Efficient Replication Management in Distributed Systems

by

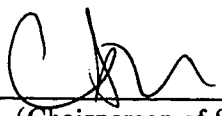
Michael Rabinovich

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1994

Approved by 
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree Dept. of Computer Science and Engineering

Date 08-25-94

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor Michigan 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature 

Date 08-26-94

University of Washington

Abstract

Efficient Replication Management in Distributed Systems

by Michael Rabinovich

Chairperson of the Supervisory Committee: Professor Edward D. Lazowska
Department of Computer Science
and Engineering

Replication is a critical aspect of large-scale distributed systems. Without it, the size of a system is limited by factors such as the risk of component failures, the overloading of popular services, and access latency to remote parts of the system.

Replication overcomes these problems by allowing service to continue despite failures using remaining replicas, and by distributing requests for a given service among multiple server replicas. However, replicated systems incur significant performance overhead for maintaining multiple replicas and keeping them mutually consistent. Managing replication efficiently is therefore important in building large-scale distributed systems.

This dissertation concentrates on quorum-based replication management. It proposes several ways to manage replication efficiently using different types of quorums.

First, we study structure-based quorums, which are attractive because they result in low-overhead replica management when the number of replicas is high. We propose a way to significantly improve system availability in protocols using these quorums. We also study in depth the performance of a particular class of these quorums based on a grid structure.

Second, for voting-based quorums, we propose a way to include new or repaired replicas into the system and exclude failed, disconnected, or deactivated replicas, asynchronously with user operations. Thus, such reconfiguration does not involve any service interruption and can be done more freely for various purposes, e.g., improving data availability, migrating data, or redistributing load.

Third, we propose a way to efficiently manage read-one-write-all (ROWA) replication. Protocols based on ROWA quorums maintain consistency by reading one copy of the data and writing all copies. They are especially attractive because they provide for highly efficient and fault-tolerant read operations, and because many services and data in distributed systems fall under the mostly-read category. The obvious weakness of ROWA, which has prevented it from being widely used in commercial wide-area networks, is lack of fault-tolerance for writes. This thesis proposes a new ROWA protocol that provides fault-tolerant write operations without any significant deterioration of read properties. This protocol offers immediate benefits to current systems.

Finally, the thesis studies some issues that are common for all types of quorums, including semantics of write operations and efficient protocols for transaction commitment.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I wish to thank my advisor, Ed Lazowska, for his invaluable help over the last few years. It ranged from professional guidance to advice and encouragement in very personal matters. I consider Ed to be not only a superb advisor, but a friend as well.

I would like to thank the other members of my doctoral committee, and especially the members of my reading committee, Hank Levy and John Zahorjan, for their time and effort on my behalf. I am especially grateful to John Zahorjan for many insightful comments and lengthy discussions that significantly improved the presentation of this thesis and cleared it of a few bugs.

I would also like to acknowledge Sandy Kaplan for scrupulously editing an early draft of this dissertation. Her numerous comments made it much more readable.

I would like to thank my two summer internship hosts, Garret Swart of the DEC Systems Research Center and Rafael Alonso of Matsushita Information Technology Lab, for the very enjoyable summers I spent there. Garret introduced me, then a first-year graduate student, to the area of distributed systems. My summer work with Rafael late in my graduate career was a tremendous amount of fun.

I have enjoyed collaborative work with Akhil Kumar of Cornell University. He has been a good colleague and friend.

I owe a great deal to my wife, Irina, for sharing with me the joys and hardships of life as a graduate student. I would also like to thank our baby daughter for letting us sleep at night.

Finally, I owe an immense debt to my parents. Their devotion and unshakeable belief in my abilities kept me going during difficult years back in the Soviet Union, when so many of my friends caved in to the system and abandoned their dreams of having a fulfilling life. In a late stage of their lives, my parents gave up their own work and left their friends, culture, and the graves of their loved ones so that their son would have an opportunity to become a scientist. Well, Mom and Dad, here is my dissertation. There is perhaps as much your sweat in it as mine.

To my parents: Semyon Rabinovich and Eugenia Treyster.

Chapter 1

Introduction

As the scale of distributed systems grows, two important problems arise. First, the risk of some data and services being unavailable due to node or network failures increases. (Moreover, any failure in a large system potentially affects numerous clients.) Second, popular services may become a bottleneck, degrading overall system performance.

A common way to address these problems is to introduce some redundancy into the system by means of data and service replication. There are several ways in which the system can benefit from such replication. First, the resilience of the system to node failures increases: if some replicas fail, the service can continue using the remaining replicas. Second, bottlenecks can be avoided by distributing the requests for a given service among multiple servers that provide this service. Moreover, in a wide-area network, appropriate placement of replicas of a service provides the potential to satisfy most requests for that service from nearby replicas, no matter where the request came from; this decreases the communication costs of an operation.

However, dangers and costs also accompany replication. Most importantly, without proper management, multiple replicas of the same data item may diverge. For instance, if the system partitions and different regions contain replicas of the same data item, updates of this item may independently succeed in these regions. When the partioning heals, however, the system will contain two inconsistent versions of the data. Reconciling (or even detecting) these inconsistencies may be difficult, and may require human inter-

vention. Maintaining consistency of multiple replicas is the most fundamental problem in replication management, and the first question the system designer must resolve is how strictly to enforce replica consistency. As we will see in Chapter 2, this decision affects all other aspects of the system, from the complexity of the protocols involved to the levels of fault-tolerance and performance that can be achieved.

Existing systems exhibit a range of solutions to replica consistency. Some systems provide "best effort" consistency, where consistency is not guaranteed even in the absence of failures (e.g., the Ficus distributed file system [52]). Other systems guarantee consistency in the absence of failures but allow replicas to diverge when certain types of failures occur (e.g., the Coda file system [66]). Still other systems maintain strict consistency even in the presence of failures (e.g., the HARP [41] and Echo [26] file systems).

A related question in designing a replica management protocol concerns the kinds of failures tolerated by the system (i.e., failures that cannot cause incorrect system behavior). Some protocols can tolerate only node failures, and may behave incorrectly when the system partitions. Other protocols tolerate both node and network failures. Not surprisingly, protocols that do not have to tolerate network partitionings generally exhibit better characteristics than protocols that must behave well under both node and network failures.

The most obvious performance overhead of replication involves the costs of maintaining multiple replicas of the data or multiple instances of the same service. For example, a system with fully duplicated data consumes at least twice as much disk space as a non-replicated system. (Disk consumption may increase more than two-fold, because some space may be used by the replica management protocol itself.) In addition, keeping multiple replicas consistent often requires that operations on the data (or any request for the service) be performed on multiple replicas. This involves communicating with multiple replicas on every operation. Moreover, each replica must perform its own computation to service the operation. Thus, the cost of an individual operation in a replicated system may be higher than the cost of the same operation in a non-replicated system.

Replication, then, is a critical aspect of large-scale distributed systems, but one with a number of complex tradeoffs. This thesis explores several ways to manage replication efficiently while guaranteeing replica consistency at all times. It addresses a broad range

of problems and tradeoffs in replica management, from theoretical issues to practical problems that have prevented the use of some existing schemes in real systems. In fact, there is a perception among practitioners that advanced replica management schemes incur too much overhead to be practically useful. As a result, there has been a widening gap between research in replica management and what is used commercially. Some of the ideas described in this thesis may help to reverse this trend.

1.1 The System Model

This thesis considers a distributed system that replicates data items on several nodes.[1] The protocols responsible for maintaining replica consistency are called *replica management* protocols or, more often, *replica control* protocols.

Two operations on the data items, read and write, should be supported by the replica control protocol. Note that, while the ideas in this thesis are described in terms of read and write operations on replicated data items, this model is not restricted to accesses to data. Indeed, in a system with replicated servers, the server state corresponds to the data item in our model, requests that do not change the server state correspond to reads, and the ones that do correspond to writes. Similarly, the model is applicable to an object-oriented distributed system that replicates objects.

Two different models are common for distributed systems with data replication. In the client-server model, the system consists of two distinct sets of nodes: servers that store the data and clients that initiate operations. The other model assumes that all nodes are equal, that each node keeps a full-rights replica, and that each can initiate operations. With respect to replica control, the difference between these models is whether the node that initiates the operation has a full-rights replica of the data and whether it can be trusted. In the client-server model, the coordinator is often not trusted and therefore may not always use message broadcast to communicate with the servers [Man89]. Most replica control protocols can be easily translated from one model to the other, which is also the case for protocols proposed in this thesis. We refer to nodes that

[1] However, we do not consider data caching in distributed systems, where transient replicas of a data item are created on client nodes. Rather, we concentrate on managing *server* replicas. Caching can then be done on top of a replicated system.

can initiate operations as *potential coordinators* or simply *coordinators*, while *node*, by default, signifies "a node having a replica".

Each node in the system is assumed to have a unique name. Nodes and communication links may fail, which may cause partitioning of the system. Failures are of *fail-stop* type, that is, nodes and communication links may crash but do not behave maliciously. Messages can be delivered out of order, or may not be delivered at all. If the sender of a message expects a response, it uses a local timeout to decide that the recipient could not be reached. However, the sender cannot tell whether the recipient failed or became disconnected due to a communication link failure. In fact, the sender cannot even tell whether the recipient actually received the message, since it might be the response (rather than the original message) that was not delivered. This model basically reflects the "at most once" semantics of RPC-style communication [6]. In fact, we will often refer to the timeout as a special `RPC.CallFailed` "response" to the sender. The failure detection mechanism described above relies on the availability of a local clock on every node.

A very important measure of fault-tolerance of a replicated system is the *data availability* it provides. Several slightly different ways to define data availability have been proposed. We will use for availability the fraction of time that all data requests from the user transaction initiated by a random operational site will be granted by the replica management protocol. In other words, we define availability as the probability that a user transaction will not be aborted by the replica management protocol provided the transaction was initiated by an operational node. A measure used by Jajodia and Mutchler [29, 31] differs in that it assumes that transactions can be initiated at a failed site as well as at an operational site. This measure appears to be less natural, because a user is not likely to choose a failed site to submit his/her transactions. In particular, under this measure, data availability of a replicated system is limited by the probability of an individual node being operational. Another definition of availability counts the fraction of time individual data items are accessible in any partition. This measure overestimates the level of fault-tolerance. Indeed, if a transaction accesses multiple data items, inaccessibility of any one of them would cause the transaction to abort, and it is the success or failure of transactions that ultimately matters for the user. We will discuss this topic

in greater detail in Chapter 6.

We distinguish between two kinds of write operation semantics, *total* and *partial*. In systems with total writes, a write always replaces the entire contents of the data item. In systems supporting partial writes, a write is allowed to update just a portion of information in the data item. For example, if we assume that the data items are files in the file system, then a write operation replacing an individual record is a partial write. On the other hand, if we consider data items at the granularity of records or disk pages, then the writes can be considered total.

Partial write semantics imply that a write operation can perform on current replicas only, whereas total writes can execute on stale replicas as well as on current replicas because all previous contents of the replica are always completely over-written. This makes protocols capable of supporting partial writes more complex. To concentrate on our main ideas, we will assume that writes are total except in Chapter 7, where partial writes are the focus of our research.

Finally, many algorithms in this thesis use the procedure `multicast(`V`, message)` to send an identical message to a set of nodes V. We make no assumptions about either the implementation of this procedure or the existence of a multicast facility for the network. However, the availability of such a facility would improve performance.

1.2 Quorums

In a system with strong consistency, applications should be unable to tell if data are replicated. While we will precisely define consistency later (in Section 2.4), it is intuitively clear that for applications to have an image of single-copy data, the following conditions are sufficient: (1) neither two write operations nor a read and a write operation on the same data item can perform concurrently, and (2) a read always returns the most recent version of the data, i.e., the data written by the previous write.[2] For example, if replicas of the same data item are split among several system partitions, due to the first condition noted previously, a write to this data item can succeed in at most

[2]Indeed, these conditions specify the multiple-reader-single-writer semantics characteristic of non-replicated systems.

one partition (otherwise, writes in two partitions could have succeeded independently and, hence, concurrently).

A common way to enforce these conditions despite node failures and/or network partitionings is by requiring that, in order to succeed, read and write operations obtain permission from certain sets of replicas, called *read* and *write quorums*. Quorums are defined in such a way that any two write quorums, as well as any read and write quorums, have at least one node in common. (We will refer to this property as the *intersection property* of quorums.) Obtaining permission from a replica involves locking that replica. Then, the first condition is enforced, because any two conflicting operations (i.e., read and write or two writes) would need to lock at least one common replica, and thus cannot perform concurrently. In addition, if a write is required to perform on all replicas from a write quorum, a subsequent read is guaranteed to contact at least one most recent replica. (This would be a replica that the quorums used by these two operations have in common.) This most recent replica can be identified using replica version numbers, thus satisfying the second condition for consistency.

Many different ways to define quorums have been proposed. For example, one may define read and write quorums to be any majority set or replicas of the data item. The resulting quorums are called *majority quorums*. Or, one may define a write quorum to include all replicas of the data item (this would be the only write quorum in the system) and read quorums to be sets that contain any single replica. These quorums are called *read-one-write-all*, or ROWA, quorums.

Obviously, any replica set that is a superset of a quorum is also a quorum, because it complies with the intersection property above. When reasoning about quorums, it is often convenient to think in terms of *minimal* quorums. A minimal quorum is one for which none of its proper subsets forms a quorum.

1.3 Transactions

The concept of *transaction* [5] has proved to be a convenient general abstraction for data manipulation in the presence of failures, especially in distributed systems. A transaction is a unit of work that exhibits the properties of *atomicity, serializability, and durability.*

It consists of primitive operations on the system state, such as reads and writes of the data items, and some computations that do not access the system state.

Atomicity means that, despite any sequence of failures during its execution, a transaction is either executed successfully (in which case it is said to *commit*) or not executed at all (in other words, it *aborts*). Since this property disallows partial executions of transactions, even if they are interrupted by failures, it becomes feasible to reason about a system's behavior under failures. The protocols that implement the atomicity property in distributed systems are called *commit protocols.*

Serializability means that, even if transactions are submitted concurrently, with their data accesses interleaved, the execution of these transactions is equivalent to some serial execution, where every transaction is submitted serially after the previous transaction terminates. In other words, every transaction T executes as if in isolation from any effects of other transactions that execute concurrently. Indeed, in the equivalent serial execution, these transactions are either completed before T begins or submitted after T terminates. The protocols that enforce serializability of transactions are called *concurrency control protocols.*

Finally, *durability* of transactions ensures that the effects of committed transactions will survive system crashes. This property is implemented by various modifications of the *write-ahead-logging protocol.*

1.3.1 Atomicity of Transactions in Distributed Systems

Part of the definition of transactions is that they are atomic, i.e., they execute in the all-or-nothing manner, even in the presence of failures. As noted, the protocols that implement the atomicity property in distributed systems are called *commit protocols.*

One example of a commit protocol is the well-known *two-phase commit protocol* [24]. In the simplest form of this protocol, the coordinator (usually the node that initiated the transaction) broadcasts the request for commitment to all participants (the sites containing the data items involved) and waits for the participants to respond as to whether they can commit the transaction locally. Upon receiving a request for commitment, each participant can either enter the *waiting* state, vote to commit, and wait for further

instructions, or vote to abort and abort the transaction without waiting. After collecting all votes (timeouts being counted as votes to abort), the coordinator decides if the whole transaction can be committed (permission from all participants is required) and broadcasts the commit or abort messages accordingly. A participant that receives the commit (abort) message commits (aborts) the transaction locally, thus completing the protocol.

Observe that, upon sending the vote to commit, a participant relinquishes the ability to terminate the transaction unilaterally and must wait for further instructions from the coordinator to learn whether the transaction can be committed or must be aborted. During this time, any further accesses to the data items involved in the transaction are usually blocked (see the description of the strict two-phase locking concurrency control protocol in the next subsection).

Moreover, if a site fails while in the waiting state, upon recovery it needs to find out (usually from the coordinator) how to terminate the transaction.[3] Therefore, the coordinator cannot just forget about the transaction after sending out the commit message. It must retain the record of a committed transaction until all participants send acknowledgments that they committed the transaction. On the other hand, if the coordinator decides to abort the transaction, it can delete records about the transaction immediately after sending out the abort message: if a participant later inquires about the outcome of this transaction, the fact that there is no record of it tells the coordinator that the transaction was aborted. Indeed, if it were committed, a record of the outcome would have been kept until all participants acknowledge committing the transaction. In this case, however, no participant could have inquired about the outcome of the transaction (because this inquiry is done only by participants in the waiting state).

This variation of the two phase commit protocol is called *presumed abort* [46], because the coordinator assumes that the transaction is aborted when it has no record about its outcome. The dual variation, *presumed commit*, is also possible [46].

Commit protocols add overhead to transaction management by requiring two rounds of message exchanges between the coordinator and the participants. One way to reduce

[3]Obviously, to do this, the node needs to remember that it was in the waiting state at the time of failure. This is done by requiring that all nodes log the fact of entering a new state in stable storage.

this overhead is to treat the acknowledgements of the individual reads and writes from the participants as their vote to commit the transaction. Then, when the coordinator is ready to commit the transaction (i.e., all individual operations have performed), it can send out the commit message immediately. This scheme, called *early prepare* [68], has lower performance overhead at the expense of an increased vulnerability window, during which a participant cannot terminate the transaction unilaterally. In the previous scheme, this window existed only between two consecutive phases of the commit protocol; with early prepare, it starts when the participant acknowledges the read or write (which may happen at the beginning of the transaction execution) and ends at the end of the transaction. So, the risk increases of data being blocked due to the failure of the coordinator or network partitioning.

For more details on the tradeoffs between variations of the two-phase commit protocol, see [46, 61].

1.3.2 Serializability of Transactions in Non-Replicated Systems

Serializability of transactions in a non-replicated system is ensured by *concurrency control protocols.* The most common concurrency control protocol is the *strict two-phase locking (S2PL) protocol* [5]. In this protocol, each data object has an associated read and write lock. The lock granting rule is that different transactions cannot hold the write lock or the write and the read lock of the same object at the same time; however, sharing the read lock among multiple transactions is allowed.

Before a read (write) operation can perform on the object, its read (write) lock must be obtained. Moreover, any locks obtained by the transaction in the course of its execution are released only after the transaction commits or aborts.[4] Thus, the transaction execution can be divided into two phases. During the first phase, the transaction accumulates locks as it accesses more and more data; during the second phase, it releases the accumulated locks.

In fact, if the only purpose were to ensure serializability of transactions, the strict two-phase locking protocol would be too restrictive. However, it provides other important

[4]This is why blocking of a participant during the execution of the commit protocol entails blocking accesses to the data items involved, as mentioned in the previous subsection.

properties, such as avoiding cascading aborts of transactions. Refer to [5] for further details.

Observe that this protocol is prone to deadlocks. For example, if transactions T_1 and T_2 both access data items x and y, and T_1 locks x while T_2 locks y, neither transaction will be able to complete. One simple way to handle the possibility of deadlocks is to abort any transaction that does not complete within a specified time. For other methods, see, for example, [5]. We do not consider the deadlock problem in this thesis any further. This problem is orthogonal to the ideas described in this thesis: any existing way to handle deadlocks can be incorporated with the protocols presented here.

1.3.3 Serializability of Transactions in Replicated Systems

In a replicated system, we need to distinguish *logical* operations on *logical* data items issued by applications from *physical* operations on individual replicas of those data items performed by the system. If locks are associated with individual replicas of data items, then the concurrency control protocol is executed on the level of physical operations on individual replicas. Thus, transactions are serialized on the level of physical operations on individual replicas, but not necessarily on the level of logical operations.

As an example, consider a system maintaining two data items, x and y, each represented by two replicas, x_1, x_2, y_1, y_2. Let transaction T_1 read data items x and y and write x, and transaction T_2 read x and y and write y. To make the system fault-tolerant, we want to allow operations on logical data items to succeed without having to perform on every replica of those data. Assume, for example, that the system implements the logical read of x and y issued by T_1 by reading from replicas x_1 and y_2, respectively, and it implements T_1's logical write to x by writing x_1. Furthermore, assume that the system implements the logical reads of T_2 by reading from x_2 and y_2, and it implements T_2's logical write to y by writing y_2.

Under replica-level concurrency control, it is possible to execute, say, T_1 first and then T_2. Obviously, this execution is serializable on the replica level (in fact, it *is* serial). However, it is not serializable on the level of logical operations. Indeed, T_1 reads the value that y had before T_2 performed, so T_1 must have occurred *before* T_2 in any equivalent

serial execution. Similarly, T_2 reads the value that x had before T_1 performed, so T_1 must have occurred *after* T_2 in any equivalent serial execution, a contradiction. Thus, intuitively, from the application's point of view, this execution is incorrect.

Therefore, in a replicated system, we need to re-formulate the serializability property of transactions in terms of operations on logical data items. Namely, concurrent execution of transactions on replicated data must be equivalent to some serial execution of these transactions on non-replicated data. To distinguish this property from serializability on the level of physical replicas, it is called *one-copy serializability* [5].

To see how one can implement one-copy serializability, recall that the quorum-based replica control protocol can be used to make logical operations on replicated data indistinguishable from those operations on non-replicated data. Then, if we associate locks used by the concurrency control protocol with logical data items rather than physical replicas and execute S2PL on the level of logical operations (in addition to quorum-based replica control protocol), we would guarantee one-copy serializable executions only.

However, if we simply assign a lock-granting server to every logical data item, we will defeat the purpose of having multiple replicas. For instance, if the node fails on which the lock server for some data item resides, no application can access this data item. Hence, fault-tolerance of the system will be lost.

Fortunately, we can employ the locks used by the replica control protocol to *emulate* the logical-level locks for the purpose of concurrency control. Indeed, it follows from the intersection property of quorums that if a (logical) write obtains permission (and hence, write locks) from a write quorum of replicas, no other logical write operation can do so until the first write starts releasing its locks. The same is true for a logical read and a logical write operation. In addition, a read always returns the most recent data, i.e., the data written by the write that locked its quorum directly before the current read did. Hence, the execution of user transactions in the system is equivalent to the execution of the same transactions over non-replicated logical data items, with the following locking rule: a data item is logically locked by an operation if this operation locks all physical replicas of this item from a quorum. The data item is logically unlocked when the operation unlocks any replica of the data item from the quorum collected.

Then, if we use S2PL on the level of physical replicas employing the same locks as

the ones used by the quorum protocol, transactions would start releasing any replica locks only after they commit or abort. According to the above rule for logical locks, this means that logical data items are unlocked under the logical locking rule also only after transactions terminate. But this behavior specifies two-phase locking of logical data items. Hence, the execution of transactions over logical data items is serializable, and the execution of transactions over physical replicas is one-copy serializable.

Thus, we showed that one-copy serializability can be achieved if the system uses the quorum protocol for replica control and S2PL on the level of physical replicas for concurrency control, with both protocols sharing the same locks. In fact, one can show that the quorum-based replica control protocol can be used in combination with *any* correct concurrency control protocol executed on the level of physical replicas[5] to achieve one-copy serializable execution [5]. This is also true for most of the protocols described in this thesis. However, since S2PL is by far the most frequently used concurrency control scheme, we will often limit our proofs to show that our protocols can be used in conjunction with S2PL to achieve one-copy serializability.

1.4 Thesis Contributions

This thesis concentrates on the quorum-based approach to replica management and makes the following main contributions to the area.

- *Improving data availability provided by protocols using structure-based quorums; studying in depth properties of a particular class of these quorums.*

  In quorum-based protocols, successful read and write operations must obtain permission from a read or write quorum of replicas, which causes a performance penalty due to communication with multiple nodes on every operation. To minimize this penalty, it is desirable that the quorums be small. Structure-based protocols (e.g., [33, 8, 3]) achieve this by defining quorums based on a logical structure imposed on the network. Unfortunately, these quorums have lower data availability than majority quorums. We propose in this thesis a method to improve

[5] In other words, any protocol that ensures serializability of transactions on the level of physical operations.

the data availability of these protocols by changing the logical structure dynamically to reflect the failures and repairs in the system. The analytical availability estimations show very significant improvements over the existing scheme.

- *Proposing a method for asynchronous quorum adjustment to reflect changes in the system topology.*

  Many schemes have been proposed for dynamically adjusting quorums to exclude and include nodes as they fail and repair. Protocol correctness in existing schemes requires that switching to new quorums be mutually exclusive with user accesses to data. Quorum adjustment may then interfere with and delay user operations. On the other hand, a long user transaction may delay quorum adjustment indefinitely. This thesis proposes a way to adjust quorums completely asynchronously with user operations.

- *Proposing a method for effective management of replicated data based on read-one-write-all quorums.*

  Replica management based on read-one-write-all quorums (called ROWA schemes) is especially attractive, because it provides for efficient and highly available read operations. Indeed, reads can be performed by the server closest to the application's node, and processing of a read request is carried by a single replica, which provides for very effective load sharing. This makes it suitable, for example, for maintaining repositories and directories across a wide-area distributed database. It is also a desirable scheme in systems in which only two replicas of data are maintained.

