View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Carolina Digital Repository

WACCO AND LOKO: STRONG CONSISTENCY AT GLOBAL SCALE

Darrell Bethea

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2015

Approved by:

Michael K. Reiter

Jay Aikat

Bruce Maggs

Z. Morley Mao

F. Donelson Smith

© 2015
Darrell Bethea
ALL RIGHTS RESERVED

ABSTRACT

DARRELL BETHEA: WACCO AND LOKO:
STRONG CONSISTENCY AT GLOBAL SCALE
(Under the direction of Michael K. Reiter)

Motivated by a vision for future global-scale services supporting frequent updates and widespread concurrent reads, we propose a scalable object-sharing system called WACCO offering strong consistency semantics. WACCO propagates read responses on a tree-based topology to satisfy broad demand and migrates objects dynamically to place them close to that demand. To demonstrate WACCO, we use it to develop a service called LOKO that could roughly encompass the current duties of the DNS and simultaneously support granular status updates (e.g., currently preferred routes) in a future Internet. We evaluate LOKO, including the performance impact of updates, migration, and fault tolerance, using both traces of DNS queries served by Akamai and traces of NFS traffic on the UNC campus.

WACCO uses a novel consistency model that is both stronger than sequential consistency and more scalable than linearizability. Our results show that this model performs better in the DNS case than the NFS case because the former represents a global, shared-object system which better fits the design goals of WACCO. We evaluate two different migration techniques, one of which considers not just client-visible latency but also the budget for the network (e.g., for public and hybrid clouds) among other factors.

To my mother and father, to whom I owe everything.

ACKNOWLEDGMENTS

I thank my advisor, Dr. Michael Reiter, for accepting me as his student and for all that he taught me at UNC. I will certainly miss our weekly meetings—sometimes insightful, sometimes hilarious, and often both. I thank my other committee members: Dr. Jay Aikat, Dr. Bruce Maggs, Dr. Morley Mao, and Dr. Don Smith for their feedback, their advice, and the time they committed to serving on my committee.

I am grateful for everyone who helped me obtain real-world data sets, including: Dr. Bruce Maggs, KC Ng, Akamai Technologies, Inc., Dr. John Strunk, NetApp, Inc., Charles Hammitt, and UNC Research Computing.

I thank all my collaborators, including: Dr. Morley Mao and everyone at the University of Michigan who helped during the early part of this work; Dr. Bruce Maggs who helped me understand the difficulties of working with large data sets; and Dr. John Strunk, who had good ideas and suggestions every time I spoke to him about my progress.

Funding for my work was generously provided by NSF Award Number 1040626 and a fellowship from NetApp, Inc.

Dr. Fred Brooks deserves a special thanks for unknowingly inspiring me in a number of ways. Thanks, too, to every other teacher I have had along the way—in school or out—so many of whom went out of their way to help me understand. I have not forgotten.

Thanks to all my friends, without whom my time at UNC would not have been nearly as enjoyable. In particular, thanks to Alana Libonati, Robby Cochran, Srinivas Krishnan, and the now-Drs. Catie Welsh, Brittany Millman, and Dave Millman: my Bandidos amigos and fellow graduate school sufferers. I also thank the people at The Think Tank for encouraging me to write things up and everyone down at Dave's Appliance Repair for distracting me from work from time to time.

I thank Alana Libonati, who knows me so well and endures me nonetheless. I lean on her when I need support, and she is always there to give it. She is my closest friend and my loving companion.

Finally, I thank my parents, Larry and Maggie Bethea, who have sacrificed more for me than I can ever repay. They, along with my brother and sister, Brian Bethea and Stacey Rigsby, have shown unwavering support and unending confidence in me, which has carried me to this point. I could never have done this without them. I love you all.

TABLE OF CONTENTS

LIST OF FIGURES . . . . **x**

1 Introduction **1**

2 Background **5**

2.1 Consistency models 5

2.1.1 PRAM consistency 6

2.1.2 Causal consistency 7

2.1.3 Sequential consistency 9

2.1.4 Linearizability 10

2.1.5 Eventual consistency 12

2.2 CAP theorem and PACELC 13

2.3 Other related work 15

3 Protocol Design **18**

3.1 Design considerations and goals 18

3.2 WACCO design 21

3.2.1 Basic protocol 22

3.2.2 Caching 24

3.2.3 Migration 25

3.2.4 Resilience 27

4 Cluster Consistency **29**

4.1 Definition 29

4.2 Proof of cluster consistency 30

5 Case study: DNS traces . . . 38

5.1 LOKO . . . 38

5.2 Traces . . . 39

5.3 Experimental setup . . . 40

5.4 Experimental results . . . 46

5.5 Limitations . . . 52

6 Case study: NFS traces . . . 54

6.1 Introduction . . . 54

6.2 Differences from DNS case study . . . 55

6.3 Design changes . . . 55

6.3.1 Reducing memory usage . . . 56

6.3.2 Reducing network load: migration . . . 56

6.3.3 Reducing network load: block requests . . . 57

6.4 Changes to the data set . . . 58

6.5 Experimental setup . . . 59

6.6 Evaluation . . . 60

6.6.1 Problems with the existing setup . . . 62

6.7 Changes to the data and setup . . . 65

6.7.1 Client duplication . . . 65

6.7.2 Log window . . . 66

6.7.3 File size . . . 67

6.8 Evaluation of NFS-β . . . 67

6.9 Conclusion . . . 69

7 Migration strategies . . . 71

7.1 Computing optimal object placement . . . 71

7.2 Unrestricted, global BILP . . . 72

7.2.1 Decision variables 73
7.2.2 Compile-time values 73
7.2.3 Run-time values explicitly measured 74
7.2.4 Run-time values derived from other values 74
7.2.5 BILP 76
7.2.6 Optional constraints 76
7.2.7 Assumptions and limitations 77
7.3 Local, per-proxy BILP 78
7.3.1 Decision variables 78
7.3.2 Compile-time values 79
7.3.3 Run-time values explicitly measured 80
7.3.4 Run-time values derived from other values 80
7.3.5 BILP 83
7.3.6 Optional constraints 84
7.3.7 Benefits, limitations, and assumptions 88
7.4 Experimental setup 89
7.4.1 Configuration and environment 89
7.4.2 Forbidden locations 90
7.4.3 Migration limits 90
7.4.4 Caching 90
7.4.5 Setting maximum request rate $\lambda_{\max}$ 91
7.4.6 Infeasibility 92
7.5 Evaluation results 93
7.6 Conclusion 95
8 Conclusion 97
BIBLIOGRAPHY 98

LIST OF FIGURES

Figure 2.1 An example execution history for a single object. Time increases left-to-right. Each row denotes one process. 6

Figure 2.2 Execution histories for a single object. Time increases left-to-right. Each row denotes one client. 7

Figure 2.3 A causally consistent execution history for a single object. Time increases left-to-right. Each row denotes one client. 8

Figure 2.4 An sequentially consistent execution history for a single object. Time increases left-to-right. Each row denotes one process. 10

Figure 2.5 Execution histories for a single object. Time increases left-to-right. Each row denotes one client. 11

Figure 3.1 Example of pausing some reads and resuming them later 21

Figure 4.1 Execution histories for a single object. Time increases left-to-right. Each row denotes one client. 31

Figure 5.1 Keyspace query and size distributions 44

Figure 5.2 CDFs of latencies (ms) as u varies. 47

Figure 5.3 Impact of varying m, with $u = 0.0$. Lines for some values of m are omitted from Figure 5.3(b) for clarity. 49

Figure 5.4 CDFs of latencies (ms) when using backups, with $u = 0.01$. 50

Figure 5.5 Throughput and messaging overhead as load factor varies, with $u = 0.01$. 50

Figure 5.6 Impact of varying load factor on median latency and total bytes sent in both the cluster consistent (CC) and linearizable (LIN) versions of LOKO. 52

Figure 6.1 The effects of changing the migration cutoff. 61

Figure 6.2 The effects of changing b. 63

Figure 6.3 CDFs of latency as c and b change 64

Figure 6.4 Effects of varying migration threshold m in NFS-β 68

Figure 6.5 Effects of varying load factor in NFS-β 70

Figure 7.1 Impact on latency and run cost of varying $P'_{\max}$, with $u = 0.01$. 94

Figure 7.2 Impact on various measurements of varying $P'_{\max}$, with $u = 0.01$. 96

Chapter 1: INTRODUCTION

Today's Internet is served by infrastructures that, in general, scale remarkably well to the massive demands placed on them. Both the Domain Name System (DNS) and content-distribution networks (CDNs) are examples of dramatic feats of engineering that facilitate global and quick access to content. The power of these infrastructures, however, derives in part from the largely static nature of the data they serve. DNS scales through caching on the basis of time-to-live (TTL) values that are typically large enough to hide updates from parts of the network for minutes or hours. CDNs serve primarily static data or else data that, if updated, need not be viewed consistently by different parts of the network.

The viability of such approaches may be challenged, however, as the Internet evolves. Multiple visions for future Internet designs anticipate the need to support more dynamic information in the network (e.g., SCION's address and path servers [83], NIRA's NRLS [81], or rendezvous servers to support mobility in content-centric networking [46]), which may enable, e.g., mobile network location, dynamic route control, or diagnosis of network anomalies. Because this information can change quickly—in some cases at the granularity of seconds or less—there is a need for infrastructure services that support dynamic updates, strong consistency, and global scalability. Even for existing uses to direct clients to servers or to exercise route control, today's DNS has limited ability to provide fine-grained control [62, 65], and we expect this shortcoming to become more acute in the future.

This dissertation describes a system called Wide-Area Cluster-Consistent Objects (WACCO). WACCO manages access to stateful, deterministic *objects* that support invocations of arbitrary types, each of which is either an *update* that may modify object state or a *read* that does not. Objects are managed on a tree-based overlay network of *proxies* that is

arranged with respect to geography; i.e., neighbors in the tree tend to be close geographically or, more to the point, enjoy low latency between them. Each client is assigned to a nearby proxy to which it connects to access objects, and object access is managed through a protocol that offers a novel consistency model that we dub *cluster consistency.* Cluster consistency is strong: it ensures sequential consistency [49] and also that *clusters* of concurrent reads see the most recent preceding update to the object on which the reads are performed. The resulting agreed-upon order and rapid visibility of updates facilitate a wide range of applications, e.g., network troubleshooting, trajectory tracking of mobile nodes, and content-oriented network applications.

Scalability of services implemented using WACCO is achieved through two strategies. First, WACCO uses the tree structure of the overlay to aggregate read demand, permitting the responses to some reads to answer others. As such, under high read concurrency, the vast majority of reads are not propagated to the location of the object; rather, most are *paused* awaiting others to complete, from which the return result can be "borrowed". Second, WACCO uses migration to dynamically change where each object resides, permitting the object to move closer to demand as it fluctuates, e.g., due to diurnal patterns.

To demonstrate and evaluate WACCO, we use it to build a service called Low-Overhead Keyspace Objects (LOKO). LOKO permits clients to create, modify and query *keyspace* objects. A keyspace is identified by a public key *pk*, and the keyspace for *pk* stores (or generates) mappings, each from a query string *qstr* to a value *val* and bearing a digital signature that can be verified by *pk*. So, for example, querying the keyspace for *pk* for the string `nytimes/publicKey` might return the signed public key certificate that the owner of *pk* believes to be for `nytimes`. Similarly, the query `www/bestRoute` on the keyspace identified by pk' might return a signed mapping indicating the currently preferred route to reach the web server representing the owner of pk'. By iterating queries to a "chain" of keyspaces, each referring the client to the next keyspace in the chain, a client could securely resolve a multipart pathname, much as is done with DNSSEC [13]. In this respect, LOKO

could encompass one of the main duties of today's DNS/DNSSEC, while supporting more dynamic mappings due to the consistency provided by WACCO.

In evaluating LOKO (and WACCO), we were handicapped in not having a global workload for such a service. So, we approximated a global workload in two case studies, each using a different data set. The first case study uses a trace of over 4.4 billion DNS requests served by Akamai servers over 36 hours to 83,448 clients in four geographic regions across Asia, North America and Europe. We used this trace to drive 76-proxy emulations of LOKO with network delays induced to represent a LOKO deployment across these four regions. Our emulations show that LOKO provides good latency for operations, e.g., with up to 89% of reads completing in under 100 ms. We also show that our implementation can sustain the full per-proxy query rate represented by the Akamai trace, while guaranteeing cluster consistency.

The second case study uses a trace of NFS requests collected on the UNC campus with the help of NetApp and their Chronicle project [43]. We adapt LOKO to handle the new challenges presented by the NFS workload and evaluate the results. We also evaluate LOKO on a version of the NFS workload that more closely fits with the original design goals of WACCO. We illustrate the effectiveness of the components of our design using measurements from these two case studies.

We also use the data set from the first case study to evaluate an alternative migration strategy that moves beyond a purely latency-based view object placement. This new strategy uses integer linear programming to allow WACCO to make migration decisions using a number of additional, user-configurable factors, including traffic cost, hosting cost, proxy capacities, etc.

We begin by presenting some background in Chapter 2. We discuss the design of WACCO in Chapter 3. We define cluster consistency and prove that our protocol implements it in Chapter 4. The first case study, in which we use DNS traces to evaluate LOKO, appears in Chapter 5. The second, which uses NFS traces, appears in Chapter 6.

Chapter 7 includes our discussion of the alternate migration strategy, and we conclude in Chapter 8.

Chapter 2: BACKGROUND

2.1 CONSISTENCY MODELS

When choosing a distributed system, an administrator might have some questions about the behavior of the system. For example: (i) When one client in a distributed system updates the value of some shared state, how long will it be before a second client can read the new value? (ii) If two clients each separately update some shared state, in which order will a third client observe those updates to have taken place? (iii) Will a fourth client's observations always agree with those of the third client? (iv) Do the answers change if the two updates came from a single client instead of two? The answers to these and related questions compose what are called the *consistency semantics* of the system. The consistency semantics are quantified guarantees by a system about what clients see what object versions and when. They describe the kinds of guarantees the administrator can expect the system to provide—and the kinds of caveats to be aware of. For example, the semantics could allow a delay between when a client writes a value and when the new value can be read back, or they may ensure that written values can always be immediately read. In our discussion below we may refer to clients as client processes or simply *processes*.

One way researchers classify distributed systems is by grouping them by their consistency semantics using different consistency models. Each model comprises a set of these semantic guarantees, and any system implementing those semantic guarantees (at least) is said to implement or to use the consistency model.

Researchers have studied these models in the contexts of distributed and shared-memory systems for decades; we cannot hope to cover the entire field here. Instead, we will explain some existing models, showing both their individual semantics and their relationships to

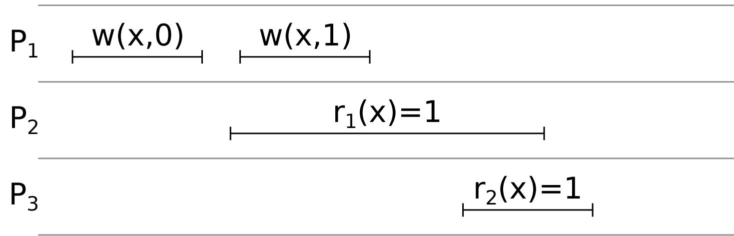


Figure 2.1: An example execution history for a single object. Time increases left-to-right. Each row denotes one process.

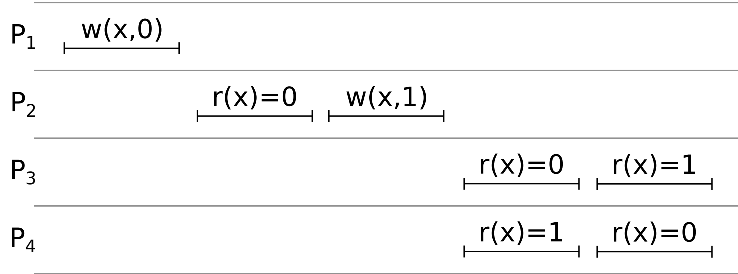
one another and to cluster consistency.

Throughout this section, we will use diagrams like the one in Figure 2.1 to illustrate execution histories of shared memory systems under various conditions and consistency models. Each row represents a separate process, with time increasing from left to right. Read and write operations performed by a process appear on its row, marked with the variable affected and the value read or written. For example, $\mathsf{w}(x, 0)$ in the first row indicates that process P_1 wrote 0 to the variable x. Similarly, $\mathsf{r}_2(x) = 1$ in the third row means that P_3 read the variable x and got the value 1. Read and write operations are sometimes subscripted (as both read operations are above) only to make them easier to isolate in our discussion—the subscripts have no bearing on the execution history itself. Finally, the lines beneath each operation indicate its duration, from invocation to completion (as seen at the process invoking the operation).

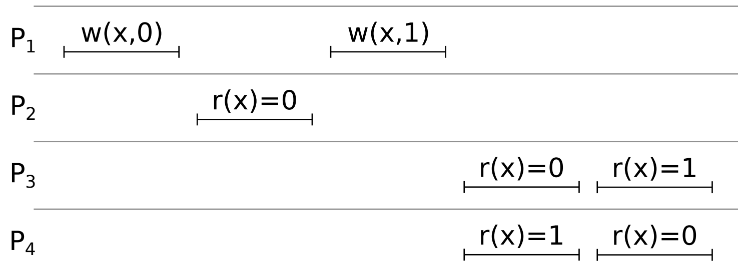
2.1.1 PRAM CONSISTENCY

The weakest consistency model we will discuss in detail is the pipelined RAM (PRAM) [51, 59] model.[1] This model guarantees only that each process P_i observe the write operations from each other process P_j in the order in which they were actually performed by P_j. However, each P_i may observe the write operations originating from *different* processes as being interleaved in a different order. Figure 2.2(a) shows an example of such an execution

[1]The PRAM consistency model is unrelated to the Parallel Random Access Machine [31], a variant of the (sequential) Random Access Machine used in algorithm analysis.



(a) A PRAM-consistent execution history for a single object.



(b) An execution history that is not PRAM-consistent, since P_4 has reordered the updates from P_1.

Figure 2.2: Execution histories for a single object. Time increases left-to-right. Each row denotes one client.

history in a system that implements PRAM consistency. Processes P_3 and P_4 disagree on the order in which the updates from P_1 and P_2 take place. But, since no P_i has re-ordered updates from a single process—each process issues at most one update in this execution history—the history is PRAM-consistent. In Figure 2.2(b) we have created a new history by altering the previous one so that P_1 performs both updates. Since P_4 is not allowed to reorder the updates (but has done so in the history), this altered history is not PRAM-consistent.

2.1.2 CAUSAL CONSISTENCY

Causal consistency [12, 41, 59] demands that all processes agree on the order of updates that are potentially causally related. There is no ordering guarantee for updates that are causally unrelated. Lamport [48] covered causality in detail, but essentially what it means

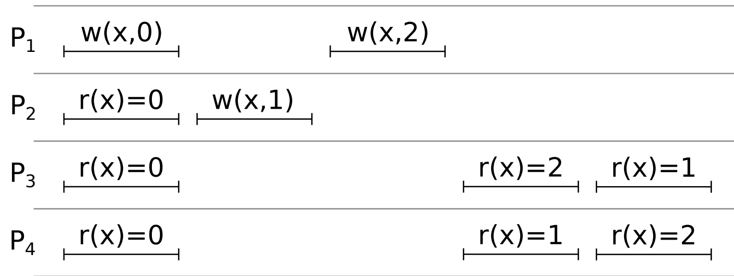


Figure 2.3: A causally consistent execution history for a single object. Time increases left-to-right. Each row denotes one client.

here is that if a process reads a value then later writes another value, there is a possibility that the value that was read influenced the choice of what to write. For example, imagine a process that reads a number, then writes back its square. The process first reads $\mathsf{r}(x) = 7$ then later does $\mathsf{w}(x, 49)$. In causal consistency, all processes must agree that $\mathsf{w}(x, 7)$ came before $\mathsf{w}(x, 49)$. Intuitively, we know that the fact that the read has occurred means that $\mathsf{w}(x, 7)$ may have influenced the choice to do $\mathsf{w}(x, 49)$ (as indeed it has).

Multiple updates by the same process share a causality link as well. Imagine our squaring process starts with 1 and starts writing squared numbers to x consecutively: e.g., $\mathsf{w}(x, 1)$, $\mathsf{w}(x, 4)$, $\mathsf{w}(x, 9)$, and so on. The update $\mathsf{w}(x, 4)$ causally precedes $\mathsf{w}(x, 9)$—it must, or $\mathsf{w}(x, 9)$ would not be written at all. Because causality holds between updates from a single process, and because all processes must see updates in causality order, all processes must see updates from a single process in the same (correct) order. Therefore, causal consistency implies PRAM consistency.

The execution history in Figure 2.3 is causally consistent (and thus PRAM consistent). There is disagreement between P_3 and P_4 about whether $\mathsf{w}(x, 1)$ or $\mathsf{w}(x, 2)$ happened first, but those two updates are not causally related. Because $\mathsf{w}(x, 0)$ is written before $\mathsf{w}(x, 2)$ by the same process, those two writes are causally related. And because P_2 reads $\mathsf{r}(x) = 0$ before $\mathsf{w}(x, 1)$, $\mathsf{w}(x, 0)$ must precede $\mathsf{w}(x, 1)$. But, no causal relationship binds the order of $\mathsf{w}(x, 1)$ and $\mathsf{w}(x, 2)$.

Note, though, that the set of causally consistent systems is a strict superset of the set

of PRAM-consistent systems. For example, the execution history in Figure 2.2(a) is not causally consistent, because the read at P_2 establishes a causal relationship between $\mathsf{w}(x, 0)$ and $\mathsf{w}(x, 1)$, which P_4 does not honor.

Though weaker than the next two models, causal consistency is sometimes used in distributed services—e.g., COPS [52], Eiger [53]—because the concurrency it allows can make it faster than linearizability, and it is stronger than eventual consistency (both discussed below). Another advantage is that causal consistency can be easy to add to existing system: Bailis et al. [17] have developed a "bolt-on" architecture that can provide causal consistency given an eventually consistent back-end. We discuss consistency choices and tradeoffs more in Section 2.2.

2.1.3 SEQUENTIAL CONSISTENCY

Sequential consistency [49] is a consistency model that implies causal consistency but is strictly stronger. To be sequentially consistent, an execution history must be indistinguishable from one in which all the operations (by all processes) were executed in some sequential order (a total ordering of all operations) that does not reorder any single process's operations (i.e., preserves local order).

The effect is equivalent to trying to find a legal execution history by transforming an execution diagram according to the following rules: (i) operations can be moved left or right along their row but cannot pass other operations in that row (preserves local order); (ii) only one operation can be active at a time (gives a sequential history); and (iii) read operations must always return the value most recently written, according to the diagram (a property called history *legality*). If (and only if) a new diagram can be formed using those rules, then the execution is sequentially consistent. Figure 2.4 shows a sequentially consistent execution. The first rule allows us to move $\mathsf{w}(x, 1)$ later, so that it effectively takes place after both read operations. The resulting sequential history preserves local order and is legal, and so it is sequentially consistent. However, the causally consistent example

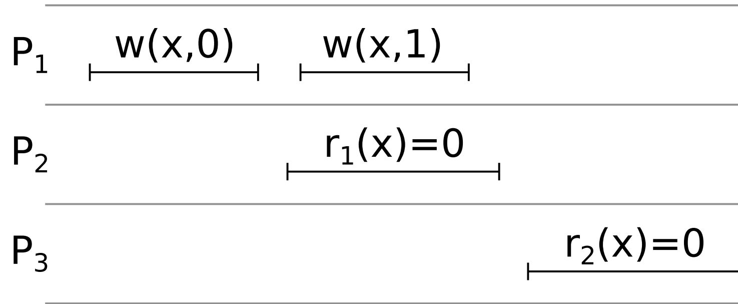


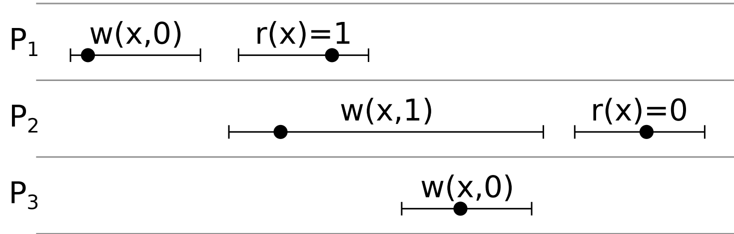
Figure 2.4: An sequentially consistent execution history for a single object. Time increases left-to-right. Each row denotes one process.

in Figure 2.3 is not sequentially consistent. Consider a potential sequential ordering of its execution history. If the ordering places $\mathsf{w}(x,1)$ before $\mathsf{w}(x,2)$, then there is no legal way to order the last two read operations at P_3 without violating local order. If instead $\mathsf{w}(x,2)$ comes before $\mathsf{w}(x,1)$ in the ordering, then we have the same problem but at P_4.

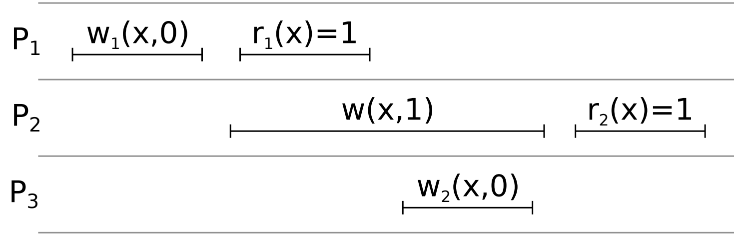
Sequential consistency implies causal consistency because in sequential consistency all processes agree on the order of all operations, not just potentially causally related ones. Also, the combination of the local-order and legality rules ensures that the sequential ordering will respect the potential causality requirements. There are many existing protocols and systems designed to guarantee sequential consistency [24, 30, 50, 73], due to its stronger guarantees than causal consistency.

