs over many objects.

4 Methodology

To examine the algorithms' performance, we simulated each algorithm 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 [10] 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 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.

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 [6]. 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 [10] 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. In section 5.4, we examine the sensitivity of our results to these parameters.

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 977,899 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 are lower than what 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 object timeout length in seconds (t) used by each algorithm, while the y-axis gives the number 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 use different curves to show different volume lease timeouts and indicate the volume lease time (t_v) in the second parameter of the label. For

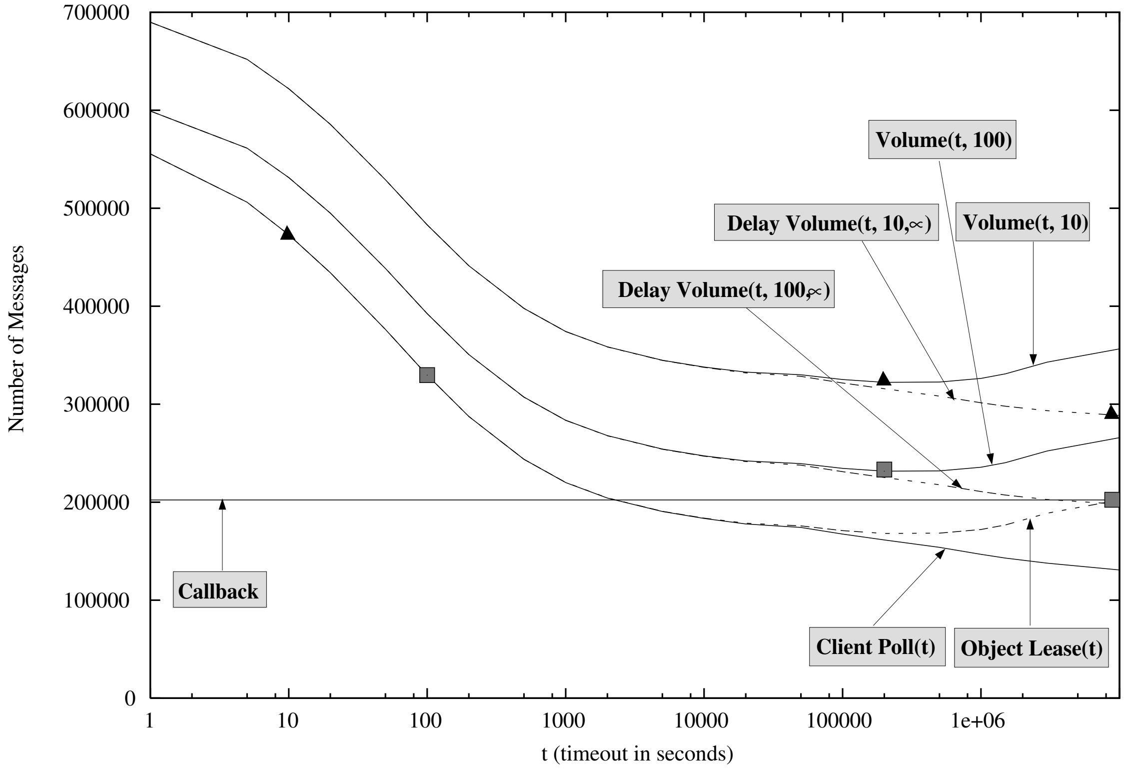

Figure 5: Number of messages vs. timeout length.

the *Delay Volume* lines, we assume an infinite *acknowledgement wait delay* (d) as signified by the third parameter; this means that a server never moves idle clients to the unreachable list. 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 two competing influences. 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. As t increases, *Client Poll* and *Delayed Invalidation* send strictly fewer messages. *Client Poll* never sends invalidation messages, and *Delayed Invalidation* avoids sending invalidations to clients that are no longer accessing a volume, even if the clients hold valid object leases. 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*(t), *Volume*($t, t_v = 10$), and *Volume*($t, t_v = 100$) 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 *Lease* 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*(t),