  An obvious weakness of ROWA, which has prevented it from being widely used in commercial wide-area networks, is low data availability for write operations. Considerable effort has been devoted to addressing this problem. However, existing proposals solve it at the cost of compromising some of the good properties of the read operations. In this thesis, we propose a new protocol that achieves better overall data availability than the existing solutions, and does so without any significant deterioration of read properties. We believe this solution should enable a

wide practical usage of the ROWA scheme.

- *Studying the effect of the semantics of write operations on the performance of the quorum-based protocols; proposing an efficient method to support writes with partial update semantics.*

  We observe that system's write operation semantics significantly affect the performance of replica management protocols. Specifically, existing quorum-based protocols suffer additional penalties when they must support write operations that update a portion of the information in the data item rather than entirely replacing an old version of the data item. We call these operations *partial* writes. File systems are examples of systems with partial writes. We propose in this thesis a protocol designed specifically for supporting partial writes that outperforms current alternatives.

- *Studying the influence that a replica management scheme may have on other aspects of transaction management.*

  Two aspects of transaction management can restrict accesses to data: the replica control protocol (to prevent the possibility of replica divergence), and the commit protocol (to avoid inconsistent transaction termination on different nodes). If these two aspects are unaware of each other, they can restrict accesses to disjoint sets of data items, which results in the diminished overall availability of data for accesses. We propose in this thesis a way for the commit protocol to take into account the presence of replication when choosing a partition for transaction termination.

Overall, the main purpose of this thesis is to propose ways to improve performance and data availability in distributed replicated systems that require strict consistency. Some methods directly address the data availability problem; others concentrate on performance. Both aspects, however, are closely interrelated. For example, if a method improves the data availability of the system, then fewer replicas are needed to achieve the same level of availability; therefore the performance overhead of replication will be lower. Thus, these aspects are two sides of the same problem this dissertation addresses, that of *efficient replication management in distributed systems.*

Chapter 2

Replica Management: Tradeoffs and Issues

In this chapter, we take a high-level view of a sample of existing replica control protocols and examine the tradeoffs among various approaches to replica management and between replicated and non-replicated systems. The purpose of this chapter is not so much to provide a survey of existing work (more existing protocols are surveyed in the following chapters), but to give a sense of the main issues and problems in the design of a replica management scheme.

First, we look at the issue of data availability, the most common motivation for data replication. In most protocols, data availability increases with the number of replicas. But the level of availability achieved for a fixed number of replicas differs for different protocols. In general, protocols with strong consistency guarantees (i.e., ones that provide a one-copy image of data to the application) achieve lower availability than those allowing inconsistent executions. We illustrate this basic tradeoff by comparing two protocols, one with unconditional consistency guarantees and one that guarantees consistency in the absence of failures only. While research has been done to improve availability of protocols with strong consistency, these protocols still cannot challenge the availability levels of protocols with weaker consistency guarantees.

Another motivation for replicating data is to improve system performance relative

to non-replicated systems. Indeed, in replicated systems, operations can be serviced by nearby replicas, while in non-replicated systems, all operations on a data item go to the same node that stores that item. Again, the best results are achieved by protocols that allow inconsistent executions. However, some progress has been made in designing protocols with strong consistency that *in some cases* provide better performance than non-replicated systems. We examine these approaches and observe their limitations.

Finally, we examine the overhead incurred by systems due to data replication. The nature of this overhead is twofold. First, additional storage is needed for multiple copies of data items and control information used by replica management protocols themselves. Second, to satisfy consistency constraints, these protocols involve communicating with multiple replicas of a data item during execution of an operation on that item. Thus, message traffic in the system increases. We look at some approaches to reduce the costs of replication and discuss the tradeoffs involved.

The chapter concludes with a section containing information from serializability theory. Besides giving a glimpse of the theoretical side of replica management research, this section provides the notation and tools for proving the correctness of protocols presented in the subsequent chapters.

2.1 Gains in Availability

2.1.1 Optimistic vs. Pessimistic Approaches

The main motivation for having multiple copies of a data item is to improve availability of data, i.e., the probability that an operation on a given data item will be successful: when some replicas of the data item fail, the operations on this item can still be serviced by the remaining functional replicas.

If the system becomes partitioned due to communication failures, the replicas in different partitions may diverge, with later reconciliation being difficult or impossible. Therefore, it is appealing to require a replicated system to always maintain a consistent view of its data. In these systems, replica control protocols must provide a one-copy image of the data to the users, so that the applications cannot tell whether the data are

replicated. Systems and protocols that satisfy this requirement are called *pessimistic*, because they are willing to incur the overhead of maintaining data consistency even in situations that are unlikely to happen.

Pessimistic protocols can be implemented using a general scheme based on the concept of quorums. Recall from Section 1.2 that quorums are sets of replicas of a data item such that any two write quorums, as well as any read and write quorums, have at least one replica in common.

In this scheme, every replica of a data item maintains a variable *replica version number*, VN, initially 0, and a read and write lock, that can be granted locally according to the standard multiple-readers-single-writer-rule.[1]

To read a data item, the client chooses a read quorum of replicas of the data item and sends a read permission request to these replicas. When a server receives this request, it obtains the local read lock for its replica and replies with permission along with requested data and its replica version number. Upon obtaining permission from a read quorum of replicas, the client uses the data from a reply with the maximum version number.

To execute a write, the client sends a write permission request to a write quorum of replicas. On receiving this request, a server obtains the local write lock for its replica and replies with permission and the replica version number. When the client gets permission from a write quorum of replicas, it determines the maximum version number, VN_m, among all responses (which is the version number of the most recent replicas before the current write performs). It then calculates the next version number of current replicas to be equal to $VN_m + 1$. Finally, the client distributes the write data and the new version number to at least a write quorum of replicas, using a two-phase commit protocol [5] to ensure all-or-nothing execution.

Since any two write quorums have a common replica, if several writes on the same data item are executed concurrently, only one write at a time can successfully collect a write quorum of permissions. Therefore, concurrent writes on the same data item are serialized. For the same reason, should the network partition, it is not possible for writes to succeed simultaneously in different partitions, and thus divergence of replicas cannot

[1] This rule precludes multiple processes from holding the write lock or the write and read locks at the same time, but it allows multiple processes to hold the read lock simultaneously.

occur. Similarly, a non-empty intersection between every read quorum and every write quorum serializes any pair of operations consisting of a read and write and guarantees that any successful read operation contacts a replica written by the most recent successful write.

One of the first quorum-based protocols was Gifford's *voting* protocol [18]. The protocol is, basically, a specialization of the general scheme just described, with the following quorum definition. Let every replica of a given data item be assigned some number of votes, with the total number of votes of all replicas of the data item equal to v. Two numbers, a read and write *quorum threshold*, r and w, are chosen, such that $r + w > v$ and $2w > v$. A read (write) quorum is any set of replicas with the total number of votes at least r (w). It is easy to verify that, with this definition, intersection property of quorums holds.

When discussing Gifford's protocol, it is convenient to refer to the simplest case in which all replicas are assigned a single vote, and both read and write quorum thresholds are defined to be a majority of replicas. Generalizations to arbitrary vote and quorum threshold assignments are usually straightforward.

The voting protocol fulfills its main goal of improving data availability in the presence of site failures without compromising data consistency in the case of network partitionings. Indeed, by increasing the number of copies of a data item, availability of this data item is increased arbitrarily close to 1, for any fixed value greater than 0.5 of the probability of a replica being operational. However, with the number of replicas increasing, the space and performance penalty incurred by the protocol also increases. In particular, since every operation involves communicating with a majority of replicas, the number of messages exchanged during execution of the operation increases linearly with the number of copies. We discuss performance implications of this protocol in more detail in Section 3. Here, with the focus on availability, we observe another shortcoming of the voting protocol: when the operational replicas become a minority (due to site failures or network partitionings), access to a data item is denied, despite the fact that there still may be many functional replicas that contain the current version of the data.

One way to address this problem is to relax the consistency requirements that the protocol must satisfy. Systems that allow inconsistent executions are called *optimistic.*

In the Coda replicated file system [66], the replica control protocol guarantees data consistency in the absence of failures, but, when the network partitions, it allows the copies in different partitions to diverge. Every client in Coda maintains a list of replicas with which it can communicate, the *accessible volume storage group (AVSG)*. This group may enlarge or shrink as a result of periodic probing of the complete set of replicas, the *volume storage group (VSG)*. The shrinking of the AVSG can also occur as a side effect of a normal file operation if this operation fails to contact all AVSG members.

Every server that keeps a replica of the file maintains state information that allows the system to detect inconsistencies among replicas. For the purpose of our discussion, we can view this state as a complete update history of the replica that logs unique operation IDs of all writes to the file performed on the replica.[2]

A client performs a read by contacting all members of the client's AVSG and indicating its *preferred server* (usually the closest replica among members of the AVSG). All contacted servers respond with their update histories, while the preferred server also sends the data. On receiving all responses, the client uses the reported update histories to check if the preferred server does indeed have the latest copy of the file. If this is *not* the case, a server with the latest copy is made the new preferred server, and the data item is refetched from it.

During a write operation, the client distributes a new update history and data to the members of its AVSG. (For an existing file, a new history can be constructed by appending the current write operation ID to the history obtained as a result of the previous read, write, or open operation; for a new file, a history consists of the current write ID.) Each server first checks if its update history is consistent with the history received, in which case it atomically performs the operation and updates its history. If an inconsistency is found, an automatic conflict resolution subsystem is invoked in an attempt to reconcile the copies. For data files, virtually no automatic conflict resolution is possible. However, for directory files, conflicts can sometimes be resolved by a compensating sequence of create-file and delete-file operations. If the resolution attempt fails, the write operation

[2]In reality, since maintaining complete histories is impractical, Coda uses a compact conservative approximation. The approximation is conservative because, while it may indicate a conflict among replicas that are actually consistent, it never fails to detect a conflict among inconsistent replicas.

is aborted.

The Coda replication scheme provides better file availability than the voting protocol. Indeed, there is no requirement that a successful operation contact a majority of servers: a file operation can succeed as long as it can contact a single replica. This improvement in availability became possible because the priority in Coda was shifted from providing unconditional consistency to detecting replica conflicts as early as possible. Indeed, in the absence of failures, all clients' AVSGs include all replicas of the file in the system, and every operation successfully contacts all replicas. Therefore, every successful read returns the most recent copy of the data, and every successful write leaves all replicas in a consistent state. However, if the network partitions, the clients in different partitions may independently access and modify the replicas of the same file in their partitions, and inconsistencies may arise. By periodically probing VSGs for all files they cache, clients can detect inconsistencies for all active files within a specified time after the partitioning is repaired. However, resolving the replica conflicts often requires human intervention and may be difficult.

The choice between optimistic and pessimistic approaches is a fundamental design issue in replicated systems that affects all other decisions. As the two described protocols illustrate, there is a direct tradeoff between consistency guarantees and data availability. The decision must be made based on workload properties and the anticipated pattern of system usage. For example, if an application involves irreversible external actions (e.g., dispensing cash by ATMs), an optimistic strategy is probably not appropriate.

The intended use of Coda is to provide a shared file repository in a distributed workstation environment where the primary activities are education, research, and software development. The authors argue that, since write sharing between users is relatively infrequent in an academic environment, the possibility of conflicts even in the presence of failures is remote. Therefore, it is acceptable to adopt an optimistic approach.

However, note that while write sharing among users is indeed infrequent, it is often the case that the *same* user moves from one workstation to another, possibly crossing partition boundaries. In this case, a situation may arise in which the user works on one part of his/her program file, moves to another workstation (in a different partition), works on another part of the same program file (not noticing that the file is stale),

and later, when the partition repairs, is presented with two conflicting versions of the program. Conflict resolution in this case may involve extensive labor costs. Moreover, the fact that situations like this would occur infrequently may aggravate the problem by lowering a programmer's alertness.

Thus, an outright optimistic approach in a replicated file system appears to involve too high a price to pay for better data availability. Instead, a mixed approach – with an optimistic strategy used to manage files that allow automatic or easy conflict resolution and a pessimistic strategy employed for all other files – seems more appropriate. Such an approach is adopted in the Echo replicated file system [26]. In this system, high-level directory files are kept by the name service, which uses an optimistic replication scheme, while more volatile lower-level directories and data files are kept in the file system, which uses a pessimistic scheme.

2.1.2 Highly Available Pessimistic Protocols

In the previous subsection, we discussed one way to obtain better availability than that of the voting protocol, namely, by relaxing consistency requirements. Here, we examine another approach, which improves data availability while retaining strong consistency guarantees.

This approach involves adjusting read and write quorums dynamically to reflect failures and repairs occurring in the system. In this case, the data item remains available as long as there is at least one operational replica, provided not too many failures accumulate undetected so that the protocol can adjust to the failures as they occur. This approach compares favorably with the voting protocol, in which a data item is unavailable unless a majority of its replicas is up.

Many dynamic quorum adjustment protocols have been proposed (e.g., [4, 25, 29, 49]). We will discuss this approach using the *dynamic voting protocol* [29] as an example. As in Gifford's voting protocol (called "simple voting" below), consistency in the dynamic voting protocol is maintained by making sure that, if the network partitions, file updates can succeed in at most one partition (the *distinguished partition*). In simple voting, the distinguished partition was defined as the one with more than half of *all* replicas of

the file. In dynamic voting, the distinguished partition is defined as the partition that contains a majority of *current* replicas.

To decide which partition (if any) has a majority of current replicas, the protocol uses two variables associated with every replica: a replica version number, VN, (as in simple voting) and an additional variable, *update cardinality*, SC. Originally, all replicas have a version number 0 and update cardinality equal to the total number of replicas of the file in the system. Then, for a replica i, update cardinality SC_i stores the number of replicas that participated in the most recent write operation involving replica i. In any case, if i is a current replica, its update cardinality indicates the number of current replicas in the system.

When a client s wants to perform a write, it sends a write permission request to all replicas of the file. Upon receiving this request, a server belonging to the partition P to which s currently belongs obtains the local write lock for its replica and responds with permission, which includes the replica's version number and update cardinality. From the replies, s learns the highest version number, VN_m, among the replicas in partition P, and the update cardinality, SC_m, of replicas with that version number. Partition P is distinguished if it contains more than $\frac{SC_m}{2}$ replicas with version number VN_m.

If P is the distinguished partition, the client broadcasts to all members of P a message that includes the new value of the data item, new version number (equal to $VN_m + 1$) and new update cardinality (equal to the number of members in P). Each site that receives this message atomically installs the new value of the data item and adopts the new version number and update cardinality. A two-phase commit protocol is used to ensure that the write operation is performed in an all-or-nothing manner.

To perform a read, a client sends a read permission request to all replicas. In response, a server obtains the read lock for its replica and sends back its permission, together with the replica's version number and update cardinality. The client determines whether it is in the distinguished partition and then reads data from a replica with version number VN_m.[3]

By recording the new value of update cardinality, every write operation adjusts the

[3]In some cases, data can be piggybacked on the permission message sent by replicas, thus avoiding the need for an additional round of message exchange.

read and write quorums to be a majority of replicas on which that write is executed. To see why this improves availability, consider the example of five replicas (A, B, C, D, and E) of a data item, all of which originally have a version number 0 and an update cardinality 5. Assume the system partitions into $P = \{A, B\}$ and $Q = \{C, D, E\}$. In both simple and dynamic voting, Q is the distinguished partition (and the data item is available there), while P is not. Assume a write w is executed in partition Q making the version number of its members 1 and the update cardinality 3 (the number of replicas on which w performs). If Q partitions again into $Q_1 = \{C\}$ and $Q_2 = \{D, E\}$, under simple voting, no partition qualifies as distinguished; under dynamic voting, however, Q_2 is the distinguished partition, and the data item continues to be available there. A similar situation occurs if, instead of partitioning, first sites A and B, and then site C fail.

Interestingly enough, there are situations in which data accesses can be granted under simple voting but not under dynamic voting. Continuing our example, suppose that after write w is executed, sites D and E fail, and site C reconnects with A and B to form partition $P_1 = \{A, B, C\}$. This partition is distinguished under simple voting (since it contains a majority of all replicas) but not under dynamic voting (because only a minority of current replicas belong to it). Thus, the data item is available in P_1 under simple voting, but it is not available in any partition under dynamic voting. Nonetheless, as shown in [29], if the number of replicas exceeds 3, the availability of dynamic voting is strictly better than that of simple voting. For exactly three replicas, if we assume that transactions cannot be initiated by failed sites, dynamic voting, again, yields better availability [42].

No matter how sophisticated pessimistic protocols may be, they have one fundamental property that does not allow them to reach the availability levels of optimistic protocols: pessimistic protocols can allow file updates to perform in at most one partition at a time. To ensure this property, these protocols sometimes disallow updates in *all* partitions. For instance, in the above example, when the system partitions into P, Q_1, and Q_2, the simple voting protocol disallows access to the data item in all three partitions. In the same situation, the (optimistic) Coda file system does not block data accesses in *any* of the partitions, with the hope that, since write sharing of data is in-

frequent, data would be actually modified in only one partition and conflicts would not occur. However, as discussed in the previous subsection, conflicts may occur not only as the result of write sharing among users, but because the same user moves from one workstation to another, and these conflicts may be very costly.

2.2 Gains in Performance

Besides improving availability, replicating data items across the network can also potentially improve system performance. There are three ways in which replication can provide better performance relative to non-replicated system.

First, when a data item is replicated, operations on it can be serviced by replicas that are "close" to the clients, with lower communication costs. This contrasts with non-replicated systems, in which operations from all clients must always go to the same server.

Second, in replicated systems, a workload can sometimes be shared among replicas, which can reduce server utilization and operation response time.

Finally, with replication, one can build reliable systems from not-so-reliable elements. Therefore, it is possible to replace a slow but reliable storage device with a set of fast but less reliable devices, attempting to achieve through replication the same overall reliability and better performance.

An example of the last approach is the Harp file system [41]. Harp is a variant of the NFS distributed file system [62], where a single file server with a disk is replaced by a set of file servers with battery-backed volatile memories (BBM) and disks, each of which has a replica of the data. Once a write is executed in the BBM of a majority of servers, it is considered stable, and control returns to the client. On every server, the write is then applied to disk in the background. The increase in the write operation's communication costs is more than offset by removing a disk access from the critical path of the write operation.

The other two approaches are discussed in greater detail below.

2.2.1 Operating on "Nearby" Replicas

Servicing an operation by servers that are close to the client is an easy way to improve performance when consistency requirements are weak. For example, in the Ficus replicated file system [52], an operation is always performed on one replica only, with the consequent asynchronous update propagation to other replicas done after control is returned to the client. In this approach, clients can always choose "nearby" replicas for submission of their operations, thus minimizing communication costs. However, this approach is extremely optimistic. Indeed, while the Coda system at least guarantees consistency in the common case of the absence of failures, Ficus provides no consistency guarantees and operates on a "best effort" basis. As an example, if client A executes a read-modify-write of an object x on replica x_1, and client B later executes the same operation on replica x_2, then, if B's operation reaches x_2 before x_2 asynchronously acquires A's update from x_1, a conflict will arise.

When stronger consistency guarantees are required, especially in pessimistic systems, it is much harder to take advantage of nearby replicas for performance improvement. For example, in the simplest case of the voting protocol, a client has to contact a majority of replicas on every operation. Thus, not only can it not avoid contacting distant replicas, but it incurs the additional penalty of communicating with multiple servers. Communicating with multiple replicas cannot be avoided if consistency is to be maintained. Thus, we observe a tradeoff between consistency guarantees provided by the system and its performance.

While pessimistic protocols cannot take advantage of nearby replicas to improve performance in the general case, these protocols can exploit certain assumptions about the system workload to achieve better (than non-replicated systems) performance in the common case. For example, in the voting protocol, one can set the read quorum threshold to 1 and the write quorum threshold to v, where v is the total number of votes of all replicas of the data item. This yields the *read-one write-all (ROWA)* protocol, in which a read is accomplished by contacting a single replica and a successful write must perform on all replicas of the data item. A read can then be serviced by a single nearby replica, while writes become very expensive (and, in addition, vulnerable to the failure of a single

replica).[4] If the workload has many more reads than writes, this arrangement can yield significant performance improvement over non-replicated systems. On the other hand, if this assumption about the operation mix proves wrong, the performance can be much worse than that of a non-replicated system.

Different assumptions about a workload are used in Triantafillou's protocol [69]. (A similar idea was independently proposed in [2].) This protocol exploits the notion of a *transaction* to improve system performance. A transaction is a unit of computation guaranteed to be done either completely or not at all. The authors assume that a transaction usually contains many accesses to data items executed sequentially.[5] Another assumption is that data sharing, which can cause conflicts, is infrequent.

Given these assumptions, an individual operation in Triantafillou's protocol is executed on the nearest replica (among those available), with subsequent asynchronous update propagation to other replicas. As the operation propagates in the network, the transaction locks the replicas. At transaction commit time, the protocol checks for any possible conflicts with other transactions. For every read (write) operation included in the transaction, the protocol waits to make sure that the operation has propagated to a read (write) quorum of replicas of the data item. After waiting sufficiently long, if any operation in the transaction fails to collect necessary votes, the protocol concludes that either a conflicting transaction locked replicas needed to complete the quorum, or those replicas are partitioned. In both cases, the whole transaction is aborted to avoid conflicts or replica diversion. Thus, for all committed transactions, consistency is guaranteed.

If the assumption holds that conflicting data sharing among transactions is infrequent, the fraction of aborted transactions will be low, and overall performance can significantly improve relative to non-replicated systems. Indeed, in a successful transaction, most individual operations will usually have executed on distant replicas asynchronously by the commit time, and so the waiting period during a commit will be short. Then, in most cases, the response time of the transaction will be determined by the latency

[4]The fault-tolerance problem of read-one-write-all replicated data is addressed in detail in Chapter 6.

[5]In database systems, the operations packaged into a transaction are usually determined by the semantics of the application. For example, a funds transfer between two accounts defines a transaction that contains two operations: withdrawing money from one account and adding the same amount to another. In research environment file systems, the operations can be packaged into transactions arbitrarily depending on how much work the application is willing to lose in the event of failures or replica conflicts.

of operations performed on nearby replicas. However, if the rate of aborts becomes significant, overall performance will degrade, due to time and resources wasted on partial executions of aborted transactions.

Another assumption necessary for this protocol to outperform non-replicated systems is that transactions contain chunks of work large enough to give most operations time for asynchronous update propagation. As an extreme, if we consider the case where every transaction contains a single operation, then, at transaction commit time, the protocol must always wait for the transaction's sole operation to perform on a quorum of replicas. Therefore, the protocol would exhibit similar performance to Gifford's voting protocol, which as discussed previously, has generally worse performance than non-replicated systems. In fact, in this case, Triantafillou's protocol will perform worse than Gifford's protocol, because an operation in the former is usually performed in two sequential stages: first on a nearby replica, and then on other replicas, with the nearby replica acting as the coordinator.

Triantafillou's protocol belongs to a family of protocols that, instead of preventing conflicts, allows them to occur and relies on conflict detection and abortion of offending transactions to maintain consistency. Such protocols are called *aggressive* [5]. (They are sometimes called optimistic, but we like to reserve this term for protocols that allow inconsistent executions.)

To conclude, we reiterate that both read-one write-all and Triantafillou's protocols exploit certain assumptions about a workload to optimize performance for the common case. This is achieved at the expense of performance degradation in situations where the assumptions do not hold. Only optimistic protocols can achieve clear-cut performance advantage over non-replicated systems regardless of workload characteristics.

2.2.2 Load Sharing

Another source of potential performance advantage of replicated over non-replicated systems is the possibility of sharing the workload among multiple replicas of the data item. Indeed, in Gifford's protocol, every operation must be performed on roughly half the replicas, so, on average, it appears that each replica would have to carry only about

half the load of a non-replicated server. However, as we show below, this is not always the case, and the degree of performance advantage that can be expected in replicated systems depends on the semantics of write operations used in the system.

Two different models of writes are common. In database systems, write operations are often assumed to always replace the whole data item by installing an entirely new value. We call this model *total* writes. Another model, prevalent in file systems, is that of *partial* writes, where a write can update a portion of information in a data item rather than replacing it entirely.

The key difference between these semantics is that, in a system with all total writes, a write can perform on current as well as on obsolete replicas (the whole replica will be replaced with a new value anyway), while in a system with partial writes, a write can be applied to current replicas only.

In Section 1.2, we saw that maintaining consistency in quorum-based protocols requires executing a write on at least a write quorum of replicas. In a system with partial writes, this implies that, once a write performs on a quorum of replicas, all subsequent writes must be executed on the same set of replicas, since all other replicas will be obsolete [18]. Thus, in these systems, the load due to write operations is not shared. Still, any read quorum of replicas can be used for a read operation, and so the read portion of a load is shared. In systems with total writes, both read and write portions of a load can be shared, and one can expect greater performance improvement over non-replicated systems in this case. We discuss performance implications of write semantics in greater detail in Chapter 7.

