A Practical Approach to Replication of Abstract Data Objects

Joshua J. Bloch

May 1990
CMU-CS-90-133

Submitted to Carnegie Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania

This work was supported by IBM, the Transarc Corporation, and by the Defense Advanced Research Projects Agency, ARPA Order No. 4976 (Amendment 20), under contract F33615-87-C-1499, monitored by the Air Force Avionics Laboratory, Wright Aeronautical Laboratories, Wright-Patterson Air Force Base.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsoring agencies or of the United States Government.

To my mother and father.

Abstract

There is a great need for computer systems that remain available with high probability at all times. *Highly available systems* can be implemented on networks of general purpose computers by *replicating* data: storing the data redundantly at two or more of the nodes comprising the system. Some *replication protocol* is necessary to control access to the replicas. In essence, the replication protocol orchestrates the replicas to form a single distributed data object. If a replicated data object is to be used in an application where data consistency is required, the replicated object must display the same semantics as its serially accessed, single-site counterpart. It is difficult to design replication protocols that combine *one-copy serializability* with high performance.

This dissertation describes an architecture that provides efficient, easy-to-use replicated implementations for a wide variety of useful data types, including *directories, record files with secondary indices on selected fields*, and *priority queues*. The data objects display single-copy serial semantics and provide high availability and concurrency. The architecture is relatively easy to implement as it derives its recovery and concurrency control properties from the support of an underlying distributed transaction system. A fairly complete prototype implementation of the architecture was built on top of the Camelot system. Experiments were performed to evaluate its performance.

The heart of the architecture is a family of efficient replication protocols that implement a class of table-like data objects called *replicated sparse memories* or RSMs. The replication protocols are based on Gifford's *weighted voting* technique. An underlying structural property of the RSM that allows efficient implementation of all its operations is proven. Simulation results are presented that suggest RSMs are time and space efficient in a wide variety of configurations. A Markov model of the RSM is constructed and analyzed. The analysis indicates that RSMs are time and space efficient in all configurations for all random operation mixes.

This dissertation introduces the concept of *optimistic two-stage protocols*, a new technique for reducing communication costs in a broad class of distributed algorithms. The architecture makes heavy use of optimistic two-stage protocols. In particular, *optimistic timestamps* are used to speed up blind writes.

Acknowledgements

Many people assisted me in the process of writing this dissertation and performing the research that led to it. My deepest thanks go to my advisor, Alfred Spector, who over the years provided me with guidance, encouragement, perspective, and friendship.

I thank Eric Cooper, Dave Gifford and Doug Tygar for taking time out of their busy schedules to serve on my thesis committee. Special thanks are due Eric Cooper, who joined the committee on short notice, and gave my dissertation a very careful reading, in spite of the many demands on his time. Every member of my committee provided me with suggestions that substantially improved the quality of this work.

I owe special thanks to Jeffrey Eppinger. He drew many of the figures contained in this dissertation, exercising a degree of care and esthetic judgment that went well beyond the call of duty. On several occasions while Alfred was busy with Transarc, he served as a surrogate advisor, giving me much-needed encouragement as well as technical advice.

This work owes much to technical discussions with Dan Bloch, Dean Daniels, Jim Driscoll, Maurice Herlihy, John Lehoczky, Dean Rubine, and Dean Thompson, among others.

Chapters 2 and 3 are derived from a paper authored jointly with Dean Daniels and Alfred Spector [10]. Dean performed the simulations that led me to do the average-case analysis in Chapter 3. I am deeply indebted to Dean and Alfred for their permission to use this work in my thesis.

The members of the Camelot group worked many long hours to build the transaction system that allowed me to make my replication architecture more than just another paper design. The Camelot group consisted of Dean Daniels, Rich Draves, Dan Duchamp, Jeffrey Eppinger, Andy Hastings, Elliot Jaffe, Toshihiko Kato, George Michaels, Lily Mummert, Randy Pausch, Alfred Spector, Peter Stout and Dean Thompson. I would especially like to thank Peter Stout for maintaining the Camelot system almost singlehandedly during the time I studied the performance of my replication system.

Many members of the Mach group helped me during the implementation and performance measurement phases of this work, including Joe Barrera, David Black,

Jonathan Chew, Rich Sanzi, Mary Thompson, Mike Young, and especially Rich Draves, whom I bothered far too often.

I thank Dave Eckhardt for proofreading large parts of this dissertation, and Fred Schwarz for providing me with definitive answers to questions of style and usage.

I thank the CMU Department of Computer Science for providing a rich, stimulating and friendly environment in which to do research. In particular, I thank Sharon Burks for answering all my questions when no one else could, and fending off the university bureaucrats.

My time at CMU was enriched by the friendship of many people, far too many to list here. I could not have completed this thesis without their support and encouragement. I give special thanks to Cindy Cosic, who put up with me on a daily basis while I wrote this dissertation.

Finally, I thank my family for providing constant support and encouragement throughout my time in graduate school.

When a list of acknowledgements is compiled, it is almost inevitable that some people who deserve to be mentioned are forgotten. In the case of this work, the problem is exacerbated by the extreme length of time over which the work was performed, and the memory loss that typically accompanies old age. So, to all of you who should have been mentioned here but are not: thanks, and sorry I forgot you.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

There is a great need for computer systems that remain available with high probability at all times. Such systems are needed, for example, to support banking, ticket sales, and factory automation. When a computer system used for such an application fails, the company that relies on the system can suffer great monetary losses due to lost sales and productivity. The longer the system remains unavailable, the greater the losses. The failure of such a system can also result in great irritation to customers and employees.

While hardware and software failures can be reduced through careful design and testing, they cannot be eliminated using present-day technology; in order to achieve high availability, computer systems must be able to withstand failures in the underlying hardware and software. This is accomplished by building systems with redundant hardware and storing redundant copies of data. If a failure occurs in some unit, a backup unit is available to take its place.

The traditional approach to highly available system design, exemplified by Tandem's NonStop system [4], is to build special purpose hardware in which each component is duplicated and components are interconnected in such a way that failures can be masked. Data is typically stored on two *mirrored* disks that are located in close proximity to one another. Data is written to both copies, and read from only one. While this approach is fast, it has certain deficiencies: special purpose hardware is expensive and availability is limited by the fact a single physical catastrophe can destroy the whole system.

Another approach to highly available system design is to store copies of data at several sites in a loosely coupled distributed computer system. This approach is known as *replication*. Some *replication protocol* is used to direct updates to the sites and retrieve data from the sites to answer queries. In essence, the replication protocol orchestrates the replicas to form a single distributed data object. With replication, highly available systems can be constructed from low-cost, off-the-shelf computer hardware. Such systems are less susceptible to physical catastrophes, as the hardware can be geographically dispersed.

Replication has other advantages that may apply depending on the particulars of the replication protocol. Some protocols can decrease the workload on individual data servers by spreading work out among multiple sites. Some protocols can reduce the latency of certain operations by allowing clients to use readily accessible replicas of the data. Some protocols permit sites at which replicas are stored to be removed from the system for hardware maintenance or software upgrades without causing any disruption in service.

As is the case for traditional, single-site systems, the task of writing and debugging complex applications with replicated data is greatly simplified if a facility is provided for the programmer to create and manipulate complex data objects.

Many critical applications for online databases (e.g. banking, sales) require that data objects display *serial consistency* [20]: although objects may be accessed concurrently by multiple clients, clients must never be allowed to see the effects of concurrent activity. Clients may group several operations on one or more data objects to form a single *transaction* [18, 20]. Serial consistency guarantees that any concurrent execution of transactions performed by the system will be equivalent to some serial execution. If the database starts out in a consistent state and each transaction would preserve consistency if executed in isolation, serial consistency guarantees that the no transaction will ever observe the database in an inconsistent state. This permits programmers to ensure that critical *database consistency constraints* are not violated. Typical consistency constraints are: ‘‘The total amount of money in all accounts is fixed,’’ ‘‘The number of Grateful Dead tickets initially available minus the number that have been sold is greater than or equal to zero,’’ and ‘‘Every pointer in array *A* points to a valid record.’’ The consequences of observing a database in an inconsistent state can be disastrous.

If a replicated data object is to be used in an application where data consistency is required, it must preserve serial consistency: the replicated object must display the same semantics as its serially accessed, single-site counterpart. This is the basic correctness criterion for replicated objects. It is known as *one-copy serializability*. A replication protocol that exhibits this property is said to be *transparent*. It is difficult to provide one-copy serializability in combination with high performance. (By *high performance*, we mean high concurrency and low communication, computation and space costs.)

This dissertation sets forth the thesis that replication of abstract data objects can be made sufficiently high in performance and easy to use that it represents a practical option to builders of highly available systems. The dissertation presents an architecture that enables the application programmer to quickly produce correct, efficient replicated implementations of a broad class of useful data types. The architecture requires the support of an underlying transaction system. The heart of the architecture is a family of efficient replication protocols that implement a class of data objects called *replicated*

sparse memories. The architecture was implemented on the Camelot distributed transaction system [19].

1.1. System Model

Our architecture runs on a distributed computer system consisting of multiple nodes connected by a communication network. The network may be a local area network, a wide area network, or any combination of the two.

The architecture requires the support of a *general purpose distributed transaction system* [27] like Camelot [19] or Argus [39]. Transactions are units of computation with three basic properties:

- Failure Atomicity - Transactions either run to completion or they have no effect at all. A transaction that runs to completion is said to *commit* and a transaction that has no effect at all is said to *abort*.
- Serializability - Although transactions may run concurrently, they appear to run serially: each transaction observes all of the effects of every transaction that occurs before it in the equivalent serial ordering, and none of the effects of the transactions that occur after it.
- Permanence - Once a transaction commits, its effects are permanent: it can no longer abort.

Together, these three properties create the illusion that transactions are executed serially and to completion. Distributed transactions access data at two or more nodes. Our replication protocols use distributed transactions to maintain consistency constraints in the replicas that collectively represent a data object. The applications that use the objects exported by our protocols use transactions to maintain consistency constraints on collections of these objects.

We assume a *client-server model*, wherein client processes communicate with server processes via *remote procedure calls* (RPCs) [45]. Transactions become distributed by performing RPCs to servers on remote nodes. We assume that *transactional RPCs* are used. Unlike normal RPCs, which have *at-most-once* semantics, transactional RPCs have *exactly-once* semantics: either an RPC executes to completion or the transaction that made the RPC is aborted. Communication links may fail, and messages may be lost or reordered, but these failures will be masked by the RPC protocol. A link failure can cause some nodes to become inaccessible from others. This situation is referred to as a *network partition*. Our protocols tolerate partitions, although some operations on some data objects may be unavailable from some nodes while a partition exists.

Nodes in a distributed system may fail, but we assume that they fail by halting: they must not continue to operate while performing incorrect computations. Such processors are said to be *fail-fast* [26]. The transaction system detects node failures and aborts all transactions that were active at a processor when it failed.

While nested transactions [44] are not required by our architecture, they simplify the implementation of our protocols. The replicated data objects exported by the architecture are fully compatible with nested transactions. The architecture can be built on stable storage that is implemented with value logging [29] and two phase locking [27]. More sophisticated locking and recovery techniques could be used to improve the concurrency of some data types, but these techniques are not required, and we did not use them in our implementation.

Communication costs tend to dominate local computation costs in distributed systems. In particular, transactional RPCs are among the most expensive basic operations exported by distributed transaction systems. Therefore, the main focus of our quest for efficiency is the reduction of the number of RPCs required by our protocols. It is substantially faster to perform multiple RPCs in parallel than to perform them in sequence, so the performance measure of greatest concern to us is the number of groups of parallel RPCs that are performed in the execution of an operation. This performance measure is often referred to in the literature as the number of *message delays incurred* by an operation. We refer to a group of parallel RPCs as a *round of message exchanges* or simply a *round of messages*. We assume that the length of a message is bounded by some system dependent constant.

While we insist that the worst-case costs of our protocols be reasonable, we are more concerned with average-case costs. A detailed discussion of the performance measures of interest to us may be found in Section 5.2.

1.2. Related Work

Early work on replication focused on simple data objects, usually called *files*, that support only whole-file read and write operations. While complex data types can be built on top of files, the resulting implementations generally display poor performance, especially in the area of concurrency.

One basic approach to replication is *unanimous update*: write operations are performed on all replicas, and reads are directed to any replica. The cost of the read operation is potentially very low, as no off-site communication is required if there is a replica at the same site as the client. But write availability is poor: a single site failure prevents write operations from proceeding. Write availability can be increased by using the communication system to buffer updates to unavailable replicas. The SDD-1 distributed database system uses this approach [51]. The *available copies* method takes a similar approach [5]. These approaches cannot tolerate network partitions.

A second basic approach, due to Alsberg and Day, is based on keeping a *primary copy* of each file and a number of *secondary copies* [3]. The primary copy receives all

operations and relays updates to the secondaries. If the primary fails, one of the secondaries takes its place as the new primary. As described by Alsberg and Day, this approach does not provide one-copy serializability, although it has been extended to do so by the addition of *synchronization sites* [47]. One disadvantage of primary copy approaches is that the site at which the primary copy is located or the links to this site can become a bottleneck. Another disadvantage is that some reconfiguration protocol is necessary to select a new primary when the primary becomes unavailable. Such protocols can be expensive and difficult to tune. If the time interval between observing a failure and attempting reconfiguration is too short, large amounts of resources can be wasted reconfiguring the system in response to transient failures. If the interval is too long, a failure of the primary can cause a noticeable interruption in service.

A third basic approach, due to Gifford, is *weighted voting* [22, 23]. In weighted voting, there are N replicas, each consisting of a copy of the file and a *version number*. Writes are directed to a subset of the replicas, called a *write quorum*, and reads are directed to a subset called a *read quorum*. A write quorum consists of W replicas and a read quorum consists of R replicas. The quorum sizes, W and R, are chosen so that $R + W > N$. Thus, every read quorum intersects every write quorum, and each read quorum includes at least one copy of the most current version of the file. The version numbers enable the reader to identify the current version. The read and write quorum sizes control an availability and performance tradeoff between the read and write operations. The weighted voting approach forms the basis for our techniques. It is discussed in more detail in Sections 2.2 and 2.2.1.

El Abbadi, Skeen and Cristian extend the unanimous update approach to allow updates during a partition [2]. In this approach, the nodes maintain *virtual partitions*, which are logical groups corresponding to perceived actual partitions. The unanimous update approach is used within each virtual partition. Only a virtual partition containing a majority of the replicas for any object can access the object. El Abbadi and Toueg extend the virtual partitions approach to gain added flexibility [1]. In their approach, nodes maintain *views*, which are similar to virtual partitions. Within each view, the weighted voting technique is used. Performance and availability tradeoffs between read and write operations can be controlled by choosing appropriate quorum sizes. The virtual partitions and views approaches incur substantial expense in reconfiguring the system when a node goes down or comes back up.

Herlihy describes a technique called *generalized quorum consensus* whereby weighted voting can be systematically applied to *any* abstract data type [31]. This technique provides extremely high concurrency and great freedom in trading off the relative availability of the operations on an object. However, it has high communication, computation, and storage costs. The generality of the technique derives from the fact that it is *event-based*: an object's state is represented by a log of the operations that have been

performed on it. The high concurrency derives from the fact that *transaction-consistent timestamps* are used instead of version numbers, to implement a *hybrid atomic* [58, 59] concurrency control mechanism. Heddaya modifies Herlihy's technique to address certain efficiency issues [30].

Joseph describes a technique for replication of arbitrary abstract data objects on a local area network based on *C-schemas* [33]. His technique is based on a much more restrictive model of distributed systems than ours. His model calls for an explicit operation scheduler and specifically excludes network partitions.

Cooper describes a technique based on *replicated procedure calls* [14, 15]. Multiple processes duplicate each other's actions in parallel to form a *troupe*. When a *client troupe* performs a replicated procedure call on a *server troupe*, each client process sends a message to each server process, but each server process executes the RPC only once. This method applies to arbitrary deterministic data types. A specialized commit protocol insures that transactions are serialized in the same order at all servers. Therefore the state of all servers in a troupe remains consistent. When a failed server recovers, it must copy all of the state information from an operational server in a transaction consistent fashion. Cooper's method uses replication instead of stable storage to achieve permanence of transactions.

Oki describes a technique called *viewstamped replication* wherein nodes maintain views and use a primary/secondary copy approach within their view [46]. Unlike the basic primary/secondary copy approach, the replicas are arbitrary servers, hence Oki's method applies to arbitrary abstract data types. To reduce latency, updates are passed from the primary to the secondaries in the background. If an update has not been passed to enough secondaries at transaction prepare time, the prepare will block. Like Cooper's method, Oki's method uses replication instead of stable storage to achieve permanence. This has some efficiency advantages, but requires that the replication system be integrated into the underlying transaction system.

The *replicated sparse memory* data structure described in this dissertation derives directly from the *replicated directory*, which was developed jointly with Daniels and Spector [10]. The replicated directory is an extension of earlier work by Daniels and Spector [16].

The major attribute of our work that distinguishes it from other recent work in the area of replication is its focus on practicality. The replication protocols underlying our architecture were designed primarily for efficiency. Our architecture does not provide complete generality in the data types that it supports. Neither does it maximize the potential concurrency offered by data objects, or the degree of freedom in trading off the relative availability of their operations. However, we believe that the generality,

concurrency, availability and performance of our architecture are high enough that it presents a practical option to builders of highly available systems. Our architecture is implementable on current day system software. It is optimized for data types that are widely used in practice.

Our work has several other unique features. This dissertation describes a fairly complete prototype implementation of our architecture. The implementation includes many of the optimizations that would be used in a commercial implementation. Implementation details are described and various performance figures are reported.

While our protocols are primarily optimized for low message counts, they also provide low storage consumption and low communication volume. We perform a mathematical analysis that shows that average computation and space costs remain low over a very wide range of system configurations and operation mixes. In essence, the analysis shows that the distributed data structure underlying our protocols is *self-cleaning* in the sense that it automatically garbage-collects out-of-date information as fast as it is produced.

1.3. Overview

Chapter 2 describes the *replicated sparse memory* or *RSM*, a replicated implementation of a table-like data object that associates values with addresses over an arbitrary address space. The object provides operations to write, read, and erase the information associated with an address. The replication algorithm is based on Gifford's *weighted voting* technique [22, 23]. The replicas in an RSM associate separate version numbers with individual addresses. In order to support the erase operation, which removes the value associated with an address, each replica must associate version numbers with every address in the address space, whether or not it has a value associated with it.

The read and write operations in the RSM are fairly straightforward, requiring one and two rounds of messages, respectively. The erase operation is more complicated. A critical step in performing the erase operation is determining the closest addresses above and below the one being erased that currently have data associated with them. The naive method for performing this step requires a potentially unbounded number of rounds of messages. In order to develop an algorithm that performs this step efficiently, we prove a basic structural property of the replicated sparse memory. The resulting algorithm requires two rounds of messages in the worst case, and normally runs in a single round. Predictive formulas derived in Chapter 3 can be used to tune the algorithm so that it almost always runs in a single round.

Chapter 3 studies the performance of the replicated sparse memory. Simulation results are presented that suggest that the average time and space performance of the RSM are good over a wide range of possible configurations. A Markov model of the RSM under

random use is constructed and analyzed using *balance equations*, mathematical assertions that a dynamic equilibrium exists in the system. Predictive formulas for the performance measures studied in the simulation are derived from the solutions of the balance equations. The predictions of the analysis agree remarkably well with the data gathered in the simulations. The results of the analysis indicate that the favorable performance observed in the simulations extends to a very wide range of system configurations and operation mixes.

Chapter 4 describes several optimizations and extensions to the replicated sparse memory. Traditionally, write operations in weighted voting algorithms have required two rounds of messages, one to determine the current version number associated with the data, and one to write the data. A technique called *optimistic timestamps* is introduced, wherein approximately synchronized real-time clocks are used to generate version numbers. This technique eliminates the first round in the great majority of write operations. The optimistic timestamp technique is generalized to form a broad class of optimizations known as *optimistic two-stage protocols*, which are used throughout this dissertation to reduce the communications costs of our algorithms. We argue that optimistic two-stage protocols represent a generally useful paradigm for the design of fast distributed algorithms in the context of transaction systems.

Other optimizations described in Chapter 4 include a technique for reducing the latency of the erase operation and a technique for reducing the cost of modifying individual fields of records stored in RSMs. Two alternative RSM implementations, called *hash table RSMs* and *array RSMs*, are presented. These implementations have significant performance advantages over ordinary RSMs, but are not as broadly applicable.

Two extensions to the functionality of RSMs are described in Chapter 4: range operations and navigation operations. Range operations permit a client to read, write or erase all of the data associated with a range of addresses. Navigation operations permit a client to scan backwards or forwards through the addresses in an RSM that currently have data associated with them. Efficient algorithms are presented for the range and navigation operations.

Chapter 5 discusses the use of replicated sparse memories. RSMs have several parameters that may be varied to provide a rich family of abstract data types. A brief taxonomy of this family is presented. RSMs provide a powerful base on top of which many interesting abstract data types can be built. The efficiency that can be obtained from such implementations is discussed. Efficient RSM implementations for several useful data types are presented, including *directories*, *record files with secondary indices on selected fields*, and *priority queues*. Not all data types can be implemented efficiently on RSMs. One useful data type that cannot is the *counter*. An efficient replicated implementation for this data type is presented, and the use of this implementation in conjunction with RSMs is discussed.

Chapter 6 describes an architecture that provides application programmers with implementations of the replicated data objects described in previous chapters. The architecture consists of a set of replica servers and a library that implements our replication protocols on top of these servers. Replicated objects are dynamically created in response to requests from application programs. Many parameters of the objects, including address space, value type, and replica representation type may be specified at runtime. The architecture is easy to use, as it insulates the programmer from the details of the replication protocols.

We implemented the architecture on the Camelot transaction system. The implementation is fairly complete, including all of the major RSM variants described in this dissertation. The implementation is described in Chapter 6.

We performed experiments to evaluate various aspects of the performance of our prototype. Chapter 7 presents these experiments. Many performance figures are presented, including some concurrent performance data. We compare our measurements with predictions based on the performance of underlying primitives. A high degree of agreement is observed between measurements and predictions, indicating that we understand the performance of our system. While the performance of the prototype is limited by that of the underlying system software and hardware, it clearly demonstrates the practicality of our approach. Our predictions can be extended to predict the performance that would result if our architecture were implemented on a different platform.

Chapter 8 contains conclusions and directions for future work.

Chapter 2
Replicated Sparse Memories

2.1. Introduction

The goals of object replication on distributed computing systems are increased parallelism, reduced communications costs, and increased resilience in the presence of failures. In particular, replication can permit increased object *availability*: continued access to an object despite the failure of one or more of the nodes on which it is stored. Unfortunately, it is difficult to achieve high performance and availability while ensuring that the semantics of replicated data objects are identical to those of their non-replicated counterparts.

This chapter describes a distributed data structure called the *replicated sparse memory* or RSM. The RSM implements a *sparse memory*, a directory-like abstract data object that maps *addresses* to *values*. Addresses are chosen from a set of constants called the *address space*. Sparse memories are accessed with the following operations:

> **Write(IN val: value, addr: address)** - Associates a value with an address. The address is said to be *occupied* if it has been written and not subsequently erased.
>
> **Read(IN addr: address; OUT occupied: boolean, val: value)** - Returns TRUE and the value associated with the address, if it is occupied, or FALSE and an undefined value, if it is unoccupied.
>
> **Erase(IN addr: address)** - Erases any value currently associated with an address, causing it to return to the unoccupied state.

The sparse memory is an attractive data type when compared with more common directory-like data types because of its simplicity and generality. It is defined by only three operations. It allows *blind writes*: clients can associate a value with an address without knowing whether a value is currently associated with the address. Sparse memories provide a good basis for implementing a wide variety of useful data types. The broad applicability of the sparse memory is discussed at length in Chapter 5.

The only condition that our data structure imposes on the address space is that it be *totally ordered*. We make no other assumptions about the structure of the address space;

it can be finite, countably infinite, or uncountably infinite, and either dense (like the rationals) or sparse (like the integers). For all of the examples in this paper, the set of finite length alphabetic strings with lexicographic ordering is used as the address space.

The RSM data structure presented in this chapter permits concurrent operations and arbitrarily high data availability. A measure of availability appropriate to this work is the number of node failures that a replicated object can tolerate while guaranteeing that an operation can be performed. The semantics of the replicated sparse memory are identical to those of a sparse memory stored on a single node and accessed serially. Thus the replication algorithm is said to be *transparent*.

RSM operations execute as part of distributed transactions, which provide uniform synchronization and recovery properties for operations on arbitrary shared abstract types. The RSM is an example of a distributed abstract data type that is constructed from a collection of more primitive, non-distributed types. Transactions simplify the maintenance of the invariants necessary to make this replication algorithm work.

The replication technique described in this chapter is an extension of an algorithm for *replicated directories* presented by Daniels and Spector [16]. It is based on Gifford's weighted voting algorithm [22, 23], and has similar performance and reliability advantages. Unlike Gifford's algorithm, this algorithm efficiently associates at each replica a separate version number with every address in the address space. This permits concurrent operations on different addresses and solves certain problems in the implementation of the Erase operation. Unlike early replication algorithms, which implemented simple objects having only *read* and *write* operations, this algorithm uses the semantic properties of sparse memories to gain increased performance.

The remainder of this chapter is organized as follows. Section 2.2 describes weighted voting and discusses its use as the basis for an RSM data structure. Section 2.3 motivates and outlines our technique and Section 2.4 describes it in detail. Section 2.5 develops an efficient algorithm for the Erase operation. Section 2.6 contains correctness arguments.

2.2. Weighted Voting

In this section, we describe Gifford's *weighted voting* algorithm [22, 23], which forms the basis for our algorithm. Gifford's algorithm implements a replicated *file*, supporting only the read and write operations.[1] A file is stored as a collection of replicas, called

[1]While the term ''file'' has traditionally been used in connection with this work, it is not entirely appropriate, as the granularity of operations is the entire object. File systems typically permit operations on individual records.

representatives, each of which is assigned a certain number of votes. A representative consists of a copy of the file and a version number. The entire collection of representatives is called a *file suite*. Write operations write an updated copy of the file to each representative in a group called a *write quorum*, associating a new version number with all of these representatives. The new version number must be higher than any version number previously associated with a representative in the file suite. Read operations read from each representative in a *read quorum* and return the data from the representative with the highest version number. Write operations establish a higher version number by incrementing the highest version number encountered in a read quorum.

A write quorum consists of any set of representatives whose votes total at least W and a read quorum consists of any set of representatives whose votes total at least R. The constants R and W are chosen so that their sum is greater than the total number of votes assigned to all representatives, N. Thus, every read quorum has a non-null intersection with every write quorum and each inquiry is guaranteed to access at least one current copy of the data. Current copies will always have a higher version number than non-current copies so the read operation will always return current data. The values chosen for R and W control a tradeoff between the availability of the read and write operation.

Recall that write operations consult a read quorum to determine the highest version number previously associated with a representative in the file suite. As a consequence, the write operation requires the services of a collection of representatives whose votes total $\max(R,W)$: the write operation cannot be made more available than the read operation. If W were chosen to be less than R, the desired increase in availability of the write operation would not materialize; instead, the availabilities of both the read and write operations would decrease. Therefore, we assume that $W \geq R$.

2.2.1. Motivation for the Use of Weighted Voting

Weighted voting has several features that make it appealing as the basis for the design of a replicated sparse memory. A key feature of weighted voting algorithms is that they automatically function correctly in the face of network partitions. They do so passively, without the need for dynamic reconfiguration. Such dynamic reconfiguration adds great complexity to replication algorithms, and substantially reduces availability during periods of reconfiguration.

Another appealing feature of weighted voting algorithms is that the sizes of the read and write quorums may be varied to adjust the relative cost and availability of the operations. For example, read quorums can be made much smaller than write quorums if data is read much more frequently than it is written. Vote assignments can be adjusted to further

refine availability tradeoffs. For instance, a node that is more likely to fail can be given fewer votes, so its absence will have less effect on system availability.

Finally, algorithms based on weighted voting are simplified because consistency and recovery are primarily the responsibility of an underlying transaction facility. The use of an underlying transaction facility greatly simplifies the task of ensuring that operations on multiple distributed objects interact properly.

While weighted voting is an appealing approach to replication, the basic algorithm cannot be applied directly to sparse memories without undesirable concurrency limitations. Even though the semantics of sparse memories permit concurrent operations on different addresses, only a single transaction at a time could modify the contents of the memory if it were stored in a file suite; each copy of the entire memory would have a single version number, which would cause the serialization of all operations that modified the memory. Furthermore, any modification would require sending the entire updated memory to each representative in a write quorum. For a large memory, this would result in prohibitive communications costs. In Section 2.3, we develop an algorithm for replicated sparse memories from the weighted voting algorithm for files. Our algorithm rectifies the deficiencies described above.

2.3. Development of the Algorithm

In Section 2.2.1, we noted that weighted voting could not be applied directly to a sparse memory without excessive concurrency limitations and communications costs. It might seem that these limitations could be overcome if each *entry* in an RSM representative were assigned a separate version number. (An *entry* is defined as the physical data associated with an address at a representative, and consists of the address and an associated value.) But this approach proves inviable; it is not always possible to determine from an arbitrary read quorum whether a given address is occupied. The problem is illustrated below.

The RSM in this example consists of three representatives. In our examples, we will assume that each representative has one vote, though all results generalize to RSMs with arbitrary vote distributions. The read and write quorum sizes for the example are each two votes. The notation *N*-*R*-*W* refers to an RSM having *N* representatives, a read quorum size of *R* and a write quorum size of *W*. Thus, we call the RSM in our example a 3-2-2 RSM.

Initially, addresses ‘‘a’’ and ‘‘c’’ have been written to a write quorum consisting of representatives A and B; A and B contain entries for addresses ‘‘a’’ and ‘‘c’’, and each

entry has version number 1 (Figure 2-1)[2]. Subsequently, ‘‘b’’ is written to a write quorum consisting of representatives A and C, with version number 1 (Figure 2-2). If a request to read the data associated with address ‘‘b’’ is sent to representatives B and C at this point, representative B will respond ‘‘not present,’’ and representative C will respond ‘‘present with version number 1.’’ If ‘‘b’’ is then erased from the RSM by deleting its entry from representatives A and B (Figure 2-3), requests to read ‘‘b’’ on representatives B and C will still elicit the responses ‘‘not present,’’ and ‘‘present with version number 1.’’ If RSM representatives fail to associate version numbers with addresses for which they have no entries, the responses from a read quorum will not, in general, be sufficient to determine if a given address is occupied.

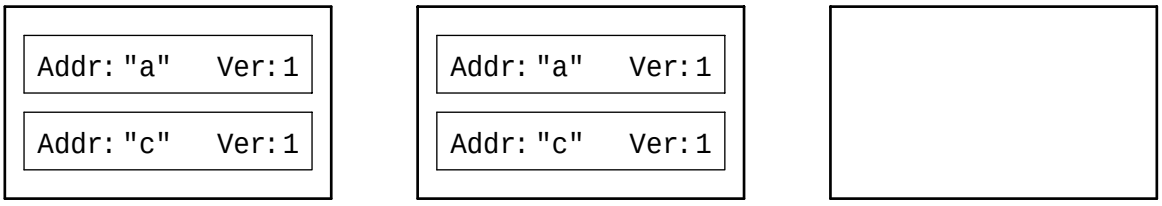

Figure 2-1: 3-2-2 RSM With Addresses ‘‘a’’ and ‘‘c’’ Occupied

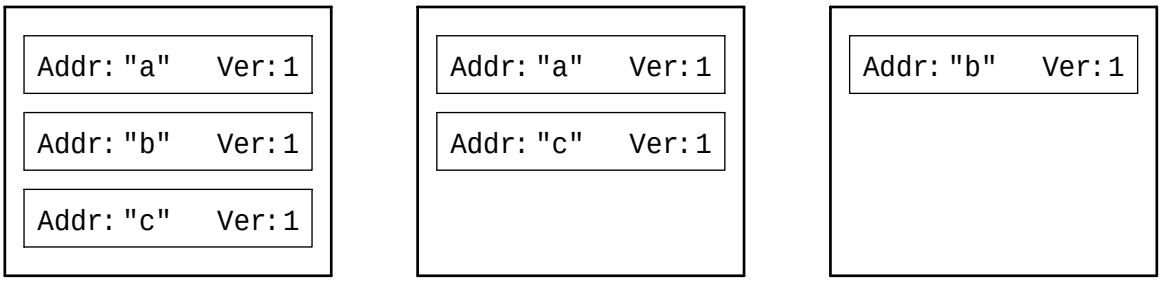

Figure 2-2: RSM After Writing to Address ‘‘b’’

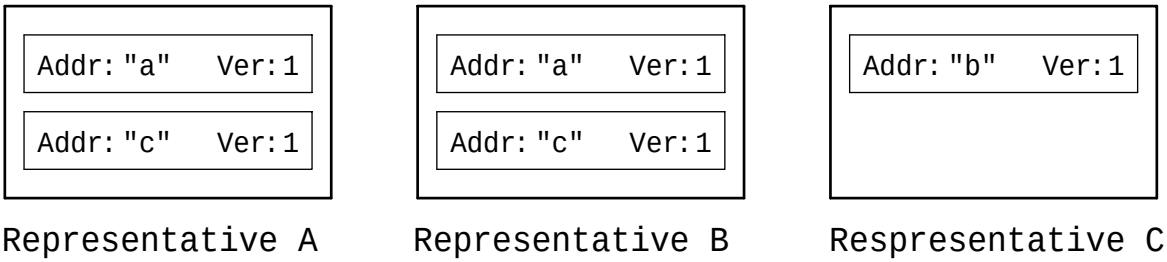

Figure 2-3: RSM After Erasing Address ‘‘b’’

[2]The value field is omitted from all figures for clarity.

The ambiguity illustrated above is associated with the Erase operation and would not occur if the operation were not permitted. Alternatively, erase could be implemented by replacing entries to be deleted with version-numbered *tombstones* and performing a garbage collection operation periodically. However, the garbage collection operation would be expensive and availability would be greatly reduced during the operation. A third strategy is to eliminate the ambiguity by consulting additional representatives whenever an inquiry to an initial set of representatives does not yield a read quorum of replies all of which agree on the presence or absence of an entry. But this approach drastically reduces availability.

None of the solutions presented thus far satisfy our demands for concurrency and availability. What is really needed is a scheme whereby each representative associates a version number with every address in the address space. This can be accomplished by partitioning the address space into disjoint sets and associating a version number with each set at every representative. The same partitions need not be used at all representatives.

One approach to partitioning is to divide the address space into ranges based on the order relation on the address space. The simplest partitioning scheme divides the address space into a number of fixed ranges. However, it is difficult to guarantee sufficient concurrency with such a *static partitioning* technique. If a small number of ranges are used, then at most that number of transactions can modify the contents of the RSM concurrently. If transactions perform operations on addresses in more than one range, concurrency will be further limited. Even if a large number of ranges are used, an uneven distribution of accesses can limit concurrency. Furthermore, static partitioning can cause excessive communications costs, as the current values associated with *every* address in a partition must be sent to all of the representatives in a write quorum each time an operation modifies the data associated with *any* address in the partition.

A more general approach is to allow the partition at each representative to vary over time, as a function of which entries currently reside at the representative. Such a *dynamic partitioning* technique is especially desirable for RSMs having sizes or access patterns that vary widely over time. A simple method of dynamically partitioning the address space at a representative is to create a partition for each address that has an entry in that representative and a partition for each range of addresses between successive entries. These inter-entry ranges are called *gaps*. This method forms the basis of our technique.

In this dynamic partitioning approach, read requests sent to a representative that contains an entry for the address being read return the version number associated with the entry. Read requests sent to a representative with no entry for the address return the version number associated with the gap in which the address lies. Write requests increase the version number of an entry, if it already exists, or split the gap into which the address

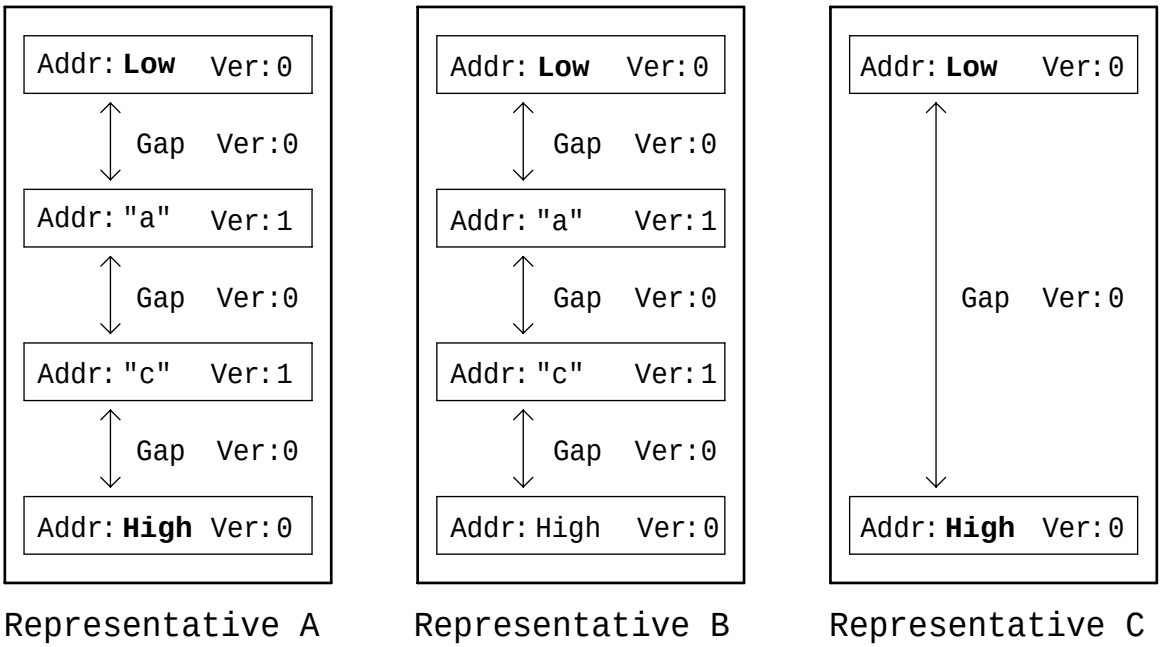

Figure 2-4: 3-2-2 RSM With Addresses ‘‘a’’ and ‘‘c’’ Occupied

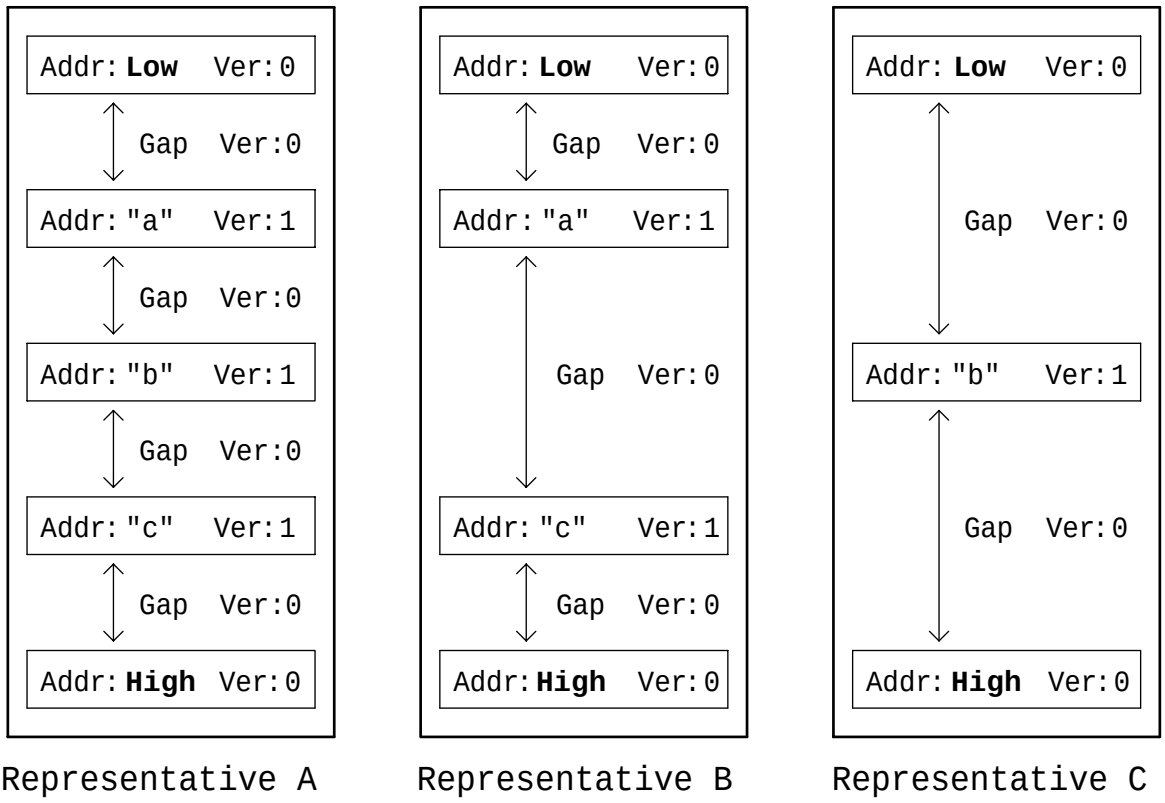

Figure 2-5: RSM After Writing to ‘‘b’’

falls if no entry exists. Erase requests coalesce gaps and entries in a range of addresses into a single gap. The details of these operations are discussed at length in Section 2.4.

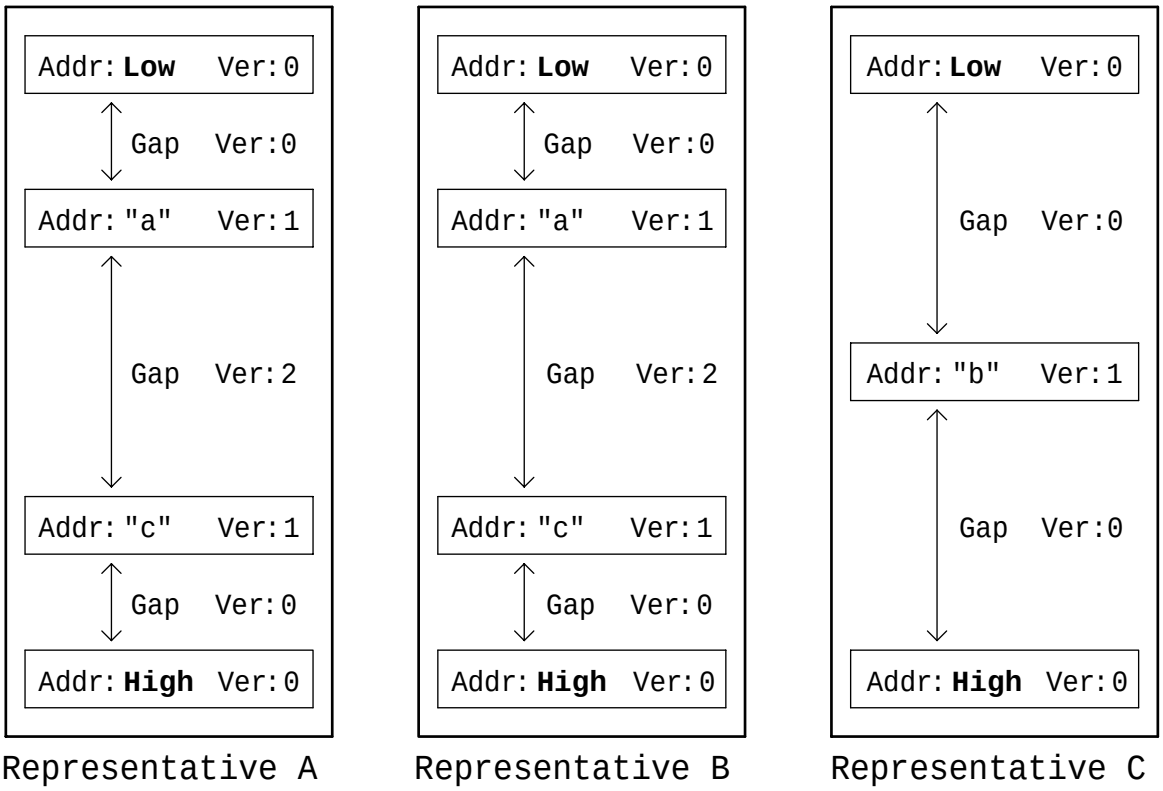

Figure 2-6: RSM After Erasing ‘‘b’’

The RSM containing entries for addresses ‘‘a’’ and ‘‘c’’ in representatives A and B of our previous example (Figure 2-1) is represented as in Figure 2-4.[3] If the address ‘‘b’’ is written to a write quorum consisting of representatives A and C, the RSM in Figure 2-5 results. Note that the entries for address ‘‘b’’ are assigned version number 1, which is one greater than highest version number previously associated with ‘‘b’’ at A or C.