2.1.4 LINEARIZABILITY

One weakness of sequential consistency is that it is defined in such a way that some operations could potentially be delayed indefinitely. For example, imagine a version of Figure 2.4 in which P_2 and P_3 do many more read operations, each time with the result $\mathsf{r}(x) = 0$. The result is still a sequentially consistent execution, no matter how far out in time the read requests span, because $\mathsf{w}(x,1)$ can always be delayed until after them, according to the rules above. Linearizability [40] addresses this weakness by adding a time constraint to sequential consistency. To be linearizable, an execution history must be sequentially consistent and also have the following property: that each operation must



(a) A linearizable execution history for a single object.



(b) An execution history that is not linearizable, since no effective time for $\mathsf{w}(x, 1)$ will result in a legal history.

Figure 2.5: Execution histories for a single object. Time increases left-to-right. Each row denotes one client.

appear—to all processes—to have taken place at a single instant in time, somewhere between its invocation time and its return time (i.e., on the line beneath it on the diagram). This liveness guarantee comes at significant performance and scalability cost, as shown both in theory [14, Ch. 9] and in practice [11, 34, 66, 76].

Figure 2.5(a) shows an execution history that is linearizable. The dot (•) on the duration line shows the time at which an operation is (effectively) visible to all other processes. Notice that $\mathsf{w}(x, 1)$ takes effect before $\mathsf{r}(x) = 1$, to maintain legality. Figure 2.5(b) shows a variant of the history in which the last read operation returns a different value. This new execution again requires that $\mathsf{w}(x, 1)$ must take effect before $\mathsf{r}_1(x) = 1$ does—i.e., before $\mathsf{r}_1(x) = 1$ ends. But $\mathsf{w}(x, 1)$ must also take effect after $\mathsf{w}_2(x, 0)$ does—which must certainly be after $\mathsf{w}_2(x, 0)$ starts—so that $\mathsf{r}_2(x) = 1$ can return the correct value. These required intervals are non-overlapping, and so there is no possible effective time for $\mathsf{w}(x, 1)$ that will preserve legality, and the execution history is not linearizable.

The example execution history from Figure 2.4 is also not linearizable, since $\mathsf{w}(x, 1)$

cannot be delayed long enough to allow $\mathsf{r}_2(x) = 0$ to read the earlier value, and so linearizability is not subject to the same potential staleness as sequential consistency.

WACCO introduces a new consistency model called cluster consistency that sits on the consistency spectrum between sequential consistency and linearizability—i.e., linearizability implies cluster consistency, and cluster consistency implies sequential consistency. We give formal definitions for both cluster consistency and sequential consistency, as well as a proof that cluster consistency implies sequential consistency in Chapter 4.

Scatter [35] is one system that provides linearizability. However, partly due to its use of distributed hash tables, it does not offer the same benefits of request aggregation and geographic proximity that WACCO achieves through its tree structure and migration. Spanner [27] also implements linearizability, though it does so in part by relying on synchronized real-time clocks, which WACCO does not, and again does not leverage request aggregation. Other systems provide either linearizability or a weaker consistency model, as a configuration option. For example, Quiver [66], from which WACCO borrows certain features, supports either linearizability or sequential consistency. Apache Cassandra [5] provides several levels of consistency configurations, from linearizability down to eventual consistency.

2.1.5 EVENTUAL CONSISTENCY

There are some weak consistency models less related to our work on cluster consistency. Most notably, a model called eventual consistency [74, 77] guarantees only that, if enough time passes with no new updates to a variable, then eventually read operations at all processes will return the same value. DNS is one notable example of a widely used, eventually consistent system on the Internet today: after the time-to-live (TTL) for all stale DNS entries has passed, all DNS clients should read the most recently written value (assuming no other updates have taken place since then).

Despite its lack of concrete guarantees, eventual consistency has been a popular choice

for distributed systems—e.g., Amazon S3 [3], Bayou [74]—because it can yield better performance with consistency that works "well enough." In practice, eventually consistent systems have advanced to the point that they can perform indistguishably from strongly consistent systems for a significant portion of operations (and after a small interval) [16]. Still, designers must often add an extra layer of application logic to check for inconsistencies and repair them when they do arise—e.g., if two customers each appear to have bought the last seat on a plane, or if a bank customer manages to withdraw more money from an ATM than was actually in his account [16, 19]. For cases in which these kinds of inconsistencies (generally caused by reading stale values or reordering updates) are not an option, a stronger consistency model is best. A popular design today is to support eventual consistency as a baseline but allow the user to optionally upgrade to stronger consistency: e.g., Apache Cassandra [5], Amazon DynamoDB [1], Amazon SimpleDB [4], and Oracle NoSQL Database [9].

2.2 CAP THEOREM AND PACELC

Proven by Gilbert and Lynch in 2002 [34], the CAP theorem is a well-known, if sometimes misunderstood, rule for designers of distributed systems. Simply put, CAP defines three desirable properties—consistency, availability, and partition-tolerance—and says that a distributed system can have at most two of these properties. So, for example, a designer can choose to create an "AP" system, which can tolerate partitions by continuing to be available to clients despite the possibility of inconsistent results. Specifically, the best achievable consistency in an "AP" system is a variant of causal consistency [55]. A "CP" system would respond to a partition by reducing availability, perhaps rejecting client requests rather than develop (or portray) an inconsistent view of the data. Note that it is not possible to forgo partition tolerance and create a "CA" system. If a partition in such a system were to occur—and whether it does is not up to the designer of the system—the system will inevitably choose how to react to new requests, either by staying consistent at

the cost of availability or vice versa.

As a result of the CAP theorem, developers might be tempted to focus on the tradeoff between consistency and availability, but that distracts attention from the more common case: when then is no partition at all [21]. In fact, we expect partitions to become even more rare in the future Internet (e.g., due to redundant routing paths [60, 78, 80]). During normal operation, when there are no partitions, there is no need to sacrifice either consistency or availability (at least, not because of CAP), and a different tradeoff takes precedent: between consistency and performance.

This tradeoff arises naturally from the semantics themselves. Intuitively, it takes more work and more communication to ensure that a value being returned is the most recent than it would to, e.g., return a locally cached copy without checking it. For any distributed system, we can ask where on the spectrum the system lies: generally, the stonger the consistency it provides, the more performance will suffer. Conversely, increases in performance often come at the cost of consistency (e.g., however short the window might be when using eventual consistency [16], inconsistency is still present). Abadi formalized this idea with a classification system called PACELC [11] (pronounced "pass-elk,") which characterizes the behavior of distributed databases in the presence and absence of partitions. The name PACELC itself spells out the key rule: If there is a **P**artition, systems face a tradeoff between **A**vailability and **C**onsistency. **E**lse, when there is no partition, the system faces a tradeoff between **L**atency and **C**onsistency. A system favoring availability during a partition and consistency otherwise would be classified as a PA/EC system under PACELC. WACCO represents a PC/EC system, having prioritized consistency in both cases. But we believe that cluster consistency itself strikes a good balance between latency and consistency, as it weakens linearizability [40] somewhat in exchange for better performance. In contrast, DNS is a PA/EL system, because once a client reads a value for a domain name, it generally cannot read any future values until the TTL expires—whether there is a partition or not—due to caching along the way.

2.3 OTHER RELATED WORK

The use of a tree-based topology in WACCO for object access is reminiscent of hierarchical caching, which has been studied and deployed extensively for wide-area systems such as the World-Wide Web [25, 58, 67]. In some respects, WACCO can be viewed as using *polling-every-time* cache validation [23], in which the authoritative object copy is consulted before returning a cached answer in order to enforce strong consistency. To reduce the overheads and response latencies induced by such polling, WACCO employs two strategies. The first is to leverage the tree structure to aggregate polling by many concurrent reads into few messages along the tree. This aggregation also allows WACCO to reduce polling latency by using ongoing polling requests to accelerate others; this strategy has implications for the consistency offered by WACCO, which we characterize precisely in Chapter 4. The second strategy is to migrate the authoritative object copy closer to where demand is largest, an option available to WACCO because it manages the authoritative copy of each object itself, in contrast to web caches that do not. Resource migration has been studied for many years in order to improve performance in terms of, e.g., client-visible latency [72]. Various other systems use migration to manage the locations of distributed objects [64, 72], files [32, 38, 47], memory pages [18, 20, 69], and processes [38, 68, 71].

One design by Maggs et al. [54] is similar to WACCO in some respects: it can migrate copies of objects throughout a tree (among other topologies) in response to demand. Maggs et al. show that by placing objects carefully in the tree, the load on every edge is minimized, thus minimizing both the total load and congestion across all links and the average latency (number of hops to access an object) in the network. This design differs from WACCO in two main ways. First, it uses a cache invalidation strategy—there can be many cached copies of each object, any one of which can be used to answer a read and all of which must be updated during a write. Second, it does not address consistency to the same degree as our work, instead assuming data-race free runs (i.e., runs in which no two updates to the

same object are concurrent).

Our work is also related to prior research in "edge services" and wide-area storage that employs migration to place objects close to their demand. GlobeDB [70] is one such example that provides only weak object consistency, and Nomad [75] is an example for data accessed primarily by a single user. Quiver [66] uses migration together with object access protocols that, for single-object accesses, can be configured to provide either sequential consistency [49] or linearizability [40]. As we will discuss, sequential consistency will generally be too weak for a service such as LOKO that requires that updates be globally visible immediately, and while linearizability would be ideal, Quiver sacrifices significant scalability to achieve it (e.g., serving every read from the authoritative object copy). WACCO strikes a novel balance in offering a new type of consistency (stronger than sequential consistency, weaker than linearizability) in a still-scalable fashion. Nevertheless, our present WACCO implementation borrows basic migration and request routing algorithms from Quiver.

If a replication (or caching) scheme is to prevent conflicting object versions and to make updates available to reads immediately, it must apply reads and updates at a quorum of replicas that intersects the quorum used in another update [33, 39]. Different designs use different quorum systems; e.g., in read-one-update-all quorum systems, every proxy (the update quorum) must be contacted on the critical path of an update. WACCO uses a quorum per object consisting of a single authoritative copy, uses a tree-based overlay to reach this copy, is optimized toward widespread concurrent read load and moderate concurrent update load, and, to our knowledge, offers a new type of consistency achieved by a novel combination of tree-based aggregation and migration.

Our implementation of LOKO as a demonstration of WACCO (see Chapter 5) is motivated by shortcomings of the current DNS for future Internet architectures or even for serving more dynamic data in support of today's mobility and content management [62, 65]. These shortcomings have led to numerous attempts to modify DNS usage [79], to enhance DNS operation [26], to replace it outright with alternative designs [28, 44, 65], and to understand

the tradeoffs between new designs and the current DNS [63]. CoDoNS [65] is a noteworthy design that, like LOKO, decouples namespace (or keyspace) management from the location and ownership of name servers (in our parlance, proxies) and accelerates the propagation of updates to clients. It provides fast read response via a dynamic replication technique that ensures that a large percentage of requests can be answered immediately by the first proxy to receive the request. However, as in the discussion of quorum systems above, consistency then requires that all of these replicas be updated (or invalidated) when an update occurs, making updates more costly. LOKO is a different point in the design space that anticipates more frequent updates and so strikes a different balance between read and update cost—one that still favors reads particularly when read load is high but that lessens the number of proxies that updates must alter.

Chapter 3: PROTOCOL DESIGN

3.1 DESIGN CONSIDERATIONS AND GOALS

We anticipate an object access workload that is generally read-dominated—maybe by orders of magnitude—but that may nevertheless involve frequent and even concurrent updates per object. Updates to an object may be frequent due to the transient nature of the information used to update an object (e.g., the current performance characteristics of a network link), and object updates may be concurrent due to contributions from many parties (e.g., one per link, for an object that calculates preferred routes based on current characteristics of many links). Such workloads temper our willingness to trade update performance for read performance arbitrarily, e.g., as in a typical read-one-update-all system (see Chapter 2). Rather, WACCO takes a more balanced approach that favors read performance but that still limits updates to a single authoritative object copy.

The consistency implemented in WACCO implies sequential consistency [49] (and more, see below). Sequential consistency is a "strong" consistency model: it implies that clients observe update operations to objects in the same total order (cf., [14, Ch. 9]). Sequential consistency implies *causal consistency* [12]: updates related by potential causality [48] (e.g., a client reads an update and then performs another) will be observed by any client in order of their potential causality. But unlike causal consistency, sequential consistency also implies that all clients will observe all updates that are *not* related by potential causality in the same order.

Despite its strength, sequential consistency does not guarantee rapid propagation of updates: in the limit, a client of a sequentially consistent (only) object store is permitted to read the same value forever for an object, even if other clients update that object (as

in the example in Section 2.1.3). As such, our goal is to enforce rapid propagation of updates, i.e., updates "take effect" (nearly) immediately. Linearizability [40] strengthens sequential consistency by mandating that an update be observed by any operation on the same object that begins after (in real time) the update operation returns to its caller. However, linearizability comes at substantial performance cost [14, Ch. 9], and so we adopt a weaker requirement that nevertheless strengthens sequential consistency to make updates take effect quickly.

The middleground we adopt allows read operations on the same object to be partitioned into *clusters* of concurrent reads,[1] so that all reads in each cluster return results based on the latest update preceding the cluster in real time (or a more recent update, i.e., one concurrent with the cluster). The resulting consistency property, which we term *cluster consistency*, is weaker than linearizability in that a read returns results based only on updates that preceded the cluster containing it, rather than all updates that precede the individual read. (Updates to the same object are still ordered according to their real-time order, however.) In exchange for this weaker property, we show that cluster consistency can be implemented scalably in wide-area settings by permitting a read to carry responses to other reads in its cluster, thereby accelerating the response times of those reads and reducing load on the authoritative copy.

Beyond applications to future Internet designs (see Chapter 1), we also see cluster consistency as potentially useful in nearer-term applications of WACCO, e.g.:

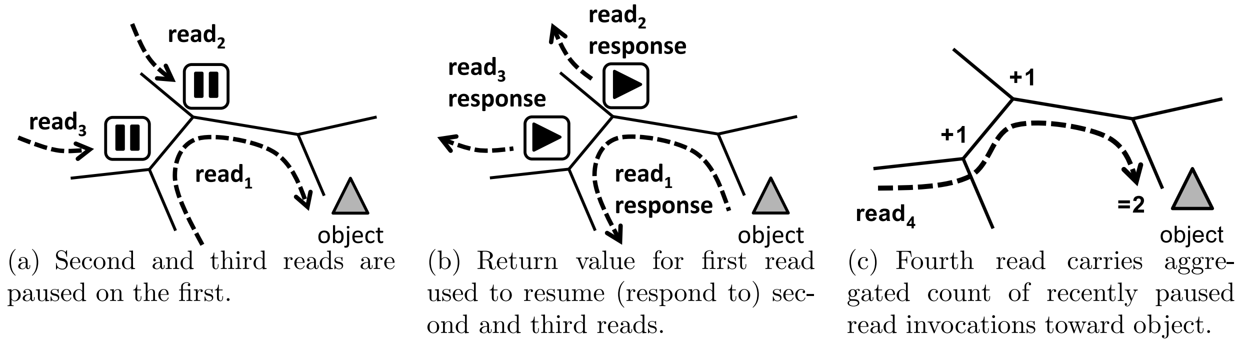
- **Network troubleshooting** Updates from network sensors that publish to WACCO will appear in the same order, enabling consistent diagnosis and actuation of the network by distributed analysis engines. For example, routing anomalies caused by MED oscillation [36] and BGP policy divergence [37] in today's Internet require distributed monitoring to quickly detect and react to an anomaly, e.g., by modifying local routing

[1]More specifically, in each cluster, the union of real-time intervals beginning with each read invocation and ending with its return, is contiguous. See Chapter 4 for details.

policies to eliminate the divergent behavior and so to minimize its impact on traffic. A cluster-consistent view of routing updates published to WACCO will make it simpler for distributed monitors to concur on the anomaly and effect changes in policy at multiple locations to rectify the problem. Another example is real-time response to routing pollution, e.g., prefix hijacking [84]. Rapid update propagation and consistent event ordering (e.g., which networks are polluted first) could help reveal the source of pollution and enable a faster reaction to the propagation of polluted routes.

- **Trajectory tracking of mobile nodes** Predicting the future location of a mobile endpoint (e.g., a train) for use in routing [61] would be greatly simplified with a cluster-consistent view of the endpoint's trajectory. For example, if each network appends its name to a WACCO object representing the endpoint's trajectory when the endpoint attaches to the network, cluster consistency implies that the trajectory will be accurate. A weaker property like causal consistency might yield incomplete and even conflicting trajectories, since appends would not be causally related (in the sense of Lamport [48]).
- **Online gaming applications** To keep online games fair to all players, it can be at least as important for users see the same content as it is that the content they see is the most up-to-date [29, 56]. Such applications can be simplified if built on objects that appear to all clients to be modified in the same order.

As suggested in Chapter 2 and detailed below, WACCO implements cluster consistency using a protocol in which each read cluster collectively polls an authoritative object copy before returning responses for the reads it contains. Prior work has generally found polling costlier than cache invalidation [23]. That said, polling serves dual purposes in WACCO; in addition to consistency, polling messages carry load information to the proxy holding the authoritative object copy, which it uses to determine if the object should be migrated. Migration enables an object to be placed closer to the predominant sources of demand and, as we will show, can significantly reduce response times for operations.



(a) Second and third reads are paused on the first.

(b) Return value for first read used to resume (respond to) second and third reads.

(c) Fourth read carries aggregated count of recently paused read invocations toward object.

Figure 3.1: Example of pausing some reads and resuming them later

3.2 WACCO DESIGN

The object-sharing protocol that underlies wacco uses a logically tree-structured overlay network that spans a collection of *proxies*. This overlay network should be assembled in a "geographically aware" manner, i.e., so that geographically close (and so presumably well-connected) proxies are also close to one another in the tree. The manner in which a client is paired with a proxy can be decoupled from the rest of our system design; our present design simply leverages a few widely-known proxies to refer each new client to a proxy near it. We assume that each client interacts with only a single proxy at a time, awaiting the completion of any operations it issued to one proxy before switching to another (if it switches at all).

The proxies provide clients with access to a set of objects. A client sends a read or update invocation for an object to its proxy and awaits a response from that same proxy. Updates (potentially) modify the object state; reads do not. Our protocol description and proof presume that a read simply returns the current object state, though a proxy can instead return to the client a customized result derived from that state. Section 5.1 gives an example of this behavior in the context of loko.

3.2.1 BASIC PROTOCOL

WACCO maintains a single authoritative copy of each object. At any point in time, the proxy at which this copy of the object resides is said to *host* the object and, synonymously, to be the *location* of the object. Proxies implement a protocol (based on Quiver [66]) to route client invocations toward the current location of the object over tree edges. Once performed on the object, an operation's response is routed back over the tree to the client that invoked it.

While all update invocations are always routed to the object itself, a read invocation will be *paused* in the tree if the invocation, while en route to the object, encounters a proxy that already forwarded a read request for the same object and has not yet received a response. The paused read will not be forwarded further in the tree; rather, it will be held by the proxy until the response to the invocation on which it paused is returned. When that response arrives, it can serve as the response for any read invocation on the same object that was paused awaiting it and that meets certain conditions described below. In this way, a single read invocation that reaches the object may, in fact, end up serving numerous read requests that are paused on it elsewhere in the tree. This effect is shown in Figure 3.1, where the second and third reads are paused waiting on the first (Figure 3.1(a)) and then adopt the response to the first read as their own (Figure 3.1(b)).

Pausing read requests in this way offers at least two benefits. First, it reduces overall latency in comparison to forwarding each request all the way to the object, since the read request on which another is paused is farther along the path to the object (and so should solicit a response sooner) than the paused read is. That is, in Figure 3.1(a), the first read is at least as close to the object as the second or third read is when each is paused, and a response may even already be traversing the path back. Second, in comparison to forwarding every read request to the object and returning each read response individually, pausing reduces bandwidth use, routing costs to proxies, and computational load on the

proxy hosting the object.

Pausing also presents some challenges. First, a paused read constitutes state that a proxy must store until the response for the read on which it is paused returns, possibly opening the door to resource exhaustion. That said, aside from read invocations submitted to a proxy directly by clients, the number of paused reads for an object that a proxy must maintain simultaneously is limited by the number of its neighbors. Reads submitted to a proxy directly by clients (and that are paused) still pose a denial-of-service risk, but it can be managed using any of several techniques [42], and moreover, dropping these read requests as needed can never interfere with other reads (since none are paused on these reads). Resource exhaustion will be discussed further in Section 3.2.4.

Second, pausing erodes the consistency of the protocol, and, indeed, to achieve cluster consistency—and specifically to achieve the sequential consistency that it implies—we must restrict which read responses can be used to respond to paused reads. Intuitively, implementing cluster consistency requires that a paused read is not answered by an incoming response that is too outdated. Specifically, as we prove in Section 4.2, the following conditions suffice to implement cluster consistency: Each read request from a client carries the largest Lamport time [48] (see Section 4.2 for a description of Lamport timestamps) at which any update that the client has observed was applied, and each read response carries the Lamport time at which the response was emitted from the authoritative object. A read response that returns to a proxy can be used to satisfy a read request paused at that proxy only if the response's timestamp exceeds the request's timestamp. If this requirement leaves any reads paused at the proxy unsatisfied, then the proxy unpauses one and forwards it along toward the object.

Finally, the potential for read pausing is the only reason that WACCO must send back the entire object as part of a response (though caching can obviate that need, see below) and why the response must travel back through the tree toward the originating proxy instead of going directly to it. If the objects themselves are too large, attempting to send

them back in their entirety in responses could be detrimental to performance. Therefore, having objects that are relatively easy to send across the wire is an important assumption of the WACCO design.

3.2.2 CACHING

Each object state has a *version number* (an integer, initially zero). Applying an update to the object increments that version number. WACCO uses these version numbers to optimize the protocol above as follows.

Each proxy maintains a cache holding at most one cached state per object. The proxy is free to delete states from this cache and manage it using policies independent of those of other proxies. Each read request is augmented to carry a version number. If upon receiving a read request with version number v (new read requests submitted by clients have $v = -1$), a proxy has a version $v' > v$ of the relevant object in cache, then the proxy can increase the read request's version number to v' when forwarding it. If it does so, the proxy is said to have *taken responsibility* for the request and is obligated to retain the cached object state until it has responded to this request. (Our current proxy implementation defaults to taking responsibility; others could do so more selectively.)

When responding to a read request, the proxy hosting the authoritative copy sends the object state (as in Section 3.2.1) if the current object version is larger than the version number in the read request, and sends `same` otherwise. On receiving a response to a read for which a proxy took responsibility, the proxy identifies the latest object version it now has—either the object state in the response or, if the response was `same`, the version in its cache —and responds to paused reads similarly (subject also to the constraints of Section 3.2.1 on Lamport timestamps). That is, it returns `same` to paused reads bearing the version number of the proxy's latest object version, and it responds with the latest object state to the rest.

A proxy that forwards a read request but that does not take responsibility for it might

receive a `same` response, at which point it may not have the latest object version and so would be unable to respond to any read it paused bearing an older object version number. These paused reads therefore remain paused while one is unpaused and forwarded toward the authoritative object, as discussed in Section 3.2.1. Note that forwarding any read request bearing an old object version number guarantees that the response will contain an object state, and so when the proxy selects one to unpause and forward, it gives preference to those with smaller object version numbers.

3.2.3 MIGRATION

WACCO responds to demand by strategically moving objects among proxies, a process called migration. For example, a proxy may migrate an object to a neighbor that is forwarding a majority of the invocations for that object, or a proxy that is becoming too heavily loaded may choose to migrate objects away. In this way, migration can be used to reduce load by moving objects closer to areas of greater interest and to otherwise reposition load as needed to deal with hotspots. The former use of migration is particularly beneficial for LOKO (see Chapter 5), since migration can be used to position objects to best accommodate the time zones that are most active at a particular time of day. Moreover, many entities will be accessed with a clear geographic preference—e.g., websites in Chinese will presumably be accessed most often from China—and so migration makes sense for positioning such an object near where it is accessed most.

WACCO is not closely tied to the mechanics of migration; it requires only the ability to migrate an object from a proxy to its neighbor between invocations. So while WACCO uses the migration mechanism in Quiver [66], other migration mechanisms would also work. That said, effective migration requires us to resolve two issues. First, we must determine from where an object is currently experiencing the most load; because of pausing reads, no single proxy observes the entire load on an object. Then, we need to determine the specific conditions under which an object should be migrated.

The first question is resolved in WACCO by amending each message carrying an object invocation to also include the number of read invocations for that same object that were *recently* paused along the path the message has traveled. If this invocation is paused, the proxy that does so accumulates the message's count into a per-object, per-neighbor counter that the proxy maintains (i.e., for the object to which the invocation pertains and the neighbor from which the proxy received the invocation) and then further increments this counter by one (for the invocation that was just paused). Otherwise, the proxy accumulates its counters for this object and for *all* of its neighbors into the field on the invocation message and forwards the message along toward the object, subsequently setting each of these counters to zero. Figure 3.1(c) shows an example where the field of a fourth read invocation, initially with value `0`, is updated to `1` at the proxy where $\texttt{read}_3$ was formerly paused and then to `2` as it travels through the proxy at which $\texttt{read}_2$ was formerly paused. In this way, a count of paused reads trickles toward the object at all times, which the proxy holding the object can similarly incorporate into per-object, per-neighbor counts of paused invocations.[2]

As described so far, this approach for conveying the numbers of paused reads to the proxy holding the object does not adjust these counters for the passage of time, but intuitively such adjustment is necessary. After all, reads paused ten minutes ago should presumably have less bearing on whether to migrate the object than reads paused within the last few seconds. For this reason, each WACCO proxy *decays* its per-object, per-neighbor counters to account for the passage of time before incorporating them into invocation messages bound for the object (or, in the case of the proxy hosting the object, before calculating whether to migrate an object). In our present implementation, the proxy decays these counters linearly as a function of the time that passed since last unpausing (and returning values for) reads for that object, i.e., the interval between the proxy seeing the last object

[2]Though an update invocation cannot be paused, the proxy holding the object incorporates each update invocation into this count, as well, so that updates too are reflected in the load calculations.

response and the subsequent object invocation.