Note that the issue of partial vs. total write semantics is, in fact, the issue of data granularity. If replication in a file system is managed at the level of blocks rather than files, then all writes become total. Managing replication at the block level would therefore improve load sharing in the system. On the other hand, this approach would increase the overhead for replica management, e.g., because the replica control state would have to be maintained per block instead of per file. Also, the presence of variable length physical blocks may complicate the implementation. Only the direct performance comparison of both implementations would conclusively tell which approach is better. Currently, this is an interesting open question.

Another observation with respect to load sharing is that, in the majority voting protocol, the number of replicas participating in every operation is no less than half the total number of replicas. This imposes a bound on the achievable degree of load sharing: regardless of the number of replicas, the average load carried by a replica can be cut by at most half. To move beyond this bound, we need quorums whose size would grow slower than linearly with the total number of replicas. Simply reducing quorum size by using non-uniform vote assignments in the voting protocol does not help. Indeed, to distribute the load evenly, all read (write) quorums must be of the same size, and every replica must participate in the same number of read (write) quorums. Quorums that satisfy this condition are called *fully distributed.* In fully distributed voting-based quorums, all write quorums contain at least $\lceil \frac{N+1}{2} \rceil$ replicas, where N is the total number of replicas. Thus, at least the write portion of an average load carried by a replica cannot be decreased in the voting protocol by more than half regardless of vote assignment and the values of quorum thresholds. This problem partially motivated a search for "smaller" quorums whose definition is not based on voting. We will see an example of such quorums in the next section.

2.3 Overhead of Replication

Two inherent sources of overhead are incurred by systems due to data replication. First, additional storage is needed to keep multiple copies of data. Second, in pessimistic systems, communication with multiple servers during operation execution is required to maintain data consistency.

Replication overhead diminishes the performance improvement that replicated systems can achieve; in many cases, it brings the performance of these systems below the level of non-replicated systems. When improving availability is the primary goal of replication, it may be acceptable to sacrifice some performance, but even then we would like to minimize the performance penalty.

2.3.1 Reducing Storage Overhead

An interesting method to significantly reduce storage overhead with only minimal degradation of data availability was proposed by Paris [50]. His method uses the concept of a *witness*, a "fake" replica that stores the control state (e.g., version number and locks in the majority consensus protocol) and participates in quorum formation, but does not keep actual data. Clearly, witnesses require much less storage than "normal" replicas. In the case of a system with N replicas of a data item and majority quorums, the method allows replacing up to $\lfloor \frac{N-1}{2} \rfloor$ replicas with these inexpensive witnesses. Then, as long as any quorum can be collected, it will include at least one "normal" replica, which explains why the reduction of data availability is very small. At the same time, the storage overhead is reduced by almost half. The witnesses can be used in combination with many other ideas in replica control, including dynamic voting and some of the protocols proposed in this thesis.

If network partitionings occur infrequently, another scheme, called the RADD protocol (for Redundant Array of Distributed Disks) [67], can reduce space overhead even more. This scheme involves splitting the data item into m data fragments of equal size and maintaining a special *parity* fragment of the same size. Each fragment (including the parity one) is kept on a separate node. The parity fragment is calculated by taking a bit-wise exclusive-OR (XOR) operation on all data fragments. Whenever a data fragment is modified, the parity fragment is re-calculated; this can be done by XOR-ing the values of the modified data fragment before and after the modification and the old value of the parity fragment.

The RADD protocol can tolerate any single node failure, because the contents of any one fragment (data or parity) can be restored by XOR-ing the values of the remaining fragments. The storage overhead for achieving this is just one-mth of the file size for the parity fragment (as opposed to a full copy of the file in the scheme with two copies and one witness). RADD can also be generalized to tolerate more than a single node failure by introducing multiple parity fragments.

However, the applicability of the RADD method is limited by the following factors:

1. RADD does not provide good availability when network partitionings occur. For

example, if the parity fragment together with at least one other fragment get separated from the rest of the system, no update is possible in the rest of the system. In addition, if fragments are split among two partitions, at least part of the data item becomes unavailable for reads.

2. RADD assumes that the system contains a large number of nodes (at least $m + 1$ nodes are required).

3. RADD works well for large multi-page objects only. For small objects (e.g., of less than one page), even assuming that fragments from different objects can be packed on one disk page,[6] performance would be poor. Indeed, small objects are likely to be read and written in their entirety, which entails accessing all $m + 1$ fragments on every operation. It would also waste a lot of buffer space in main memory. Indeed, if all fragments are read in parallel, enough buffer space must be allocated for $m + 1$ pages, whereas only one page is actually needed for the data object itself. In fact, replication is often maintained at the level of physical pages. Even for large files, however, there are some advantages to having the whole file available at one site, one administration/maintenance domain, etc.

Overall, RADD can be a valuable way to maintain large files on a one-segment local area network, where network partitionings are unlikely.

2.3.2 Reducing Communication Overhead

User operations in pessimistic quorum-based protocols communicate with all replicas from a quorum. Thus, to cut the communication costs, quorum size must be reduced. In majority quorums, the quorum size is roughly half of the total number of replicas. Quorums of smaller relative size, as well as other interesting properties, have been proposed recently based on a logical structure imposed on the system.

For instance, in the *grid* protocol [8], all nodes having a replica of the data item are viewed as being arranged in an $m \times n$ grid (see Figure 2.1). A read quorum is defined to be a set of replicas that includes one node selected from each column of the grid. A

[6] Otherwise, RADD would actually lose in terms of disk storage, because each fragment would occupy a fraction of the page, wasting the rest of the page space.

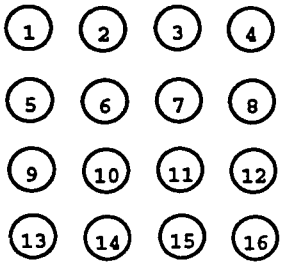


Figure 2.1: The grid for N = 16.

write quorum includes a set as above plus all replicas from some column. For example, in the grid in Figure 2.1, a set of nodes {1, 3, 4, 6, 7, 11, 15} is a write quorum, because it includes a set {1, 3, 4, 6} of representatives from each column and a set {3, 7, 11, 15} that covers all nodes from some column. It is easy to see that grid-based quorums satisfy the intersection property. Therefore, the quorum-based protocol that uses these quorums guarantees consistency.

Grid quorums are fully distributed and, at the same time, contain fewer nodes than voting quorums. Indeed, in square grids, the size of read quorums is $\sqrt{N}$, and the size of write quorums is $2\sqrt{N} - 1$, where N is the total number of replicas. This contrasts with fully distributed voting quorums, in which the size of read and write quorums in the simplest case is $\lceil \frac{N+1}{2} \rceil$, with any reduction of read quorum size causing a corresponding increase in the size of write quorums. Thus, the communication overhead in grid protocols is lower.

Equally important, as the number of replicas of a data item increases, the fraction of replicas that service an individual operation on the data item decreases and tends to 0. (Recall that an operation is serviced by a quorum of replicas.) Therefore, the average load due to reads (and, in systems with total writes, the total load) carried by a server can be made arbitrarily low by increasing the total number of replicas.

On the other hand, data availability suffers in the grid protocol, as in other protocols with small quorums. Indeed, since any write quorum intersects with any other quorum, any write quorum of nodes being down makes collecting permission from any other quorum of nodes impossible, rendering the data item unavailable. Thus, reducing the quorum size (and hence cutting communication overhead) necessarily implies reduc-

ing the number of arbitrary node failures tolerated by the system. In fact, the write availability (the probability that a write will succeed) provided by square grid quorums goes asymptotically to 0 as the total number of replicas increases [36].

Thus, we observe a tradeoff between the performance of a replicated system and the data availability it provides. We propose a way to address this problem in Chapter 4.

Also, given the asymptotically low write availability of square grids, the question arises as to whether the achievable degree of load sharing in the grid protocol is limited in practically useful systems by availability constraints. However, as shown in Chapter 3, arranging the replicas in a grid with $\log N$ rows and $\frac{N}{\log N}$ columns ensures asymptotically high read and write availability *and* load sharing at the same time. Thus, while grid quorums provide worse data availability than voting ones for a given number of replicas, they (unlike the voting quorums) can in theory be used to build a system that simultaneously exhibits high data availability and degree of load sharing.

2.4 A Primer on Serializability Theory

This section introduces useful notation, formally defines terms described above, and provides results used throughout the thesis to prove the correctness of proposed protocols.

2.4.1 Basic Theory

Most of the material in this subsection (except for Lemma 1) is adopted from [5].

A *transaction* T_i is a partial order $(S_i, <_i)$, where: (1) S_i is a set of logical read and write operations executed by T_i plus an abort a_i or commit c_i operation; (2) $<_i$ reflects the execution order of logical operations, i.e., if $op_1 <_i op_2$, then op_1 should be executed before op_2; (3) $a_i \in S_i$ iff $c_i \notin S_i$ (a transaction may either commit or abort, but not both); (4) if $c_i \in S_i$, for any other operation $op \in S_i, op <_i c_i$ (if a transaction commits, commit is the last operation of the transaction); (5) if $a_i \in S_i$, for any other operation $op \in S_i, op <_i a_i$ (if a transaction aborts, abort is the last operation of the transaction); (6) any two operations on the same data item are ordered. We denote logical read and write operations on data item x issued by T_i as $w_i[x]$ and $r_i[x]$. We assume that a transaction reads and writes data item x at most once.

Two logical operations, op_1 and op_2, *logically conflict* if they belong to different transactions, operate on the same data item, and one of them is a write.

In a replicated database, logical reads and writes on data items are translated into *physical* operations on replicas of these data items. This translation is formalized by a function h that maps each $r_i[x]$ into $r_i[x_p]$, where x_p is a replica of x, and each $w_i[x]$ into $w_i[x_{p_1}], \ldots, w_i[x_{p_m}]$ for some replicas $x_{p_1}, \ldots, x_{p_m}$ of x ($m > 0$). It is also convenient to include commit and abort operations in the domain of h by defining $h(c_i) = c_i$ and $h(a_i) = a_i$. We call transaction T_i the *owner transaction* of logical and physical operations executed by T_i.

A *complete history* (or, for the purpose of this thesis, simply *history*) H over transactions $\{T_0, \ldots, T_n\}$ is a partial order $(S_H, <_H)$, where: (1) $S_H = h(\cup_{i=0}^{n} S_i)$ for some translation function h (a history records physical operations on replicas of data items); (2) $<_H$ reflects the execution order of physical operations, i.e., if $op_1 <_H op_2$, then op_1 was executed before op_2; (3) for each T_i and all operations op_1 and op_2 in S_i, if $op_1 <_i op_2$, then every operation in $h(op_1)$ is related by $<_H$ to every operation in $h(op_2)$ (a history preserves all orderings stipulated by transactions); (4) for every r_i in S_H, S_H contains some $w_j <_H r_i$ (a transaction can read only those replicas that have been previously initialized); (5) all pairs of conflicting operations in S_H are related by $<_H$, where two operations conflict if they operate on the same replica and at least one of them is a write; (6) if $w_i[x] <_i r_i[x]$ and $h(r_i) = r_i[x_p]$, then $w_i[x_p] \in h(w_i[x_i])$ (if transaction T_i writes a data item x and later reads x, then T_i must write into the same replica x_p that it subsequently reads).[7]

Transaction T_j *reads-x-from* T_i in H if there is a replica x_p of x such that $w_i[x_p] <_H r_j[x_p]$ and there is no $w_k[x_p]$ that falls between these operations, i.e., $w_i[x_p] <_H w_k[x_p] <_H r_j[x_p]$. Given history H, $w_i[x_p]$ is a *final write for* x_p in H if $a_i \notin S_H$ and for all $w_j[x_p]$ in H ($j \neq i$), either $a_j \in S_H$ or $w_j[x_p] < w_i[x_p]$.

We will consider only *recoverable* histories, i.e., histories in which aborted transactions cannot have any effect on committed transactions. To guarantee this, it is sufficient to

[7] Transaction management mechanisms outside replica control are assumed to be responsible for enforcing the above properties of transactions and histories. For instance, commit protocols are used to make sure a transaction does not contain both commit and abort operations.

require that a transaction can commit only after the commitment of all transactions (other than itself) from which it read. Formally, history H is *recoverable* if whenever a transaction T_i reads from T_j ($i \neq j$) in H and $c_i \in S_H$, $c_j <_H c_i$. All practically useful concurrency control protocols allow only recoverable histories. Since we are interested in committed transactions only, and because aborted transactions in recoverable histories do not affect committed transactions, we can limit ourselves to considering histories that contain only committed transactions. *From this point on, we assume that all transactions in H are committed.*

A history in which translation function h translates any logical operation on a data item x into a physical operation on a single replica x_p ($h(w_i[x]) = w_i[x_p]$ and $h(r_i[x]) = r_i[x_p]$ for any i) is called a non-replicated history.

A *serial* history is a totally ordered history, such that for every pair of transactions T_i and T_j, either all physical operations executed by T_i precede all physical operations executed by T_j, or vice versa. A history H is called *conflict-serializable* if there is a serial history H_s over the same set of transactions and the same translation function h, such that if (physical) operations op_1 and op_2 conflict and $op_1 <_H op_2$, then $op_1 <_{H_s} op_2$.

The serialization graph SG(H) for a history H is a directed graph whose nodes correspond to committed transactions in H and whose edges connect transaction T_i to T_j (denote $T_i \rightarrow T_j$) if there are conflicting physical operations op_1 in T_i and op_2 in T_j, such that $op_1 <_H op_2$. A history is conflict-serializable if and only if its serialization graph is acyclic (see Theorem 2.1 in [5]).

Two histories, H_1 and H_2, are *equivalent* if: (1) they are defined over the same set of transactions (note that translation functions may differ); (2) T_i reads-x-from T_j in H_1 iff T_i reads-x-from T_j in H_2; and (3) for each data item x, if $w_i[x_p]$ is a final write in H_1 on a replica of x, then there is a final write $w_i[x_q]$ in H_2 on a replica of x (note that final writes are executed by the same transaction T_i in both histories). A history is *one-copy serializable* if it is equivalent to a serial non-replicated history.

In a directed graph, node n *precedes* node m, denoted $n \ll m$, if there is a path from n to m. Given a history H, a *replicated data serialization graph (RDSG)* for H is SG(H) with enough edges added so that, for every data item x: (1) if T_i and T_j write x, then either $T_i \ll T_j$ or $T_j \ll T_i$; and (2) if T_j reads-x-from T_i, T_k writes some replica of x

($k \neq i, k \neq j$), and $T_i \ll T_k$, then $T_j \ll T_k$. Note that an RDSG for H is not unique. The main theorem (referred to as the *One-Copy Serializability Theorem* in this thesis) states that, if H has an acyclic RDSG, then H is one-copy serializable (see Theorem 8.4 in [5]).

We conclude with a lemma that formalizes and generalizes the claim from Section 1.3.3 about combining replica control and concurrency control protocols to achieve one-copy serializability.

Lemma 1 *Let every replica of every data item have an associated version number (originally 0) and a read and write lock. If a replica control protocol ensures that:*

1. *before performing a physical operation on any replica, a logical read (write) operation on a data item must obtain the read (write) lock from a set of replicas from this data item;*

2. *no two logically conflicting operations can hold locks on all replicas from their respective sets simultaneously, so that all such operations can be ordered by the time they acquired locks from their sets of replicas; and*

3. *the ith logical write assigns version number i to all replicas on which it performs, and a logical read always reads from a replica with version number i, where i is the number of the immediately preceding write,*

then this replica control protocol can be combined with the S2PL concurrency control protocol using the same locks to allow only one-copy serializable histories.

Proof. Consider an arbitrary history H produced by the combination of a replica control protocol satisfying the lemma and S2PL. Augment serialization graph SG(H) by adding an edge $T_i \rightarrow T_j$ for any two transactions T_i and T_j containing logically conflicting operations, such that T_i's operation collected locks from its set of replicas of x before T_j's operation. The resulting graph G is an RDSG for H. Indeed, it has an edge connecting any two transactions writing to the same data item by construction. Moreover, if T_j reads-x-from T_i, T_k writes some replica of x ($k \neq i, k \neq j$), and $T_i \ll T_k$, then $T_j \ll T_k$. To see this, assume that T_i's write to x assigned version number v_i to all replicas on

which it performed, and T_k's write to x assigned version number v_k to its replicas. Since $T_i \ll T_k$, $v_i < v_k$. Because T_j reads-x-from T_i, the version number of the replica from which T_j's read operation reads is v_i. Because T_j and T_k logically conflict, there is an edge in our graph connecting them. If it goes from T_j to T_k, then there is a path desired. If the edge goes from T_k to T_j, then T_k's write to x obtained its locks before T_j's read. Therefore, by condition (3), T_j read from a replica with a version number greater than or equal to v_k, a contradiction. So, we conclude that the constructed graph is RDSG.

Now, we need only to show that it is acyclic. If there is an edge in G from T_i to T_j, then these transactions contain either logically or physically conflicting operations, op_1 and op_2. In both cases, before op_2 could lock all its replicas, op_1 must have released some replicas. But, according to the S2PL protocol, this could happen only after T_i commits. Therefore, whenever there is an edge from T_i to T_j, T_i must have committed before T_j. If we assume that there is a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_1$ in the graph, then T_1 must have committed before itself, an impossibility. Thus, we conclude that G is acyclic, and, hence, H is one-copy serializable. □

2.4.2 An Extension to the Basic Theory

Many protocols described in this thesis include recovery, where a stale replica obtains more recent information from another replica. We need to extend standard theory to include recovery operations (referred to, for short, as *recoveries*). The extension described here is analogous to that of [16], although our recovery operations differ significantly from the update transactions in [16]: unlike update transactions, our recoveries are not controlled by the concurrency control protocol that ensures serializability of transactions on the level of physical operations; thus, our recovery operations are *not* transactions.

When we need to refer to transactions and recoveries together, we will call them *actions*. Histories and serialization graphs are now defined over sets of actions, including recoveries.

Action A_j *directly reads-x-from* action A_i in history H if there is a replica x_p such that $w_i[x_p] <_H r_j[x_p]$ and there is no $w_k[x_p]$ that falls between these operations, i.e., $w_i[x_p] <_H w_k[x_p] <_H r_j[x_p]$.

Let $R_{u_1}, \ldots, R_{u_n}$ be a sequence of recoveries and A_i and A_j be two actions. A_j *indirectly reads-x-from* A_i if R_{u_1} *directly reads-x-from* A_i, R_{u_2} *directly reads-x-from* R_{u_1}, ..., and A_j *directly reads-x-from* R_{u_n}. Action A_j *reads-x-from* A_i if A_j *directly* or *indirectly reads-x-from* A_i.

Unsuccessful recoveries do not change the state of any node in the system. Therefore, they do not affect committed transactions or successful recoveries. Also, as discussed in the previous subsection, since we consider only recoverable histories, aborted transactions cannot in any way affect committed transactions. Moreover, because a recovery in our protocols reads the source replica only if it has not been written by an uncommitted transaction, aborted transactions cannot affect successful recoveries either. Hence, when we study serializability of committed transactions, we can ignore both aborted transactions and unsuccessful recoveries and consider only histories in which all transactions are committed and all recoveries are successful. From this point on, *we assume that history H contains only successful actions, that is, committed transactions and successful recoveries.*

A serialization graph SG(H) is a directed graph whose nodes are (successful) actions from H, and there is an edge $A_i \rightarrow A_j$ for any two actions A_i and A_j, such that they have conflicting physical operations op_1 in A_i and op_2 in A_j, and $op_1 <_H op_2$. Note that, if action A_j *reads-x-from* A_i (directly or indirectly), then SG(H) has a path from A_i to A_j. A replicated data serialization graph RDSG(H) for history H is SG(H) with enough edges added so that, for every data item x: (1) if transactions T_i and T_j write x, then either $T_i \ll T_j$ or $T_j \ll T_i$ (note that recoveries are not required to be ordered); and (2) for any three transactions T_i, T_j, and T_k, if T_j *reads-x-from* T_i, T_k writes x, and $T_i \ll T_k$, then $T_j \ll T_k$.

The proofs of the One-Copy Serializability Theorem in [5] and Lemma 1 in the previous subsection still hold with the extension just described.

Chapter 3

Performance Characteristics of General Grid Structures

In pessimistic replica control protocols, the coordinator of an operation must communicate with all nodes from a quorum before returning to the user. Thus, there is an inherent performance penalty these protocols incur for providing strong consistency. To minimize this penalty, it is desirable that the quorums be small. This can be achieved by defining quorums based on a logical structure imposed on the network. Many such quorum definitions have been proposed [33, 9, 36, 3, 72, 60]. This chapter concentrates on the performance of one member of the family of structure-based protocols, the grid protocol [9].

Grid-based quorums were briefly described in Section 2.3.2. As a reminder, a (minimal) read quorum in the grid protocol is formed by assembling one copy from each column of the grid (also called a *column-cover* or *c-cover*); a (minimal) write quorum is formed by the union of a read quorum and all replicas from any one column of the grid. For square grids, the size of read quorums is $\sqrt{N}$ and the size of write quorums is $2\sqrt{N} - 1$, where N is the total number of replicas. This contrasts with the voting protocol [18], where the quorum size in the simplest case is $\lceil \frac{N+1}{2} \rceil$.

Grid quorums are attractive not only due to their small size, but also because they are *fully distributed.* Quorums are called *fully distributed* if all minimal write quorums are

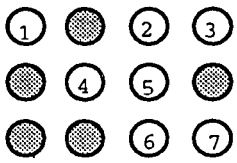


Figure 3.1: A 3×4 grid with 7 nodes.

of equal size, all minimal read quorums are of equal size, and every replica is a member of an equal number of read quorums and write quorums. It is desirable for quorums to be fully distributed, because each replica plays an equal role in these solutions, and the load distribution among replicas is uniform.[1]

Provably the best (with regard to quorum size) fully distributed solutions have a quorum size of $\Omega(\sqrt{N})$, where N is the number of replicas of the data item. Although this bound is achieved in *Maekawa's* [44], *Grid* [9], and *Hierarchical Grid* (h-grid) [36] protocols, each of these protocols has some drawback. Maekawa's protocol has the lowest constant among the three, but it provides poor availability. The grid protocol outperforms Maekawa's protocol in terms of availability. The h-grid protocol has a quorum size identical to the grid protocol and also provides asymptotically high write availability (which the grid protocol, in general, does not). However, the higher write availability in the h-grid protocol is obtained at the expense of read availability. This means that one does not dominate the other.

This chapter describes a modified grid protocol and studies the availability and load sharing behavior of grids. First, we modify the grid protocol described in [9] to improve its availability properties. We also derive the expressions for read and write availability in the case of a *general* grid, i.e., a grid where some positions are not assigned to replicas. A write quorum in such grids must contain a replica from every column and all *actual* replicas from a column. For instance, in the grid on Figure 3.1, a set of replicas $\{1, 2, 3, 4\}$ is a write quorum, since it includes a replica from every column and all replicas from the first column.

[1] If quorum size were the only criterion, we could define both read and write quorums as singletons, all containing the same replica. While these quorums have the smallest possible size, they obviously have little sense: one could as well have a non-replicated system. The requirement for quorums to be fully distributed precludes such solutions.

Considering general grids is interesting because, as we show in Section 3.4, the availability resulting from organizing N copies into an $m \times n$ grid (where $mn = N$) often can be improved by placing the N copies in an $m' \times n'$ grid (where $m'n' > N$) and leaving some grid positions empty (i.e., filling them with *holes*). Of course, permitting holes in a grid diminishes the fully distributed nature of the grid protocol. We resolve this issue by permitting at most one hole in any column of the grid so the algorithm will remain "almost fully distributed." Therefore, for a large N, the load is distributed among nodes almost as evenly as in a fully distributed solution, yet there is more flexibility in constructing the grid. In fact, for many values of N (e.g., when N is prime), no sensible *fully* distributed solution exists.

The second issue this chapter addresses is that of availability. We examine it from both the theoretical and empirical points of view. As shown in [36], the write availability for square grids goes asymptotically to 0 as the grid size grows; however, the question we pose is: If rectangular, non-square grids are permitted, would the asymptotic availability go to 1? What must the dimensions of the grid be for this to happen? We find that for this to happen, the dimensions of the grid must be close to $\log N \times N/\log N$, which produces quorum sizes that are almost linear in N, thereby defeating the main strength of grids. Interestingly enough, empirical calculations show that this asymptotic behavior of grids comes into play only for *very large* values of N. It was found that, even in square grids, availability increases towards 1 for $N \leq 23000$ and the probability p of a node being up 0.99, and it begins to decline only for values of N larger than 23000. Thus, despite the negative asymptotic result mentioned above, grids remain promising from both availability and load sharing points of view in most practical situations.