If a request to read ‘‘b’’ is sent to representatives B and C at this point, representative B responds ‘‘not present with version number 0’’ and representative C responds ‘‘present with version number 1.’’ From these responses, a client can determine that ‘‘b’’ is indeed occupied. If ‘‘b’’ is subsequently erased from representatives A and B, then the two gaps on either side of ‘‘b’’ on representative A are coalesced. On both representatives, the gap between ‘‘a’’ and ‘‘c’’ is assigned version number 2, which is higher than any version number previously associated with an address between ‘‘a’’ and ‘‘c’’ (Figure 2-6). Now, if a request to read ‘‘b’’ is sent to representatives B and C, B responds ‘‘Not present with version number 2’’ and C responds ‘‘present with version number 1.’’ These responses indicate that the address is no longer occupied, resolving the ambiguity that occurred in the initial example, wherein version numbers were associated only with entries.

---

[3]The RSM representatives in Figure 2-4 contain entries for the special addresses **Low** and **High**, which delimit the first and last gaps in the representatives.

2.4. Details of the Algorithm

This section presents the details of the algorithm sketched in the previous section. The descriptions are illustrated with program text in a Pascal-like language that includes a remote procedure call primitive [6]. Remote procedures are declared like ordinary procedures except that the first parameter is always the identifier of a remote server and other parameters must be declared as IN or OUT. Parameters are passed by value in messages. Remote procedure calls have the same syntax as local procedure calls; the general purpose distributed transaction facility assumed as the underpinning of our algorithm guarantees that remote procedure calls have exactly-once semantics. If a node failure, timeout or other system error occurs during the execution of a remote procedure call, the calling transaction is aborted. Clarity is emphasized over performance in the programs. Optimizations that would be used in practical implementations are described in accompanying text.

Section 2.4.1 presents the operations on RSM representatives, from which the RSM operations are constructed. Section 2.4.2 presents the RSM operations.

2.4.1. RSM Representative Operations

In a replicated sparse memory, each representative is an instance of an abstract object that stores one (approximate) copy of the RSM data. Arbitrarily complex atomic transactions may be constructed using the basic operations provided by RSM representatives. RSM representatives must synchronize concurrent operations performed by different transactions and store critical information in a fashion that recovers from failures. Gifford's weighted voting algorithm makes similar requirements of its file representatives.

Every RSM representative contains entries for two distinguished addresses, **Low** and **High**. **Low** is defined to be less than any address in the address space and **High** is defined analogously. The entries for **Low** and **High** ensure that every address in the address space has at least one occupied address before it and one after it. This is essential for the implementation of the Erase operation, as described in Section 2.4.2.

RSM representatives provide two simple operations to read and write data associated with an address: RepRead and RepWrite. In addition, RSM representatives provide four specialized operations that are used to implement the RSM Erase operation: RepPredecessor, RepSuccessor, RepSuperseder, and RepCoalesce. Figure 2-7 gives procedure headings for each of these operations. The last line of each procedure heading specifies the locks obtained by the operation. These locks are discussed below.

```
RepRead(IN server: RsmRep, addr: address;
        OUT present: boolean, ver: version, val: value);
{ If there is an entry for addr, returns TRUE, the version number of
  the entry and its value; otherwise returns FALSE and the version
  number of the gap containing addr.

  Locks Read(addr).}

RepWrite(IN server: RsmRep, addr: address, ver: version, val: value);
{ Creates an entry for address addr with version number ver and value
  val, or updates the entry for address addr if it already exists.

  Locks Write(addr). }

RepPredecessor(IN server: RsmRep, addr: address;
               OUT pred: address, GapVer: version);
{ Returns the highest address less than addr that has an entry at the
  representative.  Also returns the version number of the gap between
  addr and its predecessor.  (There need not be an entry for addr.)

  Locks Read(pred, addr). }

RepSuccessor(IN server: RsmRep, addr: address;
             OUT Succ: address, GapVer: version);
{ Analogous to RepPredecessor.

  Locks Read(addr, succ). }

RepCoalesce(IN server: RsmRep, l, h: address, gapver: version);
{ Inserts entries for l and h if they are not present.  Inserted entries
  get version number 0 and undefined value.  Deletes entries for any
  addresses between (but not including) l and h.  The resulting gap is
  assigned version number gapver.

  Locks Write(l, h). }

RepSuperseder(IN server: RsmRep, addr1, addr2: address, ver: version,
              OUT superseded: boolean, superseder: address);
{ Searches the range from addr1 to addr2 (exclusive), starting from
  addr1.  Returns TRUE and the address of the closest entry to addr1
  with version number greater than ver, or FALSE if there are no entries
  between addr1 and addr2 with version number greater than ver.

  Locks Read(addr1, superseder), or Read(addr1, addr2) if no superseder
  is found. }
```

Figure 2-7: RSM Representative Operations

RepPredecessor returns the highest address less than the given address that has an entry at the representative (i.e. the address of the entry immediately preceding the given address). It also returns the version number of the gap between the entry for the returned address and its successor. (Note that a version number is maintained for the gap between each pair of entries, even if the address space has no addresses in the range that the entries delimit.) RepSuccessor is analogous to RepPredecessor.

RepCoalesce deletes any entries appearing in a range between two specified addresses and assigns a single version number to the resulting gap, inserting the entries delimiting the gap, if necessary. In other words, RepCoalesce coalesces a collection of entries and gaps into a single gap.

RepSuperseder searches a range and returns the entry closest to the first endpoint of the range with a version number greater than a given version number. If the search reaches the end of the range without locating an entry to return, RepSuperseder indicates that no entry was found. Intuitively, the RepSuperseder operation locates the first entry that supersedes a specified gap.

Each RSM representative must synchronize concurrent operations of different transactions. While this can be accomplished in many ways, the locks specified in Figure 2-7 assume the use of *range locking* [54] with shared and exclusive mode locks. Range locks are just like normal shared/exclusive mode locks, except that a range lock can be granted only if it is compatible with all of the locks currently held on ranges that intersect it. In other words, a write lock on a range can only be granted if no locks are held on an intersecting range by a conflicting transaction. A read lock on a range can only be granted if no write locks are held on an intersecting range by a conflicting transaction. The notation Read(x, y) refers to a read (shared) lock on the range from address *x* to address *y*, inclusive. Read(x) is shorthand for Read(x, x). (Read(x) is a degenerate range lock that locks only a single address.) Write(x, y) and Write(x) are defined analogously.

Inquiry operations (RepRead, RepPredecessor, RepSuccessor, and RepSuperseder) set read locks on the range of addresses explicitly or implicitly accessed by the operation. Modification operations (RepWrite and RepCoalesce) set write locks on the range of addresses being modified.

The locks specified in the procedure headers are sufficiently strong to guarantee that the actions of transactions operating on an RSM representative are serializable, provided that two phase locking is used [56]. This form of synchronization simplifies the correctness arguments given in Section 2.6.

Each RSM representative is responsible for recovery processing. Recovery processing is necessary to undo the effects of partially completed transactions after a crash or when a transaction aborts. The details of recovery processing are specific to the implementation of the RSM representative and depend on the recovery approach used by the underlying transaction system. Mohan et al., Gray et al., Lindsay et al., and Schwarz, among others, present more details on general recovery algorithms [43, 28, 38, 53].

In any recovery scheme, it is necessary for an RSM representative to record enough information reliably to redo or undo the effects of the operations that modify the state of

the representative. If *value logging* is used, this will be done invisibly by the underlying transaction system. If *operation logging* is used, the programmer must specify what data to record and how to undo the operation. We describe below the data that must be recorded in order to undo each representative operation and the procedure for undoing the operation, for use with operation logging.

To undo a RepWrite operation, the value and version number previously associated with the address at the representative must be recorded. If a RepWrite operation modified a preexisting entry, it is undone by reverting the entry to its previous value and version. If a RepWrite caused a new entry to be inserted into a gap, it is undone by deleting the entry and combining the gaps preceding and following it. These gaps are guaranteed to still have the same version number they did when the RepWrite took place, as the lock secured by the RepWrite operation assures that the version numbers cannot be changed. Thus, it is not strictly necessary to record an old version number when inserting an entry into a gap.

To undo a RepCoalesce operation, a representative must record the address, value and version numbers of all entries deleted by the operation, and the version numbers of the gaps between the entries. To undo the operation, the deleted entries are reinserted and the gap version numbers are restored to their original values.

Note that the range locks specified in Figure 2-7 assume the use of operation logging. If value logging is used, transactions will hold on to physical locks until they complete, making it unnecessary to secure explicit logical locks. While these physical locks are guaranteed to ensure serializability, they may restrict concurrency excessively. This effect can be reduced somewhat by exercising care in designing recoverable data structures, and by explicitly dropping locks prematurely when it is safe to do so.

2.4.2. RSM Operations

An RSM consists of a set of N RSM representatives, an assignment of votes to the representatives, and the read and write quorum sizes, R and W. The quorum sizes are chosen to conform to the constraints described below. RSMs implement the operations Read, Write, and Erase, as specified in Section 2.1. Operations on RSM representatives are combined to implement an RSM based on the weighted voting rules described in Section 2.2.

Recall from Section 2.2 that $R+W>N$ and $W \geq R$. Combining these two inequalities, we have $2W>N$. Thus any value between $\lfloor N/2 \rfloor+1$ and N, inclusive, can be chosen for W. R will generally be set to $N-W+1$, the smallest value necessary to ensure the required quorum intersection.

The Read operation calls the procedure IRead, shown in Figure 2-8, and discards the version number, returning a boolean indicating whether or not the address is occupied and the value associated with the address, if it is occupied. IRead first calls CollectReadQuorum which returns identifiers for a read quorum of RSM representatives. Then RepRead operations are performed on the quorum and the data from the with the highest version number is returned.

```
IRead(IN addr: address;
      OUT occupied: boolean, ver:version, val:value);
{ Internal read procedure.  Returns TRUE, the version number, and the
  value associated with addr, if it is occupied; FALSE otherwise. }

var
    quorum: array[1..R] of RsmRep;
    RepVer: version;
    RepVal: value;
    present: boolean;
    i: integer;

begin
    quorum := CollectReadQuorum;

    ver := -1; { Lower than any legitimate version number }

    for i := 1 to R do
    begin
        RepRead(quorum[i], addr, present, RepVer, RepVal);
        if RepVer > Ver then
        begin
            occupied := present;
            if present then
            begin
                ver := RepVer;
                val := RepVal;
            end
        end
    end
end
```

Figure 2-8: IRead Operation

CollectReadQuorum and its companion function, CollectWriteQuorum, bind identifiers to instances of RSM representatives. This may involve message exchanges to establish communications sessions, so it is desirable that these operations cache information to be used in subsequent invocations. Furthermore, reuse of the same read and write quorums will improve the efficiency of the data structure. We discuss this at greater length in Section 3.4.

The Write operation is straightforward. It calls IRead to find the highest version number previously associated with the address to be written. This version number is incremented to create the new version number. The RepWrite operation is used to write

```
Write(IN addr: address, val: value);
{ Associates a value with an address. }
var
    quorum: array[1..W] of RsmRep;
    i: integer;
    OldVer, NewVer: version;
    OldVal: value;
    occupied: boolean;

begin
    { Read address to find current version number }
    IRead(addr, occupied, OldVer, OldVal);

    quorum := CollectWriteQuorum;

    { The new entry's version number must be higher than the
      previous version number associated with the address }
    NewVer := OldVer+1;

    for i:= 1 to W do
        RepWrite(quorum[i], addr, NewVer, val);
end
```

Figure 2-9: Write Operation

an entry for the address with the new value and version number into a write quorum of representatives. This operation is illustrated in Figure 2-9.

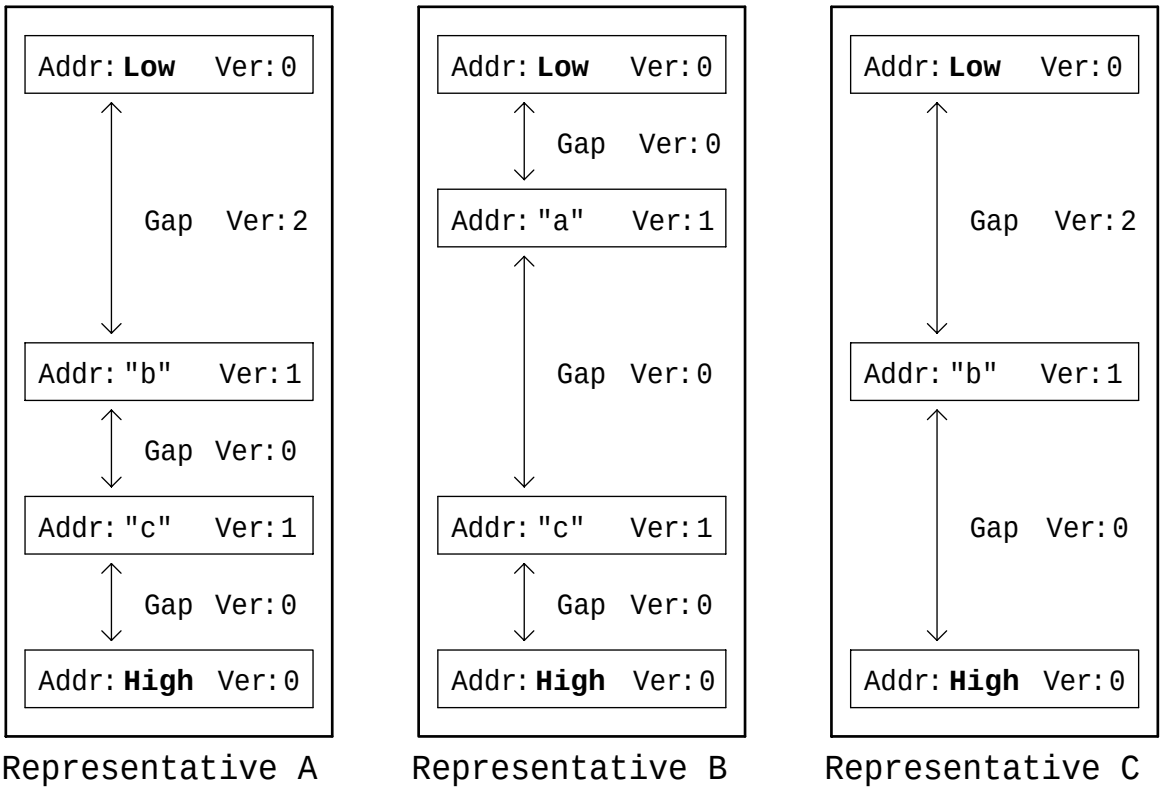

Figure 2-10: RSM from Figure 2-5 After Erasing ‘‘a’,

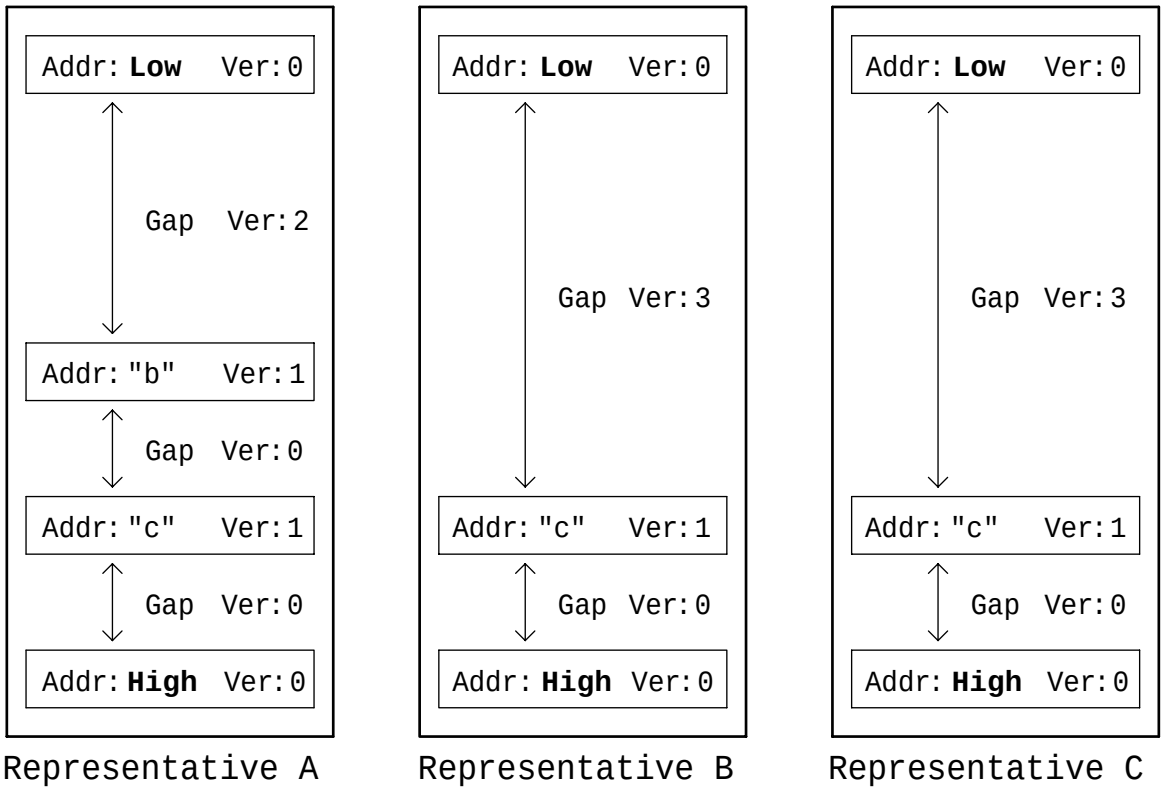

Figure 2-11: RSM from Figure 2-10 After Erasing ‘‘b’’

Erase must record the fact that an address is not currently occupied at each member of a write quorum. This is accomplished by coalescing a range of addresses that includes the entry to be erased and assigning a version number to the resulting gap that is higher than that of any entry or gap previously contained in the range. To avoid asserting that an occupied address in unoccupied, the range to be coalesced must not contain any addresses that are currently occupied, other than the one that is being erased. Erase coalesces a range that extends from the *real predecessor* of the address to be erased to its *real successor*, thereby ensuring that there are no occupied addresses in the coalesced range. The real predecessor of an address *a* is the highest address less than *a* that is occupied. The real successor of an address is defined analogously. The presence in the representatives of entries for the distinguished addresses **Low** and **High** ensure that every address in the address space will have a real predecessor and a real successor.

The coalesce operation inserts the real predecessor and successor into a representative if they are not already present, to delimit the newly formed gap. The entries between the address and its real predecessor as well as those between the address and its real successor comprise the address’s *delete list* at that representative. The delete list is so named because these entries are deleted when performing the coalesce operation required to erase the address.

Locating the real predecessor and real successor of an address to be erased is complex. There may be arbitrarily many *ghost* entries located between the address to be erased and its real predecessor or real successor. A ghost is defined as an entry for an address that is no longer occupied. Locating the real predecessor and real successor of an address is complicated by the fact that the real predecessor or real successor may not be present in some members of the read quorum.

These problems are partially illustrated in the following example. Consider the RSM in Figure 2-5. Suppose we erase address ‘‘a’’, using representatives A and C as the write quorum. This operation is straightforward, resulting in the RSM shown in Figure 2-10. Now suppose we erase address ‘‘b’’, using representatives B and C as the write quorum. Figure 2-10 shows that the real successor of ‘‘b’’ is ‘‘c’’. However, no entry for ‘‘c’’ appears in representative C, and the ghost of entry ‘‘a’’ appears between ‘‘b’’ and **Low** (the real predecessor of ‘‘b’’) in representative B. To erase ‘‘b’’ from representatives B and C, the real successor, ‘‘c’’, must be inserted into representative C. The coalescing of the range from **Low** to ‘‘c’’ eliminates the ghost of entry ‘‘a’’ from representative B. The resulting RSM is shown in Figure 2-11.

A simple Erase procedure is illustrated in Figure 2-12. Finding the real predecessor and successor of an address is the heart of this operation. Given an address, RealPredecessor returns the address’s real predecessor and the highest version number found in the range between the address and its real predecessor, exclusive. The RealSuccessor operation is analogous.

The straightforward procedure for performing the RealPredecessor operation presented by Daniels and Spector [16] suffers from a serious drawback: it requires that potentially many rounds of messages be sent between the node determining the real predecessor and the nodes comprising the read quorum. One round of messages is required for every ghost between the address being erased and its real predecessor that is found in any representative in the read quorum. While this message traffic can be reduced by combining messages, and while the simulations and analysis show that the *average* performance is not too bad, the number of fixed length messages that must be transmitted for a single Erase operation is potentially unbounded.[4] All the other RSM operations require only a small constant number of fixed length communications; it would be highly desirable to have an algorithm for the RealPredecessor operation (hence the Erase operation) that has this property as well. We develop such an algorithm in Section 2.5.

[4]In fact, it is bounded by $2R \times$ (the cardinality of the address space), where R is the read quorum size. For finite address spaces, this expression will be large but finite.

```
Erase(IN addr: address);
{ Erases the contents of an address, causing it to become unoccupied }
var
    quorum: array[1..W] of RsmRep;
    i: integer;
    occupied: boolean;
    succ, pred: address;
    val: value;
    ver, PredGapVer, SuccGapVer, NewGapVer: version;

begin
    IRead(addr, occupied, ver, val);

    RealPredecessor(addr, pred, PredGapVer);
    RealSuccessor(addr, succ, SuccGapVer);

    { The version number of the coalesced gap must be higher than
      any previous version number in the range to be coalesced. }
    NewGapVer := Max(ver, PredGapVer, SuccGapVer) + 1;

    quorum := CollectWriteQuorum;

    for i:= 1 to W do
        RepCoalesce(quorum[i], pred, succ, NewGapVer);
end
```

Figure 2-12: Erase Operation

2.5. An Efficient Algorithm for the Real Predecessor Operation

An algorithm for finding the real predecessor must in effect *prove* that a certain address is the real predecessor. Such a proof involves showing that all entries between an address and its real predecessor in each representative of a read quorum are superseded by a gap with a higher version number in some other representative of the quorum. The number of ghosts between an entry and its real predecessor is potentially unbounded at each representative, so the prospects for the existence of an algorithm that requires only a constant number of fixed length messages might appear dim.

However, RSMs have a property that constrains the system states that can occur. Because of this property, a single round of messages suffices to find a region guaranteed to contain the real predecessor as well as the minimum version number necessary for an entry in this region to represent an occupied address. With this information, the real predecessor can be found in one additional round of messages. To state and prove the property that permits this efficient location of the real predecessor, we must introduce several terms.

A *region* is a set of addresses; that is, a subset of the address space. A *range* is a region containing every address in the address space between some address and another address. These definitions are consistent with our informal use of the terms in previous sections.

The notation (a_1, a_2) refers to the range from a_1 to a_2 excluding a_1 and a_2, the *endpoints* of the range.

A gap between entries for addresses a_1 and a_2 is said to *cover* the region (a_1, a_2) and all of its subregions (subsets). The remaining terms are defined in the context of an entire RSM, rather than an isolated representative. A gap *g* is said to be *current over the region r* if the following conditions hold:

1. The gap *g* covers *r*.
2. No gap in some other representative covering any non-null subregion of *r* has a higher version number than *g* does.
3. No entry in some other representative for an address in *r* has a higher version number than *g* does.

Intuitively, a gap is current over a region for which it expresses the most up to date information. A gap's *region of currency* is the union of all regions over which it is current, intuitively, the entire region over which it is current.

For example, consider the RSM in Figure 2-13. Gap *g* covers (''c'', **High**) and all of its subregions (e.g. (''d'', ''f'')). Gap *g* is current over (''g'', ''k''), for example. Gap *g*'s region of currency is (''c'', ''d'')$\cup$(''e'', **High**).

We are now ready to state the property that allows us to construct an efficient RealPredecessor algorithm.

THEOREM 1. *In any occurring system state, every gap's region of currency can be expressed as the union of a finite number of ranges whose endpoints are currently occupied addresses.*

Before we can prove Theorem 1 or present the real predecessor algorithm we must introduce one more term and present two lemmas. A collection of ranges $\{r_i \mid_{i=1}^{n}\}$, $r_i=(a_{i1}, a_{i2})$ is said to be *canonical* if the ranges are in order $(a_{i1} < a_{(i+1)1})$ and non-intersecting $(\forall i \neq j,\ r_i \cap r_j = \varnothing)$. The following lemma justifies our use of the term *canonical*.

LEMMA 1. *For any finite collection of ranges over a dense address space, there exists a unique canonical collection of ranges whose union comprises the same set as the union of the original collection. This is referred to as the canonical form of the original collection. (An address space is dense if, for every pair of addresses a_1, a_2, with $a_1 < a_2$, there exists an address a_3 such that $a_1 < a_3 < a_2$.) Further, the endpoints of the ranges in the canonical form are all endpoints of some range in the original collection.*

LEMMA 2. *If a gap g is current over a range (a_1, a_2), and addresses a_1 and a_2 are occupied, then a_1 is a_2's real predecessor.*

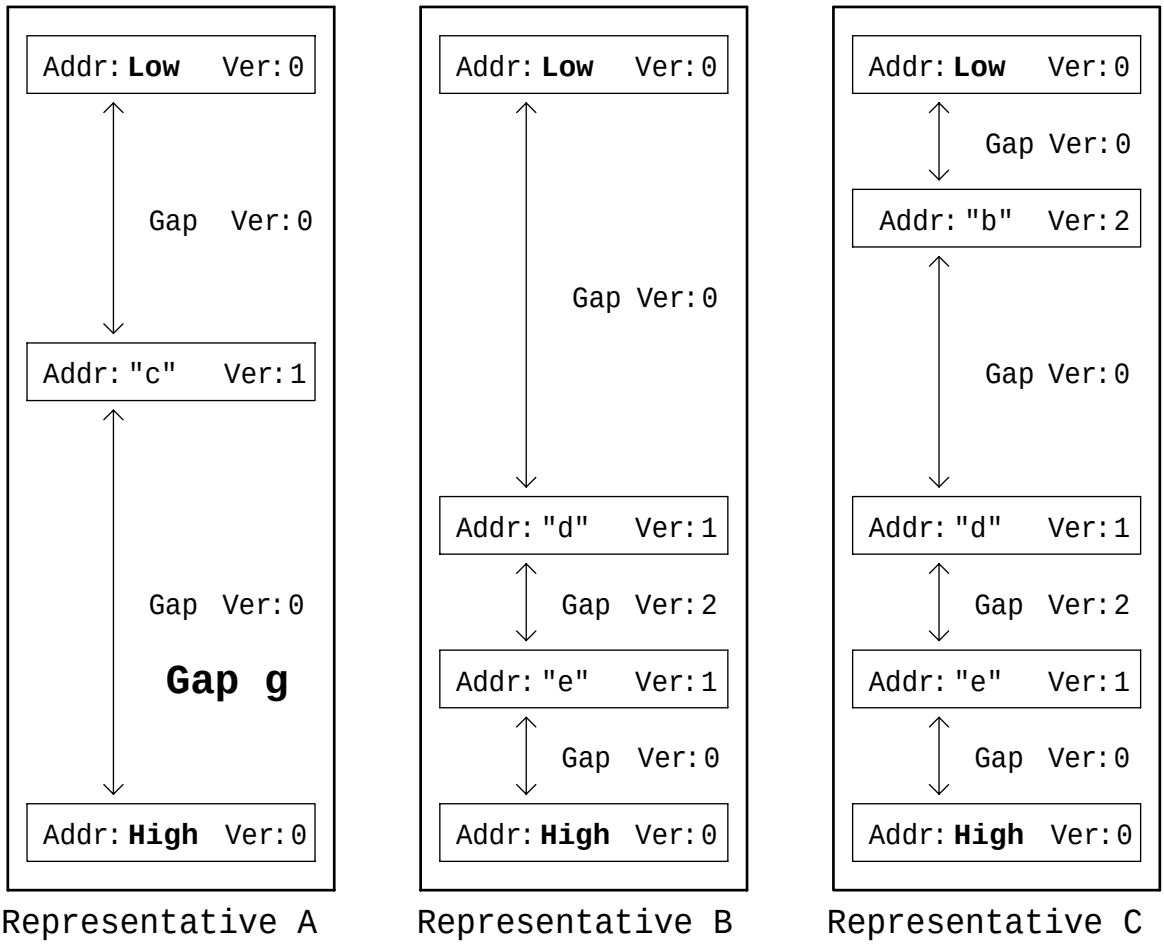

Figure 2-13: RSM For the Illustration of *Region of Currency* and Related Terminology

2.5.1. Proofs

This subsection may be skipped without loss of continuity. However, it is advised that the reader study the proof of Theorem 1 if a thorough understanding of the internal workings of the RSM data structure is desired.

A rigorous proof of Lemma 1 would be tedious, but a detailed proof sketch follows. If a collection of ranges is not in order, it can be reordered. If any pair of ranges in the resulting collection overlap, their union is a range. Thus the pair of ranges can be replaced by the range that is their union. The endpoints of the union are both endpoints of one of the original ranges. This procedure is repeated until none of the remaining ranges intersect. At this point, the collection is in canonical form, and the union of the ranges in the collection is identical to the union of the ranges in the original collection. Any two canonical collections of ranges over a dense address space that are not identical have different unions, hence the canonical form of the collection is unique. □

Lemma 2 follows immediately from the definition of currency over a region. By this definition, if g is current over (a_1, a_2), there are no entries for addresses in (a_1, a_2) with a higher version number than g's. Thus there are no occupied addresses between a_1 and a_2, and a_1 is a_2's real predecessor. □

Now we turn our attention to Theorem 1. We assume that the address space is dense. This assumption is made without loss of generality by the following argument. Any totally ordered set can be embedded in a dense set: given a sparse address space A_s there exists a dense address space A_d such that $A_s \subseteq A_d$. (For example, the integers from 1 to 10 can be embedded in the rationals from 1 to 10.) If we prove that Theorem 1 holds for an address space, we have also proven it for any subset of that address space, as the user could arbitrarily restrict his operations to members of that subset. Thus a proof that Theorem 1 holds for all dense address spaces implies that it also holds for all sparse address spaces. Note that this has no implications with regard to actual system implementation. It merely facilitates the proof.

The proof of Theorem 1 is by structural induction. For the base case, we observe that the theorem holds for an RSM in its initial state: each representative contains a single gap whose region of currency is (**Low**, **High**), and the (distinguished) addresses **Low** and **High** are occupied.

For the induction step, we must show that if the theorem holds for a given system state, then it holds for all states reachable from that state via a single Write or Erase operation. We shall consider these operations in turn. For each operation, we must show that the gaps contained in the representatives comprising the write quorum and the gaps contained in the representatives outside the write quorum satisfy the required condition after the operation. We further subdivide these gaps into those whose region of currency changes as a result of the operation and those whose region of currency remains unchanged.

First we show that the induction holds for Writes. The Write operation does not erase any address from the RSM, so any range whose endpoints were occupied prior to the Write will still have its endpoints occupied after the Write. Therefore, all gaps whose region of currency remains unchanged by the Write will still satisfy the induction hypothesis after the operation (given only that they satisfied it before). Thus, we need only consider the gaps whose regions of currency are altered by the Write operation.

The regions of currency of gaps in representatives outside of the write quorum for an Write operation are affected only if they are current over the region $\{a\}$, where a is the address being written. The new entry for this address will have a higher version number than these gaps, so the Write operation will have the effect of removing a from their regions of currency. By hypothesis, the old region of currency of each of these gaps is

expressible as a finite union of ranges whose endpoints are occupied addresses. Let us call these ranges $\{r_i \mid_{i=1}^{n}\}$, $r_i=(a_{i1}, a_{i2})$. Lemma 1 allows us to assume without loss of generality that the collection of ranges is in canonical form.

Since the gaps in question contain a in their region of currency, one of the r_i must contain a. Let us call this range r_q. (The value of q may be different for each gap in question.) When a is deleted from such a gap's region of currency, the resulting region will consist of the ranges:

$$\{r_i \mid_{i=1}^{q-1}\} \cup \{(a_{q1}, a),\ (a, a_{q2})\} \cup \{r_i \mid_{i=q+1}^{n}\}.$$

But a and all of the a_{ij} are occupied after the Write operation, so the induction hypothesis is preserved in all representatives outside of the write quorum.

Within the write quorum one of two things can happen. If an entry is already present for a, no gap's region of currency will be affected by the operation. If no entry for a exists, then the gap g into which the address falls will be split into two new gaps. Let us call them g_1 and g_2. By the induction hypothesis, g's region of currency can be expressed as a finite union of ranges whose endpoints are occupied. Let us call them $\{r_i \mid_{i=1}^{n}\}$. We assume the ranges are in canonical form, by Lemma 1. If a is in g's region of currency, it is in one of the r_i. Let us call this range r_q. Then g_1's region of currency will consist of the ranges:

$$\{r_i \mid_{i=1}^{q-1}\} \cup \{(a_{q1}, a)\},$$

and g_2's region of currency will consist of the ranges:

$$\{(a_{q1}, a)\} \cup \{r_i \mid_{i=q+1}^{n}\}$$

(Figure 2-14). All the endpoints of the ranges comprising g_1 and g_2's regions of currency are occupied after the Write. If the address being written to falls outside of the original gap's region of currency, let q be the largest integer such that $a < a_{q1}$. Then g_1's region of currency will consist of the ranges $\{r_i \mid_{i=1}^{q}\}$, and g_2's region of currency will consist of the ranges $\{r_i \mid_{i=q+1}^{n}\}$. Thus, the induction hypothesis is preserved in all representatives for Write operations.

Finally, we show that the induction holds for Erase operations. In each representative in the write quorum, a new gap is created whose region of currency is (p, s), where p is the real predecessor of the address being erased and s is the real successor. If p was not already present in a representative, it is inserted. The region of currency of the new gap

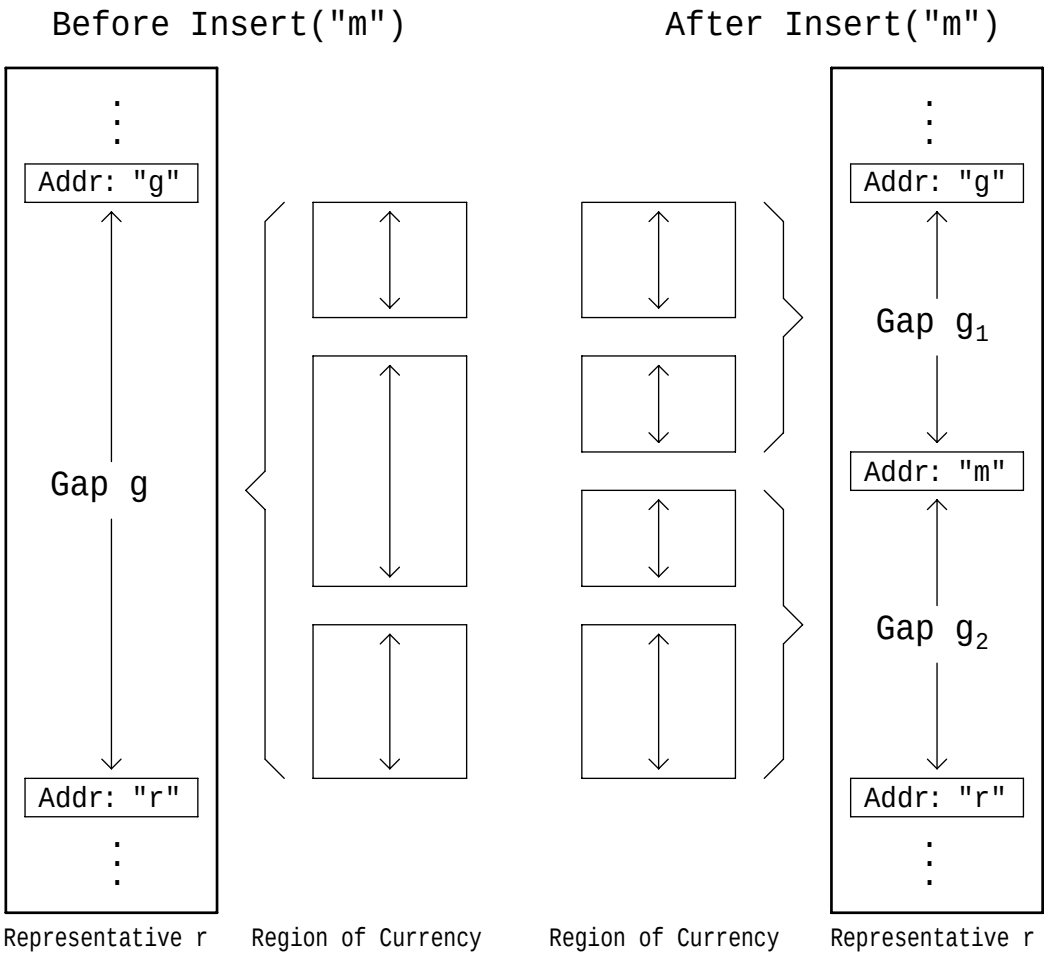

Figure 2-14: Effect of Write Operation on Regions of Currency Within Write Quorum

extending backward from *p* consists of the ranges before *p* previously in the canonical form of the region of currency of the gap from which the new gap was split off. Similarly, if *s* is inserted, the gap extending forward from *s* will have as its region of currency the ranges after *s* previously in the canonical form of the region of currency of the gap from which the new gap was split off. (Figure 2-15) The addresses *p* and *s* are, by definition, occupied, so all of the gaps whose regions of currency are modified still satisfy the induction hypothesis.

The gaps whose regions of currency were not modified could not have had any ranges bounded by *a* in the canonical forms of their regions of currency. If this were the case, the gaps would of necessity have covered or bordered *a*. In either case, the deletion of *a* from the representative would have modified the gaps' region of currency, which, a priori, did not happen. Thus, these gaps satisfy the induction hypothesis given only that they satisfied it before the Erase. Therefore, the induction hypothesis holds within the write quorum.

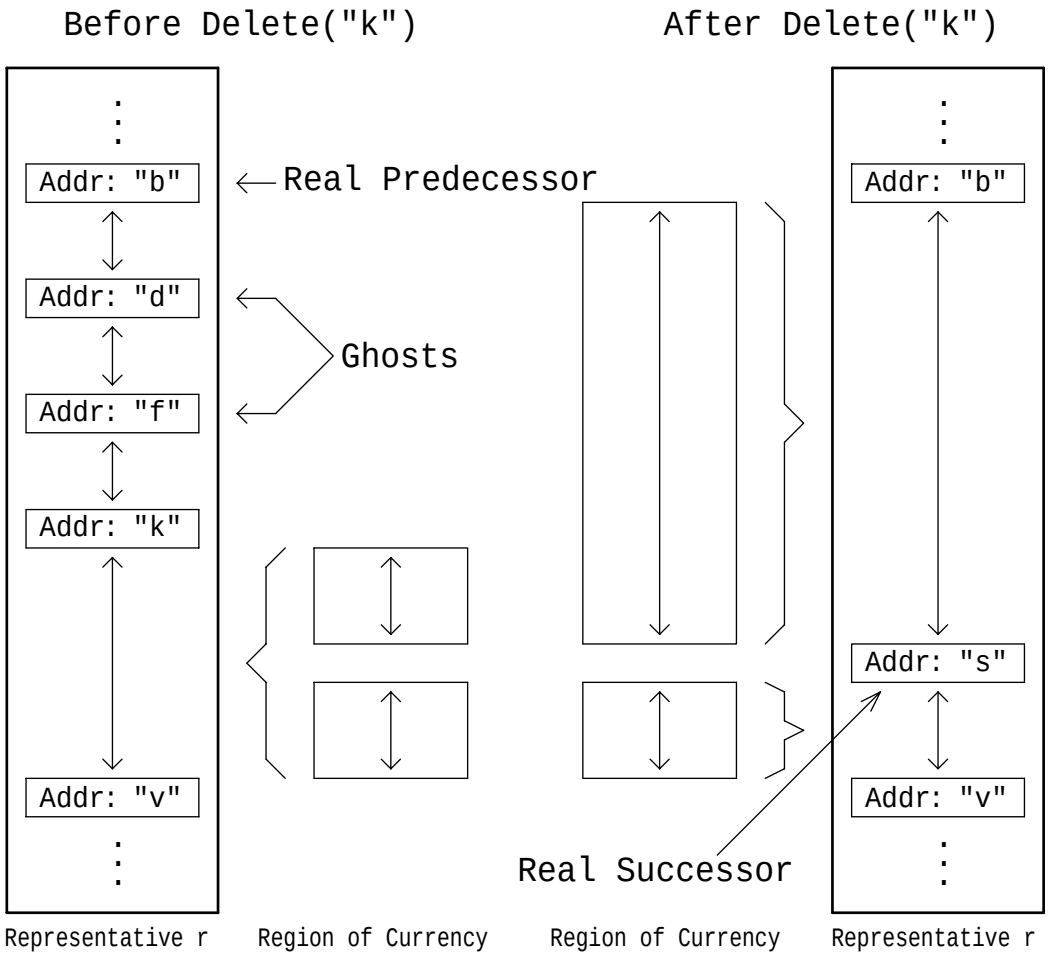

Figure 2-15: Effect of Erase Operation on Regions of Currency Within Write Quorum

Outside of the write quorum the situation is as follows: The new gap in the representatives of the write quorum covers (*p*, *s*). Gaps whose regions of currency did not intersect this region are unaffected. The new gap has a higher version number than all others in this region, so any portions of other gaps' regions of currency that lay in (*p*, *s*) are no longer in their regions of currency. Thus, the deletion has the effect of removing ranges entirely contained within (*p*, *s*) from the canonical forms of the gaps' regions of currency. By Lemma 2, any range in the canonical form that had *a* as one endpoint must have had *p* or *s* as its other endpoint and so was contained in (*p*, *s*). Thus, all ranges remaining in the canonical form after the deletion are bordered by addresses other than *a* that were previously occupied. But these addresses are still occupied after the deletion, so the induction hypothesis is preserved for gaps outside of the write quorum in an Erase operation. This completes the proof. □

2.5.2. The Algorithm

In this section we describe our real predecessor algorithm. A proof of the correctness of the algorithm is presented as the algorithm is described. A formal statement of the algorithm is given in Figure 2-16.

The node determining a's real predecessor asks each representative in a read quorum to return the gap that covers a or has a as its high boundary. All of these ''predecessor gaps'' cover some range in common, which we call (a_1, a). (a_1 is the highest of the low endpoints of the returned gaps.) Furthermore, the gaps represent information from an entire read quorum, so no representative contains any higher version numbered information pertaining to any address in (a_1, a). Thus, the predecessor gap with the highest version number, which we call g_{curr}, is current over the region (a_1, a).

By Theorem 1 and Lemma 1, g_{curr}'s region of currency can be expressed in canonical form as a union of ranges bounded by occupied addresses. Since (a_1, a) is in g_{curr}'s region of currency, it must be contained entirely in one of these ranges. The high end point of this range is a (since a is occupied prior to the Erase operation), and the low end point is a's real predecessor, by Lemma 2. Of course, the low end point must lie within g_{curr} or at its low boundary, which we call a_2.

In the final stage of the algorithm, each representative in the read quorum is asked to return the entry for the highest address less than a and greater than a_2 whose version number is higher than g_{curr}'s. If a representative contains no entry in the specified range with a sufficiently high version number, it returns a message to that effect. At this point, two things can happen. If none of the representatives return an entry, g_{curr}'s low end point, a_2, is a's real predecessor. If one or more such entries are returned, the highest address for which an entry is returned, p, is a's real predecessor, by the following argument.

All addresses for which entries are returned must lie outside g_{curr}'s region of currency, so p lies outside of g_{curr}'s region of currency. Therefore, the low endpoint of the range in the canonical form of g_{curr}'s region of currency that contains (a_1, a) must be $\geq p$. But no address between p and a, exclusive, is currently occupied; if there were such an address, at least one of the representatives in the read quorum would have contained a current entry for it, which it would have returned in the final stage of the algorithm. Thus g_{curr} is current over the range (p, a). Both p and a are occupied, so, by Lemma 2, p is a's real predecessor. □

```
RealPredecessor(IN addr: address;
                OUT pred: address, GapVer: version);
{ Returns addr's real predecessor and the highest version number
  associated with an address in the range bounded by addr and addr's
  real predecessor, exclusive. }
var quorum: array[1..R] of RsmRep,
    MaxGapVer, CandGapVer: version,
    MaxGapAddr, CandGapAddr, CandAddr: address,
    MaxGapRep: integer;
    CandFlag: boolean;
begin
    quorum := CollectReadQuorum();

    { Get info on predecessor gaps in each rep in the read quorum and
      find out which rep has the gap with the highest version number.
      (Called g-curr in accompanying text.) }
    MaxGapVer :=  -1;  {Lower than any legitimate version number}
    for i := 1 to R do
    begin
        RepPredecessor(quorum[i], addr, CandGapAddr, CandGapVer);
        if CandGapVer > MaxGapVer then
        begin
            MaxGapVer := CandGapVer;
            MaxGapAddr := CandGapAddr;
            MaxGapRep := i;
        end
    end;
    GapVer := MaxGapVer;
    pred := MaxGapAddr;  { Tentatively }

    { Find closest entry that supersedes g-curr in any rep in the read
      quorum.  This will be the real predecessor. }
    for i := 1 to R do
    begin
        if i <> MaxGapRep then
        begin
            RepSuperseder(quorum[i], addr, MaxGapAddr, MaxGapVer,
                          CandFlag, CandAddr);

            { If this rep has a candidate for real pred, and it's closer
              than closest candidate thus far, tentatively select it. }
            if CandFlag and (CandAddr > pred)
                pred := CandAddr;
       end
    end
end
```

Figure 2-16: Real Predecessor Operation

2.5.3. Enhancements to the Real Predecessor Algorithm

As in the other procedures presented, efficiency is sometimes sacrificed for clarity in the RealPredecessor procedure of Figure 2-16. There are several additional improvements that would be made in any practical implementation of the algorithm. The procedure would check if the second round were necessary before doing it; if the highest predecessor address returned in the first round has a higher version number than any of the returned gaps that cover it, then this address must be the real predecessor, and there is no need to continue searching.