Finally, this brings us to the question of how a proxy holding an object determines whether to migrate an object and if so, to which of its neighbors. In our implementation, we test two different migration strategies. In the first (used in Chapter 5 and Chapter 6), the proxy hosting an object periodically sums its per-neighbor counters for that object and, if one such counter accounts for more than a fraction m of this sum (for a fixed threshold m), then the proxy asks the neighboring proxy corresponding to that counter to migrate the object to it. That neighboring proxy might not do so, e.g., because it is already hosting too many other objects. If it decides to do so, however, then it initiates the object migration. Note that the threshold m value can be different per object, though in our experiments we simply set m the same for all objects.

The second migration strategy, detailed in Chapter 7, also takes into account the number of requests for an object coming from (the direction of) each neighbor, but it can also consider other factors: e.g., hosting capacities, traffic costs, forbidden object locations. This second strategy gives WACCO much more flexibility and its users more ability to configure its behavior to fit their specific environments.

3.2.4 RESILIENCE

Fault tolerance In WACCO as described so far, a proxy failure would disconnect the tree until the proxy recovers. A generic approach to tolerate proxy failure is to locally replicate each proxy; e.g., in our implementation, each proxy can optionally have a backup to which it commits any meaningful change in internal state [22, §8.2.1] before acting on it. In WACCO, such changes include changes to an object (due to update operations) and changes to internal Quiver routing tables [66] (e.g., due to migration). In a straightforward implementation, this primary-backup configuration would double the hardware needed for the service. In practice, we expect clusters of proxies to reside in datacenters in major metropolitan areas, in which case these proxies can provide backup service for others in

the same datacenter.

Denial-of-service defense The most acute threat of denial-of-service attacks is interfering with proxy-to-proxy communication. Multi-path routing [60, 78, 80], using private leased lines, or other suitable defenses [82] can mitigate the threat of link overload. In addition, each proxy should ensure that it reserves adequate resources to retain communication with its neighbor proxies. For example, each proxy can use two network interfaces, one dedicated to proxy-to-proxy communication and the other for serving clients that contact it directly. Moreover, proxies can prioritize tasks for managing inter-proxy activities ahead of those responding to clients and can terminate (or refuse) client requests in favor of retaining communication with neighbor proxies.

Migration opens the possibility of a degradation of service if, e.g., a flood of read requests can cause an object to be migrated (see Section 3.2.3) to a region of the network far from legitimate demand. This risk can be mitigated by each object expressing to WACCO its preferences or requirements for where it can be hosted, if the region of legitimate demand is known in advance. This mechanism is also useful to enforce regulatory constraints on where data can be hosted, for example. The migration strategy described in Chapter 7 enables these kinds of placement restrictions. In other cases, allowing only *authorized* reads to influence migration can mitigate this risk. One method for doing this is described in Section 5.1 in the context of LOKO.

Chapter 4: CLUSTER CONSISTENCY

Here we define cluster consistency precisely. We then show in Section 4.2 that our protocol implements it faithfully.

4.1 DEFINITION

An *object* consists of state and a set of methods that can be *invoked*. Each invocation returns a *response*, and an invocation/response pair is called an *operation*. Correct behavior of the object is defined by its *sequential specification*, which specifies the return results of operations invoked sequentially on the object.

We use op to denote any operation, and $r\text{-}op$ or $u\text{-}op$ denote a read or update operation, respectively. The invocation and response for any op occur at distinct real times $op.\mathsf{inv}$ and $op.\mathsf{res}$, respectively, with $op.\mathsf{inv} < op.\mathsf{res}$ and $[op.\mathsf{inv}, op.\mathsf{res}]$ denoted as $op.\mathsf{interval}$. A *history* H is a set of operations and an induced partial order $\prec_H$ defined as $op_1 \prec_H op_2 \iff op_1.\mathsf{res} < op_2.\mathsf{inv}$. If $\prec_H$ is a total order, H is *sequential*. For an object obj, the set $H|obj$ includes only those operations in H that are invoked on obj, and for a client c, the set $H|c$ includes only those operations in H that are invoked by c. By convention, we assume that $H|c$ is sequential for each client c. (In practice, each "client" is a client *thread*.) A *serialization* S of H is the set H totally ordered by a relation $\prec_S$.

Definition 1 (Sequential consistency [49])**.** *A history H is sequentially consistent if there exists a serialization S of H such that the following properties hold: (i)* Legality: *For each object obj, $S|obj$ is legal (i.e., is in the sequential specification of obj). (ii)* Local-Order: *If op_1 and op_2 are executed by the same client and $op_1 \prec_H op_2$, then $op_1 \prec_S op_2$.*

The consistency implemented in WACCO, called *cluster consistency*, implies sequential consistency. As such, there is a well-defined order in which updates are applied to each

object, and each update operation produces a new version of the object on which it operates. The version number of the new object instance is one greater than that of the object instance to which the update was applied. Let $u\text{-}op.\mathsf{ver}$ be the version number of the object instance produced by $u\text{-}op$.

Definition 2 (Read cluster). *A read cluster C is a nonempty set of read operations (i) that return the same object version, and (ii) for which $\bigcup_{op \in C} op.\mathsf{interval}$ is a contiguous interval of time. For a read cluster C, we define $C.\mathsf{inv} = \min_{op \in C} op.\mathsf{inv}$ and $C.\mathsf{res} = \min_{op \in C} op.\mathsf{res}$. Let $C.\mathsf{ver}$ be the version of the object when it was read by C.*

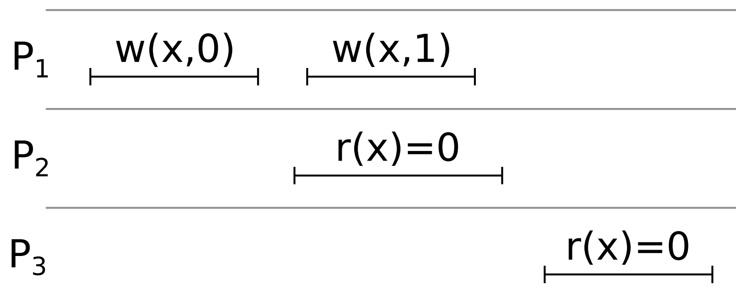
We also represent each $u\text{-}op$ as its own *update cluster* $C = \{u\text{-}op\}$, with $C.\mathsf{inv} = u\text{-}op.\mathsf{inv}$, $C.\mathsf{res} = u\text{-}op.\mathsf{res}$, and $C.\mathsf{ver} = u\text{-}op.\mathsf{ver}$. We then use $C_1 \prec_H C_2$ (where C_1 and C_2 are read or update clusters) to mean $C_1.\mathsf{res} < C_2.\mathsf{inv}$.

Definition 3 (Cluster consistency). *A set of operations is cluster-consistent if it is sequentially consistent and satisfies* Cluster-Order*: There exists a partition of the operations into clusters so that if C_1, C_2 are performed on the same object and $C_1.\mathsf{res} < C_2.\mathsf{inv}$, then $C_1.\mathsf{ver} \leq C_2.\mathsf{ver}$.*

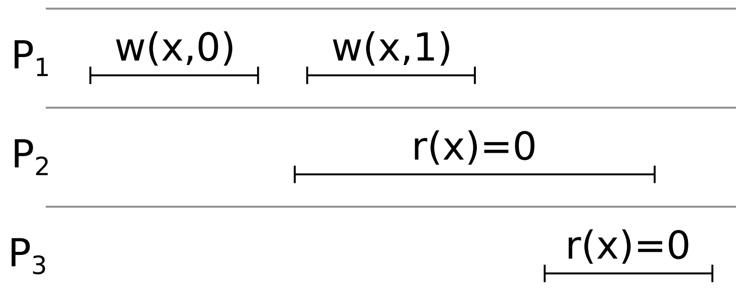
Figure 4.1(a) shows an execution that is sequentially consistent but not cluster-consistent, and so cluster consistency is strictly stronger. However, cluster consistency is weaker than linearizability [40], which requires that for any op_1 and op_2, if $op_1.\mathsf{res} < op_2.\mathsf{inv}$ then $op_1.\mathsf{ver} \leq op_2.\mathsf{ver}$; i.e., history precedence must hold at the operation level, not only the cluster level. Figure 4.1(b) shows a cluster-consistent execution that is not linearizable.

4.2 PROOF OF CLUSTER CONSISTENCY

We now prove that the protocol described in Section 3.2.1 implements cluster consistency, ignoring caching (Section 3.2.2), migration (Section 3.2.3), and proxy backups (Section 3.2.4), as these do not alter the semantics of the protocol. Given a history H, consider a directed graph $\mathcal{G}_H$ with operations in H as vertices and three types of edges:



(a) A sequentially consistent history. To be cluster-consistent, the read at P_3 must return 1 since its read cluster (itself only) occurs after $\mathsf{w}(x, 1)$.



(b) A cluster-consistent history (since the read operations form a cluster). To be linearizable, the read at P_3 must return 1 since it occurs after $\mathsf{w}(x, 1)$.

Figure 4.1: Execution histories for a single object. Time increases left-to-right. Each row denotes one client.

- **Client order** ($\xrightarrow{\text{c}}$): if op_1 and op_2 are performed by the same client and if $op_1 \prec_H op_2$, then $op_1 \xrightarrow{\text{c}} op_2$.
- **Reads-from order** ($\xrightarrow{\text{rf}}$): if $u\text{-}op$ results in an object state on which op is applied, then $u\text{-}op \xrightarrow{\text{rf}} op$.
- **Version order** ($\xrightarrow{\text{v}}$): Let $u\text{-}op_1$ and $u\text{-}op_2$ denote distinct update operations on the same object, and let op denote any other operation on that object such that $u\text{-}op_1 \xrightarrow{\text{rf}} op$. If $u\text{-}op_1.\mathsf{ver} > u\text{-}op_2.\mathsf{ver}$, then $u\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_1$ and otherwise $op \xrightarrow{\text{v}} u\text{-}op_2$.

We use natural shorthands such as $\xrightarrow{\text{c,rf}} \; = \; \xrightarrow{\text{c}} \cup \xrightarrow{\text{rf}}$. We also use $\xrightarrow{\text{c}}_+$ to denote the irreflexive transitive closure of $\xrightarrow{\text{c}}$, and similarly for other orders.

$\xrightarrow{\text{c}}$ and $\xrightarrow{\text{rf}}$ naturally capture the temporal and data-flow relationships relevant when serializing H, whereas $\xrightarrow{\text{v}}$ constrains any serialization to respect the object versions observed by operations. To prove the sequential consistency of H, we first show that $\mathcal{G}_H$ is acyclic (Lemma 6) and then that this implies that there is a serialization of H respecting *Legality* and *Local-Order* (Corollary 1). We then argue in Lemma 7 that there must be such a serialization that also satisfies *Cluster-Order*.

Our algorithm relies on Lamport timestamps [48], and so we will briefly describe them here. Lamport time is an algorithm for synchronizing logical clocks across processes in a distributed system. Each process maintains its own clock, which can simply be an integer. Before each event that occurs within a process (e.g., updating some state or sending a message), that process increments its clock, giving a new timestamp that is assigned to that event. Each message sent to another process includes the sender's current clock value (for the send event), and the recipient of the message must, on receipt of the message and before acting on it, set its own clock value to be greater than both the sender's clock value and its own clock value. The clock values induce a partial order on the events in the distributed system. Lamport showed that if event_a causally precedes event_b, then the Lamport time for event_a must be less than the time for event_b. Here, causal precedence is the irreflexive, transitive closure of the relation consisting of client order ($\xrightarrow{\text{c}}$) and the

send/receive event pair for each message.

Below we prove several lemmas which we then use to prove that $\mathcal{G}_H$ is acyclic. Our proofs below involve the following additional notation. To each operation *op* is associated a logical (Lamport) time $op.\mathsf{linv}$ at which the client invoked it and another logical time $op.\mathsf{lres}$ at which it returned its result at that client. In addition, each *u-op* has a *(logical) effective time* of $u\text{-}op.\mathsf{leff}$, which is the Lamport clock value assigned to the event applying *u-op* to the object at the proxy hosting the object. For reads, $r\text{-}op.\mathsf{leff}$ is the logical time at which a response for *r-op* was issued, either by the last proxy to pause *r-op* or (if *r-op* reached it) by the proxy hosting the object. N.B., $op.\mathsf{linv} < op.\mathsf{leff} < op.\mathsf{lres}$ for all operations.

Lemma 1. *The subgraph of $\mathcal{G}_H$ consisting of only edges in $\xrightarrow{c,rf}$ is acyclic.*

Proof. Since $op_1 \xrightarrow{\text{c}} op_2$ implies $op_1.\mathsf{lres} < op_2.\mathsf{linv}$, it also implies $op_1.\mathsf{leff} < op_2.\mathsf{leff}$. Similarly, it must be that $op_1 \xrightarrow{\text{rf}} op_2$ implies $op_1.\mathsf{leff} < op_2.\mathsf{leff}$, since an update must have been written before it can be read from. Therefore, each edge in $\xrightarrow{\text{c,rf}}$ represents an increase in $op.\mathsf{leff}$, meaning $op_1 \xrightarrow{\text{c,rf}}_+ op_2$ implies $op_1.\mathsf{leff} < op_2.\mathsf{leff}$.

If there is a cycle consisting only of edges in $\xrightarrow{\text{c,rf}}$, then we have $op \xrightarrow{\text{c,rf}}_+ op$, implying $op.\mathsf{leff} < op.\mathsf{leff}$, a contradiction. $\square$

Lemma 2. *If there is a cycle in $\mathcal{G}_H$, then there is a cycle in $\mathcal{G}_H$ in which every $\xrightarrow{v}$ edge appears in an edge sequence of the form $u\text{-}op_1 \xrightarrow{c,rf}_+ r\text{-}op_2 \xrightarrow{v} u\text{-}op_3$.*

Proof. We prove the result by first showing that for any cycle in $\mathcal{G}_H$, any $\xrightarrow{\text{v}}$ edge not already in an edge sequence of the form $op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$ can be replaced by edges not in $\xrightarrow{\text{v}}$ to produce a new cycle in $\mathcal{G}_H$. Since $\xrightarrow{\text{v}}$ edges must point to an update, we must consider $\xrightarrow{\text{v}}$ edges of only the forms $r\text{-}op \xrightarrow{\text{v}} u\text{-}op$ and $u\text{-}op' \xrightarrow{\text{v}} u\text{-}op$. In the first case, since $op \xrightarrow{\text{v}} r\text{-}op$ is impossible (again, $\xrightarrow{\text{v}}$ edges point to updates), an edge $r\text{-}op \xrightarrow{\text{v}} u\text{-}op$ already occurs within an edge sequence of the form $op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$ on the cycle. In the second case, because updates on each

object are applied sequentially, $u\text{-}op'$ is applied before $u\text{-}op$, and so there is a chain of updates to the object such that $u\text{-}op' \xrightarrow{\text{rf}}_+ u\text{-}op$. Replacing the edge $u\text{-}op' \xrightarrow{\text{v}} u\text{-}op$ with this chain produces a cycle not containing $u\text{-}op' \xrightarrow{\text{v}} u\text{-}op$.

To complete the proof, we now must argue that for any edge sequence of the form $op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$ on the cycle, there is a corresponding edge sequence $u\text{-}op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}$ on the cycle. If op_1 is an update, then setting $u\text{-}op_1 = op_1$ completes the argument. Otherwise, consider walking the cycle backward along $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{c}}$ edges from op_1, terminating at a $\xrightarrow{\text{v}}$ edge. Since this $\xrightarrow{\text{v}}$ edge must point to an update, this update suffices for $u\text{-}op_1$. □

If there is a cycle in $\mathcal{G}_H$, then Lemma 2 guarantees the existence of a cycle in which all $\xrightarrow{\text{v}}$ edges occur within edge sequences of a certain form. Below we refer to such a cycle as *constrained*.

Lemma 3. *If there is a cycle in $\mathcal{G}_H$, then within a constrained cycle, at least one edge sequence $u\text{-}op_1 \xrightarrow{c,rf}_+ r\text{-}op_2 \xrightarrow{v} u\text{-}op_3$ has $u\text{-}op_3.\mathsf{leff} \leq u\text{-}op_1.\mathsf{leff}$.*

Proof. Consider an alternative graph $\mathcal{G}'_H$ that includes all of the edges of $\mathcal{G}_H$ and additionally the edge $u\text{-}op_1 \xrightarrow{\text{s}} u\text{-}op_3$ whenever $u\text{-}op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$. From any constrained cycle in $\mathcal{G}_H$ we can construct a cycle $op \xrightarrow{\text{c,rf,s}}_+ op$ in $\mathcal{G}'_H$ by replacing edge sequences $u\text{-}op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$ on the constrained cycle with the edge $u\text{-}op_1 \xrightarrow{\text{s}} u\text{-}op_3$. Recall from the proof of Lemma 1 that $op' \xrightarrow{\text{c,rf}}_+ op$ implies $op'.\mathsf{leff} < op.\mathsf{leff}$. Moreover, if Lemma 3 were false, then $u\text{-}op_1.\mathsf{leff} < u\text{-}op_3.\mathsf{leff}$ for every edge $u\text{-}op_1 \xrightarrow{\text{s}} u\text{-}op_3$ used in the cycle in $\mathcal{G}'_H$. So, from the cycle $op \xrightarrow{\text{c,rf,s}}_+ op$ we could infer $op.\mathsf{leff} < op.\mathsf{leff}$, a contradiction. □

Each read cluster C has exactly one read operation that reads from the authoritative object itself. We call this "representative" read operation $C.\mathsf{rep}$.

Lemma 4. *If there is an edge sequence $u\text{-}op' \xrightarrow{c,rf}_+ r\text{-}op \xrightarrow{v} u\text{-}op$ in $\mathcal{G}_H$ where $u\text{-}op.\mathsf{leff} \leq u\text{-}op'.\mathsf{leff}$, then $r\text{-}op'.\mathsf{leff} \leq u\text{-}op'.\mathsf{leff}$ where C is the read cluster containing*

r-op and r-$op' = C.\mathsf{rep}$.

Proof. If u-$op'.\mathsf{leff} < r$-$op'.\mathsf{leff}$, we have u-$op.\mathsf{leff} \leq u$-$op'.\mathsf{leff} < r$-$op'.\mathsf{leff}$. Since r-op' read from the object itself and u-$op.\mathsf{leff} < r$-$op'.\mathsf{leff}$, r-op' (and thus r-op) must have read from u-op (or a later update), giving u-$op \xrightarrow{\text{rf}}_+ r$-$op$, contradicting r-$op \xrightarrow{\text{v}} u$-$op$. □

Lemmas 1–4 show that for $\mathcal{G}_H$ to have a cycle, a necessary condition is an edge sequence of the form u-$op' \xrightarrow{\text{c,rf}}_+ r$-$op \xrightarrow{\text{v}} u$-$op$ where r-op is contained in a read cluster C whose representative r-$op' = C.\mathsf{rep}$ is too outdated, i.e., r-$op'.\mathsf{leff} \leq u$-$op'.\mathsf{leff}$. Therefore, WACCO is designed to prevent this possibility, viz., by building clusters conscientiously. Specifically, each returning response to a read operation r-op carries with it the effective time of representative r-op' of the cluster containing r-op and the effective time of the update from which r-op' and thus r-op are reading, called r-$op.\mathsf{lueff}$. That is, if u-$op \xrightarrow{\text{rf}} r$-$op$, then r-$op.\mathsf{lueff} = u$-$op.\mathsf{leff}$. Responses to updates can also carry the effective time back to the requester, so that u-$op.\mathsf{lueff} = u$-$op.\mathsf{leff}$.

Each client c tracks the largest $op.\mathsf{lueff}$ for all operations op it has issued, denoted $c.\mathsf{after}$; i.e., $c.\mathsf{after} = \max_{op}\{op.\mathsf{lueff}\}$ where the maximum is taken over all operations issued by c. The outbound request for each r-op carries with it the current value of $c.\mathsf{after}$, called r-$op.\mathsf{after}$. When a read response arrives at a proxy, it carries with it the effective time of the read operation r-op' that reached the authoritative object to elicit that response. The proxy will use this read response to answer a paused read operation r-op only if r-$op'.\mathsf{leff} > r$-$op.\mathsf{after}$; in this case, r-op is added to the cluster for which r-op' serves as the representative and so r-$op.\mathsf{lueff}$ is set to r-$op'.\mathsf{lueff}$. Any reads r-op'' that were not answered by r-op' (i.e., because r-$op'.\mathsf{leff} \leq r$-$op''.\mathsf{after}$) must still be addressed, and now no response is expected inbound. Therefore, the proxy chooses any remaining r-op'' to forward along to elicit another response.

Lemma 5. *There is no edge sequence u-$op' \xrightarrow{c,rf}_+ r$-$op \xrightarrow{v} u$-$op$ in $\mathcal{G}_H$ such that u-$op.\mathsf{leff} \leq u$-$op'.\mathsf{leff}$.*

Proof. By Lemma 4, the existence of edge sequence $u\text{-}op' \xrightarrow{\text{c,rf}}_+ r\text{-}op \xrightarrow{\text{v}} u\text{-}op$ in $\mathcal{G}_H$ such that $u\text{-}op.\mathsf{leff} \le u\text{-}op'.\mathsf{leff}$ implies that $r\text{-}op'.\mathsf{leff} \le u\text{-}op'.\mathsf{leff}$ where C is the read cluster containing $r\text{-}op$ and $r\text{-}op' = C.\mathsf{rep}$. By construction, $r\text{-}op$ can be answered by a read response only if the effective time of the read operation $r\text{-}op'$ that reached the authoritative object to elicit the response satisfies $r\text{-}op'.\mathsf{leff} > r\text{-}op.\mathsf{after}$. So, to prove the lemma, it suffices to show that $u\text{-}op'.\mathsf{leff} \le r\text{-}op.\mathsf{after}$.

Given the edge sequence $u\text{-}op' \xrightarrow{\text{c,rf}}_+ r\text{-}op \xrightarrow{\text{v}} u\text{-}op$, let $u\text{-}op''$ be the update operation on this sequence that precedes and is closest to $r\text{-}op$; i.e., there is no update operation between $u\text{-}op''$ and $r\text{-}op$ along this edge sequence. Let c be the client that issued $r\text{-}op$. Either $u\text{-}op'' = u\text{-}op'$ and so $u\text{-}op'.\mathsf{leff} = u\text{-}op''.\mathsf{leff}$, or $u\text{-}op' \xrightarrow{\text{c,rf}}_+ u\text{-}op''$ and so $u\text{-}op'.\mathsf{leff} < u\text{-}op''.\mathsf{leff}$. It thus suffices to prove that $u\text{-}op''.\mathsf{leff} \le r\text{-}op.\mathsf{after}$. If the chain $u\text{-}op'' \xrightarrow{\text{c,rf}}_+ r\text{-}op$ includes no $\xrightarrow{\text{rf}}$ edges, then c also issued $u\text{-}op''$, and so $u\text{-}op''.\mathsf{leff} = u\text{-}op''.\mathsf{lueff} \le r\text{-}op.\mathsf{after}$ because $r\text{-}op.\mathsf{after}$ is defined as the maximum $op.\mathsf{lueff}$ for all operations op that c has issued so far (including $u\text{-}op''$ itself). If the chain $u\text{-}op'' \xrightarrow{\text{c,rf}}_+ r\text{-}op$ includes one $\xrightarrow{\text{rf}}$ edge, it must be the first edge, giving $u\text{-}op'' \xrightarrow{\text{rf}} r\text{-}op'' \xrightarrow{\text{c}}_+ r\text{-}op \xrightarrow{\text{v}} u\text{-}op$. Then, c also issued $r\text{-}op''$, and so $u\text{-}op''.\mathsf{leff} = r\text{-}op''.\mathsf{lueff} \le r\text{-}op.\mathsf{after}$, again due to the construction of $r\text{-}op.\mathsf{after}$. □

Lemma 6. *$\mathcal{G}_H$ is acyclic.*

Proof. If there is a cycle in $\mathcal{G}_H$, a sequence $u\text{-}op_1 \xrightarrow{\text{c,rf}}_+ r\text{-}op_2 \xrightarrow{\text{v}} u\text{-}op_3$ such that $u\text{-}op_3.\mathsf{leff} \le u\text{-}op_1.\mathsf{leff}$ must appear in the cycle (by Lemma 3), which is not possible (by Lemma 5), giving a contradiction. □

Corollary 1. *The protocol of Section 3.2.1 is sequentially consistent.*

Proof. Consider any topological sort of $\mathcal{G}_H$. Due to the $\xrightarrow{\text{c}}$ edges, it satisfies *Local-Order*. Moreover, every read and update operation appears in this serialization after the update producing the object state to which it is applied (due to $\xrightarrow{\text{rf}}$ edges) and before any subsequent update (due to $\xrightarrow{\text{v}}$ edges). Consequently, *Legality* is satisfied. □

Lemma 7. *The protocol of Section 3.2.1 satisfies Cluster-Order.*

Proof. Consider two clusters $C_1, C_2 \subseteq H|obj$ as defined above, such that $C_1 \prec_H C_2$. Therefore, $C_1.\mathsf{rep}$ was applied to the authoritative object before $C_2.\mathsf{rep}$ (in real time), and so $C_1.\mathsf{ver} \leq C_2.\mathsf{ver}$. □

Chapter 5: CASE STUDY: DNS TRACES

We have implemented WACCO in Java. Our implementation consists of roughly 17,000 physical source lines of code. To evaluate WACCO, we used it to construct a service called LOKO, which we describe in Section 5.1. We evaluate LOKO in two distinct case studies. This chapter presents the first case study. The second appears in Chapter 6.