Another issue addressed in this chapter is that of grid design subject to simultaneous availability and load-sharing constraints. Given a desired degree of load sharing and a minimum availability threshold, what is the smallest number of copies satisfying these requirements? Moreover, how should these copies be arranged in an $m \times n$ grid? An algorithm is proposed to enable a designer to solve this problem and meet the twin objectives of load sharing and availability in an optimal manner.

This chapter describes joint work with Akhil Kumar and Rakesh Sinha [37]. Theorem 1 was postulated and proven by Rakesh Sinha.

3.1 Terminology and Notation

Throughout this chapter, the terms "node" and "site" are used, often interchangeably, to refer to physical network sites storing replicas of a given data item. The term "position," on the other hand, refers to the placeholders in logical grids that may or may not be occupied by physical nodes. A grid in which all positions are occupied by physical nodes is called *solid.* Positions not occupied by nodes are called *holes.* A grid with one or more holes is called a *hollow* grid. Figure 3.1 shows a 3×4 hollow grid with 7 nodes and 5 holes.

Two grids are equivalent if their read and write quorum sizes and availabilities are the same. A column of a grid is *good* if *all* its nodes are up; it is *alive* if *at least one* of its nodes is up. A column that is not good is *bad.*

The following is standard notation in this chapter. The system contains N nodes organized in an $m \times n$ grid, where m is the number of rows, and n is the number of columns. A grid may or may not contain all N nodes. As an example, a solid $m \times n$ grid in which $mn < N$ does not use $N - mn$ nodes. A node is assumed to be operational with probability p and to have failed with probability $q = 1 - p$. Read and write availabilities are denoted by RA and WA, respectively, and represent the probabilities that there are enough operational nodes to form a corresponding quorum. Read and write quorum sizes are denoted by Q_R and Q_W. See Table 3.1 for a summary of key terminology.

Finally, as a measure of the degree of load sharing in the system, we define a *relative read (write) quorum size* to be the ratio of the read (write) quorum size to the total number of nodes used in the grid. The rationale for this measure is that, in a solid grid containing all N nodes, the relative quorum size shows the fraction of nodes participating in a single operation. Hence, if different operations choose quorums randomly, the relative quorum size gives the fraction of the total load carried on average by a single node. If not all N nodes are included in the grid, extra nodes never participate in any operation and should not be taken into account in the measure of load sharing.

Table 3.1: Summary of key terminology.

Notation	Description
N	total number of nodes in the system
m	number of rows
n	number of columns
p	probability that a given node is up
q	$1-p$
RA	read availability
WA	write availability
Q_R	size of read quorum
Q_W	size of write quorum
column is *good*	*all* nodes in the column are *up*
column is *bad*	at least one node in the column is *down*
column is *alive*	at least one node in the column is *up*
relative read quorum size	$\frac{Q_R}{Number\ of\ nodes\ in\ the\ grid}$
relative write quorum size	$\frac{Q_W}{Number\ of\ nodes\ in\ the\ grid}$

3.2 Computing Read and Write Availability for General Grids

In Section 3.2.1, we derive formulas for read and write availability for a general grid, i.e., one in which some positions may not be occupied by nodes. In previous works, only solid rectangular grids were considered [9]. However (as we illustrate in Section 3.4), hollow grids may provide better availability than solid ones. The computations in Section 3.2.1 are carried out for the existing grid protocol. Then, Section 3.2.2 describes the modified grid protocol and recomputes read availability for it. This section also shows that the modified grid protocol is optimal.

3.2.1 Availability in the Existing Grid Protocol

Consider N replicas placed in an $m \times n$ grid, where $mn \geq N$. If $mn > N$, then the grid contains $mn - N$ holes. Let n_1 columns contain m_1 nodes, n_2 columns contain m_2 nodes, ..., n_k columns contain m_k nodes, and so on. The read and write availabilities of this grid are calculated as follows.

Prob(a column with m_i nodes is alive) $= 1 - q^{m_i}$;

Prob(all columns in the grid are alive) =

$$(1-q^{m_1})^{n_1} \cdot (1-q^{m_2})^{n_2} \cdot \ldots \cdot (1-q^{m_k})^{n_k};$$

Prob(a column with m_i nodes is bad and alive) =
1− Prob(a column with m_i nodes is good) − Prob(a column with m_i nodes is dead) $= 1 - p^{m_i} - q^{m_i}$;
Prob(all columns are bad and alive) =

$$(1-p^{m_1}-q^{m_1})^{n_1} \cdot (1-p^{m_2}-q^{m_2})^{n_2} \cdot \ldots \cdot (1-p^{m_k}-q^{m_k})^{n_k};$$

Write availability WA = Prob(all columns are alive) − Prob(all columns are bad and alive) =

$$(1-q^{m_1})^{n_1} \cdot \ldots \cdot (1-q^{m_k})^{n_k} - (1-p^{m_1}-q^{m_1})^{n_1} \cdot \ldots \cdot (1-p^{m_k}-q^{m_k})^{n_k}. \quad (3.1)$$

Read availability RA = Prob(all columns are alive) =

$$(1-q^{m_1})^{n_1} \cdot (1-q^{m_2})^{n_2} \cdot \ldots \cdot (1-q^{m_k})^{n_k}. \quad (3.2)$$

In the existing grid protocol, there are two tradeoffs: (1) between read and write availabilities, and (2) between read quorum size (which is a measure of read performance) and write availability. Write availability is better in a grid where the number of rows is much smaller than the number of columns, which means the grid should contain a large number of short columns. However, in a grid with a large number of columns, the size of read quorums is also large (recall that a read quorum is formed by obtaining one node from each column of the grid). Hence, read performance in such grids suffers. Also, having too many short columns makes it harder to collect one operational node from every column and hurts read availability. These tradeoffs make designing a grid especially hard.

3.2.2 Availability in the Modified Protocol

We modified the grid protocol of [9] by redefining a read quorum in the following manner. A read quorum can be formed in one of two ways: as a set consisting of one node from

each column of the grid *or* as a set of all nodes from any single column. Write quorums remain the same: they must include a node from each column *and* all nodes from any one column. It is straightforward to show that any read quorum defined this way intersects with any write quorum, and hence the quorum intersection rule is satisfied. This protocol will be referred to as the *modified grid protocol.* (This modification was also suggested independently by Pâris and Sloope [51] and by Neilsen [47].)

This seemingly minor improvement to the protocol is significant, because it increases read availability without changing write availability. Notice that a grid with many short columns now provides high write availability and, at the same time, good read availability and performance, since a read operation can assemble all nodes from any one of many short columns to form a quorum. To calculate the read availability of the modified grid, note that the probability that a column is bad and alive is $1 - p^m - q^m$, where m is the number of nodes in the column. Then, the probability that a read quorum *cannot* be formed is equal to the probability that *all* columns are *bad* minus the probability that all columns are *bad and alive*, which is:

$$(1 - p^{m_1})^{n_1} \cdot \ldots \cdot (1 - p^{m_k})^{n_k} -$$
$$(1 - p^{m_1} - q^{m_1})^{n_1} \cdot \ldots \cdot (1 - p^{m_k} - q^{m_k})^{n_k}$$

Read availability, RA, becomes:

$$1 - ((1 - p^{m_1})^{n_1} \cdot \ldots \cdot (1 - p^{m_k})^{n_k} -$$
$$(1 - p^{m_1} - q^{m_1})^{n_1} \cdot \ldots \cdot (1 - p^{m_k} - q^{m_k})^{n_k}) \tag{3.3}$$

It is easy to show that the set of quorums defined by the modified grid is optimal, i.e., it is not dominated by any other set. The notion of *non-dominance* was first introduced in [22] to formalize the intuition behind optimal quorums. Let R and W be sets of read and write quorums. A pair of sets $Q = (R, W)$ (also called a *quorum agreement*) is *dominated* iff there exists another quorum agreement $Q' = (R', W')$ such that: (1) $Q \neq Q'$ (i.e., $R \neq R'$ or $W \neq W'$); (2) for each $r \in R$, there exists $r' \in R'$ such that $r' \subseteq r$; and (3) for each $w \in W$, there exists $w' \in W'$ such that $w' \subseteq w$. Otherwise, Q is called *non-dominated.*

Table 3.2: Unavailability of the existing and modified grid protocols with p = 0.90.

Grid Dimensions	Write Unavailability	Old read Unavailability	New read Unavailability	Improvement Ratio
2 x 2	$5.23 \cdot 10^{-2}$	$1.99 \cdot 10^{-2}$	$3.70 \cdot 10^{-3}$	5
2 x 4	$4.05 \cdot 10^{-2}$	$3.94 \cdot 10^{-2}$	$2.53 \cdot 10^{-4}$	155
2 x 6	$5.86 \cdot 10^{-2}$	$5.85 \cdot 10^{-2}$	$1.30 \cdot 10^{-5}$	4489
4 x 2	$1.18 \cdot 10^{-1}$	$2.00 \cdot 10^{-4}$	$6.88 \cdot 10^{-5}$	2
4 x 4	$1.44 \cdot 10^{-2}$	$4.00 \cdot 10^{-4}$	$1.63 \cdot 10^{-5}$	24
4 x 6	$2.25 \cdot 10^{-3}$	$6.00 \cdot 10^{-4}$	$2.88 \cdot 10^{-6}$	207
6 x 2	$2.20 \cdot 10^{-1}$	$2.00 \cdot 10^{-6}$	$9.37 \cdot 10^{-7}$	2
6 x 4	$4.82 \cdot 10^{-2}$	$4.00 \cdot 10^{-6}$	$4.11 \cdot 10^{-7}$	9
6 x 6	$1.06 \cdot 10^{-2}$	$6.00 \cdot 10^{-6}$	$1.36 \cdot 10^{-7}$	44

The notion of non-dominance characterizes the optimality of the quorums, because, if Q is dominated, there exists another quorum agreement, Q', whose quorums are subsets of quorums from Q. Then, the protocol based on Q' would have better availability and, possibly, performance properties, since it would be easier for an operation to collect a quorum of permissions.

Quorums defined by the modified grid can be described using the *disjoined set* method for constructing quorum agreements [47] if columns of the grid are chosen as the disjoined sets. Since any quorum agreement constructed using the disjoined sets method is non-dominated (Theorem 5.2 in [47]), the quorum agreement defined by the modified grid is also non-dominated.

This refinement of the grid protocol can also be easily extended to the h-grid and hierarchical quorum consensus-2 (HQC2) [35, 36] protocols. The details are omitted here because the focus of our present work is exclusively on grids.

3.3 Availability in The Existing vs. Modified Grid Protocols

In this section, we compare availability provided by the existing (*old*) grid protocol [9] and our modified (*new*) protocol described above. For purposes of this comparison, we consider solid grids of the same dimensions as those considered in [9] and recompute the

Table 3.3: Unavailability of the existing and modified grid protocols with p = 0.95.

Grid Dimensions	Write Unavailability	Old read Unavailability	New read Unavailability	Improvement Ratio
2 x 2	$1.40 \cdot 10^{-2}$	$4.99 \cdot 10^{-3}$	$4.81 \cdot 10^{-4}$	10
2 x 4	$1.00 \cdot 10^{-2}$	$9.96 \cdot 10^{-3}$	$8.92 \cdot 10^{-6}$	1117
2 x 6	$1.49 \cdot 10^{-2}$	$1.49 \cdot 10^{-2}$	$1.24 \cdot 10^{-7}$	120237
4 x 2	$3.44 \cdot 10^{-2}$	$1.25 \cdot 10^{-5}$	$2.32 \cdot 10^{-6}$	5
4 x 4	$1.21 \cdot 10^{-3}$	$2.50 \cdot 10^{-5}$	$1.60 \cdot 10^{-7}$	156
4 x 6	$7.82 \cdot 10^{-5}$	$3.75 \cdot 10^{-5}$	$8.23 \cdot 10^{-9}$	4553
6 x 2	$7.02 \cdot 10^{-2}$	$3.12 \cdot 10^{-8}$	$8.28 \cdot 10^{-9}$	3
6 x 4	$4.92 \cdot 10^{-3}$	$6.25 \cdot 10^{-8}$	$1.16 \cdot 10^{-9}$	53
6 x 6	$3.46 \cdot 10^{-4}$	$9.38 \cdot 10^{-8}$	$1.22 \cdot 10^{-10}$	766

Table 3.4: Unavailability of the existing and modified grid protocols with p = 0.99.

Grid Dimensions	Write Unavailability	Old read Unavailability	New read Unavailability	Improvement Ratio
2 x 2	$5.92 \cdot 10^{-4}$	$2.00 \cdot 10^{-4}$	$3.97 \cdot 10^{-6}$	50
2 x 4	$4.00 \cdot 10^{-4}$	$4.00 \cdot 10^{-4}$	$3.13 \cdot 10^{-9}$	127835
2 x 6	$6.00 \cdot 10^{-4}$	$6.00 \cdot 10^{-4}$	$1.85 \cdot 10^{-12}$	324404608
4 x 2	$1.55 \cdot 10^{-3}$	$2.00 \cdot 10^{-8}$	$7.88 \cdot 10^{-10}$	25
4 x 4	$2.45 \cdot 10^{-6}$	$4.00 \cdot 10^{-8}$	$2.45 \cdot 10^{-12}$	16344
4 x 6	$6.37 \cdot 10^{-8}$	$6.00 \cdot 10^{-8}$	$5.66 \cdot 10^{-15}$	10596700
6 x 2	$3.42 \cdot 10^{-3}$	$2.00 \cdot 10^{-12}$	$1.17 \cdot 10^{-13}$	17
6 x 4	$1.17 \cdot 10^{-5}$	$4.00 \cdot 10^{-12}$	$7.77 \cdot 10^{-16}$	5146
6 x 6	$4.02 \cdot 10^{-8}$	$6.00 \cdot 10^{-12}$	undetectable	—

read availability for different values of p. Note that write availability remains the same in both protocols.

The results are shown in Tables 3.2, 3.3 and 3.4, which correspond to p=0.9, 0.95, and 0.99, respectively. For ease of comparison, we follow [9] in expressing fault-tolerance properties in the tables as *unavailabilities* (defined as 1 - availability). The improvement ratio shown in the last column of the tables is computed as the read unavailability in the old protocol divided by the read unavailability in the new one; it indicates the improvement that results from using the new protocol.

Several conclusions can be drawn from these results. First, in all cases, the read unavailability of the new protocol is significantly lower than in the old protocol. Second, the gap between the two protocols, as indicated by the improvement ratio, increases for larger values of p. Third, for all values of p, the relative improvement depends considerably on the grid dimensions, m and n. It is maximum when the number of rows (m) is small and the number of columns (n) is large. This makes intuitive sense, because the new protocol makes it easy to form a read quorum from all nodes in any one column (since m is small, i.e., each column is short). On the other hand, this is also the case where it is the hardest to form a read quorum in the old protocol by a column-cover, since the number of columns is large and each column is short.

Another noteworthy point is that the modified protocol considerably lessens the tradeoffs between read and write availabilities mentioned in Section 3.2. This means that the same improvement in write availability now results from a much smaller sacrifice in read availability. In the old protocol, both read and write availability are very sensitive to actual grid dimensions m and n for a given total number of nodes N. To see this, consider the example of 4×6 and 6×4 grids in Table 3.3. In the existing protocol, write unavailability is two orders of magnitude lower for the 4×6 grid, while read unavailability is three orders of magnitude lower for the 6×4 grid. On the other hand, for the modified protocol, the two orders of magnitude decline in the write unavailability for the 4×6 grid causes the read unavailability to increase (worsen) by a factor of only less than 5. Similar arguments apply to Tables 3.2 and 3.4. Hence, the new protocol, in addition to improving read availability, is less sensitive to grid dimensions.

Finally, the new protocol can satisfy given read and write availability requirements

with a grid of fewer nodes. Consider the case where a grid must be designed so that the read and write unavailabilities cannot exceed 10^{-4} and 10^{-5}, respectively, given $p = 0.95$. With the existing protocol, no solid grid with fewer than 35 nodes can simultaneously satisfy both these requirements. On the other hand, the new protocol allows these requirements to be met using only 24 nodes (e.g., with a 4×6 grid).

In the rest of the paper, we consider only the modified grid protocol.

3.4 Finding Grids with the Highest Availability

This section describes an algorithm that, given N nodes, finds the *almost fully distributed* grid solution with the highest availability. This problem is complicated for two reasons: (1) hollow grids often have a higher availability than solid grids, and (2) in some cases, a higher availability can be achieved if some of the nodes are not used at all.

To illustrate these points, consider the availabilities of various grids for $N = 16$ computed in Table 3.5. In this table, read, write and weighted availabilities have been computed. The weighted availability is defined as $F \cdot RA + (1 - F) \cdot WA$, where F is the fraction of read operations. F is assumed to be 0.8. Table 3.5 shows that a solid 4×4 grid gives a *lower* write and weighted availability than a 4×5 grid with four holes in the bottom row. Moreover, a solid 3×5 grid containing only 15 nodes has higher values for both write and weighted availability than any solid grid containing 16 nodes. These improvements become more significant when viewed as relative decreases in *unavailability*, defined as 1− availability, rather than increases in availability.

Algorithm `OptimalWriteAvail` for finding the grid with the highest write availability, given N nodes, is shown in Figure 3.2. Since the objective is to design an "almost fully distributed" protocol, our algorithm considers only the grids containing at most one hole in any column. Moreover, since read and write availabilities are not a function of the exact location of the hole in a given column, it is assumed without loss of generality that the hole is always at the bottom of a column. For N nodes, the algorithm considers all rectangular grids such that: $m \leq n$ and $N \leq mn < N + n$. It does not make sense to consider grids with $mn \geq N + n$, because such a grid must have at least n holes, i.e., the bottom row contains all holes. and so it can be eliminated from consideration. Also,

Table 3.5: Availability of various grids for $N = 16$ and $p = 0.9$.

Grid Configuration	Read Availability	Write Availability	Weighted Availability
1×16 solid	$1 - 10^{-16}$	0.185302	0.837060
2×8 solid	0.9999994	0.922746	0.984548
4×4 solid	0.999984	0.985629	0.997113
8×2 solid	0.999999989	0.675632	0.935126
16×1 solid	$1 - 10^{-16}$	0.185302	0.837060
3×5 solid (1 node not used)	0.999973	0.993575	0.998694
4×5 with 4 holes in the bottom row	0.999972	0.994079	0.998797

the algorithm considers only $m \times n$ grids where $m \leq n$. This is because for any $m < n$, an $m \times n$ almost fully distributed grid has higher write availability than an $n \times m$ grid. Thus, considering the latter is not necessary.

Let G_N be the best (i.e., the highest availability) grid found among the grids containing exactly N nodes and WA_N be its write availability.[2] The algorithm returns the grid G_N^{opt} as the one with the higher write availability among G_{N-1}^{opt} and G_N.

In considering N-node grids, the algorithm starts with $m = 1$ and $n = N$, and then decrements n by 1 on every iteration until $n = \lfloor \sqrt{N} \rfloor$. Hence, the number of grids examined to find G_N is $O(N)$. Since a grid with $N - 1$ nodes may have a higher availability than one with N nodes, our algorithm must also examine grids in which fewer than N nodes are used. Therefore, the total number of grids examined by the algorithm is $O(N^2)$. As a side result, the algorithm also gives the maximum availability for all grids with the number of nodes in the range $[1, N]$.

Figure 3.3 plots the order of magnitude of the write unavailability of the best grid and its relative quorum size, for various values of N up to 1000. Figure 3.3 shows that as N increases, the best achievable unavailability decreases, while at the same time the relative quorum size falls. Thus, by increasing N, one can satisfy both a minimum availability

[2] In an $m \times n$ grid containing N nodes, there are $n - (mn - N)$ columns with m nodes and $mn - N$ columns with $m - 1$ nodes. Then, the write availability of this grid can be calculated using expression 3.1 of Section 3.2.

```
OptimalWriteAvail(N): (best-avail, best-m, best-n);
   if (N = 1)
      return (p, 1, 1);
   endif
   m = 1;   n = N;
   cur-best-m = m;   cur-best-n = n;
   cur-best-avail = avail(m, n, N);
   while (n >= m)
      /* find next grid */
      n = n - 1;
      if (m n < N)
         m = m + 1;
      endif
      if (avail(m, n, N) >= cur-best-avail)
         cur-best-avail = avail(m, n, N);
         cur-best-m = m;   cur-best-n = n;
      endif
   endwhile
   (prev-best-avail, prev-best-m, prev-best-n) =
      OptimalWriteAvail(N-1);
   if (prev-best-avail > cur-best-avail)
      return (prev-best-avail, prev-best-m, prev-best-n);
   else
      return (cur-best-avail, cur-best-m, cur-best-n);
   endif;
end;
```

Figure 3.2: The algorithm for finding the grid with the highest write availability.

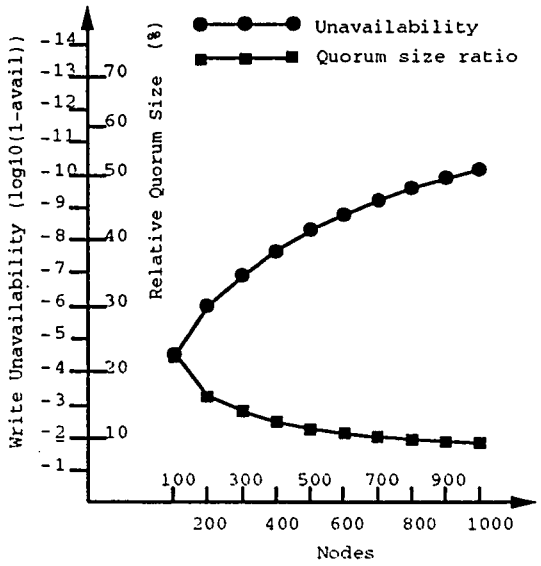


Figure 3.3: Minimum write unavailability for a given number of nodes and $p = 0.9$.

requirement and a minimum load sharing requirement at the same time. Contrast this with the majority voting algorithm, where the relative quorum size remains nearly 0.5 for all N.

Table 3.6 gives the dimensions of the best grids and the corresponding write quorum sizes. This Table shows that the maximum write availability is often achieved by not using all N nodes. For instance, in the case of $N = 30$, the best grid is 4×7, and so two nodes are not used. However, not using some nodes could negatively affect load sharing.

Table 3.6: Dimensions of grids with maximum write availability ($p = 0.9$).

N	10	20	30	500	1000
Dimensions	3×3	4×6	4×7	11×49	13×80
Write Quorum Size	5	9	10	59	92
Relative Write Quorum Size	55.6%	45%	35.7%	11.8%	9.2%

Table 3.7: Dimensions of grids with maximum combined availability and various read-write ratios ($p = 0.9$).

N	10	20	30	500	1000
Dim. for $F = .8$	3×3	4×6	4×7	11×49	13×80
Dim. for $F = .99$	3×3	4×6	4×7	11×49	13×80
Dim. for $F = .999$	2×5	4×5	4×7	11×49	13×80

Thus, suboptimal grids utilizing all available nodes may still be preferable.

Finally, we consider weighted or combined availability. Assuming F is the fraction of read operations, the combined availability is computed as: $A = F \cdot RA + (1 - F) \cdot WA$, where RA and WA are read and write availabilities. Although only grids where $m \leq n$ needed to be considered to compute the maximum write availability, imposing this condition does not always maximize read availability. Hence, all m, n combinations must be examined.

Table 3.7 shows the grids providing the best combined availability for several values of N and F. Comparing this table with Table 3.6, one concludes that the best grid configuration that maximizes combined availability is also the one that maximizes write availability alone, except for the cases where F is almost 1, i.e., when nearly all operations are reads. This supports our assertion that the modified grid protocol alleviates the tradeoff between read and write availability. It is now possible to obtain a gain in write availability at the expense of a much smaller loss in read availability than with the original grid protocol.

3.5 Asymptotic vs. Initial Behavior of Grids

In this section, we discuss the write availability of grids. It has been shown that write availability in square grids goes to 0 when N goes to infinity [36]. In Section 3.5.1, we study the asymptotic write availability of *non-square* grids. We show that an asymptotically high write availability is possible for non-square grids and characterize precisely the grid dimensions under which this is achieved. However, this result is rather pessimistic

in terms of quorum sizes, because quorum sizes in grids with asymptotically high write availability are almost linear in N. This defeats the main advantage of grids, which is small relative quorum sizes.

Therefore, in Section 3.5.2, we also examine empirically the initial behavior of WA for grids with smaller relative quorum sizes. This analysis shows that, while such grids have asymptotically low write availability, for practical values of p their write availability initially grows towards 1 and starts to decline only for extremely large values of N. Thus, these grids can often be useful in practice despite their low asymptotic write availability.