This technique can be used to reduce message traffic even further by having each representative return several gaps and entries preceding the address being erased, instead of returning just one. This increases the region over which the procedure has ‘‘complete information’’ (i.e. entries or covering gaps from all representatives in the read quorum). If there is any address in this region for which an entry has a higher version number than all covering gaps, then the entry represents an occupied location. In this case, the highest such address is the real predecessor, and no second round is necessary.

The number of entries returned by the representatives in the first stage of the algorithm controls a performance tradeoff between execution time at the nodes and inter-node message traffic. If many entries are returned, it is likely that the second round of information exchange will not be necessary; however, the execution time at each node is proportional to the number of entries sent. The number of entries between the address being erased and its real predecessor will, on average, be half of the address’s delete list size. (Recall that the delete list consists of all of the ghost entries between an address’s real predecessor and its real successor.) Thus, the formula developed in Section 3.3.5 that enables us to predict the average length of a delete list aids us in choosing an appropriate number of entries to return in the first stage. In fact, the limiting behavior described in Section 3.3.6 shows that the second stage of the algorithm can almost always be avoided if several entries are returned in the first stage.

Even if the second stage is required, it is not necessary to ask for additional information from all of the representatives in the read quorum. Any representative that has already sent entry or gap information for the entire range that has been determined to contain the real predecessor (the range covered by g_{curr}) has no more information to add and can be omitted from the second round.

The observation in the previous paragraph can be applied even more aggressively. If several gaps are tied for the highest version number in the first round, the procedure should pick the smallest gap (i.e. the one with the highest low bound) to serve as g_{curr}. If any representatives have already returned a valid *predecessor candidate* (i.e. an entry for an address that lies in g_{curr} but has a higher version number than g_{curr}), the

representative has no more information to add, and needn't be included in the second round. If any valid predecessor candidates are returned in the first round, the real predecessor must be greater than or equal to the highest such candidate. Any representative that has already sent entry or gap information for the entire range from the highest such candidate to *addr* cannot possibly have a higher candidate; these representatives can also be omitted from the second stage.

The real predecessor and real successor can be determined simultaneously by putting requests and responses for both tasks in each message, thus reducing by almost one-half the message traffic required to find the real predecessor and successor. In the actual implementation, there would be a single RealNeighbors procedure instead of separate RealPredecessor and RealSuccessor procedures. The procedure would initially ask for gaps and entries surrounding the address on both sides. If this did not provide enough information to find the address's real predecessor and successor, it would send a request for a superseder of either or both ''current gaps,'' as required.

In the procedure for the Erase operation in Figure 2-12, the address to be erased is looked up prior to determining its real neighbors. In practice the read would be combined with the first stage of the real neighbors determination.

The critical factor determining the execution speed of the RSM operation procedures presented is the number of rounds of small, fixed length messages sent in performing the operations. Thus, we use this number as a complexity measure for our algorithms. Our real predecessor algorithm, with the improvements described, is extremely fast in the average and worst cases. The average performance of this algorithm is close to the trivial lower bound of one exchange of messages with each member of a read quorum. The worst case performance is two rounds. The Erase operation requires one additional round to coalesce the range between the real predecessor and successor.

The procedure, including the improvements, is easy to implement. It also has the following useful property. The correctness of the algorithm does not depend on the fact that the address whose real predecessor is being determined is occupied. Thus, it can be used to locate the real neighbors of any address, regardless of whether it is occupied. This could, for instance, be used to implement a RangeErase operation, which erased all of the addresses between one address and another. This operation would require no more message transmissions than the basic Erase operation.

2.6. Correctness Arguments

The correctness of an RSM's operations depends on Read always returning current information about an address. Because every read quorum intersects every write quorum, Read will return current information as long as that information has a version number greater than that of any non-current information. These correctness conditions are the same as those required for Gifford's file replication algorithm.

Two phase locking applied to the locks specified in Section 2.4.1 guarantees the serializability of transactions at any single representative. Traiger et al. have shown that if all nodes participating in a set of distributed transactions follow two phase locking protocols that guarantee the serializability of transactions at individual nodes, then the resulting global schedule is equivalent to some serial schedule of transactions [56]. Thus, the RSM replication algorithm preserves the serializability of transactions that use it.

The Write operation sets the version number of the entries that it creates or modifies to be greater than the highest version number previously associated with the address to which it is writing. Therefore, the current data for each occupied address has a version number greater than that of any non-current data for the address.

Erase coalesces the range between the real predecessor and real successor of the address to be erased. By the definitions of real predecessor and real successor, there can be no current entries (other than entries for the address being erased) in the range to be coalesced. The Erase operation assigns to the gap covering the coalesced range a new version number that is higher than any version number previously associated with any address in that range. Therefore, as with Write, the current data for each address in the range has a version number greater than that of any out-of-date data for that address. Erase can cause additional entries to be created for the real predecessor and successor of the address being erased. But these entries have a lower version number than the current entries for the real predecessor and successor, and every write quorum still contains a current entry.

Chapter 3
The Performance of Replicated Sparse Memories

In this chapter, we analyze the performance of the RSM data structure developed in Chapter 2. We present the results of simulations that suggest that the average time and space performance of the data structure is good over a wide range of quorum choices. A Markov model of the RSM under random use is constructed and analyzed using *balance equations*. Predictive formulas for the performance measures studied in the simulation are derived from the solutions of the balance equations. The predictions of the analysis agree remarkably well with the performance data gathered in the simulations. The results of the analysis indicate that the favorable performance observed in the simulations extends to any possible quorum choice and any random operation mix.

In essence, the simulations and analysis show that the RSM data structure is *self-cleaning* under any possible quorum choice and any random operation mix. The ‘‘incremental garbage collection’’ that is performed automatically as part of the Erase operation is sufficient to keep the RSM representatives from accumulating a large quantity of out-of-date information. This ensures that the space cost of the data structure and the time costs of the algorithms that manipulate it are kept low.

The remainder of this chapter is organized as follows. Section 3.1 describes the system studied in the simulations and analysis, and the performance measures of interest to us. Section 3.2 presents the results of the simulations. Section 3.3 develops the model, analyzes it, and presents performance predictions. Section 3.4 discusses the realism of the system under study and the applicability of our predictions to practical applications of the RSM data structure.

3.1. The System

For the purposes of our performance studies, we subdivide Write operations into two classes: Writes to unoccupied addresses, which we call Inserts, and Writes to occupied addresses, which we call Updates.

The system studied in the simulations and analysis consists of an empty RSM into which a certain number of Inserts are performed initially. Thereafter, Inserts, Updates

and Erases occur sequentially with equal likelihood. Addresses for Inserts are chosen randomly from unoccupied addresses, and addresses for Updates and Erases are chosen randomly from occupied addresses. Read and write quorums are selected randomly for each operation. No Read operations are performed, as they would have no effect on the contents of the RSM, and yield no interesting performance data.

The address space used in the simulations consisted of the integers from one to one billion. The mathematical model is described in sufficient generality to apply to any finite address space. It does not, in general, make sense to consider the system with an infinite address space, as addresses to be inserted are chosen at random from those not already occupied. If a countably infinite address space were used, this would amount to selecting an object at random from a countably infinite set, an operation that is not well defined. Interestingly, the cardinality of the address space does not affect the analysis, except insofar as it affects the validity of several simplifying assumptions. This fact is discussed at greater length in Section 3.3.8.

Various measures can be used to evaluate the performance of our data structure. In our view, the most important performance measure is the number of rounds of message exchanges with a read or write quorum necessary to perform each RSM operation. With one exception, this measure is a constant that does not vary from instance to instance of a given operation. The exception is the Erase operation, which, with the enhancements suggested in Section 2.5.3, requires either two or three rounds of messages depending on the results of the first round. The communications cost of the RSM operations are summarized in Table 3-1.

Operation	Rounds to Read Quorum	Rounds to Write Quorum	Total # Rounds	Total # Messages
Read	1	0	1	$2R$
Write	1	1	2	$2(R+W)$
Erase	1 or 2	1	2 or 3	$2(R+W)$ or $\leq 2(2R-1+W)$

Table 3-1: Communications Costs of RSM Operations

The node doing an RSM operation has to send RPCs to read and write quorums and, in the case of read quorums, scan the responses to determine the current information. The work done by this node is proportional to the total number of messages required for an operation, and is generally small. More interesting is the work done by the nodes storing the RSM representatives. All RSM representative operations except for RepSuperseder and RepCoalesce consist of locating and reading or modifying a single entry or gap. The time required to perform this operation depends on the number of entries in the representative and the data structure used to store the entries. If balanced trees are used, the time is proportional to the logarithm of the number of entries. The storage space

required at each representative is proportional to the number of entries stored at the representative.

Thus, the first performance measure we concentrate on in our performance studies, which we call the *size ratio*, is the ratio of entries in an RSM representative to occupied addresses in the RSM. The size ratio indicates the storage required at each representative as a function of the storage required for a single site sparse memory. A size ratio of one indicates that a representative has exactly as many entries as a single site sparse memory. The simulations measured the size ratio directly, while the analytic model allows us to break the size ratio down into three *composition ratios* based on a classification of RSM entries into three categories. The size ratio is the sum of the three composition ratios.

In the second step of the Erase operation (RepSuperseder), each representative has to scan the delete list for the address being erased. In the third step (RepCoalesce), each representative has to coalesce the delete list into a single gap. In both steps, the total work required is proportional to the delete list length. Thus, the second performance measure we study in our simulations and analysis is the *delete list length*.

As explained in Section 2.5.3, the second step of the Erase operation is necessary only if one or more nodes in the read quorum did not return their entire delete list in the first step. Knowing the expected value of the delete list length allows us to ask for enough information in the first step so that the second step will usually be unnecessary. Of course this would not be feasible if the expected value of the delete list length were high. However, this turns out not to be the case.

In summary, the size ratio characterizes the space complexity of the algorithm. The size ratio and delete list length characterize the significant components of the time complexity of our algorithm. Knowledge of the expected delete list length is useful to ensure that the first round of the Erase operation returns enough data so that the second round is usually unnecessary. The size ratio and delete list length are the performance measures that form the basis of our performance studies. In the analysis, the size ratio is further subdivided into composition ratios, which tell us more about how the storage space is being used.

3.2. Simulation Results

The shaded bars in Figures 3-1 and 3-2 show the average size ratios and delete list lengths measured in simulations for a variety of RSM configurations. (The unshaded bars show predicted values obtained from the mathematical model in Section 3.3.) In the simulations, each RSM initially had one thousand occupied addresses. The duration of each simulation was twenty thousand operations, and performance measures were gathered during the final ten thousand operations.

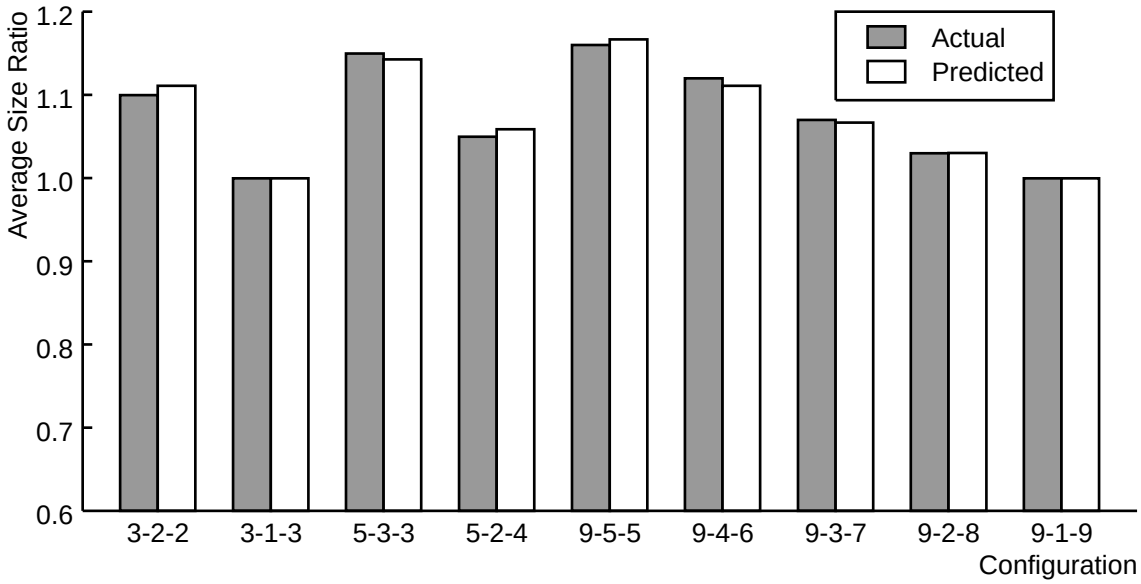

Figure 3-1: Average Size Ratios for Various RSM Configurations

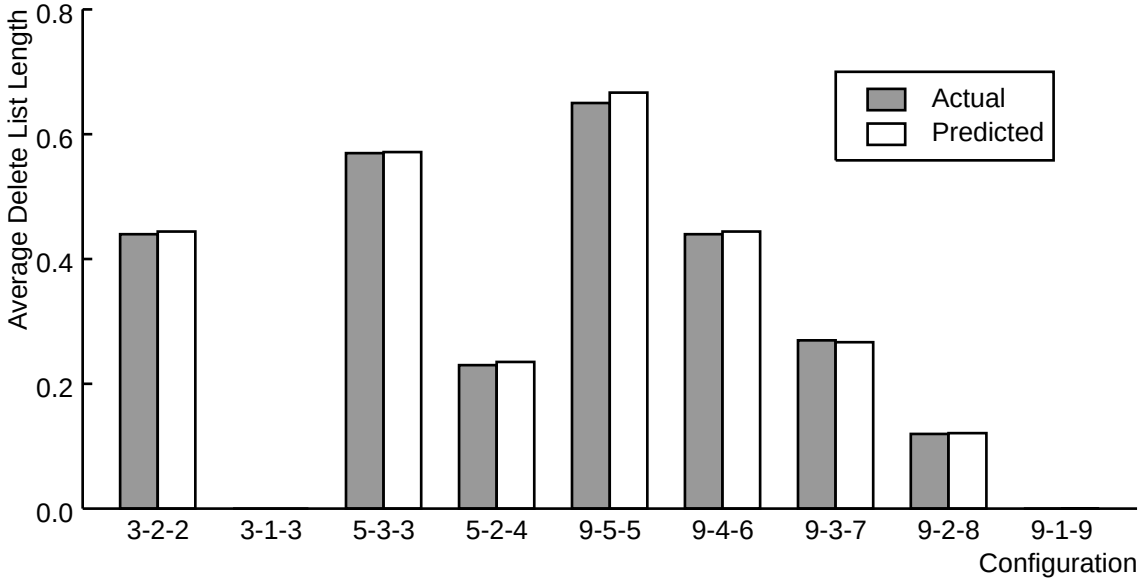

Figure 3-2: Average Delete List Lengths for Various Configurations

The simulation results in Figure 3-1 show that the average size ratio remains very close to one for all of the configurations tested. Thus the storage required at each representative and the time required to locate an entry at a representative are only slightly higher than for a single site sparse memory. The results in Figure 3-2 show that the average delete list length is less than a single entry for every configuration tested. This implies that the second and third steps of the Erase operation will run very quickly at the representatives, and the second step will rarely be necessary if a few entries are returned in the first step. Note that the average delete list length for the 3-1-3 and 9-1-9 configurations is zero, and the average size ratio for these configurations is one. For

X-1-X configurations, our algorithm, like all weighted voting algorithms, degenerates to *universal update*. All representatives are identical, containing an up-to-date entry for every occupied address, and no out-of-date entries.

More detailed simulation results for 3-2-2 RSMs with one hundred, one thousand, and ten thousand addresses initially occupied are shown in Table 3-2. The duration of each of these simulations was two hundred thousand operations, with performance data gathered during the final one hundred thousand operations.

Size Ratio			
Initial Number of Occupied Addresses	**Mean**	**Max**	**Std Dev**
100	1.11	1.27	.03
1,000	1.11	1.19	.02
10,000	1.11	1.13	.01

Delete List Length			
Initial Number of Occupied Addresses	**Mean**	**Max**	**Std Dev**
100	.44	9	.81
1,000	.44	9	.81
10,000	.44	10	.81

Table 3-2: Detailed Simulation Results for 3-2-2 RSMs

These additional simulations suggest that the average values of the performance measures do not depend on the initial number of addresses occupied. Thus, average space requirements appear to be proportional to the number of occupied addresses in the RSM, just as in a single site sparse memory. The time requirements depend on the number of occupied addresses in the RSM in the same manner as for a single site sparse memory. The standard deviation of the size ratio decreases as the number of occupied addresses increases. This is easily explained by the fact that the numerator and denominator of the size ratio are the number of entries in a representative and the number of occupied addresses in the RSM, respectively. Similar variation should be observed in both of these random processes regardless of the RSM size, but a given variation in the numerator or denominator will cause a greater change in the fraction if the denominator (the number of occupied addresses in the RSM) is small.

The maximum delete list size observed was 10. This is an indication of the *worst case* time to perform the RSM representative operations for the second and third steps of the Erase operation. Care should be taken not to interpret this as the true worst case time for any possible run. Theoretically, a delete list can be as long as the number of addresses that have ever been erased from the RSM. The longer a run, the higher the maximum observed delete list is likely to be. However, the fact that the largest delete list observed in three runs of one hundred thousand operations each was only 10 entries indicates that large delete lists will probably not be a problem in practice. We conjecture that the expected value of the largest delete list observed in a run is logarithmic in the length of the run.

3.3. Analytic Model

3.3.1. Introduction

The algorithm as applied in the simulations was modeled and analyzed to predict various performance characteristics. The goals of the analysis were to increase our confidence in the simulations by corroborating their results, to gain further insight into the behavior of the algorithm, and to produce a fast, reliable method for determining the performance of the data structure in a given application.

In this section, we describe the model and our method of analysis, and present the analysis. A set of formulas to predict performance characteristics are derived in the analysis. These formulas are used to check the results obtained from the simulations and predict performance trends exhibited by the algorithm under various conditions. The analysis is extended to handle more general operation mixes.

3.3.2. Construction of the Model

The system can be modeled as a Markov chain in a straightforward fashion. One state corresponds to each possible contents of the entire RSM, henceforth called a *system state*. The transitions correspond to the changes in system state effected by the operations. Transition probabilities are induced by the fact that the operation to be performed (Insert, Update, or Erase), the address to be operated upon, and the write quorum are chosen at random.

In the simulations, the system appeared to display equilibrium behavior: each system attribute being monitored approached an average value that did not vary over multiple runs of sufficient length. For a Markov model to be of use to us in calculating these values, it too must display this equilibrium behavior. It is sufficient that the model

achieve stochastic equilibrium. The simplest class of Markov chains achieving stochastic equilibrium are those that are finite and irreducible. (By *finite*, we mean that they contain a finite number of states, and by *irreducible*, we mean that each state can be reached from every other state.)

The straightforward model described above does not possess either of the requisite properties. It is not finite, as version numbers can grow without bound. Repeatedly updating a single address produces an infinite sequence of distinct states. Neither is the straightforward model irreducible: once the system leaves *any* state, it can never get back to that state. The version numbers associated with a fixed address in a fixed representative in successive states form an increasing sequence. Any operation results in the version number associated with some address increasing in some representative and it can never return to its original value. However, the model displays an extremely high degree of *lumpability* [34]. That is to say, many states are practically identical to some other state, so sets of similar states can be lumped together to produce a smaller, simpler model. We construct a new model that possesses the desired properties by this process of lumping.

This is not the straightforward task that it might appear to be. The obvious way to deal with the fact that version numbers increase without bound is to equate states where corresponding representatives contain entries for the same collection of addresses and corresponding pairs of entries (or gaps) have the same version number ordering. However, attempts to lump states based on order relations between version numbers run into complications. Our attempts along these lines produced models that were finite but not irreducible. An alternative approach, which involves abandoning the version numbers altogether, produces the desired result. Before we describe it, we must dispense with some preliminaries.

All of the entries in each representative of an RSM can be divided into three classes. A *current* entry is an entry for an occupied address that has the highest version number associated with the address in any representative. Current entries are the only entries that contain up-to-date information. An *outdated* entry is a non-current entry for an address that is still occupied. If an entry is outdated then some other representative contains an entry for the same address with a higher version number. A *ghost* entry is an entry for an address that is no longer occupied. A ghost entry can be thought of as the ghost of an address that used to ‘‘live’’ in the RSM. It should be clear that all entries in a representative fall into one and only one of these classes.

Let us call a representative with all version numbers removed and with the class of each entry (current, outdated or ghost) appended to the entry the *concise representation* of the representative. Note that the concise representation contains no explicit information about the gaps between entries. By extension, we call the collection of concise

representations of all representatives in an RSM the concise representation of the RSM. The concise representation has two properties that make it useful:

PROPERTY 1. *Given the concise representation of a system state, an operation to be performed on the RSM (Insert(addr), Update(addr) or Erase(addr)) and the write quorum selected for the operation, one can determine the concise representation of the resulting system state.*

PROPERTY 2. *All of the important information concerning a system state is fully determined by its concise representation; that is, all system states sharing a concise representation coincide in all* ***important attributes****. By* ***important attributes****, we mean the performance measures: delete list length and composition ratios, and several other attributes for which we assert that equilibrium distributions exist in the analysis of our model.*

The intuition behind the proof of Property 1 is that version numbers are used solely to find out which class an entry belongs to when performing the various operations on an RSM. Let us define class(*a*,*R*), the class associated with address *a* at representative *R*, to be the class of the entry for *a* at *R*, if *R* contains an entry for *a*, or *no entry* if *R* contains no entry for *a*. For every *a* and *R*, each RSM operation affects the value of class(*a*,*R*) in a deterministic fashion based solely on the previous value of class(*a*,*R*), the operation to be performed, and the write quorum *Q* for the operation. The class change effected by each operation is illustrated by the finite automaton shown in Figure 3-3. The states in the automaton represent the current value of class(*a*,*R*). The arcs are labeled with an operation and a membership relation between *R* and *Q*.[5] The automaton mirrors the inner workings of the RSM algorithms, which are described in detail in the construction of the balance equations (Appendix A). The automaton in Figure 3-3 is *deterministic* because each operation describes at most one arc emanating from each state. Property 1 follows immediately from the determinism of the automaton. □

Property 2 must be checked separately for each important attribute. It is true for composition ratios, as the concise representation of a representative clearly contains the same number of current, outdated and ghost entries as the representative itself. It is true for delete list lengths, as delete lists consist of all the ghost entries between two addresses in the RSM, and system states sharing a concise representation represent the same RSM, and have ghost entries for the same addresses at corresponding representatives. The reader can easily check that Property 2 holds for all other system attributes for which we assert the existence of an equilibrium distribution in the analysis. □

[5]The notation on the arc from *ghost* to *noentry* indicates that *a* is between the real predecessor and the real successor of the address being erased.

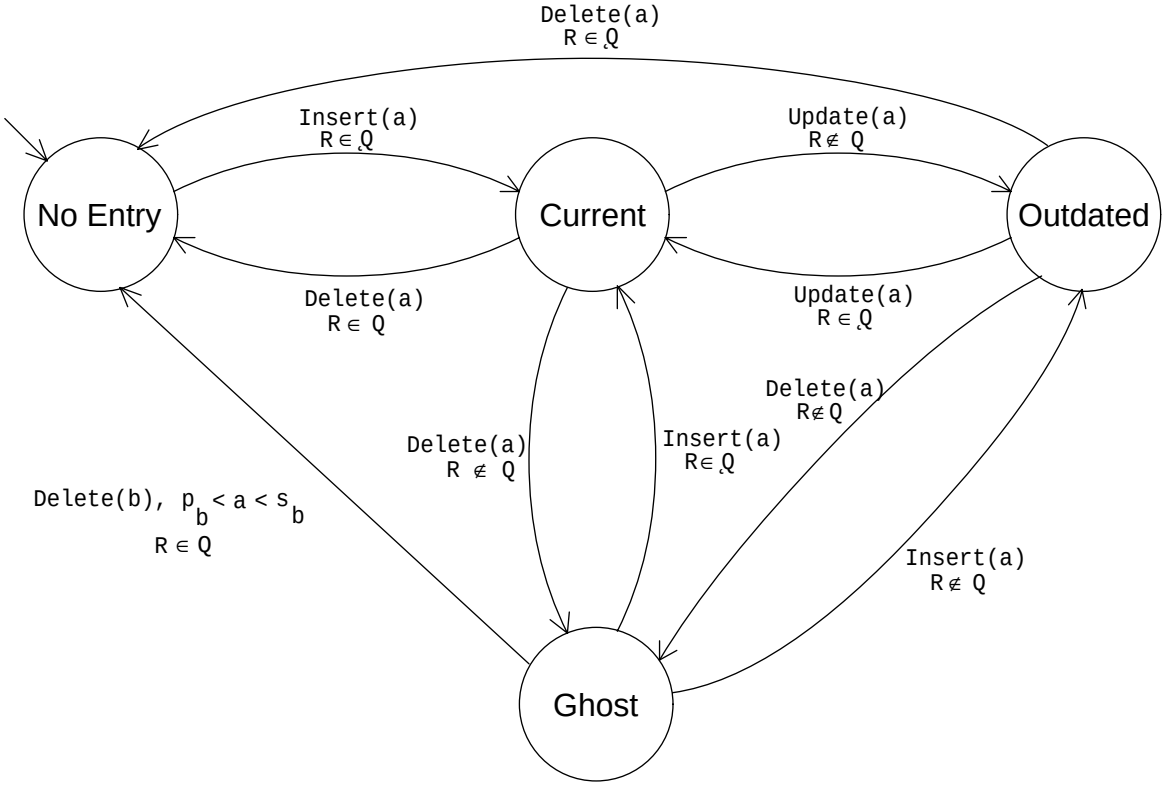

Figure 3-3: Class Change Associated with an RSM Operation

We are now ready to describe the method by which we simplify our model. We define a new model where all system states sharing each concise representation are lumped together to form the model states. Property 1 tells us that the induced transition probabilities in this model are well defined. This is required for the model to be a well defined Markov chain.

The new model is finite by the following argument. The address space is finite, and each representative contains entries for some subset of the address space. Each entry belongs to one of the three classes. Therefore, there are only a finite number of possible concise representations for representatives. An RSM consists of a fixed number of representatives, so there are only a finite number of possible concise representations for system states. This places a finite upper bound on the number of states in our model.

The model is irreducible by the following argument. From any system state, it is possible to reach a system state where all representatives contain no entries (except for the permanent entries for **Low** and **High**). This can be accomplished as follows: first erase all of the occupied addresses in any order with any write quorums. At this point, all of the representatives can contain only ghost entries. If a single address is inserted into the RSM and then erased using the same write quorum, all of the representatives in the quorum will be completely empty. Repeat this Insert-Erase process as many times as necessary to include each representative in at least one write quorum. All system states

where none of the representatives contain any entries have the same concise representation hence they are represented by a single state in the model. But this state also represents the initial system state, from which all other system states can be reached. Thus, any model state reachable from the initial state can be reached from every state.

The Markov model achieves stochastic equilibrium, because it is finite and irreducible. There is one other property that the model must have in order to fulfill our requirements: it must not lump together system states that are not really equivalent. In other words, all system states represented by each model state must be functionally identical in the sense that they coincide in all attributes for which we wish to infer the existence of an equilibrium distribution. However, this is precisely what Property 2 tells us.

3.3.3. Method of Analysis

Our model is guaranteed to achieve stochastic equilibrium, so it is theoretically possible to determine the precise probability of being in any state. In practice, this would be impossible due to the huge size of the system. Also, the resulting probability distribution would not be particularly informative as such, and the processing necessary to derive any useful figures from it would be prohibitive due to its size. However, the existence of this model proves that any attributes common to all system states represented by each state of the model have well defined expected values. Thus it makes sense to formulate relationships among such expected values and solve for them.

The performance characteristics of primary concern to us are all intimately related to the composition of each representative in terms of the three classes into which entries are divided. As a consequence of the existence of our model we can assert that a dynamic equilibrium exists in each of these classes in each representative. These assertions can take the form of *balance equations* equating the rates of flow into and out of each category in a single representative. Such equations hold equally well for all of the representatives in the RSM due to the symmetry of the system. In the course of the analysis, we focus our attention on a single representative, but the results apply to every representative in the RSM.

These balance equations are naturally constructed in terms of three independent variables, and the system parameters N and W (defined in Section 3.3.4). In constructing the balance equations, we make some simplifying assumptions in the form of approximations in the equations. Each approximation will be noted and justified. The resulting equations constitute a linear system that can be solved easily. Expressions for the desired performance measures can be constructed from the independent variables, though we need to make a simplifying approximation in one derivation.

While our analysis was developed independently, related techniques have been used by others. For example, see Yao's *fringe analysis* of 2-3 trees [60].

3.3.4. Formulation of Balance Equations

The following variables are used in formulating the balance equations. Capital letters are used to represent stochastic variables and constants (system parameters). Small letters represent unknowns in the balance equations. We use the notation $a \in RSM$ to indicate that address a is occupied in an RSM. An RSM with no occupied addresses is said to be *empty*.

- C The number of current entries in the representative under observation.
- O The number of outdated entries in the representative under observation.
- G The number of ghost entries in the representative under observation.
- E The total number of entries in the representative under observation. Note that $E = C + O + G$.
- A The number of addresses currently occupied in the RSM.
- D_a The number of entries in the *delete list* of address a in the representative under observation. (The delete list of an address consists of all of the ghost entries in a representative between the address's *real predecessor* and its *real successor*.)
- D $(\sum_{a \in RSM} D_a)/A$. D is the average delete list length in the representative under observation. Note that D is only defined in states where $A \neq 0$ (i.e. the RSM is non-empty).
- c' $\mathbf{E}[C/A]$ The expected value is taken over all states that represent non-empty RSMs. C/A is the fraction of occupied addresses that have current entries in the representative under observation. Thus, c' is equal to the probability that a randomly chosen occupied address has a current entry in the representative under observation.
- o' $\mathbf{E}[O/A]$ The expected value is taken over all states that represent non-empty RSMs. O/A is the fraction of occupied addresses that have outdated entries in the representative under observation. Thus, o' is equal to the probability that a randomly chosen occupied address has an outdated entry in the representative under observation.
- d $\mathbf{E}[D]$ The expected value is taken over all states representing non-empty RSMs. d is the expected value of the average length of a delete list in the representative under observation. A simple derivation in Section 3.3.5 shows that d is also the expected length of the delete list of the address being deleted in a transition representing an Erase operation.
- N The number of representatives in the RSM being modeled.
- W The write quorum size for the RSM being modeled.

A formal statement of the rate balance assertion for current entries is:

E[Number of entries entering current class in a chosen representative in one opr]
= **E**[Number of entries leaving current class in a chosen representative in one opr].

The expected values are computed over a space consisting of all the state transitions in our model. Analogous assertions are made for outdated and ghost entries. The expected values can be recast in terms of c', o' and d. These expansions, though relatively straightforward, are somewhat tedious, as they entail examining the inner workings of the RSM operations in great detail. They can be found in Appendix A.

The expansions yield the following balance equations, for current, outdated and ghost entries respectively:

$$c' = \frac{W}{N}$$

$$o' = \frac{(N-3W)c' + 2W}{N+3W}$$

$$d = \frac{N-W}{W}(c'+o').$$

3.3.5. Solution of Balance Equations

The solution to the balance equations derived in the previous section is:

$$c' = \frac{W}{N}$$

$$o' = \frac{3W(N-W)}{N(N+3W)}$$

$$d = \frac{4(N-W)}{N+3W}.$$

The first performance measure for which we desire a formula is:

E[The length of the delete list encountered in an Erase operation].

(The expected value is taken over all state transitions representing Erase operations.) We apply the identity $\mathbf{E}[X] = \mathbf{E}[\mathbf{E}[X|Y]]$, with Y = The system state prior to the operation:

= **E**[**E**[The length of the delete list encountered in an Erase operation
| The system state prior to the operation]]
= **E**[**E**[The length of a delete list in a given system state]]
= **E**[*D*]
= *d* .

The second performance measure is the expected value of the size ratio:

E [*E*/*A*]
= **E** [(*C* + *O* + *G*)/*A*]
= **E** [*C*/*A*] + **E** [*O*/*A*] + **E** [*G*/*A*]
= *c*′ + *o*′ + **E** [*G*/*A*].

The three terms of this expression (**E** [*C*/*A*], **E** [*O*/*A*] and **E** [*G*/*A*]) are the composition ratios. While we cannot exactly express the third term of this expression in terms of our unknowns we can make a very good approximation based on the fact that almost every ghost in a representative appears in two delete lists, that of its real predecessor and that of its real successor. The exceptions are the ghosts before the first occupied address and those after the last, which only appear in a single delete list. But in the vast majority of states, very few ghosts fall into this category. Thus the sum of the lengths of the delete lists for all occupied addresses is approximately equal to twice the number of ghosts. A formal statement of this assumption is:

$$2G = \sum_{a \in RSM} D_a .$$

Dividing both sides of this equation by 2*A* and taking expected values over all states representing non-empty RSMs, we get:

$$\mathbf{E}[G/A] = \mathbf{E}[(\sum_{a \in RSM} D_a)/2A]$$

$$= \frac{1}{2}\mathbf{E}[D]$$

$$= \frac{d}{2} .$$

Substituting back, our formula for the size ratio becomes:

$$\mathbf{E}[E/A] = c' + o' + \frac{d}{2}$$

$$= \frac{2(N+W)}{N+3W} .$$

3.3.6. Results

Figure 3-1 (p. 42) compares the average size ratios observed in the simulations with predictions obtained from the formula developed in the previous section. Figure 3-2 (p. 42) compares actual and predicted average delete list lengths. The predicted values are nearly identical to the observed values. We compared simulation and analysis results for many other system attributes and observed this level of agreement uniformly.

Figure 3-4 shows the expected composition ratios in a 20 - (21−*W*) - *W* RSM, for all possible values of *W*. Figure 3-5 shows expected delete list lengths for these RSMs. Varying the quorum sizes in a fixed size RSM in this manner controls a fairly complex performance tradeoff: increasing the write quorum size increases the availability of the Read operation while decreasing its cost, and decreases the availability of the write operation, increasing its cost. In the Erase operation, the work done at each node decreases, but the number of messages that must be sent increases. At one end of the spectrum (*W*=20) there is the universal update strategy; at the other (*W*=11), there is a strategy where roughly half the representatives are written and half are read. Note that in the universal update strategy, the size ratio is 1 and there are no outdated or ghost entries, as the representatives are just copies of the single site sparse memory. The graphs show that for the spectrum under investigation, the representatives contain at worst 20% more entries than a single site sparse memory and the expected delete list length remains shorter than a single entry.

Figures 3-6 and 3-7 show the expected composition ratios and delete list lengths in (2*i*−1) - *i* - *i* RSMs. Increasing read quorum, write quorum and RSM sizes simultaneously, as illustrated in these graphs represents a fairly straightforward performance tradeoff. As the sizes increase, the availability of all operations increases, but the number of messages that must be transmitted for all operations increases as well. Specifically, the number of representatives that can fail while still maintaining availability of all operations in a (2*i*−1) - *i* - *i* RSM is *i*−1. The flatness of the curves shows that the amount of work at each node in an Erase operation, and the size and makeup of each representative do not vary appreciably over the spectrum. Thus the cost scales up proportionately to the increased availability with no added penalty for very high availability.

Finally, we present some fairly surprising results concerning the limiting behavior of the predictive formulas for the performance measures. First let us examine the expected length of a delete list, *d*. Recall, the formula for *d* is:

$$\frac{4(N-W)}{N+3W}.$$

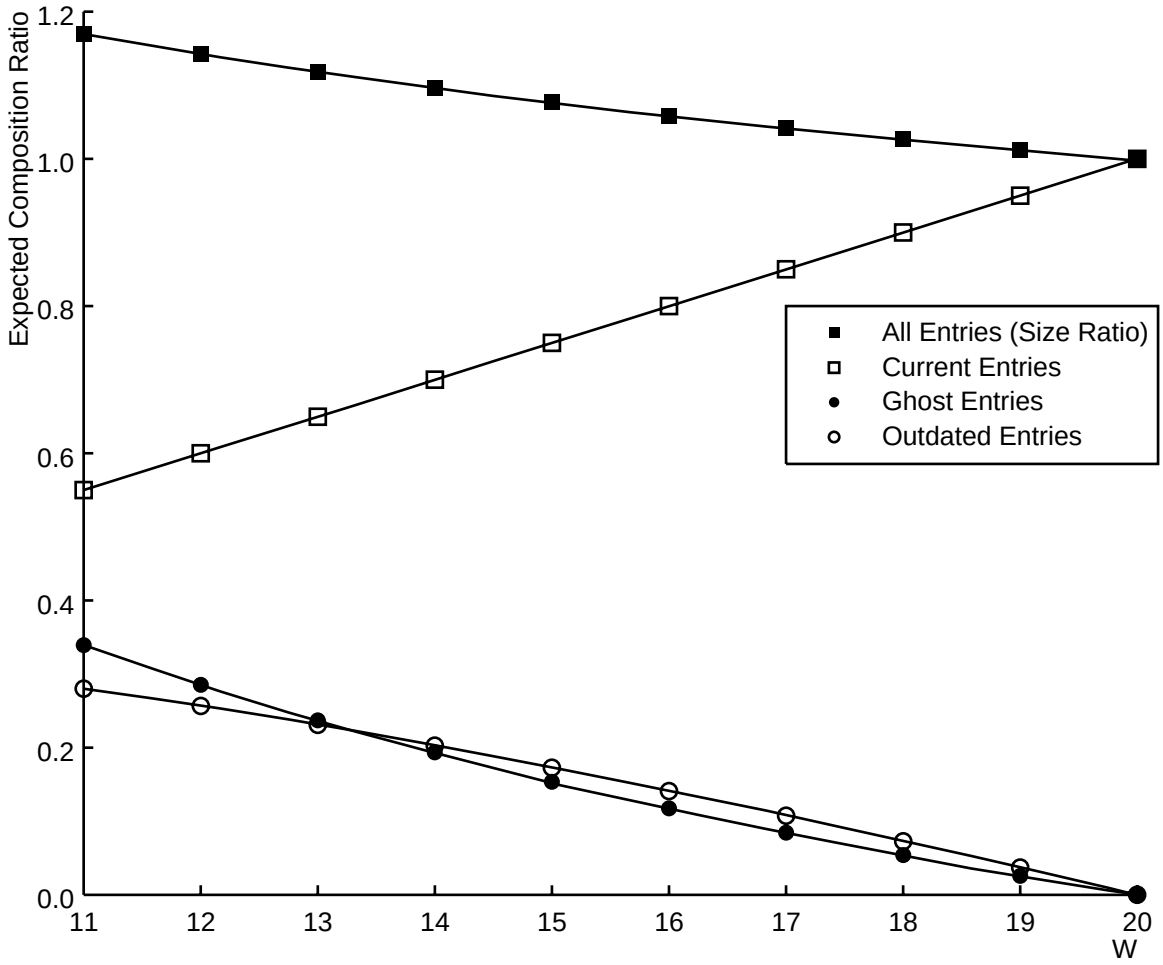

Figure 3-4: Expected Composition Ratios in a 20 - (21−*W*) - *W* RSM

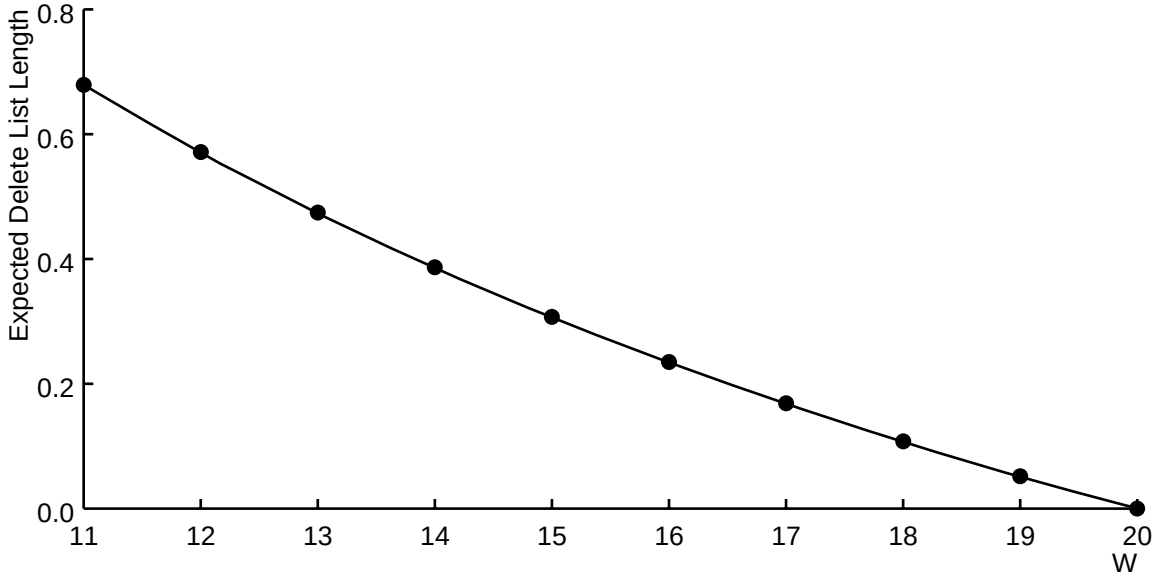

Figure 3-5: Expected Delete List Lengths in a 20 - (21−*W*) - *W* RSM

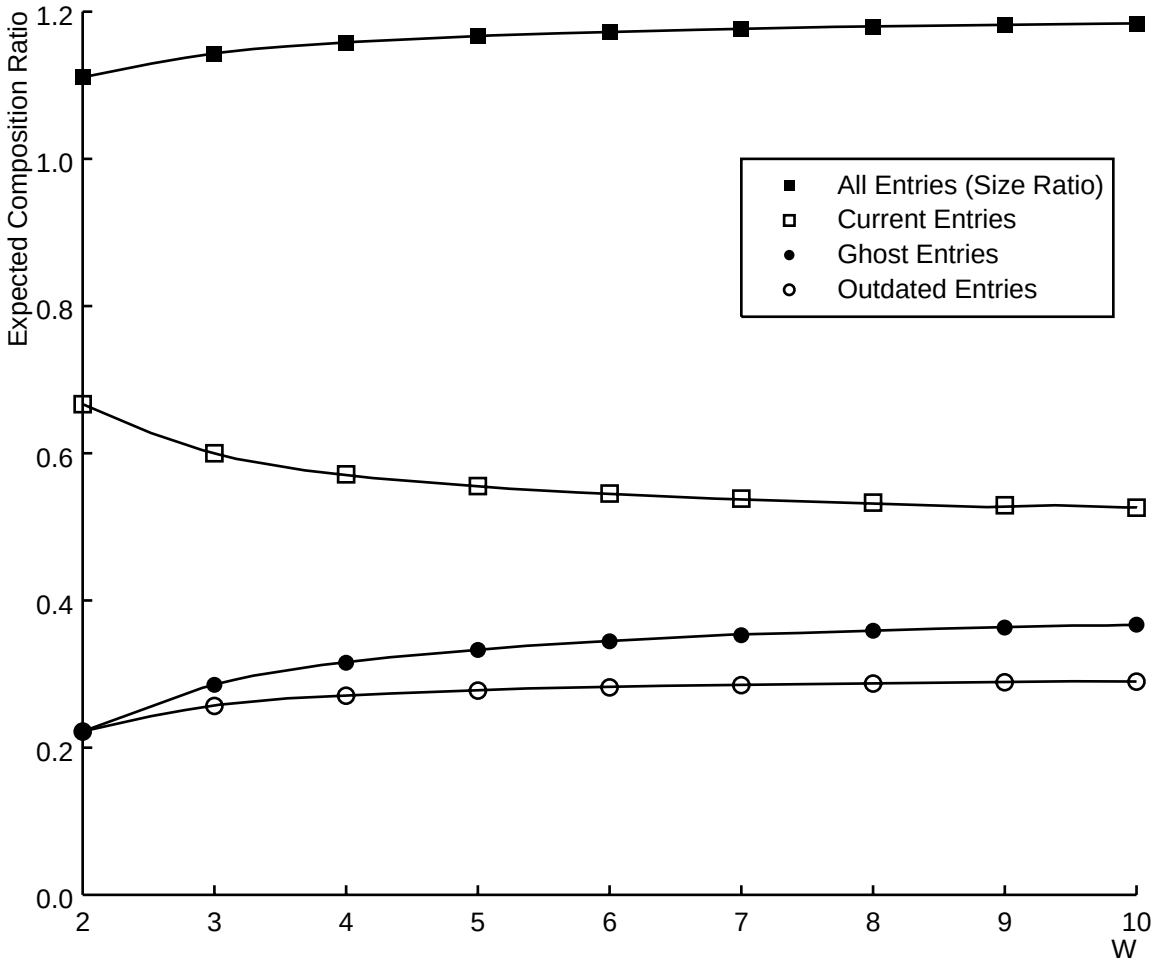

Figure 3-6: Expected Composition Ratios in a (2*i*–1) - *i* - *i* RSM

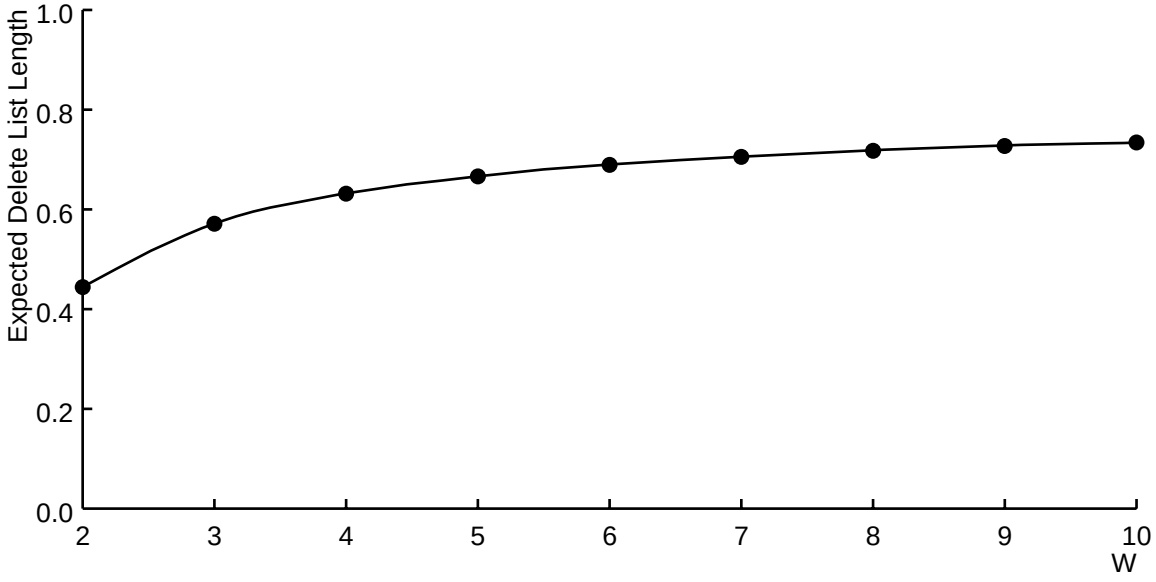

Figure 3-7: Expected Delete List Lengths in a (2*i*–1) - *i* - *i* RSM

Let us maximize it subject to the (real) constraints that $N \geq 1$ and $N/2 \leq W \leq N$. As one would expect, this expression grows as the write quorum size decreases. Thus the expression achieves its maximum when W is set to $N/2$, its lowest permissible value. So:

$$d \leq \frac{4(N - \frac{N}{2})}{N + 3\frac{N}{2}}$$

$$= \frac{4}{5}.$$

In other words, the average length of a delete list will not grow beyond .8, no matter what quorum sizes are used.