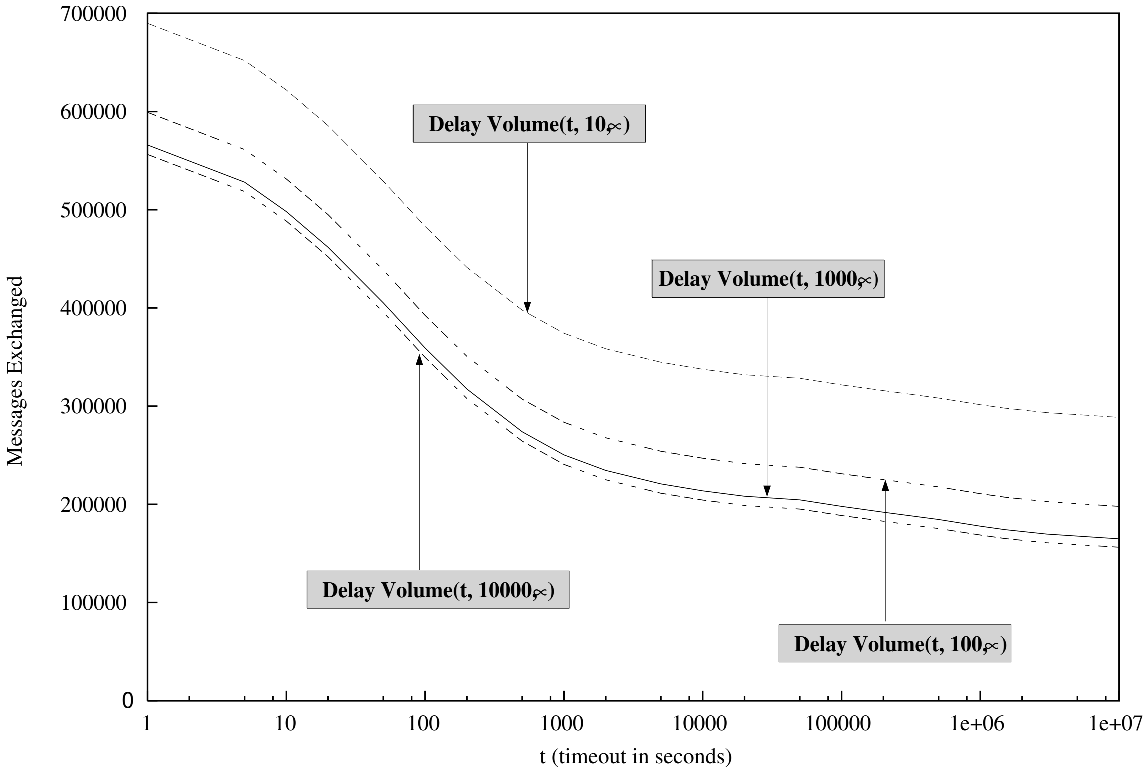

Figure 6: Number of messages vs. timeout length for *Volume Leases with Delayed Invalidates* as volume lease length is varied.

Volume$(t, t_v = 10)$, and *Delayed Invalidations*$(t, t_v = 10, d = \infty)$. *Volume*$(t = 100000, t_v = 10)$ sends 32% fewer messages than *Lease*$(t = 10)$, and *Delayed Invalidations*$(t = 10^7, t_v = 10, d = \infty)$ sends 39% fewer messages than *Lease*$(t = 10)$. Similarly, as indicated by the squares in the figure, 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%.

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

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 the difference 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.