5.1 LOKO

As discussed in Chapter 1, we have used WACCO to implement a service called LOKO that hosts *keyspace* objects. A keyspace is identified by a public key pk and stores (or generates) mappings, each from a query string $qstr$ to a value val. When responding to a query, the keyspace sends the mapping $qstr \rightarrow val$, digitally signed so that it can be verified by pk. The signature could be inserted into the keyspace through an update invocation, or the keyspace could produce the signature itself using a private key it holds. The latter strategy might be appropriate for keyspaces that generate responses dynamically.

Generating dynamic responses is useful, e.g., to support CDNs by customizing the content-server address returned in response to a read query. That is, a keyspace for pk, when queried for `nytimes/www/address`, could select the answer val from a set of candidate addresses based on load conditions and the address of the client. (This selection would be performed by the proxy directly returning the response to the client.) The cluster consistency offered by LOKO would improve the responsiveness of this mapping to changing conditions over that provided by DNS today (cf., [62]). Of course, keyspaces can also be used to store static mappings, e.g., to addresses or public keys, and keyspaces can be queried iteratively to resolve hierarchical names, analogous to DNS/DNSSEC today. That is, like DNSSEC, the result of every individual query is independently verifiable by the requesting

client. It is umambiguous which public key should be used to verify the response, as the namespace itself is referred to only by its public key. Any query that requires a chain of sequential queries to fully resolve can maintain the same property, if properly structured. For example, the result of the first query could be a mapping whose value includes a second keyspace's public key and a key to query within that keyspace. This second query (and any following queries) can similarly be checked. In this way, every client can trust every response served by LOKO.

Any LOKO object could enforce its own access control by checking a signature for each invocation—possibly the same one that it will store and return in response to read invocations later. But by virtue of it having a public key, a keyspace enables the enforcement of coarse access-control policy at the first proxy to receive a request for it, even if that proxy does not host the object. That is, we could extend LOKO so that a proxy, upon receiving a read request for the keyspace identified by *pk* from a client, confirms that the request is accompanied by a delegation credential signed by the owner of *pk* and that authorizes the read. The proxy would do so prior to acting on the read request, dropping it if the check fails. This defense would provide a first line of defense, e.g., for sensitive corporate data stored within LOKO, by preventing requests made by outsiders from succeeding, and it would also hinder attempts to migrate the keyspace away from legitimate demand by submitting unauthorized read requests in order to degrade service (see Section 3.2.4). We have not implemented this extension, however.

5.2 TRACES

The data we use in this case study to evaluate LOKO (and hence WACCO) are traces of DNS queries received by Akamai Technologies, Inc., collected from 6am, March 9, 2011 to 6pm, March 10, 2011 (36 hours) and anonymized by Akamai before disclosure to us. In addition to serving DNS queries for domain names of its own, Akamai also serves queries for the domain names of a number of customers. The data set includes queries of both

types and reportedly includes all queries Akamai received during that period by 357 of these (globally distributed) servers.

We emphasize that the goal of using Akamai data was *not* to evaluate LOKO as a DNS replacement per se but rather to obtain for our system a global workload, i.e., one with diurnal patterns and regional object affinities. Thus, in using it to populate objects and generate a workload for our evaluation (see below), we strived primarily to preserve the object-access and client distributions.

5.3 EXPERIMENTAL SETUP

Hardware Our experiments consisted of emulations run using 4 Dell servers. Each server (on which we ran multiple proxies, see below) has 64 cores running at 2.3 GHz and 128 GB of RAM. We performed our emulations with 76 proxies spread across 4 servers, resulting in an average of between 3 and 4 CPUs per proxy. The only exceptions were our fault-tolerance experiments, in which each proxy was accompanied by a backup, doubling the total number of proxies on the same hardware.

Proxy placement Recall that the number of servers (with consistency falling short of LOKO) that Akamai dedicates for the load that our traces represent is 357, and so we needed to scale down the Akamai trace to permit a realistic evaluation for 76 proxies. To do this, we selected 4 geographic regions that accounted for $72/357 = 20.2\%$ of all queries in the original trace and allocated 72 proxies to those regions proportionally to the number of requests originating there. More precisely, we first geolocated the clients in the Akamai traces using the database from IP2Location [8] and truncated each one's latitude and longitude to an integral value, yielding its "region". We allocated a number of proxies to each selected region proportional to its queries; e.g., if one region originated 10% of the 20.2% of queries selected from the original trace, then it was allocated $10\% \times 72 = 7$ proxies. (The remaining 4 of the 76 proxies in our experiments are described below.) Thus, all proxies in the same region are, by definition, within the same integral latitude

and longitude. For our purposes, we treat them as though they are in essentially the same place—i.e., within a single datacenter.

Clients at each region were then assigned to that region's proxies to yield a roughly balanced number of queries at each proxy (in a manner oblivious to the contents of those queries). The 4 selected regions included one in Asia, one in Europe, and two in North America, and so we believe this methodology produced a reasonable approximation to a global workload. While client requests drive our experiments, clients themselves are not instantiated (or measured) in our experiments. So, latency between a client and its proxy is not represented in our measurements, nor are client computational costs, e.g., for verifying signatures.

Note that by choosing the 4 regions that account for the most queries, we may have lost some ability to migrate in response to demand within each region. That is, each region in our case contained 18 proxies, on average, and the clients connected to these proxies all represented real clients in exactly the same (integral) latitude and longitude. Any diurnal or region-specific request behavior exhibited by the clients in the region may have differed from behavior of clients in other regions, but it was unlikely to differ much within a single region. So, migrating a given object toward a specific set of proxies within a region that are requesting that object could be difficult, as all the proxies in the region may have had an equal chance of having clients interested in that object. On the other hand, had we chosen more regions, each of which with a smaller portion of the total trace, we may have enabled more migration. For example, if we used 18 regions, with each containing 4 proxies on average, each region would represent a smaller client population (e.g., France, instead of all Europe). The slight differences in client behavior between these smaller regions could then drive objects to migrate between them, possibly enabling even more benefits from migration than those shown below.

Network latencies To generate the tree topology for our experiments, we added an additional *head proxy* per region and built a minimum spanning tree covering the head

proxies using geographical distance as our distance measure. Each region's other proxies were then organized in a balanced ternary tree underneath the region's head. So, the total proxies in each experiment was $72 + 4 = 76$, of which only the 72 non-head proxies accepted requests from clients directly. Once the tree was fixed, we estimated latencies between neighboring proxies as a linear function of the geographical distance between them, where this function was calculated using linear regression on real distance/latency pairs: We took round-trip latencies (ms) from AT&T [6] from Kansas City to 24 other cities in the continental US, as well as from San Francisco to Hong Kong, New York to London, and Washington to Frankfurt. We then obtained distance estimates (miles) [7] for these city pairs. Using simple linear regression, the best fit line to these distance/latency points was $y = 0.019732193x + 8.712212072$ with an R^2 of 0.96820894, indicating a strong goodness of fit. We believe our use of distance-based latencies from within a single provider's network is reasonable, since our service may well be implemented by a major global provider. Note, though, that because proxies in the same region have a distance of 0 miles between them, this formula will assign a round-trip time (RTT) of about 8.7 ms within the same datacenter. In reality, such a high RTT would almost certainly not be permitted within a modern datacenter, but we use these estimates nonetheless in order to keep a consistent policy across all inter-proxy latency estimates.

We emulated proxy-to-proxy latencies at the user level, using the method implemented in the EmuSockets toolkit [15].[1] We did not limit the bandwidth between proxies, because we do not expect LOKO to even remotely tax the capacity of future networks (or even today's).

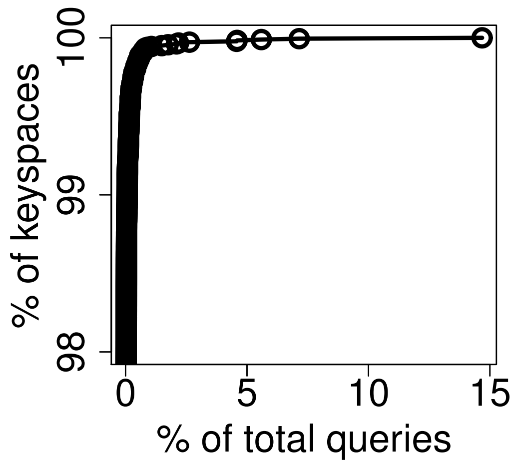
Keyspace objects The queries selected as described above were used to populate keyspace objects as follows. Every DNS query indicates a DNS zone, the requested name in that zone, and a query type. The query type can indicate an IPv4 host (`A`) record,

[1]This design is an artifact of our trying out several different platforms for our emulations, including some where we were restricted to user-level modifications only.

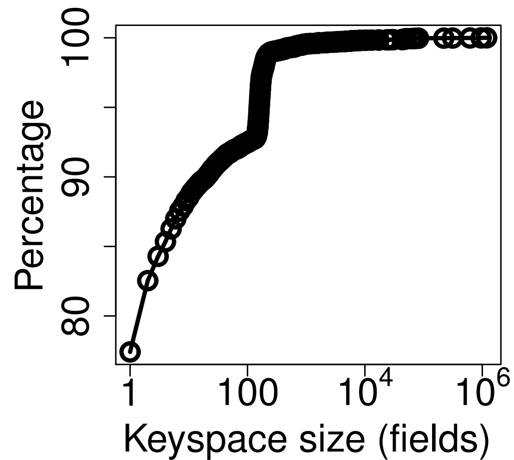
an IPv6 (`AAAA`) record, a name server (`NS`) record, etc. We created a keyspace object per zone and initialized it with a field for each name within that zone for which an `A` record was requested (e.g., "`www/A`"), since `A` records overwhelmingly constitute the most common form of query. The value assigned to each such field was a random 16-byte value. We made no effort to represent resource records in keyspaces more explicitly, remembering that the goal of using the Akamai traces is to induce a realistic global workload on LOKO rather than to make LOKO mimic DNS faithfully. Rather than signing each mapping individually, we compute a Merkle tree [57] over the mappings, signed by the private key corresponding to the keyspace's public key. The Merkle tree is transient; i.e., only the signed root is sent when the keyspace is copied (to support a read) or migrated; the interior nodes are recomputed on demand.

The 20.2% of the original trace that we used included 4,460,838,100 queries spanning 1,009,689 domain names and 83,448 clients. Figure 5.1(a) shows that when used to construct keyspace objects as described above, there were a few keyspaces which dominated the queries, in that requests for those keyspaces were a significant portion of the total requests. The most frequently queried keyspace object comprised over 14% of the total, and the 5 most frequently queried keyspace objects comprised over one third of all requests. The distribution of keyspace sizes was also far from uniform, as shown in Figure 5.1(b). While over 88% of all keyspaces contained less than 10 keys, some contained over one million.

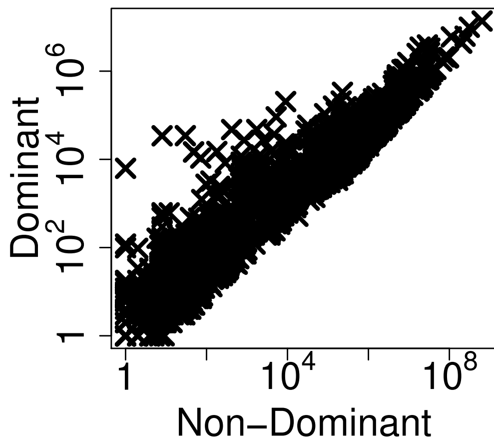
Prior to each measurement run of LOKO, we determined the starting location of each object by executing a warmup. The warmup migrated each keyspace object to its *dominant proxy*, i.e., the proxy that will make the most requests of it during the run. This warmup thus implements an optimal *static* placement of keyspace objects for the run. Nevertheless, as shown in Figure 5.1(c), the request rate by the dominant proxy for a keyspace is strongly correlated with the request rate by other, non-dominant proxies for that keyspace, implying that operation workloads will be dominated by nonlocal operations in any static placement



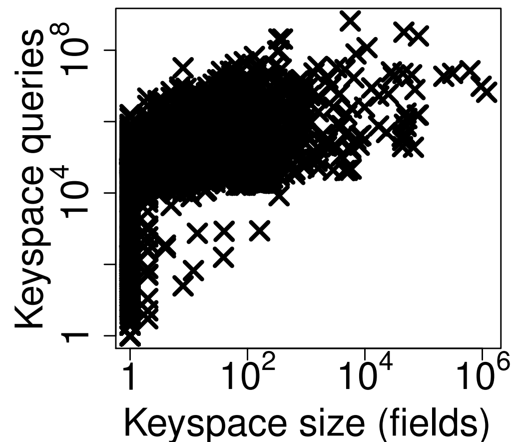
(a) CDF of queries per keyspace. A few keyspaces comprise a large portion of requests.



(b) CDF of keyspace size. Note the x-axis is log-scale. Most keyspaces are small, but some are quite large.



(c) Dominant vs. non-dominant proxy queries (one × per keyspace). These query types are strongly correlated.



(d) Keyspace size versus queries to that keyspace (one × per keyspace). The larger keyspaces tend to be accessed more frequently.

Figure 5.1: Keyspace query and size distributions

of keyspaces.

Update operations As the Akamai traces include no updates, we introduced updates artificially. Specifically, for a parameter $u \in [0, 1]$, each read operation for a keyspace submitted to its dominant proxy was converted to an update operation with probability u.[2] Because the rates of requests to keyspace objects from their dominant proxies were highly skewed (see Figure 5.1(c)), these update operations were not uniformly spread across keyspace objects but instead were concentrated in those that were also read most often, including read most often from non-dominant proxies (again, see Figure 5.1(c)). So, these updates caused many caches to become invalid and thus many object sends, and, because the keyspaces accessed the most often tended to be larger (Figure 5.1(d)), these sent objects also tended to be large.

If a query was chosen to become an update, an update was generated in its place for the relevant keyspace object, consisting of the relevant query name and query-type string (e.g., "`www/CNAME`"), a 16-byte value, and a 128-byte digital signature on the root of that keyspace's new Merkle tree (i.e., the previous Merkle tree updated to reflect the newly added or modified field). The proxy to which this update was introduced verified the signature using the public key of the keyspace. Since client costs are not included in our measurements (see above), signature generation for update operations or signature verification after a read were omitted.

Time scaling Recall that our Akamai trace was 36 hours in length. Due to the number of experiments we wished to perform with this trace, it was not possible to dedicate a full 36 hours per experiment. Simply truncating the trace would hide important trace characteristics, notably any diurnal pattern. As such, we "compacted" the trace as follows, while retaining its characteristics. Each experiment was parameterized by a *sampling rate*

---

[2]Our chosen method of inducing updates is just one of many alternatives. For example, another method might be to use the TTL of each DNS entry as a guide. That is, we could have marked the first operation for each DNS entry as an update. Then, for the remainder of the TTL (as read from the original trace), each operation for that entry could have been considered a read operation. Once the TTL had passed, the next operation would have been considered an update again, and so on.

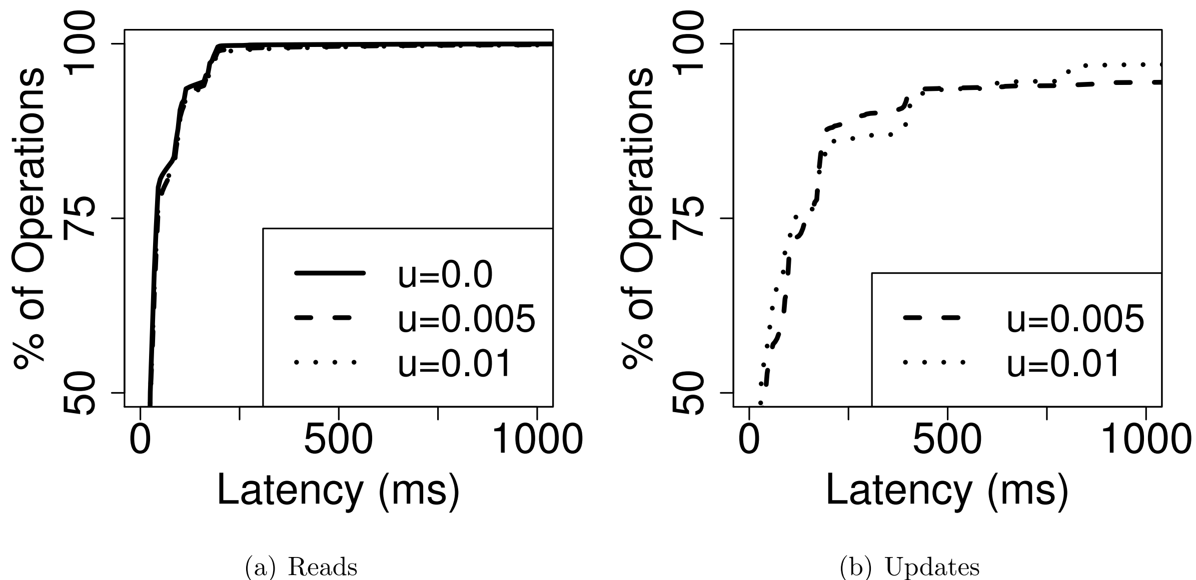
$s \in (0, 1]$ and an *acceleration* $a \geq 1$. Each query in the trace was then replayed in the experiment independently with probability s, and the trace was accelerated by a factor of a. So, in a period in which the rate of requests in the original trace was q requests per second, sampling reduced this rate to sq requests per second in expectation, and acceleration increased this to sq requests per $1/a$ second in expectation. This method shortens the trace replay to $1/a$ times the original, thereby expediting our tests; in our tests we fixed $a = 48$ so that each test required 45 minutes. However, we sometimes varied the sampling rate s between experiments. It is convenient to describe an experiment in terms of the product sa, which we will call its *load factor*. For example, an experiment with load factor $sa = 0.1$ has an expected request rate of 10% of the original Akamai trace's rate.

While this method of compacting the trace preserves any overall diurnal pattern, it can hide other features. For example, the lower the sampling rate, the more likely that some keyspaces with a small number of total requests are not chosen to be used in an experiment at all. Similarly, we see in Chapter 6 that a low sampling rate can reduce the opportunity for clustering, as concurrent requests to the same object may not all be chosen.

5.4 EXPERIMENTAL RESULTS

All performance numbers in this section were produced using the Java Runtime Environment (JRE) distributed with Java SE 7. We configured the HotSpot Server Java virtual machine to use the Concurrent Mark and Sweep garbage collector to maintain responsiveness. Except when evaluating the impact of the migration threshold m below, we set $m = 0.75$, and except when evaluating throughput below, we set the load factor to 0.1.

Updates We first explore request latencies and, in particular, the impact of varying the fraction of updates in the execution on those latencies. Figure 5.2 shows CDFs of operation latencies in experiments for update probabilities $u \in \{0.0, 0.005, 0.01\}$, where $u = 0.0$ implies no updates. In Figure 5.2(a), we see that as updates become more common, latency tends to increase for reads, because updates cause caches to become invalidated,



(a) Reads (b) Updates

Figure 5.2: CDFs of latencies (ms) as u varies.

creating the need for more network traffic. Moreover, as discussed in Section 5.3, these cache invalidations tend to be focused on the larger and more frequently accessed objects, amplifying the performance impact of updates.

Despite these effects, read latency stays low, with 89.5%, 86.7% and 84.7% of reads completing in under 100 ms for $u = 0.0$, 0.005, and 0.01, respectively. Latencies for the updates themselves appear in Figure 5.2(b). These too perform well, with 67.7% and 66.0% completing in under 100 ms for $u = 0.005$ and 0.01, respectively. This low latency is partially an artifact of our warmup method, which initially places objects at the proxy which will request them most, making many updates local (except when the object has been migrated away). Note that this behavior is part of our design—migration will tend to move an object toward the proxies requesting it most.

Migration We illustrate the impact of object migration on operation latency in Figure 5.3. Recall that m represents the fraction of the total load for which a neighbor must account in order for migration in the direction of that neighbor to begin. Thus, $m > 1$ is impossible to satisfy and allows no migration at all. We ran experiments with various

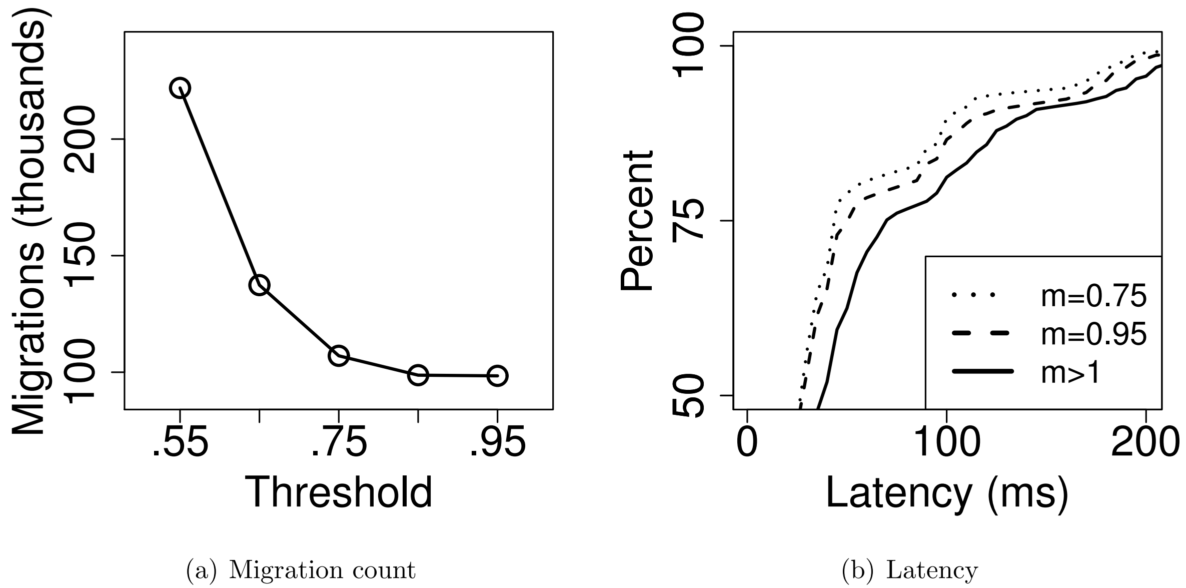
migration thresholds: $m = 0.55$ to 0.95 in increments of 0.1, as well as $m > 1$.

Figure 5.3(a) shows the total number of migrations for each setting of m, and Figure 5.3(b) shows the impact of these migrations on operation latencies. Without migration, 85% of operations finished in less than 120 ms. But even with migration enabled at a very conservative threshold ($m = 0.95$), that figure was reduced by 17% to 100 ms. Migration at that level also reduced the total number of proxy-to-proxy messages by 19%. Objects migrated within the tree in response to demand over 110,000 times, resulting in faster response times as well as fewer and smaller network messages sent.

Reducing m further increases performance. For example, at a very liberal threshold, $m = 0.55$, 85% of operations finished in less than 95 ms. In general, the performance differences resulting from different values of the migration threshold (e.g., $m = 0.55$ vs. $m = 0.95$) are much smaller than the differences between runs with migration and those without it (e.g., $m = 0.95$ vs. $m > 1$).

The reason for this disparity is that even a high migration threshold allows objects to move quite close to their areas of demand. If an object is far (in the tree) from the part of the tree where demand for the object is high, then the proxy hosting that object will see that nearly 100% of the load for that object is coming to it from whatever neighbor is in the direction of the load; the host will thus try to migrate the object to that neighbor (see Section 3.2.3). In this way, almost any migration threshold will allow migration of sufficiently out-of-place objects toward the parts of the tree where they are in the most demand. The exact value of m only becomes relevant once the object is near enough to its demand that significant fractions of demand for it come from different neighbors. But by that point, objects are already fairly close to the demand, and performance has already improved substantially.

Fault tolerance We measured the effect of fault tolerance on operation latencies when using LOKO, i.e., with a backup per proxy (see Section 3.2.4), for $u = 0.01$. The results appear in Figure 5.4. As expected, the overhead of fault tolerance is much more evident



(a) Migration count

(b) Latency

Figure 5.3: Impact of varying m, with $u = 0.0$. Lines for some values of m are omitted from Figure 5.3(b) for clarity.

for update operations, since communication with the backup is on the critical path of each update operation. One possible cause of the added read latency may be that we allocated no additional hardware to host backups, nor did we reduce the number of primary proxies to make room for their backups. Instead, the primaries and their backups shared the same resources that, in other experiments, were available exclusively to the primaries. Despite the more thinly spread resources and the synchronization costs of the primary-backup protocol, operation latencies with backups were still reasonably close to those without.

