Copyright

by

Jian Yin

2003

The Dissertation Committee for Jian Yin

certifies that this is the approved version of the following dissertation:

Volume Lease:
A Scalable Cache Consistency Framework

Committee:

Mike Dahlin, Supervisor

Lorenzo Alvisi, Supervisor

Arun Iyengar

Calvin Lin

Harrick Vin

Volume Lease:
A Scalable Cache Consistency Framework

by

Jian Yin, B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2003

To Zhao Lejing for her support

Acknowledgments

I am fortunate to have Dr. Dahlin and Dr. Alvisi as my advisors. Their belief in my technical abilities makes this dissertation possible. I am forever indebted to the huge amount of time that they have spent in helping me improve my research skills and writing skills. Their office doors have always been open to me whenever I need help with my research and writing.

I am also lucky to have Dr. Von Balaan as my undergraduate advisor. He has introduced me to the fun of doing computer science research and building experimental systems in Lisp.

Numerous people in this department make my Ph.D. study enjoyable. In particular, technical discussions with Calvin Lin, Harric Vin, Jean-Philippe Martin, Amol Pramod Nayate, Praveen Yalagandul, Gao Lei, Zeng Jiandan, and Arun Venkataramani are always enjoyable. Mike Dahlin, Lorenzo Alvisi, Arun Iyengar, Harric Vin, and Calvin Lin have spend much time reading through this dissertation. Their insightful comments have helped to improve the quality of this dissertation.

Part of this dissertation is drawn from the previous papers:

"Volume Leases for Consistency in Large-scale Systems" [91], J. Yin, L. Alvisi, and M. Dahlin, and A. Iyengar, IEEE Transactions on Knowledge and Data Engineering, 11:2, July/August 1999.

- Discussion on the basic Volume Lease to provide strong consistency

"Engineering web cache consistency" [88], J. Yin, L. Alvisi, and M. Dahlin, and A. Iyengar, ACM Transactions on Internet Technologies, 2:3, August 2002.

- Experimental results on the performance of the basic Volume Lease
- Experimental results on the scalability

"Hierarchical Cache Consistency in WAN" [90], J. Yin, L. Alvisi, and M. Dahlin, and A. Iyengar, Proceedings of the Second USENIX Symposium on Internet Technologies and Systems(USITS99), October 1999.

- Discussion on hierarchical Volume Lease

JIAN YIN

The University of Texas at Austin

December 2003

Volume Lease:
A Scalable Cache Consistency Framework

Publication No. ____________________

Jian Yin, Ph.D.
The University of Texas at Austin, 2003

Supervisors: Mike Dahlin and Lorenzo Alvisi

In this thesis, we study scalable cache consistency. Caching remotely deployed services near users can reduce network overhead, shorten user response time, increase availability, and improve scalability. The effectiveness of caching, however, is determined to a great extent by the efficiency of cache consistency protocols. Given that many services of importance are accessed by thousands or millions of clients, scalable cache consistency is essential to realize the full potential of caching.

We propose a general framework, Volume Lease, for scalable cache consistency design. Our framework allows us to decouple synchronization from invalidation. Synchronization in our framework is performed on sets of objects called volumes, which allows the cost of synchronization to be amortized over many objects in a volume. This framework captures the essential mechanism for high-performance scalable cache consistency design and is still flexible enough to allow us to explore

a wide design space.

With this framework, we are able to design various consistency protocols to provide a wide spectrum of consistency guarantees in a wide range of environments. In particular, we examine various mechanisms provided by Volume Lease to address challenges of scalable cache consistency: interactivity, flexibility, scalability, and robustness. We evaluate our consistency protocols experimentally with both trace-driven simulation. Our results show that our consistency protocols can scale to millions of machines, handle common failures gracefully, allow systems to exploit weak consistency requirements to reduce overhead, and provide close to optimal latency.

Table of Contents

Acknowledgments **v**

Abstract **vii**

Chapter 1 Introduction **1**

1.1 Volume Lease 2

1.2 Contributions 4

1.3 Overview of this Dissertation 5

Chapter 2 Background **7**

2.1 Overview 7

2.2 Web Caching 8

2.3 Challenges of Scalable Consistency 9

2.4 Existing Approaches 12

2.5 Summary 13

Chapter 3 Volume Lease Framework **14**

3.1 Overview 14

3.2 Framework 15

3.3 Algorithms 17
3.3.1 Strong Consistency Baseline Protocol 18
3.3.2 Weak Consistency 25
3.4 Analysis 26
3.5 Related Work 30
3.5.1 Client Polling 30
3.5.2 Invalidation 32
3.5.3 Consistency Semantics 33
3.5.4 Industry Standard and Commercial Systems 34
3.6 Conclusion 35

Chapter 4 Performance **36**

4.1 Methodology 36
4.2 Consistency 40
4.3 Basic Volume Lease 41
4.4 Adaptive TTL and Average TTL 52
4.5 Prefetching/Pushing Lease Renewals 55
4.6 Conclusion 57

Chapter 5 Scalability **59**

5.1 Overview 59
5.2 Fault Tolerance 62
5.2.1 Performance with Partial Failures 62
5.2.2 Recovery 63
5.3 Server State 69

5.4 Peak Server Load 72
5.5 Conclusion 75

Chapter 6 Hierarchy and Multicast **77**
6.1 Overview 77
6.2 Algorithms 79
6.2.1 Join and split 83
6.2.2 Fault tolerant static hierarchy 84
6.2.3 Dynamic hierarchy configuration 84
6.3 Evaluation 86
6.3.1 Generic hierarchy 88
6.3.2 Server cluster 98
6.3.3 Server-proxy-client 101
6.4 Related Work 102
6.5 Conclusions 104

Chapter 7 Conclusion and Future Work **105**
7.1 Future Direction 106
7.2 Summary 108

Bibliography **109**

Vita **121**

Chapter 1

Introduction

This dissertation presents Volume Lease, a scalable cache consistency framework. The Volume Lease framework is designed to improve the performance of large-scale caching systems, in which the services that are deployed at central servers are cached near users to reduce latency. Effectiveness of large-scale caching is determined to a great extent by consistency protocols that coordinate a server and its caches to ensure that the service provided by the caches reflects the updates in the server.

Designing efficient large-scale cache consistency protocols poses three challenges: interactivity, scalability, and robustness. By interactivity, we mean that cache consistency protocols should achieve the full potential of caching in reducing latency despite high communication delay between servers and caches. By scalability, we mean that a consistency protocol should perform well in a large-scale caching system with potentially millions of caches. By robustness, we mean that a cache consistency protocol can continue to function in face of cache crashes and network disconnections. In a large-scale distributed system consisting of millions

of machines, the possibilities of partial failures are high. Thus robustness is closely related to and essential to scalability.

Our study in cache consistency is motivated by web caching. Today, many important services are deployed as web applications from centralized servers. The centralized servers from which web caches replicate the content are called origin servers in the HTTP protocol specification [41]. Web caches, including browser caches, proxy caches [22, 38, 78, 84, 86], and server replicas [9, 12, 46, 52, 73, 83], are widely deployed to reduce latency. However, traditional client polling protocols used in HTTP [41] introduce high latency even when providing weak consistency and can eliminate the benefit of caching in reducing latency when providing stronger consistency guarantees. Thus, more efficient cache consistency protocols are required to achieve the full potential of web caching.

1.1 Volume Lease

We develop a general framework, Volume Lease, for scalable cache consistency design. Volume Lease maintains consistency by managing two sorts of leases, object leases and volume leases. An object lease is a lease on an individual object. A volume lease is a lease on a volume, a set of related objects. The Volume Lease framework is defined by two constraints governing how volume leases and object leases are managed. I) To read an object, a cache must have both the object lease and the volume lease on the volume that contains the object. II) To grant a cache a volume lease, a server must ensure that the cache only holds object leases on the set of cached objects that are up-to-date.

Object leases are usually long to reduce the number of object lease renewals

and thus the latency and overhead penalty resulting from these renewals. Volume leases are usually short to improve scalability and robustness. Although volume leases are short, the cost of volume lease renewals is amortized over many objects in a volume. Consequently, the Volume Lease framework allows us to design interactivity, scalability, and availability consistency protocols.

- Interactivity In most consistency protocols, a cache is required to be synchronized with a server before returning cached objects to ensure consistency. In Wide Area Networks (WAN), synchronizing means at least an Internet round trip delay, which can lead to high latency if synchronizing is performed before a read. Volume Lease allows the latency penalty of synchronization to be amortized over many objects in a volume and thus leads to low latency. Moreover, Volume Lease is flexible enough to allow synchronization to be performed in anticipation of reads, which can further reduce latency.

- Scalability Because Volume Lease protocols need to track clients and send out invalidation messages, they can potentially have two scalability challenges: server state and peak load. Volume Lease is flexible enough to allow a range of techniques to reduce server state and peak load. We can place a hard limit on server state and peak server load with those techniques. Furthermore, Volume Lease can be used to build a consistency hierarchy, organizing a set of machines to provide consistency.

- Robustness Short volume leases contribute to Volume Lease protocols' ability to tolerate client failures, server failures, and network partitions. Moreover, Volume Lease protocols can resynchronize servers and clients quickly when the

machines and networks recover, and thus resume service quickly.

1.2 Contributions

We evaluate Volume Lease in a systematic way. First, we survey consistency requirements of many web applications and propose a spectrum of consistency semantics to capture the consistency requirements of these services. Then, we study how to provide these consistency semantics efficiently in a range of environments with the general Volume Lease framework. Overall, this dissertation makes three contributions.

First, we propose a uniform conceptual framework, Volume Lease, for scalable cache consistency design. By abstracting away implementation details, Volume Lease allows us to evaluate tradeoffs among consistency guarantees, overhead, and performance of a wide varieties of algorithms proposed by both us and other researchers in a uniform framework.

Second, we design a wide range of consistency protocols as suggested by Volume Lease to provide a wide spectrum of consistency guarantees in a wide range of computing environments.

Third, we evaluate the consistency protocols with simulations to demonstrate the efficiency of these consistency protocols. The experimental results lead to the fundamental understanding on how to design scalable consistency for web applications. Our research has influenced the IETF standardization efforts [29, 57, 80] and IBM web server products [79].

1.3 Overview of this Dissertation

The dissertation consists of seven chapters. This chapter gives an overview of this dissertation.

The remaining chapters describe the Volume Lease framework in detail and evaluate the performance, scalability, and deployment issues of various protocols designed with this framework.

Chapter 2 gives the background of this dissertation. It explains why scalable cache consistency protocols are essential to the performance of large-scale caching and why designing scalable cache consistency protocols is challenging. We also compare scalable cache consistency with the consistency semantics required in other areas, in particular, database systems and distributed file systems. We discuss why design challenges of consistency protocols in these systems are different from those in large-scale caching systems.

Chapter 3 introduces the Volume Lease framework and explains how the Volume Lease framework can be used to design scalable cache consistency protocols. We emphasize the two fundamental constraints in Volume Lease. Then we discuss the performance implications of various design choices that are not fixed by the two constraints and how these design choices can be used to address various design challenges in scalable consistency protocols.

Chapter 4 experimentally evaluates various Volume Lease protocols and demonstrates how various parameters in the Volume Lease framework can affect consistency, latency, and overhead. We start with a baseline Volume Lease protocol, the basic Volume Lease protocol. We use experimental results to show how volume size, volume lease length, object lease length, and various invalidation mech-

anisms affect performance. Then we discuss how various volume lease management techniques, such as prefetching renewal, can be added to the basic Volume Lease protocol to offer a range of tradeoff between consistency, latency, and overhead.

Chapter 5 addresses three scalability challenges of Volume Lease: fault tolerance, server state, and load burstiness. The large scale of the Internet results in high possibility of partial failures. Recovering from failures gracefully is an essential requirement for scalable protocols. Resynchronizing servers' state and clients' state is essential for recovery. The performance tradeoff between two methods for resynchronization, bulk revalidation and delayed invalidation, is presented. We address server state and load burstiness by using the flexibility of Volume Lease to reduce server state and burstiness without significantly affecting the performance. Moreover, we can place a hard limit on server state and peak load so that our system can continue to provide consistency even in some extreme cases.

Chapter 6 discusses how to use a set of machines to provide consistency for demanding workloads. The set of machines are organized into a hierarchy to distributed the load. The key to scalable consistency hierarchy is the ability to dynamically adjust the hierarchy to replace fault nodes and optimize performance as workload changes. Two mechanisms, join and split, are used to serve this purpose.

Finally, Chapter 7 summarizes the contributions of this dissertation and discusses the lessons learned during this project. We also talk about how this dissertation fits into the broad effort in enhancing web computing and how other research complements this dissertation.

Chapter 2

Background

2.1 Overview

This chapter presents the background knowledge for understanding the rest of this thesis. We introduce the problem of scalable consistency and explain why this problem is challenging.

The rest of this chapter is organized as follows. First, we introduce large-scale caching systems. We use web caching as our primary example to explain the importance of scalable consistency for the effectiveness of large-scale caching. Second, we describe some early work on cache consistency in file systems and database systems. The scalable consistency problem is more challenging than consistency in file systems and database systems mainly because of the large scale of the systems in which scalable consistency operates. Third, we describe existing cache consistency protocols, client polling protocols, used in web caching systems. We then explain why client polling hurts the effectiveness of web caching in reducing latency.

2.2 Web Caching

In this section, we use web caching as an example of large-scale caching to explain the importance of consistency protocols.

Traditional web services are provided from a central server. A central server can be a high-end server with multiple processors and distributed memory or a cluster of machines. This architecture can cause scalability problems as the Internet continues to grow. Today's popular web services can have millions of users. Moreover, the peak server loads of these services can be much higher than their average load [6, 7, 50].

Caching can be used to address this dilemma. A web caching system includes a large number of caching nodes in addition to central servers and clients. Caching nodes do not provide services by themselves. However, they can replicate services from other web servers and thus allow the computing resources - CPU cycles, network bandwidth, and storage capacity - of these machines to be multiplexed among many web services. In a web caching system, we have a set of caches either deployed at the users' machines or deployed at the network gateway of an organization to be shared by all the users in the organization. When a user retrieves an object, the system first attempts to find the object in a cache. If the object is not located in the cache, the request is forwarded to the origin server. When the reply comes back from the origin server, the cache can store a local copy of the object to satisfy future requests.

Today, there are three kinds of web caching systems: local caches, proxy caches, and content distribution networks (CDN). A local cache is a cache that is deployed in a user's application program which is usually a browser. A proxy cache

is a cache that is deployed by an organization and shared by users and computers in the organization. While local and proxy caching works on clients' behalf to reduce network bandwidth consumption and user response time by storing a copy of the objects which may be accessed later, CDNs [68, 9, 46], as provided by Akamai, work on the service providers' behalf to achieve the same benefits. A CDN is composed of a set of machines strategically placed into the network [73, 52]. According to users' request patterns, the content of a service is replicated to the set of machines which are most suited to provide the service [54]. A user can then select a replica close to this user to retrieve an object. This can reduce both network bandwidth and user response time because the replicas selected to provide the service can be much closer to users than origin servers. CDNs are usually installed on the behalf of service providers and thus are trusted by service providers for security and accounting purposes. For example, service providers can trust service replicas to keep hit counts, which is important in some business models.

However, key to replication is consistency. Consistency, in web caching, means that an object retrieved from caches should reflect the object in the origin server. Consistency protocols specifies how to coordinate caches and origin servers to maintain consistency. Because a consistency protocol often requires a cache to contact the server before allowing a cached object to be returned to users, it can determine to a great extent the effectiveness of replication in reducing latency [56].

2.3 Challenges of Scalable Consistency

Although consistency has been studied for file systems and database systems [4, 11, 31, 45, 49, 63, 64, 70, 74, 75] for quite a long time, two challenges, consistency

semantics and scalability, are significant in web caching systems.

Replication and consistency [11, 31, 45, 70, 74] have been extensively studied in distributed database systems. Classic distributed database systems provide transaction-based consistency. In these system, a transaction is considered an indivisible unit. The system provides atomicity, isolation, and durability even though the underlying data are replicated. Atomicity means that the sequence of operations in a transaction are executed as a unit: either all of them are executed or none of them is executed. Isolation means that a transaction in process and not yet committed must not affect any other transaction. Durability means committed data is saved by the system such that, even in the event of a failure and system restart, the data is available in its correct state. The mechanism used here is locking. The primary difference between consistency in distributed database systems and web caching systems is that many web applications can tolerate consistency semantics that are much weaker than transaction-based consistency. Stronger consistency requirements limit the scale of replication in distributed database systems. Researchers [45] have observed that overhead increases super-linearly with the number of replicas.

Consistency has also been examined in distributed file systems [44, 49, 63, 64, 75]. Early distributed file systems [75] use client polling to maintain consistency. Later systems [44, 49, 63, 64] use invalidations and have been shown to provide better consistency and lower overhead than client polling. This lesson is still valid for web caching systems. One difference between distributed file systems and web caching systems is that a distributed file system has many writers and many readers, while a web caching system usually has only one writer, the origin server. Another difference is that the distributed file systems in previous studies usually run in a

LAN environment with up to thousands of clients, while a popular web server today can have millions of clients.

The large scale of web cache systems can have many implication. For instance, a simple invalidation protocol that requires servers to invalidate all the cached objects before completing a write will be impractical in this environment: a popular object can be cached by millions of clients; invalidating these cached objects can introduce both high server overhead and high latency. Another notable implication of the large scale of the Internet is the requirement of fault tolerance. As the number of clients increases, the possibility that at least one client either crashes or is disconnected from the server increases. For a large-scale system with millions of clients, the presence of these partial failures will be common and our system should perform well and provide consistency guarantees even with partial failures.

Web applications can also have a wide range of consistency semantics. For some applications, it is important for caches to provide the same content as origin servers. For instance, an online stock exchange application must ensure that any user will see the same stock prices from any cache in the system. In contrast, some applications care more about performance or overhead than strict consistency. For example, it is generally acceptable for an online news service to provide slightly stale news from different caches as long as the news is not too stale. Consistency protocols should be able to exploit these weak consistency requirements to improve performance and reduce overhead.

2.4 Existing Approaches

Although the HTTP protocol allows a rich set of implementations for maintaining consistency [40], the core mechanism is client polling. In client polling, clients poll the server to determine whether a cached copy of an object is valid. In response to a cache's polling, the server either notifies the cache that the cached object is still valid or sends the new version of the object to the cache.

In client polling, a cache must poll the server frequently to achieve good consistency. Polling for a read causes a cache to wait for the server's reply for the poll message before returning the cached object even when the cached object is valid. Thus, client polling can eliminate the advantage of caching in reducing latency especially when object sizes are small. Thus, in client polling, consistency can only be achieved at the expense of reducing the effectiveness of caching.

The drawbacks of client polling are quite prominent in practice. Researchers [6, 34, 50, 69] have observed that 25% of HTTP requests to well-tuned web servers can be unnecessary polls with existing consistency protocols. An unnecessary poll happens when the consistency protocol does not know whether a valid cached object is valid and has to contact the server to validate this object. Unnecessary polls eliminate the benefits of replication in reducing latency. Moreover, these researchers have also observed that a substantial fraction of reads return stale data even with a large number of unnecessary polls. This weak consistency can make the traditional consistency protocols unattractive for future web applications requiring stronger consistency guarantees.

2.5 Summary

Large-scale caching is promising in reduing latency of web applications. The effectiveness of caching is determined to a great extent by the efficiency of consistency protocols.

The primary challenge of large-scale cache consistency is scalability. A practical large-scale consistency protocol for web caching should be able to perform well in a system with millions of clients. Scalability requires robustness in a system where parts can fail. Since the possibilities of partial failures increase as the scale of a system increases, the performance of scalable consistency protocols under partial failures is critical. Moreover, a scalable consistency protocol should be able to exploit the weak consistency requirements of some web applications for better performance.

Chapter 3

Volume Lease Framework

3.1 Overview

In this chapter, we present a general framework for scalable consistency, Volume Lease. Volume Lease allows us to decouple synchronization from invalidation for aggressive optimization. Synchronization in our framework is performed on sets of objects called volumes with volume leases. Volume Lease amortizes the cost of synchronization over many objects in a volume. This framework captures the essential mechanism for high-performance scalable cache consistency design and is still flexible enough to allow us to explore a wide design space.

The rest of this chapter is organized as follows. First, we describe the Volume Lease framework. Second, we illustrate how to use Volume Lease to provide a range of consistency and how to adjust the parameters in Volume Lease to optimize performance and improve scalability. Third, we present experimental results to demonstrate the effectiveness of Volume Lease. Finally, we sum up the contributions of this chapter.

3.2 Framework

Invalidation protocols [47, 17] can be used to address the drawbacks of client polling. In invalidation protocols, a server notifies caches of updates. Thus, a cache does not need to poll before reads and hence reduces read latency. In the basic invalidation protocol, servers keep track of which clients are caching which objects. Before modifying an object, a server notifies the clients with copies of the object. The basic invalidation protocol achieves low read latency because a client can read a valid cache object immediately without contacting the server. However, writes can cause scalability problem. The basic invalidation protocol can not handle failures such as caches' crashes or network partitions separating a server from a client, which can be common in a large scale system as we discussed in the previous section. Moreover, when an object is modified, the server is required to send invalidation messages to all the caches that have ever read the object.

Lease [44] can be used to make invalidation protocols scalable. Lease can be thought of as an invalidation protocol with timeouts. In Lease, to read an object, a client first acquires a lease for it with an associated timeout. 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 handle failures. If a client or network failure prevents a server from invalidating a cached object, the lease on this object can expire and thus prevent the cache from returning this object indefinitely. Lease can also reduce the number of invalidation messages during a write. 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. Essentially, leases also capture synchronization between servers and caches. To achieve the scalability benefits of Lease, leases need to be short. However, short leases increase the cost of lease renewals and reduce the latency reduction benefits of invalidation.

Volume Lease is designed to achieve scalability benefits of short leases without sacrificing low latency. Unlike Lease in which synchronization and invalidation are both captured by the concept of lease, in volume lease, we decouple synchronization from invalidation. Invalidation is captured by object leases and synchronization is captured by volume leases. By requesting an object lease on an object, a client registers with a server the object on which the client wants to learn updates. Synchronization is performed on a volume, a set of objects. Origin servers ensure that a client has been notified of all the previous updates on all objects in a volume before the server can grant the volume lease. A volume lease gives a cache a permission to read any object with a valid object lease from the volume without contacting origin servers. The cost of synchronization, which is the cost of a volume lease renewal in Volume Lease, is amortized over many objects in a volume. Thus, Volume Lease can have a short synchronization interval to achieve scalability without paying a high penalty.

The requirements of Volume Lease are captured by two constraints: A client can only read an object if it holds both the volume lease and the object lease; a server must deliver all the invalidation messages in a volume to a cache before allowing the cache to renew the volume lease. These two constraints capture the essential mechanisms to provide scalable consistency and allows many design choices. Thus, Volume Lease is a general framework. This framework allows us to study all design

choices systematically. Volume Lease can be used to provide strong consistency and weak consistency. It also can choose to make various design choices over volume size, volume lease length, object lease length, volume lease renewal policy, and invalidation management to optimize performance for various workloads.