A key advantage of *Best Effort Volume Leases* for applications that permit relaxed consistency is the algorithm may enable longer volume lease timeouts and thus may reduce consistency overhead. Strict consistency algorithms set the volume timeout, t_v, to be the longest tolerable write delay, but the best effort algorithms can set t_v to be the longest time disconnected clients should

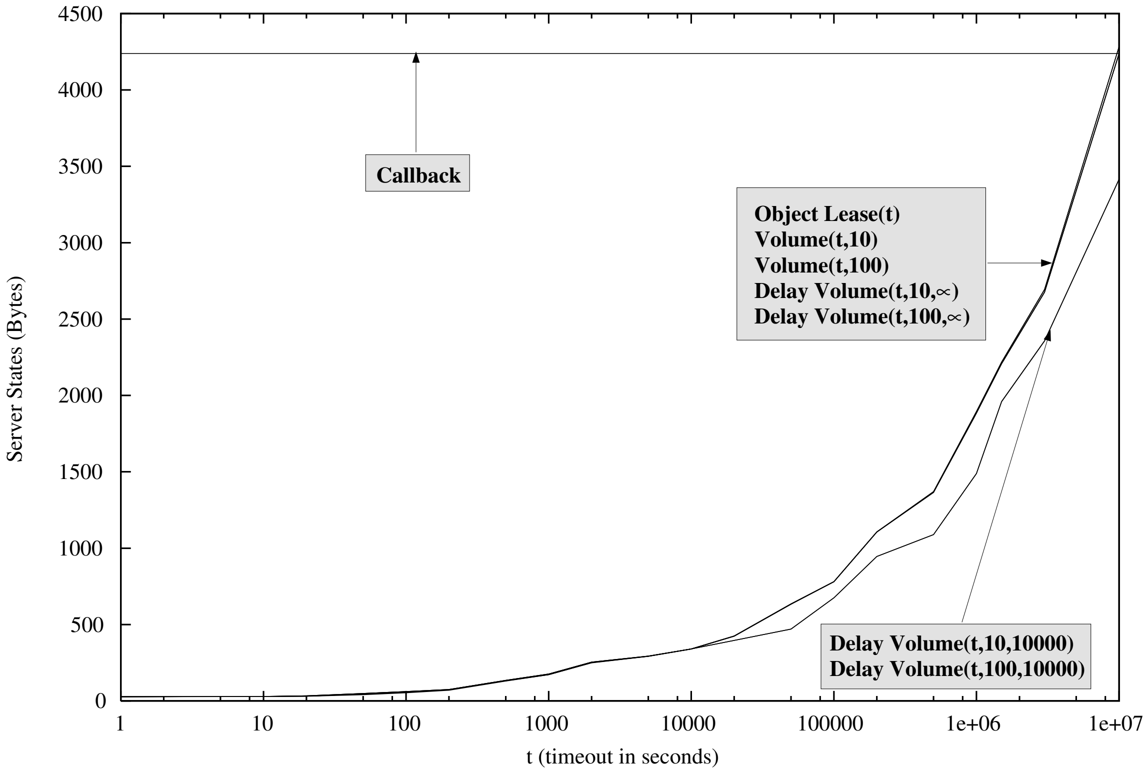

Figure 7: State at the most popular server vs. timeout.

be allowed to unknowingly access stale data; this may allow larger values of t_v for some services that use *Best Effort*. Figure 6 shows the effect of varying the volume lease timeout on the number of messages sent.

5.2 Server state

Figures 7 and 8 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 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