A similar result holds for the expected size ratio ($\mathbf{E}[E/A]$). The expression for this quantity is:

$$\frac{2(N+W)}{N+3W}.$$

Standard methods show that this expression, subject to the same constraints as above, also achieves its maximum when $W = N/2$, independent of N. Thus its value is bounded by:

$$\frac{2(N+\frac{N}{2})}{N+3\frac{N}{2}}$$

$$= \frac{6}{5}.$$

These two performance measures completely specify the significant time and space requirements of the system. Therefore, average performance cannot degrade without bound, regardless of what values we choose for the parameters.

3.3.7. Varying the Operation Mix

In the simulations and analysis, we assumed that Insert, Update and Erase operations occur with equal likelihood. In practice, the operation mix will vary from application to application. It is straightforward to extend the analysis to cover other operation mixes. This is accomplished by substituting the frequencies of each operation for the appropriate

terms in the balance equations, instead of assuming that all such terms are 1/3 (Appendix A). We extended the analysis along these lines. For brevity's sake, we will not present the details of the analysis, but briefly summarize the results.

We allow the probability that the operation is Update, which we call P_u, to vary from zero to one. If the Insert probability is unequal to the Erase probability, the number of occupied addresses in the RSM will dwindle to zero or increase without bound; thus we assume they are equal. Under this assumption, P_u completely specifies all the operation frequencies. The extended analysis consists of recasting the balance equations in terms of P_u, solving them and studying the solution. The solution to the generalized balance equations is:

$$c' = \frac{W}{N}$$

$$o' = \frac{2W(N-W)}{N((1-P_u)N+2W)}$$

$$d = \frac{(3-P_u)(N-W)}{(1-P_u)N+2W}.$$

The resulting formula for the expected value of the delete list length is:

$$\frac{(3-P_u)(N-W)}{(1-P_u)N+2W}.$$

The formula for the expected value of the size ratio is:

$$\frac{(3-P_u)(N+W)}{2(1-P_u)N+4W}.$$

As expected, when P_u is set to 1/3 these expressions reduce to those obtained from our original analysis.

For a 3-2-2 RSM, the expected delete list length does not vary significantly over the entire spectrum of P_u values, achieving a minimum of 0.43 at $P_u = 0$ and approaching a maximum of 0.5 as P_u approaches 1 (Figure 3-9). Similarly, the size ratio achieves a minimum of 1.07 at $P_u = 0$ and approaches a maximum of 1.25 as P_u approaches 1 (Figure 3-8).[6]

[6]The performance measures do not have well defined expected values when $P_u = 1$. This will be discussed at greater length in Section 3.3.8.

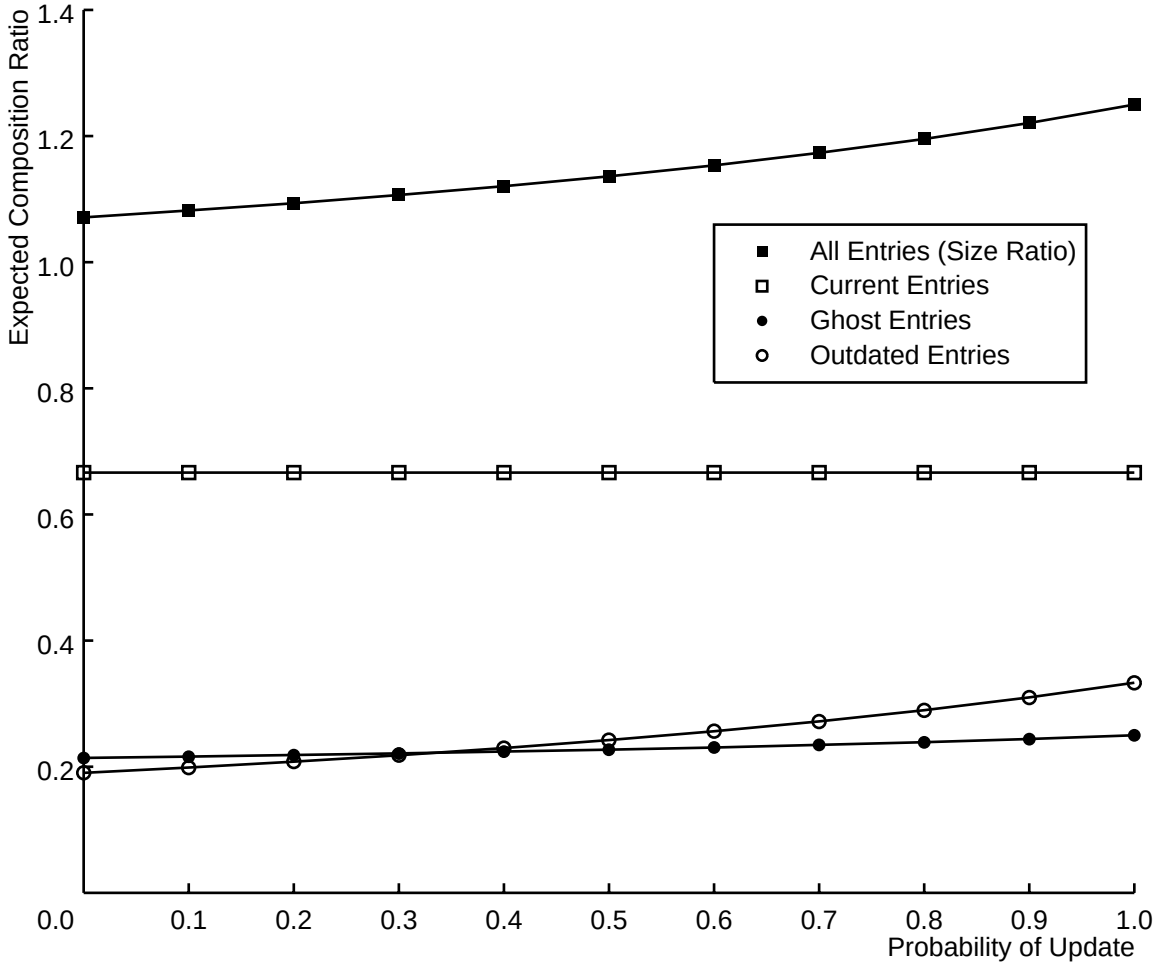

Figure 3-8: Expected Composition Ratios for Varying P_u in a 3-2-2 RSM

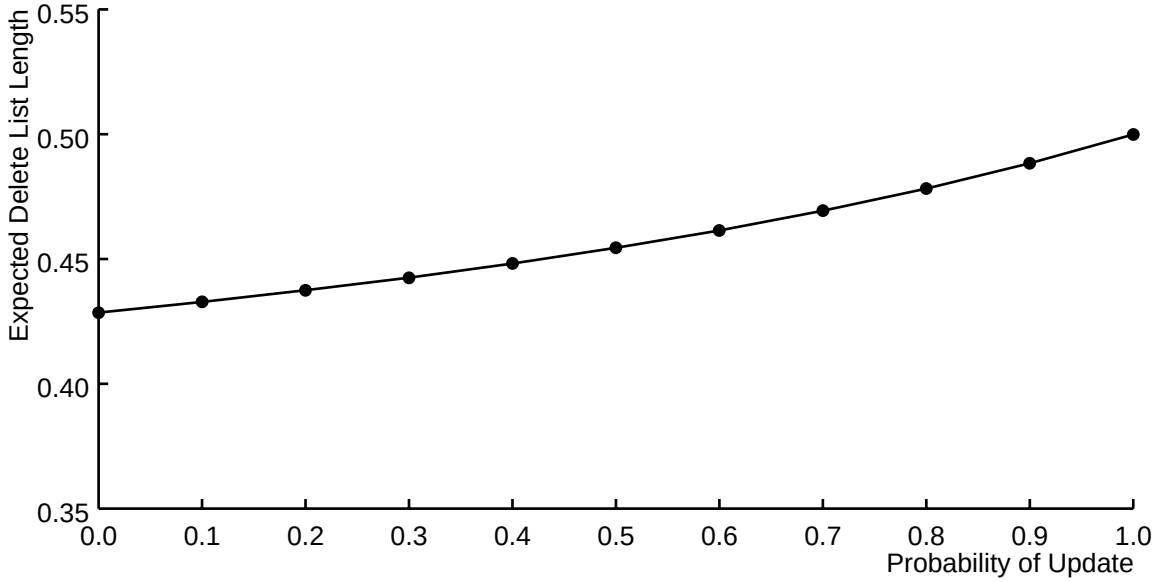

Figure 3-9: Expected Delete List Length for Varying P_u in a 3-2-2 RSM

The favorable worst case results in Section 3.3.6 can be generalized. For all legal values of N, W and P_u, the expected delete list length will always be <1, and the expected size ratio will always be <1.5. Thus the performance of the system remains good for any (random) operation mix.

3.3.8. Discussion of the Analysis

The primary purpose of this section is to discuss the validity of the analysis and the applicability of the results. The Markov model exactly describes the RSM under random usage. Three assumptions are made in the construction of the balance equations, hence the balance equations do not exactly reflect the Markov model. The balance equations are solved exactly, and a final assumption is made in deriving the expression for the composition ratio for ghost entries from the solution to the balance equations. These four assumptions represent the sole source of inaccuracy in our analysis. We enumerate and examine the assumptions below. The page number after each assumption indicates the page on which the assumption is made.

1. In constructing each balance equation, we assumed that the three operations (Insert, Update and Erase) occur with equal probability. (P. 153)
2. In constructing the balance equation for current entries, we assumed that the probability that a representative contains an entry for the real predecessor of a randomly chosen occupied address is equal to the probability that it contains a randomly chosen address in the RSM. (P. 156)
3. In constructing the balance equations for current and ghost entries we ignored the possibility of a ghost entry becoming outdated or current in an Insert operation. (Pp. 155, 158)
4. In deriving the formula for the composition ratio for ghosts entries, we assumed that each ghost in a representative appears in exactly two delete lists. (P. 51)

The first assumption holds in all states of the model except those representing RSMs that are empty or *full*. (An RSM is full if all addresses in the address space are occupied.) In the description of the system being modeled, we said that addresses to be erased are chosen from occupied addresses; Erases do not occur in states representing an empty RSM. By the definition of Insert, an address being inserted must be unoccupied; Inserts cannot occur in the state representing a full RSM. However, these boundary states represent a negligible fraction of all system states and occur with extremely low probability, assuming the address space is reasonably large. If the address space is small, it takes a much shorter run of Inserts to fill the RSM or Erases to empty it; thus these boundary states occur with much greater likelihood. The address space used in the simulations was large enough that these states were never encountered.

The second assumption concerns the probability that a representative contains an entry for the real predecessor of a given address. In any system state, the number of occupied addresses that have an entry in a given representative can differ by at most one from the number of addresses whose real predecessor has an entry in this representative. This is so because each occupied address except the last one is the real predecessor of another occupied address. Thus, the probability that a randomly selected occupied address has an entry in this representative differs by at most $1/A$ from the probability that the real predecessor of a randomly selected address has an entry in the representative. But if the address space is large, A will be large in the system states that occur with high probability and this assumption will be almost correct.

The third assumption is that ghost entries cannot enter the outdated or current class in an Insert operation. This assumption is violated when an address that has been erased from the RSM is reinserted while a ghost for the original incarnation of the address still exists in some representative. Unless the assumption is violated in a substantial fraction of all Insert operations, it will have little effect on the accuracy of our analysis. The larger the cardinality of the address space, the less likely it is that the assumption will be violated in any given Insert operation. While it would not surprise us if the assumption were violated at some point during our simulation runs, the address space used in the simulations was large enough that the overall effect on the accuracy of the analysis was almost certainly negligible.

The fourth assumption is very similar to the second. All ghosts in a representative except those before the first occupied address and after the last occupied address do occur in the delete lists of two occupied addresses. The other ghosts in a representative occur in only one delete list. However, in all reasonably likely states, the ghosts are fairly well distributed among the occupied addresses, thus on average, only a small constant number of ghosts will be on a single delete list. For representatives containing reasonably many entries, these few ghosts will be ‘‘swamped’’ by the ghosts that appear on two delete lists, and $D/2$ will be almost identical to G/A. If the address space is reasonably large, the approximation will be good in all reasonably likely states and the assumption will be valid.

In summary, all the assumptions quickly become reasonable as the address space gets large. This is the only point where the cardinality of the address space enters into our analysis. It was not used explicitly in any of the equations. None of the assumptions break down when N or W get large (assuming the address space is large); thus, the worst case results obtained from the limiting behavior of the predictive formulas are valid. This also implies that the formulas can be used with confidence for any quorum sizes.

The extension of the analysis to operation mixes wherein operations occur with unequal likelihood presents another difficulty. By the nature of the balance equations, their

solutions represent the expected values for certain stochastic variables, *assuming the variables have well defined expected values*. In Section 3.3.2 we showed that our model was finite and irreducible, which allowed us to assert the existence of well defined expected values for the requisite stochastic variables. However, the proof of irreducibility assumed that the Erase and Insert operations occur with nonzero probabilities. If this assumption is false, the proof is no longer valid. In fact, when $P_u = 1$, only a tiny fraction of the model states can be reached from any given state, hence the model is no longer irreducible.

Consequently, the delete list length and the composition ratios no longer have well defined expected values when $P_u = 1$. In any given experiment, the observed average values for these performance measures will depend heavily on the initial contents of the RSM representatives. The average delete list length will remain constant at whatever value it holds in the initial state, as no ghosts will be added to the representatives nor will any new addresses become occupied in the RSM. The size ratio will converge to one plus the (initial) value of G/A, as each representative will eventually contain one entry (current or outdated) for each occupied address, as well as any ghosts it contained initially. In practice, this means that when $P_u = 1$, the observed average values for the performance measures will be even lower than those predicted by the formulas.

Typically, when a model breaks down at some particular parameter value, it begins to break down in the vicinity of this value; our model is no exception. Observed averages for the performance measures will approach their predicted equilibrium values more and more slowly as P_u approaches one. Thus, observed averages for experiments of a given length will be more and more dependent on the initial system state. If the initial state is produced by a sequence of Inserts, the representatives will initially contain no ghosts and the performance measures will have very desirable (i.e. low) values. If the initial state is produced by prolonged operation of the system with some higher P_u value, the initial values of the performance measures are likely to be better than those predicted for $P_u = 1$, as our analysis predicts better performance for lower P_u values. In either case, observed average values for these measures will approach the true equilibrium values only slowly. Thus, early in their operation, update intensive systems will probably display even better performance than the formulas predict. Not only do the worst case results for the performance measures remain valid in the region where the model breaks down; they may be overly pessimistic.

A note should be added concerning the equilibria observed in the simulations. These equilibria definitely did not represent true equilibrium state distributions over our entire model. This is clearly demonstrated by the fact that the simulations did not generate identical average values for the number of addresses in the RSM (A) from run to run. The observed average values for A were clearly related to the initial number of occupied addresses in each run. This is not at all surprising, when one considers that the number of

states in the model is exponential in the cardinality of the address space, and the simulations were run for far fewer steps than the address space cardinality itself. We proved that a simulation of sufficient length would display equilibrium behavior over the entire model, but our runs were not of sufficient length. This leaves unexplained the fact that the runs exhibited predictable equilibrium behavior for all of the performance measures of concern to us.

The explanation for this phenomenon lies in the fact that our simplified model is still highly lumpable. Moderately sized ‘‘clumps’’ of contiguous states with reasonably high probabilities of occurrence, such as those traversed in each run of the simulation, have the same average values for the performance variables as those predicted for the entire model. In fact, our analysis captures these clumps better than it captures the entire state space, as the clumps tend not to contain the boundary states where the assumptions break down.

3.4. Discussion

The system simulated and analyzed was not entirely realistic. Read and write quorums would not be chosen randomly in practice. A node would more naturally communicate with easily accessible nodes. Because of the cost of establishing a communication session, the node would probably continue to communicate with the same nodes until it had no need for further communication or a failure occurred. Therefore, in practice, the read and write quorums used by any given node would probably change infrequently. The random distribution of operations and addresses is also unrealistic. However, we conjecture that the performance observed under real conditions will be as good as or better than that of the system studied.

One possible usage pattern for the system is that a single read/write quorum that changes infrequently is used for all operations. This is a special case of the scenario described in the previous paragraph. We performed additional simulations to investigate the behavior of the system under this usage pattern. These simulations were identical to the ones previously described except that before each operation, a decision to change the quorum was made with probability p. When it was determined that the quorum was to change, a single, randomly chosen member of the quorum was replaced with a representative chosen at random from those not already in the quorum. Thus, on any given iteration at most one member of the write quorum changed. This usage pattern could occur if an RSM were being used by a single client.

Simulations were performed on 3-2-2 directories initially containing 100 occupied addresses, with p values of 0.1, 0.01, 0.001, and 0.0001. Two hundred thousand operations were performed in each simulation and data was collected during the final one

hundred thousand operations. The results showed that as the value of *p* decreases, the average delete list length decreases significantly from the value observed under random usage. The size ratios did not change significantly from the size ratios observed under random usage. These results indicate that the total number of outdated and ghost entries remains close to the total under random usage, but they are now concentrated outside of the write quorum. Thus, the delete lists actually encountered tended to be shorter than those observed under random usage. The results of this simulation are consistent with our conjecture that the performance of the system will be at least as good under realistic usage patterns as it was under the random usage studied in the simulations and analysis.

As noted in Chapter 2, the RSM data structure can be used with infinite address spaces. In fact, a natural choice for the address space is the set of finite length alphanumeric strings, which is countably infinite. The system studied in the simulations and analysis is not well defined for countably infinite address spaces, so it is natural to ask how well the results of the analysis apply to these address spaces. In practice, the effect of using infinite address spaces is identical to that of using large but finite address spaces. Namely, it keeps the system away from boundary states where the assumptions made in the analysis break down. Thus, the analysis captures actual usage patterns over infinite address spaces as well as it captures any other actual usage patterns.

One disadvantage of our analysis technique is that it can only be used to determine expected values for the performance measures. Thus we can only characterize the average case performance of our data structure. It would be nice to have additional information on the probability distributions of the performance measures. The simulations give us some information along these lines, and we can gain some insight by reasoning directly about the worst case performance of our algorithm.

The simulations and our intuition indicate that under realistic access patterns the size ratio will not vary much from its average value. But it is worth noting that one can construct a pathological sequence of operations wherein ghosts are allowed to accumulate in one representative while the RSM remains almost empty, causing the size ratio to grow without bound. This can be accomplished by selecting one write quorum for all Insert operations and a second write quorum for all Erase operations that intersects the first in only one representative. However there is no reason this should occur in practice.

The pathological sequence of operations described in the previous paragraph is not as bad it might seem. It causes delete lists to grow without bound in the representatives only outside of the write quorum for Erases. As long as the pattern continues, the long delete lists will not be encountered (assuming Erases use the same read and write quorum). If the representatives outside the write quorum are eventually used for Erases again, the first few Erases at these representatives will encounter long delete lists. These first few Erases will run slowly at these representatives, but in the process, they will

purge the representatives of excess ghosts, so future Erase operations will run quickly. Each ghost can only slow down a single Erase operation; once it is encountered, it will be purged.

It should be noted that even in pathological cases like the one described in the previous paragraphs, a maximum of three rounds of messages are sufficient to perform the Erase operation; the extra work is all local to the representatives. If pathological behavior of the sort described were to occur in practice, it would probably not cause noticeable delays; even a very long delete list (say 100 entries) can be scanned and purged quickly if an efficient data structure is used to store the data at representatives. If it is particularly important for some application that all Erase operations run fast, care can be taken to ensure that all representatives are used frequently in write quorums for Erase operations, and so kept clean of excess ghosts.

Chapter 4

Optimizations and Extensions to Replicated Sparse Memories

In this chapter, we present several modifications to the replicated sparse memory data structure described in Chapter 2. These modifications fall into two broad classes. Section 4.1 contains *optimizations*, whose major purpose is to improve some aspect of the performance of the RSM. Section 4.2 contains *extensions*, whose major purpose is to extend the semantics of the RSM.

Section 4.1.1 presents a technique called *optimistic timestamps* for reducing the communication cost associated with Write operations. Section 4.1.2 shows how the technique can be generalized to a broad class of distributed algorithms. The resulting optimized algorithms are called *optimistic two-stage protocols*. Section 4.1.3 describes a technique for decreasing the latency associated with the Erase operation. Section 4.1.4 describes a technique for reducing the cost of modifying individual fields of records stored in RSMs. Section 4.1.5 presents a modification to the RSM data structure that permits the representatives to be stored as hash tables. The resulting data structure is faster than a normal RSM and easier to implement, but it does not support certain extensions. Section 4.1.6 presents a simple RSM implementation based on arrays. This implementation is suitable for small address spaces or RSMs in which a substantial fraction of the address space is occupied.

Section 4.2.1 presents *range operations*, analogues of the basic RSM operations that operate on a whole range of addresses. The range operations are very efficient: each range operation has the same communication cost as its single-address counterpart. Section 4.2.2 presents *navigation operations*, which permit the user of an RSM to scan though its occupied addresses.

4.1. Optimizations

4.1.1. Optimistic Timestamps

The Write operation on RSMs, as described in Section 2.4.2, requires two rounds of message exchanges. The sole purpose of the first round is to determine the highest version number currently associated with the address being written. One common pattern of data access is to read an item and then write back a new value that depends on the old value in some way. Thus, Write operations in an RSM are often preceded by Reads to the same address. Since the Read operation determines as a byproduct the highest version number associated with the address, the first round of a subsequent Write operation from within the same transaction can be eliminated.

Not all Write operations on an RSM are preceded by Reads, however. Those that are not, called *blind writes,* require two rounds of messages exchanges. The first round could be eliminated if it were possible to choose an appropriate version number without consulting a read quorum. Since the new version number must be greater than any version number previously associated with the address, it is not, in general, possible to do this. But it is possible to avoid the first round much of the time, yielding a probabilistically fast algorithm for blind writes.

The first round of the Write operation allows a client to choose a version number that it *knows* to be higher than any currently associated with the address. Instead of performing this round, the client guesses a version number that it hopes will be high enough. Then it attempts to do the Write with this version number. Each representative in the write quorum compares the guessed version number to the version number currently associated with the address at the representative. If the new version number is greater than the old, the representative accepts the write request and sends an ‘‘OK’’ response to the client. Otherwise, it sends a ‘‘Cannot Comply’’ response containing the version number currently associated with the address. If the client receives ‘‘OK’’ responses from the entire write quorum, the operation has completed in one round; if the client receives one or more ‘‘Cannot Comply’’ responses, it generates a version number by incrementing the highest one returned in a ‘‘Cannot Comply’’ response and sends another write request to the entire write quorum with the new version number.

If one or more ‘‘Cannot Comply’’ responses are sent, indicating that a follow up round is necessary, the RSM will be in an inconsistent state between the initial round and the follow-up. However, the write locks held at the representatives will prevent any other transaction from observing the inconsistency. The locks will not be dropped until the transaction commits, by which time the inconsistency will have been corrected by the follow-up round. If the follow-up round does not complete successfully, the transaction

will automatically abort and the object will be restored to consistency by the underlying transaction system's recovery protocol.

The procedure outlined above is guaranteed to write the RSM location in one or two rounds; at worst, it has the same communications cost as the naive method. It is not clear how often it is faster: it depends on the manner in which the new version numbers are guessed. If all of the nodes had exactly synchronized clocks, message delays were all identical, and the precision of the clocks were sufficiently high, then guessing the current clock time would *always* cause the operation to succeed in one round. While approximately synchronized real-time clocks do not satisfy these assumptions, they will usually cause the operation to succeed in a single round, unless the clocks are very poorly synchronized and the same location is frequently written by different clients. We call version numbers selected in this manner *optimistic timestamps*[7].

Optimistic timestamps can be used for any blind write operations in version number based weighted voting algorithms. For instance, they can be applied to the write operation in Gifford's original weighted voting algorithm [22].

Optimistic timestamps should only be used for blind writes. If a Read precedes a Write in the same transaction, a version number just greater than the one read should be used for the Write. This leaves a larger margin of error for the next optimistic timestamp used for the address. This is particularly important if the address is updated frequently by different clients.

Clients should keep track of the last optimistic timestamp they have used. This will typically be the clock value when the last timestamp was generated. If, however, the previous operation required two rounds because an optimistic timestamp was too low, it will be the number that was generated by incrementing the highest version number returned in a ''Cannot Comply'' response. If a client needs an optimistic timestamp and the current real-time clock value is less than or equal to the previous optimistic timestamp used by the client, the new timestamp should be generated by incrementing the previous one, rather than using the clock value. This policy ensures that the optimistic timestamps generated by a client form a strictly increasing sequence. This property ensures that in a sequence of Writes to a location by a single client, uninterrupted by Writes from another client, only the first Write can require two rounds; the remainder are guaranteed to succeed in a single round.

---

[7]The name *optimistic timestamp* is slightly misleading; the algorithm does *not* use optimistic concurrency control. Whether or not optimistic timestamps are used, our replication protocols use two-phase locking, which is a form of pessimistic concurrency control.

As a further optimization, when an optimistic timestamp fails at a representative (i.e., the guessed version number is too low), the representative goes ahead with the Write operation, using a version number one greater than the previous version number associated with the address at the representative. The representative still sends back a ‘‘Cannot Comply’’ response containing the old version number. If all of the representatives in the write quorum send back ‘‘Cannot Comply’’ responses containing the same version number, the operation has completed in one round. In this case, the client should record the version number used by the representatives, to preserve the property described in the previous paragraph. Otherwise, a second round is necessary. The second round need include only those representatives that did not send back ‘‘Cannot Comply’’ responses containing the highest version number returned in the first round.

For example, suppose a client attempts a Write to a quorum of two representatives. The client reads a time of 42 from its real-time clock, so it uses 42 as the version number for the Write. If both representatives in the write quorum have a version number lower than 42 associated with the address being written, the Write will succeed in one round. Suppose both representatives have version number 45 associated with the address. Both will insert (or update) the entry, giving the entry version number 46, and return ‘‘Cannot Comply’’ responses indicating version number 45. Since both responses indicate version failure with the same version number, the client knows the operation has completed successfully. Suppose the first representative has version number 20 associated with the address and the second representative has version number 45. The first representative will perform the operation and return ‘‘OK.’’ The second will insert (or update) the entry with version number 46 and return ‘‘Cannot Comply’’ with version 45. The client must perform a second round but this round only has to include the first representative. In this round, the client uses version number 46, which is the same version number already used at the second representative.

Optimistic timestamps can be used to speed up other operations on RSMs besides simple blind writes. We can provide an additional RSM operation called ReadWrite, that reads the current value out of a location and (blindly) writes a given value to the location. This operation is equivalent in function to a Read followed by a Write, but if it is implemented with optimistic timestamps, it is much more efficient, requiring only a single round of messages when the optimistic timestamp succeeds. Optimistic timestamps can also be used to implement fast queues, stacks and priority queues on top of RSMs, as described in Section 5.3.3.

Optimistic timestamps are most effective if it is cheap to read an approximately synchronized real-time clock. In the traditional Unix implementation, this requires a system call (*gettimeofday*). More modern systems have provisions for mapping a real-time clock into a user process’s address space. But even a system call is typically an order of magnitude faster than an RPC. This difference is more than sufficient to warrant the use of optimistic timestamps.

Earlier we noted that optimistic timestamps might not be effective if the same location were frequently written by different clients. While such *hot spots* represent the least favorable conditions for optimistic timestamps, the use of optimistic timestamps can still result in considerable cost savings.

Consider the case of two clients on separate nodes repeatedly performing transactions that Write to the same address in an RSM. If an optimistic timestamp fails, causing a Write to require two rounds, the next Write operation is almost guaranteed to succeed in a single round. If the next Write comes from the same client, the attempt will succeed in one round because clients generate optimistic timestamps in strictly increasing sequence. If the next attempt comes from the other client, it will almost certainly succeed in one round because the version number currently associated with the location is one more than the last timestamp generated by this client. If clients always generate timestamps that are at least *two* more than the last one they generated, the Write following a two round Write is *guaranteed* to succeed in a single round. Thus, optimistic timestamps can be guaranteed to succeed at least half the time if only two clients are writing to a hot spot.

In fact, optimistic timestamps can succeed far more than half the time when two clients are writing to a hot spot. Earlier in this section, we described an optimization wherein representatives go ahead with a Write operation even if the version number is too low. Let us assume this optimization is implemented. A Write operation will succeed in one round even if its optimistic timestamp is too low if it uses the same write quorum as the previous Write to the location. Both representatives in the write quorum will return ‘‘Cannot Comply’’ responses with the same version number. This, in combination with the optimization, causes the operation to succeed in a single round. If both clients systematically use the same write quorum, *all* of the Writes will succeed in a single round. In a 3-2-2 RSM, even if write quorums change from time to time, these ‘‘two wrongs make a right’’ successes will occur fairly often, as there are only three distinct write quorums.

Note that version numbers increase much faster when optimistic timestamps are used. If optimistic timestamps are not used, the version number associated with an address is roughly equal to the number of operations that have been performed on the address. If optimistic timestamps are used, version numbers are taken from the real-time clock, which is constantly increasing. A sufficient number of bits must be used to represent version numbers to prevent them from wrapping around. If the real-time clock has microsecond precision and sixty-four bits are used, version numbers will not wrap for approximately 600,000 years. We conjecture that this is sufficient for most applications.

4.1.2. Optimistic Two-Stage Protocols

We conjecture that there are other useful distributed algorithms with the same computational structure as the naive algorithm for writing a location in an RSM. These algorithms require two rounds of message exchanges: one round to gather information from remote data objects, and one round to modify the remote data objects on the basis of the information. We call such algorithms *two-stage protocols*. The optimistic timestamp technique can be generalized to apply to any algorithm in this class.

The first round of a two-stage protocol can be eliminated if the client can predict the information that will be returned in this round. In essence, the two rounds are combined into a single round that performs the update based on some assumption *and* returns the information that enables the client to check the assumption. If the assumption was correct, the operation has succeeded in a single round. If it was faulty, a second round is required to undo the incorrect updates made on the basis of the faulty assumption and perform the correct updates instead. The resulting algorithm still has a worst-case communication cost of two rounds. But if it is possible to predict the information in a significant fraction of cases, the *average-case* cost will be much lower. If it is usually possible to predict the information, the average-case cost will be close to one round. We call algorithms of this type *optimistic two-stage protocols*. We call the assumption made prior to the first round the *optimistic assumption*.

Optimistic two-stage protocols are only applicable in the context of an underlying transaction system. The transactions system's recovery protocols permit servers to write ''optimistic'' information in their data structures in the first stage, even though it will temporarily corrupt the distributed data structure if the optimistic assumption is incorrect. Write locks held at the servers will prevent other transactions from observing the distributed data structure in its corrupt state. These locks will not be dropped until the transaction that performed the optimistic two-stage protocol commits. The transaction will not commit until after the second stage of the protocol completes, rectifying the inconsistency in the data structure.

If some failure occurs that prevents the protocol from completing, the enclosing transaction will abort, and the recovery protocol will restore the data structure to its original state before the optimistic two-stage protocol was initiated. There is no chance of accidentally committing an incorrect write that was performed in the first stage as a result of a faulty assumption. If an optimistic assumption was correct, the client does *not* have to send explicit messages to the servers that wrote data based on the assumption. In essence, the client uses the messages sent in the transaction system's commit protocol to inform the remote servers that the optimistic assumption panned out.

While an optimistic two-stage protocol will never have a higher communication cost than the two-stage protocol from which it was derived, it can be more time-consuming. In the original two-stage protocol, the first stage involves only reading the data object; in the optimistic version, it involves writing the object as well. Also, the optimistic version requires generating the optimistic assumption, which may have some cost associated with it. (In the case of optimistic timestamps it is the cost of reading the real-time clock.) If the assumption is faulty and the second stage is required, the total time cost of the two stages will be higher than that of the original two-stage protocol by the cost of the writes in the first stage plus the cost to generate the assumption.

The observation in the previous paragraph can be formulated mathematically to produce a quantitative criterion for determining whether a given two-stage protocol will benefit by being made optimistic. The derivation also yields a formula for the amount of time that will be saved by the use of the optimistic protocol. First we define some variables:

- p_s The probability that the optimistic assumption is correct.
- c_w The cost of doing the writes in the first stage of the optimistic protocol (i.e., the portion of the difference in cost between the two-stage protocol and the assumption failure case of the optimistic protocol that is due to the writes in the first stage).
- c_a The time cost of making the optimistic assumption. (This will often be negligible.)
- c_2 The cost of doing the second stage in the optimistic protocol (i.e., the difference in cost between the assumption failure and assumption success case of the optimistic protocol).

The preceding variables are the parameters that describe a particular two-stage protocol and its optimistic variant. The following variables are introduced to facilitate the derivation.

- c The cost of the basic two-stage protocol.
- c_{of} The cost of the optimistic protocol when the optimistic assumption is incorrect. Note that $c_{of} = c + c_w + c_a$.
- c_{os} The cost of the optimistic protocol when the optimistic assumption is correct. Note that $c_{os} = c_{of} - c_2$.
- c_o The expected cost of the optimistic protocol. Note that $c_o = p_s c_{os} + (1 - p_s) c_{of}$.

We desire to know the expected time savings if the optimistic protocol is used. This is simply $c - c_o$:

$$= c - p_s c_{os} - (1-p_s) c_{of}$$

$$= c - p_s (c_{of} - c_2) - (1-p_s)(c + c_w + c_a)$$

$$= c - p_s (c + c_w + c_a - c_2) - (1-p_s)(c + c_w + c_a)$$

$$= p_s c_2 - (c_w + c_a) .$$

Intuitively, the preceding equation says that the expected cost savings are equal to the expected savings from eliminating the second stage minus the constant overhead incurred by the optimistic protocol. The optimistic version of the protocol will result in a net savings if the expected savings are greater than zero:

$$p_s c_2 - (c_w + c_a) > 0 .$$

Simplifying:

$$p_s < \frac{c_w + c_a}{c_2} .$$

Let us examine what the preceding formulas mean in practical terms. Consider the case of optimistic timestamps for writing to a simple 3-2-2 replicated file implemented on the Camelot system, running on IBM-RT/PC APC workstations connected by a 4 Megabit per second token ring. The cost of doing the writes in the first stage, c_w, is essentially the incremental cost of doing a write to recoverable memory. Although writes take place at several representatives, we only count the cost of one write, as the writes execute in parallel and we are measuring latency at the client. Thus c_w is approximately 1.0 ms. The cost of making the optimistic assumption, $_a$, is essentially the cost of reading the *gettimeofday* clock. In principle, Mach permits the clock to be mapped into the client's address space, but this facility does not work properly on RTs. To be conservative, we assume the cost of a *gettimeofday* system call, approximately 0.2 ms. The cost of doing the second stage of the optimistic protocol is the incremental cost of doing two parallel remote RPCs, each of which performs a single write to recoverable memory. This was measured to be approximately 31 ms. Thus, the minimum p_s value necessary for the optimistic protocol to be worthwhile is approximately 1.2/31 or 4 percent.

Under any usage pattern, optimistic timestamps will succeed far more than 4 percent of the time. Thus, their use is highly recommended in the example of above. While the four percent figure is specific to this example, we believe that five to ten percent is a reasonably good estimate of the minimum p_s value necessary for any optimistic two-stage protocol to be worthwhile. This is so because c_w will always be the cost of one or several writes to recoverable storage, c_a will never be more expensive than a system call (often it will be negligible), and c_2 will always be the cost of a replicated remote RPC. In any

reasonable system, c_2 will be roughly an order of magnitude higher than $c_w + c_a$, regardless of the details of the specific application or the underlying transaction system.

Let us consider the class of two-stage protocols where the information gathered in the first stage consists of the answer to a yes-or-no question (e.g., ‘‘Do any of the remote data structures have an entry for a given key?’’). The five to ten percent guideline developed in the previous paragraph has an interesting consequence with regard to these protocols. Even if no information is available that helps the client select the correct answer to the question, random guessing will produce a correct answer fifty percent of the time. This is far greater than the minimum five to ten percent required for a performance gain; an optimistic version of the protocol wherein the optimistic assumption is made by guessing at random will yield a great performance improvement.

Optimistic two-stage protocols have repeatedly proven useful in the work described in this dissertation. They yield fast algorithms for doing blind writes to RSMs (Sections 4.1.1, 4.1.4), Inserts and Modifies to directories (Section 5.3.1), Enqueues to queue-like data objects (Section 5.3.3), Erases to RSMs where the replicas are implemented as hash tables (Section 4.1.5) and Increments to bounded counters (Section 5.4.1). We conjecture that optimistic two-stage protocols represent a generally useful paradigm for designing fast distributed algorithms in the context of transaction systems.

4.1.3. Low Latency Erases

The Read operation for RSMs requires a single round of message exchanges. If optimistic timestamps are used, the Write operation usually requires one round, occasionally two. The Erase operation, however, usually requires two rounds and can require as many as three. This expense is inherent in the RSM data structure. Arguably, it is reasonable, as the Erase operation performs the ‘‘incremental garbage collection’’ that is responsible for giving the RSM data structure its favorable performance properties. But there is no reason the expense has to be incurred on the latency path of the Erase operation.

In this section, we present a simple technique that allows part of the cost of the Erase operation to be removed from its latency path. The basic idea is to replace the Erase with a Write to the same address, with a special value indicating that the address is unoccupied. This produces a fast *logical erase* operation. The actual Erase operation is performed at a later time, off the latency path of the logical erase. The latency of the Erase operation will be reduced to that of a Write.

An easy way to implement the logical erase is to add an *occupied* flag to each entry, initialized to TRUE when the entry is created. The logical erase inserts an entry with this flag set to FALSE at each member of the write quorum. This obviates the need for

reserving a special **unoccupied** value in the value domain. If the Read operation observes that the highest version number associated with an address belongs to an entry with the occupied flag set to FALSE, it reports that the address is unoccupied, just as it would if the highest version number belonged to a gap.

It is critical that the actual Erase operation eventually get performed. Otherwise the logical erase will leave behind a *tombstone*, and eventually the RSM will become cluttered with tombstones, ruining its space performance. To insure that the actual Erase get performed, one of the representatives in the write quorum for the logical erase must enter the address into a list of Erases to be performed. The list must be stored in transaction-consistent recoverable storage to ensure that the actual Erase will be performed if and only if the transaction containing the corresponding logical erase commits.

When a representative enters an address onto its list, it takes responsibility for eventually performing the Erase. When the representative decides to do the Erase, it acts as the client for the operation. It is free to select any write quorum, though it will almost certainly include itself in the quorum for efficiency.

Some care must be taken when performing the Erase: it should only be performed if the tombstone left behind by the corresponding logical erase is still current. If the address has been written between the time of the logical erase and the corresponding actual Erase, the actual Erase must be suppressed. This effect can be achieved by checking at the same time as the address's real neighbors are being determined that the highest version number associated with the address has not changed since the logical erase. If the version number has changed, the RepCoalesce phase of the Erase operation is suppressed.

A disadvantage of the technique described in the previous paragraph is that it requires that the version number associated with the tombstone be stored with each address in the list of Erases to be performed. However, we observe that it is permissible to go ahead with the RepCoalesce phase of an actual Erase as long as the address to be erased is unoccupied at the time of the actual Erase, even if the version number has increased. As a consequence, the check to confirm that the version number has not increased may be replaced with a check to confirm that the address is currently unoccupied. If this technique is used, it is no longer necessary to store the tombstone's version number in the list of Erases to be performed.

From a performance standpoint, doing low latency Erases is equivalent to preceding each Erase with a Write. These Writes are all Updates, in the terminology of Chapter 3. Thus low latency Erases increase the Update frequency of an RSM. In Section 3.3.7, we showed that this has minimal effect on the space and time performance of the RSM.

This does *not* indicate that low latency Erases should always be used. While the latency of the Erase operation is reduced to that of the Write, the overall cost of the operation is increased by the cost of a Write, plus any overhead incurred in scheduling the actual Erase and creating a transaction to perform it. This substantially reduces the maximum possible throughput. If the system goes through periods where it is not heavily loaded, all of the delayed Erases operations can be performed during those periods, and effective throughput will be increased. But if the system is heavily loaded at all times, low latency Erases should probably not be used.

It is interesting to note that an RSM can support low latency Erases and normal Erases simultaneously. This permits clients to use low latency Erases only in cases where the decreased latency is judged to be worth the added cost.

4.1.4. Frequently Modified Fields

In many typical applications for record files, clients will modify certain fields of the records much more frequently than others. For example, suppose each record represents a charge account; the field representing the outstanding balance will be modified much more frequently than the fields representing the customer's name, address and phone number. If an RSM were used to represent such a record file in the obvious fashion, writing a single field of a record would entail reading the entire record, modifying the chosen field, and writing the record back to the RSM.

The portion of the record that was not being modified would needlessly be sent from each representative in the quorum to the client, and back to the representatives. This portion of the record would also be needlessly written into recoverable storage at each representative in the write quorum, resulting in unnecessary logging activity by the underlying transaction system. Records in commercial databases are often 1-2 Kbytes in length, whereas some frequently modified fields are only several bytes in length; the unnecessary work may be substantial. Network bandwidth and log bandwidth are both scarce resources, so it would be desirable to avoid this work.

Another problem with modifying individual fields by the method described above concerns blind write operations. A blind write to an entire record can usually be performed in one round using optimistic timestamps. This optimization cannot be applied to a blind write to an individual field, as the client must read the contents of the other fields in the record before performing the Write operation.

These problems can be solved by associating a separate version number with each field that will be modified frequently, in addition to the version number associated with the remainder of the record (the *normal part*). This allows the client to write any of the frequently modified fields individually, without affecting any other part of the record.

When the entire record is written, the same version number is associated with the normal part of the record and all frequently accessed fields. This version number must be greater than any version number previously associated with any part of the record, or any gap covering the address corresponding to the key. When an individual field is written, only the data and version number associated with this field are modified. When a record is read, the client selects the data corresponding to the highest version number that was returned for each part of the record and merges all of the selected parts to yield the current value of the record.