In the next section, we examine how volume lease can be used to provide strong consistency and a spectrum of weak consistency. In the section after the next, we discuss how volume size, volume lease length, object lease length, volume lease renewal policy, and invalidation management affect performance.

3.3 Algorithms

Volume Lease can provide both strong consistency [81] and weak consistency. In the next section, we discussion how to provide strong consistency with Volume Lease.

We consider two kinds of consistency models: strong consistency and weak consistency. In strong consistency, all the reads return the results of the latest completed write. Here, a read and write starts when this operation is submitted to the system, ends when the result of this operation is returned. Thus after a write is completed, no read can return the data that have been overwritten. Note that strong consistency that we used here satisfy linearizability [61]. To guarantee strong consistency, sometime it is necessary to block the completion of a write immediately after the write has been submitted and before the data from this write has relayed to this client. On the other hand, the server may prevent a write from being completed until the server ensures that there is no client that can read the stale data.

The weak consistency actually describes a spectrum of consistency. There are two metrics for weak consistency: worst-case staleness bound and average-case

consistency. The worst-case consistency is specified by Δ consistency [77]. That is, the worst-case staleness of t units of time means that t units of time after a latest write, all the reads must return the results of this write. That means, average-case consistency include stale read rates and staleness. Specifically, stale read rate is the percentage of reads over all the reads and staleness of the data, the data staleness of all the reads, which ranges from 0 to the worst-case staleness bound.

3.3.1 Strong Consistency Baseline Protocol

To ensure strong consistency, we must ensure that after a write is completed, no data which have been overwritten can be returned from any caches. In Volume Lease, a cache needs to have both the volume lease and the object lease to return an object. Thus, before a write can be completed, the server must ensure that any cache either doesn't have the volume lease or doesn't have the object lease. When there is a write, the server sends out invalidation messages to all the caches which have the object lease. Upon receiving acknowledgment or expiring of the volume lease from every cache which hold the object lease, the server can then proceed with the write.

In the rest of this subsection, we present a protocol to maintain strong consistency. Note that this protocol is just a baseline algorithm. Much of optimization can be done given the flexibility of Volume Lease. We discuss these design choices in detail in Section 3.4.

Figures 3.1, 3.2, and 3.3 show the data structures used by the Volume Lease 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.

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. Note that the maximum write blocking time is the volume lease length. This worst-case scenario can happen when the server issues a volume lease to a cache just before a write. If the cache becomes disconnected from the server and cannot receive invalidation messages, the server needs to wait until the volume lease expires. In this case, the amount of waiting time equals the volume lease length.

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.

When a server fails we assume that the state used to maintain cache consistency is lost. 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 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.

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 3.1: Data Structures for Volume Lease algorithm.

Protocol verification

To verify the correctness of the consistency algorithm, we implemented a variation of the Volume Lease algorithm described in Figures 3.2 and 3.3 using the Teapot system [21]. 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 [33] code for mechanical verification. The Murphi system searches the protocol's state space for deadlocks or cases where the system's correctness invariants are violated. Although Murphi's exhaustive search

```
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 3.2: The Volume Leases Protocol (Server Side).

```
Client reads object o
  if ¬validLease(o.volume) ∨ ¬validLease(o.id) then
    renewLease(o.volume, o)
  read local copy of o