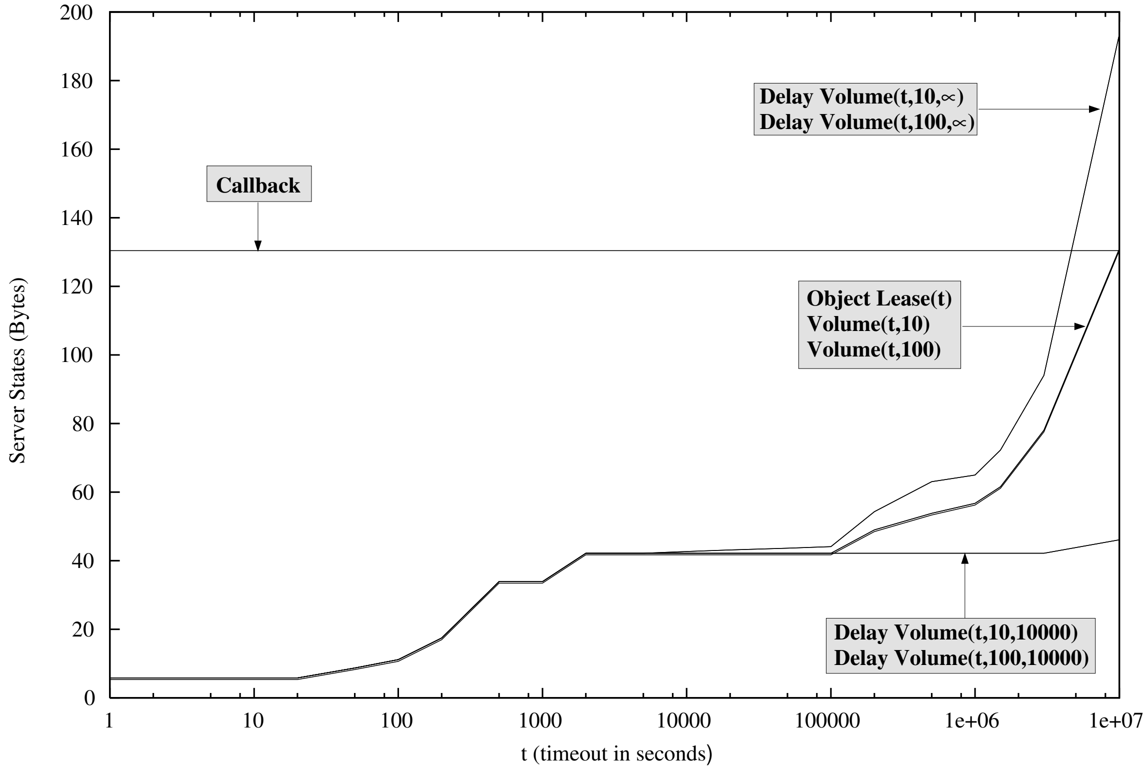

Figure 8: State at the 10^{th} most popular server vs. timeout.

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 9 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