If an individual field is written to a representative that has no entry for the given key, the representative creates a new entry and assigns version number zero to all fields other than the one being written. In effect, this creates an additional outdated entry for the key, which causes no harm. There is one problem with this technique: if the file contains no record for the given key, it can destroy the integrity of the RSM data structure. To prevent this possibility, the representative returns the version number currently associated with the key, and a flag indicating whether or not there was an entry for it. From this information, the client can determine whether the address corresponding to the key was occupied. If it was not, the client must undo the operation or abort the enclosing transaction. Note that this is an optimistic two-stage protocol.

In summary, associating separate version numbers with frequently modified fields reduces message sizes and logging activity, and allows fast blind writes to individual fields. These advantages are not without cost. The Read operation is slightly slower, as it must select the data with the highest version number for each field and merge the fields into a record. Additional storage is required at the representatives for the extra version numbers.

4.1.5. Hash Table RSMs

The description of RSMs in Chapter 2 did not specify what data structure should be used to represent the RSM representatives. The RSM representative operations require the ability to lookup, insert, modify, and delete entries for specific addresses. They also require the ability to do linear scans of the entries preceding and following a given address. (This is required, for example, to perform either stage of the real predecessor operation.) Data structures that naturally lend themselves to this set of operations include various types of balanced trees as well as *skip lists* [49].

Balanced trees and skip lists perform lookups, inserts, modifies and deletes in time proportional to the log of the number of entries in the representative. While *hash tables* allow constant time lookups, inserts, modifies and deletes, they do not permit scans of the entries surrounding a given address. Thus, it is not possible to implement RSMs directly

on top of hash tables. In this section, we describe a variant of the RSM in which the representatives are represented as hash tables.

The hash tables for all representatives in an RSM must have the same number of buckets and use the same hash function. Each entry in a representative is stored in the hash bucket corresponding to the entry's address. The entries in each bucket are maintained in a linked list, ordered by address. The gap between a pair of adjacent entries covers only a subset of the range spanned by the entries: those addresses in the range whose hash values would place them in the same bucket. The real predecessor and real successor of an address are redefined as the preceding and following occupied address *in the same bucket* as the address. This allows the Erase operation to operate entirely within a single hash bucket, like all other RSM operations. To ensure that every address has a real predecessor and real successor that lie in the bucket, each bucket must have its own **Low** and **High** entries.

In essence, the hash function partitions the address space into as many subspaces as there are buckets, and a separate RSM is maintained for each subspace. The representatives for these *sub-RSMs* are the hash buckets. Thus, representatives are really represented as linked lists, not hash tables. We need not describe the operations on hash table RSMs in detail, as they are precisely the ordinary RSM operations applied to the appropriate sub-RSM.

Each sub-RSM will rarely contain more than a few occupied addresses. This has interesting implications concerning performance, concurrency control and possible performance optimizations.

Normally, linked lists would be a poor choice of representation for RSM representatives, as access time is linear in the number of entries. However, this is effectively reduced to constant time by the fact that each sub-RSM contains very few occupied addresses. The performance analysis in Chapter 3 does not apply to hash table RSMs, as it applies only to RSMs in equilibrium, and the sub-RSMs in a hash table RSM are far too small to achieve equilibrium.

In a normal RSM, it is essential to do locking at the granularity of individual addresses, in order to achieve reasonable concurrency. In hash table RSMs, it is perfectly acceptable to lock entire sub-RSM representatives (i.e., hash buckets). If the hash function and hash table size are chosen carefully, it will be very rare that two operations are performed simultaneously on distinct addresses with the same hash value.

Bucket locking has two big advantages over the locking schemes required in balanced trees or skip lists. First, it is cheap: each representative operation in a hash table RSM requires only a single lock. Operations on concurrent balanced trees or skip lists require

several locks, even if they are implemented as efficiently as possible. Second, it is trivial to implement bucket locking on hash tables, whereas efficient concurrent B-Trees are quite complex [37, 12]. It is still an open research problem to develop an efficient concurrent skip list implementation.

The small number of occupied addresses in a sub-RSM suggests an interesting optimization to the implementation of the Erase operation. In the terminology of Section 4.1.2, the Erase operation can be considered a two-stage protocol in which the first stage determines the real predecessor and successor of the address being erased and the second stage coalesces the range between the real predecessor and successor.[8] If the hash function and table size are chosen carefully, it should frequently be the case that the sub-RSM containing the address being erased has no occupied addresses other than the address being erased. Thus, we generate an optimistic two-stage protocol based on the assumption that the real predecessor and successor of the address being erased are **High** and **Low**.

To be effective, this optimization must be combined with the use of optimistic timestamps; otherwise it would be impossible to perform the coalesce in one step even knowing the real predecessor and real successor. In effect, the protocol is ‘‘doubly optimistic’’: the optimistic assumption is that the sub-RSM contains no other occupied addresses *and* the optimistic timestamp for the Erase is greater than the highest version number previously associated with the address being erased.

When the optimistic assumption is true, the Erase operation requires only a single round of message exchanges. When it is false, an additional round is required, or two additional rounds in the extremely unlikely event that it takes two rounds to find the real predecessor and real successor. This would seem to indicate that low latency Erases as described in Section 4.1.3 should be unnecessary in a hash table RSM. However, it should be noted that the optimistic assumption made in this optimization is stronger than the optimistic timestamp assumption that enables low latency erases to succeed in a single round. Therefore, low latency erases may still be advisable in a hash table RSM if minimizing latency is of paramount importance.

Note that hash table RSMs cannot support the range operations or navigation operations described in Sections 4.2.1 and 4.2.2. While these operations could be applied directly to the sub-RSMs, this would produce incorrect results, as the hash table RSM effectively re-orders the address space.

---

[8] In fact, this operation does not precisely fit the framework outlined in Section 4.1.2, as the real predecessor operation can itself require two rounds. However, the framework is easily extended to accommodate such operations.

4.1.6. Array RSMs

Replicated sparse memories provide a very efficient replicated implementation for memories with a large, sparsely occupied address space; they are less well suited to small address spaces. The real predecessor determination phase of the Erase operation locks several addresses surrounding the address being erased. If the address space only contains a few addresses, this severely restricts the concurrency on the data object. The Erase operation requires two or three rounds of message exchanges, and representatives must store version number information for gaps between entries. These expenses are acceptable because they are responsible for the data structure's self-cleaning behavior. If the address space is small, however, the number of ghosts that can accumulate at a representative is already limited by the size of the address space. These arguments also apply to larger but finite address spaces that are densely occupied: if there are few unoccupied addresses, there can be few ghosts.

For small address spaces and densely occupied finite address spaces, there is a more suitable implementation for RSMs, wherein the representatives store their data in arrays. Each representative maintains a single array with one location for every address in the address space. Each location contains an entry consisting of a value and a version number. It is unnecessary to store the address, as each location is permanently associated with a single address. The entries are all initialized with a special **unoccupied** value, like the one described in Section 4.1.3. All entries initially have version number zero. Reads and Writes are implemented just as for ordinary RSMs. Erases are merely Writes with the special **unoccupied** value.

While this data structure supports the same operations as ordinary RSMs, it is much less powerful, in that it cannot support infinite address spaces, and it cannot be made to support the extensions described in Section 4.2. Furthermore, it can be grossly space-inefficient for large, sparsely occupied address spaces. But it has many advantages when used with appropriate address spaces. It is extremely fast. The representative operations (lookup, insert, modify and delete) all execute in constant time. No address comparisons are necessary. The Erase operation is exactly as fast as a blind Write operation. If optimistic timestamps are used, a single round of message exchanges is usually sufficient to perform any operation. All operations, including Erase, lock only the address that is actually being accessed.

It is somewhat of a misnomer to call this data structure a replicated sparse memory. While it is sparse in the sense that it supports the Erase operation, it is entirely unsuited to address spaces that are truly occupied sparsely. A more appropriate term for the data structure might be *replicated array*. If the address space is of cardinality one, the data structure could be considered a *replicated variable*. Note that such a replicated variable is essentially identical to Gifford's original file suite data structure.

4.2. Extensions

4.2.1. Range Operations

RSMs can easily be extended to support *range operations*, analogues of the basic RSM operations that operate on entire ranges of addresses:

> **ReadRange(IN low, high: address; OUT addr: array of address, val: array of value)** - Returns all occupied addresses between *low* and *high*, inclusive, and the values associated with these addresses.
>
> **EraseRange(IN low, high: address)** - Erases all occupied addresses between *low* and *high*, inclusive.

Implementation of the ReadRange operation is fairly straightforward. The client requests information on the entire range from each representative in a write quorum, and merges all of this information into a single, current version of the range. The entries in the current version represent the occupied addresses in the range.

The EraseRange operation is implemented in similar fashion to Erase. In the Erase operation, the client locates the real predecessor and successor of the address being erased, and coalesces the range from the real predecessor to the real successor into a single gap. In the EraseRange operation, the client locates the real predecessor of the low endpoint of the range being erased and the real successor of the high endpoint, and coalesces this range into a single gap. The same procedure is used to locate the real predecessor and successor for EraseRange as is used for Erase (Section 2.5).

Added care must be taken when assigning a version number to the new gap formed in the EraseRange operation. In the Erase operation, the new gap is assigned a version number that is greater than the current version numbers associated with the address being erased, its predecessor gap, and its successor gap. In the EraseRange operation, the version number must be greater than the current version numbers associated with the predecessor gap of the low endpoint, the successor gap of the high endpoint, and every entry and gap from the low endpoint to the high endpoint inclusive. Each representative in the read quorum must traverse this range to determine its highest version number in the range. Once the representatives have determined these version numbers, they must return them to the client. This step does not increase the communication cost of the operation, as it may be performed at the same time as the first step of the real predecessor algorithm.

The implementations sketched above for the ReadRange and EraseRange operations are both efficient. In fact, the communication cost of each range operations is identical to the cost of the corresponding single-address operation, with one minor exception. When the ReadRange operation is performed, the representatives will have to send back multiple message buffers if they have too many entries and gaps in the range to fit into a single

buffer. But this is inherent in the semantics of the operation: it can return arbitrarily large quantities of information.

As specified above, ReadRange and EraseRange operate on *closed* ranges (ranges that include their endpoints). This was done merely for ease of exposition. It is fairly straightforward to modify these operations to allow the client to specify *open* ranges (which exclude both endpoints) or *half-open* ranges (which exclude one endpoint). Modifying the ReadRange operation is trivial. Modifying the EraseRange operation is slightly more complex. If the client wishes to exclude the low endpoint, which we shall call *l*, from the range being erased, the low endpoint of the gap to be coalesced is determined as follows. If *l* is occupied, it is used as the low endpoint of the range; otherwise, its real predecessor is used. The check to see if *l* is occupied may be combined with the first stage of the real predecessor determination; it does not increase the communication cost of the operation. The high endpoint is treated analogously.

A range analogue of the Write operation can also be added to the RSM. Unlike ReadRange and EraseRange, the WriteRange operation fundamentally changes the character of the sparse memory abstract data type. As described in Chapter 2, RSMs can have only a finite number of occupied addresses. The WriteRange operation would permit the client to associate a value with all of the addresses in a range, giving rise to *occupied ranges*. If the address space is a dense set, occupied ranges contain infinitely many occupied addresses.

Note that RSMs with dense address spaces that support the WriteRange operation must also support EraseRange. Otherwise there would be no way to undo the effects of a WriteRange. Once a range became occupied, all but a finite number of the addresses in the range would remain occupied forever.

Internally, the WriteRange operation permits values to be associated with gaps as well as entries at representatives. This changes the workings of the Erase operation slightly. If an address being erased lies in an occupied range, the address does not have a unique real predecessor or successor: any address sufficiently close to the address being erased is occupied. In effect, the gap from the real predecessor to the real successor degenerates to a singleton range containing only the address being erased. To erase such an address, an entry that asserts that the address is unoccupied is inserted at each representative in a write quorum. We call these entries *zombies*. Unlike the *tombstones* inserted by the low-latency Erase operation, there is no follow-up operation to remove zombies: they remain until the surrounding range is erased with EraseRange.

The analysis in Chapter 3 is no longer directly applicable if the EraseRange operation is supported. However, we conjecture that the favorable space and time performance characteristics of the RSM are preserved. In particular, zombies do not represent a

garbage collection problem. As long as the range in which a zombie lies remains occupied, the zombie represents current information. If the range is erased with EraseRange, the zombies in the write quorum are deleted, and those outside the write quorum become ordinary ghosts, which are deleted by subsequent Erase or EraseRange operations.

4.2.2. Navigation Operations

The basic RSM operations offer no way for a client to determine which addresses in an RSM are occupied. If the ReadRange operation is supported, it can be applied to a portion of the address space to determine which addresses in this range are occupied. But this can be very expensive and damaging to concurrency, as ranges may contain arbitrarily many occupied addresses. RSMs can easily be extended with operations that allow the client to navigate through the occupied portions of the address space directly:

> **ReadNextOccupiedAddr(IN addr: address)** - Returns the first address greater than *addr* that is currently occupied, and the value associated with this address.
>
> **ReadPrevOccupiedAddr(IN addr: address)** - Returns the first address less than *addr* that is currently occupied, and the value associated with this address.

The **ReadNextOccupiedAddr** operation is equivalent in function to the RealSuccessor operation, and **ReadPrevOccupiedAddr** to RealPredecessor. Thus, implementing the navigation operations consists merely of exporting RealPredecessor and RealSuccessor, which must be implemented anyway to support the Erase operation.

Note that the semantics described above for the navigation operations are inappropriate in the context of an RSM that supports RangeWrite. If an address lies in an occupied range, it has no distinct next or previous occupied address. The semantics of the navigation operations could be extended in several ways to support occupied ranges. For example, **ReadNextOccupiedAddr** could be modified to return the next **unoccupied** address after the given address, if the given address lies in an occupied range, along with a flag indicating whether or not the given address lies in an occupied range.

Chapter 5
The Use of Replicated Sparse Memories

In this chapter we discuss the application of the replicated sparse memory data structure presented in Chapters 2, 3 and 4. Section 5.1 describes the variety of data structures that can be obtained by varying the implementation parameters of the RSM. Section 5.2 discusses the efficiency that can be achieved with data objects implemented on top of RSMs and Section 5.3 presents efficient RSM implementations for various data types. Section 5.4.1 discusses the *counter*, a data type that can *not* be implemented efficiently on top of RSMs. An efficient replicated implementation for counters that does not involve RSMs is presented.

5.1. Parameters

The replicated sparse memory is defined quite generally. It has several parameters that may be varied to provide a rich family of data types. The most important parameters are the address space, the type of data associated with occupied addresses and the representation of the underlying RSM representatives.

Almost any address space may be used in an RSM. The only restriction on the address space is that it be a totally ordered set. The cardinality of the set may be finite or infinite. The set may be sparse or dense. If a range of positive integers is used, a traditional random access memory organization is obtained. If the variable length character strings with lexicographic ordering are used, a directory is obtained. If the floating point numbers are used and the RangeWrite function is implemented (Section 4.2.1), certain real functions can be represented. Addresses consisting of ordered tuples with lexicographic ordering yield hierarchical file organizations. The range operations are particularly useful in combination with hierarchical file organizations.

No restrictions exist on the data that may be stored in an RSM: a byte string of arbitrary length may be associated with each occupied address. The byte string will typically be interpreted as a simple or complex data item in the host language (e.g. an integer, a string, or most commonly, an aggregate data structure). It is not necessary that all occupied addresses in an RSM have the same amount of data associated with them, or

that successive values written to an address be of equal size. Many applications, however, will have the property that all occupied addresses in a single RSM have the same amount of data associated with them. Efficient implementations will take advantage of this.

Various representations may be used for the representatives that make up an RSM. The application for which an RSM is used dictates what sorts of operations will be performed on it. This, in turn, dictates which representation for the representatives will result in efficient implementations. If an RSM is used in an application that guarantees that all operations will occur at or near the lowest or highest occupied address in the RSM, it is wasteful of time and space to represent the representatives as balanced trees or skip lists, which allow random access. Under these circumstances, a linked list is the appropriate data structure. If an application requires random access but does not require any range operations or navigation operations, a hash table RSM is appropriate (Section 4.1.5). If an application calls for a small address space, or a larger finite address space that will be almost completely occupied, an array RSM is called for (Section 4.1.6).

The number of representatives in an RSM and the number of votes assigned to each representative may be varied to select the desired availability. The read and write quorum sizes may be adjusted to vary the relative availability of different RSM operations. For a fixed number of representatives, the larger the write quorum, the more available the Read operation will be relative to the Write and Erase operations. The selection of quorum sizes is discussed in more detail in Section 3.3.6. Considerable research has been done on the topic of vote assignment and quorum size selection [21].

5.2. Efficiency Guidelines

Section 5.3 presents efficient implementations for several data types built on top of RSMs. First, we must say what it is that we mean by *efficient*. We do not have precise criteria for what constitutes satisfactory efficiency, but we do have rough guidelines. Our goal in formulating these guidelines was to demarcate a good, achievable engineering tradeoff between communication, computation and space costs, ease of implementation, concurrency and availability. The guidelines are based on the efficiency that RSMs display when used directly; essentially, we want data types implemented on top of RSMs to be of comparable efficiency to the RSMs themselves.

We are concerned with three performance measures: *communication cost*, *computation cost* and *space cost*. We are primarily concerned with *average* costs, where the averages are computed over sequences of operations that are representative of the data object in actual use. We also consider it important that the *worst case* costs are bounded by some reasonable values.

The most important measure of the complexity of a distributed algorithm is the *communication cost*. In current day distributed systems, and for the foreseeable future, communication costs tend to dominate local computation costs in efficient replication algorithms. The communication cost measure of primary concern to us is the number of rounds of message exchanges required between the node performing an operation and the remote nodes involved in the operation. Each round may involve one or more remote nodes. The messages must be bounded in length by some system dependent constant. (Eight K bytes is typical.) A communication cost measure of lesser concern is the total number of messages sent in performing the operation. It is of lesser concern because the number of rounds is a much better indicator of the latency that will be incurred in performing the operation.

Operations that do not modify an object's state should require only a single round of messages. Similarly, operations that modify an object's state in a fashion unrelated to its current state (e.g. the RSM Write operation) should usually execute in a single round of messages. We call such operations *state-independent modify* operations. We say they should *usually* execute in a single round because optimistic timestamps and other optimistic two-stage protocols are required to achieve this efficiency, and optimistic assumptions will occasionally be incorrect.

An important exception to the above guideline is the RSM Erase operation, or any other state-independent modify operation that relies on the Erase operation on an underlying RSM. Unless a hash table or array RSM can be used, these operations will require two, rarely three rounds of messages. However, they perform the ‘‘garbage collection’’ that is responsible for giving the RSM data structure its self-cleaning property, so we consider the additional round of messages to be acceptable.

Operations that modify a data object in some way that depends on its value should usually execute in two rounds. Such operations, which we call *state-dependent modify* operations, typically fall into the *two-stage protocol* framework described in Section 4.1.2. Therefore, it will sometimes be possible to execute such operations in a single round if the state on which the modification depends is predictable. It is not feasible to perform all state-dependent modify operations in two rounds. There may be operations that require two rounds just to get the state: one round to find the address of the relevant information and one round to Read it. If the modification requires an Erase, two rounds will be required after the data is read.

By *computation cost*, we mean the computation time required to perform an operation, excluding the time spent doing communication. The computation cost of an operation can be divided into two parts, the time spent at each representative, and the time spent combining the information returned by the representatives. We desire that both of these components be at worst comparable to the processing time on a standard single site

implementation of the data type. Typically, the time required to combine the information from the representatives is a small constant depending only on the quorum sizes.

We define *space cost* as the size of the data stored at each representative. We desire that the average size of each representative be comparable to the size of the corresponding non-replicated data structure. The analysis in Chapter 3 shows that, under a wide variety of usage patterns, the number of entries in each representative of an RSM will on average be less than 1.2 times the number of occupied addresses, which has good implications for the space cost of our algorithms. While this result applies only to random operation mixes, and does not apply to hash table RSMs, we conjecture that it is actually a pessimistic indicator of the space cost that can be expected from an RSM in practice. (See Section 3.4.)

Our basic concurrency requirement is that operations that commute should generally be allowed to execute concurrently. For instance, operations on different keys in a directory, or an Enqueue and a Dequeue operation on a nonempty queue should be able to execute concurrently.

Our basic availability requirement is that the system should be able to tolerate one or several node failures and still perform all of the operations offered by a data type. We are less concerned with being able to adjust the relative availability of the operations. Sometimes our implementations allow various quorum choices which permit us to trade off the relative availability of the operations on a data type. We view this as a useful feature, but not as our primary goal.

5.3. Data Types Built on RSMs

Since an RSM can be made to look like primary memory and accessed using all of the data structuring facilities of the host language, a single site serial implementation of *any* data type can be mechanically converted into a replicated implementation. Such implementations, however, will usually be inefficient and display poor concurrency performance. In this Section, we present efficient RSM implementations for several data types.

5.3.1. Singly Indexed Record Sets

Singly indexed record sets are data types with operations to associate information with a key, and read, alter or remove the information associated with a key. Examples include directories, dictionaries, sets and multisets. The RSM data structure was developed from a replicated directory data structure [10] so it should come as no surprise that it is straightforward to implement directories and their kin efficiently on top of RSMs.

The directory data type supports the following operations:

Insert(IN k: key, v: value) - Inserts the key *k* into the directory and associates the value *v* with *k*. This operation is permitted only when *k* is not already in the directory.

Update(IN k: key, v: value) - Associates the (new) value *v* with the key *k*. This operation is permitted only when *k* is already in the directory.

Delete(IN k: key) - Removes *k* from the directory. (This operation is permitted only when *k* is in the directory.)

Lookup(IN k: key; OUT present: boolean, v: value) - Returns TRUE, and the value associated with the key *k*, if *k* is in the directory. Returns FALSE and an unspecified value if *k* is not in the directory.

If the client attempts to perform an operation that is not permitted, an exception is returned by whatever mechanism the host language provides and the contents of the directory remain unchanged. If no exception handling mechanism is provided, an additional OUT parameter must be added to the Insert, Update and Delete calls to allow the directory to indicate whether the operation was permitted.

A directory is stored in an RSM whose address space is the directory's key space. The straightforward way to perform an Insert operation is to Read the relevant address in the RSM, and Write the value out if the address is unoccupied. This requires two rounds of messages. If optimistic timestamps are used, however, this can be cut down to one round, in most cases, by using the ReadWrite operation (Section 4.1.1). With the ReadWrite operation, a single round generally suffices to store the new value *and* check if the key was already in the directory prior to the operation. If the key was already in the directory, the Insert is illegal, so it is undone with a Write operation. It is interesting to note that the resulting operation is doubly optimistic: it assumes that the address to be written was previously unoccupied, and that the optimistic timestamp is higher than any version number previously associated with the address.

Using the technique described in the previous paragraph, the Insert operation usually requires only one round of messages. The same technique can be applied to the Update operation. The Lookup operation always requires only one round. The Delete operation requires one, rarely two rounds if a hash table RSM is used (Section 4.1.5).

The directory implementation does not require range operations or navigation operations, so the hash table RSM is the appropriate representation to use, assuming the approximate number of keys that will be stored in the directory is known. It is possible to extend the directory with range operations, which lookup or delete all of the keys in a given range. If this is desired, the RSM representatives should be implemented as balanced trees or skip lists. In this case, two, rarely three rounds of messages will be required for the Delete operation.

The concurrency of the directory implementation is good but not optimal. Operations on different keys can generally proceed concurrently. For each key, a single transaction can perform write operations (Insert, Update and Delete) at any given time. Multiple transactions can lookup the same key concurrently. Multiple write operations could proceed in parallel if hybrid atomic methods were used [31]. The availability of Lookups can be traded off against that of write operations, but write operations must access a majority of the representatives. Updates (though not Inserts or Deletes) could be performed with fewer representatives at the expense of lookups if hybrid atomic methods were used.

The directory implementation can be modified to implement a set by eliminating the value field from the entries in the RSM. A multiset can be represented by replacing the value field with a count field. Many similar variations are possible.

5.3.2. Multiply Indexed Record Sets

This section describes two implementations for a data type that provides record level storage facilities for databases. The data type is known as the *record file with secondary indices on selected fields*. According to Date, it is ‘‘one of the most common storage structures in current use.’’ [17]

Each record in the file consists of a primary key, k_1, secondary keys, k_2 - k_n, and a non-key data part, *d*. The primary key must be unique: no two records can have the same primary key. Files are accessed with the following operations. (The notation key_i refers to the data type of the *i*th key.)

Insert(IN r: record) - Inserts the record r into the file. If a record already exists with the same primary key, an error is signaled.

Lookup(IN kp: key_1; OUT present: boolean, r: record) - Returns TRUE and the record with primary key *kp*, if the file contains such a record. If the file contains no record with the given primary key, FALSE is returned with an unspecified record.

Lookup2(IN i: integer, ks: key_i; OUT r: array of record, n: integer) - Returns in *r* all of records in the file whose *i*th secondary key is *ks*. The number of records in *r* is returned in *n*.

Delete(IN kp: key_1) - Deletes the record with primary key *kp* from the file. If there is no record in the file with this primary key, an error is signaled.

ModifySecondaryKey(IN kp: key_1, i: integer, ks: key_i) - Changes the *i*th secondary key of the record with primary key *kp* to *ks*. If there is no record in the file with the given primary key, an error is signaled.

ModifyData(IN kp: key_1, d: data) - Changes the non-key data part of the record with primary key *kp* to *d*.

The Insert, ModifySecondaryKey and ModifyData operations could be combined into a single *Write* operation, whose function varied depending on the data previously associated with the given primary key. We chose to treat the three cases separately, as each case demands a different algorithm, and the three algorithms have different communication costs associated with them. It is fairly straightforward to combine the Insert, ModifySecondaryKey and ModifyData procedures described in this chapter into a single procedure whose behavior and performance varies depending on which case applies.

The record file with secondary indices can be implemented as follows. Records are stored in a single RSM that is indexed by primary key. A separate RSM is used to represent each secondary index. The address space for the *i*th index RSM (for $i > 1$) consists of ordered pairs of the form (ks, kp), where $ks \in key_i$ and $kp \in key_1$. The address space is ordered lexicographically. Each index RSM has one *item* (occupied address) for every record in the file. The item for the record with primary key kp in index RSM i has address (ks_i, kp), where ks_i is the *i*th secondary key of the record. No values are associated with the items.

To do an Insert, the record is written to the main RSM and appropriate entries are written to each index RSM. The client must check that there was no record in the file for the given primary key at the time of the Insert. This can be done at no added communication cost using the optimistic two-stage protocol described for the Insert operation in the Directory data type (Section 5.3.1). To do a Lookup2, the range from (ks, k_{1min}) to (ks, k_{1max}) (where k_{1min} and k_{1max} represent the highest and lowest keys in the primary key space) is read from the relevant index RSM. Each occupied address in the resulting list corresponds to a record with the correct secondary key value. The primary key of each address on the list is extracted, and the records corresponding to these keys are read from the main RSM.

To do a Delete, the record is read from the main RSM. This enables the client to determine the addresses of all of the relevant items in the index RSMs. These items, as well as the record in the main RSM are then erased. All of the erases can be performed in parallel. To do a ModifySecondaryKey, the record is read from the main RSM. Then it is written back to the main RSM with the modification. The old item is erased from the relevant index RSM, and a new item is added to this RSM to reflect the new secondary key value. The Erase and the two Writes can be performed in parallel, though care must be taken to ensure that the Erase and the Write to the index RSM do not interfere with one another. Implementation of the remaining operations (Lookup and ModifyData) is straightforward.

No range operations or navigation operations are required on the main RSM, so a hash table RSM is appropriate. The Lookup2 operation requires RangeReads on the index

RSMs, so the representatives of these RSMs are best represented as balanced trees or skip lists. The communication cost of this implementation is summarized in Table 5-1. Note that some operations have two values listed for their communication cost. The higher value indicates the cost in cases where an optimistic timestamp is too low or the real predecessor operation requires two rounds. The lower value should apply most of the time.

Operation	**Rounds of Message Exchanges**
Insert	1 or 2
Lookup	1
Lookup2	2
Delete	3 or 4
ModifySecondaryKey	3 or 4
ModifyData	2 or 3

Table 5-1: Communication Costs for the Record File with Secondary Indices

The first round of the Delete, ModifySecondaryKey and ModifyData operations merely looks up the record being deleted or modified. Therefore, the costs of these operations can be reduced by one round if they are preceded in the same transaction by a Lookup of the same record.

The communication cost of the ModifySecondaryKey operation can be reduced to two, rarely three rounds for secondary keys that are *nearly unique*. A key is nearly unique if few records have the same value for the key (e.g. employee name). Hash table RSMs can be used to represent the indices for such keys, even though range operations are required on secondary index RSMs, if the hash functions are chosen carefully. It is imperative that the range from (ks, k_{1min}) to (ks, k_{1max}) contains all of the entries with secondary key ks. If hash functions are chosen so that all keys with a given secondary key value land in the same bucket, this property will result. To achieve this effect, the hash functions merely ignore the primary key component of the address. This technique must be used with caution, as poor performance and concurrency will result if it is applied to a secondary key that does not have the required near-uniqueness property.

Our first implementation of the record file with secondary indices has one major performance problem: the Lookup2 operation requires two rounds of messages. For many applications, Lookup2 will be the most common operation, as primary keys are often artificial or obscure. Lookup2 is a read-only operation, so our efficiency guidelines dictate that it should generally run in one round (assuming the returned list of records is small enough to fit in a message buffer).

Lookup2 requires two rounds because the secondary indices store primary keys, rather than storing the records themselves. It takes one round to get the relevant primary keys and one more round to dereference them. We could avoid this expense by storing copies of the records in each secondary index, but this would require as many copies of the records at each representative as there were indices. Furthermore, the ModifyData and ModifySecondaryKey operations would have to write updated copies of the record to all indices. The space and computation costs would be prohibitive. But we make the following observation about this implementation: the usage discipline imposed by the data type ensures that all of the RSM representatives at a given node will always contain identical data for each primary key they address. Thus, each node need only store a single copy of the data for every primary key that it knows about. The entries in all of the index RSM representatives can store pointers to the data in the main RSM representative rather than storing their own copies of the data. This reduces the data storage requirement of this implementation so that it is comparable to that of the original implementation.

The revised implementation of the record file with secondary indices cannot be built on top of ordinary RSMs; in order to allow multiple RSM representatives to point to the same piece of physical data, the RSM model must be extended. We call the resulting distributed data structure a *multiple replicated sparse memory* or *MRSM*. The RSMs that constitute an MRSM are permanently ganged together, and are only accessible with MRSM operations, which are essentially identical to the record file operations. Each MRSM representative consists of one RSM representative for each of the constituent RSMs.

The Insert operation causes the record to be (logically) inserted into all of the RSMs that constitute the MRSM: at each MRSM representative in the write quorum, an entry for the record is inserted into each of the component RSM representatives. All of these entries at a given MRSM representative point to a single copy of the record. The Lookup and Lookup2 operations are straightforward.

The implementation of the Delete operation is essentially unchanged from the original implementation of the record file. There is one additional subtlety. When a record is deleted from an MRSM representative, there may still be ghost entries that point to the record in some secondary index RSM representatives. For example, suppose a record is inserted into the MRSM, a secondary key of the record is modified using a different write quorum, and then the record is deleted using the original write quorum. At each MRSM representative in the write quorum for the Insert and Delete that was not in the write quorum for the Modify, the RSM representative corresponding to the secondary key that was modified will contain a ghost entry for the original value of the modified key. Future Lookup2s on this secondary index will cause the record pointer of the ghost entry to be dereferenced, but the record no longer exists.

The obvious solution to this problem is to put a reference count in each record. But this solution is unnecessarily inefficient. It entails retaining each record until the last ghost entry for the record is deleted, which increases the average size of the representatives. More seriously, it adds to the computation cost and decreases the concurrency of the Delete and ModifySecondaryKey operations. We present two alternate solutions, both superior to reference counting.

The first solution consists of ignoring the problem. Surprisingly, this technique will yield correct behavior in most systems. The data pointed to by the dangling pointer may be *read* by the Lookup2 operation but it will never be written. The data may well be garbage, but its version number, which is stored in the entry, will be valid. The data and its version number are passed back to the client together. Since the entry is a ghost, some other representative will have a higher version number for the same address; the garbage data will always be ignored. This technique was suggested by MacManus [40]. The technique has three main advantages: it has no computation cost, no adverse effect on the concurrency of the file, and demands no implementation effort. It has one disadvantage: each entry in a secondary index RSM representative must contain a copy of its address (i.e. the secondary and primary key of the record to which it pertains). If reference counting were used, the address could be read from the record pointed to by the entry.

The technique described in the previous paragraph relies on one assumption: a valid pointer must remain valid for all time, in the sense that reading the data addressed by the pointer does no harm. This is true, for instance, in the Camelot system, where a server's recoverable segment cannot change in size or virtual address once the server has been started [7]. Even in a system where record buffers are physically deallocated when the record is deleted, it may be possible to determine that a pointer is no longer valid and suppress the dereference. It is, however, possible to imagine a system where there is no way to tell whether a pointer is valid and dereferencing an invalid pointer can cause harm to a server. For such systems, there is another solution to the dangling pointer problem.

The basic idea behind the solution is to prevent dangling pointers from forming by zeroing out incipient dangling pointers before the record they point to is deleted. If a Lookup2 operation causes a representative to read an entry whose pointer has been zeroed, the null pointer serves as an indication to the representative that the dereference is to be suppressed. The representative may return arbitrary data to the client, as it knows that the entry is a ghost. This technique is no more complex to implement than reference counting and provides superior computation performance and concurrency. As in the previous technique, entries in secondary index RSM representatives must contain copies of their addresses. Details of the technique are described in the next four paragraphs, which are fairly technical and may be skipped without loss of continuity.

Dangling pointers are formed when ghost entries in an index RSM representative point to a record at the time it is deleted. (A record is deleted from an MRSM representative when its entry in the main RSM representative is deleted.) The entries whose pointers will dangle after the record is deleted can be divided into two categories: those whose addresses correspond to the secondary key values that are currently specified in the record at the representative, and those whose addresses correspond to older values. Entries in the former category can be located at the time the record is deleted, and their record pointers can be zeroed at that time. Entries in the latter category are no longer accessible from the information contained in the record. These entries must be detected at the time they become inaccessible, and their record pointers zeroed at that time.

There are two events that can cause a record to be deleted from an MRSM representative: a Delete operation on its primary key and a RepCoalesce operation resulting from the Delete of a nearby key. In the latter event, any secondary index entries described by the record that still remain at the representative are ghosts, as the file no longer contains a record for the given primary key. The record pointers of all such entries must be zeroed. In the former event, it is only necessary to zero the record pointers of the secondary RSM entries corresponding to outdated secondary key values. Current entries in the index RSM representatives will be expunged as a result of the Delete operation, which affects both the main and index RSMs. While it is permissible to zero all of the entry pointers described by the record, it is expensive and unnecessary.

Since the Delete is preceded by a Read of the record, the client knows the current version numbers associated with each secondary key, as well as the current values for the keys. If the client passes the version numbers to the MRSM representative in the second stage of the Delete operation, cheap version number comparisons can be substituted for more expensive value comparisons to determine whether the index RSM entry for a given secondary key needs to have its record pointer zeroed.

There is only one event that causes an entry in an index RSM representative to become inaccessible from the relevant record in the MRSM representative. Suppose a ModifySecondaryKey operation takes place, and an MRSM representative in the write quorum already has a record for the given primary key. If the record's value for the given secondary key is up to date, the corresponding entry in the index RSM representative will be deleted by the Erase that occurs as part of the ModifySecondaryKey operation. But if the record's value for the given secondary key is out-of-date, the corresponding entry in the index RSM representative will not be deleted by the Erase. The secondary key value in the record that would allow the MRSM representative to find the entry will be overwritten by the new value, causing the entry to become inaccessible. The MRSM representative can detect this situation by examining the secondary key value in the record prior to overwriting it. If the old value is not the current value, the representative must locate the relevant (ghost) entry in the appropriate secondary RSM and zero its record pointer.

The ModifySecondaryKey operation is fairly straightforward. It differs from the original implementation in that the modified record must be written into every index RSM, not just the one corresponding to the changed key. This does not cause any additional communication, as all of the Writes can be done in parallel. The ModifyData operation is similarly affected: the modified record must be written to all index RSMs in addition to the main RSM. The ModifySecondaryKey operation can easily be extended to modify multiple secondary keys simultaneously.

Because records in MRSMs can be accessed via multiple RSMs, normal locking in component RSMs is not sufficient to guarantee the serializability of an MRSM; in addition, operations must secure locks on the records they access. As an optimization, only the Lookup2 ModifySecondaryKey and ModifyData operations need secure locks on records; for other operations, locking in the main RSM suffices to guarantee serializability.

In summary, the communication costs of the MRSM implementation of the record file differ from those of the original implementation only in that the cost of the Lookup2 operation is reduced to a single round. All of the other operations have the same communication costs shown in Table 5-1. The optimizations described for the original implementation apply equally well to MRSM implementation. The computation costs for the ModifySecondaryKey and ModifyData operations are slightly higher in the MRSM implementation, but this should not be very significant, especially if there are few secondary indices.

5.3.3. Queue-Like Data Types

The easiest way to implement a queue on an RSM is to use an RSM whose address space is the positive integers as a shared array, and build a standard array implementation of a queue on top of it. Two integer replicated variables (i.e. an array RSM that stores two integers) serve as head and tail cursors for the queue. Enqueues Read the tail cursor, store the item to be enqueued at the RSM location addressed by the tail cursor, and increment the tail cursor by writing back the next integer in sequence. The Writes to the RSM and the tail cursor can be done in parallel. Dequeues read the head cursor, read the RSM location addressed by the head cursor, and increment the head cursor, unless the address specified by the head cursor is unoccupied. In this case, the queue is empty. The Read of the element and the Write of the head cursor can be performed in parallel, though the Write will have to be undone if the address turns out to be unoccupied. (This is an optimistic two-stage protocol.)

This implementation requires two, rarely three rounds of messages to perform an Enqueue or a Dequeue. While this is not grossly inefficient, Enqueue is a common, state-

independent modify operation, so our efficiency guidelines say that it should generally complete in one round. We will describe a more complex implementation that achieves this goal. The techniques used in this implementation form the basis of our efficient implementations for all queue-like objects.

The implementation uses optimistic timestamps in a new way. Previously, we used them only to generate version numbers for blind write operations (Section 4.1.1). In our queue implementation, we also use them to order entries in the queue, in much the same manner as clock times might be used in a single site system. The queue is stored in an RSM whose address space is the integers large enough to hold optimistic timestamps. The basic idea of the technique is to Enqueue items by storing them at the address corresponding to the time that they are enqueued. Merely generating an optimistic timestamp at the client and storing an item at the location addressed by the timestamp is not sufficient to ensure the FIFO property. A different transaction could Enqueue with an optimistic timestamp lower than one used for the previous Enqueue, causing an item to be enqueued out of sequence. Multiple transactions could arbitrarily intersperse elements on the queue, compromising serializability.

The reason this technique does not work is that optimistic timestamps for Enqueues are not compared with the timestamps for previous Enqueues. The problem is corrected by associating a replicated variable with each queue and requiring each Enqueue operation to write this variable at the same time as it writes to the queue RSM, using the same optimistic timestamp. If the timestamp is insufficiently high to write the replicated variable, an additional stage is added to the operation. The initial Write to the queue RSM is undone and the Writes to the replicated variable and the queue RSM are redone with a sufficiently high optimistic timestamp. Using this technique, the Enqueue operation usually requires a single round of messages, rarely two rounds.

The Write to the replicated variable, which we call *EnqLock*, will only succeed if the optimistic timestamp for the Enqueue is higher than the highest optimistic timestamp previously associated with EnqLock. But the highest timestamp previously associated with EnqLock is the same as the highest address ever occupied in the queue RSM. Thus, the Write to EnqLock ensures that items will be enqueued in sequence. Furthermore, the locks secured in the process of writing to EnqLock ensure that Enqueues are serialized: once a transaction Writes to EnqLock, no other transaction can write to EnqLock until the first transaction commits. Items placed on the queue by different transactions will not be interspersed.

It makes no difference what value is written to EnqLock; it is acceptable to associate no data whatsoever with the replicated variable. All that matters is that the write lock on EnqLock is obtained at each representative in the write quorum, and the optimistic timestamp for the Write is confirmed to be higher than the highest version number associated with EnqLock at the time of the Write.

Note that this implementation requires a new RSM representative operation to undo the effects of an erroneous RepWrite to the queue RSM. Unlike our previous use of optimistic timestamps, the effects of attempting an Enqueue with a timestamp that is too low cannot be undone merely by doing another Write with a legitimate timestamp. The erroneous entry in the queue RSM has a different address from the correct entry, so it must be removed explicitly. It is entirely straightforward to implement the operation, which we call RepUndoWrite.

To Dequeue an item, the client must locate the oldest item in the queue RSM and Erase it. The oldest item will always be found at NextOccupiedAddr(**Low**). Locking for the Dequeue will occur naturally as a consequence of performing the underlying RSM operations: once a transaction has dequeued an element, no other transaction will be able to dequeue an element until the first transaction commits. If a transaction attempts to dequeue an element and observes that the queue is empty, no other transaction will be able to dequeue or enqueue an element until the first transaction commits.

Naively, the dequeue operation appears to require three, rarely four or five rounds of message exchanges: one, rarely two rounds to find NextOccupiedAddr(**Low**) and two, rarely three more rounds to Erase it. In fact, the second round of the real successor operation that is required to do the Erase will never be necessary: there will never be any ghosts between the addresses corresponding to the first and second items in the queue. It is often possible to locate the high boundary of the region to be coalesced in the Erase (i.e. the real successor of the real successor of **Low**) from the information returned in the first round of the NextOccupiedAddr(**Low**) operation. When this occurs, the Dequeue operation can be performed in two rounds.

The Dequeue operation cannot be performed in two rounds if it is implemented on top of the NextOccupiedAddr and Erase operations. To achieve this performance, the RSM must be extended with an operation that finds the real successor of an address *and* the real successor of the successor as quickly as possible. A natural form for this operation to take is a *DeleteNextOccupiedAddr* operation, which finds the next occupied address after a given address, returns the data associated with the occupied address, and erases it. Such an operation would require two rounds in the common case, three or four in rare cases. Implementation of this operation is straightforward. In fact, it does not require any new RSM *representative* operations beyond those required for the Erase operation.

In summary, the Enqueue operation requires one, rarely two rounds of messages and the Dequeue operation requires two, rarely three or four. Since the RSM used to represent a queue is always accessed from the bottom, the representatives are best represented as linked lists. With this representation the storage required at each representative will be comparable to the storage required for a non-replicated queue. The concurrency of the implementation is fairly good but not optimal. As long as the queue is not empty, it

allows one enqueueing transaction and one dequeueing transaction to run concurrently. An unfortunate artifact of the implementation is that when a transaction removes the last element from the queue, enqueueing transactions are prevented from running concurrently even if the dequeueing transaction never observes that the queue is empty by attempting to dequeue another element. In this respect, the array implementation is superior to the timestamp implementation. Hybrid atomic methods would permit multiple concurrent enqueueing transactions.

Stacks can be implemented in a manner entirely analogous to queues. The Push operation is identical to the Enqueue operation. The Pop operation is performed by reading and erasing the *last* occupied address in the RSM, instead of the first.

Priority queues can be implemented using a technique very similar to the one used for queues. The priority queue is represented as an RSM whose address space consists of ordered pairs of the form (Priority, Optimistic Timestamp), with lexicographic ordering. Priorities are chosen from the integers between one and some maximum value. Higher values represent lower priorities. Enqueues are performed by storing an item at the appropriate address in the RSM. In order to ensure serializability and FIFO ordering within priority classes, each priority class must have its own EnqLock. Thus we require a separate array RSM indexed by priority class alone to provide EnqLocks. Each Enqueue operation must secure the EnqLock for the priority class of the Enqueue. Dequeues are performed exactly as in ordinary queues.

Like our queue implementation, our priority queue implementation requires one round, rarely two, to Enqueue, and two rounds, rarely three or four to Dequeue. The representatives in the RSM used to represent a priority queue should be represented as balanced trees or skip lists, as random access is required for Enqueues. Hash table RSMs cannot be used, as the elements in the RSM must be organized sequentially in order for the Dequeue operation to function properly. The concurrency of the priority queue implementation is similar to that of the queue implementation: one dequeueing transaction and one enqueueing transaction in each priority class may run concurrently. A transaction that dequeues the last element from the priority queue may not run concurrently with any enqueueing transactions.

5.4. Replicated Counters

Not all data types can be implemented efficiently on top of RSMs. In this section we describe a useful data type called the *counter*, for which an RSM implementation would provide poor performance and concurrency. We present an efficient replicated implementation for this data type that provides the performance and concurrency lacking in RSM implementations. We discuss the use of the implementation in conjunction with RSMs.