renewLease(volume v, object o)
  epoch ← max(v.epoch, −1)
  vnum ← 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 ← ∅
    for all objects o for which ((o.volume = v) ∧ (validLease(o))
      leaseSet ← leaseSet ∪ ⟨o.id, o.version⟩
    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 ∈ invalList
      let o be the object for which o.id = objId
      o.expire = −1; delete o.data; o.data ← NULL
    for all ⟨objId, version, expire⟩ ∈ renewList
      let o be the object for which o.id = objId
      assert(o.version = version)
      o.expire ← 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 ← NULL
  send(ACK_INVALIDATE, o.id) to server
```

Figure 3.3: The Volume Leases Protocol (Client Side).

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 [20].

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.

Volume Lease with delayed invalidations

The performance of Volume Lease 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 Lease with Delayed Invalidations* algorithm modifies the basic Volume Lease algorithm 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.

3.3.2 Weak Consistency

Volume Lease can be easily adapted to provide weak consistency. Recalled that the two constraints of the Volume Lease framework guarantee that whenever a client receives a new volume lease, it becomes synchronized with the server by learning all the invalidations in the volume prior to the volume lease renewal. Thus, the server can complete a write without waiting for acknowledgments for its invalidation messages and still provide a staleness bound equal to the volume lease length.

A small modification to the strong consistency protocol discussed in the previous subsection can implement this idea and result in a protocol providing staleness bounds. The modification is that the server is no longer required to wait for the acknowledgments of invalidation messages or the expiration of the volume lease from a client before completing a write. The server can complete a write immediately after the write is submitted. The server can choose to send out invalidation messages to reduce stale reads before the expiration of clients' volume leases or to buffer these invalidation messages. By the time that a client renew its volume lease, all the unacknowledged invalidation messages are piggybacked to the message that grants

the client a new volume lease. Thus, no client can read any data which have been overwritten for more than the volume-lease-length amount of time.

Relaxing consistency semantics yields two advantages. First, it allows a write to complete quickly. A server starts to overwrite the data as soon as a write is submitted without waiting for acknowledgments of invalidation messages and the expiration of the volume leases. Thus, weak consistency not only eliminates write blocking, but also allows some clients to read new content as soon as a write is submitted. Second, weak consistency also makes sending invalidation messages flexible. The server can choose either to send invalidation messages as soon as a write is submitted to reduce average staleness, or to delay these invalidation messages to reduce peak server load.

3.4 Analysis

In this section, we analyze how consistency, performance and overhead are affected by five parameters of Volume Lease, which are volume size, volume lease length, object lease length, volume lease renewal policy and invalidation management. As we can see from this section, Volume Lease is a powerful framework because these parameters separate implementation from high level design and relate overhead and performance to consistency.

Volume Size The Volume Lease mechanism gains its efficiency by aggregating objects into volumes. Since a client can read any valid object within a volume during the volume lease period after the client renews the volume lease, the cost of a volume lease renewal is amortized over all objects contained in the volume. Large

volume sizes can reduce network overhead by reducing the number of volume lease renewals. They can also reduce read latency by reducing the chances that a client must wait for a volume lease renewal before reading a valid cache object.

However, increasing volume size is effective only if the objects contained in a volume are related. That is, if a client accesses an object in a volume, then this client is also likely to access another object in the same volume within the volume lease period. There is no benefit in grouping unrelated objects into a volume since reading of each object results in a volume lease renewal. In general, it is beneficial to group all objects from one web server into a volume.

Volume Lease Length Volume lease length can have a big impact on network overhead, consistency guarantees and read latency. A client must contact an origin server to renew a volume lease if the client tries to read an object from the volume when the volume lease expires, which can increase read latency. Thus long volume lease lengths can reduce read latency. Moreover, long volume lease lengths can reduce network overhead since it reduces the number of volume lease renewals. However, we cannot arbitrarily increase volume lease length to reduce overhead and read latency because the volume lease length is equal to the staleness bound provided by the Volume Lease protocols. Thus volume lease lengths cannot be bigger than the staleness bounds required by web services. In some scenarios, we want to make the volume lease length smaller than that required by web services to reduce write burstiness since servers do not need to send invalidation messages to clients if clients' volume leases expire. The reason is that a client is not allowed to read any object from a volume while the volume lease expires. Thus it is not necessary to send the invalidation messages before the client renews its volume lease.

Volume Lease Management Clients have two policies for requesting volume leases, *requesting on demand* or *prefetching.*

In requesting on demand, a client only requests a volume lease when it needs the volume lease to perform a read. Because requesting on demand only supplies volume leases when clients need them, it reduces server and network overhead. Moreover, requesting on demand reduces the number of clients holding a volume lease at a given time. Because on a write a server only needs to send invalidation messages to the clients holding the corresponding volume lease, requesting on demand reduces the number of invalidation messages.

However, requesting on demand presents a dilemma for web services that need both small staleness bound and low read latency. Because the volume lease length equals the staleness bound, a small staleness bound means a small volume lease length. When the volume lease length is small, the length of time for which a client holds a volume lease after each request is short. Thus, the smaller the volume lease length, the more volume lease requests that a client is forced to send before it can read objects, and the more volume lease requests, the higher the read latency. Prefetching volume leases can be used to address this dilemma. In this approach, a volume lease can be prefetched even though it is not needed immediately. This increases the time that a client holds a volume lease, reduces the chance that a client needs to request a volume lease before reading an object, and thus reduces read latency. The drawback of this approach is increasing server and network load because clients prefetch some volume leases which are not used.

Object Lease length Although both volume leases and object leases are leases, their purposes in the Volume Lease framework are different. Object leases are used

to track which objects are held by a client. An object with an object lease is usually also included in a volume so the staleness bound can be provided. Thus, object leases allow a server to generate the set of invalidation messages needed by a client. Not surprisingly, the tradeoff for object lease length is different from that of volume lease length. Because staleness bounds are generally provided by volume leases, object leases can be long to reduce the number of object lease renewals. Reducing object lease renewals can reduce server overhead and read latency. In many situations, object leases can even be infinite so object leases are always valid until they are invalidated. On the other hand, long object lease length does have a cost. Because servers must track clients' object leases, long object lease lengths can increase server state.

Invalidation Management To provide staleness bounds with Volume Lease protocols, servers do not need to promptly send out invalidation messages resulting from writes. Servers are only required to notify clients of invalidations before clients' volume leases are renewed. However, how promptly servers send invalidation messages does present a tradeoff between overhead and consistency. We consider three policies for delivering invalidation message: *prompt delivery*, *delayed delivery*, and *lazy delivery*. In prompt delivery, invalidation messages are always sent as soon as they are generated. In delayed delivery, an invalidation message is delivered immediately to a client only when the client holds a valid volume lease. In lazy delivery, invalidation messages are piggybacked to clients on other traffic between servers and clients before clients renew their volume leases. Although these three policies provide the same staleness bound for a given volume lease length, the average consistency and overhead differ. Moving from prompt delivery to delayed delivery does not hurt

average staleness since a client cannot read any object from a volume without the volume lease. However, delayed delivery does reduce network overhead since it allows some invalidation messages to be piggybacked to messages that grant clients volume leases. Lazy invalidation results in higher average staleness although it does not hurt staleness bound compared to the other two policies. The advantage of lazy invalidation delivery is lower server and network overhead because all invalidation messages are piggybacked.

3.5 Related Work

3.5.1 Client Polling

The current version of HTTP protocol, HTTP version 1.1 [41], provides a rich set of methods to maintain consistency. However, the core mechanism is client polling. Client polling is performed by a `GET` request with a `If-Modified-Since` header that specifies the last update time of the cached object. The server replies to a `GET` request with either a `Not Modified` status code or with a new version of the object with a `Last-Modified` header. The server can specify the TTL of an object with the `Expires` header. A client can choose to ignore this suggestion by polling more or less often. In cases where polling with TTL can not provide sufficient consistency for an object, the server can tag an object with `Cache-Control: no-cache`. Researchers [6, 34, 50, 69] have observed that HTTP 1.1 results in high latency and many stale reads because of client polling.

Based on their simulation study with write traces collected from web servers and synthetic read traces, Gwertzman and Seltzer [48] conclude that adaptive TTL

can provide good performance for the applications that can tolerate 4% stale reads.

Dingle and Partl [35] propose specific mechanisms to improve consistency and latency by augmenting the HTTP client polling protocols. These mechanisms include opportunistically refreshing a group of cached objects' TTLs to reduce access latency for these objects in case of communication between servers and caches and a more accurate method to calculating expiration times in a hierarchical cache to improve consistency. Dingle and Partl have also speculated that invalidation-base consistency protocols are probably unrealistic for web caching given a set of scalability challenges. We show in Chapter 5 that the Volume Lease framework can address these scalability challenges through its flexibility.

Cohen et al. [26] observe that polling cached objects when TTLs expire while the cached objects are still valid can significantly increase read latency. We have made similar observations. They propose to proactively refresh TTLs, which is the client polling consistency protocols counterpart of Volume Lease Prefetching as discussed in Chapter 4.

Periodic polling and refreshing have also be analyzed mathematically. Cohen and Kaplan [25] and Jung et. al [51] propose mathematical models for the performance of TTLs. Jung et. al [51] derive a closed formula for the performance of TTL and validate the formula with simulation. Cho and Garcia-Molina [**?**] mathematically derive the optimal strategy to synchronize a cache and server under a range of object functions in a model where writes and reads follows Poisson distributions.

Mogul [62] independently proposes a notion of grouping files into volumes to reduce the overhead of client polling. Cohen and Kaplan [27] study the use of volumes for prefetching and consistency. The consistency algorithms they examine

are best-effort algorithms based on client polling.

Krishnamurthy and Wills [36, 35, 55, 59, 60] examine ways to improve polling-based consistency by piggybacking optional invalidation messages on other traffic between a client and server. While their study doesn't provide the worst case staleness bounds, several techniques used in their study can also be exploited to improve performance and scalability of server-driven consistency. For example, our protocol allows servers to send delayed invalidations to clients by piggybacking them on top of other traffic between servers and clients.

3.5.2 Invalidation

A early study by Worrell [87] concludes invalidations can be cheaper than polls in a cache hierarchy environment. Belloum and Hertberger [10] have observed that it can costly for an invalidation protocol to continue to send invalidation messages to a cache after a cached object has been invalidated by a previous invalidation message. However, invalidation protocols can track cache state to avoid this drawback.

Evaluating a prototype with real-world traces, Liu and Cao [17] find that much of the bandwidth-saving benefit of adaptive TTL is derived from reading stale data by clients and find that server-driven consistency protocols can improve freshness of client reading without introducing significant overhead. However, they discover that there are two challenges in deploying server-driven consistency protocols, scalability and fault tolerance. In particular, they point to (i) the bursts of server load caused by invalidations sent when popular objects are written, (ii) the growth of the state that the server maintains to track the caches of its clients, and (iii) if partial failures prevent servers from contacting clients, then clients may continue to

return stale data. These three challenges are addressed in this dissertation.

Perret et. al [71] propose the Distributed Object Consistency Protocol (DOCP) that starts from default client polling for all objects and switches to Lease when an object becomes frequently accessed. They show that the Lease part of DOCP reduces latency and improves consistency for frequently accessed objects. They further argue that using a single protocol eliminates the confusions and mistakes caused by forcing users to choose from the large set of consistency mechanisms in HTTP [41]. We make the same observation on the benefit of Lease for frequently accessed objects. However, we observe that the drawback of Lease on less frequently accessed objects are bounded because one read only incurs at most one invalidation message. Volume Lease further reduces this drawback by allowing invalidation messages to be piggybacked onto volume lease granting messages.

Duvvuri et al. [37] examine adapting object lease length to reduce server state and messages with both analysis and simulation. These techniques can also be employed in the Volume Lease framework to improve scalability.

There are also a large body of studies [25, 25, 39, 58, 65, 76, 66, 67, 87, 92] on cache consistency in a hierarchical web cache environment. We compare these studies with our hierarchical cache consistency in Chapter 6.

3.5.3 Consistency Semantics

Kermarrec, et. al [53] have proposed to provide different consistency semantics for different web objects by encapsulating the strategies and implementation for maintaining consistency for each class of objects into a distributed object architecture. The consistency semantics in Volume Lease is per-volume instead of per-object. It is

permissible to group objects with less strict consistency requirements with a volume with more strict consistency guarantees. Volume Lease encourages systems to make volumes as big as possible to amortize the cost of volume lease renewals.

Some researchers also examine how to provide sequential consistency instead of the strong consistency used in this chapter. Bradley and Bestavros [13, 14] have proposed to use vector clocks to provide sequential consistency for web caches.

3.5.4 Industry Standard and Commercial Systems

Most commercial web caching systems provide only weak consistency. Many of them [2, 3] just implement the HTTP protocols. However, Akamai [1] uses a serial number to group the content that is always served from the same set of Akamai servers. A serial number combined with a type code that indicates the version of the content together implements expiration times in a different way from HTTP. Serial numbers and type codes can be used as volumes and epoch numbers of volumes if we implement Volume Lease protocols under the Akamai architecture.

However, commercial web caching systems [29, 57, 80, 79] are moving toward more efficient consistency protocols as a result of recent research effort in invalidation-base protocols, which include the work described in this dissertation. Noticeably, IBM Websphere [79] can provide a range of consistency guarantees with invalidations on a group objects called object groups or volumes. Tewari et. al [80] are proposing the consistency mechanisms in IBM Websphere as an IETF standard. Li et. al. [57] have proposed RUP, a protocol in which a client subscribes the invalidation messages on a group of objects by subscribing to a channel in server. Another IETF effort [29] is currently ongoing and has finished the requirement gathering

stage.

3.6 Conclusion

This chapter introduces a general framework for scalable cache consistency design, Volume Lease. Volume Lease decouples synchronization from invalidation. Synchronization is performed on sets of objects called volumes and amortizes the cost of synchronization over many objects in a volume. This framework captures the essential mechanisms for high-performance scalable cache consistency design and is still flexible enough to allow a wide design space.

Note that this chapter only provides an overview on Volume Lease. Various issues are closely examined in the following chapters. Chapter 4 evaluates the performance of Volume Lease, which includes latency and overhead when Volume Lease protocols provide a range of consistency semantics. Chapter 5 examines the scalability issues of the Volume Lease protocols. Chapter 6 examines how to implement Volume Lease protocols on more than one machines.

Chapter 4

Performance

In this chapter, we present some experimental results to show the effectiveness of Volume Lease. We choose the IBM Olympics workload and the Macy's workload for our study. Among all the performance measures, we focus on read latency.

4.1 Methodology

In this section, we describe the workloads and simulator used in our simulation study.

Workload

We use two workloads with significantly different characteristics to conduct our study: one is from an online news site, the other an e-commerce site. The first workload is taken from a major IBM Sporting and Event web site. This site contains about $60,000$ objects, and over 60% of the objects were dynamically generated [50]. The peak request rate for the Sporting and Event site exceeds 56.8 million requests

per day. The Sporting and Event web service was hosted on four geographically distributed web clusters; each of them served about one fourth of all requests. We take the server access log from one of these clusters on February 19th, 1998, and a modification log of dynamically generated objects for our simulation study. The server access log is in the Common Log Format [28], recording the IP address, timestamp, request line, status code, and object size for each access. This log contains about 9 million entries. The modification log for dynamically generated objects contains $20,549$ entries. Our second workload is taken from an e-commerce web site for a national retail store. This trace contains the web access logs from March 3, 2000 to March 9, 2000. It indicates that 177,978 clients accessed 69,608 URLs during the 7-days period. Overall, 9,504,953 requests are logged.

We classify URIs in our workloads as static or dynamic based on whether a URI contains the name of a service program. We can further divide static objects into image objects and non-image objects by examining the suffix of the URIs. In our Sporting and Event workload, 12% of the requests were made to pages dynamically generated by scripts. In our e-commerce workload, 6.4% of the requests are to dynamically generated objects.