3.5.1 Asymptotic Behavior

Consider a rectangular grid arrangement of N nodes into m rows and n columns, where $m = f(N)$, $n = g(N)$ and $mn \geq N$ (the last condition says that all N nodes are used in the grid). We are interested in finding $f(N)$ and $g(N)$ such that write availability increases asymptotically towards 1 in the limit when $N \rightarrow \infty$. The following theorem proves that the asymptotic value of write availability is 1 in a small range near $n = \frac{N}{\log N}$.

Theorem 1 *In a rectangular grid arrangement of N nodes (each assumed to be up with probability p and down with probability $q = 1-p$) into m rows and n columns ($mn \geq N$), there are constants $c_1 = 2\log(1/p)$ and $c_2 = \log(1/q)$ such that:*

1. *The write availability WA of this grid asymptotically approaches 1 if $c_1 \frac{N}{\log N} \leq n \leq c_2 \frac{N}{\log N}$.*

2. *The write availability WA asymptotically approaches 0 if $n \leq \frac{c_1}{2} \frac{N}{\log N}$ or $n \geq 2c_2 \frac{N}{\log N}$.*

Proof. We first define two events (see Section 3.1 for terminology) as follows: $E1 \equiv$ All columns are alive; $E2 \equiv$ At least one column is good. Recall, from Section 3.2, that the write availability WA = Prob($E1 \wedge E2$).

The theorem follows from the following four claims, which will be proved shortly.

1. If $n \leq c_2 \frac{N}{\log N}$, then Prob($E1$) $\rightarrow$ 1.
2. If $n \geq c_1 \frac{N}{\log N}$, then Prob($E2$) $\rightarrow$ 1.
3. If $n \geq 2c_2 \frac{N}{\log N}$, then Prob($E1$) $\rightarrow$ 0.
4. If $n \leq \frac{c_1}{2} \frac{N}{\log N}$, then Prob($E2$) $\rightarrow$ 0.

Given these claims, the two parts of the theorem are proved separately below.
(Part 1)
We derive a lower bound on WA in the following manner.
WA = Prob($E1 \wedge E2$) = Prob($E1$)+ Prob($E2$)−Prob($E1 \cup E2$).
Since Prob($E1 \cup E2$) $\leq$ 1, we get WA $\geq$ Prob($E1$)+ Prob($E2$) − 1.
If $c_1 \frac{N}{\log N} \leq n \leq c_2 \frac{N}{\log N}$, then from Claims 1 and 2 we conclude that both Prob($E1$) and Prob($E2$) approach 1, which in turn implies that WA approaches 1.

(Part 2)
If $n \geq 2c_2 \frac{N}{\log N}$, then from Claim 3 we have Prob($E1$) — 0.
Similarly, if $n \leq \frac{c_1}{2} \frac{N}{\log N}$, then from Claim 4 we have Prob($E2$) $\rightarrow$ 0.
Since WA $\leq$ minimum(Prob(E1), Prob(E2)), either of these implies that WA approaches 0.
□

We shall now prove the four claims made above.

Proof (of Claims 1-4) In the introduction of algorithm `OptimalWriteAvail`, it was explained that we restrict our attention to *general* grids with at most one hole per column. To keep the exposition simple, we will use expressions for computing various probabilities for solid grids, i.e., ones with no holes. It can be verified that this does not affect the asymptotic analysis.
As shown in Section 3.2, Prob($E1$) = $(1 - q^{\frac{N}{n}})^n$ and Prob($E2$) = $1 - (1 - p^{\frac{N}{n}})^n$. Using the fact that $(1 - X)^Y = (1 - X)^{\frac{1}{X} \cdot XY} = [(1 - X)^{\frac{1}{X}}]^{XY}$ approaches $(\frac{1}{e})^{XY}$, we get that Prob($E1$) $\rightarrow$ $(\frac{1}{e})^{nq^{N/n}}$ and Prob($E2$) $\rightarrow$ $1 - (\frac{1}{e})^{np^{N/n}}$. Below, we will prove claims 1 and 2. The proofs of claims 3 and 4 are analogous and are therefore omitted.

(Claim 1)
Suppose $n \leq c_2 \frac{N}{\log N}$. Substituting the value of c_2, we get $n \leq \frac{N}{\log N} \log(1/q)$. So $(N/n)\log(1/q) \geq \log N$. $nq^{\frac{N}{n}} = \frac{n}{(1/q)^{\frac{N}{n}}} = \frac{n}{2^{\frac{N}{n}\log(1/q)}} \leq \frac{n}{2^{\log N}} = \frac{n}{N} \leq \frac{c_2}{\log N} \rightarrow 0$. Prob$(E1) \rightarrow (\frac{1}{e})^{nq\frac{N}{n}} \rightarrow 1$.

(Claim 2)
If $n \geq c_1 \frac{N}{\log N}$, then substituting the value of c_1, we get $n \geq \frac{2N}{\log N} \log(1/p)$.
So $(N/n)\log(1/p) \leq \frac{\log N}{2}$. $np^{\frac{N}{n}} = \frac{n}{(1/p)^{\frac{N}{n}}} = \frac{n}{2^{\frac{N}{n}\log(1/p)}} \geq \frac{n}{2^{\frac{\log N}{2}}} = \frac{n}{\sqrt{N}} \geq \frac{c_1\sqrt{N}}{\log N} \rightarrow \infty$. 1− Prob$(E2) \rightarrow (\frac{1}{e})^{np\frac{N}{n}} \rightarrow 0$, which implies that Prob$(E2) \rightarrow 1$. □

In summary, we have exactly characterized the conditions under which the write availability WA of a rectangular grid will asymptotically approach 1. The dimensions of the grid must be $m = \Theta(\log N)$ and $n = \Theta(\frac{N}{\log N})$; the resulting write quorum size is $\Theta(\frac{N}{\log N})$, which is very close to being linear in N.

3.5.2 Initial Behavior

The result stated above implies that, for the write availability to asymptotically approach 1, the write quorum size must be close to $N/\log N$, which is almost linear in N. This is undesirable, because it produces large quorum sizes and a low degree of load sharing. However, as we show in this subsection, this asymptotic behavior manifests itself only for very large N, so that it should not affect the usefulness of grids in most cases. Specifically, this subsection studies the write availability of grids with values of m larger than $\log N$.

Figures 3.4 and 3.5 plot the write unavailability against N for $m \times n$ grids such that $m = \lfloor N^t \rfloor$ and $n = \lceil N/m \rceil$, for several values of t and p. The write quorum size Q_W $= m + n - 1 = O(N^{1-t})$. Notice that when $p = 0.9$ and $t = 0.33$, $Q_W = O(N^{0.67})$, and write availability continues to grow until $N = 23000$. In fact, for $p = 0.99$, even write availability of a square grid ($t = 0.5$, $Q_W = O(N^{0.5})$) grows until $N = 23000$ nodes. Thus, for sufficiently large p, the write availability of grids with small write quorum sizes grows with N, approaches 1, and begins to fall only when N becomes very

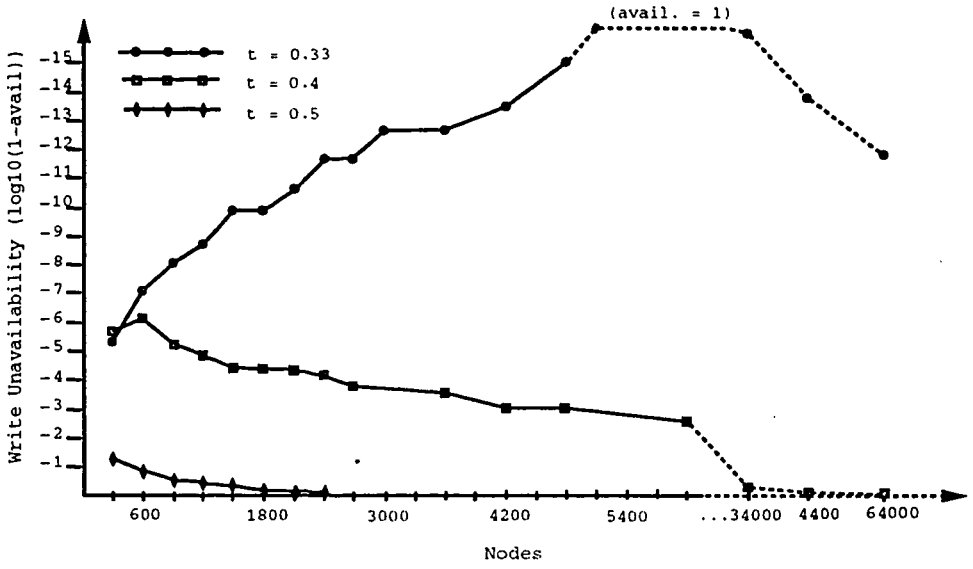


Figure 3.4: Write unavailability of $m \times n$ grids, for $m = \lfloor N^t \rfloor$ and $p = 0.9$.

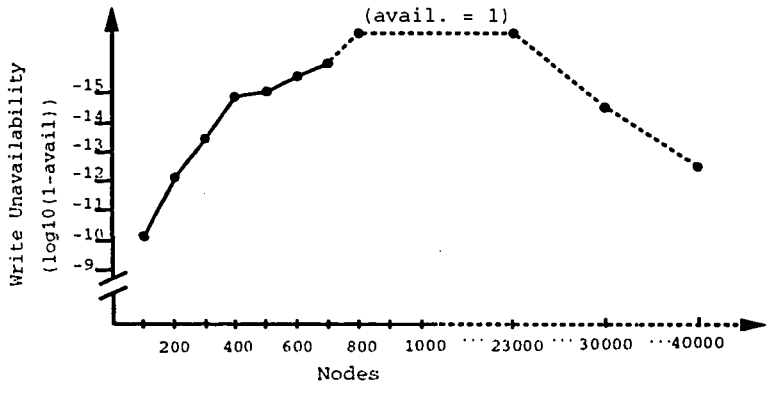


Figure 3.5: Write unavailability of square grids ($m = \lfloor N^{0.5} \rfloor$) and $p = 0.99$.

```
FindingGrid(rel-wr-quorum-size-cutoff, wr-avail-cutoff): (m, n)
// Find a grid such that its relative write quorum size does
// not exceed rel-wr-quorum-size-cutoff, its write availability
// is not lower than wr-avail-cutoff, and N is as small as possible.
   N = 4; /* No load sharing is possible for N < 4. */
   do-forever
      // Start with the grid that has the smallest quorum size for a given N.
      m = ⌊√N⌋;   n = ⌈√N⌉;
      if (mn < N)
         m = m+1;
      endif;
      cur-wr-avail = wr-avail(m, n, N);
      while ( m > 1 and cur-wr-avail < wr-avail-cutoff)
         /* Find next grid. */
         n = n+1;
         while (mn > N+n)
            m = m-1;
         endwhile;
         cur-wr-avail = wr-avail(m, n, N);
      endwhile;
      if (cur-wr-avail >= wr-avail-cutoff and
         rel-wr-quorum-size <= rel-wr-quorum-size-cutoff)
         return(m, n);
      else
         N = N + 1;
      endif;
   enddo;
end
```

Figure 3.6: The algorithm for finding the grid with a given degree of load sharing and availability.

large ($N > 23000$). We conclude that, in spite of the negative theoretical result from the previous subsection, grids are still promising in terms of both availability and load sharing in most practical situations.

3.6 Availability and Load Sharing

In this section, we turn to the problem of satisfying both a load-sharing objective and a data-availability objective simultaneously. The algorithm shown in Figure 3.6 finds the smallest N and the corresponding $m \times n$ grid that satisfy both objectives. The algorithm

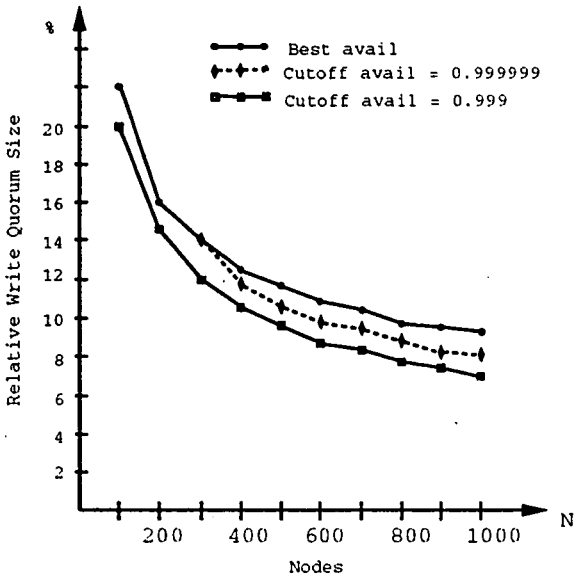


Figure 3.7: Relative write quorum size of grids with a given write availability ($p = 0.9$).

begins with a small N and calculates the write availability and relative quorum size of all grids containing all N nodes. As before, only grids with no more than one hole in any column are considered. For a given N, the algorithm examines grids in increasing order of write quorum size and stops as soon as it finds a grid satisfying both quorum size and availability requirements. It follows from Theorem 1 that this algorithm will always terminate. Indeed, a $\log N \times \frac{N}{\log N}$ grid provides asymptotically high write availability and low relative quorum size of $O(\frac{\log N}{N})$.

In Figure 3.7, the minimum relative write quorum size satisfying a cutoff write availability is plotted against N for several cutoff values. Also shown for comparison purposes is the relative write quorum size of the grids giving the *best* achievable write availability for a given N (the cutoff does not apply here).

Although the asymptotic theoretical result suggests that the relative write quorum size would drop at a very slow rate, the actual decline is much steeper for values of N used in practice. However, in view of the discussion in Section 3.5.2, this result is not surprising. Another point is that the spread of relative quorum sizes provided

by grids with different availability requirements is rather narrow. For example, when $N = 500$ and $WA = 0.999$, the relative write quorum size is 9.6%. On the other hand, for the same N and the *maximum possible* write availability (0.99999999 in this case), the relative write quorum size is 11.8%.

In this example, choosing the maximum availability grid clearly produces a large improvement in availability at only a small loss in the degree of load sharing. However, the other point to consider is the absolute value of quorum sizes. For $N = 500$, the maximum availability grid has dimensions of 11×49 and a write quorum of 59 nodes, while the grid with 0.999 availability has dimensions of 16×33 and a write quorum of 48 nodes. Therefore, although the difference in the degree of load sharing between these two grids is small (only 2.2%), the difference in absolute quorum size is more than 20%, which will translate into a large performance impact. Moreover, Figure 3.7 shows that a write availability threshold of 0.999 and a load sharing threshold of 11.8% can be achieved by a grid with only 300 nodes. Thus, it is not a good idea to always select a grid with the highest availability for a given N. Instead, the algorithm of this section should be used to choose the grid.

An algorithm similar to the one shown here can be used to deal with the case where thresholds for weighted availability and weighted relative quorum size are stated, assuming that the read-ratio F is known. The only difference is that, as explained earlier, in this case all grids containing N nodes must be considered (not just grids where $m \leq n$). For a given N, the algorithm would identify the grids with an availability greater than the threshold and choose the grid with the smallest quorum size among them. It would stop if this grid satisfies the quorum size requirement. Otherwise, the number of nodes N would be incremented and the process repeated.

3.7 Summary

In this chapter, we described a modified grid protocol that dominates the existing grid protocol, and studied various performance aspects of both solid and hollow grids. We showed that the modified protocol is optimal, and also compared its availability to that of the existing protocol. In addition, we presented algorithms for maximizing availability

in an unconstrained manner and for designing grids subject to both availability and load-sharing constraints.

We also studied the asymptotic behavior of grids and proved that, for write availability to increase asymptotically, the dimensions of an N-node grid must be approximately $\log N \times N/\log N$. This result is rather pessimistic, because quorum size in these grids is almost linear in N. On the other hand, an empirical study showed that the asymptotic behavior is significant only for very large values of N. Indeed, for practical values of the probability of a node being operational, grids with smaller quorum sizes exhibit write availability that initially grows towards 1 and starts to decline only for extremely large values of N. Therefore, in spite of the negative theoretical result on the asymptotic availability, grids are still promising in terms of both availability and load sharing in most practical situations.

Chapter 4

High-Availability Replica Management Using Structure-Based Quorums

Structure-based quorums have been proposed with the primary goal of reducing message traffic generated by user operations. Message traffic is reduced, because these quorums have smaller size than voting quorums. For instance, in the in the grid protocol [9] considered in Chapter 3, the nodes replicating a data item are viewed as arranged in a rectangular grid. A read quorum is defined as any set of nodes that includes a representative from every column of the grid, and a write quorum is defined as a read quorum plus an entire column of the grid. For square grids, the size of read quorums is $\sqrt{N}$ and the size of write quorums is $2\sqrt{N} - 1$, where N is the total number of replicas. This contrasts with the voting protocol [18], where the quorum size in the simplest case is $\lceil \frac{N+1}{2} \rceil$.

Reducing quorum size, however, reduces the number of arbitrary node failures tolerated by the protocol. Indeed, if a write quorum of replicas fails, the data item becomes unavailable, due to the intersection property of quorums. Worse, in existing structure-based protocols, operations rely on their knowledge of the statically pre-defined logical structure of the network; therefore, these protocols are static by their nature. This

means they cannot adjust the read and write quorums to reflect failures and recoveries occurring in the system. Thus, the system becomes unavailable even if a quorum of failed replicas is accumulated over time. In contrast, the voting protocol allows such re-adjustment [29] because it defines the quorums based on the number of votes regardless of their identity. As a result, the dynamic voting protocol can keep a data item available as long as there is one accessible replica, provided not too many failures occur between consecutive failure-detecting operations so that the protocol could adjust to the failures as they occur.

In this chapter, we describe a quorum re-adjustment mechanism for protocols using structure-based quorums [53]. Moreover, we argue that our approach is a preferable way to adjust quorums even in the voting protocol.

We observe that, given an ordered set of nodes, one can usually devise a rule that unambiguously imposes a desired logical structure on this set. Read and write operations can then rely on this rule rather than on their knowledge of the statically pre-defined logical structure of the network in determining which replica sets include quorums. In addition, if at any time all operations agree on a set of replicas from which the quorums are drawn, then the protocol can dynamically adjust this set to reflect detected failures and repairs and at the same time guarantee consistency.

In our protocol, we assume that each node is assigned a name and that all names are linearly ordered. Among all replicas of a data item, we identify a set of replicas considered to be the current *epoch*. At any time, a data item may have only one current epoch associated with it. Initially, all replicas of a data item form the current epoch. The system periodically runs a special operation, *epoch checking*, that polls all replicas of this item. If any members of the current epoch are not accessible (failures detected), or any replicas outside the current epoch have been successfully contacted (repairs detected), an attempt is made to form a new epoch. (Epochs are distinguished by their *epoch numbers*, with later epochs assigned greater epoch numbers.) For this attempt to succeed, the new epoch must contain a write quorum of the previous epoch, and the list of new epoch members (the *epoch list*), along with the new epoch number, must be recorded on every member of the new epoch. Then, due to the intersection property of quorums, if the network partitions, the attempt to form a new epoch will succeed in at most one

partition, and hence the uniqueness of the current epoch will be preserved. For the same reason, any successful read or write operation must contact at least one member of the current epoch and therefore obtain the current epoch list. Hence, the operation can reconstruct the logical structure of the current epoch (by using the universally known rule) and identify read or write quorums based on that structure. As in dynamic voting, the system will be available as long as some small number of nodes (the number depends on the specific quorum definition) is up and connected.

4.1 General Protocol

We assume that all nodes agree on a *quorum rule* that defines which sets of nodes constitute quorums. Given two sets of nodes, V and S, `IsWriteQuorum(`V`,`S`)` is `true` if S includes a write quorum over V, and `false` otherwise. Similarly, `IsReadQuorum(`V`,`S`)` defines read quorums. We also assume that there are *quorum functions* `ReadQuorum(`V`, name)` and `WriteQuorum(`V`, name)` that, given a set of nodes V and the coordinator name, yield a list of nodes representing some quorum over V. For better load sharing, the quorum functions should yield different quorums for different coordinators.

The protocol consists of the following asynchronous procedures: write/read (referred to as user operations), and epoch checking and switching to a new epoch if necessary (called the epoch operation or epoch-changing operation, when it actually changes epochs). Each node maintains the following state for every data item it keeps:[1] a version number, an epoch number, and a list of node names representing a current epoch (the *epoch list*).[2] Initially, all nodes have identical replicas; version numbers and epoch numbers are all 0; and epoch lists include all nodes that have a replica of the data item.

4.1.1 User Operations

[1] All considerations in this chapter apply on a per-data item basis. However, when a group of data items is replicated on the same set of nodes (such groups of data items are often called *volumes*), epoch management can be done for all these data together. Then, the epoch list and number must be maintained for every volume, rather than for every data item.

[2] As an implementation detail, sets of nodes can be encoded very tightly as, e.g., a binary vector with the i-th element set to 1 if the i-th node is included and 0 otherwise.

```
Write(input:  new-data)
// This algorithm is run by the coordinator.  Actions taken by the
// other nodes are given within the comments.
   quorum-list := WriteQuorum(my-epoch-list, my-node-name);
   multicast(quorum-list, (''write-request'', new-data));
   // Each node that receives the write-request stores new-data
   // stably, obtains the lock for its replica and responds with a
   // tuple of the form (node, version, clist, enumber), where elements
   // in the tuple are the node name, version number, epoch list, and
   // epoch number respectively.
   receive RESPONSES;
   let (node_m, version_m, elist_m, enumber_m) be a response with the
      maximum epoch number;
   if NOT IsWriteQuorum(elist_m, all-nodes-that-responded) then
      multicast(all-nodes-that-have-a-replica,
         (''write-request'', new-data));
      receive RESPONSES;
      let (node_m, version_m, clist_m, enumber_m) be a response with the
         maximum epoch number;
      if NOT IsWriteQuorum(elist_m, all-nodes-that-responded)
         abort;
      endif;
   endif;
   // A write quorum of permissions has been collected.
   let max-version be the maximum version number in RESPONSES;
   multicast(all-nodes-that-responded, (''commit-write'', max-version + 1));
   // Upon receiving this message, each node installs a new data
   // value and a new replica version number and releases the
   // replica's write lock.  */
end;
```

Figure 4.1: The algorithm for the write operation in the structure-based protocol with epochs.

The algorithm in pseudo-code for the write operation is shown in Figure 4.1. The algorithm embeds the two-phase commit protocol to make sure that the write either performs on every node from which permission had been obtained or on no nodes at all.

The coordinator sends out a request for permission to all nodes from some write quorum over its epoch list. With this request, the coordinator includes the new value of the data item. Each node that receives the request obtains the write lock for its replica, logs the update, and responds with its state. (This response signifies the node's agreement to participate in the operation; in line with the two-phase commit protocol, the participating node will be unable to unlock the replica until it learns the outcome of the operation from the coordinator.)

Upon receiving all responses, the coordinator faces two cases.

1. The coordinator has collected non-`RPC.CallFailed` responses that include some write quorum over the epoch list from a response with the maximum epoch number. (This is the branch taken in the common case of absence of failures.) In this case, the coordinator distributes a `commit-write` message to all participants that responded and returns control to the user. Along with the `commit-write` message, the coordinator includes a new version number, equal to the maximum version number reported by participants incremented by one. On receiving this message, each participant atomically performs the write locally, adopts the received version number, and releases the replica's lock.[3]

2. The coordinator has failed to collect a quorum of non-`RPC.CallFailed` responses. The coordinator sends the request for permission with the new data to *all* nodes (except, perhaps, those polled before). After receiving all responses, it checks if it has been able to obtain a quorum of non-`RPC.CallFailed` responses over the epoch list from the response with the maximum epoch number.[4] If this is the case,

[3]Sending the `commit-write` message and releasing replica locks does not mean that we consider only transactions consisting of a single write. Complex transactions are built from individual operations described here using their own concurrency control and commit protocols. However, simplifications of the presented algorithm are possible if strict two-phase locking and two-phase commit are used by transactions. See further details at the end of this subsection.

[4]Various optimizations are possible to minimize the number of nodes with which the coordinator must communicate and the number of nodes from which it must wait for responses before deciding whether to proceed with the next step.

the coordinator performs the write on all replicas that responded by distributing to them the `commit-write` message with the new version number.

If no quorum of non-`RPC.CallFailed` responses has been obtained after polling all replicas, the coordinator aborts the operation and sends out an abort message to participants so they can unlock their replicas. There is no reason to wait for a possible epoch change, because such an operation can succeed only if it can obtain a quorum as well.

The read operation, shown in Figure 4.2, is very similar to the write. The coordinator issues the request for permission for the read to all nodes from a read quorum over its epoch list. Participants obtain the read lock for their replica and respond with the state. If the set of non-`RPC.CallFailed` responses includes a read quorum over the epoch list from a response with the maximum epoch number, the coordinator reads the data from any replica that reported the maximum version number, distributes a `commit-read` message to all participants, and returns to the application.

Otherwise, similarly to the write protocol above, the coordinator tries to perform the read by contacting the rest of the replicas in the system.