Throughput We next present experiments that offer insights into the achievable throughput of our system. In these tests, we increased the sampling rate s and so the load factor, up to a load factor of 1.0, i.e., the same query rate per proxy as Akamai supported in the original trace. Figure 5.5(a) shows the achieved throughput in operations per second with $u = 0.01$. This figure shows that our LOKO implementation absorbs the full per-proxy query rate of the Akamai trace. Figure 5.5(b) illustrates one reason behind this throughput, namely that as the operation rate increases, the effectiveness of read pausing also increases, since more reads are concurrent. This increase in read pausing then results in

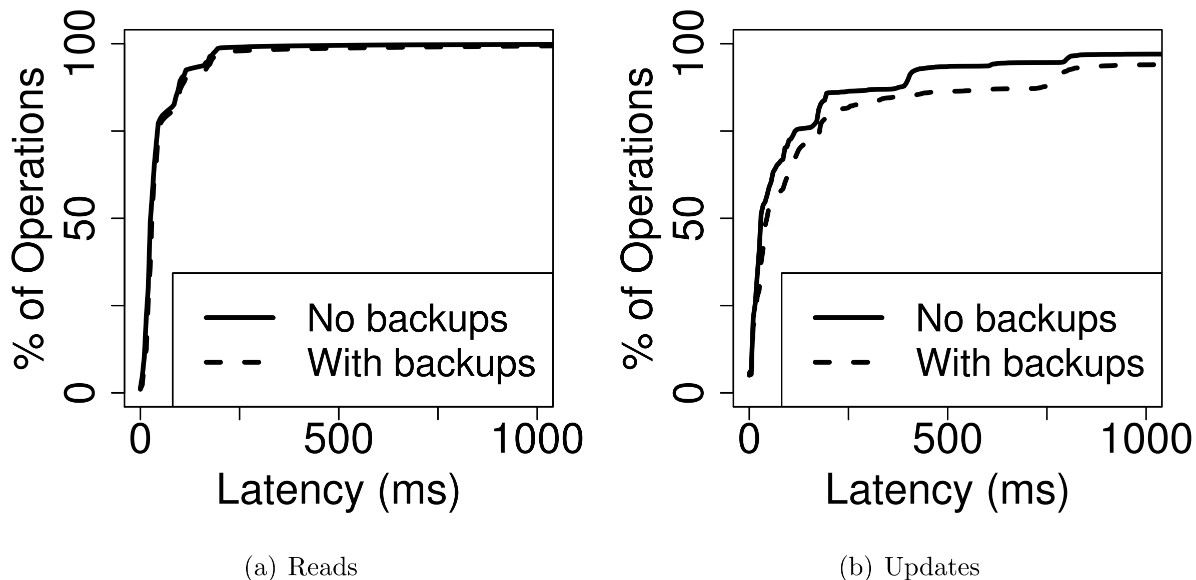


Figure 5.4: CDFs of latencies (ms) when using backups, with $u = 0.01$.

a reduced number of messages needed per operation, on average (Figure 5.5(b)). Finally, Figure 5.5(c) shows that the average number of proxy-to-proxy hops a read request travels before it is paused or reaches the object is stable, even as the load factor increases. When the load factor reaches 1.0, each read request travels about 1.75 hops on average.

Consistency In order to better show the performance improvement from cluster consistency, we also built a linearizable version of LOKO that we subject to the same load as the cluster-consistent version. Because cluster consistency itself represents a very specific

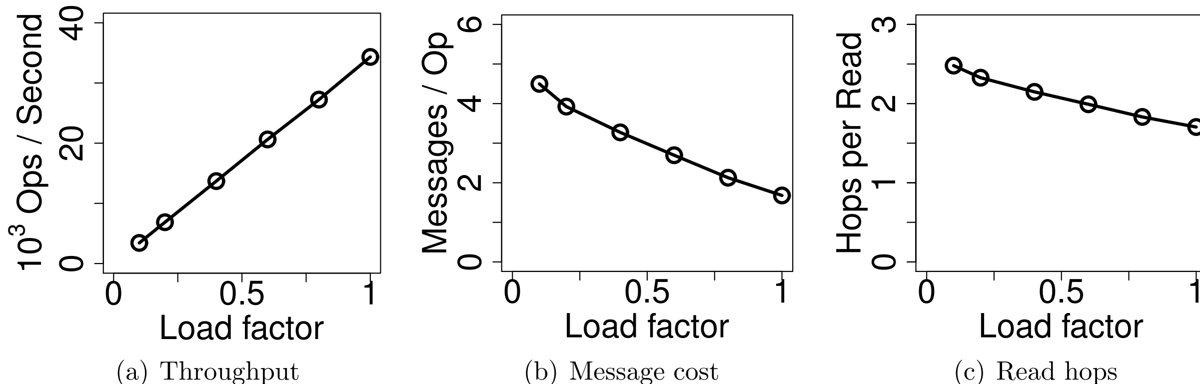


Figure 5.5: Throughput and messaging overhead as load factor varies, with $u = 0.01$.

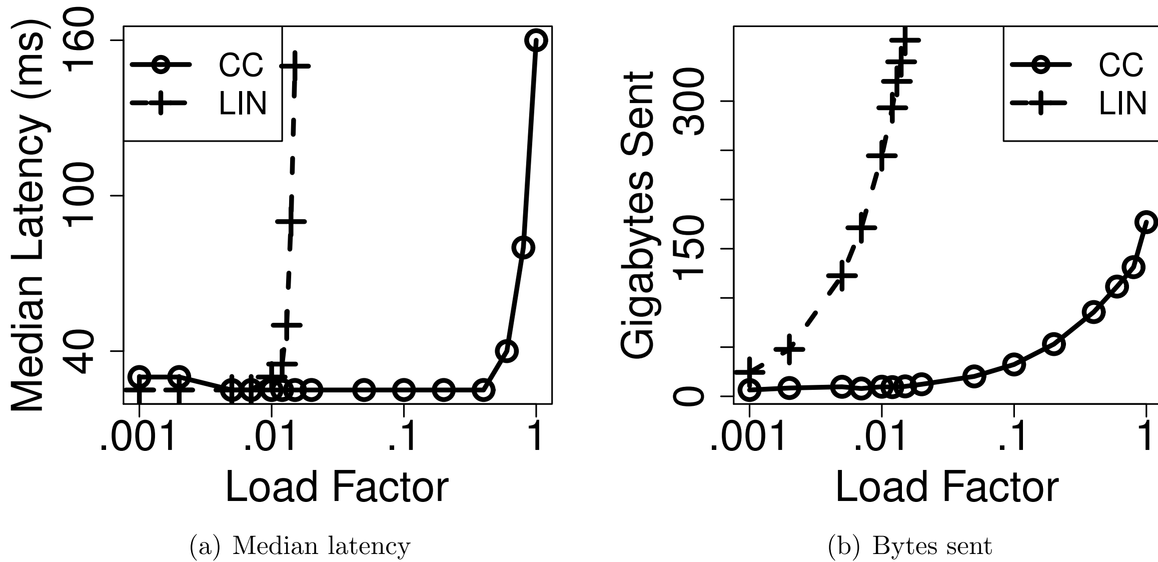
weakening of linearizability, we can revert to a linearizable version of LOKO with a few changes. To aid in our discussion, in this section we will refer to the linearizable version of LOKO as LIN-LOKO and the standard, cluster consistent version of LOKO as CC-LOKO.

The main change, of course, is that in LIN-LOKO there can be no read clusters. Instead, every read request is forwarded through the tree all the way to the object, even if many other requests are concurrent for the same object. Since read clusters contribute significantly to the scalability of LOKO, we would expect LIN-LOKO to succumb to heavy loads far sooner than CC-LOKO does.

Lacking clusters, LIN-LOKO can make a minor optimization. Instead of forwarding responses back through the tree, the hosting proxy can send them directly to the proxy that originated the request. Recall from Section 3.2.1 that the main reason responses returned through the tree was so that they could be used to answer paused reads along the way, which does not apply to LIN-LOKO.

One final change to produce LIN-LOKO is that, since responses are not sent through the tree, proxies forwarding requests no longer update the object version numbers in messages. That is, it is pointless for a proxy along the request path to take responsibility (see Section 3.2.2) for a request if the response never actually reaches that proxy.

Figure 5.6 gives the results of our comparison between CC-LOKO and LIN-LOKO using $u = 0.01$. Figure 5.6(a) shows the median request latency vs. load factor, with each point representing a single run at the given configuration. The graph shows that the two versions perform essentially equally at load factors of 0.012 and below. As the load factor increases to 0.015, LIN-LOKO quickly climbs to a median latency of 150 ms. We were unable to complete LIN-LOKO runs with load factors higher than 0.015 due to the massive computational burden placed in the proxies. In contrast, the median latency for CC-LOKO remains relatively constant until the load factor reaches 0.6, at which point it begins to rise, only reaching a median latency of 160 ms at load factor 1—equivalent to the full Akamai load factor. With either version, increasing load factor will eventually cause a



(a) Median latency

(b) Bytes sent

Figure 5.6: Impact of varying load factor on median latency and total bytes sent in both the cluster consistent (CC) and linearizable (LIN) versions of LOKO.

spike in median latency. But the spike in the CC-LOKO case occurs at load factors almost two orders of magnitude higher than in the LIN-LOKO case.

Figure 5.6(b) shows that the total amount of data sent in the two cases. The graph clearly shows that LIN-LOKO sends far more traffic than CC-LOKO, due to the lack of read clustering. For example, at load factor 1, CC-LOKO sent 177 GB. In contrast, LIN-LOKO sent 171 GB at only 0.007, a difference of 2–3 orders of magnitude. Together, these results show that the relaxation of consistency—bounded though it is in our case—can indeed lead to dramatic improvements in scalability.

5.5 LIMITATIONS

The Akamai data that we employed in our experiments is the best data we have found for a realistic, global workload. That said, it is important to recognize that this data set has limitations for the purposes it is used here. First, Akamai customers tend to be large organizations for which domain-name query activity might be heavier and more widespread than most domain names not served by Akamai or than other objects that one

might envision in a future application (e.g., a mobile device's location). This tendency might yield an overly optimistic evaluation of LOKO, since it makes more opportunities to aggregate (i.e., pause) reads in the tree, but it also might yield an overly conservative evaluation, since global demand reduces the ability to improve access latencies through migration. Second, as already noted, the Akamai data set contains no update operations, and so it was necessary to fabricate them.

Chapter 6: CASE STUDY: NFS TRACES

6.1 INTRODUCTION

One weakness of the DNS case study in Chapter 5 is that, as our DNS traces contained no update operations, we were forced to induce updates artificially. And, because these simulated updates did not reflect actual user behavior, testing LOKO's ability to serve them necessarily lacked some realism. With that in mind, and in order to explore the flexibility and performance properties of LOKO more fully, we subjected it to a second data set.

We collected more than two weeks of NFS traces on UNC-Chapel Hill's network with the help of NetApp, Inc. using their Chronicle [43] framework. These NFS logs (unlike the DNS logs before them) naturally include a subset of real-world, user-initiated writes to files, giving us a valuable way to subject LOKO to updates organically. On the other hand, because the NFS logs were collected on campus, they lack the global characteristics of the DNS data set. No data set is perfect, but by applying data from multiple sources to LOKO, we can get a better idea of its general usefulness and learn more about where it can be improved. Though, it is important to be clear that we do not intend by these experiments to create an NFS replacement or implement an actual NFS service—just as it was not our intention in Chapter 5 to faithfully implement the DNS protocol or create a DNS replacement. Instead, we use these data sets because they represent real user activity and thus have some properties which are useful in testing LOKO, despite LOKO itself being an implementation of WACCO—by design an entirely different kind of system from DNS or NFS.

For the sake of clarity, when distinguishing between LOKO in the contexts of the DNS or NFS cases, we will sometimes refer to DNS-LOKO or NFS-LOKO, respectively.

6.2 DIFFERENCES FROM DNS CASE STUDY

Using an NFS trace presents different challenges for LOKO than we faced with DNS-LOKO. The main differences are:

Server set The NFS data set was collected using NFS traffic to/from a specific set of NFS servers in UNC-Chapel Hill's research cluster. There were only four such servers, and they were all located in the same data center. By contrast, in the DNS case, there were 357 globally distributed servers represented in the logs.

Client set Because the NFS resources hosted by the UNC servers was only available to clients on campus, the set of clients represented in the NFS case is far smaller and less geographically diverse than in the DNS case.

Keyspace size Keyspaces in NFS-LOKO each represent a single filesystem object—i.e., either a file or a directory. All keyspaces contain the same three keys: The first, called `owner`, contains the user ID of the owner of the file. The second, `perms`, holds the file's permission bits. Finally, the key `contents` maps to the actual contents of the file. The main challenge with adapting LOKO to NFS keyspaces is that, in the existing LOKO design, we always return the entire object in response to, e.g., a read request. Since NFS files can be quite large, sending the whole keyspace is infeasible. We present our solution to this problem below in Section 6.3.3.

6.3 DESIGN CHANGES

Of all the challenges presented by the NFS data set, the most serious was the presence of large files. In this section we detail the three changes we made to our LOKO implementation to allow for these large files.

6.3.1 REDUCING MEMORY USAGE

Given the potentially large sizes of NFS files, we can no longer expect LOKO to store all objects in main memory. For keyspaces whose total size exceeds a threshold d, LOKO stores the value of the `contents` field on disk instead of in main memory. Without this change, LOKO proxies would run out of memory before even starting an experiment. Note that since `owner` and `perms` can both be represented within a fixed number of bits (e.g., an integer), keyspaces are large if and only if the `contents` of the files they represent are large, and so storing `contents` on disk is sufficient to mitigate main-memory capacity concerns from large files.

We expect a tradeoff when setting d: Large values will allow more objects to reside in main memory, potentially filling it to capacity and causing thrashing. Small values will cause more objects to be stored on disk, freeing space in main memory but increasing latency for any queries to those objects, due to increased disk seek time (vs. main memory).

6.3.2 REDUCING NETWORK LOAD: MIGRATION

In order to prevent network congestion due to migration of large keyspaces, NFS-LOKO disables migration of keyspaces exceeding a certain size c—the "migration cutoff." Small values for c will disable migration for many objects, reducing network load due to migration but thus increasing—or, more accurately, potentially failing to reduce—request latency for requests to those objects. By not reducing the number of hops per request, the lack of migration would also fail to reduce the number of messages for requests to the affected objects, and so we expect that a value of c that is too small may cause more harm than good from a performance standpoint. On the other hand, values that are too large will allow large keyspaces to migrate among proxies, potentially causing network slowdowns throughout the network, particulary in the case of low migration thresholds m that could cause some migration jitter in extreme cases.

6.3.3 REDUCING NETWORK LOAD: BLOCK REQUESTS

LOKO is designed to respond to read requests with a copy of the entire keyspace requested, in order to facilitate caching and read clustering (see Section 4.1). Very large files, as frequently encountered in NFS filesystems, are impractical to repeatedly send across the network on the critical path of a response. Furthermore, doing so would not faithfully mirror the behavior of NFS, which forces clients who wish to fully download a large file to do so over the course of many block-level requests, not all at once. Similarly, we have adapted NFS-LOKO to use block-level requests in the case of large files. We made several changes to LOKO to that end.

First, all requests for a keyspace's `contents` now carry with them the block they wish to access and the corresponding payload size, obtained from the original NFS logs. That is, each read request carries the offset (within the file) being read and the length of the payload that was returned by the read (in the original data set). Similarly, write requests carry with them an offset and a payload to write to that offset. Requests and responses also carry a boolean flag called `is-block`.

For small files, LOKO handles read requests normally, forming clusters when possible and caching objects along the way at its discretion. However, if a file's size exceeds a threshold b, then responses to requests for that object will not contain the entire keyspace. Instead, their `is-block` flag will be set to true, and their only payload will be the payload needed to answer the particular read request that reached the object.

When a response reaches a proxy along the path back to the originating proxy, it may encounter read requests that were paused awaiting its arrival. In the `is-block=false` case, the response would answer these reads and proceed along its path. But in the `is-block=true` case, the response does not carry enough information to satisfy any read except the one that reached the object.[1] So, the response marks these waiting requests as

[1]We ignore the case in which some paused reads asked for exactly the same block and wanted exactly the same response payload, as we believed this case to be relatively rare and not worth the extra bookkeeping

`is-block=true` and then unpauses them, freeing them all to proceed to the object. Future proxies receiving these unpaused requests will see the `is-block=true` flag and know not to pause those requests, as no clustering is possible for block requests. The added network overhead incurred by the loss of read clustering for these large objects should be neglible compared to the savings from not transferring the objects themselves over the network.

6.4 CHANGES TO THE DATA SET

There were a few features of the NFS logs that we thought we should address before using them to evaluate LOKO. First, as mentioned in Section 6.2, there are only four NFS servers represented in the logs, which may be too few to allow much opportunity for read clustering and migration.

Another problem is that NFS logs inherently present fewer opportunities for read clustering. The reason is that, in the NFS case, keyspaces correspond to individual files, which are likely to be accesible only by the user that created them. By contrast, our DNS logs included many instances of zones (i.e., keyspaces) being accessed by many clients simultaneously, as one might expect.

In an attempt to solve both of these problems at once, we altered the logs in the following way. First, we expanded the number of requests by a factor of 4 by duplicating each client and file. That is, from each file F, we created 4 copies, $F_0, \ldots, F_3$. Similarly, from each client C, we created 4 copies, $C_0, \ldots, C_3$. This expansion gave us 4 times the number of requests, which allowed us to use 4 times the number of servers, giving us a bigger tree and a better chance for read clustering and migration to have an effect.

The mapping of these new clients to the new files was done in a way designed to improve object sharing. For each client C that accessed a file F in the original trace, we mapped each client C_i to a file F_j chosen uniformly at random from among $F_0, \ldots, F_3$. We then

---

overhead to optimize for.

mapped all requests from that C_i to F_j, for all requests—that is, the mapping of C_i to F_j was chosen before the experiment and that same mapping was used through the entire experiment. This strategy allowed for some added sharing of files between clients. For example, it is possible that both C_0 and C_1 both choose F_0. If instead we had chosen a simple, one-to-one mapping of clients to files (wherein each C_i maps to F_i), there would be more requests and servers, but the same amount of sharing.

Lastly, in order to avoid giving ourselves at unfair advantage by having all clients C_i all issuing requests at exactly the same time (perhaps all to the same file), we altered the timestamps of each request so that the client duplicates all appeared to be in different time zones. So C_0 used the original, unaltered times but C_i was i time zones (i.e., hours) behind C_0. Note that for files that are accessed sufficiently often, this change will still allow for the time-shifted clients to issue requests that can be clustered.

6.5 EXPERIMENTAL SETUP

We ran experiments on the same hardware as in the DNS-LOKO case: 4 Dell servers, each with 64 cores running at 2.3 GHz and 128 GB of RAM. There were 16 proxies plus the root proxy (which did not serve client requests) spread evenly across these four machines.

Our experiments used a section of the logs beginning at midnight on September 9, 2014 and lasting 24 hours. We chose to experiment on a shorter log section than in DNS-LOKO because there are no global diurnal features in the data—all requests come from clients on the UNC campus. We mapped clients to proxies arbitrarily, with the only rule being that each proxy have about the same number of total requests issued to it (from all its clients combined).

As with DNS-LOKO, we could not afford to dedicate a full 24 hours for each experiment. As a result, we again parameterized our experiments with an acceleration a and a sampling rate s (see Section 5.3 for more about these parameters). Unless specified otherwise, the results below are for experiments using $a = 48$ (so that the main part of each experiment

takes only 30 minutes) and s set such that the load factor $sa = 0.1$. Other default values are: default migration threshold of $m = 0.75$, migration size cutoff of $c = 1$ MB, block response threshold of $b = 23$ KB (chosen because 23 KB is the average size of a response payload in the NFS trace, so LOKO would never send a larger payload than NFS would).

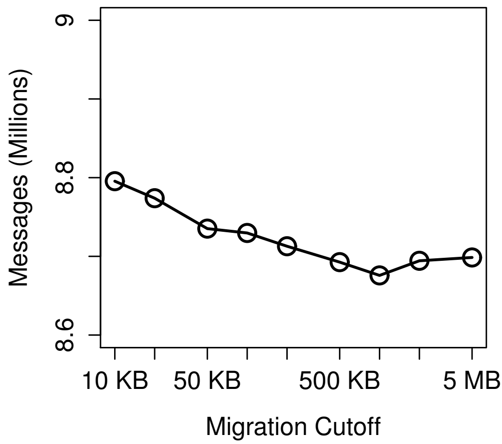
6.6 EVALUATION

Two important changes made for NFS-LOKO are the the addition of the new thresholds, which determine how large an object can grow before: (i) its migration is disabled (via c) and (ii) it uses block messages to respond to requests (because of b). We now explore how the choices for these thresholds can affect LOKO behavior and resource usage.

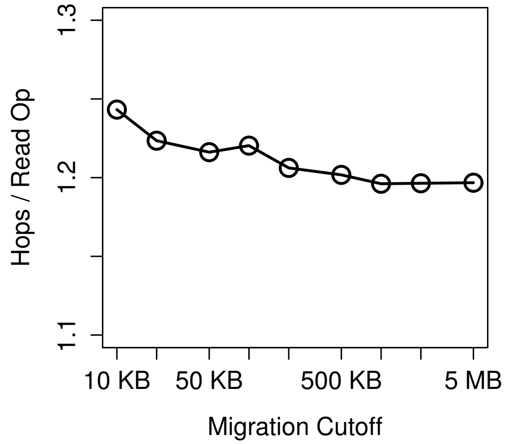
Figure 6.1(a) shows that the choice of c has a very slight negative effect on the total number of messages sent during a run, as one might expect: a rising c results in a few more messages sent during the extra migrations it enables, but the migration would never take place unless doing so would cause a reduction in average number of hops (and thus messages sent). Figure 6.1(d) shows that increasing c also slightly reduces the percentage of responses that are block responses. One explanation for this effect is that, since block responses are only sent for large objects, increasing c allows large objects to move closer to their demand, reducing the hop count for block messages and thus the total number of these messages. Figure 6.1(b) shows evidence for this reduction in the average number of proxy-to-proxy hops per read operation.

The real effect of increasing c is that, obviously, more and more migrations can occur, as seen in Figure 6.1(c). Because they are triggered by the change in c, these new migrations are of ever larger objects, which naturally cause greater and greater network load. Figure 6.1(e) shows that eventually this load becomes quite noticeable, with $c = 5$ MB resulting in over 12% more bytes sent than $c = 10$ KB.

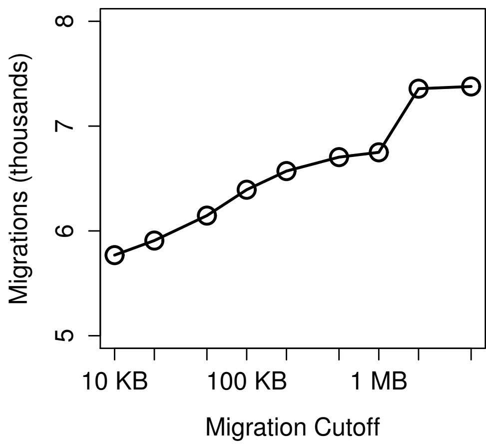
Finally, changing c has little effect on the latency of requests. The latency CDFs of all our migration cutoff experiments could be covered by a single band less than 4% wide—that



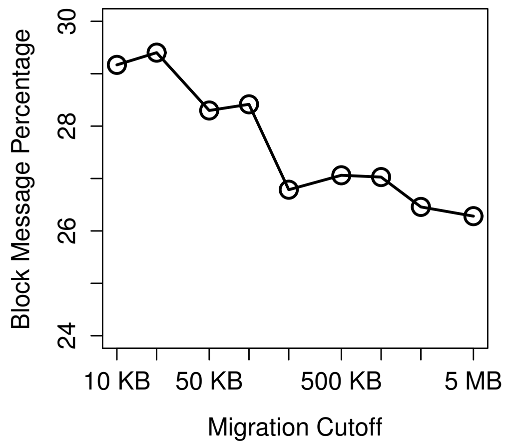
(a) Messages sent vs. migration cutoff



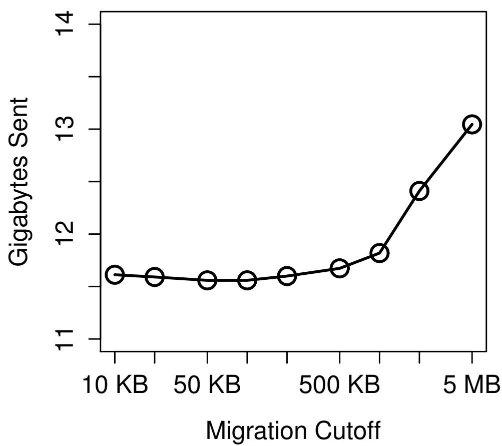
(b) Read hops vs. migration cutoff



(c) Total migrations vs. migration cutoff



(d) Percent block messages vs. migration cutoff



(e) Bytes sent vs. migration cutoff

Figure 6.1: The effects of changing the migration cutoff.

is, for every possible millisecond latency the percentage of requests that finished with at most that latency differed by less than 4% over all choices of c (though, see the discussion in Section 6.6.1 for more about these latencies).

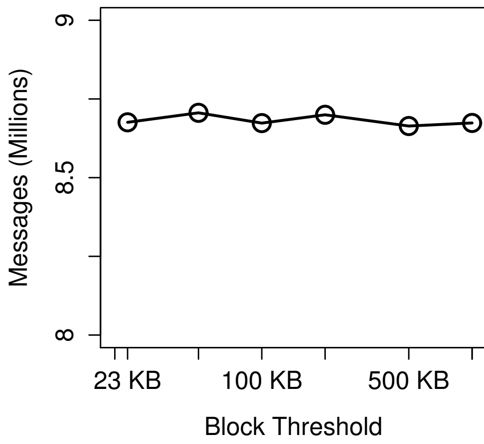
Figure 6.2 shows the effects of varying b. In particular, we can see that, as expected, the amount of block messages has no direct impact on the total number of migrations (Figure 6.2(c)). However, we do expect, naturally, that increasing b will cause the percentage of block messages to fall (as seen in Figure 6.2(d)), because as b gets larger fewer objects will have size exceeding b and thus be required to use block responses. So, raising b causes increasingly large objects to use standard LOKO responses, which send the entire object back to the requesting proxy. Figure 6.2(e) shows the resulting increase in the number of bytes sent through the network during a run. There are over 16% more bytes sent when $b = 1$ MB than when $b = 23$ KB, and the rate of increase seems to be rising.