We detect updates in two ways. First, the IBM Sporting and Event server logs the updates of the dynamically generated data and the modification log is available for our simulation study. Second, for the static data and dynamically generated data in our e-commerce workload, we use changes of object sizes to infer updates. We use this method because i) it is unlikely that two version of a document in our workload have the same size. ii) we observe that the sizes of the dynamically generated objects in the IBM Sporting and Event always change when they are

updated.

Although write records of static data and dynamic data generated by SSI scripts are not available, we infer writes to these types of objects by observing changes of object sizes in our trace. This method is highly precise because i) it is unlikely that two version of a static document have the same size in our workload. ii) we have a modification log of a set of objects and we find that the object size always change when there is a write.

In our study, all the objects from a web server form a volume. We expect this method of volume construction to be used in practice since it is simple to implement. Moreover, the bigger the volume, the lower the server load and the read latency since volume lease renewals are amortized over more objects. However, it is generally not beneficial to group objects from several servers into one volume since these objects can be disconnected from a clients independently. We treat all the reads in our server access logs the same and we do not care whether these reads are from a browser or a proxies.

We use the client requests in server access logs to drive our simulation since these logs are the only workloads that contain requests from large numbers of clients to single popular servers. All client reads to dynamically generated objects are included in server access logs since dynamically generated objects are not cached in the current web cache scheme. However, some reads to static objects are not recorded in server access logs since web access logs are taken while the clients are running some client polling protocols. There are two potential inaccuracies. First, a request is filtered out by the local cache when the TTL is valid and the cached object is also valid. This read would turn to a cache hit for both server-driven protocols with

long volume leases and client polling protocols with long TTLs.[1] Thus, it affects both server-driven consistency protocols and client polling protocols. Second, a client request is filtered out by the local cache when the TTL is valid and the cached object is invalidated. This request could result a cache miss for server driven protocols but a cache hit for long TTLs. Since the role of consistency protocols is to fetch the new data when data changes, it is preferable to suffer a cache miss to load valid data than to avoid a load by reading stale data. Moreover, a cache miss would eventually result in client polling protocols if the client continues to read the object after the TTL expires. Thus, our results are conservative estimates of the benefits of server-driven protocols over client polling protocols.

Simulator

To study the impact of different design decisions on the performance of server-driven consistency, we built a simulator that reads a web access log and a modification log and outputs read latency, server load, and network bandwidth consumption. Our simulator simulates both cache state (whether an object is cached) and consistency state (whether TTLs are valid or expired in client polling protocols and whether volume leases and object leases are valid in server-driven protocols). Given a workload, our simulator processes the web access log and the modification log in the order of timestamps, updating cache and consistency state and recording consistency, read load, server load information. To estimate server and network load, we only track the number of messages and total number of bytes in all messages, and we do not

[1]When the TTL and the volume lease are short, this hidden read is more likely to trigger a TTL refresh than a volume lease renewal since a volume lease renewal is amortized over many objects. Thus, we underestimate the cache miss rates more for client polling than for Volume Lease protocols in this case.

simulate network queuing and round-trip delays. Counting the number of messages and bytes provides a reasonable approximation to network load independent of many detailed network parameters such as packet sizes and congestion conditions.

4.2 Consistency

The Volume Lease protocols can provide both weak consistency and strong consistency. To provide weak consistency, a server sends out invalidation messages without waiting for acknowledgments and overwrites the old version of data with the new data immediately, and thus volume lease lengths correspond to worst-case staleness bounds. To provide strong consistency, a server waits for the acknowledgments for the invalidation messages before making the new data available, and thus volume lease lengths correspond to worst-case write delay.

When we discuss consistency guarantees in the following discussion, we focus on staleness bounds because we believe that staleness bounds are more relevant for this workload. However, note the experimental results can also be interpreted for strong consistency guarantees: I) the staleness bound is translated into worst-case writing blocking time for strong consistency; II) the cache hit rates are the same for both weak consistency and strong consistency; III) in absence of failures, the numbers of messages exchanged between servers and clients are the same for weak consistency and strong consistency because we count an invalidation to a client and the acknowledge as one message just as we count a poll and the reply to the poll as one message.

4.3 Basic Volume Lease

Our performance evaluation stresses read latency. To put the read latency results in perspective, we also examine the network costs of different protocols in terms of messages transmitted. Read latency is primarily determined by *local hit rate*, the fraction of reads that a client cannot serve locally. There are two conditions under which the system has to contact the server to satisfy a read. First, the current version of the requested object is not cached. We call this a *data miss*. Second, even if the right data is cached locally, the consistency protocol may need to contact the server to determine whether the cached copy is valid. We call this a *consistency miss*. Read latency for consistency misses may be orders of magnitude higher than that for local hits, especially when the network is congested or the server is busy. Thus, to a first-order approximation, reduction in miss rates yields the same proportion of reduction in average read latency.

Volume Lease has two advantages over traditional client-polling algorithms. First, it reduces the cost of providing a given worst case staleness by amortizing lease renewals across multiple objects in a volume. In particular, under a standard TTL algorithm, if a client references a set of objects whose TTLs have expired, each reference must go to the server to validate an object. In contrast, under a Volume Lease protocol, the first object reference will trigger a volume lease renewal message to the server, which will suffice to re-validate all of the cached objects. Second, Volume Lease enables the separation of average-case staleness from worst-case staleness by allowing servers to choose whether to notify clients when objects change.

Figures 4.1 through 4.6 illustrate the impact of different consistency param-

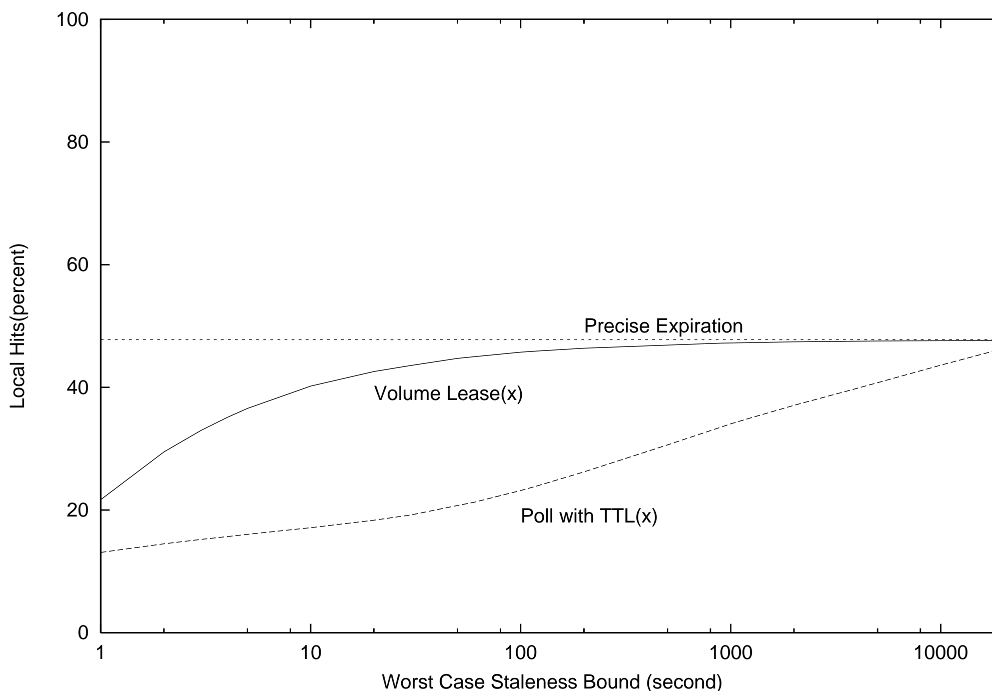


Figure 4.1: Local hit rates vs. worst case staleness bound of Volume Lease and TTL for the IBM workload. *Volume Lease(x)* represents the Volume Lease protocol with the volume lease length equal to x seconds; *Poll with TTL(x)* represent the traditional client polling protocol with the TTL equal to x seconds. Note that in the common case, Volume Lease caches are invalidated within a few seconds of an update independent of worst case staleness bounds.

eters for the IBM Sporting and Event workload and the e-commerce workload. In these figures, the x axis represents the worst-case staleness bounds for the Volume Lease protocol; these bounds correspond to the volume lease length for Volume Lease protocols and the TTL for TTL algorithms. The y axes in these figures show the fraction of local hits, network traffic, stale rate, and average staleness. The access patterns, such as read frequencies and numbers of repeated accesses of web objects, are quite different for the IBM web service and the e-commerce web service. Thus hit rates achieved by a given consistency protocol are different for these workloads.

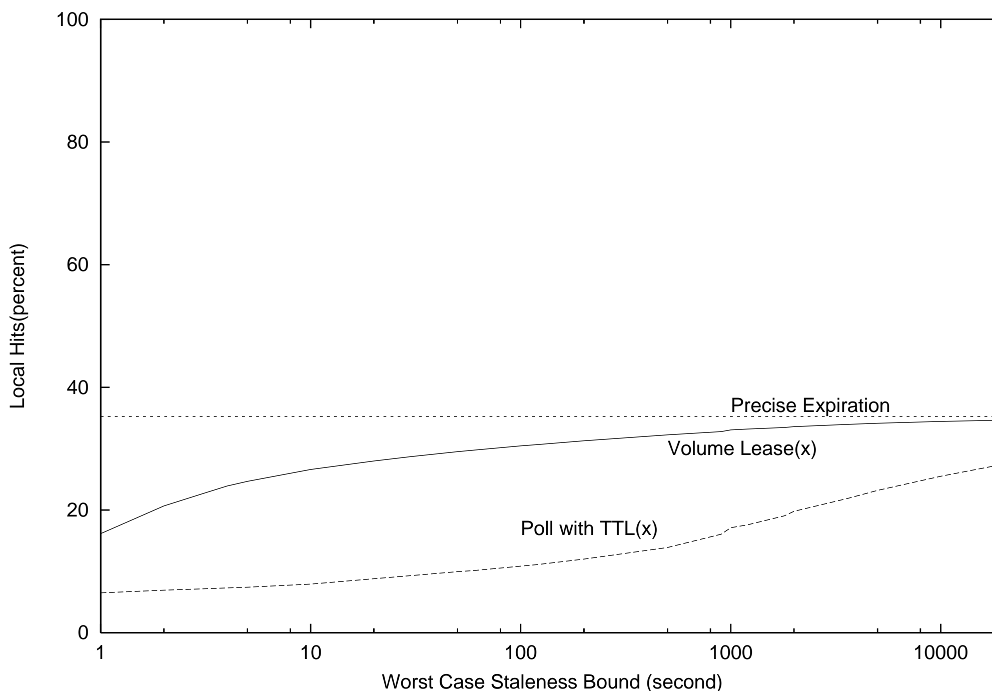


Figure 4.2: Local hit rates vs. worst case staleness bound of Volume Lease and TTL for the e-commerce workload.

We can see that Volume Lease protocols consistently provide larger advantages for both workloads due to the impact of amortizing lease renewal overheads across volumes. In particular, for short worst-case staleness bounds, Volume Lease protocols achieve significantly higher hit rates and incur lower server overheads compared to TTL algorithms. As indicated in Figures 4.1 and 4.4, providing worst-case staleness bounds of 100 seconds by Volume Lease protocols is cheaper than providing 10,000-second worst-case staleness bounds by traditional polling in terms of higher local hit-rate and lower network messages. And, as Figures 4.5 and 4.6 indicate, this comparison actually understates the advantages of Volume Lease protocols because for traditional polling algorithms, the number of stale reads and their average

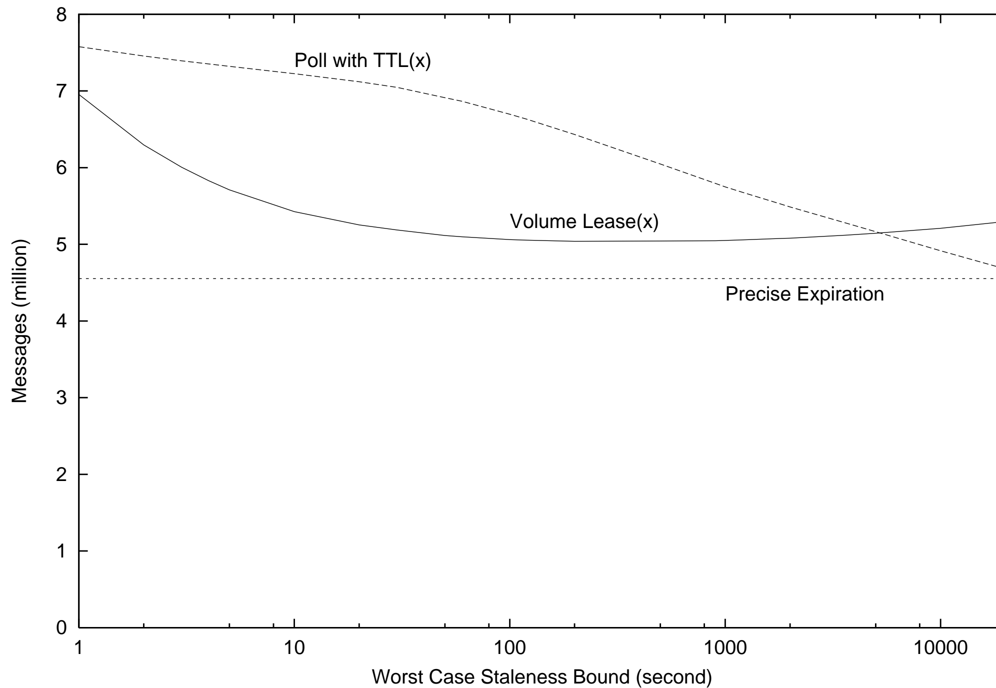


Figure 4.3: Number of messages vs. staleness bound of Volume Lease and TTL for the IBM workload.

staleness increase rapidly as the worst case bound increases. In contrast, Volume Lease schemes require that servers notify active clients of updates as soon as possible regardless of the worst-case staleness guarantees. Also notice that the server load decreases when volume lease length increases from several seconds up to several thousand seconds and then increases slightly. As volume lease length increases, the number of volume lease renewal decreases. However, the number of invalidation messages also increases: the server sends a client invalidation messages immediate instead of piggyback these invalidation messages if the client hold a volume lease. When the volume lease length is bigger than several thousand seconds, the effect of invalidation message increases overtakes the effect of volume lease renewal decreases.

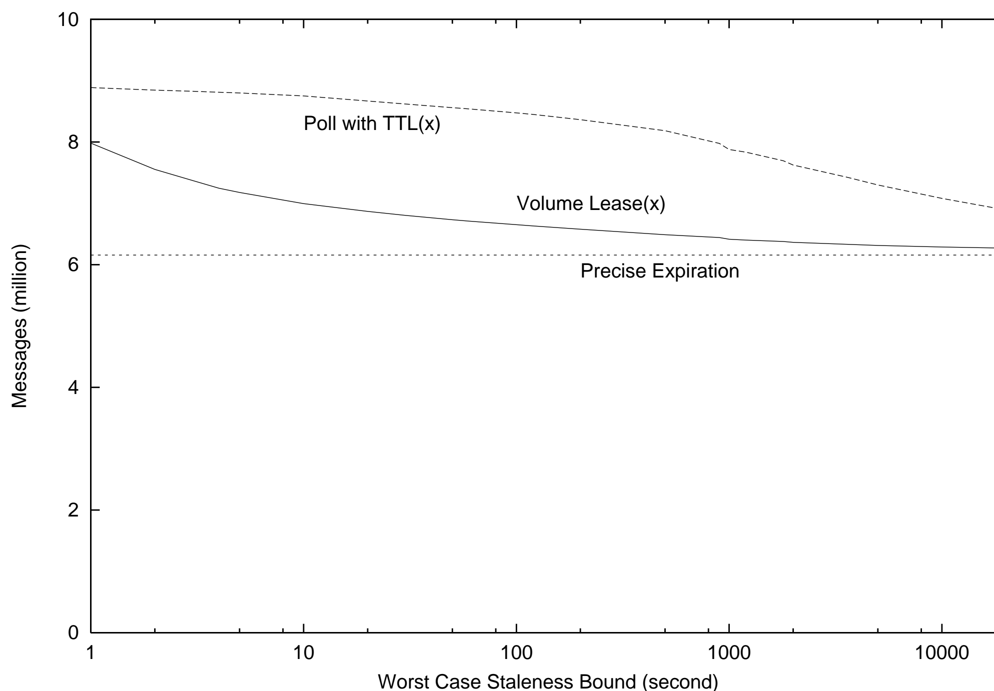


Figure 4.4: Number of messages vs. staleness bound of Volume Lease and TTL for the e-commerce workload.

However, server overhead with long volume lease is not much higher than that with the volume lease length of several thousand seconds. Most importantly, volume lease increases the hit rates by a factor of 1.5-3 compared to client polling protocols when the staleness bound is between 10 seconds and 100 second. Furthermore when the volume lease length exceeds 1000 seconds, Volume Lease protocols achieves local hit rates that are within 5% of Precise Expiration, the theoretical optimal protocol for maintaining cache consistency.

In Figures 4.7 through 4.9, we examine two key subsets of the requests in the workloads. We examine the response time and average staleness for the dynamically generated pages and the non-image objects fetched in the workload. Table 4.1 shows

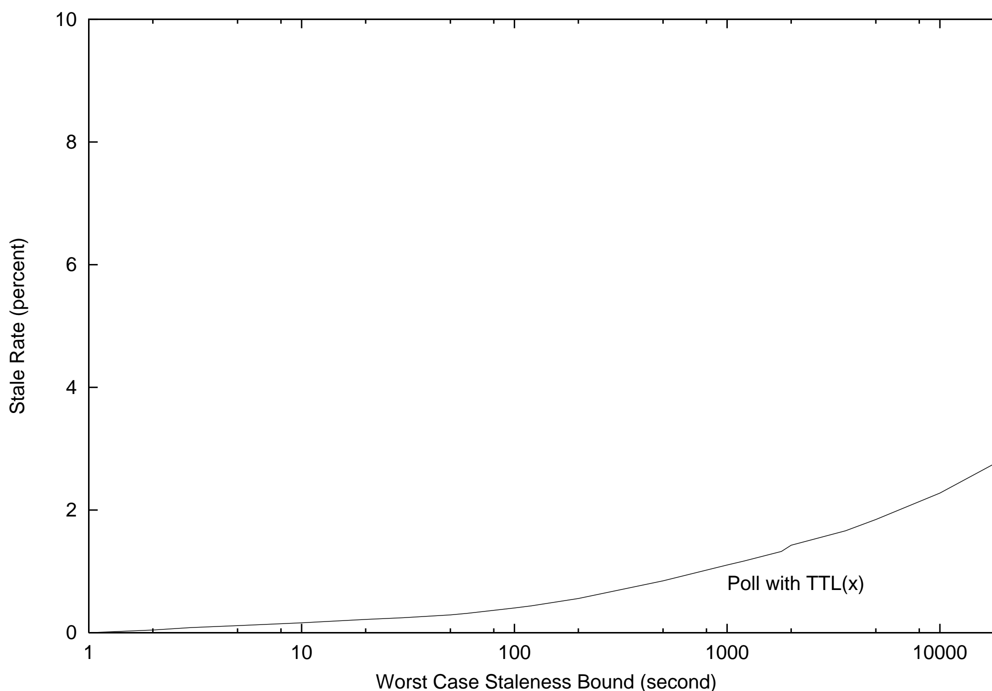


Figure 4.5: Stale rate vs. staleness bound of TTL for the IBM workloads.

that for the IBM workload the non-image objects account for 67.6% of all objects and requests to non-image objects account for 29.3% of all requests, while the dynamic objects account for 60.8% of all objects and requests to dynamic objects account for 12% of all requests. The fraction of requests to dynamic data rises to 40.9% when we exclude requests to image objects. Table 4.2 shows that there are also a significant number of requests to dynamically generated objects in our e-commerce workload.

The dynamic and other non-image data are of interest for two reasons. First, few current systems allow dynamically generated content to be cached. Our system provides a framework for doing so, and no studies to date have examined the impact of server-driven consistency on the cachability of dynamic data. Several

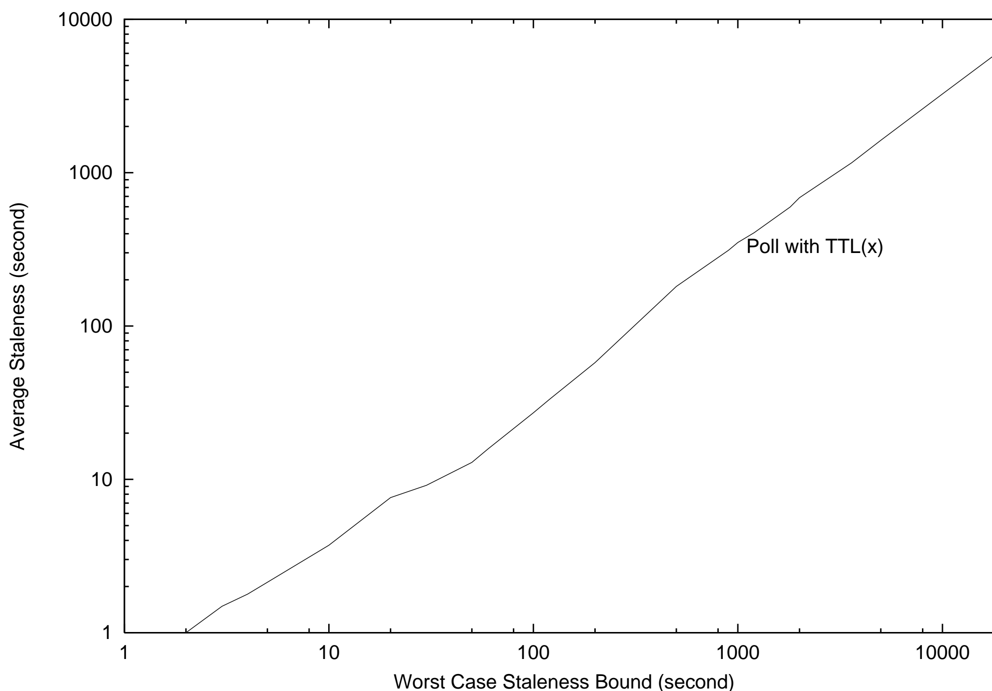


Figure 4.6: Average staleness vs. staleness bound of TTL for the IBM workload.

studies have suggested that uncachable data significantly limits achievable cache performance [85, 86], so reducing uncachable data is a key problem. Second, the cache behavior of these subsets of data may disproportionately affect end-user response time. This is because dynamically generated pages and non-image objects may form the bottleneck in response time since they must often be fetched before images may be fetched or rendered. In other words, the overall hit rate data shown in Figure 4.1 may not directly bear on end-user response time if a high hit rate to static images masks a poor hit rate to the HTML pages.

In current systems, four factors limit the cachability of dynamically generated data: (1) the need to determine which objects must be invalidated or updated when underlying data (e.g., databases) change [19], (2) the need for an efficient

	Object		Request	
	Number	Percent	Number	Percent
image	9027	32.4	6165803	70.7
non-image	18857	67.6	2553543	29.3
dynamic	16960	60.8	1044712	12.0
other non-image	1897	6.8	1508831	17.3
total	27884	100	8719346	100

Table 4.1: Classifying objects and requests in the IBM trace according to URL types.

	Object		Request	
	Number	Percent	Number	Percent
image	9255	13.3	8004528	84.2
non-image	60353	86.7	1500425	15.8
dynamic	59083	84.9	606073	6.4
other non-image	1270	1.8	894352	9.4
total	69608	100	9504953	100

Table 4.2: Classifying objects and requests in the e-commerce trace according to URL types.

cache consistency protocol, (3) the inherent limits to caching that arise when data change rapidly, and (4) privacy requirements that prevent some dynamic data from being cached at proxy caches. As a result, most current systems use cache control metadata to disable caching for dynamically generated data. Two mechanisms allow our system to cache dynamically generated data effectively. First, our system provides an efficient method for identifying web pages that must be invalidated when underlying data change. Second, as Figures 4.7 through 4.9 indicate, Volume Lease strategies can significantly increase the hit rate for both dynamic pages and for the "bottleneck" non-image pages. Simulations with our e-commerce workload yield similar results.

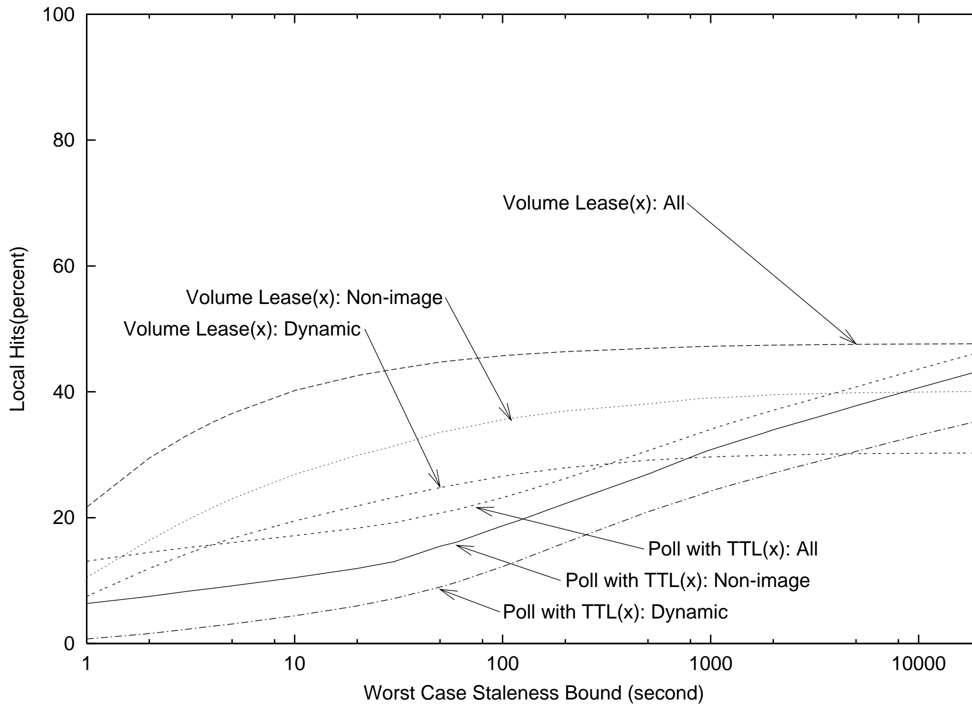


Figure 4.7: Local hit rates vs. staleness bound for TTL for the IBM workload. Note that in the common case, Volume Lease caches are invalidated within a few seconds of an update independent of worst case staleness bounds.

Finally, the figures quantify the third limitation. One concern about caching dynamic objects is that dynamic objects may change so quickly that caching them would be ineffective. This concern appears justified at least for client polling protocols. To reduce stale read rates for dynamic objects in client polling, clients have to frequently resynchronize with servers by using short TTLs, which leads to low cache hit rates. As shown in Figures 4.8 through 4.9, client polling leads to high stale hit rates and high average staleness when the TTL exceeds 1000 seconds. However, the cache hit rate is low when the TTL is small. Fortunately, Volume Lease allows us to eliminate stale reads in the common case and to achieve high cache hit rates. Hit

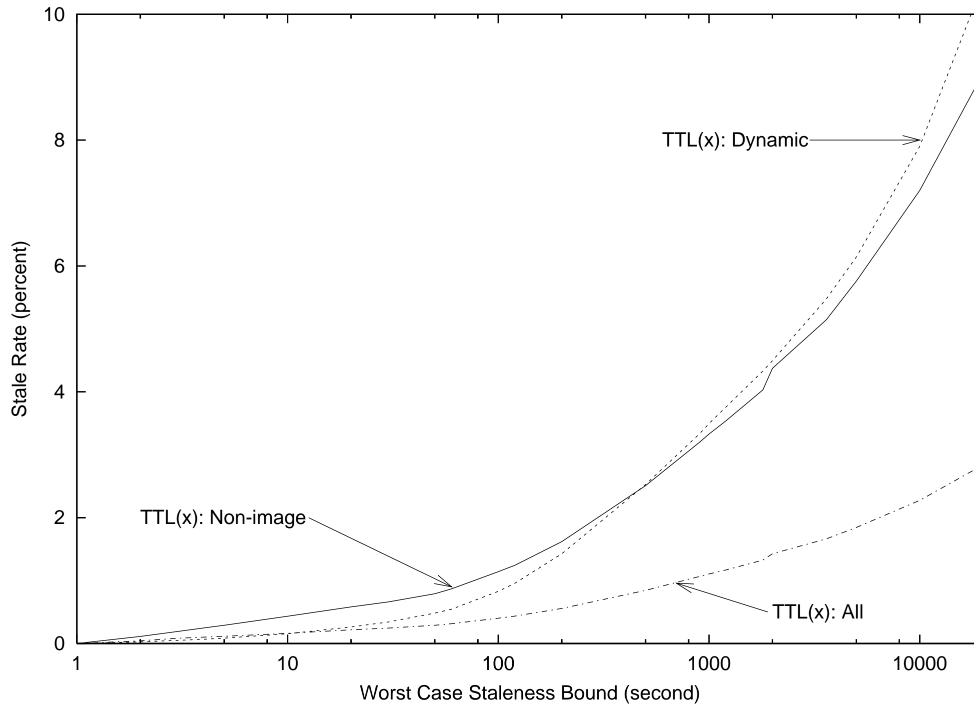


Figure 4.8: Stale rate vs. staleness bound for TTL for the IBM workload.

rates for dynamic objects are lower than for all objects. However, as many as 25% of reads to dynamically-generated data can be returned locally with long leases, which increases the local hit rate for non-image data by up to 10%. Since the local hit rate of non-image data may determine the actual response time experienced by users, caching dynamic data with server-driven consistency can improve cache performance by as much as 10%. Further performance improvements can be made by prefetching up-to-date versions of dynamically-generated objects after the cached versions have been invalidated.

Notice that Figure 4.7 shows that dynamic pages and non-image pages are significantly more sensitive to short volume lease lengths than average pages. This sensitivity supports the hypothesis that these pages represent "bottlenecks" to dis-

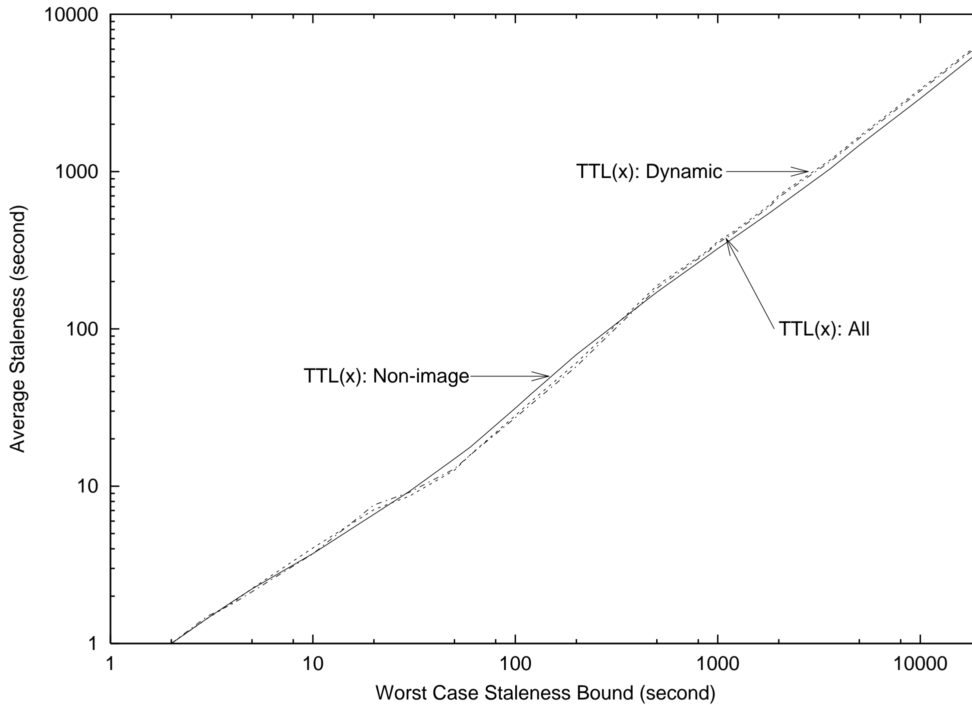


Figure 4.9: Average staleness vs. staleness bound for TTL for the IBM workload.

playing other images; dynamic pages and non-image pages are particularly likely to cause a miss due to volume lease renewal because they are often the first elements fetched when a burst of associated objects are fetched in a group. In the next subsection, we examine techniques for reducing the hit rate impact of short worst-case guarantees.

As mentioned in the previous section, these results here can be easily interpreted for strong consistency when we view worst-case staleness bounds as worst-case write delay. Thus, we can draw the following conclusions for strong consistency: I) the basic Volume Lease protocol can reduce read latency compared to TTL even when providing strong consistency, although the write delay can be as high as volume lease lengths in the worst case; II) the basic Volume Lease protocol does not

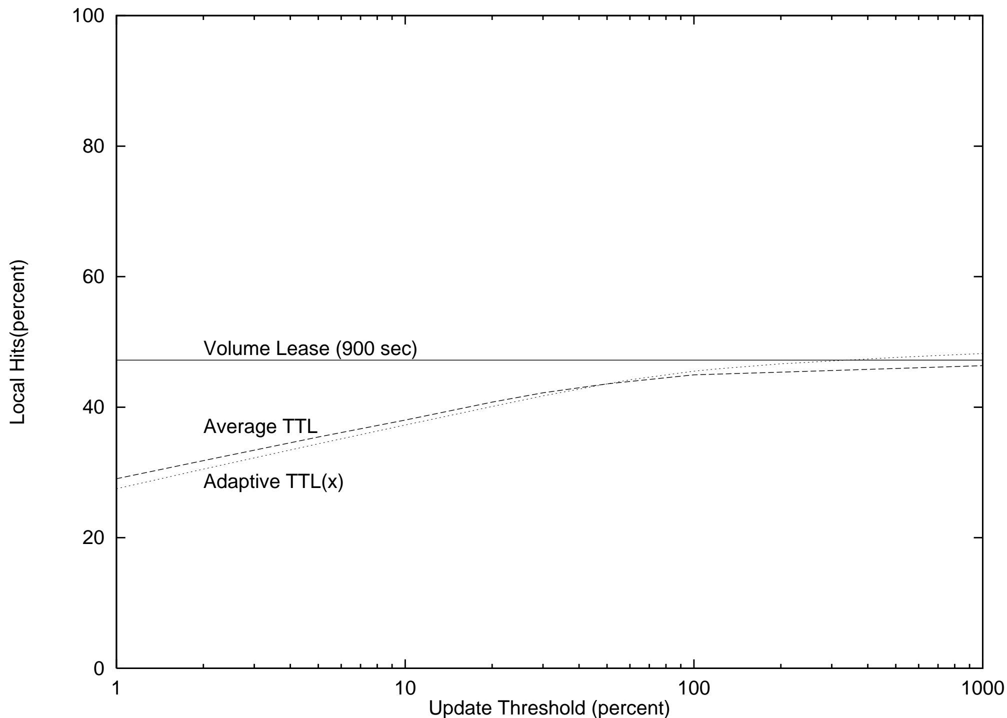


Figure 4.10: Local hit rates of Adaptive TTL, Average TTL, and Volume Lease(900) for the IBM workload.

notably increase server and network overhead compared to TTL even when providing strong consistency; and III) Caching dynamically changing data with the basic Volume Lease protocol can still be beneficial, although the benefit of caching decreases as changes become more frequent.

4.4 Adaptive TTL and Average TTL

In this section, we evaluate the set of client polling protocols that only intend to provide low stale read rates. Since these protocols do not provide worst case staleness bounds, they are not compared against Volume Lease protocols in the previous sections. One such TTL algorithm is adaptive TTL [18]. Adaptive TTL is designed to

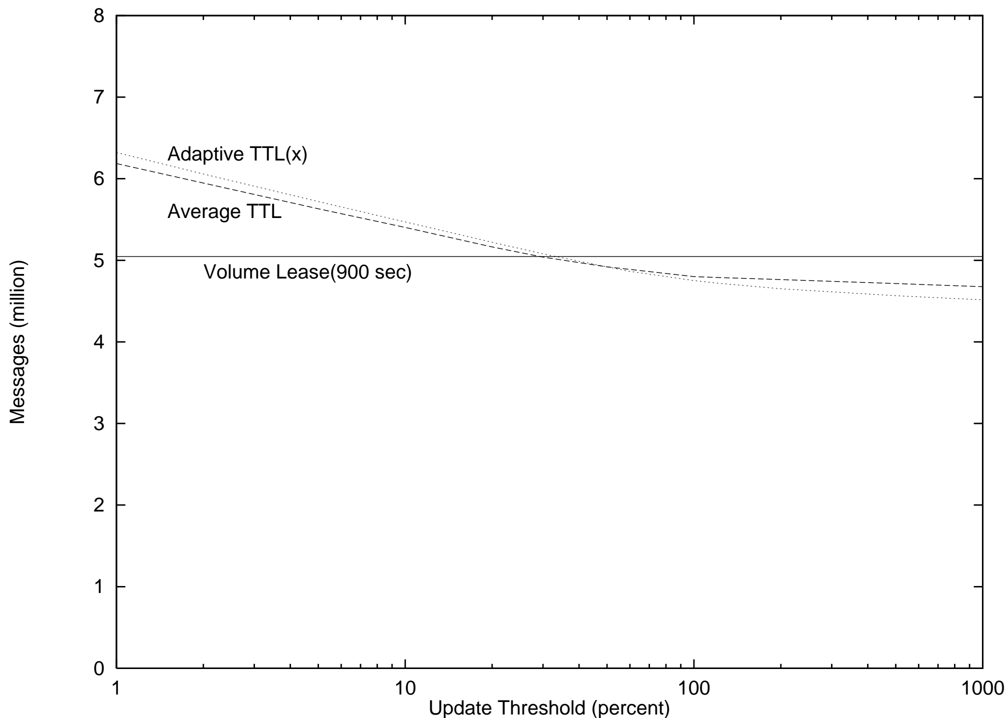


Figure 4.11: Number of network messages generated by Adaptive TTL, Average TTL, and Volume Lease(900)for the IBM workload.

reduce stale reads by adjusting client polling frequencies to modification frequencies of web objects. In adaptive TTL, the TTL is set to be proportional to an object's age, the amount of time since the last modification. We call this proportion the *update threshold* of the adaptive TTL. The second algorithm is Average TTL. In Average TTL, the TTL of an object is set to be the product of its average life time over the entire trace and the update threshold. Here an object's life is defined as the amount of time between two adjacent updates. Average TTL can be a reasonable approach because researchers have observed that update patterns of some web objects can be closely approximated by a Poisson Distribution with fixed modification frequencies [16, 23].

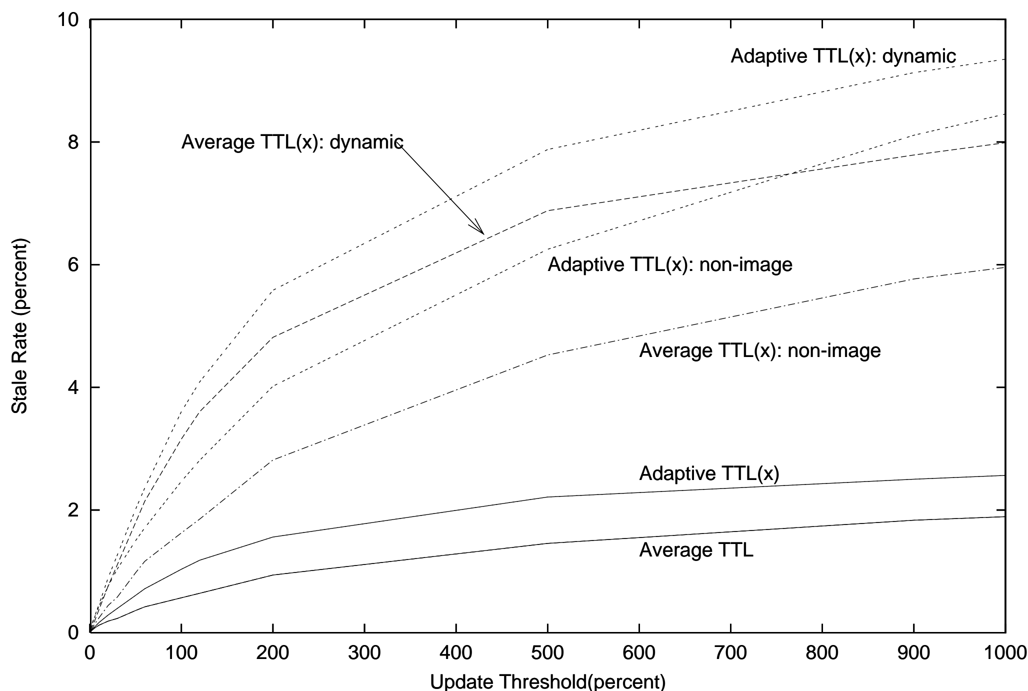


Figure 4.12: Stale hit rates of all the objects, the non-image objects, and the dynamic objects running Adaptive TTL, Average TTL for the IBM workload.

As shown in Figures 4.10 through 4.12, Volume Lease protocols with reasonable volume length can achieve higher local hit rates than average TTL and adaptive TTL with higher than 100% of update threshold while only introducing insignificant amount of additional network overhead. While adaptive TTL with higher than 100% threshold and Average TTL introduce significant stale read rates, Volume Lease protocols always return fresh data in absence of failures. The results from simulations with our e-commerce workload are similar. Thus, Volume Lease protocols are attractive even for applications that do not need worst-case staleness bound guarantees.

4.5 Prefetching/Pushing Lease Renewals

The above experiments assume that when a volume lease expires, the next request must go to the server to renew it. A potential optimization is to prefetch or push volume lease renewals to clients before their leases expire. For example, a client whose volume lease is about to expire might piggyback a volume lease renewal request on its next message to the server, or it might send an additional volume lease renewal prefetch request even if no requests for the server are pending. Alternately, servers might periodically push volume lease renewals. More aggressive prefetching keeps clients and servers synchronized for longer periods of time, increases cache hit rates, but increases network costs, server load, and client load.

In Figure 4.13 and Figure 4.14, we examine the relationship between pushing or prefetching renewals, read latency, and network overhead. In interpreting these graphs, consider that in order to improve read latency by a given amount, one could increase the volume lease length by a factor of K. Alternatively, one could get the same improvement in read latency by prefetching the lease K times as it expires. We would expect that most services would choose the worst case staleness guarantee they desire and then add volume lease prefetching if the improvement in read latency justifies the increase in network overhead.