Note that, if the participants include the value of the data with permission, one round of message exchange for reading from the current replica could be eliminated from the protocol: the coordinator would have that data immediately after obtaining a read quorum of permissions. The choice between including the data with permission and reading it separately depends on the size of the data and the performance characteristics of the system.

If the S2PL protocol is used in conjunction with this replica control protocol, then the locks would actually be released only during the commit protocol performed at the end of the transaction, not at the end of individual operations. Also, the coordinator of an individual operation would not have to send `write-commit` or `read-commit` messages: these messages could be piggybacked on commit messages sent by the commit protocol at the end the transaction. Individual user operations would then have only one phase.

However, we do not make any assumptions about the nature of concurrency control mechanisms used in the system (e.g., these mechanisms may use separate locks or no locks at all); we therefore describe a policy that is sufficient for the replica control protocol to

```
Read
// This algorithm is run by the coordinator.  Actions taken by the
// other nodes are given within the comments.
   quorum-list := ReadQuorum(my-epoch-list, my-node-name);
   multicast(quorum-list, ''read-request'');
   // Each node that receives the read-request obtains the read lock
   // for its replica and responds with a tuple of the form
   // (node, version, elist, enumber).
   receive RESPONSES;
   let (node_m, version_m, elist_m, enumber_m) be a response with the
      maximum epoch number;
   if NOT IsReadQuorum(elist_m, all-nodes-that-responded) then
      multicast(all-nodes-that-have-a-replica, ''read-request'');
      receive RESPONSES;
      let (node_m, version_m, elist_m, enumber_m) be a response with the
         maximum epoch number;
      if NOT IsReadQuorum(elist_m, all-nodes-that-responded) then
         abort;
      endif;
   endif;
   // A read quorum of permissions has been collected.
   let max-version be the maximum version number in RESPONSES;
   try to read data from any node that reported max-version;
   if data has been read successfully from any node above
      multicast(all-nodes-that-responded, ''commit-read'');
      // A recipient of this message releases its read lock.
      return data to application;
   else
      abort;
   endif;
end;
```

Figure 4.2: The algorithm for the read operation in the structure-based protocol with epochs.

be correct provided *any* correct concurrency control protocol is used.

4.1.2 Epoch Checking/Changing Protocol

Once enough failures are accumulated in the system to make collecting of a quorum impossible, the data item becomes unavailable. To avoid failure accumulation, there should be a relatively steady (although low, since failures are infrequent) rate of epoch-checking operations that detect failures and repairs. Also, epoch checking may be initiated immediately when a failure or repair is detected during a user operation.

Any node can initiate an epoch operation. This node is called the *coordinator* below. Figure 4.3 shows the algorithm in pseudo-code for the epoch operation. The first, preliminary, phase of the protocol simply polls all replicas to determine if a new epoch formation is needed and possible. The coordinator sends a request for epoch checking to all nodes. Each node responds with its state. Upon receiving all responses, the coordinator checks if a set of responded replicas differs from the current epoch (i.e., whether the epoch change is needed), and if this set contains a write quorum of responses over the the current epoch (i.e., whether the epoch changing is possible).

If both conditions are met, the coordinator attempts to perform epoch-changing as follows. The coordinator sends out a request for epoch *changing* to all nodes. Each node obtains the write lock for its replica and responds with its state (the state is sent again because it was previously obtained without locking replicas and, hence, may have changed). The coordinator re-checks again the necessary and sufficient conditions for epoch changing. If they are still satisfied, the coordinator forms a new epoch list that includes all nodes that responded, and determines the most recent value of the data by reading it from any replica that reported the maximum version number.

Finally, the coordinator distributes the new epoch list and number to all members of the new epoch. With this message, it also includes the current value of the data and the most recent version number. Every participant that receives this message atomically installs the new data, new replica version number, and epoch parameters. Thus, immediately after a new epoch is formed, all its members are current.[5].

[5]Alternatively, epoch changing may not include bringing obsolete replicas up-to-date. Instead, the coordinator may just include the most recent version number in the last message, in which case the

```
CheckEpoch
// This algorithm is run by the initiator of the epoch checking.
// Actions taken by the other nodes are given in the comments.
   multicast(all-nodes-that-have-a-replica, ''epoch-check-request'');
   // Each node that receives the epoch-check-request responds with a tuple
   // of the form (node, elist, enumber), where elements in the
   // tuple are the node name, epoch list, and epoch number.
   receive RESPONSES;
   let (node_m, version_m, elist_m, enumber_m) be a response with the
      maximum epoch number;
   let NEW-EPOCH be all-nodes-that-responded;
   if NEW-EPOCH ≠ elist_m and IsWriteQuorum(elist_m, NEW-EPOCH)
      multicast(all-nodes-that-have-a-replica, ''epoch-change-request'');
      // On receiving this message, a node obtains the write lock for
      // its replica and responds with the tuple
      // (node, version, elist, enumber), where elements in the tuple are
      // the node name, replica version number, epoch list, and epoch number.
      let (node_m, version_m, elist_m, enumber_m) be a response with the
         maximum epoch number;
      let NEW-EPOCH be all-nodes-that-responded;
      if NEW-EPOCH ≠ elist_m and IsWriteQuorum(elist_m, NEW-EPOCH)
         new-epoch-number := enumber_m + 1;
         let max-version be the maximum version number reported in response
            to ''epoch-change-request'';
         let cur-node be the name of any node that reported max-version;
         try to read data from cur-node;
         if the read was not successful (cur-node responded with
            RPC.CallFailed)
            abort;
         end if;
         multicast(NEW-EPOCH, (''new-epoch'', NEW-EPOCH, new-epoch-number,
           max-version, data));
         // Upon receiving this message, each node atomically updates
         // its epoch list and epoch number to be equal to NEW-EPOCH
         // and new-epoch-number and installs new data and a version number.
      endif;
   endif;
end;
```

Figure 4.3: The algorithm for epoch checking/changing in structure-based protocols with epochs.

If the coordinator of the epoch-changing operation fails after obtaining replica locks and before committing or aborting the operation, further accesses to the data item (as well as further epoch operations) will be blocked. This is because the protocol embeds the two-phase commit to ensure all-or-nothing execution. One could instead embed a known *quorum-based commit* protocol [64] into the algorithm to provide tolerance to the coordinator's failure. However, since the quorum-based commit protocol requires three phases, this would add an extra phase to the protocol described above.

In the absence of failures, epoch checking does not interfere with reads and writes. Interference may occur only when epoch changing is actually needed, i.e., if any failures or repairs have occurred since the previous epoch check. Of course, it would be preferable for both epoch checking and changing to be done asynchronously with user operations. We propose such a reconfiguration scheme for voting-based quorums in Chapter 5 and for read-one-write-all quorums in Chapter 6. Whether it can be done efficiently for structure-based quorums remains a question for future work.

4.2 Proof of Correctness

This section proves that the protocol described in Section 4.1 provides one-copy serializability when combined with the S2PL concurrency control protocol that guarantees serializability at the level of physical replicas.

The epoch operation in our protocol is equivalent to the write operation in the standard dynamic voting protocol of [32, 29], where the write is performed on a special data item containing the epoch list, and the epoch number kept by a node serves as this data's replica version number. The only difference is that our protocol uses a general function `IsWriteQuorum` to verify that an operation obtained locks from a write quorum of replicas, whereas in the dynamic voting protocol this function is concretized for majority quorums. (The existence of a preliminary epoch-checking phase in our epoch operation does not change state of any nodes.)

recipients of this message would mark themselves *stale* if their version number is less than the one received. Before data can be read from a stale replica, the replica must bring itself up-to-date by copying data from a current replica. See Chapter 7 and [53] for further details on exploiting the distinction between stale and current replicas.

Thus, epoch operations manage replicas of the epoch list in exactly the same way as writes in the dynamic voting protocol would have. In particular, the following lemma is true (see [32, 29, 4]).

Lemma 2 *(1) At all times, all nodes that have the same epoch number e also have the same epoch list* $elist_e$. *(2) At most one epoch operation at a time can lock enough replicas in its second phase to proceed with the epoch change. (3) At all times, the only quorums that can be collected consist of nodes with the current system-wide maximum epoch number.*[6] □

By Lemma 2, all nodes with the same epoch number have the same epoch lists. In addition, an epoch operation coordinator distributes a newly constructed epoch list to the members of this epoch list only. Hence, a node is always a member of the epoch list it keeps. So, we will say interchangeably that a node has epoch number e, or a node is a member of epoch e, or it is a member of epoch list e. Similarly, we will say "quorum from epoch e" meaning a quorum of nodes over the epoch list stored at nodes with epoch number e. Finally, we will say that an epoch change occurred if a new epoch number was recorded on any node.

Lemma 3 *Every successful read (write) operation must at some point hold the read (write) lock from a set of replicas from this data item, and no two logically conflicting operations can hold locks on all replicas from their respective sets simultaneously: one operation must release the lock on some replicas before another can lock all necessary replicas.*

Proof. The first part of the claim follows from the fact that no operation can succeed unless the `IsReadQuorum` (`IsWriteQuorum`) procedure verifies that all replicas from a quorum from some epoch are locked. We next need to show that these locks cannot be held simultaneously by conflicting operations.

Let p and q be any two successful user operations, one of which is a write. Let e_p and e_q be the numbers of epochs from which p and q collected their respective quorums. If

[6] Recall that all considerations are on a per-data item basis. So, the maximum epoch number is taken among nodes that have a replica of the data item.

$e_p = e_q$, both operations collected quorums from the same epoch. But, by the intersection property, any two write quorums from the same epoch - as well as read and write quorums from the same epoch - have at least one node in common. This node cannot be locked simultaneously by p and q. Thus, the lemma is correct when epochs do not change.

Now consider the case where $e_p \neq e_q$ (without loss of generality, assume $e_p < e_q$). By Lemma 2, all epoch numbers greater than e_p were introduced in the system after p collected its quorum. Also, by claim 2 of Lemma 2, all epoch-changing operations are serialized. Let s be the first epoch-changing operation that introduced into the system an epoch number e_s greater than e_p. (Thus, we have $e_p < e_s \leq e_q$.)

Operation s must have collected a write quorum before recording the new epoch number on any node. By Lemma 2, it could only be a write quorum from epoch e_p. Due to the intersection property of quorums from the same epoch, p and s could not hold the locks from their quorums simultaneously. Moreover, p must have released some of its locks before s collected its quorum. (Otherwise, e_s would have been introduced into the system before p collected its quorum, a contradiction with Lemma 2.) On the other hand, q must have collected its quorum only after e_s was introduced. (Otherwise, it would use a quorum from epoch e_p, which was the maximum epoch number before e_s was introduced into the system.) Thus, we showed that q locked its quorum after s recorded epoch number e_s on some node, which happened after s locked its quorum, which happened after p released a lock on some nodes from its quorum. □

The above lemma shows in particular that all logically conflicting operations are serializable according to the order in which they lock their quorums.

Lemma 4 *If all writes are numbered according to the order in which they lock their quorums, then the ith write records version number i on all replicas on which it performs, and any read operation reads from a replica with version number j, where j is the number of the preceding write.*

Proof. We will give the proof for writes only; the proof for reads is similar. We showed in Lemma 3 that writes are serializable according to the order in which they lock their quorums. Therefore, one can use induction on the sequential number of the write operation. (i) For the 1-st write, the lemma is trivially correct (all replicas initially have

version number 0, so the first write will distribute version number 1). (ii) Assume that the lemma is correct for the i-th write and consider the $(i+1)$-st write operation.

Case 1: No epoch change occurred between the i-th and $(i+1)$-st writes. Then, the $(i+1)$-st write must collect a quorum of the same epoch as the i-th write. Hence, these quorums will have at least one node in common. Denote this node p. By the induction hypothesis, the i-th write recorded version number i on all replicas on which it performed, including p. Hence, `max-version` found by the $(i+1)$-st write cannot be less than the version number reported by p, i.e., i. Also by the induction hypothesis, since all previous writes distributed version numbers equal to their sequential order number, at the time `max-version` is calculated by the $(i+1)$-st write, no replica can have a version number greater than i. We conclude that the $(i+1)$-st write finds `max-version` to be equal to i, and it thus distributes the new version number $i+1$ to all replicas on which it performs.

Case 2: The epoch change occurred between the i-th and $(i+1)$-st writes. Using reasoning similar to Case 1, we can show that, after the epoch change, all nodes from the new epoch are identical and have version number i, which is the maximum version number among all replicas in the system. By Lemma 2, the $(i+1)$-st write must have locked some replicas from the latest epoch. Hence, it finds `max-version` to be equal to i and distributes the new version number $i+1$ to all replicas on which it performs. □

Theorem 2 *If the S2PL concurrency control protocol is used in conjunction with our protocol, the overall execution of transactions is one-copy serializable.*

Proof. The lemmas proved above show that our protocol satisfies the conditions of Lemma 1. The claim of the theorem follows. □

In fact, as shown in [32], if the reconfiguration operation (in our case epoch changing) runs as a special update transaction, the conditions proved in Lemmas 3 and 4 are sufficient for a replica control protocol to provide one-copy serializability when combined with *any* concurrency control scheme that correctly ensures serializability on the level of physical replicas.

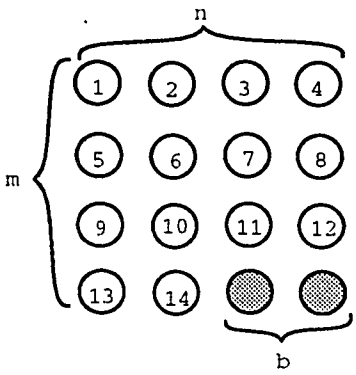


Figure 4.4: The grid for N = 14.

4.3 Example 1: Dynamic Grid Protocol

We now illustrate our approach by applying it to the grid protocol [9] considered in Chapter 3. The grid protocol arranges all nodes having a replica of the data item into a logical $m \times n$ grid (see Figure 4.4). To succeed, a read operation must collect permission from a set of nodes, such that one node is selected from each column of the grid.[7] A write operation must collect permission from a read quorum *and* from all nodes in one column. For example, in the grid in Figure 4.4, a set of nodes {1, 6, 3, 7, 11, 4} is a write quorum, because it includes a set {1, 6, 3, 4} of representatives from each column and a set {3, 7, 11} that covers all nodes in a column.

To allow the dynamic adjustment of these quorums using the general protocol described in Section 4.1, we must only design a rule to construct the grid given an arbitrary set V of ordered nodes and use this rule to define read and write quorum rules and quorum functions utilized in the protocol from Section 4.1.

As a first step, we show how a grid with m rows and n columns can be constructed given an ordered set of nodes V. Let N be the number of nodes in V. There are many subtleties involved in choosing grid dimensions for a given N. Chapter 3 studied this question in depth. Our goal here is to devise a simple and efficient rule that would unambiguously construct grids with reasonably good properties. Our rule will use the

[7] For the ease of comparison, we are using the original grid protocol of [9]. Using the modified grid protocol of Chapter 3 would only require a trivial modification of the `IsReadQuorum` function.

following guidelines for grid construction (see Chapter 3 for more of the rationale behind this design).

1. $m+n$ must be as small as possible. This determines the size of the write quorums. The fewer nodes it includes, the better the load sharing and message traffic. This implies that the rule will try to construct square grids. In fact, our rule will only allow grids where m and n do not differ by more than 1.

2. When a non-square grid is constructed, the rule chooses $m < n$. For instance, for $N = 20$, the rule will construct a 4×5 rather than a 5×4 grid.

3. Constructed grids have the property that $m \times n \geq N$. Otherwise, some of the available replicas would not be utilized. However, we want $m \times n$ to be as small as possible subject to the other constrains noted above.

This rule constructs square (or almost square) grids. We know from Chapter 3 that square grids are optimal in terms of performance, but have relatively low write availability. By introducing dynamic grid reconfiguration, we alleviate the availability concern. Thus, square grids can be used more freely.

The `DefineGrid` subroutine in Figure 4.5 returns the dimensions of the grid m, n and the number of unoccupied positions, b. It uses the fact that, among all m, n such that $m \times n = N$, $m + n$ is minimum when $m = n = \sqrt{N}$.

It is easy to see that, for the parameters returned by `DefineGrid`, b is always less than n. We assume that the unoccupied positions are all in the bottom row and right-justified. The nodes from V are assigned positions in the grid in the increasing order (columns first). For instance, for $V = \{1, \ldots, 14\}$, the rule constructs the grid in Figure 4.4; for $V = \{2, 3, 6, 7, 9, 12\}$ the resulting grid is shown in Figure 4.6.

We are now ready to give the quorum rules that, given sets of nodes V and S, tell if S includes a quorum over V. The algorithms, `IsReadQuorum` and `IsWriteQuorum`, first determine parameters of the grid defined by V. Then, `IsReadQuorum` checks if S includes a representative from each column of the grid. `IsWriteQuorum` does the same, plus it checks if S covers completely the nodes from one of the columns. The `IsWriteQuorum` algorithm is shown in Figure 4.7. The `IsReadQuorum` can be obtained by disregarding

```
DefineGrid(
   input:  integer N;
   output:
      integer m, n, /* dimensions of the grid */
                b /* number of unoccupied positions */ );
   m := ⌊√N⌋;
   n := ⌈√N⌉;
   if m*n < N then
      m := m+1;
   endif;
   b := m*n - N;
end;
```

Figure 4.5: The algorithm for defining parameters of a grid.

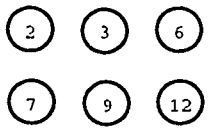


Figure 4.6: Reconstruction of the grid for N = 6.

the part that involves array COLUMN.

For simplicity, we will assume that the quorum function returns random quorums, regardless of the coordinator's name (recall that the choice of quorum function affects load sharing only). The algorithm for such a function is trivial and is not shown here.

4.4 Example 2: Dynamic Tree Protocol

A different logical structure, and the quorum definition based on it, is described in [1]. In this scheme, called the *tree protocol*, all nodes view the system as a binary tree in which each node occupies a pre-determined position. Given this tree, read and write quorums are sets of nodes defined recursively, as follows: any set of nodes that forms a path from the root to a leaf is a quorum (called a *basic quorum* below); if a path from some node A to a leaf in a quorum is replaced by paths from both children of A to leaves, the resulting set of nodes is a quorum. For instance, in the tree shown in Figure 4.8,

```
IsWriteQuorum(input:  sets of nodes V, S): Boolean;
// We assume that S ⊆ V.
   /* Variables:  */
   set of integers:  COLUMN-COVER;
   array:  COLUMN[1:n] of sets of integers;
   (m, n, b) := DefineGrid(|V|);
   COLUMN-COVER := ∅;
   for all j from 1 to n
      COLUMN[j] := ∅;
   endfor;
   for each node s from S do
      // Calculate coordinates (i, j) of the position that the
      // node s occupies in the grid.  Assume that the coordinates
      // start from (1,1).
      k := ordered-number(V, s);
      /* The function ordered-number above returns the position
      number that the node s occupies in the ordered set of
      nodes V (starting from 1).  */
      i := quotient((k-1), n) + 1;
      j := remainder((k-1), n) + 1;
      COLUMN-COVER := COLUMN-COVER ∪ {j};
      COLUMN[j] := COLUMN[j] ∪ {i};
   endfor;
   if COLUMN-COVER = {1,..., n} and there exists j such that
      (COLUMN[j] = {1, ..., m} if j <= n-b,
         or {1, ..., m-1} otherwise)
   then
      return(true);
   else
      return(false);
   endif;
end;
```

Figure 4.7: The write quorum rule in the dynamic grid protocol.

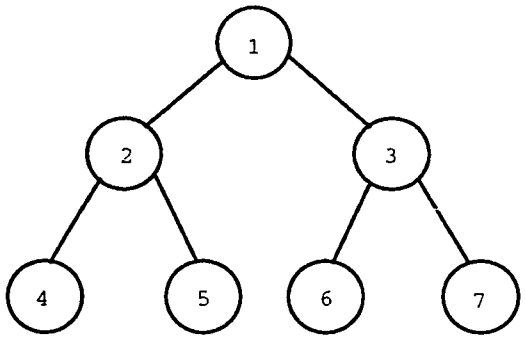


Figure 4.8: The tree for a 7 node network.

the set {1,2,4} is a basic quorum, and the set {2,4,3,6} is a quorum. It can be shown [1] that any two quorums defined this way indeed have a non-empty intersection. In the absence of failures, an operation can succeed by locking a basic quorum of nodes which requires communication with only $\lceil \log(n+1) \rceil$ nodes. The protocol can tolerate up to $\lfloor \log(n+1) \rfloor - 1$ arbitrary node failures.

In the tree protocol, quorums are not fully-distributed: nodes closer to the root are assigned heavier responsibility. Because of this, it is possible to keep message traffic low in the common case of absence of failures. If a node assigned heavy responsibility fails, other nodes help out and form non-basic quorums. Therefore, the system stays operational at the expense of worsened performance. This property of distributed protocols is known as *graceful degradation.*

In fact, graceful degradation is not always as graceful as it may seem. Once a node fails, it can remain down for a relatively long time. A node can also be down for maintenance, backups, and other reasons. In all these cases, the performance of the tree protocol will degrade for quite some time.

When applied to the tree quorums, our method of system reconfiguration not only improves system availability, but makes it possible to avoid degradation when changes in the network topology caused by failures and repairs are infrequent [54].

To concretize our general method, we need a rule to construct a tree given an arbitrary set V of ordered nodes, read and write quorum rules, and quorum functions. Given an ordered set of nodes V = `nodes[0:n]`, we can reconstruct the corresponding binary

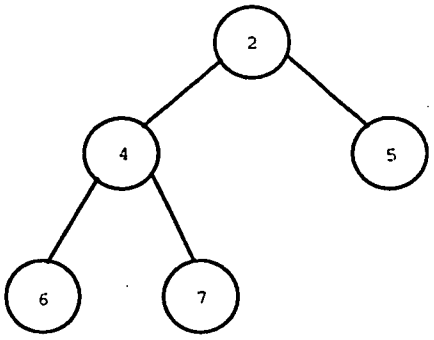


Figure 4.9: The tree of nodes $\{2,4,5,6,7\}$.

tree using a classic method: `nodes[0]` is assumed to correspond to the root; and the left and right children of `nodes[i]` are the elements `nodes[2*i+1]` and `nodes[2*i+2]`, respectively. If `2i+2 > n`, then `nodes[i]` does not have a right child. If `2i+1 > n`, then `nodes[i]` does not have any children, and thus is a leaf. For example, the tree for $V = \{1,\ldots,7\}$ is shown in Figure 4.8, and the tree for $V = \{2,4,5,6,7\}$ is depicted in Figure 4.9.

It should now be clear why our method avoids performance degradation in the presence of failures. Indeed, in the structure on Figure 4.8, if nodes 1 and 3 fail, the existing tree protocol has to lock a non-basic quorum to perform a user operation, for example $\{2,4,6,7\}$. Our protocol would reconfigure the system to the structure shown in Figure 4.9, and then the protocol would again use basic quorums from this new structure, e.g., $\{2,4,7\}$.

Since read and write quorums are the same in this protocol, read and write quorum rules are represented by a single algorithm, `IsQuorum`, on Figure 4.10. The `IsQuorum` function tells whether a set of nodes `S` includes a quorum over a set of nodes `epoch[0:n]`.

The quorum function in this protocol should always return a basic quorum, which contains the fewest number of replicas (thus, operations that use basic quorums have to communicate with fewer nodes). Again, we use a single procedure, `GetBasicQuorum`, to find both read and write quorums. The algorithm for this function is shown in Figure 4.11. This function takes an epoch list `epoch[0:n]` as an argument and returns a set of nodes that form a (random) path from the root of the corresponding binary tree to a leaf.

```
IsQuorum(epoch[0:n], S): Boolean;
    return(IsPartialQuorum(epoch[0:n], S, 0));
end;
IsPartialQuorum(tree[0:n], S /* a set of nodes */ , i):  Boolean;
// This function returns "YES" if there exists a quorum Q over the subtree of
// tree[0:n] with the root in tree[i], such that Q ⊆ S.
    if n < 2*i+1 then /* tree[i] is a leaf */
        if tree[i] ∈ S then
            return(true);
        else
            return(false);
        endif;
    endif;
    if n < 2*i+2 then /* tree[i] has only one child */
        if tree[i] ∈ S and tree[2*i+1] ∈ S then
            return(true);
        else
            return(false);
        endif;
    endif;
    if tree[i] ∈ S then
        return(IsPartialQuorum(tree[0:n], S, 2*i+1) or
                IsPartialQuorum(tree[0:n], S, 2*i+2));
    else
        return(IsPartialQuorum(tree[0:n], S, 2*i+1) and
                IsPartialQuorum(tree[0:n], S, 2*i+2));
    endif;
end;
```

Figure 4.10: The quorum rule in the dynamic tree protocol.

```
GetBasicQuorum(epoch[0:n]):  Quorum;
    return(GetPath(epoch[0:n], 0));
end;
GetPath(tree[0:n], i):  SetOfNodes;
// This function takes an array representation of a tree and an index as
// arguments and returns a set of nodes that form a a path from the node that
// corresponds to tree[i] to a leaf.
    if n < 2*i+1 then /* tree[i] is a leaf */
        return({tree[i]});
    endif;
    choose
        /* The node tree[i] has a left child.  */
        (n >= 2*i+1) -> return({tree[i]} ∪ GetPath(tree[0:n], 2*i+1));
        /* The node tree[i] has a right child.  */
        (n >= 2*i+2) -> return({tree[i]} ∪ GetPath(tree[0:n], 2*i+2));
    endchoose;
end;
```

Figure 4.11: The algorithm for finding a basic quorum over a tree.