As with c, changes to b do not seem to have a great effect on request latency—all latency CDFs can be similarly covered by a band less than 5% wide. Though, as we discuss below, the overall latency is still too slow.

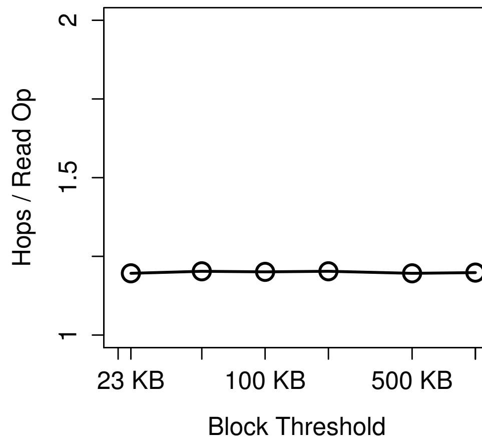
6.6.1 PROBLEMS WITH THE EXISTING SETUP

We have seen that the thresholds c and b have little relative effect on request latency in LOKO, but the latency itself is quite poor across all experiments. Figure 6.3 shows a clearer picture of the overall performance when c and b change. We have omitted some of the lines to make the individual curves easier to distinguish. On average, read requests are much faster than write requests, as one would expect. But even for reads only 90% complete in under about 1–4 seconds for most values of c and b. Only a bit more than half of write requests finish in the same amount of time, and there is a long tail.

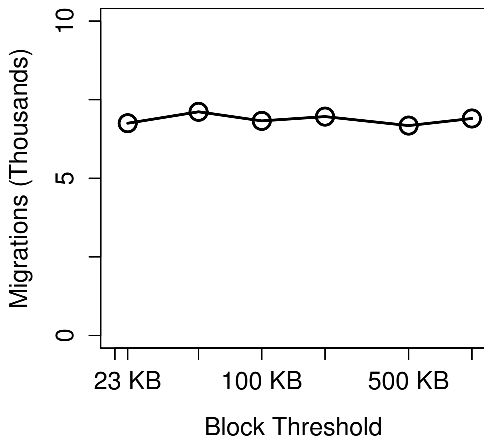
The truth is that trying to faithfully mimic the NFS service as seen in the logs is not what LOKO was designed for. Recall that we originally sought these NFS logs for their embedded write requests (so that we could test LOKO without artificially inducing writes).



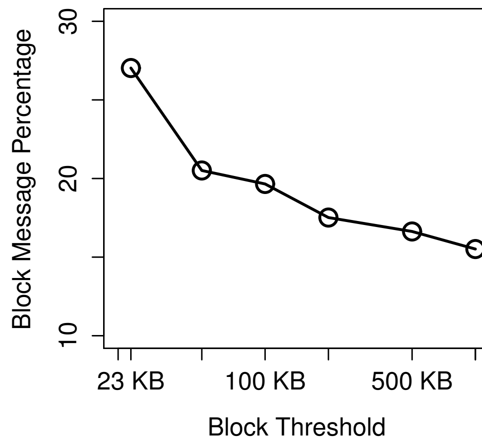
(a) Messages sent vs. block threshold



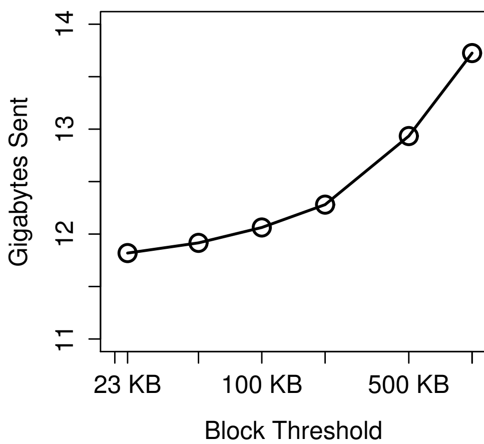
(b) Read hops vs. block threshold



(c) Total migrations vs. block threshold

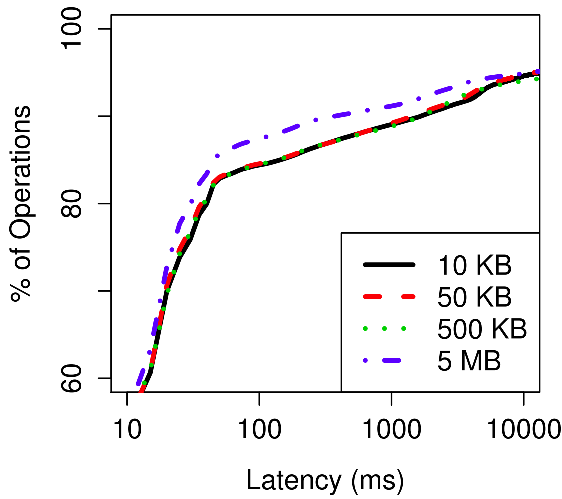


(d) Percent block messages vs. block threshold

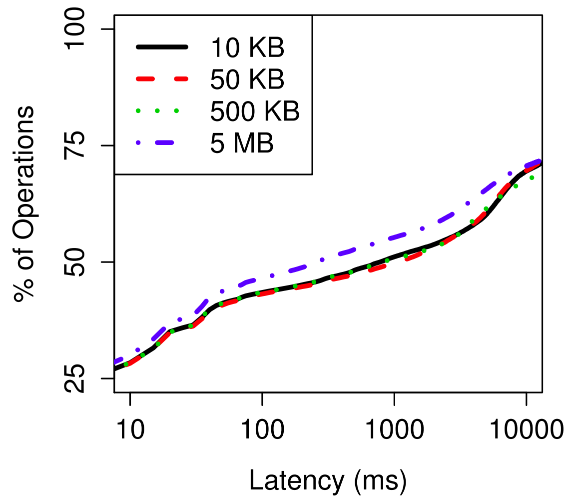


(e) Bytes sent vs. block threshold

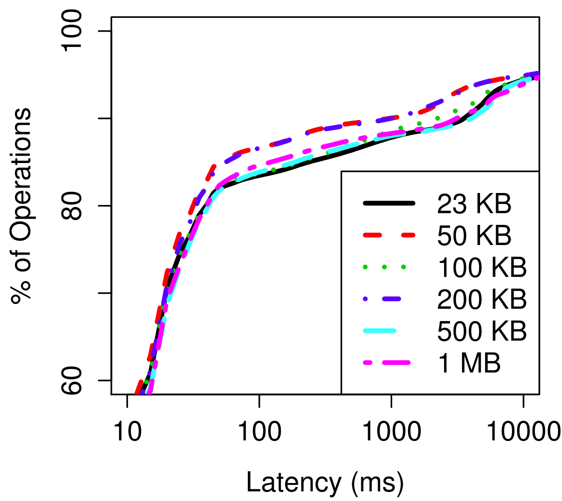
Figure 6.2: The effects of changing b.



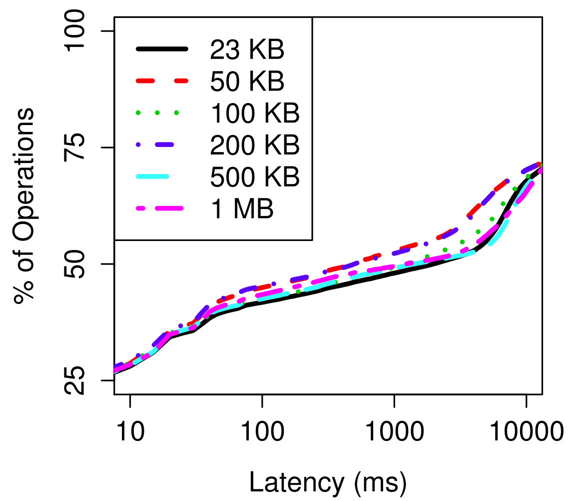
(a) CDF of Read Latency as c changes



(b) CDF of Write Latency as c changes



(c) CDF of Read Latency as b changes



(d) CDF of Write Latency as b changes

Figure 6.3: CDFs of latency as c and b change

Our method of duplicating the logs to 4x the size did not introduce enough inter-client file sharing to trigger much read clustering, and (perhaps more importantly) the large file sizes cause too much network traffic, too much disk latency, or both. Both features are counter to the original design goals for LOKO objects, which we expect to be distributed, globally-shared objects of smaller size. To that end, we reevaluated the changes we had made to the data set (see Section 6.4) and our experimental setup (see Section 6.5) and devised a new setup.

6.7 CHANGES TO THE DATA AND SETUP

As discussed above, the two main incompatibilities between the nature of the NFS traces and what LOKO expects are the lack of inter-client object sharing and the large file size. We will address both of these incompatibilities before running a second set of experiments. To avoid confusion between these new experiments and those already presented, we will sometimes refer to them as NFS-β and NFS-α, respectively.

6.7.1 CLIENT DUPLICATION

Recall that for NFS-α, we expanded our data set by duplicating each client C and each file F 4 times, then randomly pairing clients $C_0, \ldots, C_3$ with files $F_0, \ldots, F_3$. For NFS-β, we duplicate clients in the same way, but we do no file duplication. That is, each client $C_0, \ldots, C_3$ will all use file F. In NFS-α this kind of all-duplicate client file sharing only occured one time in 64, on average.

In NFS-α, we assigned client duplicates to different time zones. Thus, even if clients C_0 and C_1 happened to choose the same F_i, they would only have temporally overlapping requests (and thus the ability to merge those requests) if C_0 issued requests separated by exactly an hour (plus or minus the actual request duration). For very busy objects, we expected that condition to hold, but our experiments show that for most objects it did not hold, and the sharing we hoped to induce was lost. For NFS-β, we have chosen to

separate clients by only about 10 ms each. This tighter grouping mimics (in short bursts) the concurrent read load we expect for popular, globally-shared objects.

6.7.2 LOG WINDOW

Recall that NFS-α used a log window beginning at midnight on September 9, 2014 and spanning 24 hours. In order to run experiments quickly, though, we used an acceleration of $a = 48$. At load factor 0.1, that meant we were using only one in every 480 requests. By "squeezing" such a long window into 30 minutes, we skip enough messages that we again lose some ability to form clusters. As an example, imagine that a certain request is chosen from the log to be run in the experiment. This request is from one of 15 clients that all issued requests (to that same object) that overlap in time. That is, if all 15 messages were used in the experiment, they could potentially all form a read cluster. With the configuration used in NFS-α, the chances that none of the other requests is used in the experiment is over 97%—so probably no clustering can happen for that request.

To avoid this problem, for NFS-β, we chose to use a 1-hour log window beginning at 3pm, September 9, 2014. To run experiments in 30 minutes, we need only accelerate by $a = 2$. So, at load factor 0.1, we can use one in every 20 requests in the log window. Using the example above, the odds are better than half that there would be at least one other, overlapping request chosen for the experiment, which would potentially allow for some read clustering.

In DNS-LOKO, it was important to have a long log window, because one key feature of the DNS data was its global distribution, and we wanted to test LOKO's behavior in the face of geographically diverse diurnal patterns. In the NFS case, however, that feature is missing, and so we benefit more by using a smaller window from which we use relatively more requests.

6.7.3 FILE SIZE

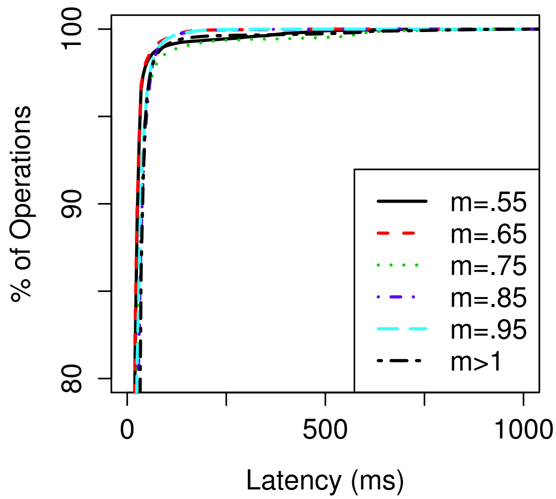
The sheer size of many of the files used in the NFS-α experiments was enough to cause problems with LOKO running out of both RAM and disk space. The total size of all the files served was several terabytes. It should come as little surprise that these large file sizes caused problems for LOKO, which was not designed to handle them.

To alleviate the problems caused solely by the large files, we limited the size of each file in NFS-β to a maximum of 1 MB. This choice may seem somewhat extreme, but over 90% of requests are for files that were already less than 1 MB and so are unaffected by this limitation. We then set both $c > 1$ MB and $b > 1$ MB, effectively disabling block messages and allowing any object to migrate based on load. These changes make files easier to send as responses to requests, while preserving the real-world write load present in the NFS traces, allowing us to evaluate LOKO's write performance on real-world behavior.

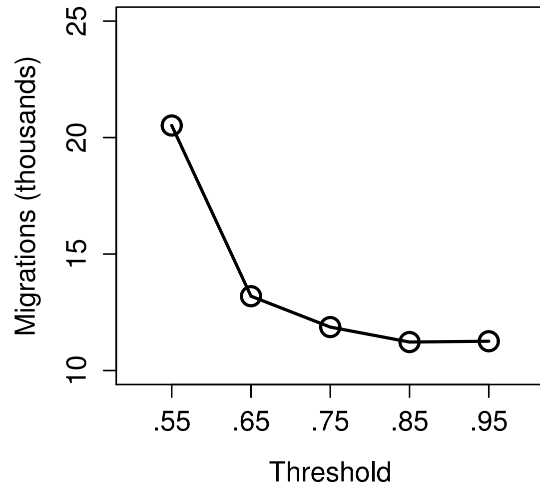
6.8 EVALUATION OF NFS-β

Average latency for operations in NFS-β is very good. Figure 6.4 shows that migration has little effect on the latency of operations, presumably because, with only 17 total proxies, the tree is much smaller than in DNS-LOKO. Also, since duplicate clients C_i issue requests only a few tens of milliseconds apart, the request load probably often appears to come equally from several neighbors much of the time, resulting in fewer migrations. Comparing Figure 6.4(b) with Figure 5.3(a) (which shows total migrations per m in DNS-LOKO), we see that there are about 10 times as many migrations in a DNS-LOKO run as in an NFS-LOKO run, for equivalent m, which seems to confirm our suspicions.

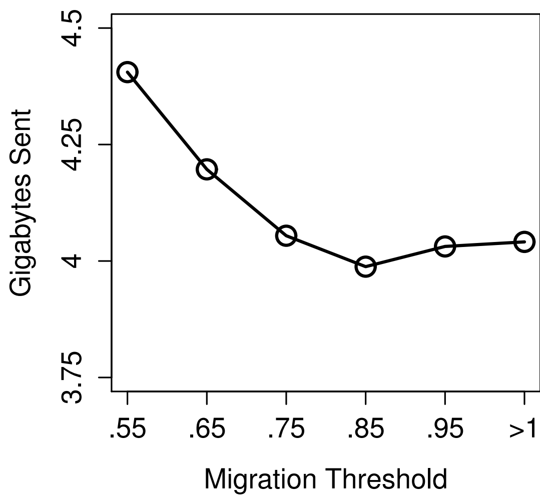
Figure 6.4(b) and Figure 6.4(c) show that the choice of m is not completely irrelevant, though. Lower values for m seem to cause some spurious migrations that results in more bytes sent over the network. When compared with the experiment using $m = 0.55$, the one using $m = 0.85$ sends about 90% as many bytes across the network and has only about



(a) CDF of average latency as m changes



(b) Total migrations vs. migration threshold m



(c) Bytes sent vs. migration threshold m

Figure 6.4: Effects of varying migration threshold m in NFS-β

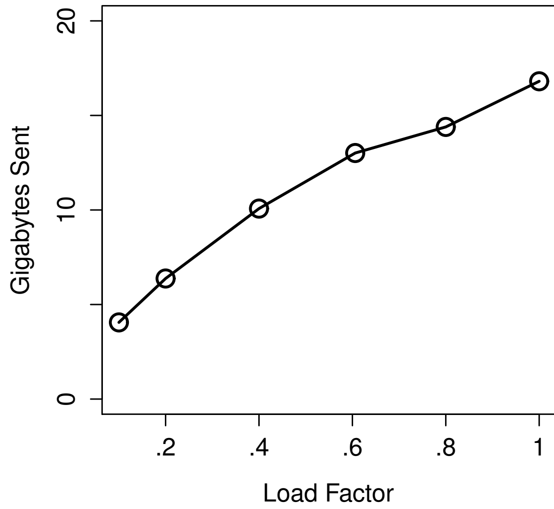
55% as many total migrations.

Figure 6.5 shows the effects of varying load factor in our NFS-β. As expected, increasing load factor will increase the total number of bytes sent over the network, as well as the total number of messages sent, as seen in Figure 6.5(a) and Figure 6.5(b), respectively.

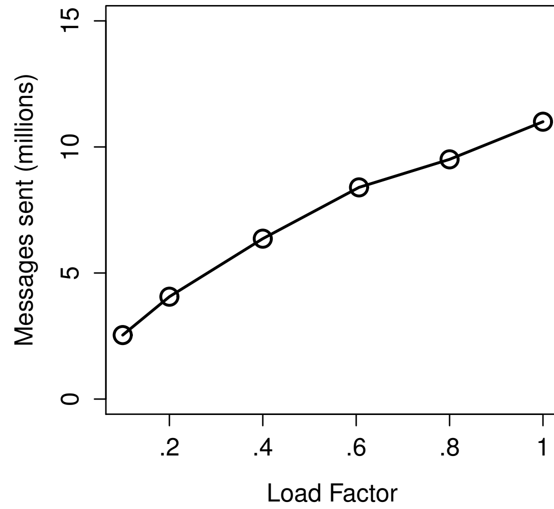
Finally, as with DNS-LOKO, we can see in Figure 6.5(c) that the average number of messages needed per operation decreases the busier NFS-LOKO gets, i.e., as the load factor increases. Most of our NFS-β experiments ran with load factor 0.1, which used 2.33 messages for each operation. At load factor 1, LOKO needed only 1.02 messages per operation, about 43.8% as many. The higher the rate of incoming messages, the more efficient LOKO messages become, until there is nearly a 1:1 ratio of messages to requests. Similarly, the average number of proxy-to-proxy hops per read operation falls as load factor increases, as shown in Figure 6.5(d).

6.9 CONCLUSION

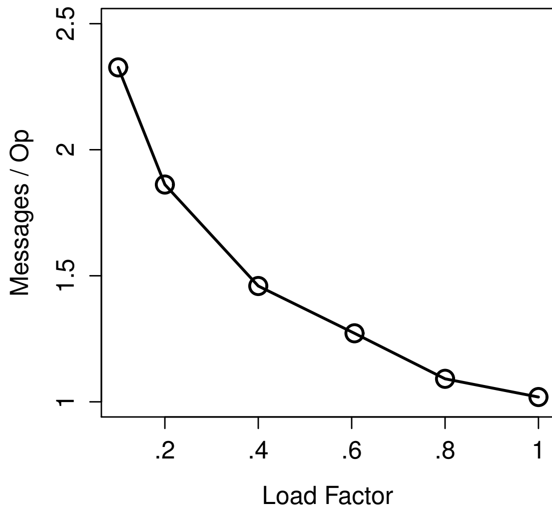
Using real-world NFS traces, we have subjected LOKO to a load that includes genuine updates, rather than artificially generated ones. This new data set enabled an exploration of how LOKO might be adapted to support large objects. In the end, these adaptations proved insufficient to cope with objects as large as those in the NFS trace, but they may be helpful for any future changes to WACCO and LOKO. With the NFS traces adapted somewhat to meet LOKO's expected load signature (by adding some potential for clustering and tempering object sizes), LOKO performed well, showing that its design and implementation were not overfit to the DNS data used in Chapter 5.



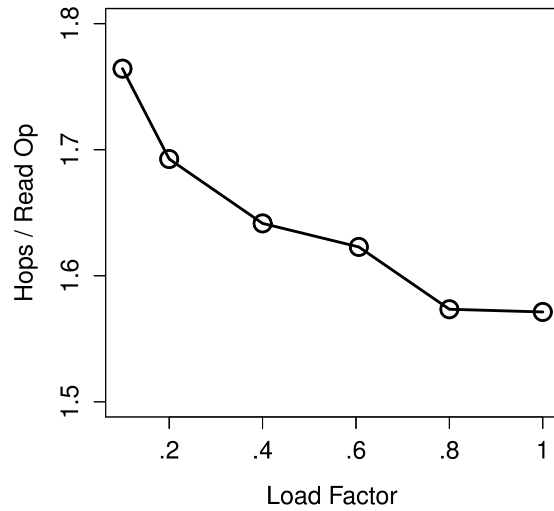
(a) Bytes sent vs. load factor



(b) Messages sent vs. load factor



(c) Total messages sent per operation vs. load factor



(d) Average read hops per operation vs. load factor

Figure 6.5: Effects of varying load factor in NFS-β

Chapter 7: MIGRATION STRATEGIES

7.1 COMPUTING OPTIMAL OBJECT PLACEMENT

In the migration strategy described in Chapter 3, objects move throughout the tree towards the areas of greatest load. In this chapter, we consider more complex stategies that place objects based on many other variables (in addition to area of greatest load). For example, in our original strategy, which we call MIGRATE-LOAD-ONLY, objects move entirely independently from one another, so that a proxy could potentially become overwhelmed simply by hosting too many objects. In addition, MIGRATE-LOAD-ONLY does not consider the financial aspect of migration: costs for object hosting and serving traffic can vary among proxies, making choices for object placement important for more than just client-visible latency but for budgets as well. In fact, the strategies we present in this chapter will prioritize budgets over latency when making migration decisions.

Organizations wishing to use WACCO may choose to host their globally-placed proxies on a public cloud (e.g., Amazon Elastic Compute Cloud (EC2), Google Compute Engine, Microsoft Azure—we will keep our design independent of cloud provider), citing both convenience and cost. But many organizations also now use *hybrid clouds*, in which some machines are private—owned and controlled by the organization—while others are part of a public cloud. In hybrid clouds, the cost of object placement can be an important factor. For example, it may be free to store an object on a private proxy but cost a certain amount per gigabyte-month to store an object on a public proxy. Traffic costs can also be asymmetric. For example, messages sent from a proxy in the private cloud to a neighboring proxy in the public cloud may be free, while messages sent in the other direction may incur a cost per byte (mirroring the asymmetric traffic billing in Amazon EC2). In LOKO, the

response to an operation can be much larger than the initial request, so, in the above pricing scenario (and ignoring for a moment all other constraints) when deciding which of two neighbors (one public, one private) will host an object, it might be better to place it at the private neighbor, since the requests for the object from the public cloud will cost less to send than their returning responses will. These nuances are ignored entirely by MIGRATE-LOAD-ONLY.

In this chapter, we present two related strategies for intelligently migrating objects in WACCO based on latency, hosting cost, and traffic cost, among other factors. Furthermore, unlike MIGRATE-LOAD-ONLY, these strategies consider many objects simultaneously, because objects and the messages serving them must share the same resources, and so migrations do not occur in a vacuum. Our techniques leverage integer linear programming (ILP), a well-known optimization technique. Specifically, we use 0-1 integer linear programming, also known as binary integer linear programming (BILP) to solve the following optimization problem: While keeping the cost of serving and hosting WACCO objects within a given budget, what placement of objects will give the lowest latency for clients? The remainder of this chapter is organized as follows: Section 7.2 and Section 7.3 present two migration strategies based on BILP. In Section 7.5 we evaluate one of these strategies using the DNS data from Chapter 5, with our experimental setup in given in Section 7.4. We conclude in Section 7.6.

7.2 UNRESTRICTED, GLOBAL BILP

In this section, we present a binary integer linear program (BILP) that can be used to decide the optimal placement for all objects hosted by WACCO. Specifically, the BILP we use decides the placement for each object such that the overall latency is minimized, while the cost to host and serve all the objects remains below some threshold. The global BILP presented in this section is meant only as a hypothetical model which will serve as an interim step on the path to the local BILP we present in Section 7.3 and evaluate in

Section 7.5. Section 7.2.6 defines some other constraints that could optionally be applied to further restrict object placement. Section 7.2.7 describes the assumptions and limitations of this approach.

7.2.1 DECISION VARIABLES

We define the set of objects to be $\{o_i : i \in \{1, 2, 3, \ldots, n\}\}$ and the set of proxies to be $\{h_j : j \in \{1, 2, 3, \ldots, m\}\}$. The decision variables are

$$x_{ij} = \begin{cases} 1 & \text{if we decide to migrate } o_i \text{ to } h_j \\ 0 & \text{otherwise} \end{cases}$$

All other variables defined in this section (except subscripts) adhere to the following convention: Upper-case English letters denote values known at compile time (e.g., because they are constants or are based on tree topology). Upper-case Greek letters denote values that must be collected directly during the course of a run, because their values may change (e.g., the size of each object). Lower-case Greek letters denote values that are derived from other values during the run (e.g., the total cost of a given configuration).

7.2.2 COMPILE-TIME VALUES

The following values are known before a run begins.

The maximum allowable spending rate is $P_{\max}$ dollars per second. The distance from h_j to h_k (through the tree) is D_{jk} hops. Note that $\forall j, k : D_{jk} = D_{kj}$. We also define:

$$T_{jk\ell} = \begin{cases} 1 & \text{if traffic from } h_\ell \text{ bound for } h_j \text{ passes through } h_k \\ 0 & \text{otherwise} \end{cases}$$

$$N_{jk\ell} = \begin{cases} 1 & \text{if } T_{jk\ell} = 1 \wedge D_{j\ell} = D_{jk} + 1 \\ 0 & \text{otherwise} \end{cases}$$

That is, $N_{jk\ell} = 1$ iff h_k is the next hop for traffic leaving h_ℓ bound for h_j.