The counter supports three operations: *Increment*, *Decrement* and *Value*. Increment adds some positive integer to the counter, Decrement subtracts from it, and Value returns the counter's current value. We will assume that all counters are initialized to zero, although the data type and our implementation of it can be trivially modified to allow initialization to arbitrary values. Counters that have no minimum or maximum legal values are called *unbounded* counters. Counters that have a minimum or a maximum but not both are called *bounded* counters. Counters that have both a minimum and a maximum are called *doubly bounded* counters. In bounded or doubly bounded counters, an attempt to increment or decrement beyond the bounds generates an error. The total sales volume for a cash register, or the total number of keys in a directory are natural examples of unbounded counters. A bank account is a natural example of a bounded counter.

It is entirely straightforward to implement a counter on top of a replicated variable. The Increment and Decrement operations read the value associated with the variable, add or subtract as appropriate, and write the new value back to the variable. The Value operation merely reads the value associated with the variable. The resulting implementation requires two rounds of message exchanges for an Increment or Decrement and one for a Value. The Increment and Decrement operations each secure write locks on the replicated variable, thus only one transaction at a time can increment or decrement the counter.

In unbounded counters, Increment and Decrement operations commute freely, so our efficiency guidelines say that multiple Incrementing and Decrementing transactions should be able to execute concurrently. For bounded counters with lower bounds, Increment operations commute, so multiple incrementing transactions should be able to execute concurrently. While it is not impossible to achieve this concurrency in an RSM based implementation, we strongly conjecture that it is impossible to do so efficiently. In Section 5.4.1 we present an efficient, highly concurrent distributed data structure for replicated counters.

5.4.1. An Efficient Implementation for Replicated Counters

This implementation is built on top of local (non-replicated) highly concurrent counters. It is trivial to implement such counters in a transaction system that supports *operation logging* [54, 36], and slightly more complex on a system that supports only *value logging*. Unlike our RSM implementation, our replicated counter implementation does not use version numbers. It does, however, rely on quorum intersection.

A replicated counter consists of N representatives, each of which has a certain number of votes associated with it; a Value quorum size, V; and an Increment quorum size, I.

The same quorum size (*I*) is used for Increment and Decrement operations. *V* and *I* must be chosen so that $I + V > N$, which assures that every value quorum intersects every intersect quorum. As in our treatment of RSMs, we assume that all representatives have one vote assigned to them, but all of our results generalize to arbitrary vote assignments. We use the notation *N-V-I* to refer to a replicated counter with *N* representatives, a Vote quorum size of *V* and an Increment quorum size of *I*. Each replicated counter representative consists of a collection of local (non-replicated) counters, one for each Increment quorum that the representative is a member of. All of the counters in a representative are initialized to zero.

Let us first consider unbounded counters. To perform an Increment operation, the client selects an increment quorum consisting of *I* representatives. The client sends a *CounterRepIncrement* RPC to each member of the Increment quorum. The RPC takes two arguments, the number to be added to the replicated counter, and the increment quorum being used for the operation. (Each increment quorum must have an ID associated with it. An easy way to generate these IDs is to assign one bit in a word to each representative in the counter, and set the bits of all representatives in a quorum to form its ID.) Each representative processes the RPC by selecting the local counter corresponding to the given Increment quorum and adding the specified number to it. This causes the transaction to secure *increment locks* on the (local) counters. (An increment lock allows other transactions to increment a counter, but prohibits them from reading it.)

The decrement operation is similar to the increment operation in all respects, except that the specified value is subtracted from the relevant local counter at each representative instead of being added to it.

There are several key facts to notice about the Increment operation. Each invocation of the operation increments the local counters pertaining to one and only one Increment quorum. Furthermore, each invocation increments *all* the local counters pertaining to its Increment quorum. The Increment and Decrement operations are the only operations that modify the local counters, so all of the counters pertaining to a given Increment quorum will always be equal. Let us call the value shared by all of the counters for an Increment quorum its *counter value*. The Increment operation preserves the invariant that the sum of the counter values for every Increment quorum is equal to the value of the replicated counter.

To perform a Value operation, the client selects a Value quorum consisting of *V* representatives. The client sends a *CounterRepValue* RPC to each member of the Value quorum. In response to this RPC, each representative reads the values of *all* of its local counters and sends them back in some form that allows the client to discern which Increment quorum each counter value pertains to. Since every Value quorum intersects

all Increment quorums, there is at least one representative in the Value quorum that is a member of any given Increment quorum. Therefore, when the client has received responses to all of the *CounterRepValue* RPCs, it knows the counter value for every Increment quorum. But the value of the replicated counter is merely the sum of all the counter values. The client adds up all of counter values it receives, discarding any duplicates, and the result is the current value of the replicated counter.

In the process of executing the *CounterRepValue* RPC at a representative, the transaction acquires *value locks* on all of the counters in the representative in question. A *value lock* allows other transactions to read the value of a counter, but prohibits them from incrementing it. Since the Value operation acquires values locks on at least one counter for each Increment quorum, no other transaction can execute an Increment operation concurrently. Since Increment operations acquire only increment locks and Value operations acquire only value locks, multiple concurrent Increment *or* Value transactions are permitted.

Increments on bounded counters with upper bounds and doubly bounded counters are implemented by executing the Value operation as described above, and then executing the Increment operation only if the value is sufficiently low to allow the Increment. Note that this causes Increment operations to conflict with one another as well as with Value operations. Decrements on bounded counters with lower bounds are analogous.

5.4.2. The Performance of Replicated Counters

In unbounded counters, Increment and Value operations each require one round of message exchanges. Recall that two rounds were required for Increments in the RSM implementation. If Increments on bounded counters are implemented as described in Section 5.4.1, two rounds of message exchanges will be required. However, this can be reduced to a single round in all cases where the operation is legal with the use of an optimistic two-stage protocol.

The purpose of the first stage of the Increment operation on a bounded counter is to determine whether the counter is low enough to allow the increment. In the optimistic two-stage protocol, the client assumes that the operation is legal and performs CounterRepRead and CouterRepIncrement operations in a single round. If the results of the CounterRepReads indicate that the operation was illegal, the client performs a second round to decrement the local counters that were incremented erroneously.

As described in Section 5.4.1, the CounterRepValue operation acquires multiple locks at each member of the Increment quorum, one lock for each counter at the representative. In fact, it is possible to replace all of these locks with a single lock on the entire collection of counters. To make this work, the CounterRepIncrement operation must

secure an Increment lock on the single lock representing the entire collection, instead of securing a lock on the individual counter being incremented. This optimization greatly reduces the locking cost associated with the Value operation and the space required to store locks at the representative. Somewhat surprisingly, it does not decrease concurrency at all.[9]

Let us consider the space performance of the replicated counter. Each representative consists of one local counter for each Increment quorum in which it participates. In an *N-V-I* counter, each representative participates in $\binom{N-1}{I-1}$ Increment quorums. For a given *N*, this expression will be highest when $V = I$ (i.e. the Value and Increment operations are equally available). Unfortunately, this may well be the most common usage.

In an $N - (N+1)/2 - (N+1)/2$ counter, the number of Increment quorums in which each representative participates is:

$$\binom{N-1}{(N-1)/2}.$$

By Stirling's formula, this is approximately equal to:

$$\frac{2^N}{\sqrt{2\pi(N-1)}}.$$

In other words, the amount of storage required at each node is essentially exponential in the number of representatives in the replicated counter. While this might appear troubling at first, we conjecture that replication levels will never be high enough to make it a problem in practice: as a rough guideline, we doubt that replication levels greater than five are of practical significance. For $N = 3$, each representative has two counters. For $N = 5$, each representative has six counters. Even for the unrealistically high replication level of $N = 7$, each representative has only twenty counters.

The implementation allows complete flexibility in trading off the availability of Increment and Value operations: *I* and *V* can take on any value between one and *N*, inclusive, subject to the constraint that $I + V > N$. In practice, it is unlikely that anyone would want to use $I = 1$, as the Value operation would no longer be available in the face of even one representative failure. Similarly, choosing $V = 1$ would unacceptably restrict the availability of the Increment operation.

---

[9] In a paper on the Camelot library [7], we argued that low-level operations like (local) reads and writes should *not* secure locks automatically, so that the programmer could exercise flexibility in locking to improve concurrency and performance. The optimization described in this paragraph would be impossible if the (local) increment operation secured a lock automatically.

5.4.3. The Use of Replicated Counters in Conjunction with RSMs

In this section we present two examples of the use of replicated counters in conjunction with RSMs. While the details are specific to replicated counters, the techniques presented in this section are illustrative of the manner in which other replicated data objects can be combined with RSMs to produce efficient replicated implementations of complex objects.

Suppose we want to extend an RSM implementation of a directory (Section 5.3.1) with a *Size* operation, which returns the number of keys in the directory. Reasonable performance can be obtained only if an explicit counter is maintained.

Let us examine the effect it would have on the performance of the directory if the counter were implemented on top of a replicated variable. The Insert operation in a normal directory requires a single round of messages, assuming the optimistic two-stage protocol is used. In a directory with the Size operation, each legal Insert operation must increment the counter. The increment operation on a replicated variable always requires two rounds so the cost of the Insert operation will almost double. In a normal directory, multiple transactions can Insert different keys concurrently. If the count is maintained in a replicated variable, only a single transaction at a time can increment it, hence multiple transactions can no longer do Inserts concurrently. Representing the count as a replicated variable dramatically reduces the concurrency of the directory.

Maintaining the count in a replicated counter instead of a replicated variable solves both of the problems described in the previous paragraph. In a normal directory, the Insert operation usually requires one round unless the key is already in the directory, in which case a second round is required to undo the effects of the erroneous Write. The replicated counter can be incremented in the first round, and decremented in the second round, if it takes place; either way, the communication cost of the operation is unchanged. Multiple transactions can increment a replicated counter concurrently, so the concurrency of the Insert operation will not be reduced by incrementing the counter.

Suppose we use an RSM to represent a file of bank accounts, indexed by account number. Account records consist of the customer’s name, address and bank balance. If we represent the balance as ordinary data in the RSM, deposits and withdrawals are implemented by reading the record, modifying the balance field, and writing it back. This requires two rounds of messages, and prevents multiple deposits to the same account from taking place concurrently.

These deficiencies can be corrected by associating a replicated counter with each occupied address in the RSM. This is done by adding a counter representative to each entry in the RSM representatives. The Value quorum size for each replicated counter is

the same as the read quorum size for the RSM, and the Increment quorum size is the same as the write quorum size. When an entry is inserted into an RSM representative, all local counters in its counter representative are initialized to zero. When a preexisting entry in the RSM representative is modified, its counter representative is not affected. When an entry is deleted, its counter representative is deleted as well.

When an entry is read from the RSM, each representative returns its entry for the given account and all the counter values in the entry's counter representative. If a representative has no entry for the account, all of its counter values are assumed to be zero. The client selects the data from the entry with the highest version number, with the exception of its counter data. If the highest version number for the account belongs to a gap, the file contains no record for the account and the counter values are irrelevant. Otherwise, the account balance is determined by adding all of the returned counter values, discarding duplicates. In essence, the client is executing an RSM Read and a counter Value operation in parallel.

In the Deposit operation, each RSM representative in the Value quorum looks up the relevant entry. If no entry is found at a representative, an entry with version number zero is inserted. This entry associates a counter representative with the account without affecting the contents of the RSM. (The local counters in the new counter representative are initialized to zero.) After looking up or inserting an entry for the account, each representative adds the amount of the deposit to the relevant local counter in the entry's counter representative.

The withdrawal operation is just like the Deposit operation, except that the RSM representatives in the Value quorum send back their counter values, and the client checks to make sure that there are sufficient funds in the account to cover the withdrawal. If there are insufficient funds, another round is performed to undo the attempted overdraft. The second round is identical in implementation to the Deposit operation.

Unlike the pure RSM implementation, the hybrid implementation performs deposits in one round, usually performs withdrawals in one round, and allows concurrent deposits to the same account. The communication costs and concurrency of other operations are identical in the two implementations. Note that the hybrid implementation cannot be built on top of the representative operations for RSMs and replicated counters. The representative operations must be combined to implement the protocols described above.

There is one problem with the implementation as described above. If an account is erased and another account is inserted with the same account number, ghost entries remaining from the first incarnation of the account can cause erroneous balance readings. Version numbers assure that the data in these ghost entries will not be used, but an entry's version number does not apply to its counter representative. The easiest

‘‘solution’’ to this problem is not to reuse account numbers. If it is essential that account numbers be reusable, the problem can be solved by associating a version number with each counter representative, and storing in each record the version number with which the record was originally created. This allows the client to detect responses from counter representatives that were created before the account to which they apply. For brevity’s sake, we omit the details of this scheme.

Chapter 6
An Architecture for Replication

In this chapter, we present an architecture that provides programmers of a general purpose distributed transaction system with the replicated data objects described in Chapters 2 - 5. The main features of the architecture are the ease with which it can be used and the wide variety of data types that it provides. We built a fairly complete prototype system embodying the architecture. The prototype implements all four of the major RSM variants. It incorporates many of the optimizations that would be used in a commercial implementation.

The remainder of this chapter is organized as follows. Section 6.1 describes and motivates the architecture and Section 6.2 describes our prototype implementation in detail. Section 6.3 briefly evaluates the architecture and presents several lessons we learned from the building the prototype.

6.1. Architecture

Our replication architecture assumes an underlying general purpose distributed transaction system supporting the client-server model. The architecture consists of two main layers: a collection of *representative servers* and a *replicated data system library* (RDS library). Each representative server manages a collection of representatives, one for each replicated object in the system. Representative servers support operations to create, lookup and destroy representatives, and *representative operations*, like the ones in Section 2.4.1. Operations on replicated objects are invoked by calling procedures in the RDS library. These procedures execute RPCs to the appropriate representative servers and process the results, implementing the replication protocols described in previous chapters.

Conceptually, there need only be one representative server per node, managing data for arbitrarily many clients. However, if only one representative server were used at each node, the set of object names would constitute a shared global name space. Concerns of naming, resource allocation, security and failure isolation suggest that a separate collection of representative server processes should run for each application that uses our system.

The system provides its clients with objects of several *primitive data types*, principally the RSM variants described in Chapters 2 - 5. Other primitive data types might include replicated counters (Section 5.4) and nondeterministic types like *weakly fifo queues* (not covered in this dissertation). The primitive data types provided by the system are *generic*: the base types from which complex primitive types are constructed may be specified by the client. For instance, the system's RSM type allows the client to specify the RSM's address space and value space.

The system might also implement one or more *derived data types* on top of the primitive data types. For example, a priority queue could be implemented on top of an RSM and a replicated variable using the technique described in Section 5.3.3. The architecture is illustrated in Figure 6-1. Note that multiple clients may share replicated objects.

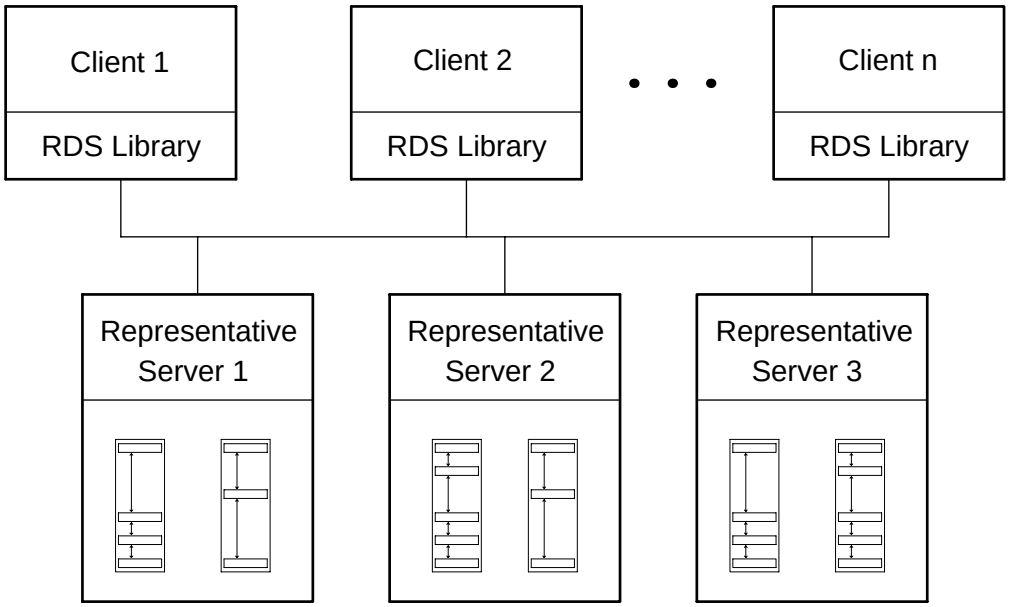

Figure 6-1: Replication System Architecture

A client wishing to create a replicated object calls the *Create* procedure for the appropriate type in the RDS library. The create procedure takes a name by which the object will be referred to and various parameters describing the particulars of the object. These parameters include the names of the representative servers at which representatives for the object are to reside, quorum sizes, and various *type-specific parameters*. An RSM, for example, has the following type-specific parameters: the data type of addresses in the RSM, the data type, or at least the size in bytes, of the values in the RSM, and the representation and size of the underlying RSM representatives.

The Create procedure generates a *serial number* for the new object, to be stored at each of the object's representatives. This allows the RDS library to confirm that a set of

representatives all refer to the same object. Ideally, the serial numbers generated by Create should be unique over all time. Alternatively, a pseudo-random number generator can be used to generate serial numbers. Serial numbers generated in this fashion give only a statistical guarantee that a set of representatives refer to the same object, but the probability of error can be made small by using random numbers of sufficient size. Care must be taken in seeding random number generators so that serial numbers generated in different client processes are nearly independent.

The RDS library Create procedure executes *RepCreate* RPCs to each of the indicated representative servers to create representatives for the new object. If any of the representative servers is unavailable or already has a representative for an object with the given name, the Create procedure returns an error.

The RepCreate RPC takes as parameters the object's serial number, quorum sizes and type-specific parameters. RepCreate causes a representative server to dynamically allocate space for a representative from its persistent storage pool and initialize the new representative. The representative has a header in which the object's serial number, quorum sizes and type-specific parameters are stored. The RepCreate RPC returns a handle for the new representative that is its address in persistent storage. The RDS library uses the handle to refer to the representative in future RPCs to the server.

Each representative server contains a local (non-replicated), persistent directory of all the representatives currently stored at the server. The RepCreate RPC causes the server to make an entry in its directory for the newly created representative. The directory allows the server to translate object names to representative handles, which enables clients other than an object's creator to use the object via the RDS library's *Lookup* procedure, described below.

The RDS library maintains at each client a local, volatile directory of the replicated objects that it currently knows about. This directory is indexed by object name. Each entry contains the object's type, its quorum sizes and type-specific parameters, the names of the servers at which representatives reside, and the handles of the representatives. The Create procedure inserts an entry in this directory for the replicated object being created, and returns a handle that is a pointer to this entry. The client uses this handle to refer to the object in all of the RDS library procedures that operate on the object. The handle is only valid as long as the client process continues to run.

In each entry in the replicated object directory, the server names and associated representative handles are stored in an ordered list. The representatives at the head of the list constitute the current read and write quorums for the replicated object. If any of these representatives become unavailable, the RDS library reconfigures the quorums to exclude the unavailable representative by reordering the list.

A client wishing to use a pre-existing replicated object calls the Lookup procedure in the RDS library. Conceptually, the Lookup procedure needs only the name of the replicated object and the names of the servers at which representatives are stored. However, it may be prudent to include as parameters the type of the object, its quorum sizes, and its type-specific parameters. This allows Lookup to check that the client's notion of the named object is correct, providing additional type safety.

The Lookup procedure performs RepLookup RPCs to the given representative servers. The RepLookup RPC causes a server to locate the given object name in its representative directory and return the handle for its representative, as well as the information in the representative's header, including the object's serial number. The Lookup procedure checks to see that all of the returned serial numbers match, and that the information returned by the servers is consistent with the parameters provided by the client. If any of these checks fail, Lookup returns an error. If the checks succeed, Lookup creates an entry in the client's replicated object directory, functionally identical to the entry that was made when the object was created. Like the Create procedure, the Lookup procedure returns a pointer to the object's directory entry, which the client uses as a handle for the object.

Unlike the Create procedure, the Lookup procedure must succeed even if one or more of the object's representative servers are unavailable, assuming enough servers are available to perform operations on the object. Therefore, provisions are made in the object directory to indicate that the client has not yet succeeded in determining one or more of an object's representative handles. This permits the RDS library to defer the RepLookup operation on an unavailable representative.

The RDS library exports procedures for all of the operations supported by each of the replicated objects that it implements. For the RSM, it exports Read, Write and Erase procedures. Once a client has obtained the handle of a replicated object by creating it or looking it up, the client can access the object using any of the procedures exported by the RDS library for objects of the given type.

The primary motivating factor for our architecture is that distributed programming is difficult and time consuming. In the framework provided by our architecture, the application programmer is insulated from the difficulties of distributed programming. The data types provided by the RDS library provide the sole programming interface to the system; the representative servers themselves are never referenced directly by the application programmer. The generic nature of the data types provided by the RDS library allows the programmer to produce a wide variety of replicated abstract data types by merely parameterizing the library types appropriately. More complex data types can be implemented on top of one or more of the objects provided by the library.

In our generic type implementation, the client passes data types as parameters to the RDS library Create and Lookup procedures. This implies that the language in which the system is implemented should support *parametric polymorphism* [11]. But it is easy to build such support on top of a language that does not provide it, assuming the language is not too strongly typed. Our implementation takes this approach (Section 6.2.1.1). An alternate approach is to leave type parameters unspecified at object creation time, and to specify them implicitly when operations are performed on the object. This would be the natural approach if the system were implemented in a late-binding object oriented language like Smalltalk [25]. This approach would allow individual RSMs to contain elements of multiple types, though it is not clear how useful this feature would be in practice. The approach has some performance drawbacks, and requires special effort to ensure that all of the address types used for a given object are compatible.

A note should be added concerning the handles used in our architecture. Neither the handles used by the client to refer to replicated objects nor the handles used by the RDS library to refer to representatives are fundamentally important to the architecture. Clients could refer to objects by name in all of their dealings with the RDS library. Similarly, the RDS library could refer to an object's representatives by the object's name in all of its dealings with replica servers. However, this approach would require several additional hash table lookups for each operation on a replicated object. While the additional computation would probably not be very significant, it is unnecessary, and the use of handles does not add appreciably to the system's complexity or detract from its cleanliness.

6.2. Implementation

Our prototype implementation runs on the Camelot distributed transaction system [19], which relies on the support of the Mach operating system [50]. Our prototype was written in a C language [35] extension called the Camelot library [7]. The primitive data types supported by the prototype are the four major RSM variants described in this dissertation: RSMs, hash table RSMs, array RSMs and MRSMs. The prototype includes support for derived data types, though none were actually implemented.

The prototype is fairly complete. It faithfully implements the architecture described in Section 6.1, with several exceptions. The RDS library does not include a Destroy function; there is no way to reclaim the recoverable storage that was occupied by an object's representatives short of reinitializing the representative servers. The Lookup function requires that all representative servers with representatives for an object be running at the time the object is looked up. The prototype faithfully implements the replication protocols described in Chapters 2 - 5, including most of the suggested optimizations. The primitive data type implementations are quite realistic, with the few

exceptions discussed in Section 6.2.3. Most notably, the skip list RSM representative implementation does not display paging and concurrency performance comparable to that of a commercial B-Tree package.

The prototype was tested and found to be fairly robust (Section 6.2.5). Even after debugging was complete, the system still crashed on occasion, especially during highly concurrent use, but all of these crashes were traced to failures in underlying system software. The system was sufficiently stable that we were able to generate large quantities of high quality performance data, as described in Chapter 7.

6.2.1. System Structure

The representative servers in our prototype are ordinary Camelot servers. The representative server program takes a single command line argument, the server's name. The RDS library is an ordinary Unix library whose constituent object files are generated from Camelot library source files. To use the system, the programmer merely compiles an application with the RDS library, starts the representative servers for the application if they are not already running, and runs the application.

The replication protocols described in Chapters 2 - 5 do not make explicit use of nested transactions. However, when implementing these protocols, the failure isolation provided by nested transactions is invaluable in masking failures and providing good error semantics. The RPCs provided by the Camelot library are *transactional RPCs*: they provide exactly-once semantics by aborting the enclosing transaction if they do not succeed. If the RDS library makes RPCs to a set of representative servers from within the direct scope of the calling transaction, and one of the called servers is unavailable, the calling transaction aborts. All work done by the transaction prior to calling the RDS operation is lost, and the caller must retry the transaction in order to take advantage of the high availability furnished by the replication protocols. Furthermore, the abort code passed back to the calling transaction comes directly from the underlying communication facility; it provides little information as to the source of the abort. All in all, this is not a satisfactory implementation.

One solution to the problems in the previous paragraph is to nest each RPC in its own subtransaction. (The Argus language [39] automatically does this to all RPCs.) This allows the RDS library to identify an unavailable server and retry the failed representative operation on another server. The calling transaction is not aborted unless the RDS library chooses to do so, for example, because it cannot contact enough servers to complete the operation. If the RDS library must abort the enclosing transaction, it passes back an abort code indicating precisely the nature of the problem.

We experimented with this approach initially, but found the cost of the nested transactions to be prohibitive. The same effect can be had by nesting each entire *round* (i.e., each set of parallel RPCs to a collection of representative servers) in a single nested transaction. This allows the RDS library to identify an unavailable server and redo the entire round. The added cost of aborting and redoing the representative operations that succeeded is small. More significantly, this cost is only incurred when a server fails, which happens rarely. Therefore, it represents a good engineering tradeoff to enclose rounds rather than individual RPCs in subtransactions. Our prototype implementation takes this approach.

The RDS library is instrumented to maintain histograms of the numbers of rounds of RPCs required by each operation whose communication cost is not constant. The library has a *GetStats* call, which returns the contents of the histogram for an object. This facility allows us to monitor the effectiveness of optimistic timestamps and other optimistic two-stage protocols used in our system.

Several facilities required by our system were not provided by the C programming language, the Camelot library, or the standard Mach programming environment. We wrote several utility packages, described below, to provide these missing facilities.

6.2.1.1. Dynamic Data Specification Facility

As previously noted, our architecture requires a facility for passing data types as parameters to local and remote procedure calls. While the C programming language does not provide this facility, C is *weakly typed*: the *cast* operator allows great flexibility in choosing the interpretation of a piece of data. We wrote a package called *DDS* (short for Dynamic Data Specification), which takes advantage of C's weak type structure to add a *type descriptor* data type to the language.

DDS data objects consist of a pointer to some data and a type descriptor, which describes the interpretation of the data. DDS type descriptors can be constructed to describe simple data types (C's *char*, *int*, *short*, or *float* types), arrays, and aggregate structure types. The DDS package consists of a macro for creating a type descriptor constant and various functions for manipulating DDS data objects. Functions are provided to compare DDS data objects (lexical comparison), hash them, print them, compute their sizes, and compute byte offsets of fields within them.

DDS operations run interpretively (viewing the type descriptor as a program). They typically require at least one function call per field in a DDS object. This implementation does not yield extremely high performance, but it is sufficient for our purposes; in our prototype system, DDS operations always execute in conjunction with an RPC, whose cost swamps that of the DDS operations.

6.2.1.2. RPC Batching Facility

The MRSM data type (Section 5.3.2) stores each index in a separate RSM. Certain operations on the MRSM perform complex multistage operations on each of these component RSMs. In order to achieve the low communication costs detailed in Section 5.3.2, the component RSM operations must be parallelized: RPCs that perform representative operations on component RSMs must be combined. In essence, the RPCs going to each server must form ‘‘carpools’’ to save on communication costs. In this way, the number of rounds of RPCs required to perform an MRSM operation will equal the maximum number of rounds required to perform a component RSM operation, not the total of the numbers of rounds required to perform each component operation. There are several ways to achieve this effect.

One could manually combine logical RPCs on several component RSM representatives into a single RPC. This would entail providing the stub generator with interface descriptions for each possible ‘‘compound RPC’’. Since the multistage operations on component RSMs are optimistic in nature (i.e., they may omit stages), the number of possible compound RPCs is large, even if only two component RSMs are involved. This approach would at best be tedious, and the resulting program would be large and messy. The approach could not be applied to MRSMs with arbitrarily many indices, as the number of possible compound RPCs would be unbounded.

A far more satisfactory approach is to combine RPCs dynamically at runtime. This is done by extending the communication system to support three new operations: an operation to initialize a *batch*, an operation to enqueue an RPC into a batch for later execution, and an operation to execute in a single message exchange all of the RPCs that have been enqueued into a batch. Of course, the return arguments for all of the enqueued RPCs are not set until the batch is executed. This facility is similar to the *asynchronous RPC* facility in the Mercury system [24]. A major advantage of this approach over the one described in the previous paragraph is that the stub generator needs to be provided with only the interface descriptions for operations on individual component RSM representatives. The resulting program is far smaller and less messy than the one that would result from the previous approach.

We wrote a package called *batch* to implement the RPC batching technique described in the previous paragraph. The package exports one data type, the *batch*, and the three batch operations described above. The package allows an existing server to be trivially modified to accept batched RPCs in addition to normal RPCs.

One performance disadvantage of the batching approach is that batched RPC arguments are marshaled twice: once by the BatchEnqueue operation, and again by the client stub for the BatchExecute function. This problem could be corrected by integrating the batching facility into the stub generator.

6.2.1.3. Other Packages

The representative server requires the services of a dynamic recoverable storage allocator to allocate and free object representatives and records contained therein. Initial experiments showed that the storage allocator distributed with the Camelot library was far too slow for use in our prototype. The storage allocator's poor performance stems primarily from the fact that it achieves concurrency by implementing simple operation logging on top of Camelot's value logging facilities. This requires that two top-level transactions be executed at a server for every recoverable malloc, one when the malloc occurs and one when the enclosing transaction family commits.

We implemented a fast recoverable storage allocator package, called *RFA*, short for *Recoverable Fixed Allocator*. RFA does not allow arbitrary size blocks to be allocated from a single storage pool. Instead, it allows the server to create multiple storage pools, each containing blocks of one fixed size. Instead of taking a size, RFA's malloc and free procedures take a *pool*. The fixed block size approach was taken for ease of implementation. RFA's malloc operation requires only one recoverable storage modify and one lock. The free operation requires two modifies and one lock. RFA is conservatively an order of magnitude faster than the allocator distributed with the Camelot library. The details of RFA's operation are outside the scope of this thesis.

The representative server's representative directory is simply a recoverable hash table containing a record for each representative at the server. We were able to implement it with a preexisting generic recoverable hash table package called *RHT*.

6.2.2. Version Numbers

Our prototype uses 32 bit version numbers. All of the primitive data types in the RDS library use optimistic timestamps for blind write operations. Optimistic timestamps are generated from the operating system's real-time clock. Ideally, the operating system should allow the RDS library to map the real-time clock into the client's address space. This would result in a clock access latency equal to the latency of a memory read. In principle, Mach provides this capability, but it did not function properly on the IBM RT-PC at the time of our experiments. Therefore, we were forced to rely on the *gettimeofday* system call.

Our prototype generates optimistic timestamps by subtracting a compile time constant from the current time and dividing the result down to millisecond precision. (The constant is set equal to the time value returned by *gettimeofday* at the time the program was written.) A simple computation shows that our version numbers wrap every fifty days. This would be unacceptable in a commercial implementation, hence we advise the use of 64 bit version numbers in such implementations (Section 4.1.1).

We do not view the cost of the *gettimeofday* system call as a fair cost in generating an optimistic timestamp. Therefore, we provided an alternate mode in the RDS library, wherein a counter is used to generate optimistic timestamps instead of the real-time clock. This mode avoids the cost of the system call, while guaranteeing that operations using optimistic timestamps will always succeed in one round if objects are accessed by only a single client. (The same guarantee is provided by optimistic timestamps generated from a real-time clock.) This mode is not suited for use when objects are accessed concurrently. We used the alternate mode for the experiments involving only a single client (Sections 7.2.5 and 7.2.6). In retrospect, this was unnecessary, as the cost of a *gettimeofday* call on the system used in the experiments is 0.2 ms, and the fastest operation measured in any of the experiments was 18 ms.

6.2.3. Primitive Data Types

6.2.3.1. Replicated Sparse Memories

The RDS library implementation of the RSM data structure is a fairly straightforward rendering of the protocols described in Chapter 2. All of the optimizations to the real predecessor algorithm described in Section 2.5.3 were implemented. The maximum number of entries returned in the first stage of the real neighbors determination is specified by a compile time constant, which we initially set to 8. We did not implement range operations (Section 4.2.1), low latency Erases (Section 4.1.3), or separate version numbers for frequently modified fields (Section 4.1.4) in our prototype.

The representative server uses *skip lists* [49] to represent RSM representatives. The skip list, invented by William Pugh, is a new data structure for representing directories (dictionaries). A skip list consists of a sequence of linked lists, each containing only a fraction p of the elements on the previous list in the sequence. The first list in the sequence contains all of the elements in the directory. The storage requirements for a skip list are comparable to those of a balanced tree. Like balanced trees, the average time to access an element in a skip list containing n elements is $O(\log n)$. Unlike balanced trees, no rebalancing is required to maintain this performance. Skip lists derive their good performance from the use of random numbers in the insertion procedure. We used skip lists in our prototype implementation because they are much easier to implement than balanced trees.

The skip lists that we used differ from Pugh's in that we added backward links to the linked list containing all of the elements in the skip list. This allows us to traverse the list backwards as well as forwards, which makes it possible to read the entries surrounding a given entry. This operation is required in the first phase of the real neighbor determination algorithm (Section 2.5.3). We used $p = 0.5$, the standard value for this parameter. In addition to the standard insert, lookup, and delete operations on skip lists,

we implemented the *range delete* operation, which is required for RepCoalesce, the representative operation used in the final stage of the Erase operation (Section 2.4.1).

Perhaps the most significant deficiency in our prototype is that our skip list implementation allows no concurrency; a single lock is used for the entire list. We felt that locking at the pointer level would be unduly expensive, but the design of an efficient locking algorithm for skip lists was outside the scope of this thesis. In retrospect, locking costs were such an insignificant fraction of operation costs that pointer locking would not have been prohibitive.

A related deficiency in our implementation is that our skip lists do not have the good paging properties associated with commercial B-Tree implementations [12]. This was not an issue in our experiments because the data sets were small enough to fit in primary memory. Modifying the skip list data structure to yield paging performance comparable to that of a B-Tree is an open problem, outside the scope of this thesis.

While the deficiencies described in the preceding paragraphs would be unacceptable in a commercial implementation of our architecture, they do little harm to our prototype. The design and implementation of a commercial quality B-Tree package is a very large undertaking, entirely orthogonal to the subject matter of this thesis. In practice, a commercial implementation of our architecture would likely use a pre-existing B-Tree package to represent RSM representatives. We view as an attractive feature of our architecture that it naturally inherits so much functionality from a readily available software package.

6.2.3.2. Hash Table RSMs

The RDS library uses a single implementation for both hash table RSMs and ordinary RSMs. The code was originally written for hash table RSMs and ran on normal RSMs without modification. The representative server has two sets of RSM representative functions, one for normal (skip list) RSM representatives and one for hash tables. The RPCs for the RSM representative operations check the representation type of the indicated representative and execute the appropriate function. In the parlance of object oriented programming, the RPC dispatches the message to the method appropriate to the class of the RSM representative object.

In addition to all of the optimizations implemented for ordinary RSMs, our hash table RSM implementation incorporates the optimistic two-stage protocol described in Section 4.1.5, which allows some Erase operations to complete in a single round. This optimization has a complication not found in our other optimistic two-stage protocols: if the optimistic assumption turns out to be false (i.e., the bucket contains other occupied addresses besides the one being erased), representatives that erroneously assumed that the

assumption was true will need the information they have overwritten in order to restore consistency.

In theory, it would be possible to restore consistency in the face of a failed optimistic two-stage protocol by aborting the subtransactions that performed erroneous modifications. However, the expense of doing aborts is such that the performance of this approach would be intolerable. As a rule of thumb, operations should never abort transactions during forward processing, except when it is necessary to break deadlocks. Therefore, each representative that believes that the optimistic assumption is true must save enough information to restore consistency if it turns out to be false.

While this information does not need to be stored recoverably, the use of volatile storage would pose difficulties in ensuring proper deallocation in the case of abort. Therefore, we store the information in a special field in the (recoverable) hash bucket data structure, which we call the *optimistic erase record*.

To keep the amount of information that must be stored to a minimum, representatives reject the optimistic assumption if they have any entries in the relevant hash bucket for addresses other that the one being erased. This policy ensures that the only data that representatives must record in the optimistic erase record are the version numbers associated with the gaps preceding and following the address being erased.

6.2.3.3. Array RSMs

Our array RSM implementation is entirely straightforward. The only feature of note is that the client is not permitted to specify an arbitrary DDS data type as the address space of an array RSM. The address spaces of array RSMs are restricted to ranges of integers: the size of the address space is specified when an array RSM is created, and the integers less than the size constitute the address space. There is no good reason for this restriction. The performance of the implementation would not change noticeably if we extended it to allow ranges drawn from arbitrary DDS data types to be used as address spaces.

6.2.3.4. MRSMs

Our MRSM implementation is somewhat limited, in that it allows only a single secondary index per MRSM. Both the primary and secondary indices are represented as hash tables. This implies that our implementation should only be used with secondary keys that are *nearly unique* if good performance is to result (Page 90).

On page 91, we described a dangling pointer problem associated with the MRSM Delete operation and presented several solutions. Our implementation incorporated the easiest solution, which consists of ignoring the dangling pointers.

The use of a batching facility to combine logical RPCs on the primary and secondary index RSMs had a strong effect on the shape of the MRSM code in the RDS library. While concurrent multistage operations on the two index RSMs were logically separate, the semantics of the Batch Enqueue operation imposed a program structure that grouped them together artificially. Each stage in complex MRSM operations was divided into two phases, one phase to enqueue the component logical RPCs and one phase to process all of the results. This resulted in complex procedures with names like *DelDoThirdRound*, *DelEnqThirdRound*, and *DelProcessThirdRoundResults*, whose functions cannot be concisely described, except by appealing to the notion of RPC batching. These procedures usually have excessive numbers of arguments, often as many as seventeen or eighteen.

While much of the logic for the MRSM implementation was taken from the hash table RSM implementation, programming the MRSM was time-consuming and tedious due to the code restructuring necessitated by our use of RPC batching. It is apparent that the use of RPC batching on concurrent optimistic protocols has fairly serious consequences with respect to program structure. However, the performance gains made possible by combining RPCs on component RSMs are essential to achieve reasonable performance from the MRSM data structure. In Section 6.3, we describe an alternate approach to RPC batching that performs almost as well as the approach we used, without imposing unnatural program structure.

6.2.4. Program Size

The sizes of the modules comprising our prototype implementation are shown in Table 6-1. The sizes of major modules are broken down into submodules. The line counts include comments but not blanks lines.

Several things should be borne in mind when examining the Table 6-1:

RPC stubs were generated with MIG [32], the RPC stub generator for the Mach operating system. MIG is somewhat lacking in its support for variable length arguments, allowing only one such argument to be passed in-line in each direction in each RPC. The generic nature of our representative implementations required extensive use of variable length arguments, which left us with no choice but to pack multiple logical arguments into a single MIG argument. This required a fair amount of argument packing and unpacking code.

The Camelot library has a similar restriction with respect to concurrent threads, which it inherits from the *cthreads* facility [13]: each thread executes a function that is permitted only a single argument. If multiple arguments are desired, they must be packed into a single *argument block* structure [7]. Our protocols make extensive use of concurrency, so this restriction caused a substantial increase in code size.

Component	Line Count
RDS Library	4566
RSM/Hash table RSM	1255
Array RSM	550
MRSM	2084
Shared Code	677
Replica Server	3090
RSM Representative	848
Hash Table RSM Representative	819
Array RSM Representative	201
MRSM Representative	1075
Shared Code	147
Utility Packages	1487
DDS	498
Batching Facility	635
RFA	354
System Total	9143

Table 6-1: Sizes of Replicated Data System Components

As mentioned previously, the RSM representative implementation is not entirely realistic: it lacks the paging and concurrency performance that would be required of a real implementation. A real implementation, however, would likely use a pre-existing B-Tree package, in which case it might well be smaller than our prototype.

Finally, our decision to nest each round of concurrent RPCs in a separate subtransaction caused an unnecessary increase in code size and complexity. This is discussed in more detail in Section 6.3.

6.2.5. Testing and Debugging

We subjected our data type implementations to a two stage testing process. In the first stage, we used an interactive driver program that allows the user to create an object, or look up a pre-existing object, and manually construct transactions that perform operations on the object. The driver program is similar in spirit and structure to the Camelot demo program called *jack* [9]. The driver program was used to get a data type off the ground. It allowed us to start testing a data type even before all of its operations were written, and to closely observe the effects of individual operations on an object. Objects created with the driver program used various address and value spaces, like personal names and phone numbers. When all of a data type's operations appeared to work under interactive use, we subjected the data type to more rigorous *torture testing*.

Torture testing was done with the aid of a *basher* program that allows the user to configure an object and perform a specified number of transactions, each containing a specified number of operations on the object. There are separate basher programs for RSMs and MRSMs. The details of this description correspond to the RSM version, but the MRSM version is similar. Operations are chosen at random from the operations that modify the object (Writes and Erases, for RSMs). The basher maintains a local, non-replicated table that tracks the RSM that is under test. Each time a transaction commits, all of its operations are recorded in the local table.

The address for each Write operation is selected at random from the address space, which consists of the integers less than some user-specified value. The value for each write operation is its sequence number in the run. If an error occurs, values generated in this fashion can be of some assistance in locating the operation that is responsible for the error. The address for each Erase operation is chosen randomly from the set of currently occupied addresses in the RSM, as indicated in the local table. Since operations are not recorded in the local table until a transaction commits, unoccupied locations are occasionally erased.

Before each run, the user specifies a quorum change interval and a consistency check interval. These intervals indicate the number of transactions that occur in between each occurrence of the designated event. A value of zero for either of these intervals indicates that the event will not occur during testing, though a consistency check is performed at the end of each run even if its consistency check interval is zero.

The consistency check is fairly thorough: it consists of looking up every address that has ever been occupied in the RSM and making sure that it has the correct value associated with it, or that it is unoccupied if it has been Erased more recently than it has been written. If the consistency check fails, it prints a message detailing the nature of the discrepancy and exits. For MRSMs, the consistency check looks up every primary *and secondary* key that has ever been used.

For torture testing, objects were configured to cause as many code paths as possible to be executed. For example, hash table RSM representatives were set to consist of just a few buckets. This ensured that many collisions would occur and delete list would grow large enough to drive the real predecessor algorithm into its second round occasionally. It would not have been as effective to set hash table RSM representative size to a single bucket; while this would have caused more collisions, it would not have tested the hash value generation process.

Torture tests runs were long, between 10,000 and 25,000 transactions, each containing ten operations. The quorum change interval was set to fifty transactions for all runs. The consistency check interval was initially set to ten transactions, to shake out simple bugs. Each time a consistency check failed, we attempted to locate the bug that caused the failure and fix it. Once a bug was located and fixed, the run was repeated. The pseudo-random number received the same seed for each run, so all runs attempted the same sequence of operations. Therefore, we knew that we had located the bug that was responsible for a consistency check failure if the test ran beyond the point where it had failed in the previous run. The effectiveness of this technique depended on the fact that there were no significant timing dependencies in our test runs.

The test-debug cycle was repeated until the entire run completed with no consistency check failures. As bugs were corrected and the data type implementation became more robust, we increased the consistency check interval up to a thousand transactions. This was necessary because consistency check transactions become extremely expensive as the data object is used. Even with a consistency check interval of a thousand transactions, consistency checks account for a large fraction of the total run time. Successful runs consumed roughly 4 to 6 hours of computer time, depending on the data type. All tests were conducted *locally* to keep run times reasonable: the basher and all three representative servers ran on the same machine. While this reduced the amount of underlying system code that was exercised, it has no effect on the execution paths through our replication system.

The fact that our data objects are persistent was very useful for debugging. The driver program used in the initial phase of our testing could be run on a data object that had just failed a consistency check. This allowed us to further investigate the condition of the now inconsistent object. For one particularly thorny bug in our MRSM implementation, it was necessary to examine the contents of MRSM representatives directly. Since the object was persistent, we were able to add an MRSM representative dump operation to the representative server, recompile the server, and perform the new operation, without redoing the run that corrupted the MRSM. A single application of this technique saved us many hours.

The repeatability of our test runs enabled us to use another powerful debugging technique, which we call the *surgical strike*. The RDS library and the representative servers both contained a fair number of debugging print statements. These statements were enabled during early phases of testing with the interactive driver. It made no sense to enable these print statements during torture tests because of the tremendous volume of output they would have generated. However, we modified the system so that debugging output could be turned on automatically each time an operation was performed on a designated address, and turned off when the operation finished. When a consistency check failure occurred, we could repeat the run with debugging output enabled only for operations on the address that caused the failure.

One bug in our RSM implementation was resistant to the debugging techniques described thus far. By the time the consistency check in the basher program failed, the skip lists representing the RSM *representative* were so thoroughly corrupted that it was impossible to determine which operation was originally responsible for their fall from grace. In order to locate this bug, we had to outfit the *representative server* with a consistency check procedure for the skip list data structure, and call this procedure after every representative operation that modified the skip list. This local consistency check in the server quickly pinpointed the source of the trouble, which lay in the skip list range delete operation.