(As in the case of grids, we assume that the function returns a random quorum regardless of the coordinator's name.) The pseudo-code uses Dijkstra's guarded commands notation to express non-determinism [12].

4.5 Availability

We now analyze data availability provided by our system reconfiguration mechanism. The analysis is carried out on the grid quorums example. We concentrate on the write availability provided by the dynamic grid protocol as compared to the static one. We omit the analysis for read availability, which is completely analogous.[8] We will use the *site model* of availability [29] because a similar model is used in [9] for the static grid protocol with which we compare our results. Assumptions in the site model are: (1) communication links are reliable, so only sites can fail; (2) failures and repairs at the various nodes are independent Poisson processes with rates λ and μ, respectively; (3)

[8] In addition, because any read quorum in the grid protocol is a proper subset of a write quorum, the read availability is strictly better than the write availability; so the latter can serve as a crude lower bound for the former.

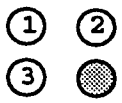


Figure 4.12: The grid for N = 3.

no failures or repairs can occur during any operation (operations are instantaneous). In addition, we assume that (4) epoch checking-operations run frequently compared to failures and repairs (after any failure or repair, epoch checking always runs before the next failure or repair).[9]

The analysis uses Markov chains and goes along the lines of [29]. Initially, all N nodes are in the latest epoch. As nodes fail and get repaired, the epoch-checking operation, according to the site model assumptions, instantaneously updates the latest epoch. Due to assumption (4) of the site model of availability, only one failure can occur between two consecutive epoch checking operations. Our grid reconstruction rule, `DefineGrid`, provides that any grid that contains at least four nodes tolerates a single failure (i.e., such a failure does not prevent a write quorum of operational nodes from being found). Therefore, the above process of epoch changes continues successfully, provided there are at least three operational replicas in the system at all times. During this time, the data item is available for user operations, since they can collect quorums from the current epoch.

If there are only three nodes in the latest epoch and one of them fails, data availability and the possibility of an epoch change depend on the identity of the failed node. For example, in Figure 4.12, if node 2 fails, no write or epoch changing is possible until this node comes back up, no matter how many other nodes get repaired. Indeed, for these operations to succeed, they would need to lock a write quorum from the grid in Figure 4.12, and any such quorum includes node 2. However, a failure of either node 1 or node 3 does not prevent these operations.

For simplicity, we will conservatively assume that, in a situation where there are

[9]The last assumption replaces the more restrictive assumption that write operations arrive frequently, which was used in [29] to analyze the availability of the dynamic voting protocol. Our assumption is less restrictive because, while an overall rate of writes is higher than that of epoch checking, the former is highly irregular and beyond control of the system.

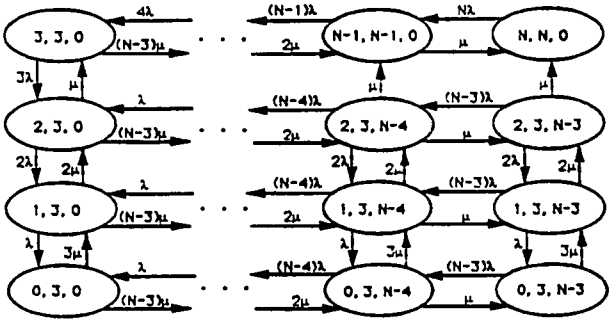


Figure 4.13: The state diagram for the dynamic grid protocol.

only three nodes in the latest epoch and one of them fails, subsequent write and epoch-changing operations will fail until all three nodes become simultaneously available again. Once all three nodes from the latest epoch are repaired, a new epoch can be formed to include all other nodes that are up at the moment.

The state diagram is shown in Figure 4.13. State (x, y, z) is the state in which the latest epoch contains y nodes: x of the y nodes from that epoch are up; z of the $N - y$ other nodes are up. The system is available for write operations if it is in one of the states in the upper row.

We use the classical *global balance* technique (see, for example, [70]) to solve the diagram, i.e., to find the probabilities of the system being in particular states. Then, the availability of the system, which is equal to the probability that the system is in one of the states in the upper row of the diagram, is calculated as the sum of the individual state probabilities. For compatibility with [9], the diagram has been solved for the same probability that a node is up (i.e., $p = 0.95$) which is achieved when $\mu/\lambda = 19/1$. Also following [9], we present the results in the form of *unavailability*, i.e., 1 - availability.

Table 4.1 shows the best write unavailability achieved by the conventional grid protocol for various numbers of nodes taken from [9] (it can vary depending on the dimensions of the grid) and the unavailability provided by our protocol. The improvement is significant and is achieved while preserving the good load-sharing and message traffic characteristics of the conventional grid protocol. Additional traffic caused by periodic epoch checkings should not be noticeable, because epoch checking is an infrequent operation

Table 4.1: Unavailability of a conventional and dynamic grid with p = 0.95.

Num. of Nodes	Static Grid		Dynamic Grid
	Best Dimensions	Unavailability	Unavailability
9	3×3	3268.59×10^{-6}	0.18×10^{-6}
12	3×4	912.25×10^{-6}	0.6×10^{-10}
15	3×5	683.60×10^{-6}	1.564×10^{-14}
16	4×4	1208.75×10^{-6}	negligible
20	4×5	250.82×10^{-6}	
24	4×6	78.23×10^{-6}	
30	5×6	135.90×10^{-6}	

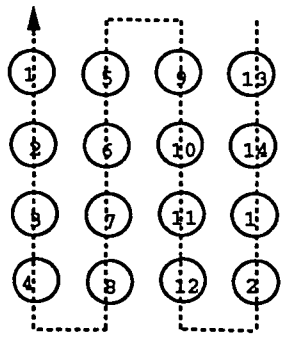


Figure 4.14: Pâris and Sloope's grid for N = 14.

done in the background.

4.6 Related Work

Independently of (and published slightly later than) the protocol presented here, two other ideas addressing the availability problem of structure-based quorums have been proposed.

4.6.1 Pâris and Sloope's Protocol

In [51], Pâris and Sloope proposed a method for adjusting grid quorums that keeps the size of columns in the grid constant. If the number of positions in the grid exceeds the

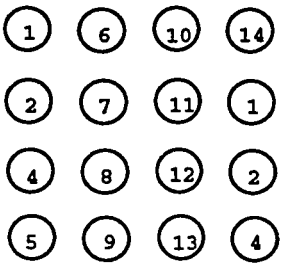


Figure 4.15: Paris and Sloope's grid for N = 14 when node 3 fails.

number of replicas, some replicas are placed into duplicate positions. For example, in a grid structure for fourteen replicas shown in Figure 4.14, replicas 1 and 2 occupy two positions in the grid. If some replicas fail but collecting a write quorum is still possible, the grid structure is adjusted (using a procedure similar to our epoch changing) by shifting the remaining replicas along the arrow in Figure 4.14. Extra unoccupied grid positions are assigned as duplicates to operational replicas. So, if replica 3 in Figure 4.14 fails, the new structure will look like that in Figure 4.15. When the right-most column contains only replicas assigned to multiple positions, this column is disregarded, leaving the grid with one fewer column. Finally, if the number of columns drops to two, the protocol switches to the dynamic voting protocol [29] (also outlined in Chapter 2), because a grid with only two large columns is highly vulnerable to failures (failure of a single node from each column would shut off the system).

While close in spirit to our approach, this scheme lacks the generality of our protocol, since it addresses grid quorums only. Also, the grid reconstruction rule in this protocol has certain disadvantages as compared to the rule proposed in Section 4.3. Because some replicas are assigned duplicate positions, it can be shown that, for any given arrangement, our scheme achieves better data availability.[10] In addition, as more failures occur in the system, the grid structure in this protocol gets "skinnier," since column size remains the same while the number of columns decreases. This causes undesirable

[10]This shortcoming, however, is easy to fix: as in our grid reconstruction method, extra positions in Pâris and Sloope's grid can be left unoccupied.

changes in the performance properties of the system (see Chapter 3 for details on how grid dimensions affect performance). In our method, grid dimensions remain proportionally the same. Finally, our method does not have to resort to dynamic voting as the number of operational replicas falls below a certain threshold. (In fact, this last feature of Pâris and Sloope's protocol results from the just-mentioned property that their grid gets skinnier as failures accumulate. At some point, it gets so skinny that using grid quorums in no longer feasible.)

4.6.2 Agrawal and El Abbadi's Protocol

Another method of system reconfiguration, proposed in [3], is based on the concept of a *view* [14, 15, 16]. Like our epochs, views are sets of nodes that are believed to be operational and connected. Views are also identified with an identification number, and later views have greater view numbers. Unlike epochs (as described in this chapter), views are managed system-wide rather than per set of nodes replicating a given data item. Also unlike epochs, a new view can always be formed in a partition with no restrictions (recall that a new epoch must contain a write quorum from the previous epoch).

However, to ensure one-copy serializability, not all data items represented in a view can be accessed by user operations. A data item is marked *accessible* in a view only if it includes all replicas from a *reconfiguration quorum*, which is defined statically over the initial set of replicas of the data item. These reconfiguration quorums are defined to intersect each other and any write quorum. Data availability is improved, because read and write quorums are defined dynamically over the replicas represented in a view. When failures/repairs are detected, a special *reconfiguration* transaction is executed in an atomic step. This transaction installs the new view by recording on all member nodes the new view number, the view membership, and a list of data items accessible in this view. For every accessible data item, the reconfiguration transaction also determines the current value and updates all replicas to be included in the view accordingly.

To apply this approach to grids, the authors begin by generalizing the definition of grid quorums. Given an $m \times n$ grid and two integers, l and w, a read quorum is defined

as any set that includes l replicas from each of w different columns, and a write quorum includes a read quorum plus $m - l + 1$ replicas from each of $n - w + 1$ different columns. This definition reduces to the definition used in Section 4.3 when $l = 1$ and $w = n$.

The authors then concentrate on the case where $l = 1$ and $w = \lceil \frac{n}{2} \rceil$ and define a reconfiguration quorum to be any set that includes more than half of the replicas from more than half of the different columns. For example, in the grid from Figure 4.4, nodes $\{1, 6\}$ form a read quorum (they include a replica from half of the columns), nodes $\{1, 2, 4, 5, 6, 8, 9, 10, 12, 13, 14\}$ form a write quorum (they include all replicas from a majority of columns), and nodes $\{2, 4, 5, 6, 8, 9, 11, 14\}$ provide an example of a reconfiguration quorum. Moreover, read and write quorums are defined dynamically over a set of replicas present in the view, as follows. Let us call a replica present in a view an *available* replica, and a column represented by any available replicas an *available* column. If a data item is marked accessible in the view, and there are t available columns in the view, a write quorum in the view is defined as any set containing all replicas that are available in the view from a majority of columns; a read quorum is a set containing any available replica from $t - \frac{n}{2}$ available columns. It can be shown that reconfiguration quorums defined above intersect with each other and with any dynamically defined write quorum, provided the latter is collected in a view where the data item is accessible.

For example, in the grid from Figure 4.4, if all replicas in the first row and the second column fail (see Figure 4.16), a reconfiguration transaction would mark this data item as accessible in the new view, because the remaining replicas include a reconfiguration quorum $\{5, 9, 13, 7, 11, 8, 12\}$. In the new view, there are $t = 3$ available columns. A write quorum is formed by locking all available replicas in all of these columns, i.e., only one write quorum containing all available replicas can be formed in this case. A read quorum can be formed by locking any available replica from any $t - 2 = 1$ available column. Thus, any single available replica constitutes a read quorum.

The main difference between this protocol and our approach is that new views can be formed with no restrictions. Consequently, data items are accessible in a view only if they are represented in the view with enough replicas to form a reconfiguration quorum. Because reconfiguration quorums are defined statically, data availability suffers. Indeed, any additional failure in the grid on Figure 4.16 would cause the reconfiguration

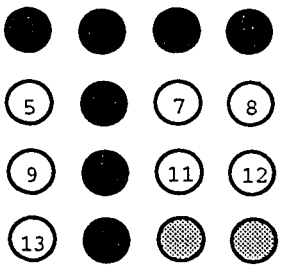


Figure 4.16: Agrawal and El Abbadi's grid for N = 14 when nodes in the first row and second column fail.

transaction to mark the data item inaccessible, since no reconfiguration quorum can be collected. In general, for square grids, the views protocol can never provide data access if more than three-quarters of replicas of the data fail. (Indeed, this makes collecting a reconfiguration quorum impossible.) Our protocol continues providing access as long as there are three operational replicas, independently of the total number of failures.

Also, Agrawal and El Abbadi's approach relies on finding a suitable definition for reconfiguration quorums, so that they intersect with any dynamically defined write quorum. Defining such reconfiguration quorums is not always easy. For instance, it seems that no reasonable definition exists for the grid quorums when $l = 1$ and $w = n$, which were used in Section 4.3 to illustrate our approach. Triangular lattice quorums [72] offer another example where it is not clear how to define reconfiguration quorums properly. Thus, we conclude that the generality of this approach is limited by a non-trivial problem that must be resolved in an ad hoc manner in every case.

4.6.3 Dynamic Voting Protocol

Our epochs are in essence analogous to distinguished partitions in dynamic voting [29]. However, the members of our epochs know the full list of members of their epoch, as opposed to just the cardinality of the distinguished partition. This allows structured quorum protocols to become dynamic.

In addition, our epoch management separates read and write operations from the operation that checks whether epoch changing is needed. This separation has many benefits. First, epoch checking must be done much less frequently than reads and writes.

It makes sense, then, to separate frequent operations from less frequently needed work. Second, epoch checking requires an attempt to communicate with all nodes, whereas we want to avoid this in reads and writes. Third, while a stream of reads and writes is beyond the control of the system and usually dries out during off-hours, we want a steady (albeit infrequent) pulse of epoch-checking operations to avoid the accumulation of failures.[11] Finally, if several data items are replicated on the same set of nodes, epoch management can be done for this whole group of data. Thus, the overhead is amortized over several data items. In contrast, if epoch management is bundled with writes, it must be done separately for each data item.

The style of system reconfiguration that we argue for, one in which the system first checks in the background for any changes in system topology and then runs a special atomic reconfiguration operation if needed, was first proposed in [4] (see their *group consensus protocol*). Of course, it would be even more beneficial if both checking for failures/repairs *and* actual reconfiguration could be done asynchronously with user operations. We propose such a reconfiguration scheme for voting-based quorums in Chapter 5 and for read-one-write-all quorums in Chapter 6. Whether it can be done efficiently for structure-based quorums remains to be seen.

4.7 Summary

This chapter describes a general method that allows dynamic adjustment of quorum sets when quorums are defined based on a logical network structure. Thus, this technique to improve availability of replica control protocols, previously applicable only to voting protocols, has been generalized to become applicable to more efficient structure-based protocols as well. Improving availability in these protocols is especially important, because these otherwise efficient protocols normally exhibit lower data availability than do protocols using majority-based quorums.

We illustrate the general approach by applying it to two structures proposed in the literature, grids and trees. The availability analysis carried out on the example of the grid

[11] Fixing this problem by running "dummy" writes, as suggested in [29], can hurt performance, since the dummy writes may interfere and hence delay the normal reads and writes. In contrast, epoch checking does not interfere with reads and writes in the absence of failures.

quorums shows the significant improvement achieved by our approach. For tree quorums, our approach has the additional benefit of avoiding performance degradation when some replicas are down. Indeed, after a failure or the planned downtime of some non-leaf node, the performance of the existing tree protocol degrades and stays at the lower level until all nodes in the system become accessible again. In contrast, in our protocol, after a new epoch is formed to exclude non-operational replicas, user operations perform at least as efficiently as in the existing tree protocol with no failures.

Chapter 5

Asynchronous System Reconfiguration in Replica Management with Voting-Based Quorums

Thus far, the focus of our discussion has centered on quorums based on a logical structure imposed on the system. In the next two chapters, we turn to more traditional *voting-based* quorums. The simplest instance of these quorums, majority quorums, was introduced in Section 1.2.

In many practical situations, voting-based quorums are preferable to structure-based ones. The main advantage of structure-based quorums is their small relative size. However, this advantage becomes significant only when the number of replicas is large; otherwise, voting-based quorums are more attractive. As an extreme example, for $N = 3$, the majority quorum size is 2, the same as for the tree quorums, while there is no grid arrangement for three replicas that would tolerate an arbitrary single node failure (see Figure 4.12 for a sample arrangement).

In voting-based quorums [18], each node replicating a data item is assigned some

number of votes; two numbers, read and write quorum thresholds r and w, are chosen such that $r + w > v$ and $2w > v$, where v is the total number of votes assigned to nodes that have a replica of the data item. A read quorum is defined as any set of nodes with a total number of votes equal to or exceeding r, and a write quorum is any set of nodes with a total number of votes equaling at least w. Voting-based quorums reduce to majority quorums if every node is given a single vote and $r = w = \lceil \frac{v}{2} \rceil$. Read-one-write-all quorums can be viewed as voting-based quorums where every node has a single vote, $r = 1$, and $w = v$.

In all quorum-based solutions to replica management, any write quorum of replicas being down makes the data item unavailable, that is, even though there are up-to-date operational replicas available, access to the data item is denied. Therefore, it is desirable to avoid failure accumulation by reconfiguring the system to reflect failures and repairs occurring in the system. Then, if not too many failures occur between consecutive node-polling operations that detect failures, the system can adjust to failures as they occur. The data item will then be available as long as some small number of replicas (the number depends on the specific quorums used) are up and connected. We described a protocol that allows such reconfiguration for systems using structure-based quorums in Chapter 4. This chapter presents an efficient method for system reconfiguration when voting-based quorums are used [58].

Many protocols for voting-based quorum re-adjustment[1] have been proposed. Some of these protocols perform quorum re-adjustment within the protocol for the write operation (see the dynamic voting protocol of [29], also described in Section 2.1.2). This approach eliminates the need for a separate mechanism for failure detection and quorum re-adjustment. On the other hand, failures and repairs are detected only during write operations. Therefore, when the rate of writes declines, failures may accumulate unnoticed by the quorum-adjustment mechanism. Fixing this problem by running "dummy" writes, as suggested in [29], can hurt performance, since dummy writes can interfere with and delay "normal" read and write operations. Also, this scheme involves communicating with all replicas (as opposed to a write quorum of replicas) on every write operation,

[1] A system reconfigures by changing the quorum sets used for replica management. Therefore, we use "system reconfiguration" and "quorum re-adjustment" interchangeably as synonyms.

making writes less efficient.

In [16, 53] and in the group consensus protocol of [4], a check for changes in the system topology is done as a separate operation, asynchronously with user reads and writes. However, actual system re-adjustment is mutually exclusive with user operations. Thus, it can interfere with the latter. This problem can become severe in an unstable network, where transient failures and repairs can cause frequent (and intrusive) system re-adjustments. On the other hand, user operations in these schemes can delay system reconfiguration. Sometimes, when the rate of user operations is high, or when failures occur during the execution of a user operation, this delay can be very long, which can negatively affect system availability.

In this chapter, we describe a protocol in which both the detection of changes in the system topology and actual system re-adjustment are done completely asynchronously with reads and writes [58]. Thus, user operations can be executed while the system reconfigures, and no interference is possible.

In addition to the traditional problem of system reconfiguration in the aftermath of failures and repairs, the ability of our protocol to include/exclude nodes asynchronously with user operations allows it to be applied in other areas as well. For example, in a mobile computing system, if a user moves from Seattle to New York, our protocol would allow the set of servers on which the user's data is replicated to change asynchronously, without any service interruption, to a set of servers that are close to the current user location. Thus, the user's data could smoothly follow the user as (s)he moves. Previously, such a change in the set of servers would require delaying service while the reconfiguration is in progress. Another possible application is asynchronous file migration in a wide-area shared-data environment, where data is moved close to the most frequent user, with data location following the access patterns. Again, the fact that such migration does not involve any service interruption allows more flexible policies to be used.

5.1 The Protocol

Our protocol uses the notion of an *epoch*, introduced in Chapter 4, which is a set of replicas of the data item believed at some point to be operational and connected. Origi-

nally, all replicas of the data item form the current epoch. The system periodically runs a special operation, epoch checking, that polls all the replicas. If any members of the current epoch are not accessible (failures detected), or any nodes outside the current epoch are contacted (repairs detected), an attempt is made to form a new epoch. For this attempt to succeed, the new epoch must contain a write quorum of replicas drawn from the previous epoch. Then, due to the intersection property of write quorums, if the system partitions, the attempt to form a new epoch will be successful in at most one partition.

The protocol consists of three asynchronous procedures: read/write (user operations); epoch checking and switching to a new epoch if necessary (the epoch operation or epoch-changing operation, when it actually changes epochs); and recovery. To highlight the main idea and simplify the proofs, we describe in this section a bare-bones protocol, where each node is assumed to have a single vote, and read and write quorums are majority sets of nodes. Extensions that generalize our protocol to weighted voting and non-equal read and write quorum thresholds are described in Section 5.2.

Every node i in the system maintains the following state for every data item it keeps: *enumber*, the epoch number of i's epoch; epoch list $ELIST$, a list of node names that are allowed to contribute to quorum formation, normally consisting of all members of i's epoch; *total-votes*, the total number of votes, based on which majority sets are determined (we refer to epoch list, epoch number, and the total number of votes together as *epoch parameters*); *stale*, a flag indicating that the replica may be stale and must recover before any read operation on this replica is allowed; and *version*, a version number of the replica, with the meaning depending on the value of the *stale* flag. For non-stale replicas, *version* indicates how fresh the replica is; for a replica marked stale, it gives a *desired* version number, i.e., the version number of replicas from which this replica can obtain the data during recovery.

Finally, i maintains a log, where it records identities of operations that locked but subsequently released i's replica. The log records referring to an operation p are purged by a distributed garbage-collector when all nodes ever locked by p, or on behalf of p, have been released. Hence, a node keeps the record for p only from the time its lock is released by p until *all locks* held by p are released. Since we assume strict two-phase locking, where

all locks acquired by a transaction are released at once after the transaction commits, this period of time is typically very short, and the log contains few (if any) records.

Initially, all nodes are members of epoch 0, and all replicas are current. So, on all nodes, *enumber* = 0, *ELIST* includes all nodes in the system, *version* = 0, *stale* is set to "*NO*," and *total-votes* equals the number of all nodes in the system.

5.1.1 Overview

We begin by describing informally the ideas behind our protocol. The novelty of our approach is in the *method* of switching to a new epoch, one that allows reads and writes to proceed while epoch switching is in progress. In our scheme, all replicas keep control information that restricts the ways in which quorums can be formed. When switching from the current epoch to a new epoch, the protocol specifies transitional values of the control information in such a way that any quorum formed according to the transitional control information includes a quorum from the current epoch as well as a quorum from the new epoch.

Epoch switching is then done in two stages. First, transitional control information is asynchronously recorded on all members of the new epoch. During this stage, some read and write operations may use quorums from the current epoch, while others may used quorums defined by the transitional control state. However, since the latter quorums always include the former ones, all operations will in effect execute in the current epoch.

Once a write quorum of replicas from the current epoch confirms switching to the transitional state, the second stage of the protocol begins, during which replicas asynchronously switch to the new epoch. During the second stage, some reads and writes may use "transitional" quorums, while some may already use quorums from the new epoch. But since any transitional quorum includes a quorum from the new epoch, all operations will in effect execute in the new epoch. Thus, the overall effect of protocol execution is as if the system switched from the old to the new epoch instantaneously at the moment the second stage of the protocol began. Then, with a little care in a situation where a user operation obtains permission from a quorum of replicas before and executes after "instantaneous" epoch switching, the consistency of replicated data

can be guaranteed.

User operations define quorums based on epoch parameters obtained with permissions from replicas. An operation is considered to have collected a quorum if it obtained more than half of the *total-votes*$_m$ of permissions from nodes that are members of the epoch list *ELIST*$_m$, where *total-votes*$_m$ and *ELIST*$_m$ are epoch parameters from a permission message with the greatest epoch number.

Consider a system with 5 replicas, x_1 through x_5, where every replica has a single vote and both read and write quorums are defined as majority sets. Assume that all of the nodes initially have epoch number 0, the epoch list that includes all replicas, and *total-votes* = 5.