We define P_{jk} to be the price for sending data from h_j to h_k in dollars per byte. If h_j and h_k are not neighbors, no messages will be sent between them, so the value for P_{jk} has no effect and need not be set. Due to asymmetric traffic billing, it is possible that $P_{jk} \neq P_{kj}$.

7.2.3 RUN-TIME VALUES EXPLICITLY MEASURED

The following values must be directly measured, collected, and maintained during the run.

The size of each object o_i is Γ_i bytes. The number of requests for o_i submitted by clients to h_j each second is Λ_{ij}.

7.2.4 RUN-TIME VALUES DERIVED FROM OTHER VALUES

The following values are computed during the run.

We define λ_{ijk} as the number of requests for o_i that would actually leave h_k each second if the object were hosted at h_j. Note that λ_{ijk} differs from Λ_{ik} in that λ_{ijk} includes requests that would be be forwarded through h_k (but were not initiated there), and λ_{ijk} also takes into account the effects of read clustering. We define the maximum such rate to be $\lambda_{\max}$, which is the rate that would be seen if a new request for an object is forwarded by a proxy as soon as that proxy receives the response to the previously forwarded request for

that object. We assume this rate is the same for all objects across all proxies. It may be necessary to adjust $\lambda_{\max}$ at run time. We can calculate λ_{ijk} as follows:

$$\lambda_{ijk} = \min\left(\lambda_{\max}, \Lambda_{ik} + \sum_{\ell=1}^{m} N_{jk\ell}\lambda_{ij\ell}\right)$$

That is, λ_{ijk} is the sum of the request rate from h_k itself (i.e., Λ_{ik}) and the actual request rate issued from all h_k's neighbors whose traffic for o_i passes through h_k (i.e., $\sum_{\ell=1}^{m} N_{jk\ell}\lambda_{ij\ell}$), up to a maximum of $\lambda_{\max}$ requests per second.

We define the total cost κ for a given placement configuration (i.e., the cost for all traffic for all objects) as the sum of the costs for each o_i, based on where it is hosted. The cost for an object is the sum of the per-link charges for requests for that object, over all links. We can calculate κ as follows:

$$\begin{aligned}
\kappa &= \text{(total cost for traffic for all objects)} \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij}\,\text{(total cost for traffic for } o_i\text{)} \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij}\sum_{k=1}^{m}\text{(cost of traffic } \lambda_{ijk} \text{ across its next hop toward } h_j\text{)} \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij}\sum_{k=1}^{m}\lambda_{ijk}\Gamma_i\sum_{\ell=1}^{m} N_{j\ell k}P_{k\ell}
\end{aligned}$$

We can compute the average latency τ (in hops per request) over the whole system using:

$$\begin{aligned}
\tau &= \frac{\text{(total distance each second over all requests)}}{\text{(total requests each second)}} \\
&= \frac{\sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij}\sum_{k=1}^{m} D_{jk}\Lambda_{ik}}{\sum_{i=1}^{n}\sum_{j=1}^{m}\Lambda_{ij}}
\end{aligned}$$

7.2.5 BILP

We would like to minimize the average latency τ, but to arrive at the canonical form, we must change the expression to a maximization:

$$\text{maximize} \quad -\tau$$

The constraints on the BILP are as follows. Each object must be hosted at exactly one proxy.

$$\forall o_i : \sum_{j=1}^{m} x_{ij} = 1$$

And we must also keep the total cost per second κ below $P_{\max}$.

$$\kappa \leq P_{\max}$$

7.2.6 OPTIONAL CONSTRAINTS

As defined above, our BILP depends only on latency and the cost of traffic between proxies. Below are some additional constraints that may be useful.

7.2.6.1 CAPACITY

We could define the compile-time, constant capacity C_j to be the maximum number of objects that can be hosted at h_j. Then we can add the constraints:

$$\forall h_j : \sum_{i=1}^{n} x_{ij} \leq C_j$$

7.2.6.2 OBJECT RESTRICTIONS

We might want to place specific restrictions on which objects can be hosted at which proxies (e.g., for objects that should only be hosted on certain proxies, perhaps for legal

reasons). We can do so by defining the following values that are directly measured at run-time:

$$\Phi_{ij} = \begin{cases} 1 & \text{if } h_j \text{ is forbidden to host } o_i \\ 0 & \text{otherwise} \end{cases}$$

Then, we must add the constraints that:

$$\forall i, j : x_{ij} + \Phi_{ij} \leq 1$$

That is, at most one of the following must be true: (i) o_i is moved to h_j and (ii) h_j is forbidden to host o_i.

7.2.6.3 HOSTING PRICE

Define the compile-time, constant price S_j to be the number of dollars per byte-second that it costs to host objects at h_j. Then we can define κ' to be κ plus the hosting costs

$$\kappa' = \kappa + \sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij} S_j \Gamma_i$$

and replace the constraint on κ with

$$\kappa' \leq P_{\max}$$

7.2.7 ASSUMPTIONS AND LIMITATIONS

We have not implemented this global, unrestricted BILP. Having a single agent to which all proxies must report all traffic about all objects may simply not be practical. Furthermore, the size of the BILP (i.e., the number of variables) is equal to the number of objects times the number of proxies, so a global solution is unlikely to scale to very large

installations of WACCO, even if the average number of objects per proxy remains constant.

There are some other, more low-level limitations of this approach, as we have presented it. First, it is unclear how to best arrive at a value for $\lambda_{\max}$—and whether it should be a global constant or a separate value computed per object, per placement, or per proxy. Second, in computing κ, we assumed Γ_i was the number of bytes per request, though caching may reduce that value in practice. Finally, in computing τ, we use distance D_{jk} as an estimate of the number of hops per request, though read merging should lower the average hop count for sufficiently busy objects.

7.3 LOCAL, PER-PROXY BILP

We now present a second strategy for seeking optimal placement for all objects in WACCO. Like the previous strategy, it uses a binary integer linear program (BILP) to perform the computation. But in this iteration, the migration decisions are made independently by each proxy for the objects it hosts—i.e., we use a local, per-proxy BILP instead of a global one. Specifically, the proxy can use the BILP to decide, for each object it currently hosts, whether to keep that object or migrate it to a neighbor, such that the overall latency is minimized and the cost to host and serve all the objects remains below some threshold. Section 7.3.6 defines some other constraints that could optionally be applied to further restrict object placement. Section 7.3.7 describes the assumptions and limitations of this approach.

Here we reset our BILP notation and variable names to avoid confusion with those from the global BILP. We define all those needed for the local BILP below.

7.3.1 DECISION VARIABLES

Each proxy will maintain and use its own BILP, which it will solve periodically to determine which objects to migrate and where. We will define all our notation relative to a single proxy, for its own BILP and from its own point of view, and each proxy will thus

maintain its own, independent copies of all the variables defined below. The proxy running the BILP is denoted as h_0. The set of objects hosted at h_0 (just before running the BILP) is $\Omega = \{o_i : i \in \{1, 2, 3, \ldots n\}\}$, and the set of h_0's neighbors is $\{h_j : j \in \{0, 1, 2, \ldots m\}\}$. Note that for convenience h_0 is considered its own neighbor.

The decision variables are

$$x_{ij} = \begin{cases} 1 & \text{if we decide to migrate } o_i \text{ to } h_j \\ 0 & \text{otherwise} \end{cases}$$

All other variables defined in this section (except subscripts) adhere to same conventions used in the last section: Upper-case English letters denote values known at compile time (e.g., because they are constants or are based on tree topology). Upper-case Greek letters denote values that must be collected directly during the course of a run, because their values may change (e.g., the size of each object). Lower-case Greek letters denote values that are derived from other values during the run (e.g., the predicted latency savings for a given configuration).

7.3.2 COMPILE-TIME VALUES

The following values are known before a run begins.

The maximum allowable spending rate is $P_{\max}$ dollars per second. That is, $P_{\max}$ is the maximum amount that should be incurred in serving requests to all objects o_i hosted at h_0. We define P_{jk} to be the price for sending data from h_j to h_k in dollars per byte. Due to asymmetric traffic billing, it is possible that $P_{jk} \neq P_{kj}$, and $P_{jj} = 0$ always.

We also define H'_j to be the set of all proxies "hidden" behind h_j, from the point of view of h_0. That is, H'_j is the set of proxies whose traffic from h_0 passes through h_j (excluding h_j itself). Note that H'_0 is empty, since no proxies are hidden behind h_0.

7.3.3 RUN-TIME VALUES EXPLICITLY MEASURED

The following values must be directly measured, collected, and maintained during the run.

The size of each object o_i is Γ_i bytes (and $\Gamma_i > 0$ always). The number of requests for o_i submitted each second by clients to h_j is Υ_{ij}. The number of requests for o_i submitted each second by clients to members of H'_j is Υ'_{ij}.

Similarly, we define Λ_{ij} as the number of requests per second actually arriving at h_0 over the network from h_j (where $\Lambda_{i0} = 0$, since h_0 does not send itself messages over the network) and Λ'_{ij} as the number of requests per second arriving at h_j from members of H'_j. Note that Λ_{ij} differs from Υ_{ij} (and Λ'_{ij} from Υ'_{ij}) in that the former gives the total number of requests actually sent over the network (after read clustering), while the latter gives a total number of queries issued by clients. One implication of this distinction is that it is possible to have $\Lambda_{ij} > \Upsilon_{ij}$, since Λ_{ij} includes requests forwarded through h_j that were not issued there (i.e., requests from Λ'_{ij}).

We also track the cost Π'_{ij} incurred for all the traffic for Υ'_{ij} except that portion of the traffic passing along the link directly between h_0 and h_j.

7.3.4 RUN-TIME VALUES DERIVED FROM OTHER VALUES

The following values are computed during the run.

We define λ_{ij} as the number of requests for o_i that would actually leave h_0 each second if the object were hosted at h_j. We define the maximum such rate to be $\lambda_{\max}$, which is the rate that would be seen if h_0 forwarded a new request for an object as soon as it received the response to the previously forwarded request for that object. For now, we assume this rate is the same for all objects across all proxies (an assumption which we withdraw in Section 7.4); it may be necessary to adjust $\lambda_{\max}$ at run time. We can calculate λ_{ij} as follows:

$$\lambda_{ij} = \begin{cases} 0 & \text{if } j = 0 \\ \min\left(\lambda_{\max}, \left(\sum_{k=1}^{m} \Lambda_{ik}\right) - \Lambda_{ij} + \Upsilon_{i0}\right) & \text{otherwise} \end{cases}$$

That is, λ_{ij} is the sum of the request rates seen from all the neighbors of h_0 (except h_j itself) and the rate of requests being issued to h_0 by its clients, up to a maximum of $\lambda_{\max}$ requests per second.

We define the cost κ_{ij} to be the total cost of serving all traffic for o_i if it were placed at h_j. It can be computed as the cost of all traffic from h_0's neighbors to h_0, minus the cost of the traffic from h_j to h_0, plus the cost of the new traffic from h_0 to h_j, plus the other costs Π'_{ik}:

$$\kappa_{ij} = \left(\sum_{k=1}^{m} \Lambda_{ik} \Gamma_i P_{0k}\right) - \Lambda_{ij} \Gamma_i P_{0j} + \lambda_{ij} \Gamma_i P_{j0} + \sum_{k=1}^{m} \Pi'_{ik}$$

Here we are counting the cost of the response traffic as opposed to the request traffic, since the requests themselves are small, but the responses can contain the entire object and thus be much larger. The equation for κ_{ij} above also assumes (though this assumption, too, is withdrawn in Section 7.4) that Γ_i—which is a value in bytes—can be used as the number of bytes per request, which is what we need in the equation. That is, it assumes the entire object is sent back with each response, and so we are ignoring the effects of caching in reducing network costs. Note that the cost of not migrating the object is

$$\kappa_{i0} = \sum_{k=1}^{m} \Lambda_{ik} \Gamma_i P_{0k} + \sum_{k=1}^{m} \Pi'_{ik}$$

since $\Lambda_{i0} = 0$ and $\lambda_{i0} = 0$.

We can then define the total cost κ for a given placement configuration (i.e., the cost for all traffic for all objects) as:

$$\kappa = \sum_{i=1}^{n} \sum_{j=0}^{m} x_{ij} \kappa_{ij}$$

We will use this total κ in the BILP constraints below, but it is interesting to look at the cost difference incurred when moving an object, as a sanity check. For example, a migration of o_i from h_0 to h_j produces a drop in the cost for o_i's traffic only when

$$\begin{aligned}
\kappa_{ij} &< \kappa_{i0} \\
\left(\sum_{k=1}^{m} \Lambda_{ik}\Gamma_i P_{0k}\right) - \Lambda_{ij}\Gamma_i P_{0j} + \lambda_{ij}\Gamma_i P_{j0} + \sum_{k=1}^{m} \Pi'_{ik} &< \sum_{k=1}^{m} \Lambda_{ik}\Gamma_i P_{0k} + \sum_{k=1}^{m} \Pi'_{ik} \\
-\Lambda_{ij}\Gamma_i P_{0j} + \lambda_{ij}\Gamma_i P_{j0} &< 0 \\
\lambda_{ij}\Gamma_i P_{j0} &< \Lambda_{ij}\Gamma_i P_{0j} \\
\lambda_{ij} P_{j0} &< \Lambda_{ij} P_{0j} \qquad (7.1)
\end{aligned}$$

which fits with our intuition—there is only a cost savings in moving o_i to h_j when the cost of responding to the traffic h_0 would send to h_j is less than the cost already being paid for h_0 to respond to traffic from h_j.

We define the latency savings of a given decision configuration to be τ hops per second. Since a migration can only increase or decrease the hop count for any request by one hop, it suffices to count the number of requests per second that get quicker, then subtract the number of requests per second that get slower.

$$\begin{aligned}
\tau &= (\text{hops saved} \times \text{request rate}) - (\text{hops added} \times \text{request rate}) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij}\left(\left(\Upsilon_{ij} + \Upsilon'_{ij}\right) - \left(\left(\sum_{k=0}^{m} \Upsilon_{ik} + \Upsilon'_{ik}\right) - \Upsilon_{ij} - \Upsilon'_{ij}\right)\right) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij}\left(2\left(\Upsilon_{ij} + \Upsilon'_{ij}\right) - \left(\sum_{k=0}^{m} \Upsilon_{ik} + \Upsilon'_{ik}\right)\right)
\end{aligned}$$

We are careful to ignore the case when $j = 0$, because in that case the object is not migrated, so there is no change in latency. Notice also that, for a particular o_i being

migrated to h_j, the latency savings is positive only when

$$\begin{aligned} 2\left(\Upsilon_{ij} + \Upsilon'_{ij}\right) &> \sum_{k=0}^{m} \Upsilon_{ik} + \Upsilon'_{ik} \\ \Upsilon_{ij} + \Upsilon'_{ij} &> \frac{1}{2} \sum_{k=0}^{m} \Upsilon_{ik} + \Upsilon'_{ik} \end{aligned} \tag{7.2}$$

That is, there is a positive savings (in latency) only when the number of requests per second coming from the direction of h_j is more than half of—i.e., is a majority of—the total number of requests per second, which again fits with our intuition. In fact, the idea that objects should move to a neighbor only if a majority of requests for the objects come from that neighbor's direction is exactly what we used in MIGRATE-LOAD-ONLY, our original, pre-BILP migration strategy. This local BILP (ignoring the constraints and adjustments in Section 7.3.6 and Section 7.4), given a large enough cost budget, behaves like our original algorithm, with a migration threshold of just over 50%.

7.3.5 BILP

We would like the BILP to maximize the latency savings τ:

$$\text{maximize} \quad \tau$$

The constraints on the BILP are as follows. Each object must be hosted at exactly one proxy.

$$\forall o_i : \sum_{j=0}^{m} x_{ij} = 1$$

And we must also keep the total cost per second κ below $P_{\max}$.

$$\kappa \leq P_{\max}$$

7.3.6 OPTIONAL CONSTRAINTS

As defined above, our BILP depends only on latency and the cost of traffic between proxies. Below are some additional constraints that may be useful.

7.3.6.1 CAPACITY

We could define the compile-time, constant capacity limit C_j to be the maximum number of objects that should be hosted at h_j. Neighbors of h_0 can transmit during the run their current object counts Σ_j. We can add the constraint that

$$\forall h_j : \sum_{i=1}^{n} x_{ij} \leq C_j - \Sigma_j$$

Because every proxy is running the BILP independently, it is important to decide whether C_j will be a hard or a soft limit. If it is a hard limit, then h_j will simply not accept new objects once $\Sigma_j = C_j$. If it is a soft limit, then it could potentially be violated—e.g., if h_0 has enough capacity to host only one additional object, but two of its neighbors echo simultaneously decide to migrate one object to it, then h_0 will exceed its capacity C_0. By always adding the constraint

$$\sum_{i=1}^{n} x_{i0} \leq C_0$$

we guarantee that C_0 will be satisfied after the BILP is run.

Additionally, if h_0 has some neighbors that are at capacity (i.e., for which $\Sigma_j \geq C_j$), it may be wise to add a constraint that guarantees that h_0 is not itself at capacity, since we want the neighbors to have at least one place to migrate objects if necessary. First, we can define a new variable ϕ_j that indicates whether h_j is at capacity. That is,

$$\phi_j = \begin{cases} 1 & \text{if } \Sigma_j \geq C_j \\ 0 & \text{otherwise} \end{cases}$$

Then, replacing the constraint above with

$$\sum_{i=1}^{n} x_{i0} \leq C_0 - \sum_{j=1}^{m} \phi_j$$

will leave enough room at h_0 for each full neighbor to move at least one object to h_0. We believe this property will help prevent the development of over-saturated parts of the tree, in which many neighboring full proxies pass objects among themselves which no proxy has the capacity to keep, because proxies at the edge of such an area will always have neighbors with capacity to accept some of their objects, causing the perimeter of the over-saturated area to shrink.

The only case in which the above constraint should not be applied is when

$$\text{total objects} > \text{total available space for objects}$$

$$\text{total objects} > (\text{total space at } h_0) + (\text{total space at neighbors})$$

$$n > \left(C_0 - \sum_{j=1}^{m} \phi_j\right) + \left(\sum_{j=1}^{m} (C_j - \Sigma_j)\right)$$

in which case we should drop the constraint, leaving h_0 over-saturated.

7.3.6.2 OBJECT RESTRICTIONS

We might want to place specific restrictions on which objects can be hosted at which proxies (e.g., for objects that should only be hosted on certain proxies, perhaps again for legal reasons). We can do so by defining the following values that are directly measured at run-time:[1]

[1]As we explain in Section 7.4.2, object restrictions are known at compile time in our experiments, but in general administrators may not know the restrictions or even the set of objects at compile time.

$$\Phi_{ij} = \begin{cases} 1 & \text{if } h_j \text{ is forbidden to host } o_i \\ 0 & \text{otherwise} \end{cases}$$

Then, we must add the constraints that:

$$\forall i, j : x_{ij} + \Phi_{ij} \leq 1$$

That is, at most one of the following must be true: (i) o_i is moved to h_j and (ii) h_j is forbidden to host o_i.

7.3.6.3 HOSTING PRICE

Define the compile-time, constant price S_j to be the number of dollars per byte-second that it costs to host objects at h_j. Then we can define κ' to be κ plus the hosting costs

$$\kappa' = \kappa + \sum_{i=1}^{n} \sum_{j=0}^{m} x_{ij} S_j \Gamma_i$$

and replace the constraint on κ with

$$\kappa' \leq P_{\max}$$

7.3.6.4 MAXIMUM FRACTION MOVED

It may be helpful to reduce the rate at which objects can migrate among proxies by adding a constraint limiting the fraction of objects each proxy can move in a single BILP iteration. To that effect, we define this fraction as the compile-time constant $F_{\max} \in (0, 1]$. Then, the maximum amount of objects a proxy can choose to migrate away from itself is $\lfloor nF_{\max} \rfloor$. We can then add the constraint:

$$\sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij} \leq \lfloor nF_{\max} \rfloor$$

7.3.6.5 MINIMUM OBJECT ACTIVITY

Migrating infrequently requested objects may be inefficient and can waste resources. For example, if an object is requested only rarely during a certain BILP iteration (and always by a particular neighbor h_j of h_0), h_0 may decide to move the object to h_j. We know the move would reduce overall latency if and only if Equation 7.2 is satisfied. In this case, $\forall k : \Upsilon'_{ik} = 0$ and $\forall k \neq j : \Upsilon_{ik} = 0$. Putting those into Equation 7.2, we get

$$\Upsilon_{ij} > \frac{1}{2}\Upsilon_{ij}$$

which is true, since we know that $\Upsilon_{ij} > 0$. We also know that moving the object would reduce costs if Equation 7.1 is satisfied. In this case, $\lambda_{ij} = 0$ and $\Lambda_{ij} > 0$, so the migration will save money as long as $P_{0j} > 0$. The problem is that, while the BILP does not consider the traffic cost of the migration itself, the migration will indeed result in some traffic and thus some cost. For busy objects, this cost should be negligible, but for infrequently used objects, it can dominate the cost of serving the requests. Aside from cost, migrating infrequently used objects also wastes precious network bandwidth that would be better allocated for popular objects.

We can define a compile-time constant $U_{\min}$ as the minimum number of requests per second an object must receive in order to be considered for migration in the current BILP iteration. Recall that each proxy will solve the BILP periodically, based on the most recent data, so classifying an object as too infrequently accessed to migrate has no lasting effects, just as objects that are popular during a particular BILP iteration must remain busy in

order to continue to be considered for migration. So, for any object whose

$$\sum_{j=0}^{m} \Upsilon_{ij} + \Upsilon'_{ij} < U_{\min}$$

we can add the constraint

$$x_{i0} = 1$$

to ensure the object is kept at h_0. Note that these constraints may conflict with the host capacity constraints described in Section 7.3.6.1 if there are many unpopular objects.

7.3.7 BENEFITS, LIMITATIONS, AND ASSUMPTIONS

Switching from a global to a local strategy carries an inherent tradeoff. With locality comes scalability—as long as the number of proxies rises along with the number of objects, we expect the local BILP to continue to work well. But the nature of solving the BILP locally at each proxy is that the overall placement of objects throughout the tree may not be globally optimal and may be subject to local minima and maxima. For example, the globally optimal location for an object may be two hops away, just beyond a neighboring proxy that has reached its full capacity.

We have given no insight into how to best arrive at a value for $\lambda_{\max}$, and, though we assumed above that it would be a the same for all objects on all proxies, there are more intuitive ways. Furthermore, in computing κ, we assumed Γ_i (the size of o_i) was also the number of bytes per request needed to serve the object, though caching may reduce that value in practice. Finally, our use of $P_{\max}$ is unlikely to conform to the way an organization actually makes its budget—i.e., with each proxy having an equal share. We address these three limitations below.

The main assumption we are making when migrating using the BILP is that there will be some temporal locality to the request load. That is, we expect that the statistics gathered about the load and used to solve the BILP will also give a reasonably accurate

prediction of what the load will be like after the BILP has run. We make the same assumption in MIGRATE-LOAD-ONLY but to a lesser degree; since MIGRATE-LOAD-ONLY updates its statistics and makes migration decisions with each request, it can respond more quickly to changing request profiles.

7.4 EXPERIMENTAL SETUP

Our experiments use the local BILP defined in Section 7.3, with some changes or specific configuration choices, detailed below. Unless otherwise noted, we enabled all the optional constraints given in Section 7.3.6.

7.4.1 CONFIGURATION AND ENVIRONMENT

All experiments in this chapter used the Akamai data set described in Chapter 5 with $u = 0.01$ and load factor 0.1 (with jobs again configured to finish in 45 minutes each). We use the same hardware as in previous chapters to run the proxies. Each proxy uses Gurobi's [10] Java API to solve the BILP every 5 seconds.

We use a hybrid tree in which about half (40/76 = 53%) of all proxies are marked as public. All public proxies are part of a single, contiguous subtree that does not include the root. Prices for traffic and storage are user-configurable per-link and per-proxy. The price we chose for our experiments—based very loosely on the pricing for Amazon EC2 [2]—are as follows: Storage at public proxies costs 0.125 $/(GB $\cdot$ month), while storage at private proxies is free. Traffic between two (different) public proxies costs 0.01 $/GB, and traffic from a public proxy to a private proxy costs 0.09 $/GB. Traffic from a private proxy to any other proxy is free.

7.4.2 FORBIDDEN LOCATIONS

We marked certain objects as forbidden on certain proxies using the following method. First, we chose 1% of objects uniformly at random to be in a set that will have forbidden placements. Then, for each of the objects in that set, we marked each proxy as forbidden with probability 1/8, so that in expectation, the object would be forbidden at 9–10 proxies. Obviously, some objects were forbidden at more or fewer proxies, and in particular, some objects in the set may not be forbidden anywhere, due to the random choice. The root of the tree was excluded from the process, so that no object was forbidden to be placed there. Each experiment used the same set of forbidden placements—i.e., placements were computed only once, not once per experiment.

7.4.3 MIGRATION LIMITS

We limited the rate of migration using $F_{\max} = 0.05$, meaning that at most 5% of a proxy's objects could be migrated away in a single BILP iteration. We also set $U_{\min} = 0.5$, meaning that an object must receive at least one request every two seconds (on average, during the window over which the BILP is being computed) in order be considered for migration. We disabled host capacity constraints in favor of this restriction.

7.4.4 CACHING

As we mentioned in Section 7.3.7, we used Γ_i (the size of each object in bytes) as the number of bytes per request needed to serve o_i to clients. But this use ignored the effects of caching in reducing the number of bytes actually sent over the network.