As illustrated in Figure 4.13, volume lease pull or push can achieve higher local hit rates than the basic Volume Lease protocol for the same freshness bound. In a *push-K* algorithm, if a client is idle when a demand-fetched volume lease expires, the client prefetches or the server pushes to the client up to $K-1$ successive volume lease renewals. Thus, if each volume renewal is for length V, the volume lease remains valid for $K \cdot V$ units of time after a client becomes idle. If a client's accesses

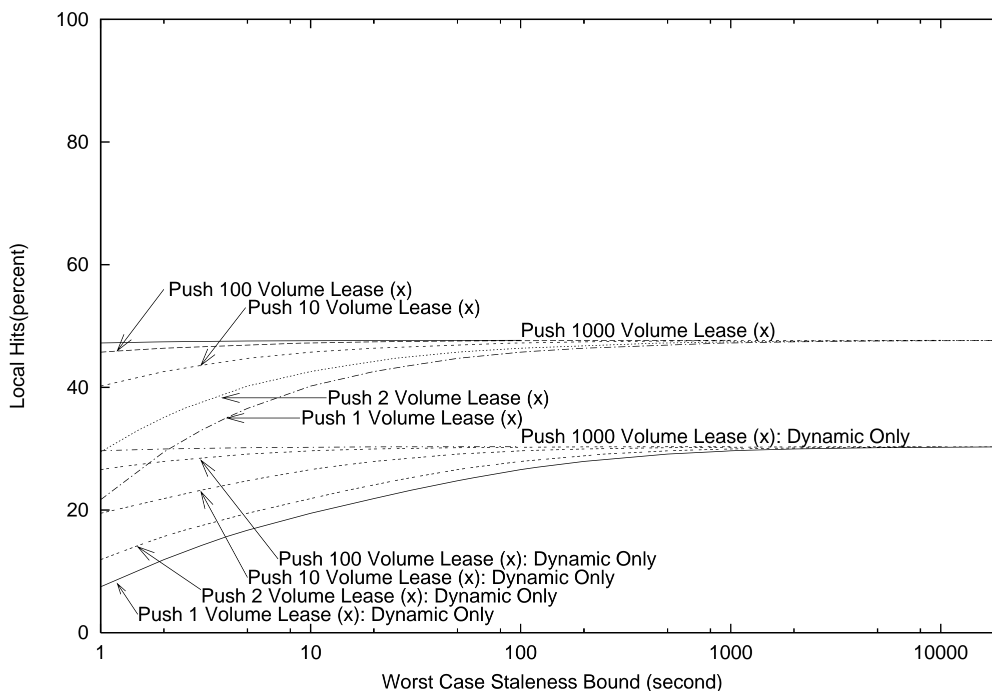


Figure 4.13: Prefetching/pushing volume leases to reduce read latency for the IBM workload.Push y Volume Leases (x) represents a Volume Lease protocol with volume lease length equal to x which pushes or prefetches volume leases $y-1$ times after a volume lease expires.

to the server resume during that period, they are not delayed by the need for an initial volume lease renewal request.

Both *push-2* and *push-10* shift the basic Volume Lease curve upward for short volume leases and larger values of K increase these shifts. Also note that the benefits are larger for the dynamic elements in the workload, suggesting that prefetching may improve access to the bottleneck elements of a page.

However, pulling or pushing extra volume lease renewals does increase client load, server load, and network overhead. This overhead increases with the number of renewals prefetched after a client's accesses to a volume cease. For a given number

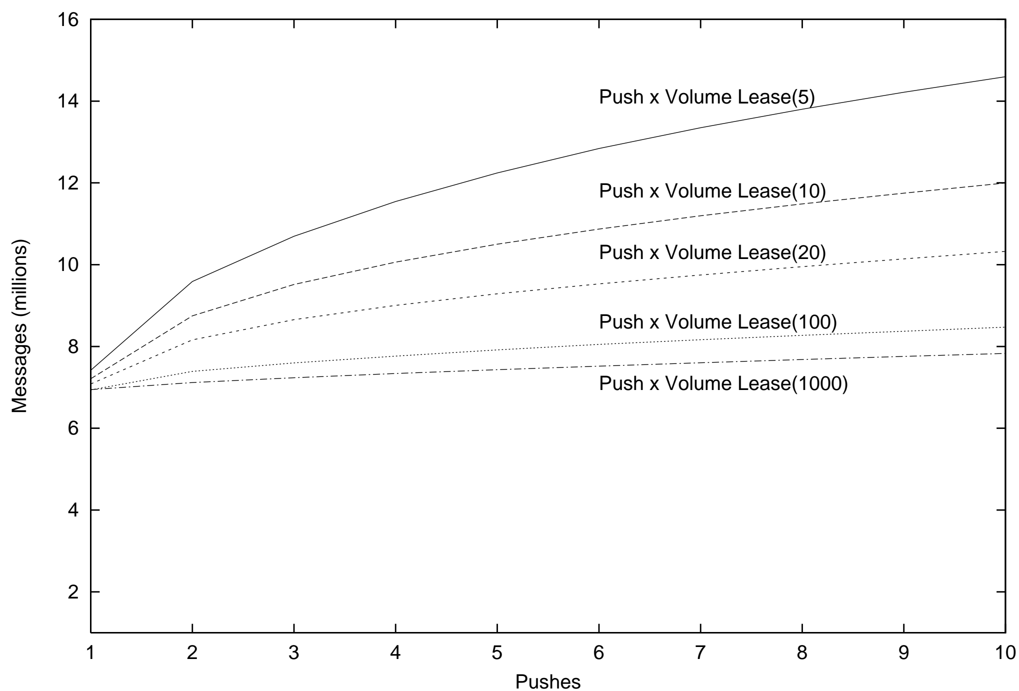


Figure 4.14: Cost of pushing volume leases for the IBM workload.

of renewals, this overhead is lower for long volume leases than for short ones.

As discussed in Section 4.2, these results can also be interpreted for strong consistency. Prefetching volume leases can be used to improve read latency when short worst-case write delay is required.

4.6 Conclusion

This chapter evaluates the performance of Volume Lease protocols. We show that Volume Lease can provide a range of consistency efficiently. In particular, our experimental results demonstrate that Volume Lease incurs little overhead, reduces user response time, and improves scalability compared to traditional client polling

protocols. Moreover the performance of Volume Lease is close to theoretical optimal.

Chapter 5

Scalability

5.1 Overview

In this chapter we discuss the scalability challenges of Volume Lease protocols and how we address them. Today's popular web servers can have millions of clients; these clients are connected to servers through millions of network links and routers. These client machines and networks can crash as a result of hardware failures, but mostly software bugs. Moreover, the number of clients that access a server can grow over time. Such a large scale system can have three scalability challenges: fault tolerance, server state, and server burstiness.

How well a system can tolerate failures can determine how well this system can scale up. The larger a distributed system, the higher the possibility of partial failures. In a large scale web service with a large number of clients, the possibility that some clients either crash or are disconnected from the server is the possibility of failures for one client times the number of clients. A web service with millions of clients must almost always operate under partial failures. Thus, preventing partial

failures from degrading system performance is critical for scalability. Fault tolerance in large scale Volume Lease includes two aspects: the performance of the system with partial failures and the efficiency of the system to recover from failures. The flexibility of Volume Lease allows us to effectively address these two aspects. More specifically, Volume Lease enables a system to continue to produce high performance even when some clients or networks to some clients are faulty; as these faulty clients and networks recover, disconnected clients can resynchronize with server quickly and efficiently to resume service.

The second scalability issue is server state, the potentially overwhelming amount of memory that web servers may need to use in order to track the information about clients. This information includes the volume lease and the number of object leases held by each client. A server only needs to track its clients' volume leases by tracking the client's ID and the expiration times of these volume leases, which only consumes a small constant amount of memory for each server. The object lease state, however, can grow to be proportional to the total number of clients times the total number of objects. In this chapter, we show that the average-case server state is not overwhelming even for a popular web server, the IBM Olympics server. Even though server state can potentially grow over time as more clients access a server, dropping the state of idle clients allows us to place a hard limit on the growth of server state without imposing significant overhead.

The third scalability issue is peak server load. In Volume Lease, peak server load usually occurs during writes. Recall from the last chapter that in response to a write the server sends out invalidation messages to clients caching the data. For popular objects, the number of invalidation messages can potentially equal the total

number of clients. Recall also that the fundamental constraint governing invalidation message delivery in Volume Lease only requires that invalidations messages to be delivered by the time that a client gets an new volume lease. The constraint allows two mechanisms to reduce peak server loads: delayed invalidations and background invalidations. Delayed invalidations allow a server to buffer invalidation messages to a client if the volume lease of this client has expired. These buffered invalidation messages can be piggybacked to the message that grants a new volume lease to this client without incurring the server overhead of individual messages. Delayed invalidation reduces peak server load without compromising any consistency, because without a volume lease, a client can not read any object including the invalidated object. Background invalidations allow us to delay an invalidation message to a client even when a client holds a volume lease. Although hurting average-case consistency, background invalidations still uphold the worst-case staleness bound in a system providing weak consistency since the constraints of Volume Lease are still met. Background invalidations allow us to delay an invalidation message until we have spare cycles to serve it and thus place a hard limit on server load.

In the rest of this chapter, we discuss how to address the scalability challenges. Our general approach is to use the flexibility of volume lease to limit the impact of partial failures to performance, the state to be held by one machine, and the peak server load. The flexibility allows our system to perform well even with partial failures. Moreover, we can place a hard limit on server state and peak server load. The rest of this chapter is organized as follows. Section 5.2 , we describe how failures are handled in Volume Lease. Section 5.3 discusses various techniques to reduce server state. Section 5.4 discusses various techniques to reduce peak server

load.

5.2 Fault Tolerance

As a system scales up, partial failures become common. In a large-scale web service with millions of clients, it is almost certain that at any time some clients are either down or are unable to communicate with the server because of network partitions. Thus, the ability for such a system to handle failures efficiently is essential for performance. Fault tolerance here includes two aspects: the performance of the system with partial failures and the efficiency of the system to recover from failures. These two aspects are to be discussed in the next two subsections.

5.2.1 Performance with Partial Failures

Large scale Volume Lease should perform well even under partial failures. In the ideal case, the failure-free part of a system can perform without any effect from the failed part as if the system is only composed of the failure-free part. There are three kinds of partial failures in a large scale web service: server failures, client failures, and network partitions.

In server failures, a server is down. Note that the probability of server failures is typically smaller than that of client failures, because in a web service, there is only one server and many clients. Without the server, a client cannot get new leases. However, this client can continue to serve the objects with valid object leases when the volume lease of this client does not expire. After this volume lease expires, the client cannot serve any data without violating the prespecified consistency guarantee. However, this client can still allow users to decide if they

want to read data from cache even without the prespecified consistency guarantee.

In client failures, some caches are down because of either software failures or hardware failures. To provide strong consistency, the server can simply wait for the expiration of the volume leases held by faulty clients before allowing a write to complete. Thus, the reads by non-faulty clients can be blocked by no more than the volume lease length amount of time. In a system providing only a staleness bound, the server can continue to issue volume leases and object leases to non-faulty clients even if the server cannot contact some clients. As a result, the non-faulty part of the system performs as if the system only contains these non-faulty clients and the faulty clients do not exist.

In network partitions, the server cannot communicate with some clients because the links from the server to these clients are down. We call these clients disconnected clients. The server cannot contact these disconnected clients and acts exactly like the server in the client failure scenario; these disconnected clients can neither renew volume leases and object leases nor receive invalidation messages from the server and behave exactly like these clients in the server failure scenario.

5.2.2 Recovery

In last section, we have discussed how the system perform under partial failures; in this section, we discuss how Volume Lease can recover after the underlying systems that Volume Lease runs on have recovered.

Most partial failures in the underlying systems can be recovered from quickly. In particular, most server and client failures are not the result of hardware failures, but software failure, from which the systems can recover quickly. [8] Most network

failures are also the result of software bugs and misconfigurations which allow quick recovery. Additionally, the redundancy in large-scale networks can also help recover fast by quickly switching to redundant parts when a part failed. Resilient Overlay Networks [5] can quickly detect and route around network failures to recover from network partitions. Quick recovery of the underlying systems demand Volume Lease to recover quickly to reduce service down time. Additionally, to achieve scalability, servers may drop the state of idle clients to limit server state, which places these clients into the same state as client crashing.

Failures and dropping idle clients' state cause the synchronization between servers and clients to be lost. That is, a client may fail to know that some cached objects are no longer valid because the server fail to send invalidation messages or the server fail to receive these messages. When failures of the underlying systems are recovered or idle clients wake up and start to read, a server and its clients need to be resynchronized to provide consistency under Volume Lease.

These disconnections can be roughly divided into two groups based on whether the consistency state before a disconnection survives the disconnection. *State-preserving disconnections* caused by network partitions preserve the consistency state prior to the disconnections. *State-losing disconnections* caused by server crashes, client crashes, and deliberate protocol disconnections result in loss of consistency state. In this section, we systematically study the design space of resynchronization to recover from all these disconnections.

There are three potential policies to control how aggressively clients resynchronize with servers. At one extreme, *demand revalidation* marks all cached objects as potentially stale after reconnection and revalidates each object individually as it

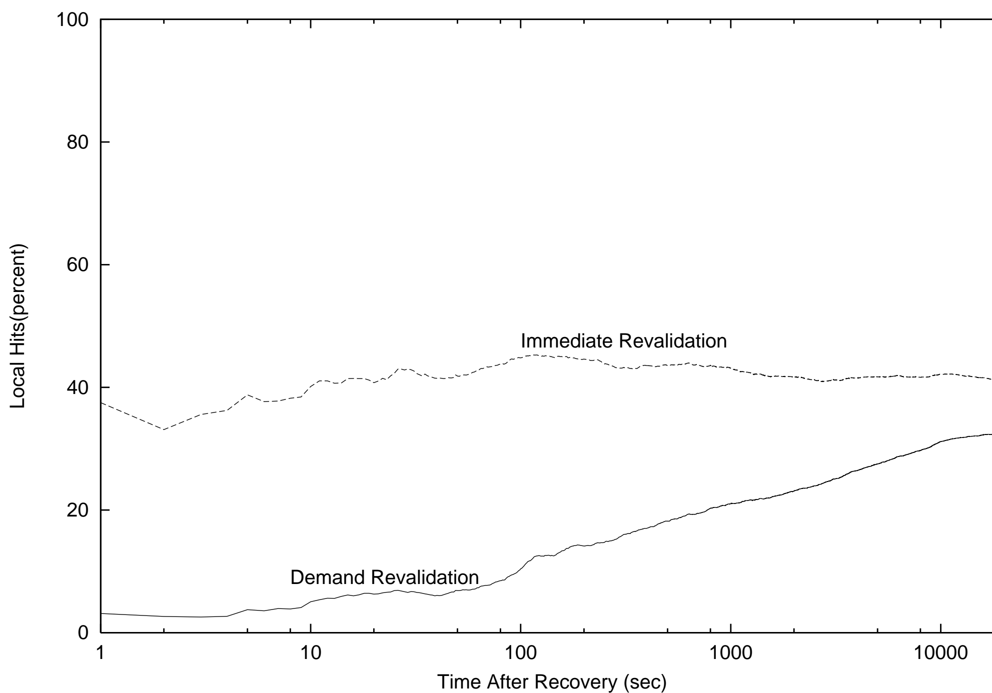


Figure 5.1: Hit rates after recovery for a disconnection of 1 second for the IBM workload.

is referenced. At the other extreme, *immediate revalidation* revalidates all cached objects immediately after reconnections to reduce the read latency associated with revalidating each object individually. When the overhead of revalidating all cached objects is high, immediate revalidation may delay clients' access to servers immediately after reconnections. To address this problem, *background revalidation* allows bulk revalidation to be processed in the background.

Figures 5.1 and 5.2 show that immediate revalidation achieves higher average local hit rates than demand revalidation. The performance disparity between immediate revalidation is larger immediately after failures and decreases over time as more cached objects are accessed and validated in demand revalidation. The

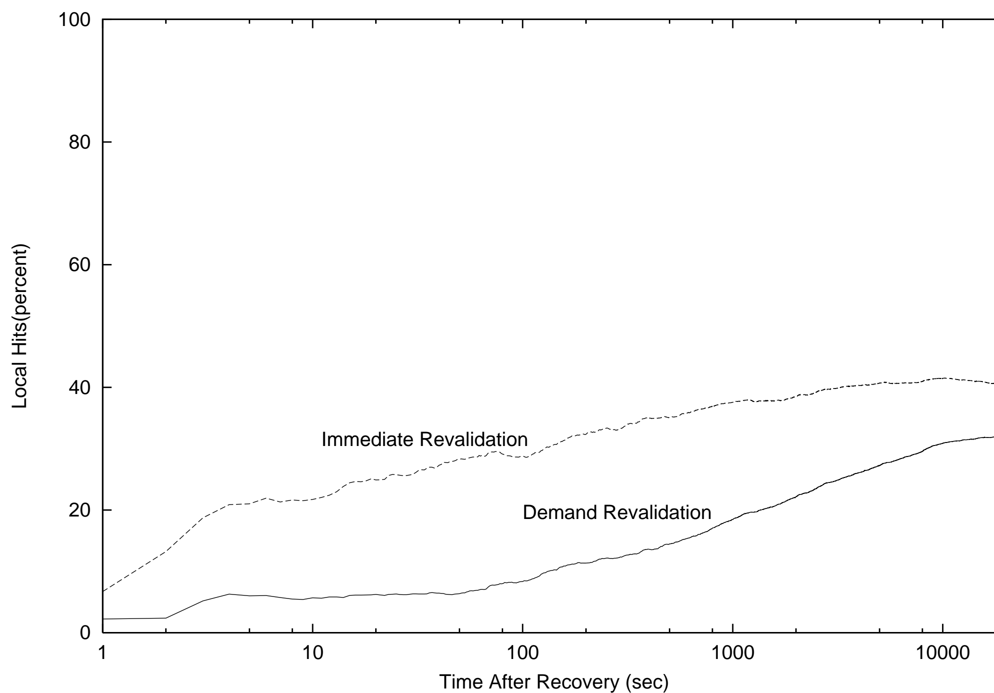


Figure 5.2: Hit rates after recovery for a disconnection of 1000 seconds for the IBM workload.

local hit rates of background revalidation would range between these two lines in the graph - they equal those of demand revalidation immediately after reconnection and would increase to those of immediate revalidation as background revalidation completes. The benefit of immediate and background revalidation is also affected by disconnection duration. When disconnection duration is short, the number of cached objects that are invalidated during disconnections is small. Moreover, because of read locality, the chance of reading these cached objects after recovery is high. Hence, as shown by Figures 5.1 and 5.2, the benefit of immediate and background revalidation is significant for short disconnections. Conversely, when a disconnection duration is long, demand revalidation may be sufficient.

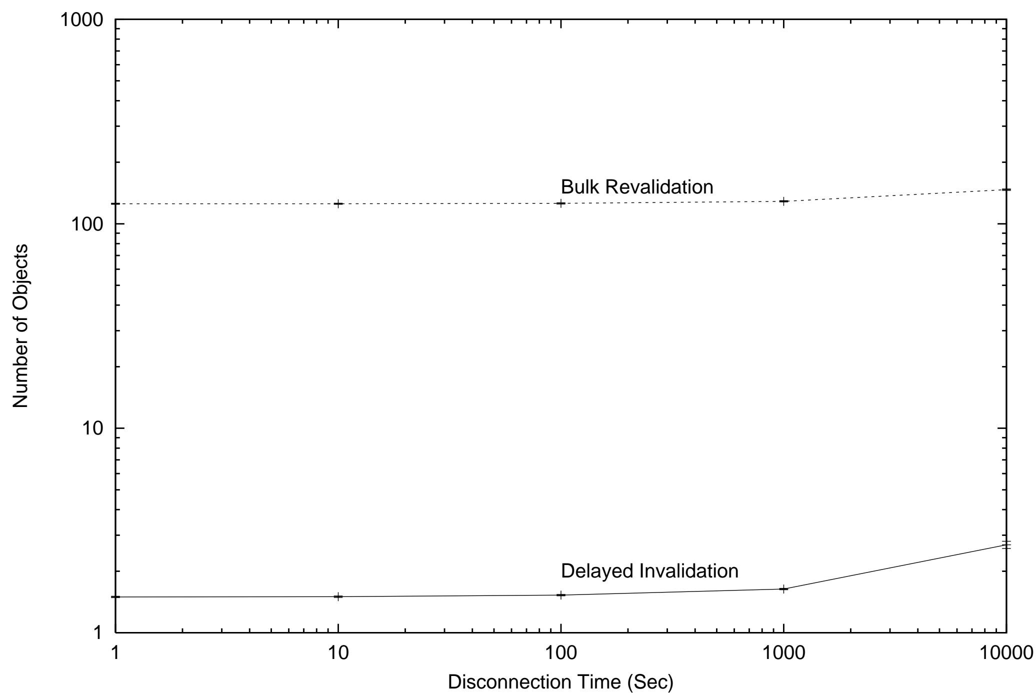


Figure 5.3: Resynchronization cost for bulk revalidation and delay invalidation for the IBM workload.

To implement demand revalidation, systems only need to detect reconnections and mark all cached objects as potentially stale by dropping all object leases. Revalidating cached objects in demand revalidation is the same as validating cached objects after the object leases expire. Two additional mechanisms can be added to support immediate or background revalidation. First, in *bulk revalidation*, a client simply sends a revalidation message containing requests to revalidate a collection of objects. The server processes each included request as it would have if it had been sent separately and on demand, except that the server replies with a bulk revalidation message containing object leases for all unchanged objects and invalidations for objects that have changed. Second, in *delayed invalidation*, the server buffers the

invalidations that should be sent to a client when a network partition makes a client unreachable from the server or when the server decides to delay sending invalidation messages to an idle client to reduce server load. When the server receives a volume lease request message from the client, the server piggybacks the buffered invalidations on the reply message granting the client a volume lease. The client applies these buffered invalidations to resynchronize with the server. Note that delayed invalidation may only be used after state-preserving disconnections.

The overhead of bulk revalidation and delayed invalidation primarily depends on the number of cached objects and on the number of objects invalidated during the disconnection. In the case of bulk revalidation, server load and network bandwidth are proportional to the number of cached objects; in delayed invalidation, they are instead determined by the number of invalidated objects. As Figure 5.3 shows, bulk revalidation must examine an average of more than 100 objects. For some recovered clients, several thousand objects must be compared during bulk revalidation. Delayed invalidation can be used to reduce the cost of immediate revalidation for state preserving disconnection, since the number of cached objects is two orders of magnitude less than the number of invalidated objects for disconnections shorter than 1000 seconds. Unfortunately, for state-losing disconnections, delayed invalidation is not an option. Because bulk revalidation may have to revalidate hundreds or thousands of objects, the system should support background revalidation rather than relying solely on immediate revalidation. These conclusions are also applied to the results from simulations with our e-commerce workload.

In conclusion, server-driven consistency protocols must implement some resynchronization mechanisms for fault tolerance and scalability. Demand resynchroniza-

tion is a good default choice since it handles all disconnections and is simple to implement. Background bulk revalidation may be needed to reduce read latency when recovering from short disconnections, and delayed invalidation may be desirable to reduce resynchronization overheads for short state-preserving disconnections.

5.3 Server State

In Volume Lease, the server needs to track both object leases and volume leases of clients. The server state concerning object leases is generally predominant because a client can hold many object leases from a server, while it can only have one volume lease per server. A popular web server can host millions of objects and can be accessed by millions of clients. If we do not design our system carefully, terabytes may be needed to track object leases. This amount of memory is obviously beyond the limit which can be provided by any commercially available machine. There are reasons to believe that the average case object lease state may not be too bad because not all clients hold object leases on all the objects. Small average-case server state means that a system can perform well most of the time. However, our system still needs to ensure that in the worst case, all the server state can still be handled with the limited amount of memory that a server has, which means techniques are needed to put a hard limit on server state.

Server state can be limited with Forgetting Idle Clients, in which the server throw away the state associated with the clients that have been idle to limit the server state. The server can choose to throw away the state of the clients that have been idle for a longer time first to reduce penalty. As the time for which a client has been idle increases, the likelihood that the client will continue to be idle increases.

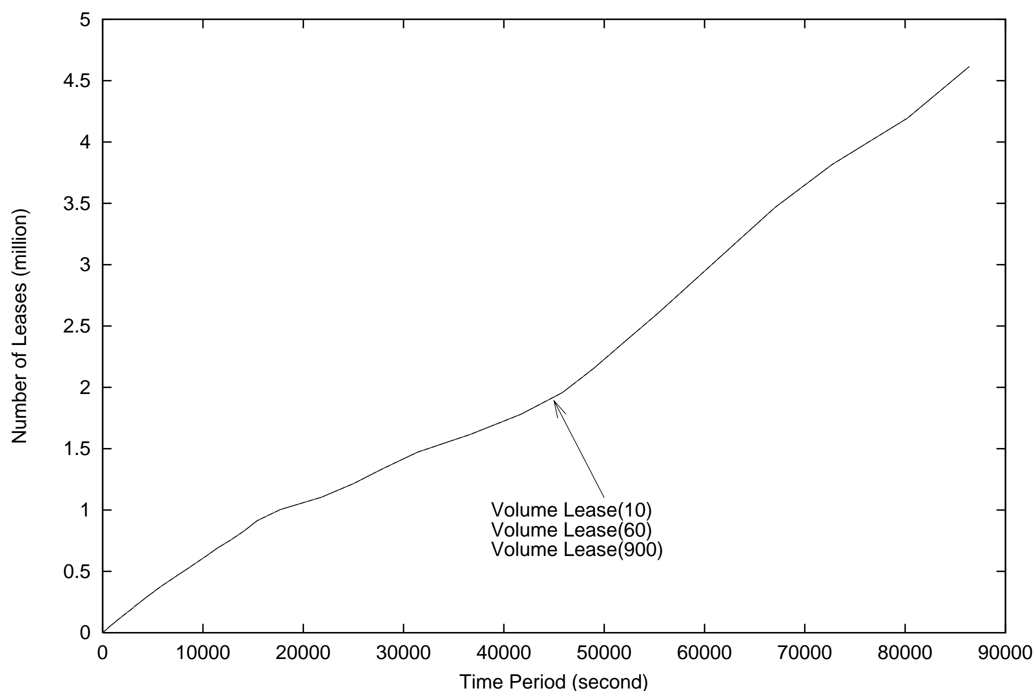


Figure 5.4: Callback state increases with elapsed time for the IBM workload.

When an idle client wakes up and starts to read again, the client and the server need to be resynchronized to resume operations. One advantage of Forgetting Idle Clients is that it can adapt according to the available memory to limit server state of each server.

In the rest of this section, we measure the amount of server state in a large scale web server, the IBM Olympics web server. We measure both the amount of server state in the average case and that in the worst case. Then we discuss how Forgetting Idle Clients can put a hard limit on server state. All the experimental results presented in the rest of this section are drew from the simulation study with the IBM Olympics workload.

Figure 5.4 shows the number of object leases stored as a function of elapsed

time in the trace.Note that whether the server holds an object lease on an object for a client is determined only by whether the client caches the object and is independent of volume lease length. For the time period covered in our trace, server memory consumption increases linearly. Although, for a longer trace, higher hit rates might reduce the rate of growth, for the Zipf workload distributions common on the web [15, 85, 86], hit rates improve only slowly with increasing trace length, and a nearly constant fraction of requests will be compulsory cache misses. Nearly linear growth in state therefore may be expected even over long time scales for many systems.

Although the near linear growth in state illustrates the need to bound worst case consumption, the rate of increase for this workload is modest. After 24 hours, fewer than 5 million leases exist in one of the four Sporting and Event server clusters even with infinite object leases. Our prototype consumes 62 bytes per object lease, so this workload consumes 310 million bytes per day under the baseline algorithm for the whole system. This corresponds to 0.4% of the memory capacity of the 143-processor system that actually served this workload in a production environment. In other words, this busy server could keep complete callback information for 10 days and increase its memory requirements by less than 4%. We also observe that the hit rate difference between *hot cache* and *cool cache* is less than 0.3%. In hot cache, we assume that valid objects hold valid object leases before the simulation; in cool cache, we assume that none of the objects holds a valid object lease before the simulation. The hit rate in cool cache is higher than that in a system throwing away the state of any client that has been idle for 10 days and the hit rate in hot cache equals that of a system keeping all the clients' state. Thus, throwing away