6.3. Conclusions

Our prototype demonstrates that it is feasible to implement the architecture described in Section 6.1 and the replication protocols described in Chapters 2 - 5. We wrote several applications on top of our prototype, including the *interactive driver* and *basher* described in Section 6.2.5, and the *performance measurement application* described in Section 7.1. Experience with these programs strengthens our belief that the architecture is easy to use and reasonably flexible. Its ease of use stems primarily from the fact that the sole interface to the system is the RDS library, and in particular, that new data objects are created by simply calling a function in this library. The flexibility of the architecture is due primarily to the generic nature of the data objects it exports.

Our debugging and testing procedures were effective. No additional bugs in our prototype surfaced during the performance measurement process described in Chapter 7.

The process of implementing the architecture was reasonably straightforward. However, we made two design decisions that made this process more difficult than it had to be, and increased the complexity of the resulting software. The first of these decisions concerned the use of subtransactions. The second concerned the use of RPC batching. Below, we discuss these decisions and the changes that should be made if our architecture is re-implemented.

For performance reasons, we nested each round of every replicated operation in a subtransaction, rather than nesting individual RPCs (Section 6.2.1). In retrospect, we should have taken this line of reasoning one step further, by nesting each replicated operation in a single subtransaction. This would have had two major benefits. First of all, it would have greatly reduced the amount and complexity of the error handling code in the RDS library. In our prototype, we had separate error handling procedures for each round of every operation. Some of this code was quite complex. If we had nested entire operations, a single error handling procedure would have sufficed for all operations. This procedure would be extremely simple: it would reconfigure the quorum and retry the entire operation.

The second benefit of nesting entire replicated operations in a single subtransaction is increased performance. The cost of recovering from an RPC failure would increase slightly, as a failure would necessitate repeating an entire replicated operation, instead of just the round containing the failure. But failures are rare; in the normal case, the cost of the second and successive rounds of an operation would be reduced by the cost associated with a nested transaction. Under Camelot, this cost is *extremely* significant (Section 7.2.8). Note that this change would only improve the performance of multi-round operations; single round operations would be unaffected.

To secure good performance from our MRSM implementation, we used an RPC batching facility to combine multiple RPCs on the same server (Section 6.2.1.2). However, this resulted in a fairly convoluted program (Section 6.2.3.4). The problem results from the use of RPC batching to merge complex multistage operations that are performed concurrently. To remedy this problem, we propose a new RPC batching technique, which we call *auto-batching*.

The operations performed on the component RSMs of an MRSM are independent of one another. The natural idiom for expressing concurrent execution of independent operations is *threads*. However, the direct use of RPC batching makes the use of this idiom impossible. Ideally, we would like each thread to perform its operations oblivious to the existence of the other threads, with RPCs being combined to the maximum extent possible. This is precisely what auto-batching allows us to do.

There are four operations associated with the *auto-batch* type. Each auto-batch must be initialized once by calling *AutoBatchInitialize*. This operation takes an auto-batch and a server, which it associates with the auto-batch. A thread can *Join* an auto-batch, *Quit* an auto-batch, and perform *AutoBatchRPCs*.

The RDS library maintains one auto-batch for each representative server. Each thread performing an independent multistage operation initially joins the library's auto-batch for

each server involved in the operation.[10] Whenever a thread wants to call a server, it makes an AutoBatchRPC, which differs from a normal RPC only in that it takes one additional parameter, the auto-batch for the relevant server. The beauty of the auto-batching approach is that the AutoBatchRPC has essentially the same semantics as an ordinary RPC; the only difference is that AutoBatchRPC automatically waits for all other threads using the same auto-batch to contribute an AutoBatchRPC. When the last thread using the auto-batch does an AutoBatchRPC, the auto-batch is executed in a single RPC, and all threads using the auto-batch automatically continue execution. When each thread finishes its multistage operation, it calls AutoBatchQuit for all auto-batches used by the operation.

It is fairly straightforward to implement an auto-batches on top of ordinary batches, using a *condition variable* [13] to synchronize threads, and two integers. One integer is used to keep track of the number of threads that are currently *members* of the auto-batch (i.e. they have joined the auto-batch but not yet quit it). The other integer is used to keep track of the number of threads that have already contributed an AutoBatchRPC to the next real RPC.

The auto-batching approach would cause a slight decrease in the performance of our MRSM operations due to the use of additional threads, but we do not believe that the effect would be significant.

Auto-batching can be used to improve our architecture in another, fairly significant, way. If *all operations on all primitive types* are implemented using auto-batches, concurrent calls to operations on objects exported by our system will ‘‘carpool’’ automatically, to the maximum extent possible. This means that derived types built from multiple replicated objects can be easily and efficiently implemented on top of primitive types. In our prototype, derived types had to be implemented directly on top of representative operations using the RPC batching facility if they were to display reasonable performance. This essentially excluded end users from implementing efficient derived data types.

[10]For efficiency, the Join operation should take a list of auto-batches and join all of them. For symmetry, the Quit operation should do likewise, though it is not as important from an efficiency standpoint.

Chapter 7
The Performance of Our Architecture

We performed experiments to evaluate various aspects of the performance of the prototype system described in Chapter 6. We were quite successful in explaining its performance in terms of the performance of underlying transaction system primitives. While the performance of the prototype is limited by that of the underlying system software and hardware, it clearly demonstrates the practicality of our approach.

The remainder of this chapter is organized as follows. Section 7.1 describes our experimental setup. Section 7.2 presents basic timings of the operations on our data objects when accessed by a single client, and a comparison of these timings to predictions based on the performance of underlying primitives. Section 7.3 presents timings for an array RSM under concurrent use. Section 7.4 presents an experiment that investigates the performance of optimistic timestamps in the presence of *hot spots*. Section 7.5 discusses failure recovery performance and Section 7.6 briefly evaluates the overall performance of our prototype system and of our architecture.

7.1. Experimental Setup

Our experiments were done with a performance measurement program that allows the user to configure and create a data object, perform an initial run consisting solely of Write operations, and subject the object to one or more timing runs. Each timing run consists of a number of timed *sets*, each of which consists of a number of transactions. Each transaction performs a number of operations. All operations in a set are of the same type. For each run, the user selects an *operation mix*, a quorum change interval, the number of sets in the run, the number of transactions in a set and the number of operations in a transaction. The operation mix consists of a sequence of integers describing the relative frequencies with which the operations supported by the data type are to be performed. After each set ends, the program prints the average time per transaction for the set. After each run ends, the program prints the average time per transaction observed over the duration of the run, for each operation in the mix. The program also prints the communication cost histograms gathered by the RDS library during the run (Section 6.2.1).

Our replication system presents us with a large number of parameters whose values affect its performance. To keep the number of experiments from getting out of hand, we held many parameters constant for all experiments. All objects consisted of 3 representatives. All used read and write quorum sizes of 2. All used 32 bit integer address and value spaces. Addresses for operations were chosen randomly from the integers between 0 and $2^{16} - 1$. Hash table RSM representatives had 1000 buckets. In each experiment, 500 Write operations were performed initially, yielding 500 occupied addresses. The operation mixes for all runs were uniform: all of the operations supported by the object were performed with equal likelihood. The quorum was changed every 1000 operations, cycling through the three possible choices.

All experiments were performed on IBM-RT/PC APC workstations [57], running Camelot 1.0 (Version 83) on top of Mach 2.5 (Version CS7d). The workstations are rated at approximately 2.5 VAX MIPS. Local experiments were performed on a workstation with 16 megabytes of main memory and distributed experiments were performed on workstations with 12 megabytes each. The workstations used in the distributed experiments are connected via a large, complex local area network. The relevant portions of this network are coarsely illustrated in Figure 7-1. The network segments shown in this figure have many other machines attached to them besides those used our experiments.

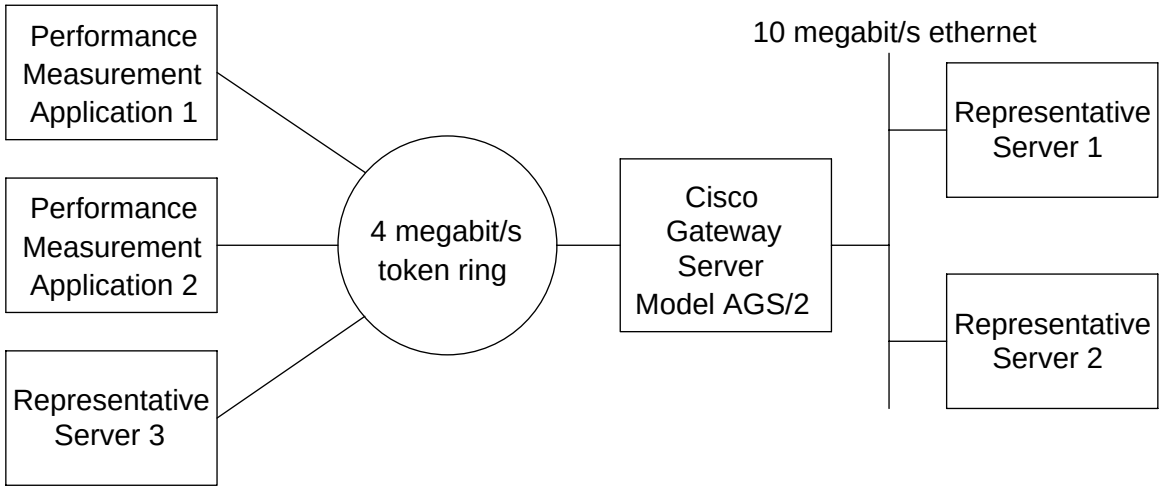

Figure 7-1: Network for Distributed Experiments

In Figure 7-1, we see that both performance measurement applications were on the token ring, while two of the three servers were on the Ethernet. Therefore, each read or write quorum contained at least one server on the token ring; at least one RPC in every round of two parallel RPCs had to cross the gateway server. The latency associated with a round is determined by the slowest RPC in the round, so the communication cost of a two-server round in our experiments was roughly independent of which quorum was chosen for the round. This was checked experimentally and found to be correct. The

latency for a Camelot RPC from a client on one side of the gateway to a server on the other was measured to be 27 ms. The latency for a round-trip UDP datagram [48] between the same pair of machines was measured to be 10 ms.

For local experiments, the workstation was *not* running in single user mode. While no extraneous user processes were running, no attempt was made to disable system demons or limit their activity. For distributed experiments, extraneous user processes, even large ones like display managers and text editors, were allowed to run on the workstations hosting the clients and servers. However, no one was actively using these machines for any other purpose while the experiments were in progress. All distributed tests were run late at night, when the network was lightly loaded.

7.2. Basic Timings

For each of the primitive data types supported by our prototype, we performed local and distributed timings. In these experiments, a single server performed operations on a single replicated object.

7.2.1. Methodology

We used the *primitive analysis* methodology of Spector and Daniels [55]. This methodology treats a set of operations and a set of primitives used to implement them. The idea behind the methodology is to compare the measured performance of the operations with predictions based on the measured performance of the primitives. If the difference between the measured and predicted performance is small, it can be taken as evidence in favor of two things: the experimenter's knowledge of the decomposition of the operations under study in terms of the primitives is correct; and the primitives represent the bulk of the work required to perform the operations. Broadly, this indicates that the experimenter understands the operation of the system, and that the measured performance of the operations under study is an accurate representation of their performance on the test system. The decomposition used to make the predictions can then be used with some confidence to predict the performance that would be observed if the operations were implemented on a different platform, given only the performance of the primitives on the new platform.

The performance measure used in Spector and Daniels's methodology is *incremental latency*, defined as the time added to the latency of a transaction when a single operation of a given type is added to the work performed by the transaction.[11] The incremental

[11]Spector and Daniels use a more general definition for incremental latency, making the technique applicable to other units of work besides transactions.

latency of a given operation is measured as follows. The average latency to execute a transaction that performs the operation once is measured, as is the average latency to execute a transaction that performs the operation $n + 1$ times. The difference between these latencies is computed and divided by n to yield the incremental latency for the operation. The choice of an appropriate value for n depends on the details of the operation whose incremental latency is being measured.

To apply Spector and Daniels's methodology, the incremental latencies of the operations under study are measured, as are those of the primitives, unless they are already known. The decomposition of each operation O_i into primitives is characterized by a *primitive usage vector* $\vec{u}_i$, which consists of the average number of times that each of the primitives is executed in the process of executing O_i. The *primitive cost vector* $\vec{c}$ consists of the incremental latency of each primitive operation. The predicted incremental latency of O_i is merely $\vec{u}_i \cdot \vec{c}$.

7.2.2. Primitives and Their Costs

We used three primitives in the analysis of the operations on our data types. The incremental latencies of these operations were measured with a simple Camelot client and server written for the purpose.

The first primitive used was the *Round*, a nested transaction containing a parallel RPC to two servers. This is the computational structure associated with a round in any of our replicated protocols as implemented in our prototype system. The round represents the communication costs associated with our protocols. It would be more natural to have the nested transaction and the parallel RPCs be two separate primitives. The reason we do not do this is discussed at length in Sections 7.2.9. The RPCs in the rounds that we timed had a single integer argument and no return arguments.

There are two types of Rounds, local and distributed, corresponding to the rounds that occur in our local and distributed experiments. In the local case, the client and both servers reside on the same machine. In the distributed case, all three processes reside on different machines. In measuring the cost of a distributed round, we made sure that the client and servers were situated on machines that would host a performance measurement application and representative servers, respectively, in our experiments. This ensured that the latency measured for the distributed Round primitive would correspond roughly to the latency incurred for a distributed round in the experiments.

The second primitive we used was the *Modify*, a recoverable storage modify operation. The amount of recoverable data affected by each modify operation in our experiments varied, though it was never very large. The Modify operations that we timed affected four bytes.

The third primitive we used was the *Lock*, an operation that secures a lock on behalf of a transaction. We measured two latencies for this primitive, one corresponding to a completely uncontested lock, and one corresponding to a lock held by fifty transactions in the same family as the locking transaction. The latter latency more accurately reflected the latency observed in experiments on data types that used the recoverable storage allocator (RSMs, hash table RSMs and MRSMs). Note that this effect is an experimental artifact, a subtle interaction of our experimental procedure with our recoverable storage allocator and the Camelot library's lock manager.

The primitives and their measured incremental latencies are illustrated in Table 7-1.

Primitive	**Latency (ms)**
Round (local)	17.7
Round (distributed)	43.0
Modify	1.0
Lock (uncontested)	0.2
Lock (inherited)	0.4

Table 7-1: Incremental Latencies of Primitive Operations

One more incremental latency should be noted, that of the *Non-replicated Round* (i.e., a subtransaction containing an RPC). While this figure is not used in our analysis, it provides a useful baseline for interpreting the results of the analysis. The measured value of this latency in the distributed case is 38 ms.

7.2.3. Primitive Usage

Decomposing the operations on our replicated data objects into the primitives is complicated by the complexity of our algorithms, and in particular by our use of optimistic two-stage protocols. The code executed by a representative server when performing certain complex representative operations is very much a function of the contents of the representative. For example, the RepCoalesce operation must create entries for the real predecessor and successor of the address that is being erased only if it does not already contain these entries. Because we use of optimistic two-stage protocols, some stages of some operations will be skipped on occasion. The primitive costs associated with an optional stage must be multiplied by the probability of executing the stage, but our protocols are complex enough that it is not always practical to calculate the probability of executing a given stage.

The approach we took to dealing with representative operations whose use of the primitives varies from call to call was simple, if slightly crude. We examined the code,

and in cases where we had some intuition as to the probability of a primitive being executed, we assumed this probability. In all other cases, we assumed that the program took either branch of each *if* statement with equal probability. From these probabilities, we were able to calculate an approximate decomposition of complex representative operations into Update and Lock primitives.

The approach we took to performing the decomposition of our operations that do not run in a fixed number of rounds was to do a rough estimate as described in the previous paragraph *for each possible length of the operation in rounds*. Using the terminology introduced above, we computed a separate $\vec{u}_i$ vector for each running length. Then we used the actual relative frequency of the different running lengths of the operation observed in the experiments to compute a weighted average of the $\vec{u}_i$ values. This weighted average was used to compute the predicted cost of the operation.

The technique described in the previous paragraph might sound somewhat suspect, in that we are using data gathered in an experiment to compute ‘‘predictions’’ for the performance measured in the experiment. Several things should be kept in mind, however. For most of our optimistic protocols, we had strong intuitions about how many stages would actually be executed. For instance, we knew that the Erase operation in the RSM would almost always require two rounds. For such operations, we did not really use the technique. If the observed relative frequency of running lengths for such an operation had not matched our intuition, we would have investigated the discrepancy. For some optimistic protocols, like the one round delete for the hash table RSM, our understanding of the efficacy of the optimization is based almost solely on empirical data gleaned from our experiments. Therefore, it seems fair to use the relevant experimental data in the decompositions that are supposed to represent our understanding of the system. Note that there are only three operations whose primitive usage vector relied on histogram data to a significant degree among the fifteen operations we studied: the hash table RSM Erase operation and the MRSM Delete and Modify2 operations.

The primitive usage estimates are shown in Table 7-2. These estimates combine the estimates we made by studying the source code with the communication cost histogram data gathered in our experiments, as described above. Histograms from local and distributed experiments were combined to make this table. As expected, histograms from corresponding local and distributed experiments were nearly identical. Note that the rows in Table 7-2 correspond to primitive usage vectors ($\vec{u}_i$).

Operation	Rounds	Modifies	Locks
RSM			
Read	1	0	1
Write	1	5	2
Erase	2	8	2
Hash Table RSM			
Read	1	0	1
Write	1	3	1
Erase	1.6	4.5	1.6
Array RSM			
Read	1	0	1
Write	1	1	1
Erase	1	1	1
MRSM			
Lookup	1	0	1
Lookup2	1	0	2
Insert	1	8	5
Delete	2.9	10.9	8.4
ModifyData	2	4.7	4
Modify2	2.8	11.7	10.1

Table 7-2: Estimated Primitive Operation Counts for Operations on Replicated Objects

7.2.4. Experimental Details

Our timing experiments consisted of three phases. In the first phase, we performed 200 sets of 11 transactions, each transaction containing 5 operations, for a total of 11,000 operations. The purpose of this phase was to allow the data structure to come to equilibrium; we did not use the data gathered in this phase. The first phase was omitted for the array RSM, whose performance does not depend on the contents of the representatives. The second phase was identical to the first phase, except that the data was used to compute the average latency of a transaction containing 11 operations, for

each operation supported by the data type. The final phase consisted of 200 sets of 10 transactions, each containing 1 operation. The results of the second and third phases were used to compute the incremental latency of the operations supported by the data type. Latencies were computed to the nearest millisecond. The histogram data gathered in the second and third phases was used as described in Section 7.2.3 to compute the decompositions of the operations into primitives.

7.2.5. Local Performance

Using the methodology described above, we measured the local performance of our prototype. Note that these experiments are not at all realistic. Our replication protocols are by their nature distributed: they depend on distribution for their high availability. However, these experiments were very useful in testing our understanding of the system's performance.

Latencies measured in distributed experiments are typically dominated by communication times. In addition to being large, distributed communication times tend to be fairly dispersive, especially in complex networks like the one on which our experiments were performed. Even if our understanding of the other components of the latency besides communication were completely flawed, it would probably not be apparent from the results of distributed experiments alone.

On our system, the local round cost is less that half of the distributed cost. Furthermore, operations on multiple representative servers do not parallelize in the local case. Therefore, the latency of an operation absorbs the latencies for Modify and Lock operations on *both* representatives in each quorum, rather only one, as in the distributed case. Combining these effects, our local experiments magnify the importance of the update and locking components of operational latency by a factor of four relative to the communication component. Also, local RPC times are quite consistent. If our understanding of the number of Modifies or Locks performed by an operation were wrong, it would be much more likely to come out in the local experiments than in the distributed ones.

The results of our local experiment on the RSM data type are shown in Figure 7-2. The word *time* in the caption of Figure 7-2 refers to incremental latency. We will adopt this usage for the remainder of Section 7.2. The bars illustrating the predicted cost for operations are subdivided into three parts, representing the portions of the latency due to communication (Rounds), update (Modifies) and locking (Locks). Note that the Read operation has no update cost associated with it. The results of our local experiments on hash table RSMs, array RSMs and MRSMs are shown in Figures 7-3, 7-4 and 7-5. The same scale is used in the graphs for all three RSM implementations to facilitate comparison. The results of all of our local experiments are summarized in Table 7-6.

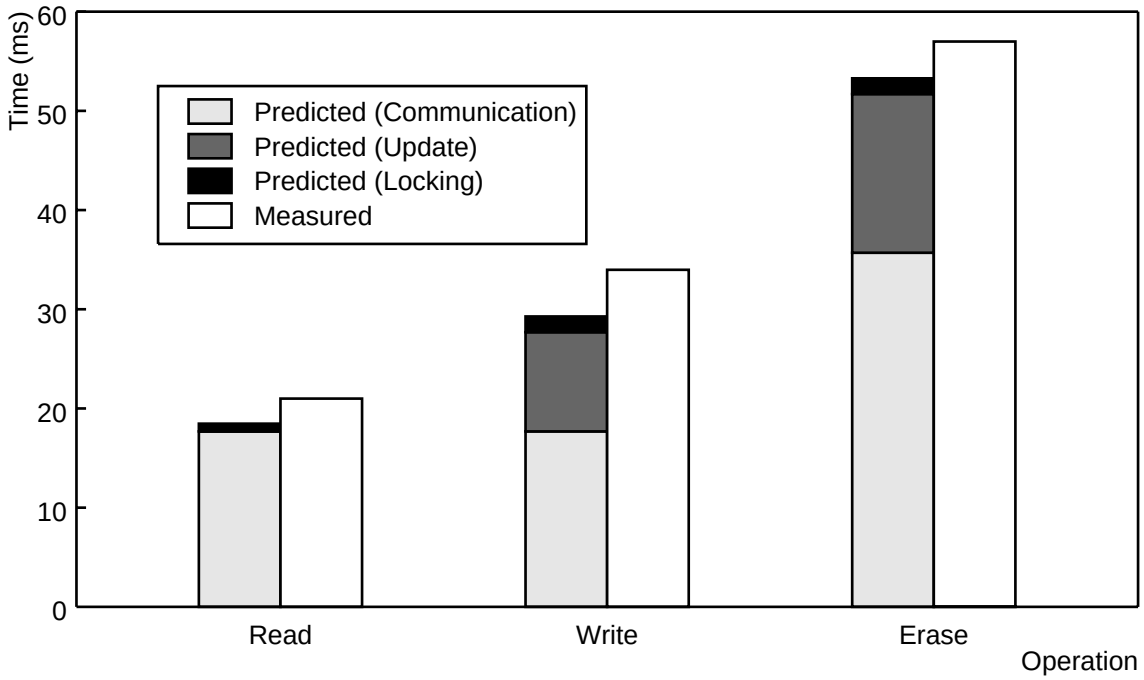

Figure 7-2: Predicted and Measured Times for Local RSM Operations

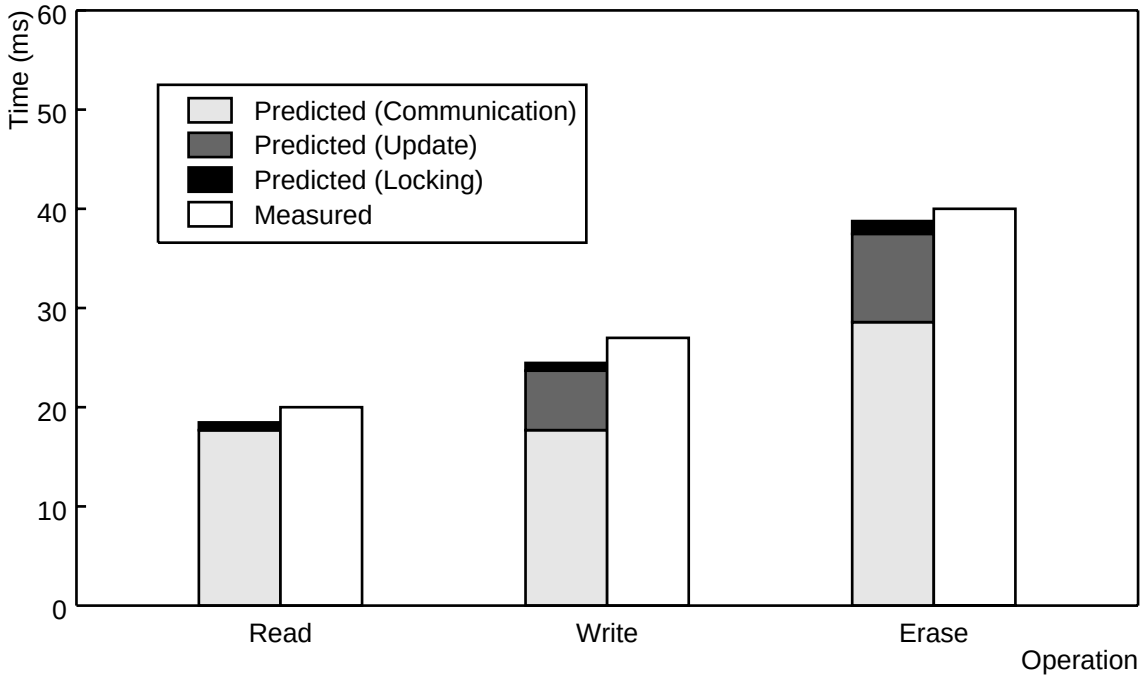

Figure 7-3: Predicted and Measured Times for Local Hash Table RSM Operations

On the whole, the level of agreement between measured and predicted times observed in our local experiments is quite satisfactory. For all operations on the three RSM implementations, with the exception of the Write operation on the normal RSM, the measured time is within 2 ms of the predicted time. This is easily accounted for by

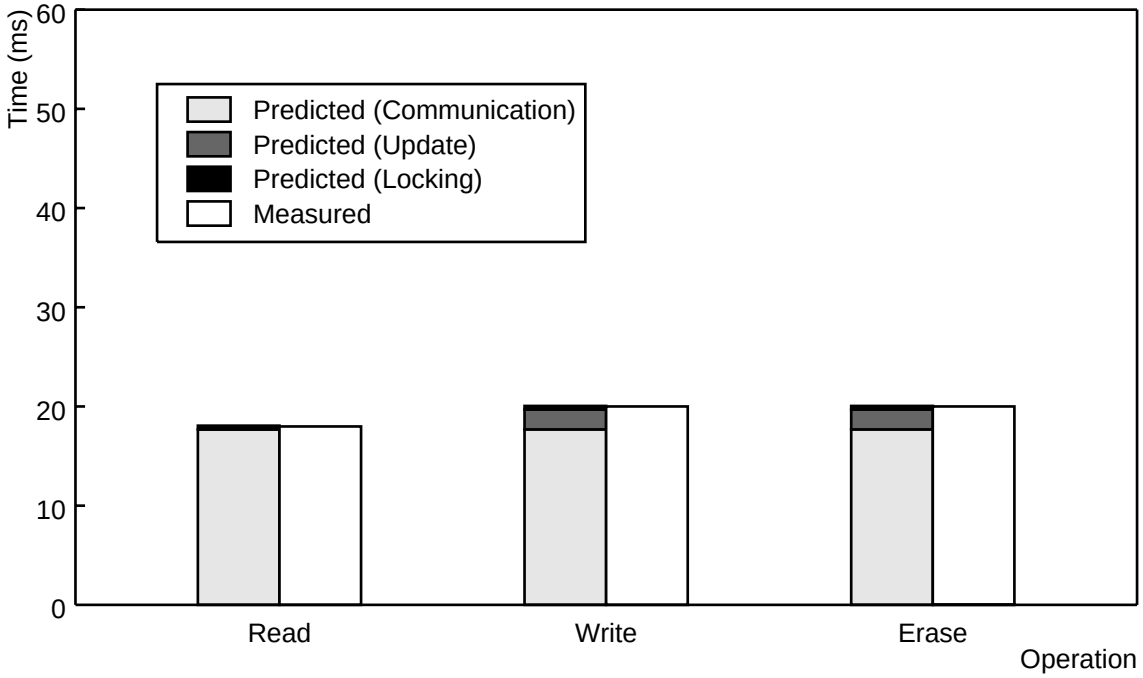

Figure 7-4: Predicted and Measured Times for Local Array RSM Operations

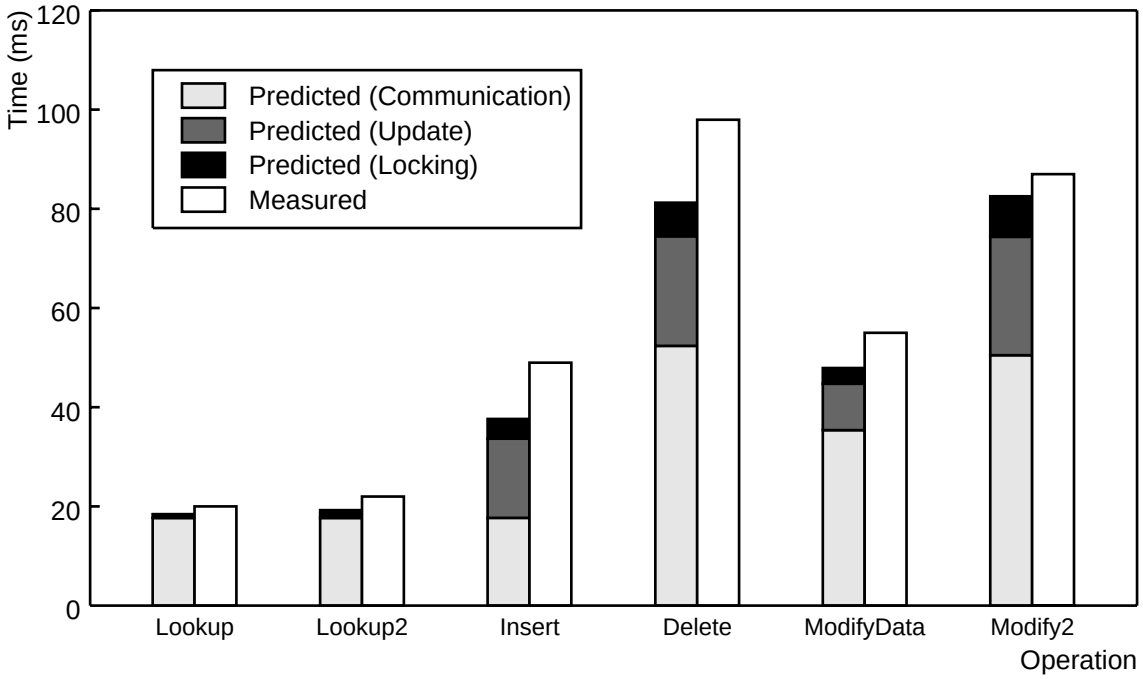

Figure 7-5: Predicted and Measured Times for Local MRSM Operations

computation costs and experimental error. Besides the computation inherent in our protocols, significant sources of computation not accounted for in our predictions include marshaling of arguments in RPCs and hashing and comparison of addresses by the DDS package. Significant sources of experimental error include measurement error and background computation on the test machine. The 5 ms disparity observed in the RSM

Operation	Predicted Time (ms)	Measured Time (ms)	Difference (ms)	Percent Difference
RSM				
Read	19	21	2	11
Write	29	34	5	17
Erase	53	57	4	8
Hash Table RSM				
Read	19	20	1	5
Write	25	27	2	8
Erase	39	40	1	3
Array RSM				
Read	18	18	0	0
Write	20	20	0	0
Erase	20	20	0	0
MRSM				
Lookup	19	20	1	5
Lookup2	19	22	3	16
Insert	38	49	11	29
Delete	81	98	17	21
ModifyData	48	55	7	15
Modify2	82	87	5	6

Figure 7-6: Predicted and Measured Times for Operations on Local Replicated Objects

Write operation (Figure 7-2) is somewhat mysterious: 5 ms represents a fair amount of computation. This disparity merits further investigation, but we did not have the time to pursue it.

The level of agreement observed between measurements and predictions is inversely related to the complexity of the data type. The more complex the implementation, the more computation is performed, and the less accurate our decomposition of its operations into primitives. For the array RSM operations, the predictions are exact within the precision allowed by our measurements. This is not all that surprising, when one

considers that the array RSM operations are very similar to the primitives whose observed latencies were used to make the predictions. The hash table RSM implementation is substantially more complex than the array RSM, and the average of the percent differences between measured and predicted latencies for its operations is 5%. The RSM implementation is substantially more complex than the hash table RSM, and we have less confidence in our decomposition of its operations into primitives due to the complexity of the skip list operations. The average difference between measurements and predictions for RSM operations is 11%.

The MRSM implementation is more complex than any of the RSM implementations. The average difference between measurements and predictions for MRSM operations is 14%. Unlike the RSM implementations, the MRSM implementation incurs additional RPC marshaling expenses due to the use of the RPC batching facility (Section 6.2.1.2). We examined the operating system's virtual memory statistics during some MRSM runs and found that the system was doing a nontrivial amount of paging. These additional sources of latency amply account for the 14% discrepancy.

7.2.6. Distributed Performance

In our distributed experiments, the performance measurement applications and each server ran on a separate machine (Figure 7-1), as they would in a practical implementation of our architecture. The network that connected the workstations used in our distributed experiments serves the communication needs of the Carnegie-Mellon University computing community. We ran our experiments at night to minimize externally generated network load, but backups, software distribution, and other people's experiments still caused high loads from time to time. In a commercial application of our architecture, we anticipate that a dedicated network with sufficient bandwidth to assure reasonable communication performance would be used. Therefore, we felt no compunctions about discarding runs wherein latencies were clearly increased by external network traffic. Note, however, that we never threw away individual *sets*: all of our results come from consecutive runs of 11,000 operations.

The results of our distributed experiments are shown in Figures 7-7, 7-8, 7-9 and 7-10. The format of these graphs is identical to the format used in the graphs illustrating local performance. Again, the same scale is used in the graphs for all three RSM implementations. The results of our distributed experiments are summarized in Table 7-6.

As in the local case, the level of agreement between measured and predicted times observed in our distributed experiments is quite satisfactory. The interpretation of the distributed results is fairly different from that of the local results. It can be seen in

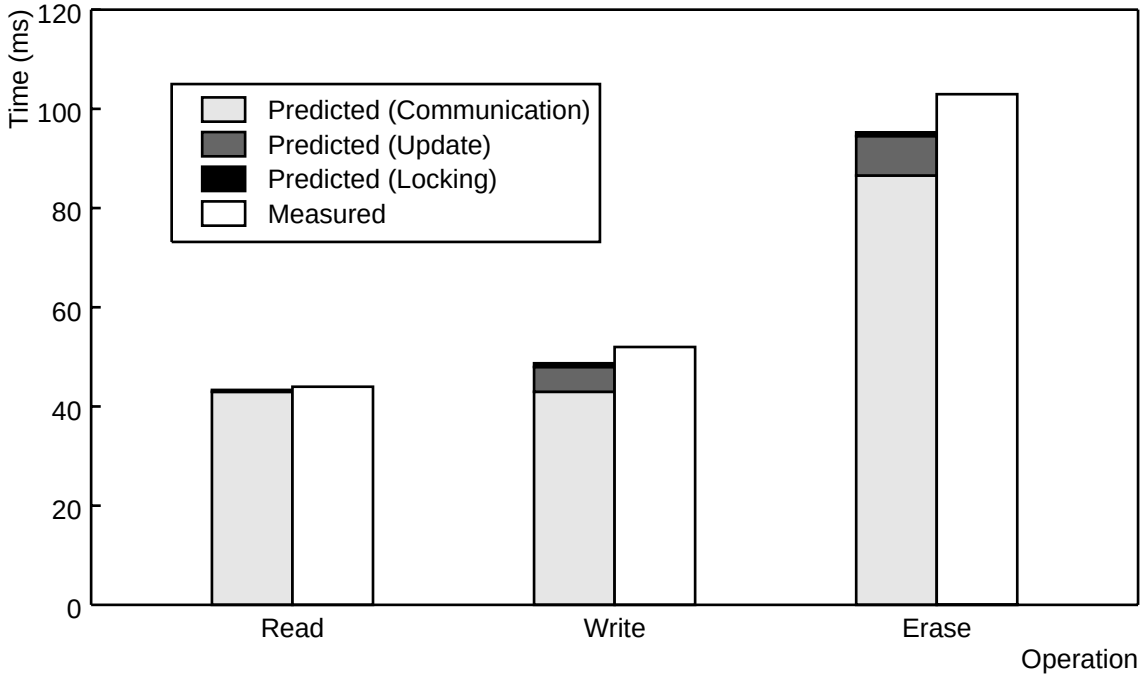

Figure 7-7: Predicted and Measured Times for Distributed RSM Operations

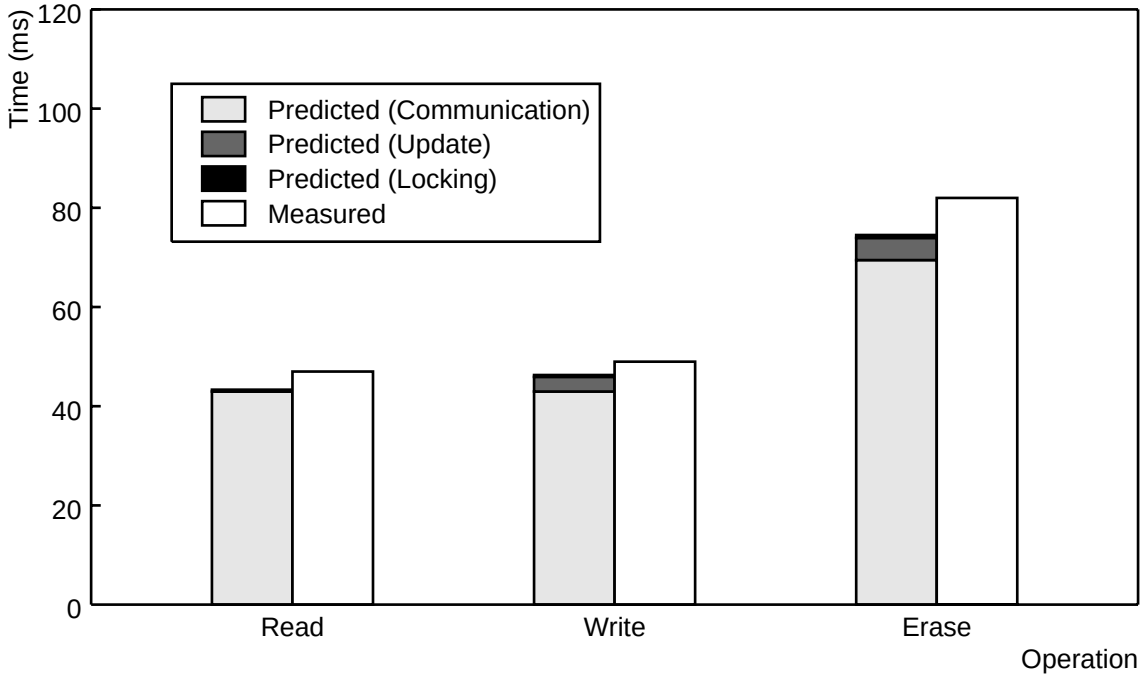

Figure 7-8: Predicted and Measured Times for Distributed Hash Table RSM Operations

Figures 7-7, 7-8, 7-9 and 7-10 our distributed times really are dominated by communication. Because of our methodology, the component in our primitive cost vectors representing communication cost is always exact. The accuracy of our prediction of the cost component due to communication, then, is determined by how closely the average cost of an RPC in the experimental run matches the cost assigned to the RPC

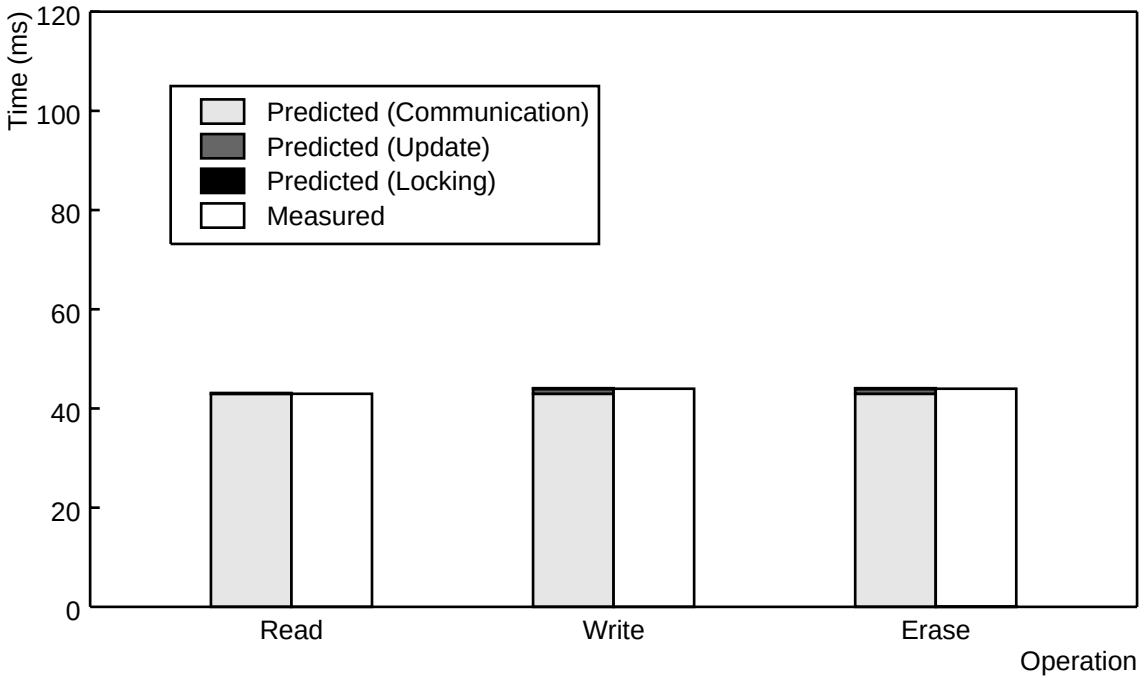

Figure 7-9: Predicted and Measured Times for Distributed Array RSM Operations

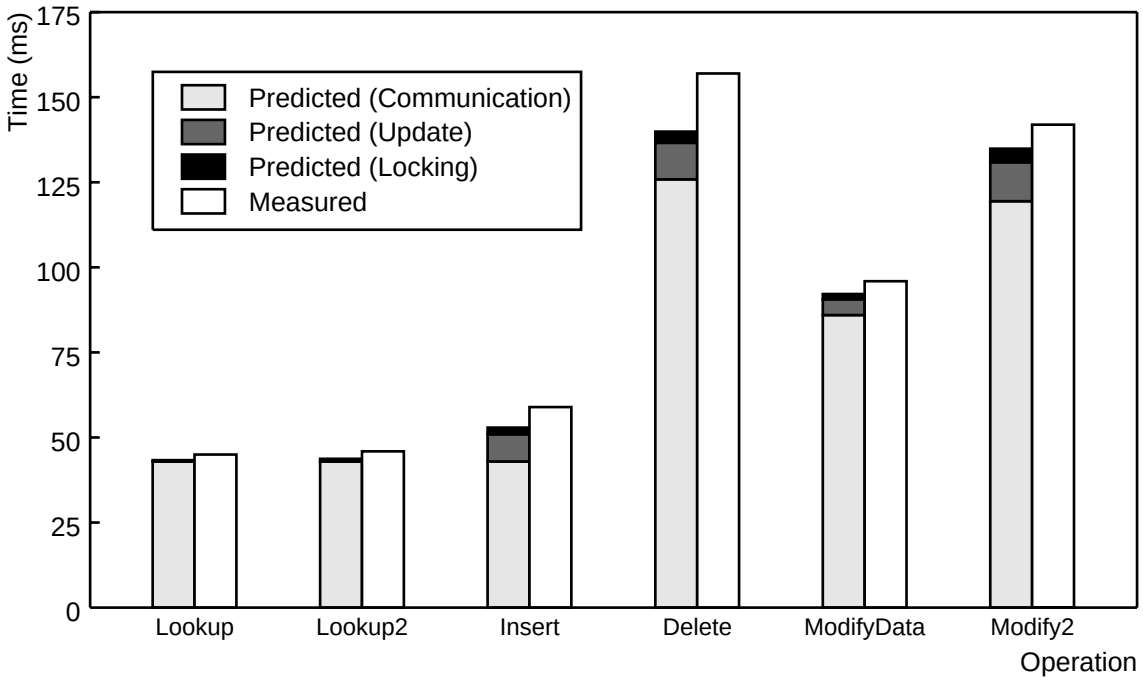

Figure 7-10: Predicted and Measured Times for Distributed MRSM Operations

primitive. In essence, the accuracy of our predictions for distributed experiments is primarily a function of how well-behaved the network was during the run. This is evidenced by the fact that the average discrepancies for Hash Table RSMs, RSMs, and MRSMs are 8%, 5%, and 7%; these figures bear no relation to the relative complexities of the data types.