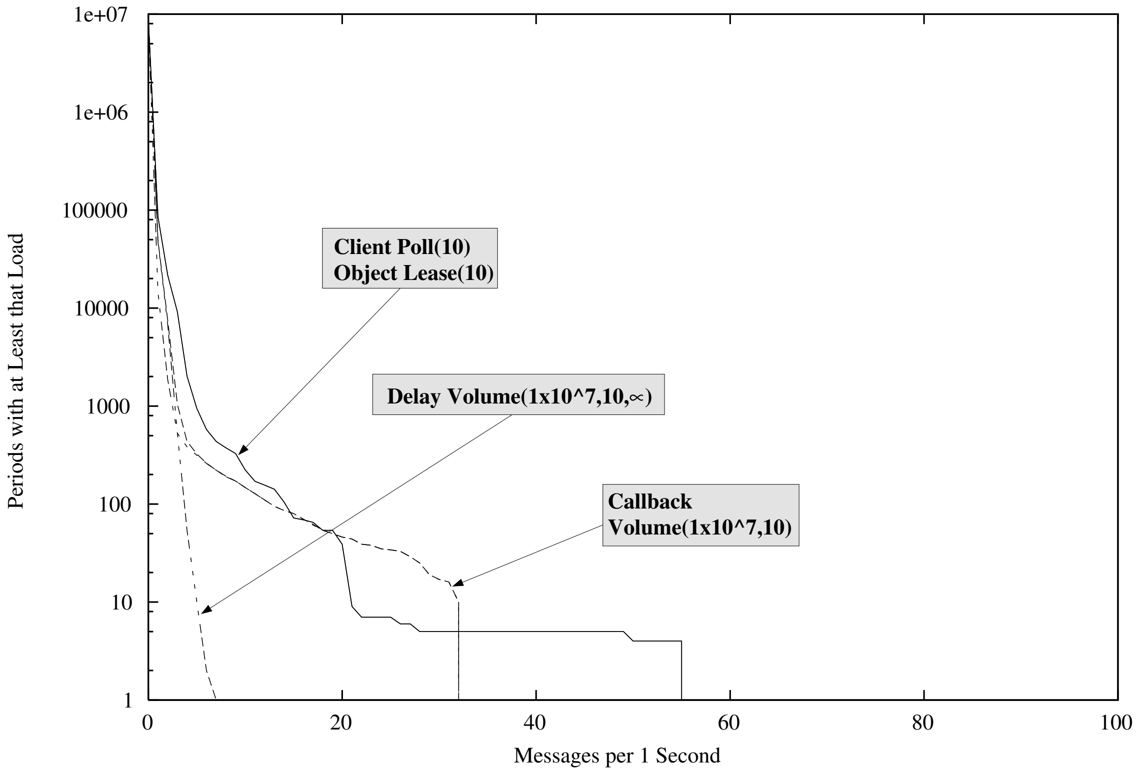

Figure 9: Periods of heavy server load under default workload for the most heavily loaded server.

more spread out in time. The second reason for *Callback* and *Volume*'s advantage in this experiment is that in the trace clients 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 10 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*.

5.4 Sensitivity

Our workload utilizes a trace of read events, but it generates write events synthetically. In this subsection, we examine how different assumptions about write frequency affect our results.

Figure 11 shows the performance of the algorithms for representative parameters as we vary the write frequency. Our default workload gives the 10% most referenced files a per-day change probability of 0.5%, 3% of the files a per-day change probability of 20%, 10% of the files a probability of 5%, and 77% of the files a per-day change probability of 2%. For each point on the graph, we multiply those per-day probabilities by the value indicated by the x-axis. Note that our workload generator converts per-day change probabilities to per-second change probabilities, so per-day probabilities greater than 100% are possible

We examine the lease algorithms as they might be parameterized in a system that never wishes to delay writes more than 100 seconds and compare to a poll algorithm with a 100-second timeout