idle clients' state after 10 days effectively limits server state to 4% of the system memory at the cost of reducing hit rate by 0.3%.

These results suggest that either of the "forget idle clients" approaches can limit maximum memory state without significantly hurting hit rates or increasing lease renewal overhead, and that performance will be relatively insensitive to the detailed parameters of these algorithms.

5.4 Peak Server Load

In this section, we discuss how to reduce peak server load. The major factor contributing to peak server load is the large number of invalidation messages during a write. Thus, delaying when invalidation messages are sent is the key to reduce bursts of load. There are two variations. *Delayed invalidations* enqueue invalidation messages to clients whose volume leases have expired and send the enqueued messages in a group when a client renews its volume lease. *Background invalidations* place invalidation messages in a separate send queue from replies to client requests and send invalidations only when spare capacity is available. Note that background invalidations may increase the average staleness of data observed by clients, while delayed invalidations have no impact on average staleness of data reads. At the same time, while both techniques reduce bursts of load, background invalidations also have the ability to impose a hard upper bound on the maximum load.

Figure 5.5 shows the cumulative distribution of server load, approximated by the number of messages sent and received by a server with no hierarchy. As we can see from the right edge of this graph, the Volume Lease protocols with callbacks reduce average server load compared to TTL. However, as can be seen from the left

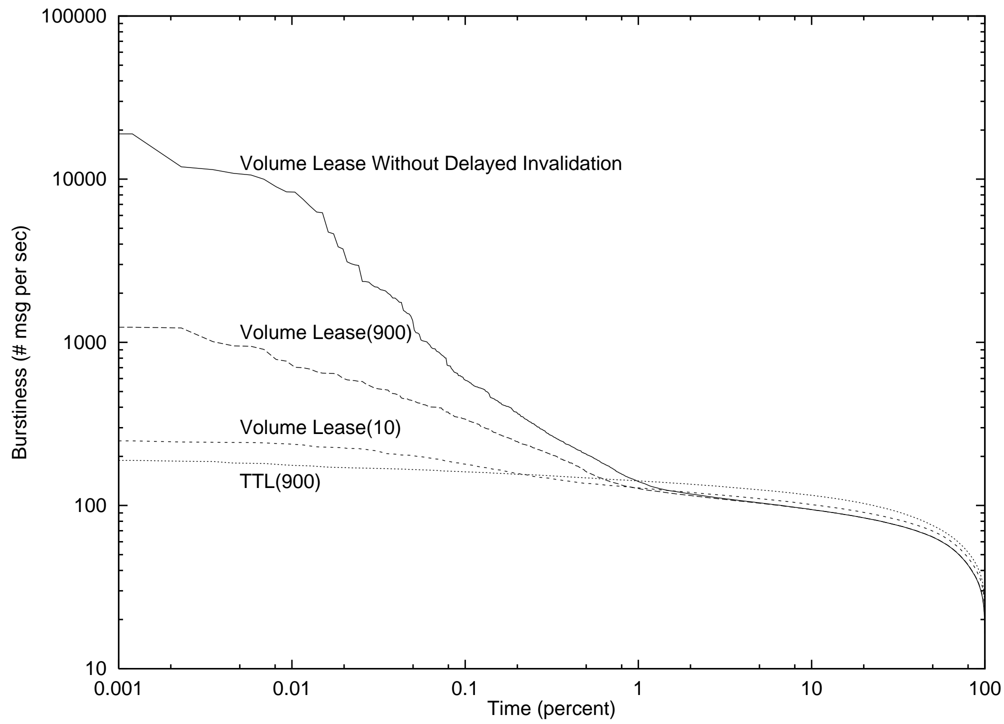


Figure 5.5: Distribution of invalidation burstiness for the IBM workload. A point of (x, y) means that the server generates at least y messages per second during x percentage of time.

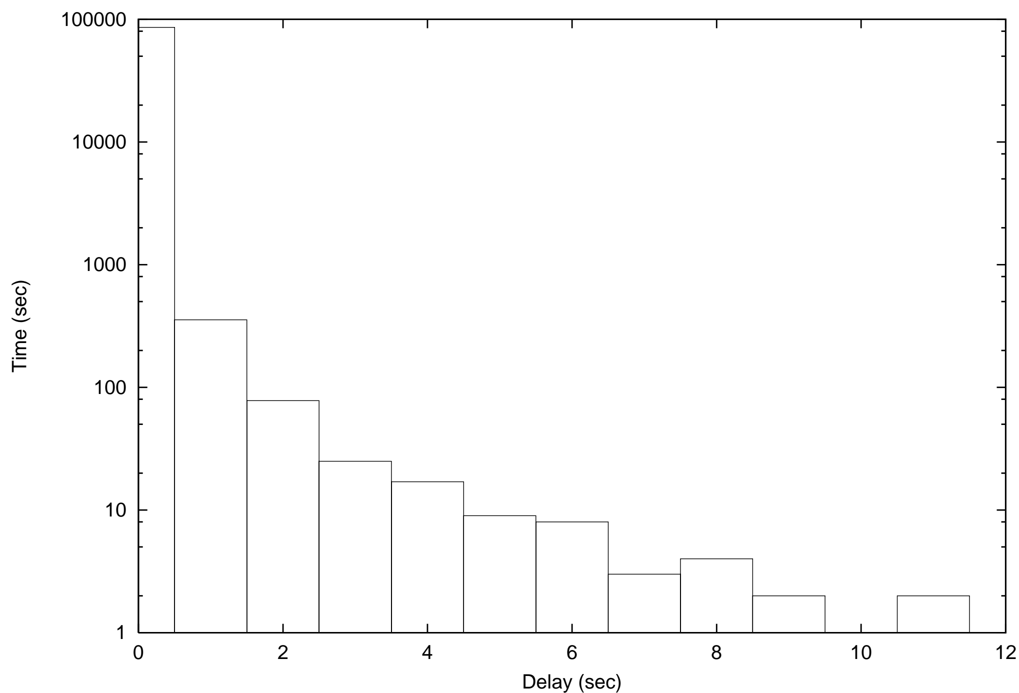


Figure 5.6: Distribution of delay of invalidation messages under background invalidations for Volume Lease(900).

side of the graph, the peak server load increases by a factor of 100 for the Volume Lease protocols without delayed invalidations.

This figure shows that delayed invalidations can reduce peak load by a factor of 76 for short volume lease periods and 15 for long volume lease periods; but even with delayed invalidations, peak load is increased by a factor of 6 for 15 minute Volume Lease protcols. This increase is smaller for short volume leases and larger for long volume leases since delayed invalidations' advantage stems from delaying messages to clients whose volume leases have expired.

Further improvements can be gained by also using background invalidations. In Figure 5.6, we limit the server message rate to 200 messages per second, which

is approximately the average load of TTL, and we send invalidation messages as soon as possible but only using the spare capacity. Well over 99.9% of invalidation messages are transmitted during the same second they are created, and no messages are delayed more than 11 seconds. Thus, background invalidation allows the server to place a hard bound on peak server load without significantly hurting average staleness.

Figure 4.5 showed the average staleness for the traditional TTL polling protocol. The data in Figure 5.6 allow us to understand the average staleness that can be delivered by invalidation with volume leases. Clients may observe stale data if they read objects between when the objects are updated at the server and when the updates appear at the client. There are two primary cases to consider. First, the network connection between the client and server fails. In that case, the client may not see the invalidation message, and data staleness will be determined by the worst-case staleness bound from leases. Fortunately, failures are relatively uncommon, and this case will have little effect on average staleness. Second, server queuing and message propagation time will leave a window when clients can observe stale data. The data in Figure 5.6 suggest that this window will likely be at most a few seconds plus whatever propagation delays are introduced by the network. We conclude that delaying invalidation messages makes unicast invalidation feasible with respect to server load.

5.5 Conclusion

In this chapter, we examine scalability challenges of Volume Lease introduced by large scale web services. We identify three challenges: fault tolerance, server state,

and peak server load. We quantify these challenges with simulation and evaluate various techniques allowed by the flexibility of Volume Lease to addresses these challenges. Our experimental results has shown: Volume Lease can perform well even with partial failures; we can put a hard limit on server state and peak server load without significantly impacting performance.

Chapter 6

Hierarchy and Multicast

6.1 Overview

In this chapter, we describe how to use multiple machines to maintain consistency with Volume Lease. Although a single machine can be sufficient for Volume Lease to maintain consistency in a small-scale web service, maintaining consistency for today's popular web servers may introduce too high a load for a single machine to handle. In particular, it may not be feasible to store all the clients' state and to send many invalidation messages in response to a write with just a single machine. Although some high-end machines can have much memory and CPU power, it is much more cost effective to use multiple machines instead of just one as observed by Dahlin in his thesis [30].

The key to using multiple machines for maintaining consistency is to form a consistency hierarchy. In addition to improving server scalability by distributing load and server state across a collection of nodes, hierarchical Volume Lease can introduce two more benefits. First, latency can be reduced if clients can renew

leases by going to a nearby node in the consistency hierarchy rather than to the server. Second, hierarchy can improve network efficiency by forming a multicast tree for sending invalidation messages to caches and a reduction tree for gathering their replies.

However, using hierarchy for Volume Lease introduces its own challenges. Availability may suffer because hierarchical structures consist of multiple nodes that can fail independently. Also, a deep hierarchy can increase latency if the hierarchy must be traversed to satisfy requests; a flat hierarchy can hurt both scalability and latency when a large number of requests that could be filtered out by additional levels of nodes are sent to the root. Thus, it is unclear how best to configure the hierarchy.

This chapter develops solutions for hierarchical consistency and addresses these issues. We define two primitive mechanisms, *split* and *join*, for growing and shrinking hierarchies, and we show how these primitives can be implemented with a simple mechanism already present in the Volume Leases algorithms. Using split and join we can address the fault tolerance and performance challenges of hierarchies.

When a node in a hierarchy failed, we can use join to remove this node from the hierarchy and connect its children to a predecessor of this node. When a failed node is recovered, we can use join to reconnect this node back into this hierarchy.

Join and split can also be used to flat out or deepen a hierarchy to improve performance. We also identify and quantify the ways in which specific characteristics of data-access workloads affect the scalability of hierarchical cache consistency. We explore three configurations. First, in *generic hierarchy*, consistency proxies can be placed anywhere in the Internet. The second configuration, *server-proxy-client*,

is designed to exploit widely deployed web proxies, which serve as gateways between enterprise LAN and web servers to improve security and network efficiency. In this configuration, web proxies are augmented to serve as consistency proxies that forward invalidation messages from web servers. This addresses the engineering challenge introduced by firewalls that generally prevent external machines from sending invalidation messages directly to clients within the firewall. The third configuration is the *server cluster* configuration in which hierarchy is introduced only within a LAN cluster of servers to improve scalability.

We evaluate our algorithms using simulation. To study scalability and to evaluate how the system is affected by different workload characteristics, we first use a series of synthetic workloads. To calibrate these results with realistic workloads, we also examine some smaller trace-based workloads. Overall, we find that even without hierarchies, Volume Leases can scale to services with tens of thousands of clients; with hierarchies, scalability beyond millions of clients appears feasible.

6.2 Algorithms

We first describe a naive algorithm based on a static consistency hierarchy and discuss its performance and fault tolerance properties. Next, we present two primitive mechanisms, split and join, for reconfiguring the hierarchy. These mechanisms can be constructed with trivial additions to the basic Volume Lease algorithm. We then describe policies that use these mechanisms to enhance the fault tolerance and performance of the basic static hierarchy.

Both the static and dynamic versions of the algorithm assume that nodes participating in the consistency service have been identified and organized into an

initial hierarchy. This study does not specify a particular mechanism for doing so. For some systems, constructing the hierarchy manually suffices; for some, such as the server-proxy-client configuration in Section 6.3.3, automatic construction is trivial; for others, more sophisticated automatic strategies such as those described by Plaxton et. al [72] might be required. This hierarchy may be embedded on current clients and proxies, it might be coincident with a larger cache hierarchy [22] or it might be part of a separate data-location-metadata hierarchy [42, 78].

Our consistency hierarchy is a tree structure of interconnected nodes. We refer to the root as the *origin server*, to the leaves as *clients*, and to the intermediate nodes as *consistency servers.* Each node runs the standard Volume Lease algorithm; each intermediate node acts both as a client and as a server, treating its parent as its server and its children as its clients. Each node thus satisfies lease requests from its children by returning a valid lease if it has one cached, or—if it does not—by requesting a lease from its parent, caching the lease, and returning the lease to its child. Similarly, each node passes to any children that have valid leases the invalidation messages that it receives.

Hierarchical Volume Lease can be used to provide strong consistency. Before the server writes an object, it first invalidates all object leases held by nodes in the hierarchy that also hold a valid volume lease. It does this by sending invalidation messages to all children that hold both valid volume and object leases. Its children are then responsible for invalidating the leases in their subtree by recursively sending invalidation messages to their children. When a leaf node receives an invalidation message, it invalidates the object lease and sends an invalidation acknowledgment to its parent. For every invalidation message, an internal node either receives an

acknowledgment or waits until the volume lease expires before invalidating its own lease and sending an acknowledgment to its parent. The server can modify the object when all invalidations have been acknowledged or when the volume leases held by nodes that have not yet acknowledged have expired.

Recall that our flat volume lease algorithms maintains strong consistency because that all the clients with both valid volume lease and object leases for an object, (this is the set of the clients who can read the objects), receive invalidation messages before the server updates the objects. The same principle holds for the hierarchy consistency. Since an intermediate nodes always caches a local lease copy before it forwards the lease to lower level nodes, it is easy to see that the leases maintained by the hierarchy have the inclusion property, which is defined as that whenever a node hold a valid object lease or volume lease, its parent has to hold the lease also. Furthermore, all the internal proxy nodes in the consistency hierarchy always pass down the invalidation messages to its children which hold both valid volume lease and object leases. Hence, by induction, we can see that all clients which hold both volume lease and objects receive invalidation messages before updates.

We only need to modify the algorithm that we have just discussed slightly to allow applications to exploit their weak consistency semantics for better performance. Hierarchical Volume Lease can be adapted easily to provide a staleness bound which equals the volume lease length. Just as flat Volume Lease protocols, we only need to drop the requirement that requires all the invalidations to be acknowledged before allowing a write to complete. Thus, a server can complete a write immediately upon a write is submitted to enable some clients to read the new data as soon as possible. The nodes can still choose to send out invalidation messages to

all their children who hold both the valid volume leases and object leases to reduce stale read rates. All the nodes including intermediate nodes can throw away their own object leases immediately upon receiving an invalidation message. No node needs to send acknowledgments for invalidations.

Such hierarchies have the potential to improve performance by reducing both server load and the latency of client lease renewals. In the Internet, a popular site might serve millions of clients, and by using a hierarchy, a server tracks and communicates with only its immediate children and hence reduces memory state, average load for lease renewals, and bursts of load when popular objects are modified. In essence, the consistency hierarchy forms a precise multicast tree for sending invalidation message and forms a reduction tree for gathering replies. By the same token, if clients can renew leases by going to nearby intermediate consistency servers rather than to the root server, read latency and network load may be reduced.

However, the use of leases in the hierarchy is not guaranteed to reduce either server load or latency. When volumes are popular and frequently accessed, it is likely that consistency servers will hold valid leases and will respond to client requests without consulting their parents, and it is likely that the hierarchical "multicast" will achieve a large fan-out and significantly reduce server load. However, for unpopular or infrequently accessed volumes, the time between accesses to consistency nodes is likely to be longer than the volume lease, so the cached leases may often have expired when they are accessed. In these cases, many messages would traverse the entire hierarchy, increasing the average read latency without reducing server load.

A second problem with a static hierarchy is reliability. The hierarchy consists of a large number of nodes that can fail independently, and one node failure can

effectively disconnect a subtree.

6.2.1 Join and split

The solution to both problems is to reconfigure the consistency hierarchy dynamically without breaking consistency guarantees. We propose a mechanism that uses two primitives: *join*, which removes an intermediate node from the hierarchy, and *split*, which adds an intermediate node to the hierarchy. Both primitives work on a per-volume basis—in our system different volumes can use different hierarchies.

Join and split can be trivially implemented using mechanisms already required by the Volume Lease algorithm. Recall that join removes a node from the hierarchy, connecting the children of the node directly to the node's parent. To implement join we augment the volume epoch number to include the parent node's identity. When a child initiates a join for a particular volume, it simply begins using its former grandparent as a parent. The volume epoch number held by the child will not match its new parent, so the new parent initiates the standard volume reconnection protocol to synchronize its state with its new child. Thus, going to a new parent in the hierarchical algorithm is no different than going to a server that has crashed and lost a client's state in the original Volume Lease algorithm. Similarly, to split the hierarchy, a child chooses a descendant of its parent and starts using the new node as its parent, again using the reconnection protocol to synchronize the state. For both split and join, the decision to use a new parent can be made by children at arbitrary times. The criteria for such decisions are a matter of policy. Children can thus decide to find new parents to improve fault tolerance or they can be advise to use new parents to improve performance. Once the reconnection is

performed, a child node can choose to resynchronize with its parent aggressive or lazily as discussed in the previous chapter.

6.2.2 Fault tolerant static hierarchy

Using join and split, an intermediate node failure is handled as follows. If a node N cannot contact its parent P to renew a lease, it sends the renewal message to one of its ancestors A, triggering the volume reconnection protocol between N and A. Note that if A cannot send an invalidation to P, it does not try to contact N, but instead waits for the volume lease timeout, which means that parents need to know only about their immediate children, not their more distant descendents. Finally, when node P recovers, it can send hints to its list of (former) children suggesting that they split from A and join P instead.

6.2.3 Dynamic hierarchy configuration

For volumes with high read frequencies and many active clients, a deep hierarchy can reduce read latency and distribute load. However, for less popular objects, or for popular objects with low read frequency, intermediate hops can increase read latency without significantly reducing server load. Therefore, it is useful for different volumes to construct different dynamic hierarchies. These hierarchies can be constructed from the static hierarchy using the split and join mechanisms in response to changing workloads. Hence, a node can have different children in the static and dynamic hierarchies: we refer to the former as *static children*, and to the latter simply as *children*.

In the dynamic configuration algorithm a node N monitors the number of

lease requests it receives from its children and the fraction of these requests that it can satisfy locally during time intervals of length T. Using this data, N instructs its children to join with its parent if (1) the load from its children would not cause the load on its parent to exceed a threshold value and (2) its children would receive better read latency by skipping N and going directly to the parent. N performs the latency calculation as follows.

Let $RenewCost(N)$ be the cost for a child of N to renew a lease cached at N, and let $RenewCost(P)$ be the cost for N to renew a lease cached at its parent. If the fraction of renewals that N satisfies locally is F, then the expected latency that a child of N pays to renew a lease is $RenewCost(N) + (1 - F)RenewCost(P)$. Assuming that the cost of accessing N's parent is about the same for both N and N's child, the expected cost after a join is $RenewCost(P)$. When $RenewCost(N) + (1 - F)RenewCost(P)$ is greater than $RenewCost(P)$ by some threshold, N instructs its children to perform a join unless doing so would raise the load of the parent to an unacceptable level.

Similarly, to determine when to initiate a split, a node monitors the requests from its children and initiates a split if (1) its local load exceeds some threshold or (2) connecting a set of children to a skipped node would reduce their expected read latency by some threshold.

A node N performs this read latency calculation by simulating the performance of its skipped children as follows. For each static child S, N maintains a simulated request count $ReqCount(S)$, hit count $HitCount(S)$, and volume lease expiration time $VolExp(S)$. When a child C of N contacts N to renew a lease, N updates the statistics for the skipped child S that is a static ancestor of C by (1)

incrementing $ReqCount(S)$, (2) incrementing $HitCount(S)$ if the current time is before $VolExp(S)$, and (3) setting $VolExp(S)$ to the current time plus the volume lease length. Let $RenewCost(N)$ be the cost for C to renew its lease at N and $RenewCost(S)$ be the cost for C to renew its lease from the skipped child S instead. N tells C and its siblings to split from N and instead use S as their parent if $RenewCost(S) + (1 - \frac{HitCount(S)}{ReqCount(S)}) \cdot RenewCost(N) < RenewCost(N) - threshold$.

6.3 Evaluation