In our implementation, we track the historical byte count actual sent over the network (on average) per object. This value, which we call Γ'_i, directly accounts for the effects of caching on reducing network traffic. We must still use Γ_i when computing hosting costs, though.

7.4.5 SETTING MAXIMUM REQUEST RATE λ_{MAX}

Rather than trying to measure $\lambda_{\max}$ directly or find a suitable value that would suffice for all objects and all proxies, we used the round trip time (RTT) between proxies to give us the value. That is, we reasoned that the RTT between h_j and h_0 is a good lower bound on the amount of time h_j must wait between requests to h_0 (for the same object). This value is a lower bound because h_j must, at a minimum, wait for its previous message to return, and the limit is good because we expect network communication to dominate the total request time. From the RTT we can compute the maximum number of requests per second $\lambda_{\max}$ between each pair of proxies, and our BILP solver uses these new values instead of a global constant.

7.4.5.1 DIVIDING TOTAL COST

Our local BILP construction assumed that the total budget for the whole network is divided equally among all proxies. That is, it assumes that an organization running WACCO, knowing the total budget $P'_{\max}$ for the entire network, computes the $P_{\max}$ each proxy will use by:

$$P_{\max} = P'_{\max}/m$$

One can imagine many alternate strategies for partitioning $P'_{\max}$ such that each proxy can use a different $P_{\max}$.

We leave the partitioning strategy as a configuration choice for WACCO users by first allowing them to set $P'_{\max}$. Then, each proxy can be configured to have a certain number of "shares" of the total. To determine the $P_{\max}$ that a given proxy will use, we divide $P'_{\max}$ by the total number of shares across all proxies, then multiply by the shares assigned to the given proxy.

In our experiments, we have chosen to allocate shares as follows: Public proxies with no private neighbor have a number of shares equal to the number of their neighbors (including

their parents and themselves). Public proxies with private neighbors (and private proxies with public neighbors) follow the same rule but receive one extra share per private (public) neighbor for being on the border.[2] Private proxies with only private neighbors receive a single share.

One can imagine other, more adaptable ways to divide the total cost $P'_{\max}$ among the m proxies. For example, instead of allocating to each proxy a fixed share of the total budget, to be used throughout the entire experiment, we could continually adjust these shares at run time, based on feedback from the proxies. That is, proxies that can solve their BILP and still stay well below their allocated budget (perhaps for several consecutive iterations) could choose to reduce their share of $P'_{\max}$, relinquishing the unused portion into a pool. Other proxies that are struggling with insufficient budgets could take from this pool in order to better provision themselves. In this way, the amount of shares of $P'_{\max}$ held by each proxy could shift with changing demand, perhaps increasing performance by allowing the WACCO to spend more of its budget.

7.4.6 INFEASIBILITY

Recall that the high-level goal of the BILP is to give the best latency while staying below some budget. But whether due to insufficient budgets or unexpected surges in demand, there will inevitably be a point when WACCO is unable to solve the BILP because every possible migration choice will exceed $P_{\max}$. In that event, we alter the BILP such that it abandons any interest in latency and switches to minimizing price (removing the old constraint on price). This new BILP configuration is guaranteed to be feasible, because it can always choose to keep all the objects at h_0 if necessary, though it will, of course, choose whatever placement best reduces cost.

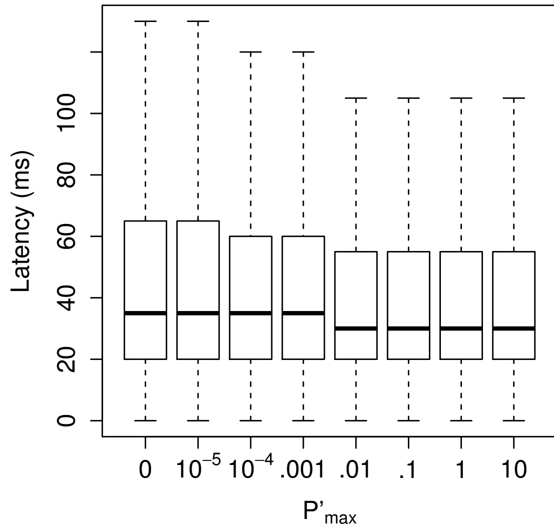
[2] A public proxy can only have at most one private neighbor (its parent) because there is only one, contiguous private region, which includes the root.

7.5 EVALUATION RESULTS

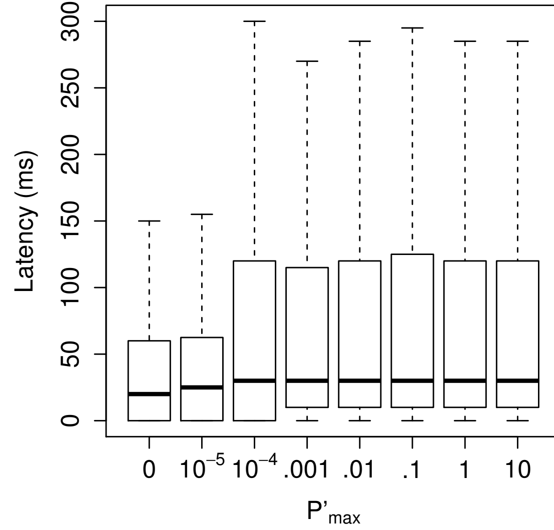
In our experiments we vary $P'_{\max}$ to see what effect it has on the total cost of the run and the performance of the system. We varied $P'_{\max}$ from 10^{-5} \$/s to 10 \$/s, by factors of ten. We also ran with $P'_{\max} = 0$.

The main results of our experiments appear in Figure 7.1. Figure 7.1(a) and Figure 7.1(b) show box-and-whiskers plots of the read and write request latency (respectively) for each run. The whiskers are extend to cover all points that are within 1.5 times the interquartile range of the box, and the outliers are omitted. Figure 7.1(a) shows a downward trend in the latency as $P'_{\max}$ increases, which we expect, since higher $P'_{\max}$ means the BILP is more free to place objects for lower latency without being constrained by price. At the 75th percentile, read latency drops by 10 ms, or about 17%, as $P'_{\max}$ increases from 10^{-5} to 0.01. This $P'_{\max}$ increase results in a decrease of nearly 27,000 seconds (7.5 hours) of total client request latency over the course of the over 9 million requests served during the runs. Figure 7.1(b) shows that for writes the opposite is true: as $P'_{\max}$ increases, latency actually increases. This also fits our intuition: recall from Chapter 5 that write requests are only served from dominant proxies. So, if objects migrate away from these proxies (and increases in $P'_{\max}$ give proxies more options, in general), then write operations will require more hops and incur more latency. Note also that since $u = 0.01$, there are many more read operations than write operations, and so write operations themselves are unlikely to have much influence on object placement.

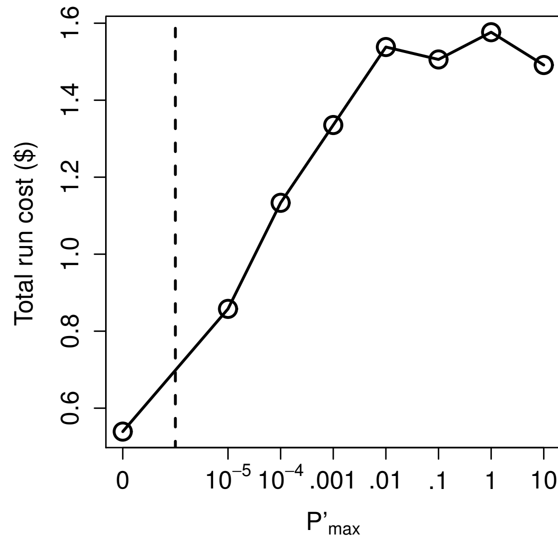
Figure 7.1(c) shows how the total run cost varies between runs. The x-axis is log-scale, except for the leftmost point ($P'_{\max} = 0$). Notice that as $P'_{\max}$ increases, total run cost increases, as expected, but then it plateaus for the final four runs, because after a certain point (around $P'_{\max} = 0.01$) the BILP has a high enough budget that price is effectively not a factor. This property is also visible in Figure 7.1(a) and Figure 7.1(b), as well as the remaining graphs in this chapter.



(a) A box plot of read latencies for various values of $P'_{\max}$. The whiskers cover all points that are within 1.5 times the interquartile range of the box, and outliers are not shown.



(b) A box plot of write latencies for various values of $P'_{\max}$. The whiskers cover all points that are within 1.5 times the interquartile range of the box, and outliers are not shown.



(c) Total run cost (including all traffic and storage costs) for runs with various values of $P'_{\max}$. The x-axis is log-scale, except for the leftmost point, which represents $P'_{\max} = 0$.

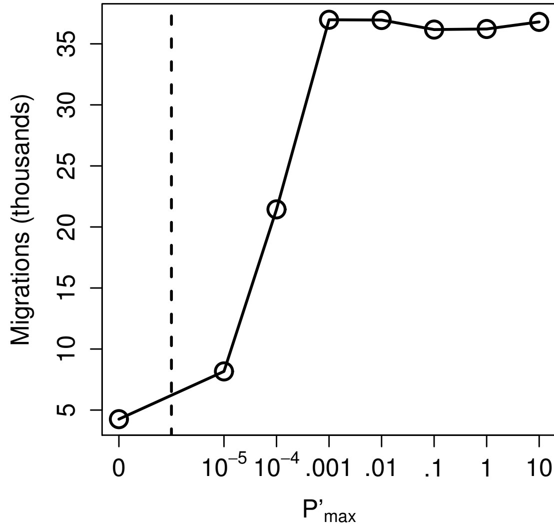
Figure 7.1: Impact on latency and run cost of varying $P'_{\max}$, with $u = 0.01$.

Figure 7.2 shows why increasing $P'_{\max}$ reduces latency: Larger budgets allow the BILP solver greater latitude in placing objects, resulting in more total migrations throughout the run (Figure 7.2(a)). Better object placement means requests need not travel as far to reach the objects (Figure 7.2(b)), resulting in fewer messages per request, on average (Figure 7.2(c)). Each of these values plateaus after a certain $P'_{\max}$ threshold (again around $P'_{\max} = 0.01$) has been reached, as in Figure 7.1. We have argued that the plateau occurs because, beyond the threshold, each BILP has sufficient budget to find desirable solutions. Figure 7.2(d) confirms this argument: after $P'_{\max} = 0.01$, the percentage of BILP iterations that is infeasible (and so must try again, having abandoned interest in latency, see Section 7.4.6) drops to 0.

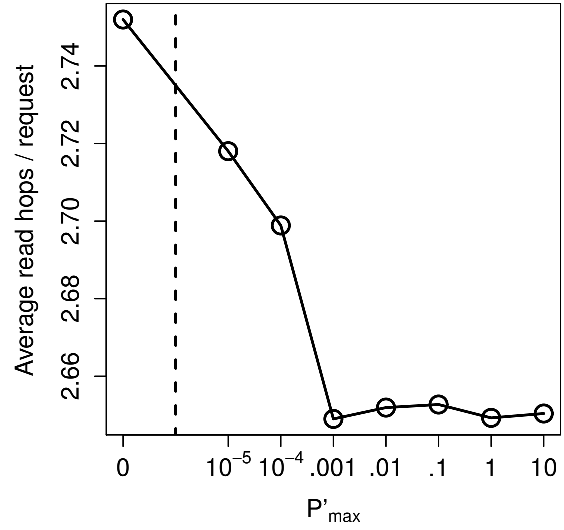
In general, the problem of solving a BILP is NP-complete [45], so initially we suspected that even a local BILP might be too difficult to solve quickly and frequently during a LOKO run. However, our experiments show that the time needed to solve each BILP was quite reasonable: Over 99.994% of all BILP solutions (i.e., all but 18 of the 326,571 across all proxies and all runs) took 100 ms or less to find, and no solution took more than 731 ms.

7.6 CONCLUSION

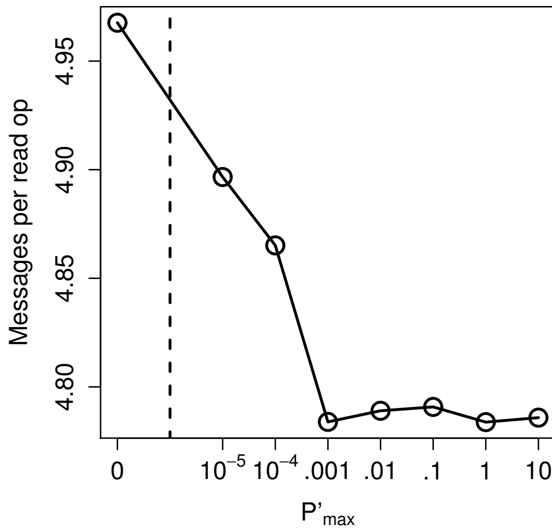
A BILP-based migration strategy gives WACCO more flexibility by allowing it to decide when and where to migrate objects based not just on latency but also on budget, host capacity limits, forbidden object placements, minimum activity, etc. Users of WACCO can decide how to allocate their budget for their specific tree of proxies (on a per-link and per-proxy basis). With larger budgets, WACCO has more latitude in choosing object locations and can thus arrange them to further reduce load the total number of messages (and hops) for read requests.



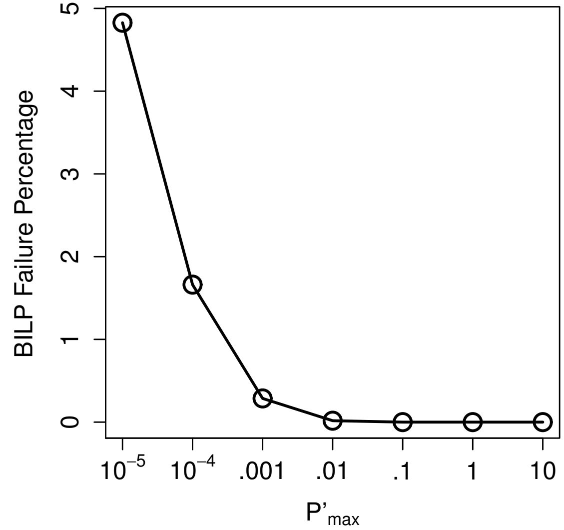
(a) Total number of migrations for runs with various values of $P'_{\max}$. The x-axis is log-scale, except for the leftmost point, which represents $P'_{\max} = 0$.



(b) Average number of hops per read request for runs with various values of $P'_{\max}$. The x-axis is log-scale, except for the leftmost point, which represents $P'_{\max} = 0$.



(c) Average number of messages per operation for runs with various values of $P'_{\max}$. The x-axis is log-scale, except for the leftmost point, which represents $P'_{\max} = 0$.



(d) Percentage of BILP iterations that failed to find a solution when constrained by budget, for runs with various values of $P'_{\max}$. The x-axis is log-scale.

Figure 7.2: Impact on various measurements of varying $P'_{\max}$, with $u = 0.01$.

Chapter 8: CONCLUSION

This dissertation describes the design and evaluation of WACCO, a system for implementing object-based services that need to support both frequent updates and widespread, massive read demand with strong consistency. A contribution of our work is a novel type of strong consistency dubbed *cluster consistency*, which implies both sequential consistency and rapid update propagation and, we argue, can be useful in a range of future networked applications. We used WACCO to implement a service called LOKO that supports keyspace objects and, in one style of usage, could roughly encompass the current duties of DNSSEC. We evaluated LOKO using two real-world data sets, the first of DNS queries to Akamai and the second of NFS traffic on UNC's campus. Our evaluation shows that LOKO provides good responsiveness and can scale to large demand, if the objects being served are not overly large and have sufficiently distributed requests. Through our evaluation, we also document the importance of object migration and read pausing (and hence cluster consistency) to the performance LOKO achieves. We also evaluated a second migration strategy that allows LOKO users to prioritize other demands such as budgets ahead of performance, if needed.

BIBLIOGRAPHY

[1] Amazon DynamoDB frequently asked questions. http://aws.amazon.com/dynamodb/faqs/. Last accessed: 2015-05-28.

[2] Amazon EC2. http://aws.amazon.com/ec2/. Last accessed: 2015-06-04.

[3] Amazon S3 frequently asked questions. http://aws.amazon.com/s3/faqs/. Last accessed: 2015-05-28.

[4] Amazon SimpleDB frequently asked questions. http://aws.amazon.com/simpledb/. Last accessed: 2015-05-28.

[5] Apache Cassandra architecture overview. http://wiki.apache.org/cassandra/ArchitectureOverview. Last accessed: 2015-05-28.

[6] AT&T current network performance. http://ipnetwork.bgtmo.ip.att.net/pws/current_network_performance.shtml. Last accessed: 2011-10-09.

[7] Geobytes city distance calculator. http://www.geobytes.com/CityDistanceTool.htm. Last accessed: 2011-10-09.

[8] IP2Location. http://ip2location.com. Last accessed: 2015-06-07.

[9] Oracle NoSQL database. http://www.oracle.com/technetwork/products/nosqldb/documentation/consistency-explained-1659908.pdf. Last accessed: 2015-05-28.

[10] Gurobi optimizer reference manual. http://www.gurobi.com, 2015. Last accessed: 2015-05-28.

[11] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

[12] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[13] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS security introduction and requirements. Request for Comments: 4033, 2005.

[14] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc., second edition, 2004.

[15] M. Avvenuti and A. Vecchio. Application-level network emulation: The EmuSocket toolkit. *Journal of Network and Computer Applications*, 29(4):343–360, 2006.

[16] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, 2013.

[17] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *ACM SIGMOD International Conference on Management of Data*, pages 761–772. ACM, 2013.

[18] Y. Bartal, M. Charikar, and P. Indyk. On page migration and other relaxed task systems. *Theoretical Computer Science*, 268(1):43–66, 2001.

[19] P. A. Bernstein and S. Das. Rethinking eventual consistency. In *ACM SIGMOD International Conference on Management of Data*, pages 923–928. ACM, 2013.

[20] D. Black and D. Sleator. Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[21] E. Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[22] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In *Distributed Systems, 2nd edition*, pages 199–216. Addison-Wesley, 1993.

[23] P. Cao and C. Liu. Maintaining strong cache consistency in the World Wide Web. *IEEE Transactions on Computers*, 47(4), 1998.

[24] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *International Symposium on Computer Architecture*, pages 278–289. ACM, 2007.

[25] A. Chankhunthod, P. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *USENIX 1996 Annual Technical Conference*, 1996.

[26] X. Chen, H. Wang, S. Ren, and X. Zhang. Maintaining strong cache consistency for the Domain Name System. *IEEE Transactions on Knowledge and Data Engineering*, 19(8), 2007.

[27] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[28] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a peer-to-peer lookup service. In *1st International Workshop on Peer-to-Peer Systems*, 2002.

[29] E. Cronin, B. Filstrup, A. B. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. In *1st Workshop on Network and System Support for Games*, 2002.

[30] A. Fekete, M. F. Kaashoek, and N. Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. *Journal of the ACM*, 45(1):35–69, 1998.

[31] S. Fortune and J. Wyllie. Parallelism in random access machines. In *ACM Symposium on Theory of Computing*, pages 114–118. ACM, 1978.

[32] B. Gavish and O. R. Liu Sheng. Dynamic file migration in distributed computer systems. *Communications of the ACM*, 33(2):177–189, 1990.

[33] D. K. Gifford. Weighted voting for replicated data. In *7th ACM Symposium on Operating Systems Principles*, 1979.

[34] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, and partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002.

[35] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *23rd ACM Symposium on Operating Systems Principles*, 2011.

[36] T. Griffin and G. Wilfong. Analysis of the MED oscillation problem in BGP. In *IEEE International Conference on Network Protocols*, 2002.

[37] T. G. Griffin and G. Wilfong. An analysis of BGP convergence properties. In *ACM SIGCOMM*, 2009.

[38] A. Hac. A distributed algorithm for performance improvement through file replication, file migration, and process migration. *IEEE Transactions on Software Engineering*, 15(11):1459–1470, 1989.

[39] M. P. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1), 1986.

[40] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[41] P. Hutto and M. Ahamad. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In *International Conference on Distributed Computing Systems*, pages 302–309, 1990.

[42] A. Juels and J. Brainard. Client puzzle: A cryptographic defense against connection depletion attacks. In *ISOC Network and Distributed System Security Symposium*, 1999.

[43] A. Kangarlou, S. Shete, and J. D. Strunk. Chronicle: Capture and analysis of NFS workloads at line rate. In *13th USENIX Conference on File and Storage Technologies*, pages 345–358. USENIX Association, 2015.

[44] J. Kangasharju and K. W. Ross. A replicated architecture for the Domain Name System. In *19th IEEE INFOCOM*, 2000.

[45] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

[46] D. Kim, J. Kim, Y. Kim, H. Yoon, and I. Yeom. Mobility support in content centric networks. In *Workshop on Information-Centric Networking*, 2012.

[47] Q. Kure. Optimization of file migration in distributed systems. Technical Report UCB/CSD-88-413, EECS Department, University of California, Berkeley, 1988.

[48] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21, 1978.

[49] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), 1979.

[50] C. Lin, V. Nagarajan, and R. Gupta. Efficient sequential consistency using conditional fences. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 295–306. ACM, 2010.

[51] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, 1988.

[52] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *23rd ACM Symposium on Operating Systems Principles*, 2011.

[53] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *USENIX Conference on Networked Systems Design and Implementation*, pages 313–328. USENIX Association, 2013.

[54] B. M. Maggs, F. M. A. D. Heide, B. Vcking, and M. Westermann. Exploiting locality for data management in systems of limited bandwidth. In *IEEE Symposium on Foundations of Computer Science*, pages 284–293, 1997.

[55] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical Report TR-11-22, Computer Science Department, University of Texas at Austin, 2011.

[56] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: Providing consistency for replicated continuous applications. *IEEE Transactions on Multimedia*, 6(1), 2004.

[57] R. C. Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Department of Electrical Engineering, Stanford University, 1979.

[58] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching: Towards a new global caching architecture. *Computer Networks and ISDN Systems*, 30, 1998.

[59] D. Mosberger. Memory consistency models. *SIGOPS Operating Systems Review*, 27(1):18–26, 1993.

[60] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path splicing. In *ACM SIGCOMM*, 2008.

[61] J. Paek, K. Kim, J. P. Singh, and R. Govindan. Energy-efficient positioning for smartphone applications using cell-ID sequence matching. In *9th International Conference on Mobile Systems, Applications, and Services*, 2011.

[62] J. Pang, A. Akella, A. Shaikhy, B. Krishnamurthyz, and S. Seshan. On the responsiveness of DNS-based network control. In *Internet Measurement Conference*, 2004.

[63] V. Pappas, D. Massey, A. Terzis, and L. Zhang. A comparative study of the DNS design with DHT-based alternatives. In *25th IEEE INFOCOM*, 2006.

[64] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *IEEE International Conference on Distributed Computing Systems*, pages 101–113, 1999.

[65] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the Internet. In *ACM SIGCOMM*, 2004.

[66] M. K. Reiter and A. Samar. Quiver: Consistent object sharing for edge services. *IEEE Transactions on Parallel and Distributed Systems*, 19(7), 2008.

[67] P. Rodriguez, C. Spanner, and E. W. Biersack. Analysis of web caching architectures: Hierarchical and distributed caching. *IEEE/ACM Transactions on Networking*, 9(4), 2001.

[68] J. Ruscio, M. Heffner, and S. Varadarajan. Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

[69] C. Scheurich and M. Dubois. Dynamic page migration in multiprocessors with distributed global memory. *IEEE Transactions on Computers*, 38(8):1154–1163, 1989.

[70] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. GlobeDB: Autonomic data replication for web applications. In *14th International Conference on the World Wide Web*, 2005.

[71] T. Suen and J. Wong. Efficient task migration algorithm for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):488–499, 1992.

[72] E. Swildens, M. Cinquini, A. Chavarkar, and A. Agarwal. Automatic migration of data via a distributed computer network, 2006. US Patent 7,143,170.

[73] C. Tapus, D. Noblet, V. Grama, and J. Hickey. Mojavefs: Providing sequential consistency in a distributed objects system. In *International Symposium on Parallel and Distributed Computing*, pages 66–73. IEEE Computer Society, 2006.

[74] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *ACM Symposium on Operating Systems Principles*, pages 172–182. ACM, 1995.

[75] N. Tran, M. K. Aguilera, and M. Balakrishnan. Online migration for geo-distributed storage systems. In *USENIX Annual Technical Conference*, 2011.

[76] J. Valerio, P. Sutra, E. Rivière, and P. Felber. Evaluating the price of consistency in distributed file storage services. In *Distributed Applications and Interoperable Systems*, volume 7891 of *Lecture Notes in Computer Science*, pages 141–154. Springer Berlin Heidelberg, 2013.

[77] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[78] X. Wang and D. Wetherall. Source selectable path diversity via routing deflections. In *ACM SIGCOMM*, 2006.

[79] Y. Wu, J. Tuononen, and M. Latvala. Performance analysis of DNS with TTL value 0 as location repository in mobile Internet. In *IEEE Wireless Communications and Networking Conference*, 2007.

[80] W. Xu and J. Rexford. MIRO: Multi-path Interdomain ROuting. In *ACM SIGCOMM*, 2006.

[81] X. Yang, D. Clark, and A. W. Berger. NIRA: A new inter-domain routing architecture. *IEEE/ACM Transactions on Networking*, 15(4), 2007.

[82] X. Yang, D. Wetherall, and T. Anderson. TVA: A DoS-limiting network architecture. *IEEE/ACM Transactions on Networking*, 16(6), 2008.

[83] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. Andersen. SCION: Scalability, control, and isolation on next-generation networks. In *IEEE Symposium on Security and Privacy*, 2011.

[84] Z. Zhang, Y. Zhang, Y. C. Hu, and Z. M. Mao. iSPY: Detecting IP prefix hijacking on my own. In *ACM SIGCOMM*, 2008.