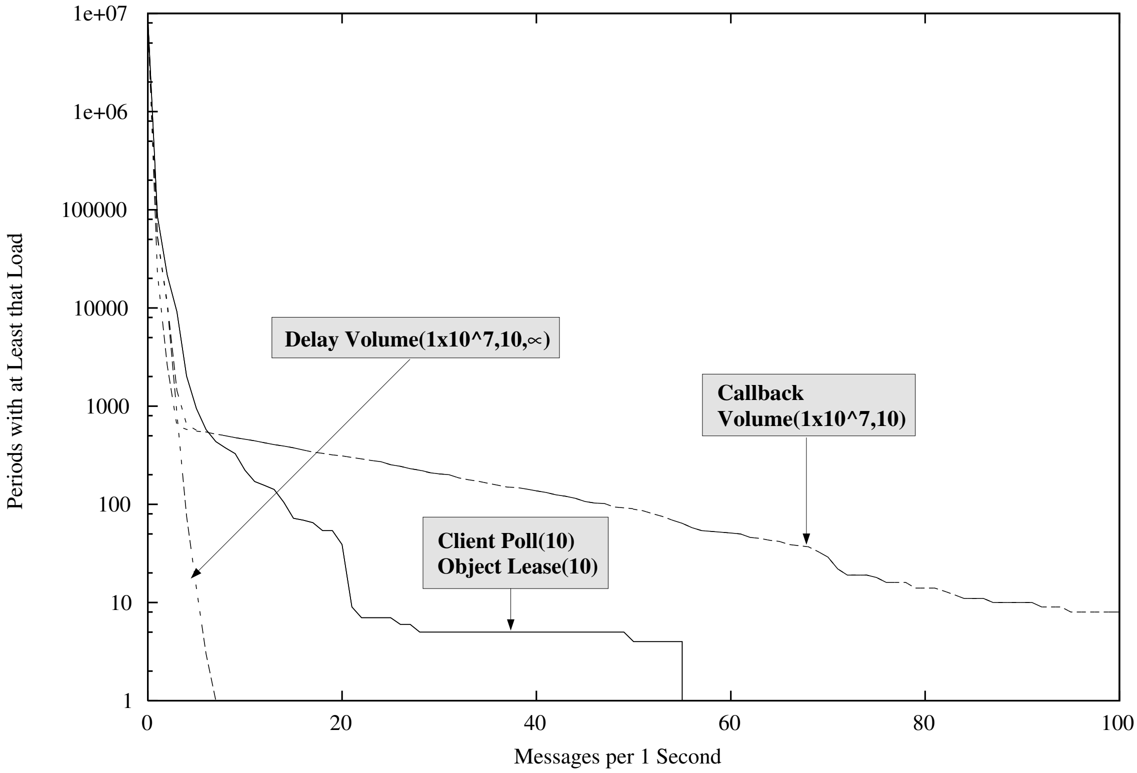

Figure 10: Periods of heavy server load under "bursty write" workload for the most heavily loaded server.

and a callback algorithm with infinite timeout. These results indicate that the *Client Poll*($t = 100$) and *Lease*($t = 100$) are little affected by changing write rates. This is because the object timeouts are so short that writes are unlikely to cause many invalidations even when their frequency is increased 100-fold. The volume lease algorithms and *Callback* all cost more as write frequency increases. The cost of *Volume*($t = 1000000, t_v = 100$) and *Callback* increase more quickly than the cost of *Delayed Volume*($t = 10000000, t_v = 100, d = \infty$) because the first two algorithms have long object callback periods and thus send invalidation messages to all clients that have done reads between a pair of writes. *Delayed Volume* rises more slowly because it does not send object invalidations once a volume lease expires.

6 Related work

Our study builds on efforts to assess the cost of strong consistency in wide area networks. Gwertzman and Seltzer [10] 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 [14]. Also, we use volume leases to address many of the challenges to strong consistency.

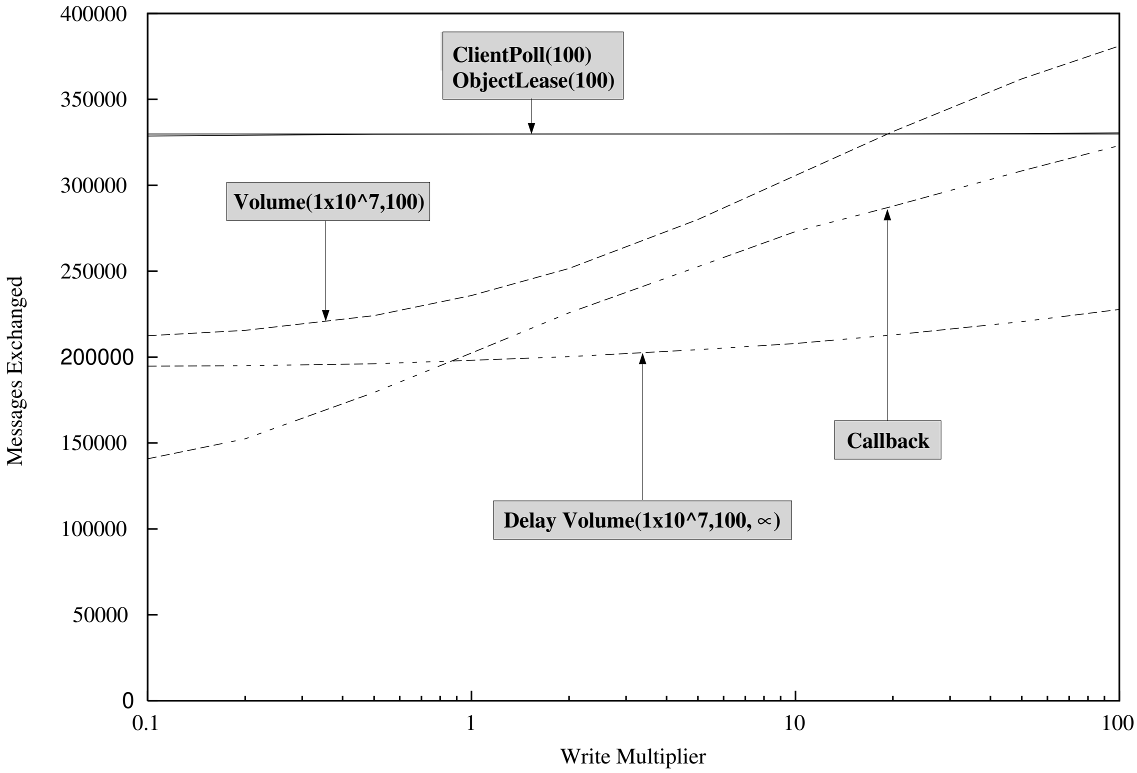

Figure 11: Messages sent under different write frequencies. The x-axis represents a multiplier to the write frequency compared to our default workload.