We evaluate hierarchical consistency in three different deployment configurations. First, we examine an aggressive deployment model, *generic hierarchy*, to characterize the factors that affect the behavior of the core algorithms and to determine the performance limits of our approach. Second, we examine a simple *clustered-server* configuration in which the hierarchy is used to distribute the algorithm across a LAN cluster in order to improve scalability but not latency. This configuration might be used if a service wishes to provide strong consistency for its data without relying on having consistency-enabled intermediate proxies deployed across the WAN. Third, we examine a *server-proxy-client* configuration that maps well to infrastructure that is common today.

We evaluate these algorithms using simulations. To study scalability and to evaluate how different aspects of workloads impact scalability, we first use a series of synthetic workloads. We run each of these experiments five times using different random seeds for workload generation and show the 90% confidence interval for each point. Then, to calibrate these results, we examine a smaller, trace-based workload in the context of the server-proxy-client configuration.

Based on these experiments, we reach the following conclusions:

- For the aggressive deployment scenario with flexible hierarchy configurations, static hierarchies can reduce latency compared to the flat Volume Lease algorithm for high request-rate services, but they can increase latency for low request-rate services. In contrast, the dynamic version always performs as well as the flat algorithm for low request rates and as well as the static hierarchy for high request rates.

- For workloads with modest request rates in the range of many current web services, the flat Volume Lease algorithm with a single server can scale to client populations in the tens or hundreds of thousands of nodes; distributing the consistency algorithm across a group of nodes—either in a cluster or across a WAN—via hierarchies can provide scalability to millions of clients even under aggressive workloads.

- In the server-proxy-client configuration, the simple static hierarchy performs well for our web trace workload; this configuration has the added benefit that it might also provide a controlled way to traverse firewalls in order to deliver consistency signals. The synthetic workload suggests that there may be other workloads for which the dynamic algorithm's flexibility is desirable.

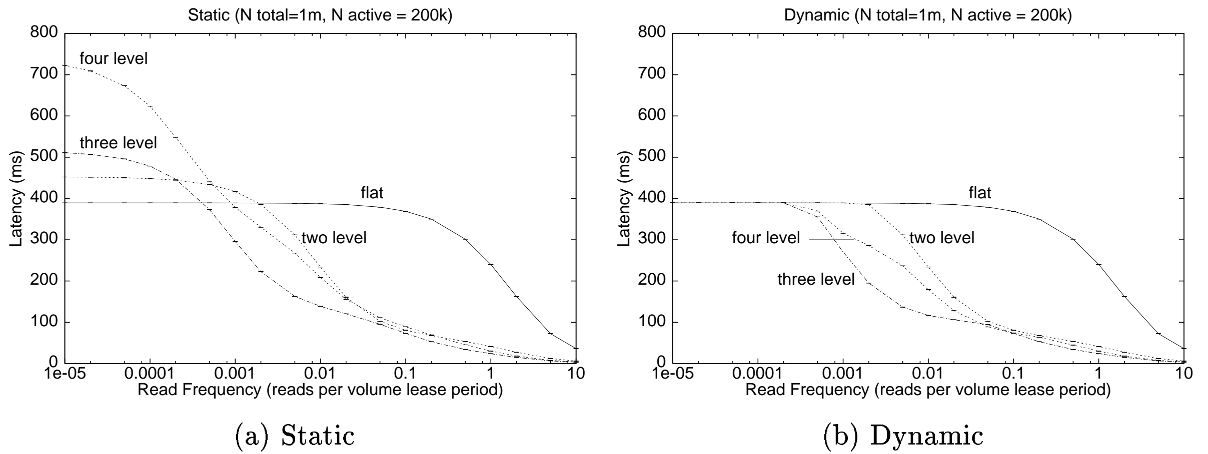
Our methodology makes several significant assumptions and simplifications. For our latency estimates, we do not simulate network or server contention. We use a simple network topology and delay model to make our analysis tractable. We also do not consider the overhead of resynchronization in join and split because we assume that they happen very infrequently. In a system with a lots of failures,

our results may underestimate the overhead of hierarchies. Moreover, our synthetic workload assume stable workloads, which may overestimate the benefits of dynamic hierarchies. Finally, our default synthetic workloads simulate one object per volume. This may understate the apparent benefit of hierarchies because long-lived object leases are much easier to cache in the hierarchy than short volume leases; furthermore, the small number of objects per volume may also hurt the relative performance of the static algorithm.

6.3.1 Generic hierarchy

Our Generic Hierarchy configuration represents a system with few constraints on deployment. We examine this configuration to understand the behavior of the core algorithms as we vary several key parameters. This configuration also models an aggressive deployment strategy such as might be employed within a large cache service or in a system where collections of servers and cache systems coordinate to provide consistency.

The consistency hierarchy is a tree with one server at its root, C clients at its leaves, and $l - 1$ levels of intermediate nodes. We designate the server to be the level 0 node of the consistency hierarchy. For simplicity, we assume that at all levels of the tree the degree d is the same, with $d^l = C$. We defer the evaluation of hierarchies with different fan-out at different levels for future work. We use a simple cost model for accessing consistency servers. First, we assume that all leaf nodes and internal nodes within a subtree experience the same latency when they renew a lease with the root of that subtree. Second, we assume that the latency experienced within a subtree increases with the number of leaves in the subtree as

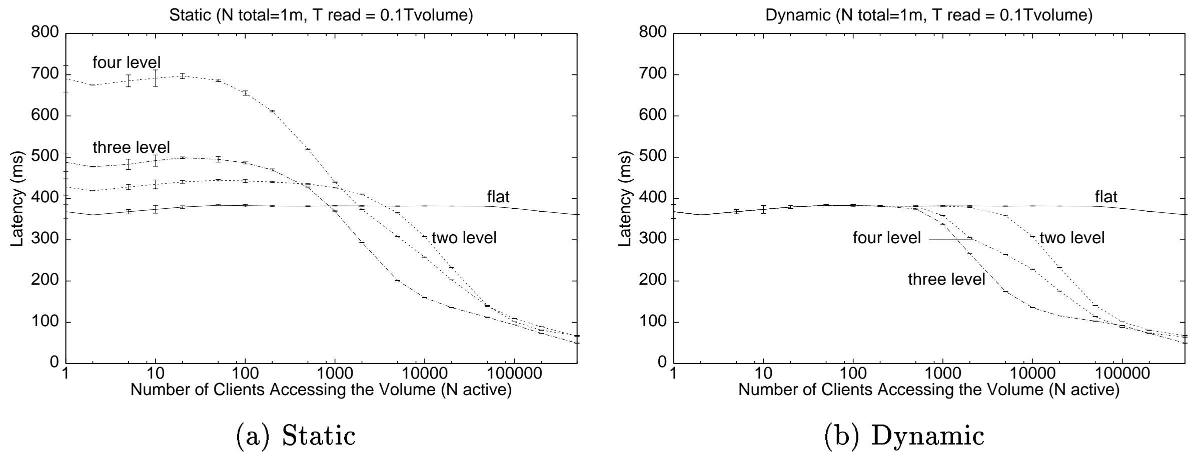


(a) Static (b) Dynamic

Figure 6.1: Average read latency as the per-client read frequency is varied for a hierarchy of one million clients, of which 200,000 access the volume in question.

follows: subtrees with 100 or fewer leaves have a latency of 30 ms, subtrees with 10,000 leaves have a latency of 100 ms, and subtrees with more than 100,000 leaves have a latency of 400 ms; latencies for subtrees with 100-10,000 nodes and 10,000 to 100,000 nodes are estimated through interpolation. These latencies are meant to be suggestive of department-, enterprise-, and Internet-scale delays, but do not represent any specific system.

We use a synthetic workload and compute the average read latency and server load when we simulate the accesses of a collection of clients to a single volume. Out of N_{total} clients, we choose a subset of clients of size N_{active} that access the volume with per-client inter-access times determined using an exponential distribution around an average value t_{read}, which is expressed as a ratio of the average inter-access time to the volume lease renewal time. In our initial experiment, each volume contains a single object; we relax this assumption later in this section.

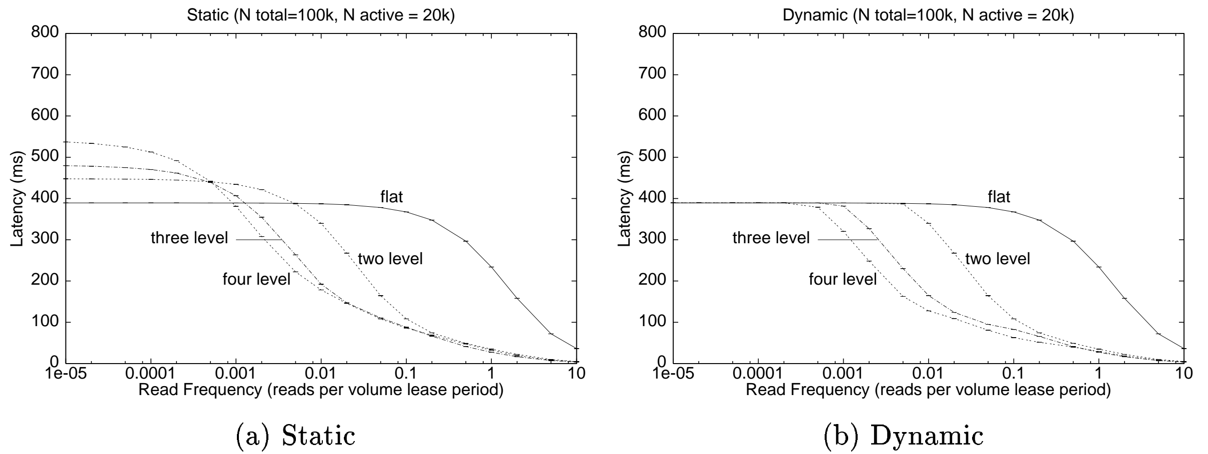


(a) Static (b) Dynamic

Figure 6.2: Average read latency as the number of active clients varies for a hierarchy of one million clients, each issuing requests to a volume at a rate of 0.1 requests per volume lease period.

Read Frequency A server's read frequency has a large impact on the performance of hierarchical consistency. The higher the collective read frequency of the clients below a proxy, the more often the proxy holds the lease. Hence, if the read frequency is high the lease hit ratio will be high and the proxy can reduce read latency. Otherwise, if the collective read frequency is low, then the lease hit ratio will be low.

Figure 6.1 shows the average lease renewal latency as the per-client read frequency is varied. Figure 6.2 shows lease renewal latency as the fraction of clients that access the volume in question is varied. The graphs compare the performance of a flat, 2-level, 3-level, and 4-level consistency hierarchy with part (a) of each figure showing performance for the static algorithm and part (b) showing performance for the dynamic algorithm. Figures 6.1 and 6.2 have the same general shape because as one moves to the right along the x axis the total request rate increases in both sets of graphs. But, these graphs represent different dimensions of the design space.



(a) Static

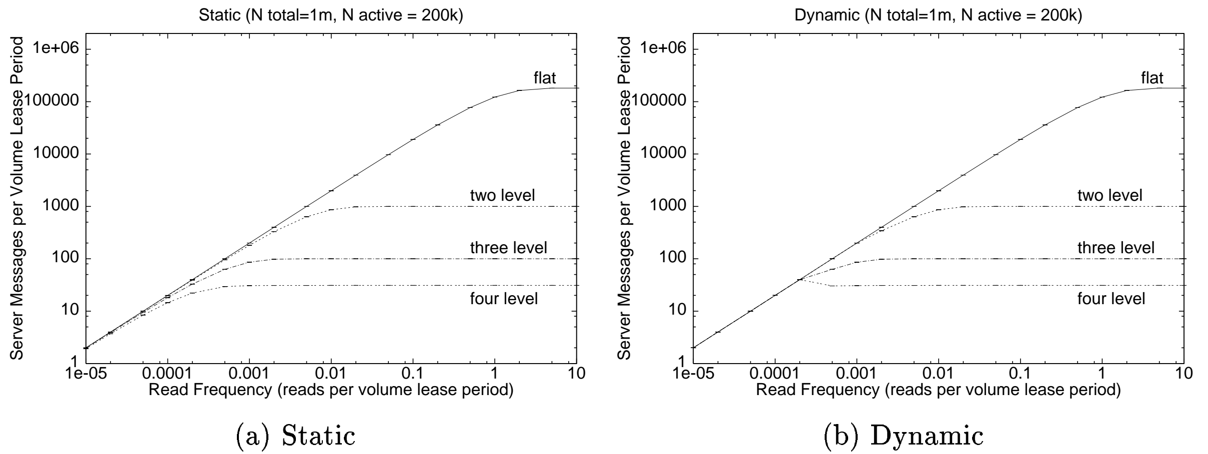
(b) Dynamic

Figure 6.3: Average read latency as the per-client read frequency is varied for a hierarchy of 100,000, of which 20,000 access the volume in question.

Read latency generally decreases as read frequency increases. For high request rates, the read latency falls even for the flat configuration because a client issues multiple reads within the period the client's volume lease is valid.

To interpret these graphs it is helpful to consider where different classes of services might lie or where a single service might lie under different workloads. For example, a weather service that is visited by an average client once a day for one minute and that uses a 10-second lease period would correspond to a read frequency of less than 0.001 reads per volume lease period per client. Similarly, a news service whose typical users visit for 5 minutes during the 8-hour working day would correspond to a volume renewal frequency near 0.01 per volume lease period per client. The read frequency of that same service might jump above 0.1 or even near 1 for periods of time during news events of widespread interest as clients constantly monitor the news for new developments. Similarly, emerging program-driven applications might span a wide range of the parameter space.

With respect to lease renewal latency as a function of read frequency, the



(a) Static (b) Dynamic

Figure 6.4: Average server load for handling renewal requests as the per-client read frequency is varied for a hierarchy of one million clients, of which 200,000 access the volume in question.

main observations are as follows:

- Hierarchies can significantly reduce latency for active and popular services.
- The dynamic hierarchy succeeds in matching the latency of the flat Volume Lease algorithm for less active or less popular services while matching the performance of the static hierarchy for busier services. Relative to flat Volume Lease, the static hierarchy can hurt latency for less active or less popular services but can help latency for active and popular services.
- The dynamic hierarchy appears to be a good default choice for this configuration. If a service's access patterns are known and if these access patterns do not change much, then either flat Volume Lease or a static hierarchy may be reasonable, depending on the workload.

Finally, note that the variations among different depths of underlying static trees depend both on interactions between the number of clients under each level

of a subtree and on our assumptions on the network distances between subtrees as a function of subtree size. Hence, this experiment should not be used for general comparisons between the number of levels that should be used in the underlying hierarchy.

Figure 6.3 shows similar experiments but with 100,000 total clients (20,000 of them active) rather than 1,000,000. Comparing these results to those with more clients provides intuition about the effects of scaling the client population, which may help predict system behavior for populations larger than the 1,000,000 that we are able to simulate.

- As expected, increasing (decreasing) the total number of clients decreases (increases) the per-client request rate for which hierarchies begin to pay off relative to the flat Volume Lease configuration. We observe similar results when we vary the number of active clients (graph omitted).

Figure 6.4 shows how server load varies with client request rate hierarchies spanning one million clients. (Results for varying the number of active clients or simulating a universe of 100,000 clients are omitted, but are qualitatively similar). Assuming that a server can handle a few thousand requests per volume lease period, we conclude:

- The flat Volume Lease algorithm scales to tens of thousands of clients under workloads corresponding to a range of reasonable web access patterns.
- The addition of hierarchies supports scalability to many millions of clients under nearly arbitrary workloads because it bounds the rate of requests at the root to one request per volume lease period per immediate child of the root.

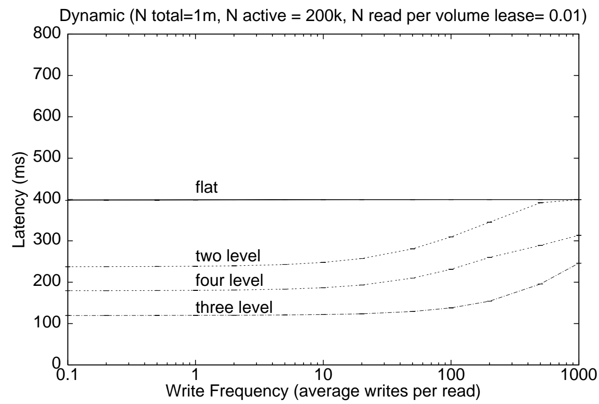


Figure 6.5: Average read latency under the dynamic algorithm as the write frequency for 10 objects in a server volume is varied. Average read frequency per client is fixed at 0.01 read per client per volume lease period.

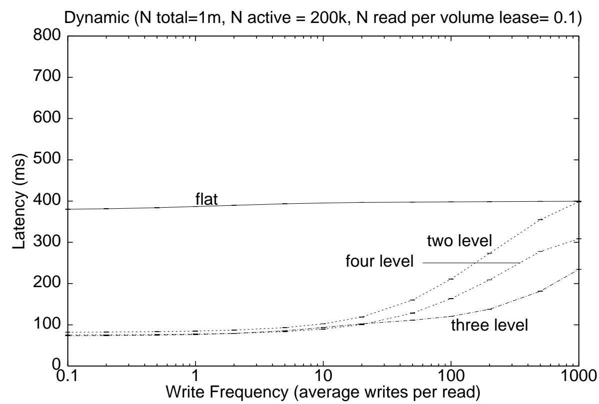


Figure 6.6: Same as Figure 6.5, except that the read frequency is 0.1 read per client per volume lease period.

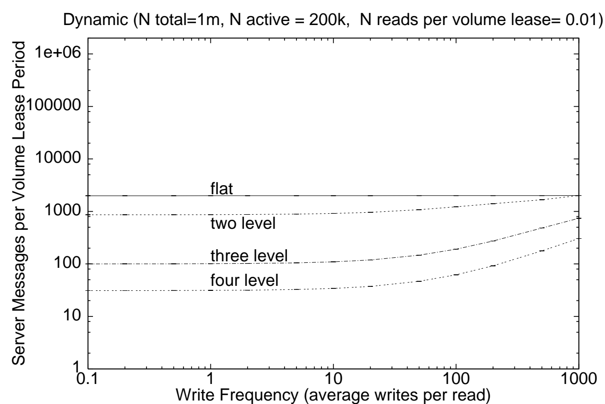


Figure 6.7: Average server messages per volume lease period under the dynamic algorithm as the write frequency for 10 objects in a server volume is varied. Average read frequency per client is fixed at 0.01 read per client per volume lease period.

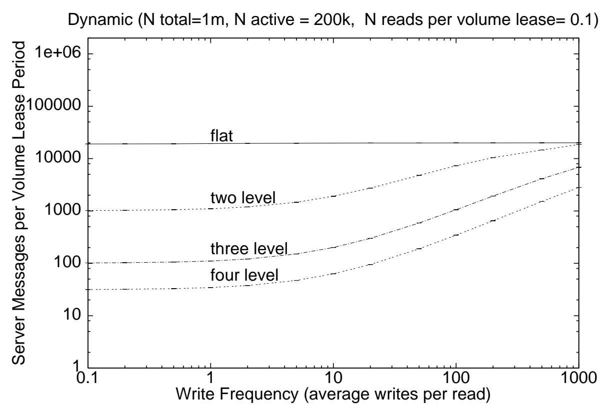


Figure 6.8: Same as Figure 6.7, except that the read frequency per client is 0.1 read per client per volume lease period.

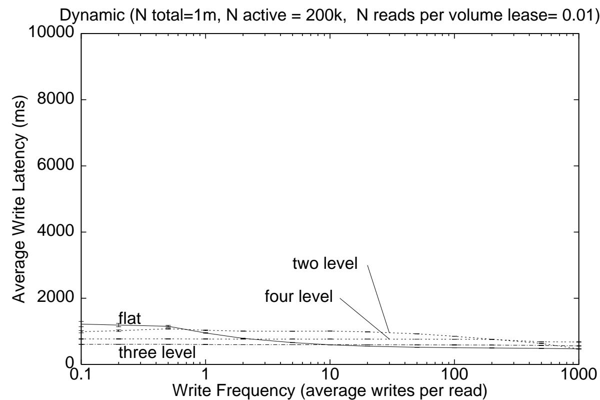


Figure 6.9: Average write latency seen by server for the dynamic algorithm as the write frequency for 10 objects in a server volume is varied. Average read frequency per client is fixed at 0.01 read per client per volume lease period.

Writes to multiple-object volumes To make simulating a large scale hierarchy feasible, we have so far considered only the cost of renewing volume leases. We have not examined the cost of renewing or invalidating object leases. This simplification affects our results in two ways. First, it causes us to understate average read time because in reality clients will sometimes have valid volume leases but still need to fetch object leases. This effect should be modest because object leases are much longer than volume leases. Second, this simplification may cause us to understate the benefit of consistency proxies, particularly when read frequency is low, because consistency proxies can cache long object leases more effectively than short volume leases. To calibrate the effect of object leases for popular servers, we run several simulations with multiple object leases per volume. The results are illustrated in Figures 6.5 to 6.10. Due to space constraints, we show the graphs for the dynamic hierarchy algorithm and omit those for the static algorithm; the static results differ little from the dynamic ones.

In these experiments, the server volume contains 10 objects. Each object

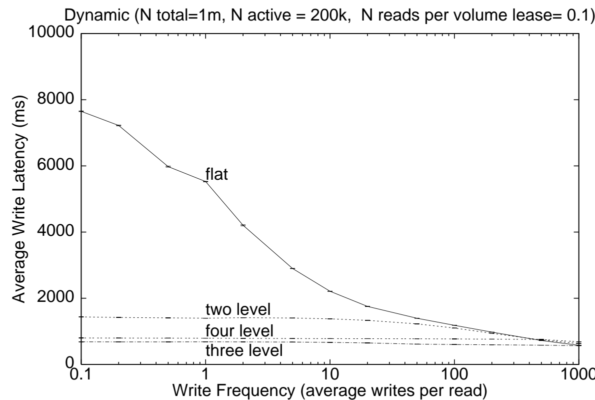


Figure 6.10: Same as Figure 6.9, except that the read frequency per client is 0.1 read per client per volume lease period.

is modified independently with average write frequency varying from 0.1 writes to 1000 writes per client read. We illustrate performance for servers that have average 0.01 read and 0.1 read per volume lease per client. For example, a system with 30 second volume leases, 3000 seconds between client reads, and 300 seconds between writes would correspond to the *Write Frequency = 10* point in Figure 6.5.

Figure 6.5 shows the average read latency as the write frequency changes. Each client issues an average of 0.01 reads per volume lease period. When write frequency is between 0.1 to 10 writes per client read, the results closely match our simplified experiment. Only after write frequency gets higher than 10 writes per client read does the read latency increase become noticeable. Figure 6.6 is similar to Figure 6.5, except the read frequency is set to 0.1 reads per volume lease period instead of 0.01. Figures 6.7 and 6.8 show the average server load under the same workload as Figures 6.5 and 6.6.

Figures 6.9 and 6.10 show the average write latency as the write frequency changes. We calculate the write latency by finding the critical path from when the server sends its first invalidation until it receives the final reply. At each node, N,

of the hierarchy, we charge $writeCost(N) = nValidChildren(N) \cdot costPerChild + latencyToChild(N) + \max_{\forall c | c \in children(N)}(writeCost(c))$ where $nValidChildren(N)$ is the number of N's direct children that have valid volume leases. Note that by using the *delayed invalidation* optimization [89], the sending of invalidation messages to nodes whose volume leases have expired can be removed from the critical path. The latency to a child is determined by the topology according to our standard formula, and the cost per child is set to 1 ms.

The data indicate three main effects:

- When write frequency increases, the benefit of hierarchy for reads is eroded.
- As write frequency decreases, the write latency increases. This is because the number of valid leases that accumulate between writes, and thus must be invalidated, increases.
- For a frequently accessed popular server, flat volume leases can introduce significant write delays, but hierarchies can remedy this problem.

6.3.2 Server cluster

The hierarchical consistency mechanisms can be used not only to distribute consistency algorithms across a WAN, but also to split a consistency service across a clustered web server on a local area network (LAN) or system area network (SAN). Although an algorithm optimized for splitting consistency state and load across a cluster with a fast network might marginally outperform our more general mechanisms, such an algorithm must solve the same basic problems of fault tolerance, distributing invalidations, gathering acknowledgments, and partitioning state that

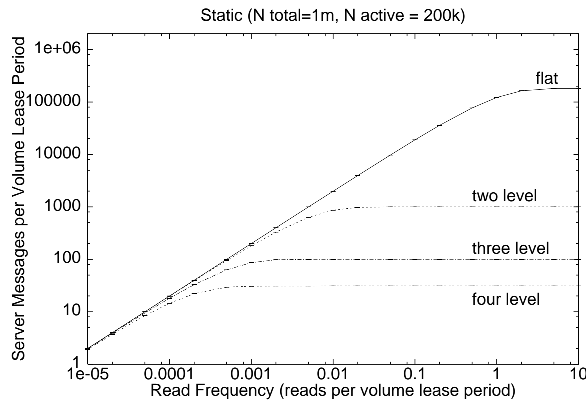


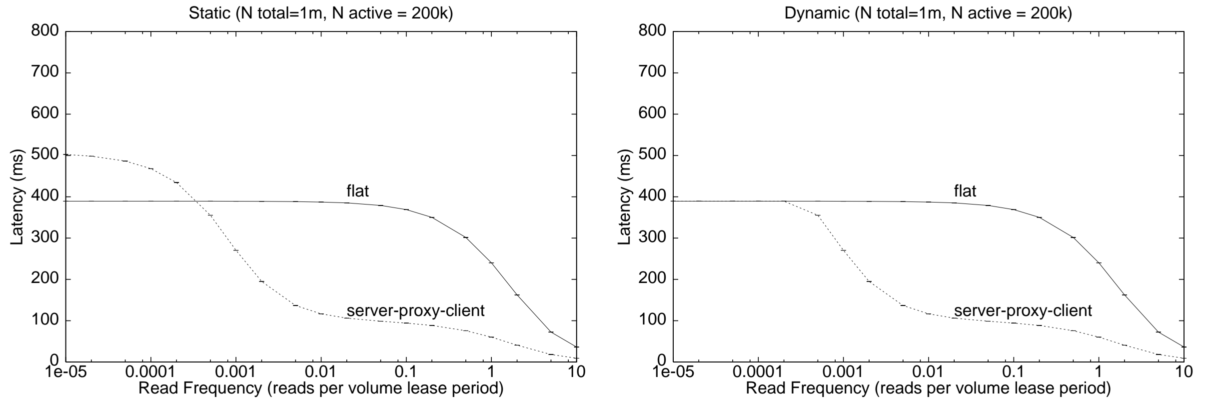
Figure 6.11: Server lease renewal load as the per-client read frequency is varied for a static server cluster hierarchy serving one million clients, of which 200,000 access the volume in question.

our algorithm handles, so the simplicity of using a single framework for both LAN and WAN distribution appears attractive.

Figure 6.11 shows the load on the server in the server cluster hierarchy where the server and all of the internal nodes of the consistency hierarchy are located in a tightly-coupled cluster, and the lowest internal nodes in the hierarchy communicate across a WAN with the clients. This configuration is not designed to improve latency, just load-scalability. As a result, read latency cannot be affected significantly by modifying the consistency structure. Hence, it is not necessary to build a dynamic hierarchy in this circumstance.

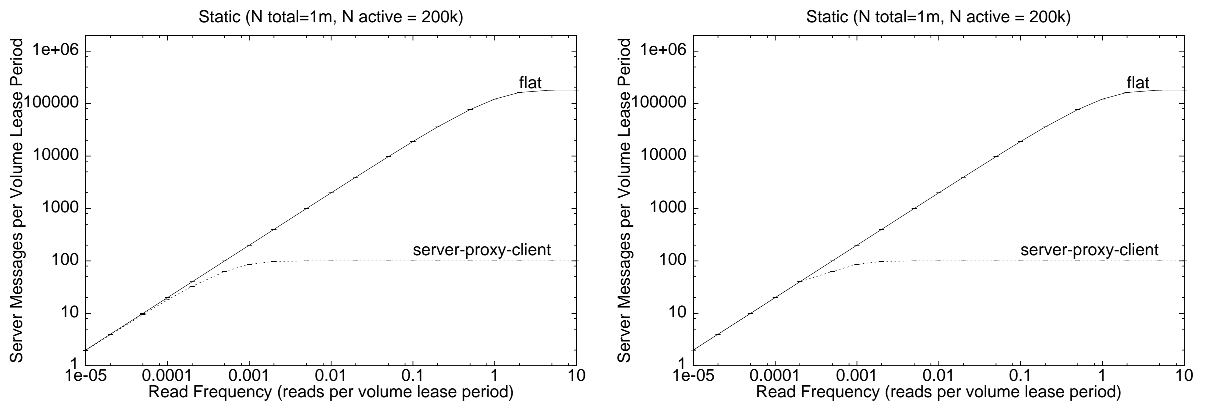
Based on this experiment, we conclude:

- For the server-cluster configuration, the static hierarchy (with split and join for fault tolerance) provides a simple mechanism to scale the flat Volume Leases algorithm by distributing it across a group of nodes in a cluster; dynamic configuration to minimize latency is not required.



(a) Static (b) Dynamic

Figure 6.12: Average read latency as the per-client read frequency is varied for a server-proxy-client hierarchy of one million clients, of which 200,000 access the volume in question. In the server-proxy-client hierarchy the internal nodes in the consistency hierarchy are all proxies serving 10,000 clients each.



(a) Static (b) Dynamic

Figure 6.13: Server lease renewal load as the per-client read frequency is varied for a server-proxy-client hierarchy serving one million clients, of which 200,000 access the volume in question.

	Med 1			Med 2			Med 3			Med 4		
	flat	static	dyn	flat	static	dyn	flat	static	dyn	flat	static	dyn
Latency	160.5	129.4	135.4	99.0	89.5	92.1	55.6	61.2	57.3	276.3	297.0	279.7
Load	0.41	0.23	0.27	0.25	0.16	0.20	0.14	0.12	0.14	0.69	0.57	0.64

(a) Trace results for four medium-loaded volumes.

	Large 1			Large 2			Large 3			Large 4		
	flat	static	dyn	flat	static	dyn	flat	static	dyn	flat	static	dyn
Latency	84.1	30.8	30.7	123.2	51.1	51.2	133.0	46.7	46.7	68.9	39.3	39.6
Load	0.21	0.03	0.03	0.31	0.05	0.05	0.33	0.03	0.03	0.18	0.06	0.07

(b) Trace results for four heavily-loaded volumes.

Figure 6.14: Average read latency and fraction of renewal requests sent to the server for the four medium-loaded and four heavily-loaded volumes from the DEC trace workload under a server/proxy/client hierarchy in which the internal node in the consistency hierarchy is the proxy serving the DEC clients. Latency is in term of milliseconds and load is in term of the number of server messages normalized by the number of reads.

6.3.3 Server-proxy-client

Figures 6.12 and 6.13 show the latency and load measurements when the hierarchy algorithms are run on the server-proxy-client underlying hierarchy with one million clients grouped into 100 proxy-groups of 10,000 clients per group. This experiment suggests two points:

- For low read frequencies, the dynamic hierarchy where clients may contact servers directly has a modest advantage over the static hierarchy.
- At high read frequencies both the static and dynamic configurations significantly outperform the flat configuration.

Figure 6.14 shows latency for several selected volumes under a trace workload. The workload is the DEC trace [32], and we configure the system with all clients under a single proxy. We map each server in the trace to a different volume. We

present results for 8 selected servers: the 4 most popular ones and 4 of medium popularity.

- The trace workloads include multiple objects per volume, and long object leases are easier to cache in a hierarchy. As a result, the static hierarchy begins to pay dividends even with relatively low access rates.

This suggests that for many current web workloads, the simple static hierarchy using the simple server-proxy-client hierarchy may be a reasonable deployment option. This configuration might also provide a practical way to traverse firewalls to deliver consistency signals.

6.4 Related Work

The work described in this chapter fits into a bigger field, multicast Volume Lease, in which leases and invalidations are delivered with multicast to increase system scalability. The studies in this field can be roughly classified into two groups: precise multicast Volume Lease and imprecise multicast Volume Lease.