Operation	Predicted Time (ms)	Measured Time (ms)	Difference (ms)	Percent Difference
RSM				
Read	43	44	1	2
Write	49	52	3	6
Erase	96	103	7	7
Hash Table RSM				
Read	43	47	4	9
Write	46	49	3	7
Erase	75	82	7	9
Array RSM				
Read	43	43	0	0
Write	44	44	0	0
Erase	44	44	0	0
MRSM				
Lookup	43	45	2	5
Lookup2	44	46	2	5
Insert	53	59	6	11
Delete	140	157	17	12
ModifyData	92	96	4	4
Modify2	135	142	7	5

Table 7-3: Predicted and Measured Times for Distributed Replicated Objects

Our results for array RSM implementation deserve special note. Once again, the predicted values exactly match the measured values, to within the precision of our measurements. This is somewhat artificial. While the extreme similarity of the operations to the primitives is still partially responsible for this result, it is caused in part by the fact that we knew what to look for in selecting a run to analyze. By the nature of the array RSM data type, the cost of executing each operation should not vary from instance to instance. Therefore, any significant variation in the times for sets of the same operation was assumed to indicate a network anomaly and the run was discarded.

Because the runs were so fast compared to those on other data types, it was much more likely that the network remained free of anomalies for the duration of a run.

Our results delineate a clear cost-performance tradeoff among the three RSM implementations. All array RSM operations cost approximately 43 ms (the cost of a Round), but array RSMs only work for small address spaces, and do not support efficient range or navigation operations. The Read and Write costs are similar for all three implementations, but the Erase costs vary substantially. The hash table RSM Erase operation costs an additional 86% beyond that of the array RSM. In exchange, the hash table RSM allows the use of infinite address spaces. The normal RSM Erase operation costs a further 26% beyond the cost of the hash table RSM Erase operation, but the normal RSM supports efficient range and navigation operations.

The cost of the Erase operations for the hash table RSM and the normal RSM, as well as the cost of all other multi-round operations, would be reduced substantially if we had nested each operation in a subtransaction instead of nesting each round (Section 6.3). The primitive analysis methodology allows us to quantify this. For each operation requiring more than one round, the cost savings is the number of rounds in excess of one multiplied by the difference in cost between a parallel RPC and a round, which is 12 ms. From table 7-2, we see that the savings would be 12 ms for the RSM Erase operation and 7 ms for the hash table RSM Erase operation. Thus, the hash table RSM Erase operation would have cost only 59% beyond the cost of the array RSM Erase operation if we had made the correct design decision with regard to subtransaction usage.

7.2.7. Histogram Data

Several pieces of data from the histograms generated by the performance measurement application deserve to be reported directly. The RSM Write operation required three rounds 2% of the time. This figure is highly dependent on a system constant, the number of entries returned in the first stage of the real neighbors determination. This constant was set to 8 in all of our experiments. Preliminary studies indicate that if this constant were increased to 10, the third round would only be necessary approximately 0.2% of the time.

The hash table RSM Write operation completed in a single round 38% of the time. This indicates that the relevant optimistic two stage protocol (Section 6.2.3.2) was well worth implementing. The third round was virtually never required for this operation, which indicates that the value of 8 for the system constant referred to in the previous paragraph is ample for hash table RSMs. It is possible that a lower value would suffice.

The MRSM uses the same optimistic two stage protocol referred to in the previous paragraph in both of its component RSMs. This occasionally allows the MRSM Delete

and Modify2 operations to complete in two rounds. The Delete operation completed in two rounds 6% of the time and the Modify2 operation completed in two rounds 18% of the time. The 6% figure is right on the border in terms of justifying the use of the optimistic protocol, while the 18% figure clearly justifies it, though the savings are modest. The Delete and Modify2 operations virtually never required their fourth rounds, which indicates that the value of 8 for the system constant referred to above is ample for MRSMs as well.

7.2.8. Performing a Primitive Analysis

The level of agreement between predictions and measurements in all of the experiments described above is reasonable. The graphs are pretty and the explanations are plausible. But these results do not tell the whole story. In our initial runs, the predicted and measured times were not even close. In fact, they showed little relation to one another. The process of bringing them into accord is the process of coming to understand the performance of the system. In this process, both the system and the predictions are modified. When a reasonable level of agreement is achieved, the process is deemed complete. We briefly describe this process below in the hopes that it will help others who seek to understand the performance of a distributed system.

Whenever there is a difference between predicted and measured latencies, the question to ask oneself is: ‘‘Where is the additional time going?’’ A tool that can be of great help in answering this question is the *profiler*, which reports on the dynamic behavior of a program: how often each function is called and how much time is spent in each function. It is sometimes difficult to get profilers to work properly in a distributed environment, but it is worth the trouble. A cruder form of profiling, which should always precede the use of a profiler, is checking which components of a distributed system are accumulating the bulk of the CPU time. All operating systems provide a utility like Unix’s *ps* program that can be used to perform this function. The knowledge of which processes are consuming the time allows one to target one’s profiling effort to the correct components.

For example, in early runs of our experiments, ps indicated that the performance measurement application was accumulating substantially more CPU time than the representative servers, which was highly suspect. We profiled the application, and found that it was spending a substantial fraction of its time in a function called *spin_lock*. A little sleuthing revealed that this function resided in the *cthreads* library [13], which had been erroneously configured to do spin locking on uniprocessors. We modified cthreads to rectify the situation, recompiled our system, and the performance problem went away.

An important source of problems in primitive analyses is analyzing in terms of the wrong primitives. Initially, instead of using the Round primitive, we used two separate

primitives, the *Subtransaction* and the *Parallel RPC*. These primitives seem more natural, and should provide more information about where the time is going; nested transaction costs and communication costs are two separate categories. Our measured costs were consistently higher than our predicted costs even for array RSMs, which are barely more complex than the primitives. This led us to think about how array RSM timings differed from primitive timings. The biggest difference that we could think of was that the array RSM operations combined several primitives.

This observation led us to time a parallel RPC nested in a subtransaction, using the primitive timing program. The incremental latency for the local form of this operation was 17.7 ms, while the latencies for Subtransactions and local Parallel RPCs were 4.4 ms and 3.9 ms.[12] The Round cost is more than double the sum of the costs of the Subtransaction and the Parallel RPC. A similar discrepancy exists in the distributed case: the incremental latency of a distributed parallel RPC is 31 ms, while that of a distributed Round is 43 ms. We switched to using the Round as a primitive operation, and our predicted costs came much more into line with our measured costs. The choice of primitives in primitive analyses is discussed in greater detail in Section 7.2.9.

Several more problems had to be discovered and corrected to achieve the level of agreement reported in Sections 7.2.5 and 7.2.6. For the most part, the techniques involved were similar to those used in the scenarios described above. Profiling revealed the fact that some transactional locks were substantially more expensive than others, causing us to use two forms of the locking primitive in our analysis. Use of the ps program alone revealed that a transaction system component in a distributed test was incorrectly configured. Closer examination showed that it was running with debugging output enabled, resulting in extremely high latencies for all operations involving the component.

7.2.9. A Note on the Primitive Analysis Methodology

The problem with our initial decision to use the Subtransaction and Parallel RPC primitives points to a basic problem with the primitive analysis methodology. The methodology relies on the property that the incremental latency of a compound operation is equal to the sum of the incremental latencies of the component primitives. Let us call a set of primitives for which this property holds an *independent* set of primitives.

Before performing a primitive analysis, it would be nice to know that the chosen set of primitives was independent. For a given set of primitives, however, there are infinitely

[12]Internally, this discrepancy results from the fact that each server has to execute a Camelot *TS_Join* call for each subtransaction that it participates in.

many ways in which the primitives can be combined to form operations. As a consequence, it is not possible to check by enumeration whether a given set of primitives is independent. But it is possible to make strides in that direction.

Suppose we have a set of primitives such that, for any two distinct primitives p and q drawn from the set, the incremental latency of the compound operation $p \circ q$ is equal to the sum of the incremental latencies of the primitive operations p and q. We say that such a set of primitives is *pairwise independent*.

Unlike independence, it is practical to check whether a set of primitives is pairwise independent. If there are n primitives in the set, one must time the incremental latencies of the of the $n(n-1)$ ordered pairs of distinct primitives to test for pairwise independence. Note that order *is* important. For example, consider the primitives that we used initially in our experiments. A Parallel RPC in which each RPC does a Subtransaction *will* have an incremental latency equal to the sum of the latencies of the Subtransaction and the Parallel RPC, but the set is not pairwise independent.

Pairwise independence is clearly a necessary condition for independence. While it is not a sufficient condition, we conjecture that in many practical systems, pairwise independence of a set of primitives is a sufficient condition for primitive analysis to work reasonably well most of the time. That is to say, we believe that it is unlikely that there will be any ‘‘severely nonlinear’’ interactions of three or more primitives if there are no such interactions between any two primitives.

One more note should be added concerning the primitive analysis methodology. The methodology is defined in terms of *incremental latency*. In order for this term to be well defined, it is critical that the latency of an operation be linear in the number of occurrences of any primitive in the execution of the operation. In real systems, this often is not the case. Many transaction system primitives involve computation that is quadratic or worse in the number of occurrences of the primitive in a given transaction. However, it typically takes hundreds or thousands of occurrences of these operations in a single transaction for the quadratic behavior to significantly affect the latency of the transaction. In practice, a primitive analysis will yield reasonable results if the latency of an operation is approximately linear in the number of occurrences of each primitive for reasonable numbers of occurrences per operation.

7.3. Concurrent Performance

We performed an experiment to test the performance of the hash table RSM under concurrent use. In this experiment, we compared the performance of a single RSM simultaneously accessed by two clients to that of two separate RSMs each accessed by one client. In the latter case, both RSMs had their representatives at the same three representative servers. One would expect the performance of concurrent operations on separate RSMs to be slightly better, as these operations will never conflict for locks, whereas operations on the same RSM will occasionally conflict, resulting in increased latency. In this experiment, we measured latencies of transactions containing a single operation, rather than incremental latencies, as no additional information would be gained from the use of incremental latencies.

The experiment consisted of two parts, one for the single RSM and one for the two separate RSMs. Part 1 consisted of three phases. In the first phase, each client performed 250 Writes to the RSM, so that it contained approximately 500 occupied locations. In the second phase, each client performed 100 sets of 100 transactions, each containing 1 operation, for a combined total of 20,000 operations. This phase allowed the RSM to come to equilibrium. In the final phase, during which data was gathered, each client performed 100 sets of 100 transactions each containing 1 operation.

Part 2 consisted of three phases, each similar in nature to the corresponding phase in Part 1. In the first phase, each client performed 500 Writes to its own RSM, so that both RSMs contained 500 occupied addresses. In the second phase, each client performed 200 sets of 10 transactions each containing 10 operations, for a total of 20,000 operations on each RSM. The final phase was identical to the final phase in Part 1, except that each client performed the operations on its own RSM.

The results of this experiment are presented in figure 7-11. The transaction times illustrated in this graph were output directly by the performance measurement application in the third phases of the two parts of the experiment. As expected, the latencies of concurrent operations on a single RSM are slightly higher than those on separate RSMs. The average increase in latency is 6%, a reasonable overhead for performing concurrent operations on the same data object. Note that this figure is sensitive to many parameters, including the number of hash buckets in the representatives and the length of the concurrent transactions.

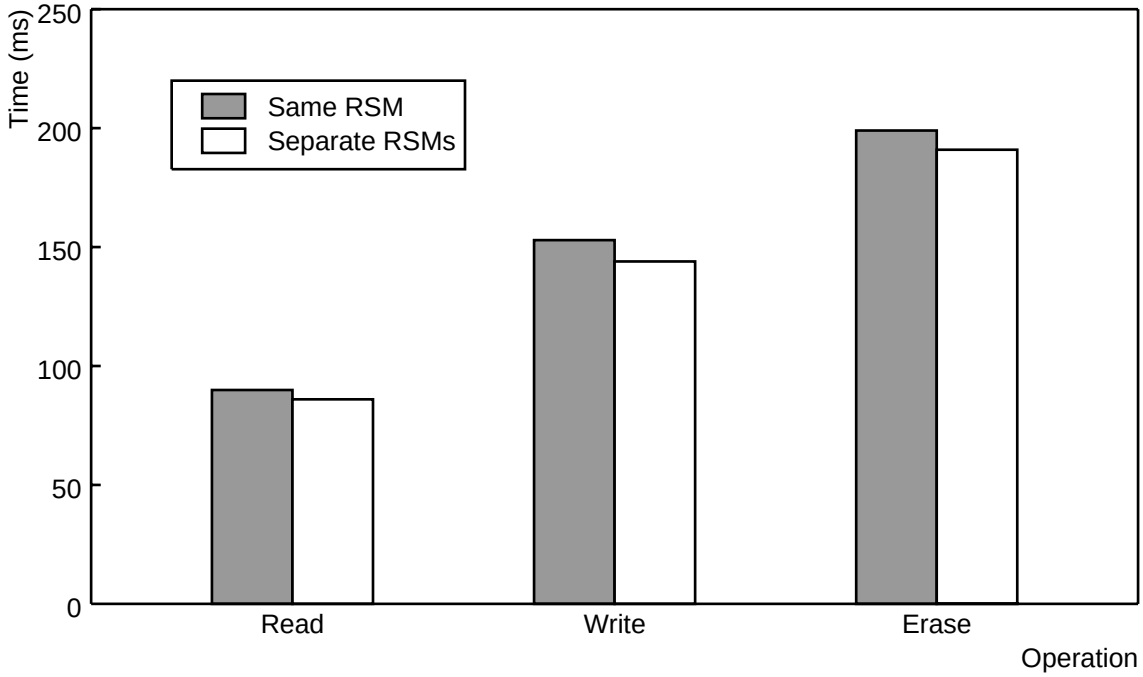

Figure 7-11: Transaction Times Under Concurrent Usage

7.4. Optimistic Timestamp Performance

In Section 4.1.1, we introduced the notion of the optimistic timestamp. We noted that optimistic timestamps are more prone to fail when there are *hot spots*: locations that are frequently written by multiple clients. However, we conjectured that optimistic timestamps would not perform badly even when there were hot spots. We performed an experiment to test this conjecture.

In the experiment, we created a severe hot spot. Two clients repeatedly performed transactions that wrote to the same location in an array RSM. This is the fastest transaction possible that performs a Write operation on a replicated object. Each client performed 100 sets of 100 transactions containing 1 Write operation, for a total of 10,000 Write operations per client. We were not interested in the times reported by the clients, only the communication cost histograms.

We were surprised by the results of the experiment. In both clients, well over 99% of the write operations required only a single round. This means that in our prototype system, virtually all Write operations require only a single round. It is a testament to the synchronization of the real-time clocks of the workstations used in the tests. This synchronization is performed by the Internet standard NTP clock synchronization protocol [41, 42], developed by David Mills.

7.5. Failure Recovery Performance

One aspect of our prototype's performance that is conspicuously missing from this chapter is failure recovery performance. We consider the failure recovery performance of our protocols to be a major selling point. While many replication protocols perform costly distributed reconfiguration to furnish continued availability in the face of a replica failure, ours do not. It would seem natural to perform experiments measuring the delay introduced into an operation by the failure of an underlying representative server. We initially planned to perform such experiments, but on reflection, we came to the conclusion that it would be difficult to measure the desired quantity, due primarily to the coarse granularity of the Unix real-time clock. However, it is possible to roughly estimate the failure recovery performance of our protocols without the need for experiment.

Our protocols react to the failure of a representative server by retrying all or part of the failed operation with a new quorum containing a replacement for the failed server. The delay introduced by the failure is the sum of the time required to sense the failure and the time spent performing and aborting the failed operation. In the worst case, the time to sense a failure is the timeout interval on the RPC to the representative server. The time spent performing and aborting the failed operation will typically be negligible compared to any reasonable distributed RPC timeout interval, so the maximum delay introduced by a failure is essentially the RPC timeout interval.

Most representative server failures will probably cause a delay that is much shorter than the RPC timeout interval, as Mach and Camelot attempt to actively propagate news of server crashes. If a client is aware that a server process has crashed before an RPC is made to the server, the RPC will be cut short immediately at the client side. This case may be relatively common, as many clients will perform RSM operations only intermittently (e.g., interactive clients). If a server's node is aware that the server has crashed but the client's node is not yet aware, the RPC will return as soon as it reaches the server's node. Even if a server failure occurs during the execution of an RPC, the RPC will probably terminate before it would time out, because the enclosing transaction will be automatically aborted by the Camelot transaction manager at the server's node.[13]

If a server's *node* crashes, as opposed to just the server *process*, it will take longer for the client's node to become aware of the failure; system processes on the server's node will no longer be around to report the failure to their peers on the client's node. But even in the case of a node failure, an RPC may well terminate before it would time out, as the

[13]The *death pill* mechanism [8] in the Camelot library causes RPCs to be promptly terminated when the enclosing transaction aborts.

"ping" timeout interval used by Mach's *network message server* to detect node crashes is likely to be shorter than the RPC timeout interval.

In summary, the delay introduced by the failure of a representative server may vary from under a millisecond to the timeout interval on RPCs to the server, which may be as high as 60 seconds. The distribution of delay times is affected by many things, including the frequency with which the client performs RSM operations, and the details of the underlying transactional RPC implementation. While it is difficult to make a good quantitative estimate for the average delay, a reasonable guess might be several times the average cost of a distributed Round on the underlying platform.

7.6. Conclusions

In this chapter, we evaluated various aspects of the performance of our prototype. While the prototype is not blazingly fast, it is not slow either. Its lackluster performance is primarily a reflection of the speed of the underlying hardware and system software, in particular, the transaction and communication facilities. This is demonstrated by comparing the replicated operation costs in Table 7-3 to the baseline *non-replicated* Round cost of 38 ms.

On the hardware and system software used in the experiments, we estimate that the time required to perform RSM Read and Write operations is within 10 percent of the time that would be required to perform the corresponding operations on a non-replicated remote data object. Our estimate for the replication overhead for the Erase operation varies from 10 to 120 percent, depending on the type of RSM that is used.

The prototype demonstrates the feasibility of our architecture. The experimental results suggest that the performance of the architecture is quite good relative to that of the underlying transaction and communication primitives. The transaction and communication systems underlying the prototype are in themselves research prototypes. Neither Camelot nor Mach excel in the key areas that determine the performance of the prototype.

Mach's distributed RPC times on the hardware described in Section 7.1 are on the order of 20 ms. Much better RPC performance has been achieved on similar hardware by several research operating systems with highly optimized RPC facilities. See Schroeder and Burrows for a comparison of RPC times for such systems [52]. There are few transaction systems available today that offer facilities comparable to Camelot's, and none have been optimized for performance. However, we are aware of many inefficiencies in Camelot's protocols and we believe that the next generation of general purpose distributed transaction systems will be much faster. When the bright outlook for the future of high speed transaction and communication facilities is taken into account,

our performance results lead us to believe that our architecture shows great promise of practicality.

Our success in predicting the performance of the prototype from the performance of the underlying primitives indicates that we have a good understanding of the system, and of our algorithms. As a result, we can have a fairly high degree of trust in our decompositions of operations into primitives (Table 7-2). These decompositions can thus be used with some confidence to predict the performance of our architecture on a faster platform.

The communication cost histogram data from the basic timings, as well as the results of the optimistic timestamp experiment, show that our optimistic two-stage protocols are extremely effective in reducing communication costs. But the results show that communication still accounts for the lion's share of the latency. This validates our focus on reducing the number of rounds of RPCs as a means of decreasing latency. The dominance of communication costs in the prototype is exacerbated by the poor transactional RPC performance of Camelot and Mach, but even the most highly optimized communication systems have distributed RPC times two orders of magnitude higher than their normal procedure call times [52]. Therefore, our focus on reducing RPCs is valid independent of the underlying platform, and our optimistic two-stage protocols will remain effective in reducing latency on any platform.

Chapter 8
Conclusions

In this chapter, we enumerate the contributions of this work, present directions for future study, and summarize our main results.

8.1. Contributions

The contributions of this dissertation to computer science can be divided into two areas: systems and theory. The primary contribution in the systems area is the demonstration of the practicality of replication. In the process of pursuing this goal, several secondary contributions were made. One such contribution is the invention of *optimistic timestamps*, a technique for reducing communication costs in blind-write operations on replicated objects. A related contribution is the invention of *optimistic two-stage protocols*, a generalization of the optimistic timestamp technique to a broad class of distributed algorithms. Another secondary contribution is the invention of *auto-batching*, a technique for reducing communication costs by automatically combining concurrent RPCs.

Our prototype implementation is an interesting demonstration of the use of a general purpose transaction system; it is one of the largest and most sophisticated applications ever built on the Camelot system. Our performance studies on the prototype provide a good example of the use of the *primitive analysis* technique to aid in the understanding of the performance of a complex distributed system. In the process of performing the analysis, we formalized the primitive analysis methodology and made several observations concerning its use. In particular, our notion of *pairwise independence* should assist future systems practitioners in choosing appropriate primitives for use in primitive analyses.

The primary contribution of this dissertation in the theory area is the development and analysis of the *replicated sparse memory* and its variants, which comprise a new family of distributed data structures and related algorithms. A key component of the development is the proof of a basic structural property of the RSM that allows us to construct a fast algorithm for the Erase operation. Our average-case performance

analysis of the RSM represents an interesting mathematical analysis of a complex distributed data structure. The analysis was remarkably successful in predicting the performance of simulations and yielded many insights into the operation of the data structure. A secondary contribution in the theory area is the development of another distributed data structure, the *replicated counter*.

8.2. Directions for Future Work

An internal attribute of the RSM that merits further study is *delete list length* (Section 3.1). While the performance analysis in Chapter 3 tells us the *average* delete list length that will be encountered in an RSM under random use, it does not yield any additional information about the distribution. This distribution relates to the tradeoff between the number of entries returned in the first stage of the real predecessor algorithm and the likelihood that the second stage will be necessary. We know from our experiments that returning eight entries in the first stage is sufficient to eliminate the second stage 98% of the time (Section 7.2.3) but we do not know any other points on the spectrum. The simplest approach to obtaining the distribution is to instrument our prototype implementation to keep a histogram of delete list lengths encountered. Alternatively, a mathematical analysis could be attempted.

The performance studies in Chapter 7 concentrate on the latency of the operations on our data objects. It would also be worthwhile to study the throughput of these operations. The concurrent performance studies in Section 7.3 were fairly limited: they could be expanded to include other RSM variants and access patterns. The optimistic timestamp performance studies in Section 7.4 could be expanded to include more severe hot spots, produced by more than two clients concurrently accessing the same RSM location.

It would be very interesting to do a performance comparison between our algorithms and other replication algorithms. Unfortunately, a meaningful comparison would be very difficult because of differences in the function exported by various replication algorithms, differences in the underlying system models they assume, and the tendency of many researchers in the field to omit detail from descriptions of their algorithms. Perhaps the most promising approach to comparing the performance of replication algorithms is to publish a carefully designed benchmark for these algorithms and encourage practitioners in the field to report on the benchmark performance of their algorithms. The success of this approach would be aided by the emergence of a standard transaction system base, which would provide a natural platform for implementing the many replication algorithms that require the services of a transaction system.

Several techniques presented in this dissertation might have applications outside our replication algorithms. Among these techniques are optimistic timestamps, optimistic

two-stage protocols, and auto-batching. One possible area for further exploration is the breadth of applicability of these techniques. Methods from queueing theory might be used to analyze the sensitivity of optimistic timestamps to concurrency level, update frequency, clock accuracy, clock precision, and communication speed. A search could be undertaken for distributed algorithms that fit into the *two-stage protocol* framework (Section 4.1.2); such algorithms are candidates for optimistic two-stage protocols. Similarly, a search could be undertaken for applications wherein a single process sends multiple independent streams of RPCs to a server; such applications are candidates for auto-batching. Unlike the other techniques discussed in this paragraph, auto-batching has not been implemented. It might be worthwhile to implement it and study its performance.

8.3. Summary

In this dissertation, we presented efficient replication algorithms for a family of table-like data objects known as *replicated sparse memories*. The algorithms, which rely on the support of an underlying distributed transaction system, run on a collection of general-purpose computers connected by a network. They were designed primarily for communication efficiency, using *optimistic timestamps* and other *optimistic two-stage protocols*. We showed how a wide variety of useful data objects could be implemented efficiently on top of replicated sparse memories. We presented an architecture that implements our algorithms to provide application programmers with easy-to-use, generic replicated data objects. We built a prototype system to demonstrate the feasibility of our approach, and performed experiments to evaluate its performance. While the absolute performance of the prototype was not outstanding, it was excellent relative to that of the underlying communication and transaction system software.

It seems clear that the dominant hardware paradigm in the upcoming years will be high speed local area networks of general purpose computers, file servers, display servers, and the like, which may in turn be connected by long-haul networks. As microprocessor technology matures, inexpensive computers are becoming extremely fast. It is likely that distributed transaction system technology, which has been with us for over twenty years, will soon come of age. The need for highly available computer systems has never been greater. While replication has been discussed in the literature for years, the only highly available systems technology generally viewed as commercially viable is special purpose hardware with locally replicated data. Distributed replication, if it can be made practical, offers much lower cost, and potentially, higher availability. Our thesis is that it can be made practical.

Appendix A
Detailed Formulation of Balance Equations

Let us first construct the balance equation for current entries. A formal statement of the rate balance assertion is:

E[Number of entries entering current class in a chosen representative in one opr]
= **E**[Number of entries leaving current class in a chosen representative in one opr].

These expected values are computed over a space consisting of all of the possible state transitions in our model. We expand the expectation values on both sides of the equation by breaking the space up into three subspaces: the transitions that result from Insert operations, Update operations and Erase operations:

P[Opr is Insert] × **E**[Number of entries entering current class in one Insert]
+ **P**[Opr is Update] × **E**[Number of entries entering current class in one Update]
+ **P**[Opr is Erase] × **E**[Number of entries entering current class in one Erase]
= **P**[Opr is Insert] × **E**[Number of entries leaving current class in one Insert]
+ **P**[Opr is Update] × **E**[Number of entries leaving current class in one Update]
+ **P**[Opr is Erase] × **E**[Number of entries leaving current class in one Erase].

We will assume that all of the probabilities in this equation are 1/3, as Inserts, Updates and Erases occur with almost equal likelihood. The reason that they do not occur with exactly equal likelihood is that Erases and Updates cannot occur in states where the RSM is empty, and Inserts cannot occur in states where the RSM already contains every address in the address space. However, these states represent a negligible fraction of the state space and they all occur with extremely low probability. Each term has one of these factors, so under the assumption, they all cancel out.

To formulate the first balance equation in terms of the unknowns, we expand the expected values in the order they appear in the equation. The first term is:

E[Number of entries entering the current class in one Insert operation] .

A single entry will enter the current class if and only if the representative under observation is chosen for the write quorum of the Insert operation. Thus the expected value is merely the probability that the representative is chosen. Since there are N representatives in the suite, and W are chosen at random for the write quorum, this is W/N.

The second term is:

E[Number of entries entering the current class in one Update operation] .

Again, an entry can enter the current class only if the representative is chosen for the write quorum. This time, however, the entry for the address being updated will not necessarily enter the current class, as the representative could already have contained a current entry for this address. In that case, no entry that was not already current would become current. Thus, the value of the term is:

P[The representative is chosen for the write quorum]
$\times$ (1 – **P**[Rep already contains a current entry for address being updated]).

The probability that the representative is chosen for the write quorum is W/N. The address to be updated is chosen at random from occupied addresses in the RSM so:

P[Representative already contains a current entry for address being updated]
= **P**[Representative contains a current entry for randomly chosen occupied address]
= c'

Thus, the value of the second term is:

$$\frac{W}{N}(1-c').$$

The third term is:

E[Number of entries entering the current class in one Erase operation].

There is no way for an entry to become current in the Erase operation, so this term vanishes.

Now we come to the terms on the right hand side of the balance equation. The first term on the right hand side is:

E[Number of entries leaving the current class in one Insert operation].

This term vanishes, as no entries leave the current class in Insert operations.

The second term on the right hand side is:

E[Number of entries leaving the current class in one Update operation] .

If the representative under observation contains a current entry for the address being updated, and the representative is *not* chosen for the write quorum, then the current entry becomes outdated. Thus the value of this term is:

(1 – **P**[The representative is chosen for the write quorum])
$\times$ **P**[Rep contains a current entry for a randomly chosen occupied address]

$$= (1-\frac{W}{N})\,c' .$$

The third term on the right hand side is:

E[Number of entries leaving the current class in one Erase operation] .

If the representative under observation contains a current entry for the address being erased, the entry will leave the current class regardless of whether the representative is chosen for the write quorum. If it is chosen, the entry will be deleted outright; otherwise, the entry will become a ghost. Thus the value of this term is:

P[The representative contains a current entry for the address being erased]
$= c'$.

Combining all these terms, the balance equation for current entries is:

$$\frac{W}{N} + \frac{W}{N}(1 - c') = (1 - \frac{W}{N})c' + c'.$$

Simplifying, we get:

$$c' = \frac{W}{N}.$$

We now construct the balance equation for outdated entries. By the same argument used in the construction of the first balance equation, a formal statement of the rate balance assertion becomes:

E[Number of entries entering the outdated class in one Insert operation]
+ **E**[Number of entries entering the outdated class in one Update operation]
+ **E**[Number of entries entering the outdated class in one Erase operation]
= **E**[Number of entries leaving the outdated class in one Insert operation]
+ **E**[Number of entries leaving the outdated class in one Update operation]
+ **E**[Number of entries leaving the outdated class in one Erase operation] .

We make the assumption that entries cannot enter the outdated class in Insert operations, so the first term of the left hand side of the equation vanishes. In fact, if an address is inserted when a ghosts for a previous incarnation of that address still remains in a representative outside of the write quorum for the Insert operation, the ghosts will become outdated. However, this is an extremely unlikely event, hence this term of the equation is negligible compared to the others. Furthermore, it is not expressible in terms of the unknowns.

In the Update operation an entry can become outdated as follows. If the representative is not chosen for the write quorum and it contains a current entry for the address being updated, the entry becomes outdated. Thus the value of the second terms is:

(1 − **P**[The representative is chosen for the write quorum])
× **P**[Rep contains a current entry for a randomly chosen occupied address]
$= (1 - \frac{W}{N})c'$.

When an Erase operation occurs, entries for the real predecessor and real successor of the address being erased are inserted into each member of the write quorum where they do not already appear. They are inserted with version number zero, which assures that they are outdated entries. This is the only way entries can enter the outdated class in an Erase operation. Thus the number of entries entering the outdated class in the observed representative in an Erase operation is zero if the representative is not chosen for the write quorum. If it is chosen for the write quorum, then one entry will become outdated if the representative does not contain an entry for the real predecessor of the address being erased, and another entry will become outdated if the representative does not contain an entry for the real successor.

We introduce some notation for events, to simplify the discussion that follows:

P = {Representative contains an entry for real predecessor of address being erased}
S = {Representative contains an entry for real successor of address being erased} .

On the basis of the previous observations, the value of the term being expanded is:

$$\mathbf{P}[\text{The representative is chosen for the write quorum}] \times (\mathbf{P}[P^c] + \mathbf{P}[S^c])$$

$$= \frac{W}{N}((1-\mathbf{P}[P]) + (1-\mathbf{P}[S])) .$$

While $\mathbf{P}[P]$ and $\mathbf{P}[S]$ cannot be exactly expressed in terms of our unknowns, they can be very closely approximated. The address to be erased is chosen at random from the occupied addresses in the RSM, and its real predecessor is merely the occupied address immediately preceding it in the RSM. If the address being erased is the first occupied address in the RSM, its real predecessor is the dummy address **LOW**, which is always present in every representative. Thus the probability that the real predecessor is present in the representative ($\mathbf{P}[P]$) is just slightly higher than the probability that a randomly chosen occupied address is present in the representative. For a large address space like the one used in the simulations they will be practically identical. By symmetry, the same argument holds for the real successor. In fact, it shows that $\mathbf{P}[P] = \mathbf{P}[S]$. Therefore, we make the assumption that:

$\mathbf{P}[P] = \mathbf{P}$[Representative contains an entry for a randomly chosen occupied address]
= **P**[Rep contains a current entry for a randomly chosen occupied address]
+**P**[Rep contains an outdated entry for a randomly chosen occupied address]
$= c' + o'$,

The third term becomes:

$$2\frac{W}{N}(1-(c'+o')).$$

Entries cannot leave the outdated class in Insert operations, so the first term of the right hand side of the equation vanishes. In an Update operation, an entry can leave the outdated class as follows. If the representative is chosen for the write quorum and it

contains an outdated entry for the address being updated, then this entry is replaced by a current one. Thus, the second term on the right hand side is:

P[The representative is chosen for the write quorum]
$\times$ **P**[Representative contains an outdated entry for the address being updated]
= **P**[The representative is chosen for the write quorum]
$\times$ **P**[Rep contains an outdated entry for a randomly chosen occupied address]
$= \frac{W}{N} o'$.

In an Erase operation, an entry can leave the outdated class as follows: If the representative contains an outdated entry for the address being erased, then the entry disappears if the representative is chosen for the write quorum, and it becomes a ghost if the representative is not chosen for the write quorum. Thus the third term on the right hand side is:

P[The representative contains an outdated entry for the address being erased]
= **P**[Rep contains an outdated entry for a randomly chosen occupied address]
$= o'$.

Putting it all together, the balance equation for outdated entries is:

$$(1-\frac{W}{N})c' + 2\frac{W}{N}(1-(c'+o')) = \frac{W}{N}o' + o'.$$

Simplifying, this becomes:

$$o' = \frac{(N-3W)c' + 2W}{N+3W}.$$

Finally, we construct the balance equation for ghost entries. A formal statement of the balance assertion becomes:

E[Number of entries entering the ghost class in one Insert operation]
+ **E**[Number of entries entering the ghost class in one Update operation]
+ **E**[Number of entries entering the ghost class in one Erase operation]
= **E**[Number of entries leaving the ghost class in one Insert operation]
+ **E**[Number of entries leaving the ghost class in one Update operation]
+ **E**[Number of entries leaving the ghost class in one Erase operation] .

Entries can only enter the ghost class in Erase operations; thus, the first and second terms of the equation vanish. An entry becomes a ghost in a representative if its address is being erased and that representative is not chosen for the write quorum of the erase operation. Thus the second term is:

(1 − **P**[The representative is chosen for the write quorum])
× **P**Representative contains an entry for a randomly chosen occupied address]

$= (1-\frac{W}{N})(c'+o').$

Entries rarely leave the ghost class in Insert operations, thus we shall assume the first term on the right hand side vanishes. (This is essentially the same assumption we made on page 155 when constructing the balance equation for outdated entries.) Entries cannot leave the ghost class in Update operations, so the second term on the right hand side actually does vanish. If the representative is chosen for the write quorum of the Erase operation then all of the ghosts comprising the delete list of the address being erased will be removed from the representative. Thus the third term of the right hand side is:

P[The representative is chosen for the write quorum]
× **E**[The size of the delete list of the address being erased]

$= \frac{W}{N} d\ .$

Putting the terms together, the balance equation for ghosts is:

$$(1-\frac{W}{N})(c'+o') = \frac{W}{N} d.$$

Simplifying:

$$d = \frac{N-W}{W}(c'+o').$$

References

[1] Amr El Abbadi, Sam Toueg.
Availability in Partitioned Replicated Databases.
In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. 1986.

[2] Amr El Abbadi, Dale Skeen, Flaviu Cristian.
An Efficient, Fault-Tolerant Protocol for Replicated Data Management.
In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. March, 1985.

[3] P. A. Alsberg, J. D. Day.
A Principle for Resilient Sharing of Distributed Resources.
In *Proceedings of the Second International Conference on Software Engineering*, pages 562-570. October, 1976.

[4] Joel Bartlett.
A NonStop (TM) Kernel.
In *Proceedings of the Eighth Symposium on Operating System Principles*. ACM, 1981.

[5] P. Bernstein, N. Goodman.
An algorithm for concurrency control and recovery in replicated distributed databases.
ACM Transactions on Database Systems 9(4):596-615, December, 1984.

[6] Andrew D. Birrell, Bruce J. Nelson.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):39-59, February, 1984.

[7] Joshua J. Bloch.
The Camelot Library: A C Language Extension for Programming A General Purpose Distributed Transaction System.
In *Proceedings of the Ninth International Conference on Distributed Computing Systems*. June, 1989.

[8] Joshua J. Bloch.
The Design of The Camelot Library.
In Jeffrey L. Eppinger, Lily B.Mummert, Alfred Z. Spector (editor), *Guide to the Camelot Distributed Transaction Facility including the Avalon Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[9] Joshua J. Bloch, Jeanette Wing.
A Sample Camelot Application and Server.
In Jeffrey L. Eppinger, Lily B.Mummert, Alfred Z. Spector (editor), *Guide to the Camelot Distributed Transaction Facility including the Avalon Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[10] Joshua J. Bloch, Dean S. Daniels, Alfred Z. Spector.
A Weighted Voting Algorithm for Replicated Directories.
JACM 34(4), October, 1987.
Also available as Technical Report CMU-CS-86-132, Carnegie Mellon University, July 1986.

[11] Luca Cardelli, Peter Wegner.
On Understanding Types, Data Abstraction and Polymorphism.
ACM Computing Surveys 17(4):471-522, December, 1985.

[12] Douglas Comer.
The Ubiquitous B-Tree.
ACM Computing Surveys 11(2):121-137, June, 1979.

[13] Eric C. Cooper, Richard P. Draves.
C Threads.
Technical Report CMU-CS-88-154, Carnegie Mellon University, June, 1988.

[14] Eric C. Cooper.
Replicated Distributed Programs.
In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 63-78. December, 1985.
Published as *Operating Systems Review*, 19(5).

[15] Eric C. Cooper.
Replicated Distributed Programs.
PhD thesis, Computer Science Division, University of California, Berkeley, April, 1985.
Published as report UCB/CSD/85/231.

[16] Dean S. Daniels, Alfred Z. Spector.
An Algorithm for Replicated Directories.
In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 104-113. ACM, August, 1983.
Also available in Operating Systems Review 20(1), January 1986, pp. 24-43.

[17] C. J. Date.
The System Programming Series: *An Introduction to Database Systems.*
Addison-Wesley, Reading, MA, 1981.

[18] Charles T. Davies.
Recovery Semantics for a DB/DC System.
In *Proceedings of the ACM National Conference*. ACM, 1973.

[19] Jeffrey L. Eppinger, Alfred Z. Spector.
Transaction Processing in Unix: A Camelot Perspective.
Unix Review 7(1):58-67, January, 1989.

[20] K. P. Eswaran, James N. Gray, Raymond A. Lorie, Irving L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.

[21] H. Garcia-Molina, D. Barbara.
How to assign votes in a distributed system.
Journal of the ACM 34(2):841-861, October, 1985.

[22] David K. Gifford .
Weighted Voting for Replicated Data.
In *Proceedings of the Seventh Symposium on Operating System Principles*, pages 150-162. ACM, December, 1979.

[23] David K. Gifford.
Information Storage in a Decentralized Computer System.
PhD thesis, Stanford University, 1981.
Available as Xerox Palo Alto Research Center Report CSL-81-8, March 1982.

[24] David K. Gifford, Nathan Glasser.
Remote Pipes and Procedures for Efficient Distributed Communication.
ACM Transactions on Computer Systems 6(3), August, 1988.
Also Available as MIT LCS TR-384.

[25] A. Goldberg, D. Robson.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, 1983.

[26] James N. Gray.
Notes on Database Operating Systems.
In R. Bayer, R. M. Graham, G. Seegmuller (editor), *Lecture Notes in Computer Science*. Volume 60: *Operating Systems - An Advanced Course*, pages 393-481. Springer-Verlag, 1978.
Also available as Technical Report RJ2188, IBM Research Laboratory, San Jose, California, 1978.

[27] James N. Gray.
A Transaction Model.
Technical Report RJ2895, IBM Research Laboratory, San Jose, California, August, 1980.

[28] James N. Gray, et al.
The Recovery Manager of the System R Database Manager.
ACM Computing Surveys 13(2):223-242, June, 1981.

[29] Theo Haerder, Andreas Reuter.
Principles of Transaction-Oriented Database Recovery.
ACM Computing Surveys 15(4):287-318, December, 1983.

[30] Abdelsalam Abdelhamid Heddaya.
Managing Event-Based Replication for Abstract Data Types in Distributed Systems.
PhD thesis, Aiken Laboratory, Harvard Univerisity, October, 1988.

[31] Maurice P. Herlihy.
A Quorum-Consensus Replication Method for Abstract Data Types.
ACM Transactions on Computer Systems 4(1), February, 1986.

[32] Michael B. Jones, Richard P. Draves, Mary R. Thompson.
MIG - The Mach Interface Generator.
1987.
Mach Group document.

[33] T. A. Joseph.
Low Cost Management of Replicated Data.
PhD thesis, Cornell, November, 1985.

[34] John G. Kemeny, J. Laurie Snell.
Finite Markov Chains.
D. Van Nostrand & Co., New York, 1960.

[35] Brian Kernighan, Dennis Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[36] Henry F. Korth.
Locking Primitives in a Database System.
Journal of the ACM 30(1):55-79, January, 1983.

[37] Philip L. Lehman, S. Bing Yao.
Efficient Locking for Concurrent Operations on B-Trees.
ACM Transactions on Database Systems 6(4), December, 1981.

[38] Bruce G. Lindsay, et al.
Notes on Distributed Databases.
Technical Report RJ2571, IBM Research Laboratory, San Jose, California, July, 1979.
Also appears in Droffen and Poole (editors), *Distributed Databases*, Cambridge University Press, 1980.

[39] Barbara H. Liskov, Robert W. Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[40] D.P.A. MacManus.
Let Him Dangle.
In Elvis Costello (editor), *Spike*. Warner Brothers, Burbank, CA, 1989.

[41] D. L. Mills.
Network Time Protocol (Version 2) Specification and Implementation.
Technical Report DARPA Network Working Group Report RFC-1119, University of Delaware, September, 1989.

[42] D. L. Mills.
Internet Time Snychronization: the Network Time Protocol.
IEEE Transactions on Communications , 1990.
To Appear.

[43] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz.
Aries: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging.
Technical Report RJ6649, IBM Almaden Research Center, January, 1989.

[44] J. Eliot B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
MIT Press, 1985.

[45] Bruce Jay Nelson.
Remote Procedure Call.
PhD thesis, Carnegie Mellon University, May, 1981.
Available as Technical Report CMU-CS-81-119a, Carnegie Mellon University.

[46] Brian M. Oki.
Viewstamped Replication: A General Primary Copy Method to Support Highly-Available Distributed Systems.
In *Proceedings of the Seventh ACM Symposium on the Principles of Distributed Computing*. ACM, August, 1988.

[47] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel.
LOCUS: A Network Transparent, High Reliability Distributed System.
In *Proceedings of the Eighth Symposium on Operating System Principles*, pages 169-177. ACM, 1981.

[48] J. B. Postel.
User Datagram Protocol.
Technical Report RFC 768, Network Working Group, August, 1980.

[49] William Pugh.
Skip Lists: A Probabilistic Alternative to B-Trees.
In *Algorithms and Data Structures Workshop, WADS '89*. Springer-Verlag, August, 1989.

[50] Richard F. Rashid.
Threads of a New System.
Unix Review 4(8):37-49, August, 1986.

[51] J. B. Rothnie, N. Goodman, P. A. Bernstein.
The Redundant Update Methodology of SDD-1: A System for Distributed Databases (The Fully Redundant Case).
Technical Report CCA-77-02, Computer Corporation of America, 1977.

[52] Michael D. Schroeder, Michael Burrows.
Performance of Firefly RPC.
In *Proceedings of the Twelfth Symposium on Operating System Principles*, pages 83-90. ACM, December, 1989.

[53] Peter M. Schwarz.
Transactions on Typed Objects.
PhD thesis, Carnegie Mellon University, December, 1984.
Available as Technical Report CMU-CS-84-166, Carnegie Mellon University.

[54] Peter M. Schwarz, Alfred Z. Spector.
Synchronizing Shared Abstract Types.
ACM Transactions on Computer Systems 2(3):223-250, August, 1984.
Also available in Stanley Zdonik and David Maier (editors), Readings in Object-Oriented Databases. Morgan Kaufmann, 1988. Also available as Technical Report CMU-CS-83-163, Carnegie Mellon University, November 1983.

[55] Alfred Z. Spector, Dean S. Daniels.
Performance Evaluation of Distributed Transaction Facilities.
September, 1985.
Presented at the Workshop on High Performance Transaction Processing, Asilomar, September, 1985.

[56] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, Bruce G. Lindsay.
Transactions and Consistency in Distributed Database Systems.
ACM Transactions on Database Systems 7(3):323-342, September, 1982.

[57] F. Waters (editor).
IBM RT Personal Computer Technology.
International Business Machines Corporation, 1986.

[58] William E. Weihl.
Specification and Implementation of Atomic Data Types.
PhD thesis, MIT, March, 1984.

[59] William E. Weihl.
Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types.
ACM Transactions on Programming Languages and Systems 11(2):249-283, April, 1989.

[60] Andrew Chi-Chih Yao.
On Random 2-3 Trees.
Acta Informatica (9):159-170, 1978.