We also build on the work of Liu and Cao [14], 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 [11, 17, 19]. Several aspects of Coda's [13] consistency protocol are reflected in our algorithms. In particular, our notion of a volume is similar to that used in Coda [16]. However, ours differ in 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). In our environment, clients are associated with a larger universe of servers, so we only renew volume leases when a client is actively accessing the server. Also, in our algorithm when an object is modified, the server does not send volume invalidation messages to clients that hold volume leases but not object leases on the object in question. We thus avoid the *false sharing* problem of which Mummert warns [16].

Our best effort leases algorithm provides similar semantics to and was inspired by Coda's optimistic concurrency protocol [13]. Bayou [20] and Rover [12] also implement optimistic concurrency, but they can detect and react to more general types of conflicts than can Coda.

Worrell [21] studied invalidation-based protocols in a hierarchical caching system and concluded that server-driven consistency was practical for the web. We plan to explore ways to add hierarchy to our algorithms in the future.

Cache consistency protocols have long been studied for distributed file systems [18, 17, 19]. Howard et. al [11] reached the somewhat counter-intuitive conclusion that server-driven consistency generally imposed less load on the server than client polling even though server-driven algorithms provide stronger guarantees for clients. This is because servers have enough information to know exactly when messages need to be sent.

Mogul's draft proposal for HTTP 1.1 [15] includes a notion of grouping files into volumes to reduce the overhead of HTTP's polling-based consistency protocol. We are not aware of any implementations of this idea.

Finally, we note that volume leases on the set of all objects provided by a server can be thought of as providing a framework for the "heartbeat" messages used in many distributed state systems.

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 the timeout-based *Client Poll* algorithm with 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 *Client Poll*. 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 Volume 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

Some of the work described here appeared in an earlier paper [22]. We thank James Gwertzman and Margo Seltzer for making their simulator available to us so we could validate our simulator. We thank Carlos Cunha, Azer Bestavros and Mark Crovella for making the BU web traces available to us. This work was funded in part by a NSF CISE grant (CDA-9624082), gifts from Novell and Sun Microsystems, and DARPA/SPAWAR grant number N66001-98-8911. Dahlin and Alvisi were supported by NSF CAREER awards (CCR-9733842) and (CCR-9734185), respectively.

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 Dissemination and Service to Reduce Server Load, Network Traffic, and Service Time in Distributed Information Systems. In *International Conference on Data Engineering*, March 1996.

[3] S. Chandra, M. Dahlin, B. Richards, R. Wang, T. Anderson, and J. Larus. Experience with a Language for Writing Coherence Protocols. In *USENIX Conference on Domain-Specific Languages*, October 1997.

[4] S. Chandra, B. Richards, and J. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[5] 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.

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

[7] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[8] 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.

[9] 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. Springer-Verlag, 1977. Lecture Notes on Computer Science 60.

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

[11] 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.

[12] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford, and M. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, December 1995.

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

[14] 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*, pages 12–21, May 1997.

[15] J. Mogul. A Design for Caching in HTTP 1.1 Preliminary Draft. Technical report, Internet Engineering Task Force (IETF), January 1996. Work in Progress.

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

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

[18] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, June 1985.

[19] 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.

[20] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 172–183, December 1995.

[21] K. Worrell. Invalidation in Large Scale Network Object Caches. Master's thesis, University of Colorado, Boulder, 1994.

[22] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of the Eighteenth International Conference on Distributed Computing Systems*, May 1998.