Our protocol is a precise multicast Volume Lease protocol implemented with application-specific multicast. A consistency hierarchy can be thought of as an application-specific multicast tree. Invalidation messages and lease renewals are multicasted through the multicast tree. The root and intermediate nodes keep state on clients' caches to filter out irrelevant invalidation messages, the invalidation messages on the objects which are not cached by individual clients. An earlier study done by Worrell [87] concludes that precise multicast in a hierarchy cache environment does not increase overhead compared to client polling. In this chapter,

we further demonstrate that precise multicast can be robust enough to scale up to a millions of clients and be adaptive to minimize latency. Subsequent studies [65, 76, 66, 67] have reached simliar conclusions.

In imprecise multicast Volume Lease [92, 58], the root and intermediate nodes do not keep state and thus can not filter invalidation messages along the multicast tree. Yu et. al [92] uses IP multicast while Li and Cheriton [58] use application-level multicast.

The drawback of precise multicast is that we need to keep state about clients' caches. Keeping state not only consumes memory, but also complicates fault tolerance and system reconfiguration because we may need to reconstruct state. However, by filtering out irrelevant invalidation messages, precise multicast can reduce network bandwidth consumption and client overhead compared to imprecise multicast. The advantage is especially significant in an environment where clients do not share many reads.

The interactions between maintaining consistency and delivering content are examined by some researchers [58, 39]. Li and Cheriton show that multicasting updates, which allow a client to invalidate a old version of an object and load a new version at the same time, reduces network and server overhead when updates are frequent. Fei [39] proposes a system that alternates between multicasting updates and multicasting only invalidations according to update and access frequency.

TTLs in a hierarchical cache environment is also been studied. Cohen and Kaplan [24, 25] propose a mathematical model for the performance of TTLs in a hierarchical cache environment and suggest prefetching TTLs can be beneficial.

6.5 Conclusions

In this chapter we have shown that the Volume Lease algorithm can provide strong consistency and a range of weak consistency for Internet services with hundreds of thousands of clients. We have also shown how the Volume Lease can be applied to hierarchical caches to perform well for workloads with millions of clients. The key mechanisms, join and split, can be implemented using a simple extension of the Volume Lease algorithm. Finally, we have evaluated a number of hierarchy configurations, and our results show that a dynamically configurable hierarchy provides tremendous amounts of scalability.

Chapter 7

Conclusion and Future Work

This dissertation studies how to design scalable cache consistency protocols for web replication systems. Scalable cache consistency protocols are critical for the effectiveness of replication systems in the web. The challenge here is the large scale of Internet-wide services and the fault tolerance requirements resulting from the large scale. We propose a scalable cache consistency framework, Volume Lease. Volume Lease decouples synchronization from invalidation. Synchronization is performed on sets of objects - called volumes - with volume leases, which amortize the cost of synchronization over many objects in a volume. This framework captures the essential mechanism for high-performance scalable cache consistency design and is still flexible enough to encompass a wide design space. This flexibility allows us to design for scalability and a range of consistency semantics while retaining low latency. Experiments have shown that Volume Lease is interactive, flexible enough to provide a range of guarantees, and scalable to demanding workloads.

- Interactive

Replication is used to improve the interactivity of a web application by replicating the service near users. The main design goal of scalable cache consistency is to allow replication to achieve its full potential in reducing latency. Our experimental results show that Volume Lease can improve read latency by a factor of two over the traditional client polling consistency protocols. Moreover, it provides latency close to the theoretical optimal.

- Flexible to Provide a Range of Guarantees

  Volume Lease can provide strong consistency. It can also allow web services to take advantage of weak consistency requirements to achieve better performance. When providing weak consistency, Volume Lease can still provide worst-case staleness bounds for reads in addition to low stale read rates.

- Scalable to Demanding Workloads

  Volume Lease can provide consistency for large-scale web services with millions of clients. The consistency protocols can perform well even with partial failures including client crashes and network partitions.

7.1 Future Direction

In this dissertation, we focus on scalable consistency protocols for web caching. In such systems, there is only one writer, the web server, and many readers, the web clients. A followup study can be the evaluation of Volume Lease in many-writer and many-reader systems. Such systems can possibly include replicated E-commerce systems. Volume Lease can conceivably provide scalable consistency for many-writer and many-reader systems efficiently. The future work is to collect such workloads

and evaluate Volume Lease with these workloads.

Another followup study is multiple-object consistency [82], the consistency across several objects. The exact requirements of multiple-object consistency are not clear to the author at this point in time. One conceivable requirement can be atomic writing, in which either all the writes to a set of objects are visible to clients or none of these writes is visible to clients. The research questions here are how to understand, describe, and classify the requirements of multiple-object consistency by web services, whether Volume Lease is the best protocol in providing multiple-object consistency, and how to adapt Volume Lease for multiple-object consistency.

It would also be interesting to see whether Volume Lease can be used to provide consistency semantics for other applications [43, 93]. Volume Lease should be beneficial as long as we can amortize the cost of a volume lease over many objects.

The goal of scalable cache consistency is to make the web the ideal platform in deploying software applications. Another challenge is security. The security challenge of web service stems from the fact that once a web service is deployed over the Internet, it can be accessed from anywhere in the Internet. While the Internet allows web services to be used by a large user population, it exposes web services to attacks from anywhere. Strengthening web security is a challenging and important problem. Given a potentially large user population that an important web service can serve, a large amount of hardware can be devoted to these services. Security through redundancy seems to be a promising technique.

7.2 Summary

As the web becomes the predominant platform for deploying software services, replication becomes essential in reducing user response time and improving availability. The effectiveness of web replication systems is determined by the efficiency of scalable cache consistency. This dissertation proposes and evaluates Volume Lease to provide scalable cache consistency. Our experimental results show that Volume Lease incurs little overhead, reduces user response time, and improves scalability compared to traditional client polling protocols. Moreover the performance of Volume Lease is close to the theoretical optimal. Scalable consistency combined with redundant security can make the web the ideal platform for deploying software applications.

Bibliography

[1] Fast internet content delievery with freeflow. *Akamai white paper*, November 1999.

[2] Web cache control protocol (wccp). *Cisco white paper*, 2001.

[3] Web cache coordination protocol version 2. *Cisco white paper*, 2002.

[4] A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Technical Report MIT/LCS/TR-786, MIT, 1999.

[5] D. Andersen, H. Balakrishnan, M. Kaashoek, and R. Morris. Resilient Overlay Networks. In *18th ACM Symposium on Operating Systems Principles*, pages 131–145, October 2001.

[6] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. *IEEE Network*, 14(3):201–262, May/June 2000.

[7] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *Measurement and Modeling of Computer Systems*, pages 126–137, 1996.

[8] M. Baker. *Fast Crash Recovery in Distributed File Systems.* PhD thesis, University of California at Berkeley, 1994.

[9] A. Barbir, B. Cain, F. Douglis, M. Hofmann, R. Nar, D. Potter, and O. Spatscheck. Known cdn request-routing mechanisms. IETF Internet Draft, December 2001. http://www.content-peering.org/docs/draft-cain-cdnp-known-request-routing-00.html.

[10] A. Belloum and L.O. Hertberger. Maintaining web cache coherency. *Information Research*, 6(1), 2000. http://informationr.net/ir/6-1/paper91.html.

[11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD'95*, pages 1–10, 1995.

[12] A. Biliris, C. Cranor, F. Douglis, M. Rabinovich, S. Sibal, O. Spatscheck, and W. Sturm. CDN brokering. *Computer Communications*, 25(4):393–402, March 2002.

[13] A. D. Bradley and A. Bestavros. Basis token consistency: Extending and evaluating a novel web consistency algorithm. In *Proceedings of the Workshop on Caching, Coherence and Consistency (in conjunction with ICS'02)*, June 2002.

[14] A. D. Bradley and A. Bestavros. Basis token consistency: supporting strong web cache consistency. In *Global Internet 2002*, volume 3, pages 2225 –2229, November 2002.

[15] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and

zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[16] B. E. Brewington and G. Cybenko. How dynamic is the Web? *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):257–276, 2000.

[17] P. Cao and C. Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4):445–457, 1998.

[18] V. Cate. Alex – A Global File System. In *Proceedings of the USENIX File System Workshop*, pages 1–11, Ann Arbor, Michigan, 1992.

[19] J. Challenger, A. Iyengar, and P. Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE INFOCOM'99*, volume 1, pages 294–303, March 1999.

[20] 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*, pages 56–61, October 1997.

[21] 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)*, pages 237–248, May 1996.

[22] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[23] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshnes. In *VLDB*, pages 117–128, 2000.

[24] E. Cohen and H. Kaplan. The age penalty and its effect on cache performance. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 73–84, May 2001.

[25] E. Cohen and H. Kaplan. Aging through cascaded caches: Performance issues in the distribution of web content. *Proceedings of SIGCOMM'2001*, page 13, 2001.

[26] E. Cohen and H. Kaplan. Refreshment policies for web content caches. In *INFOCOM*, pages 1398–1406, 2001.

[27] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving end-to-end performance of the web using server volumes and proxy filters. In *SIGCOMM*, pages 241–253, 1998.

[28] http://httpd.apache.org/docs/logs.html.

[29] I. Cooper, D. Li, M. Dahlin, and M. Hamilton. Requirements and guidelines for a resource update protocol. *IETF(draft-ietf-webi-rup-reqs-03)*, August 2002.

[30] M. Dahlin. *Serverless Network File Systems*. PhD thesis, University of California at Berkeley, December 1995.

[31] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[32] Digital Equipment Corporation. Digital's Web Proxy Traces. ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html, September 1996.

[33] 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.

[34] J. Dilley. The effect of consistency on cache response time. Technical Report 1999-107, HPL, 1999. http://www.hpl.hp.com/techreports/1999/HPL-1999-107.html.

[35] A. Dingle and T. Partl. Web cache coherence. *Computer Networks and ISDN Systems*, 28:907–920, May 1996.

[36] A. Dingle and T. Partl. Web cache coherence. In *Proceedings of the 5th International WWW Conference*, Paris, France, May 1996. http://www5conf.inria.fr/fich_html/papers/P2/Overview.html.

[37] V. Duvvuri, P. Shenoy, and R. Tewariy. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *INFOCOM'2000*, volume 2, pages 834–843, March 2000.

[38] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proc. of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1998.

[39] Z. Fei. A novel approach to managing consistency in content distribution networks. In *Proceedings of Web Caching and Content Distribution Workshop (WCW'01)*, pages 71–86, Boston, MA, June 2001.

[40] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Internet-Draft draft-ietf-http-v11-spec-07, HTTP Working Group, August 1996.

[41] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, 1999.

[42] S. Gadde, J. Chase, and M. Rabinovich. Directory Structures for Scalable Internet Caches. Technical Report CS-1997-18, Duke University Department of Computer Science, November 1997.

[43] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *Proceedings of the 2003 International World Wide Web*, Budapest, HUNGARY, May 2003.

[44] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[45] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD'96*, pages 173–182, 1996.

[46] M. Green, B. Cain, G. Tomlinson, and S. Thomas. Cdn peering architectural overview. IETF Internet Draft, May 2001. http://www.content-peering.org/docs/draft-green-cdnp-gen-arch-02.html.

[47] J. Gwertzman and M. I. Seltzer. World wide web cache consistency. In *USENIX Annual Technical Conference*, pages 141–152, 1996.

[48] J. Gwertzman and M. I. Seltzer. World wide web cache consistency. In *USENIX Annual Technical Conference*, pages 141–152, 1996.

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

[50] A. Iyengar, M. S. Squillante, and L. Zhang. Analysis and characterization of large-scale web server access patterns and performance. *World Wide Web*, 2(1-2):85–100, 1999.

[51] J. Jung, A. Berger, and H. Balakrishnan. Modeling ttl-based internet caches, April 2003. http://www.ieee-infocom.org/2003/technical_programs.htm#Caching%20and%20Web%20performance.

[52] J. Kangasharju, J. Roberts, and K. Ross. Object replication strategies in content distribution networks. *Computer Communications*, pages 27–42, February 2002.

[53] A. Kermarrec, I. Kuz, M. van Steen, and A. S. Tanenbaum. A framework for consistent, replicated web objects. In *International Conference on Distributed Computing Systems*, pages 276–291, 1998.

[54] M. R. Korupolu and M. Dahlin. Coordinated placement and replacement for large-scale distributed caches. *IEEE Transactions of Knowledge and Data Engineering*, 14(6):1317–1329, Nov/Dec 2002.

[55] B. Krishnamurthy and C. E. Wills. Piggyback server invalidation for proxy

cache coherency. *Computer Networks and ISDN Systems*, 30(1–7):185–193, 1998.

[56] B. Krishnamurthy and C. E. Wills. Proxy cache coherency and replacement - towards a more complete picture. In *International Conference on Distributed Computing Systems*, pages 332–339, 1999.

[57] D. Li, P. Cao, and M. Dahlin. Wcip: Web cache invalidation protocol. *IETF(draft-danli-wrec-wcip-01)*, March 2001.

[58] D. Li and D. Cheriton. Scalable Web Caching of Frequently Updated Objects Using Reliable Multicast. In *Second USENIX Symposium on Internet Technologies and Systems*, pages 1–12, Oct 1999.

[59] M. Mikhailov and C. E. Wills. Exploiting object relationships for deterministic web object management. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution*, Boulder, Colorado, August 2002.

[60] M. Mikhailov and C. E. Wills. Evaluating a new approach to strong web cache consistency with snapshots of collected content. In *Proceedings of the Twelfth International World Wide Web Conference*, pages 599–608, Budapest, HUNGARY, May 2003.

[61] N. Mittal and V. K. Garg. Consistency conditions for multi-object distributed operations. In *International Conference on Distributed Computing Systems*, pages 582–599, 1998.

[62] J. Mogul. http://www.roads.lut.ac.uk/lists/http-caching/1996/01/0002.html.

[63] J. Mogul. Recovery in Spritely NFS. *Computing Systems*, 7(2):201–262, Spring 1994.

[64] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[65] A. Ninan. Maintaining cache consistency in content distribution networks, 2001.

[66] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Cooperative leases: Scalable consistency maintenance in content distribution networks. In *Proceedings of the Eleventh World Wide Web Conference (WWW-10)*, pages 1–12, Honolulu, Hawaii, May 2002.

[67] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Scalable consistency maintenance in content distribution networks using cooperative leases. *IEEE Transactions of Knowledge and Data Engineering*, 15(4):813–828, July/August 2003.

[68] W. B. Norton. Internet service providers and peering. In *Proceedings of NANOG 19*, Albuquerque, New Mexico, June 2000.

[69] V. N. Padmanabhan and L. Qui. The content and access dynamics of a busy web site: findings and implicatins. In *SIGCOMM*, pages 111–123, 2000.

[70] F. Pedone and R. Guerraoui. On transaction liveness in replicated databases. In *Proceedings of IEEE Pacific Rime International Symposium on Fault-Tolerant Systems (PRFTS'97)*, pages 104–109, 1997.

[71] S. Perret, J. Dilley, and M. Arlitt. Performance evaluation of the dis-

tributed object consistency protocol. Technical Report 1999-108, HPL, 1999. http://citeseer.nj.nec.com/perret99performance.html.

[72] C. Plaxton, R. Rajaram, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, June 1997.

[73] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *INFOCOM*, pages 1587–1596, 2001.

[74] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. Coordinated Atomic Actions: from Concept to Implementation. (595), 1997.

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

[76] S. Shah, K. Ramamritham, and P. Shenoy. Maintaining Coherency of Dynamic Data in Cooperating Repositories. In *Proceedings of the 28th Conference on Very Large Data Bases (VLDB)*, pages 526–537, Hong Kong, August 2002.

[77] A. Singla, U. Ramachandran, and J. K. Hodgins. Temporal notions of synchronization and consistency in beehive. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 211–220, 1997.

[78] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design considerations for distributed caching on the internet. In *International Conference on Distributed Computing Systems*, pages 273–284, 1999.

[79] R. Tewari, T. Niranajan, and S. Ramamurthy. Wcdp: A protocol for web cache consistency. In *Proceedings of Web Caching and Content Distribution Workshop (WCW'02)*, Boulder, Colorado, August 2001.

[80] R. Tewari, T. Niranajan, and S. Ramamurthy. Requirements and guidelines for a resource update protocol. *IETF(draft-tewari-webi-wcdp-00.txt)*, February 2002.

[81] F. J. Torres-Rojas, M. Ahamad, and M. Raynal. Timed consistency for shared distributed objects. In *Symposium on Principles of Distributed Computing*, pages 163–172, 1999.

[82] B. Urgaonkar, A. Ninan, P. Shenoy, M. Raunak, and K. Ramamritham. Maintaining mutual consistency for cached web objects. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 371–380, Phoenix, AZ, April 2001.

[83] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirecion on cdn robustness. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 345–360, Boston, MA USA, December 2002.

[84] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-Based Analysis of Web-Object Sharing and Caching. In *Second USENIX Symposium on Internet Technologies and Systems*, pages 1–11, October 1999.

[85] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-Based Analysis of Web-Object

Sharing and Caching. In *Second USENIX Symposium on Internet Technologies and Systems*, pages 1–11, October 1999.

[86] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles*, pages 16–31, 1999.

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

[88] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering web cache consistency. *ACM Transactions on Internet Technology (TOIT)*, 2(3):224–259, 2002.

[89] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using leases to support server-driven consistency in large-scale systems. In *International Conference on Distributed Computing Systems*, pages 285–294, 1998.

[90] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[91] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume leases for consistency in large-scale systems. *Knowledge and Data Engineering*, 11(4):563–576, 1999.

[92] H. Yu, L. Breslau, and S. Shenker. A scalable web cache consistency architecture. In *SIGCOMM*, pages 163–174, 1999.

[93] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. In *Symposium on Operating Systems Principles*, pages 29–42, 2001.

Vita

Jian Yin was born in Shangdong, China, on April 6th, 1972. He completed his B.S. at the University of Wyoming in 1994. He graduated as the Best Computer Science Graduate and among the Top Twenty Graduates in the Art and Science College. After working for a year as a research assistant, he started his Ph.D. study in Computer Sciences in 1995 with a MCD fellowship. He has published several papers in the top conferences and journals in distributed systems. One of his papers won the Best Paper Award from the Tenth World Wide Web conference.

Permanent Address: 1 Charter Circle
Yorktown, NY 10560

This dissertation was typeset with LaTeX 2_ε by the author.