Assume that x_4 and x_5 fail, and the failure detection mechanism invokes epoch switching to form a new epoch consisting of replicas x_1, x_2, and x_3. The epoch-changing operation will use *transitional values* of *ELIST* and *total-votes*. The transitional *ELIST* includes the intersection of the old and new epoch lists, in our case, $\{x_1, x_2, x_3\}$. The transitional *total-votes* is equal to the number of nodes in the union of the old and new epoch lists. In our case, the old epoch includes the new one, so the transitional *total-votes* remains 5. This transitional information specifies that quorums must contain three replicas (more than half of the transitional *total-votes*), and the replicas allowed to participate in any quorum must be among x_1, x_2 and x_3 (members of the transitional epoch list). Thus, the only transitional quorum possible in our case includes all three active replicas. Clearly, this quorum includes a quorum $\{x_1, x_2, x_3\}$ from the old epoch as well as a quorum from the new epoch (e.g., $\{x_1, x_2\}$).

The epoch-changing operation will first update to the transitional values *ELIST* and *total-votes*, kept by the members of the prospective epoch. Thus, in this stage, replicas x_1, x_2, and x_3 will get transitional *ELIST* and *total-votes* and keep their old epoch number. During this stage, some reads and writes may still use quorums from the old epoch, and some may already use transitional quorums. However, since the only possible transitional quorum also includes a quorum from the old epoch, all operations in effect continue using quorums from the old epoch.

Only after the transitional values are reflected in a quorum of replicas from the old epoch (in our case, after x_1, x_2, and x_3 confirm adopting the transitional state), can

the second stage of the protocol begin. During this stage, the state of the new epoch's members is updated to the new epoch parameters ($ELIST = \{x_1, x_2, x_3\}$, *enumber* = 1, and *total-votes* = 3). At any time during this stage, all nodes from some write quorum from the old epoch have either transitional or new epoch states. Thus, any successful user operation that executes after this stage begins is guaranteed to see either transitional or new values of epoch parameters and therefore to use either transitional or new quorums. Since any (or, in our case, the only) transitional quorum includes a quorum from the new epoch, all user operations effectively use new quorums during this stage. The only difference is that operations using transitional quorums must obtain one extra permission.

This execution is equivalent to instantaneous epoch switching at the moment the second stage begins. Hence, if at this time there are no user operations in progress, the consistency of replicated data is guaranteed, as in the existing schemes where epoch switching is mutually exclusive with user operations.

We now consider the situation where the second phase of epoch switching begins while a user operation is underway.[2]

Assume that the epoch-changing operation from the previous example finds x_3 locked for a write operation, i.e., x_3 has given its permission for a write operation to a write coordinator and is waiting for a message from that coordinator instructing it to perform the update (the *update* message). Furthermore, replicas x_1 and x_2 are found to have version number 1, while x_3 has version number 0.

Several scenarios could have led to this situation. One possible scenario is that a write w_1 executed on replicas x_1, x_2, x_5 and gave them version number 1; then another write, w_2, collected permission from a quorum $\{x_3, x_4, x_5\}$, performed on x_4 and x_5 (giving them version number 2), and, while the update message from w_2's coordinator to x_3 was in transit, x_4 and x_5 became partitioned from the rest of the system. In another scenario, a write w_1 collected permission from a quorum $\{x_1, x_2, x_3\}$, performed on x_1 and x_2 (giving them version number 1), and then, while the update message to replica

[2]We do not consider an example where newly repaired nodes are included in the new epoch. The reader is encouraged to work through such an example using the general description of the protocol in Section 5.1.2. Also useful would be considering an example in which some repaired nodes are being included in the new epoch concurrently with some newly failed nodes being excluded.

x_3 was in transit, x_4 and x_5 became partitioned.

If actions taken during epoch switching were limited to only those described in the first example, then, after the new epoch is formed, a read operation could succeed by collecting a quorum $\{x_1, x_2\}$ and therefore would not locate the latest replica in the first scenario described. This situation can be dealt with in one of two ways, orthogonal to the main idea of the protocol (which is a two-stage epoch switching via a transitional state). One way is simply to postpone the final phase of transition to the new epoch until x_3 unlocks its replica, and then finish the transition to the new epoch. So, in the first phase of transition, when x_3 receives the transitional parameters, it updates its state but does not respond to the epoch-changing coordinator until the completion of the user operations that locked x_3 before it switched to the transitional state. The epoch-changing coordinator begins the final phase of new epoch formation only after a write quorum of replicas from the current epoch (in our case, x_1, x_2, and x_3) responds.

In this approach, user operations never wait for epoch operations (hence, epoch changing never interferes with user operations). Moreover, since an epoch-changing operation must wait for only those user operations on a given replica that locked the replica *before* it adopted the transitional state (and replicas adopt the transitional state immediately when the message carrying this information arrives), there is no possibility of starvation of epoch-changing operations. The advantage of this approach is its simplicity: the protocol using this approach requires only three phases [57]. The drawback is that, if a user operation coordinator fails before unlocking some nodes, epoch-changing operations may be blocked for a long time.

The other way to handle this problem, first sketched in [4] to deal with a similar situation and used in our protocol here, is to propagate x_3's lock to x_1 and x_2 and assign x_3 the responsibility of unlocking x_1 and x_2 after its lock is released. Also, x_1 and x_2 can unlock themselves by contacting a node that committed the operation holding their locks. In this approach, one should be careful to avoid excessive lock propagation. In our example, if the system evolved according to the second scenario, x_1 and x_2 already finished the write that is locking x_3. Therefore, propagating x_3's lock to them is not only unnecessary, but harmful: if x_3 fails before unlocking either of the remaining replicas, the data item will be permanently blocked for future accesses until x_3 recovers. We use

the logs to distinguish between the two scenarios. In the first scenario, there will be no record referring to w_2 in the logs, and both x_1 and x_2 will mark themselves as locked for w_2. In the second scenario, x_1 and x_2 will find a record referring to w_1 in their logs, and will not accept lock propagation.

5.1.2 Epoch Changing Protocol

The epoch operation can be initiated by any node, called the *coordinator* below. The operation as described here executes in five phases. At the end of this subsection we show how to reduce this number to four.

Phase 1. The purpose of the first phase is to decide whether forming a new epoch is needed and possible, and to arbitrate among possibly competing epoch operations, so that at most one operation can proceed to the second phase.

During the first phase, the coordinator sends a request to all nodes. If it is not already participating in another epoch operation, each node responds with its epoch parameters. (Otherwise, it ignores the request.) Upon receiving all responses, the coordinator checks if it has been able to obtain responses from a write quorum, as defined by the epoch parameters from a response with the maximum epoch number. (We denote these epoch parameters as $enumber_m$, $ELIST_m$, and $total\text{-}votes_m$ below.) If this is the case, $ELIST_m$ and $total\text{-}votes_m$ are in fact the epoch parameters of the current (most recent) epoch. Once the coordinator obtains the current epoch parameters, it checks whether a set of nodes that responded, $ELIST_{new}$, is different from the current epoch, $ELIST_m$, in which case the coordinator attempts to form a new epoch by initiating the second phase of the protocol.

Note that nodes respond to an epoch operation coordinator only if they are not participating in other epoch operations. Therefore, at most one epoch operation coordinator at a time can obtain enough responses to proceed to the second phase. In effect, our protocol uses a standard dynamic voting approach (see, e.g., [4]) to ensure that epoch changing operations are mutually exclusive.

Phase 2. In this phase, the coordinator distributes the *transitional values* of *total-votes* and *ELIST* to all members of the prospective epoch. It also includes $enumber_m$

into the message. The transitional value of *total-votes* includes votes from all nodes from the current epoch (both operational and failed), as well as the new nodes outside the current epoch that will be members of the prospective epoch. Transitional *ELIST* includes only those nodes from the current epoch that will also be members of the prospective epoch; it does not include either failed nodes from the current epoch or new nodes. Formally, let $CURRENT\text{-}NODES = ELIST_m \cap ELIST_{new}$ and $NEW\text{-}NODES = ELIST_{new} \backslash CURRENT\text{-}NODES$. ("\" denotes set subtraction.) Then, $ELIST_{trans} = CURRENT\text{-}NODES$, and

$total\text{-}votes_{trans} = total\text{-}votes_m$ + (the number of nodes in *NEW-NODES*).

The key property of these transitional values is that any set of nodes from the transitional epoch list with a number of nodes greater than half of transitional *total-votes* includes both a majority set of the current epoch list and a majority set of the prospective epoch list.

Upon receiving the transitional epoch values, each participant atomically updates its *ELIST* and *total-votes* to equal the transitional values, adopts $enumber_m$ received as its new epoch number, and responds to the coordinator with its version number and a list of operations currently locking its replica, called a *current operation list* below. When the number of nodes that responded exceeds half of $total\text{-}votes_m$ (i.e., when a write quorum of nodes from the current epoch responds), the coordinator initiates the third phase of the protocol.

Phase 3. In this phase, the coordinator merges the received lists of current operations into one and distributes the combined list, *COMBINED-OP-LIST*, to members of the prospective epoch. In other words, if $\{OP\text{-}LIST_i\}$ are current operation lists received, $COMBINED\text{-}OP\text{-}LIST = \cup_i OP\text{-}LIST_i$. On receiving *COMBINED-OP-LIST*, each participant filters out those operations found in its log of committed operations and returns the resulting list to the coordinator. So, for a replica i, $FILTERED\text{-}OP\text{-}LIST_i = COMBINED\text{-}OP\text{-}LIST \backslash LOG_i$, where LOG_i is the replica's log.

Phase 4. The fourth phase begins when the coordinator gets filtered lists from all members of the prospective epoch. If some operation was filtered by any participant, then that operation is in the lock releasing stage. Hence, its locks should not be propagated. Therefore, to find operations whose locks must be propagated, the coordinator computes

the intersection of all the filtered lists it received: $FINAL\text{-}OP\text{-}LIST = \cap_i$ $FILTERED\text{-}OP\text{-}LIST_i$. Then, the coordinator distributes *FINAL-OP-LIST* to all members of the prospective epoch $ELIST_{new}$. For every operation from the list received, a participant checks if its replica is not already locked for the operation and if there is still no record of this operation in its log, which would mean that the replica has completed the operation. If both conditions are met, the replica marks itself locked for the operation. Otherwise, the replica makes a note that, when its lock is no longer held by the operation, the lock release should be propagated to other members of the new epoch. The participants conclude this phase by sending an acknowledgement to the coordinator.

Phase 5. When all participants have acknowledged the execution of the fourth phase, the coordinator begins the final phase. It identifies the version number of the most current replicas in the new epoch $version_{max}$, which is the maximum version number reported in the second phase, and generates the new epoch number, $enumber_{new} = enumber_m + 1$. Finally, the coordinator distributes the new epoch parameters (i.e., $enumber_{new}$, $ELIST_{new}$, and $total\text{-}votes_{new} = |ELIST_{new}|$) as well as the version number of the most current replicas, $version_{max}$, to members of the new epoch. In response, each replica atomically updates its epoch parameters to equal the ones received and, if its version number is lower than $version_{max}$, marks itself stale and adopts the received version number.

No log records should be garbage-collected on a node from the time it sent the filtered list of current operations to the coordinator until the time it finishes its participation in the protocol.[3] Also, if the coordinator fails to receive a response from some of the participants during Phases 3 or 4, it distributes an abort message, which causes all participants to terminate the protocol and restore their original state. Thus, in effect, the epoch-changing algorithm embodies the two-phase commit protocol to ensure that epoch switching is done in an all-or-nothing manner. (However, no atomicity of switching to a new epoch with regard to read and write operations is implied.) As already mentioned in Chapter 4, two-phase commit makes the epoch-changing mechanism vulnerable to

[3]Otherwise, a situation like this could occur: In the first scenario from Section 5.1.1, w_2 was holding x_3's lock at the time x_3 constructed a list of current operations in the second phase of epoch switching. However, w_2 finished and was purged from x_3's log by the time x_3 enters the final phase of epoch switching. Then, x_3 would mark itself locked for an operation it already performed.

failures of the coordinator that occur during the transition to the new epoch, since such a failure can leave the system in a state where no further epoch operations are possible. To deal with this problem, one can incorporate the fault-tolerant *quorum-based commit* [64] protocol into the epoch-changing algorithm above. In fact, since epoch changing in this protocol is done in more than three phases anyway, quorum-based commit can ride on the existing rounds of message exchanges, so that no additional phases are needed.

The protocol above executes in five phases. However, Phases 3 and 4 deal exclusively with lock propagation and can be deleted if different design choices are made. For instance, the presented design is driven by the desire to avoid sending replica logs over the net. But, as discussed before, records usually stay in these logs for a very short time, and the logs are expected to be short or empty. Therefore, in the second phase, participants can include their logs in their message to the coordinator. The coordinator can then filter the current operation lists itself, making Phase 3 of the protocol unnecessary. In any case, since epoch transition in our scheme executes in the background completely asynchronously with user operations, the number of phases it requires is no longer as crucial as before.

Note that the advantage of our protocol is fully realized only if the number of new nodes being included in the new epoch that are not members of the current epoch number is smaller than half the number of nodes in the current epoch (i.e., $|NEW\text{-}NODES| < total\text{-}votes_m/2$). Otherwise, the number of nodes in $ELIST_{trans}$ would be less than half of $total\text{-}votes_{trans}$, and user operations that rely on transitional *ELIST* and *total-votes* to determine quorums would never be able to collect a quorum of permissions. These user operations would have to wait until they can use the new epoch parameters, and the protocol would therefore lose its ability to execute user operations during epoch transition. Therefore, when the number of fixed nodes is large, it makes sense to perform system reconfiguration in several consecutive stages, forming a new epoch at every stage, so that the number of new nodes included in every next epoch does not exceed half the number of nodes in the previous epoch.

5.1.3 User Operations

Read and write coordinators determine whether they collected a quorum of permissions using the most recent values of epoch parameters obtained along with permissions from replicas. The required number of permissions is determined by the most recent *total-votes* (more than half the total votes is needed); replicas whose permissions may count towards quorum formation are determined by the most recent epoch list (only members of this epoch list are allowed to participate).

Below, we will call epoch parameters distributed by an epoch changing operation during its final phase *true* epoch parameters, as opposed to *transitional* epoch parameters distributed during the second phase. When no epoch changing is taking place, all nodes keep true epoch parameters.

If a node keeps true epoch parameters, then its value of *total-votes* is equal to the number of nodes in its $ELIST$. This is because $\textit{total-votes}_{new}$, distributed during the final phase of an epoch-changing operation, is constructed to be equal to the number of elements in $ELIST_{new}$, and both values are recorded atomically on every node. In contrast, if $\textit{total-votes}_{trans}$ and $ELIST_{trans}$ are transitional parameters kept by some node, $\textit{total-votes}_{trans} > |ELIST_{trans}|$. Indeed, in the algorithm for epoch changing, $\textit{total-votes}_{trans} \geq |ELIST_{trans}|$ by construction, with equality reached when $ELIST_{new} = ELIST_m$. In this case, however, no epoch change is needed, and epoch changing would not be attempted.

The algorithm for user operations is simple. To perform a read or write, the operation coordinator requests permission from all replicas.[4] Upon receiving this request, a node obtains the local read or write lock for its replica and responds with its epoch parameters, stale flag, and version number. After getting all responses (recall that `RPC.CallFailed` is returned from failed or disconnected nodes), the coordinator identifies the maximum epoch number reported, $enumber_m$. Due to asynchronous epoch changing, $ELIST$ and *total-votes* from responses with the same epoch number may not be identical: some may contain true values, while others may contain transitional values. Responses with transitional values can be identified by the fact that their number of total votes exceeds

[4] However, see the end of this subsection for an easy way of avoiding having to contact all replicas on every operation.

the number of replicas in their epoch list. From the epoch-changing algorithm, it is clear that the transitional values of epoch list and *total-votes* are more recent than true values from responses with the same epoch number.

Let $ELIST_m$ and $total\text{-}votes_m$ be either transitional values from some response with epoch number $enumber_m$ (if any response contains transitional values) or the epoch list and *total-votes* from any response with epoch number $enumber_m$ (if all responses contain the same epoch parameters). The coordinator is considered to have collected a quorum if it obtained more than half of $total\text{-}votes_m$ non-`RPC.CallFailed` responses from members of $ELIST_m$.

Once the coordinator has collected a quorum of permissions, it identifies the maximum version number reported, $version_m$. Then, if the operation is a read, the data item is read from any non-stale replica that reported the maximum version number. If the operation is a write, the new version number is generated to equal $version_m + 1$, and the new data along with the new version number is written on all replicas from the quorum collected.

We do not specify when replicas are unlocked, since this is done according to strict two-phase locking at the commit or abort time of user transactions. Also, the second phase of user operations can ride on messages exchanged during a transaction's commit protocol, as explained in Section 4.1.1.

A real implementation would make use of the fact that epoch changes are infrequent to reduce the number of nodes with which a read/write coordinator must communicate in the common case. Indeed, potential read/write coordinators can cache the current epoch parameters obtained during the last successful operation as a hint and request permission from only a majority of replicas from the cached value of the epoch list. Normally, when no epoch change occurred since the previous operation, the cached epoch parameters will still be current, and permission from the above replicas will be sufficient to collect a quorum. Otherwise, either the coordinator will fail to collect all requested permissions, or some permissions will carry an epoch number greater than the cached value. In both cases, the operation can be re-tried on all replicas of the data item.

5.1.4 Recovery Operation

A replica with the *stale* flag set to *"YES"* may have older data than indicated by its version number. Therefore, such replicas are required to run a *recovery* operation to eliminate this discrepancy before servicing any reads.

To recover, a replica x_i (the target replica) finds any non-stale replica x_j of the same data item (the source replica) whose version number is equal or greater than the version number of x_i, and sends a recovery request together with its version number to x_j. When x_j gets the recovery request, it obtains the local read lock for the replica; checks if it is indeed not stale and has a version number that is equal to or greater than the one requested; if so, sends its data and version number back to x_i; and releases the read lock. Upon receiving the data, x_i atomically checks if its version number still does not exceed the one received from x_j, in which case x_i adopts the received data and version number.

We require the source replica to obtain its read lock before responding to the recovery request. In this way, the recovery operation does not propagate a transient value written by a transaction in progress that aborts later. (Strict two-phase locking employed in our protocol ensures that transactions release their locks only after they commit or abort.) Note that locking one replica does not prevent concurrent execution of user operations. Also, the source replica is locked only during a local read operation, not while messages are exchanged.

There are two reasons the target replica compares version numbers before adopting the data received. First, an asynchronous write operation w may have performed on the target replica x_i since the recovery request was sent, making x_i non-stale again. Copying data from the source replica x_j in this case could violate protocol correctness. Indeed, if the data sent from x_j to x_i is older than the data recorded on x_i by the write w, then the effect of adopting x_j's data by x_i would be as if the write w finished without performing on x_i. If x_i happens to be part of w's quorum, this would mean that w did not perform on all nodes from its quorum, which is an essential condition for the correctness of the protocol. The second reason for comparing version numbers is that the version number kept by x_i might have increased due to an epoch-changing operation performed after the recovery was initiated, in which case x_i should not accept data with earlier versions.

5.1.5 Proof of Correctness

Consider true values of *ELIST* and *total-votes* together as a special data item and *enumber* as its replica version number. Once a node decides to participate in an epoch operation, it ignores any messages from any other epoch operations. This is equivalent to every node maintaining an exclusive lock that epoch operations must obtain. Then, the first and last phases of the algorithm for epoch changing are the same as the algorithm for the write operation in the standard dynamic voting protocol of [32, 29] (see also Section 2.1.2 for a brief description).[5] Additional work done between the first and last phases of the algorithm does not affect either the true values of *ELIST* and *total-votes* or *enumber*. Thus, the whole operation manages replicas of this special data item in exactly the same way as the dynamic voting protocol would have. In particular, the following three lemmas are true (see [32, 29, 4]).

Lemma 5 *When no epoch changing is taking place in the system: (1) all nodes that have the same epoch number e also have the same epoch list* $ELIST_e$ *and total votes number* $total\text{-}votes_e$*, and (2) if* e_m *is the maximum epoch number among all replicas in the system, then the only quorums that can be collected consist of replicas with epoch number* e_m.[6] □

Lemma 6 *At most one epoch operation at a time can obtain enough responses in its first phase to proceed to the second phase. (Epoch-changing operations are mutually exclusive.)* □

Lemma 7 *The ith successful epoch-changing operation that runs since system initialization distributes epoch number i with the true epoch parameters during its final phase.*

By Lemma 5, and because epoch-changing operations are mutually exclusive, all true epoch parameters kept by nodes with the same epoch number, as well as all transitional

[5] The only difference is that our protocol stores the full list of replicas that participated in the last update, *ELIST*, instead of the number of them, *SC*, on all replicas on which the write performs. However, our protocol only uses the number of elements in *ELIST* to determine whether the operation collected a write quorum. This is the same as recording *SC* directly.

[6] In a standard dynamic voting protocol, the statements of this Lemma are correct *at all times*, not only when no write operations are taking place. We have to weaken the lemma by this additional condition because, in our protocol, epoch changing involves intermediate phases during which transitional values of epoch parameters are recorded.

epoch parameters kept by nodes with the same epoch number, are identical. We will call true (transitional) epoch parameters kept by nodes with epoch number e true (transitional) parameters of epoch e. Also, replicas that are members of the true epoch list of epoch e will be called members of epoch e. Quorums defined by true (transitional) parameters of epoch e (i.e., sets of nodes Q such that $Q \subseteq ELIST_e$ and $|Q| > total\text{-}votes_e/2$) will be called true (transitional) quorums of epoch e. Finally, if epoch operation p constructed true parameters of epoch e, we will say that p formed epoch e.

Corollary 1 *The ith successful epoch-changing operation that runs since system initialization distributes epoch number $i-1$ with the transitional epoch parameters during its second phase.*

Proof. By Lemma 7, the ith epoch-changing operation distributes epoch number i during its final phase, which exceeds by one the epoch number distributed during its second phase. □

Corollary 2 (The inclusion property) *Any transitional quorum from epoch i includes a true quorum from epoch i and a true quorum from epoch $i+1$. Consequently, any transitional quorum from epoch i has a non-empty intersection with any (true or transitional) quorum from epoch i as well as with any (true or transitional) quorum from epoch $i+1$.*

Proof. Follows directly from the way the $(i+1)$th epoch-changing operation constructs transitional parameters of epoch i and true parameters of epoch $(i+1)$. □

Corollary 3 *If epoch-changing operation p constructed true parameters of epoch e, and if q is the epoch-changing operation that directly follows p, then q in its second phase must collect responses from a true quorum of replicas from epoch e.*

Proof. By Lemma 7, when q starts, e is the maximum epoch number in the system. Then, since q must collect a true quorum of responses from some epoch, and a true quorum of any epoch contains a majority of members of that epoch, the corollary follows from claim 2 of Lemma 5. □

Corollary 4 *Epoch numbers increase monotonically on every node. In particular, if at some point a majority of members of some epoch e have epoch numbers greater than e*

(and thus no user operation can use a true or transitional quorum from e), the same will be true at any later time. □

Now we will establish that our protocol achieves one-copy serializability by proving that it satisfies the conditions of Lemma 1.

Lemma 8 *Neither two writes nor any read and any write can lock their quorums simultaneously: one must unlock at least one replica from its quorum before the other can lock all necessary replicas.*

Consider any two user operations, at least one of which is a write, w_1 and w_2. By the conflict rule of read and write locks, no replica can be locked simultaneously by w_1 and w_2. Let e_1 and e_2 be the epoch numbers whose epoch parameters were used to define quorums by w_1 and w_2 respectively, and let p_{e_1} and p_{e_2} be the epoch-changing operations that constructed those epoch parameters.

If $e_1 = e_2$, then the quorums used by both operations were true or transitional quorums from the same epoch. By Corollary 2, any two such quorums intersect. Thus, the two quorums could not be locked simultaneously: the operation that locked its quorum first would have to unlock the common replicas before the other operation can complete its quorum.

Now consider the case where $e_1 \neq e_2$. Without loss of generality, we will assume $e_1 < e_2$ and consider the following possibilities:
(1) w_1 used a transitional quorum, and $e_2 = e_1 + 1$. By Corollary 2, the quorums used by both operations intersect and cannot be locked concurrently.
(2) w_1 used a transitional quorum and $e_2 > e_1 + 1$, or w_1 used a true quorum. We will show that w_2 can lock its quorum only after w_1 unlocks some replica.

By Lemma 7 and Corollary 1, there is an epoch-changing operation p that follows directly p_{e_1} and, in its final phase, distributes an epoch number that is less than or equal to e_2. By Corollary 3, during its second phase, p must collect a true quorum from the epoch formed by p_{e_1}. This epoch is either epoch e_1 (if w_1 used a true quorum of e_1), or, by Corollary 1, $e_1 + 1$ (if w_1 used a transitional quorum of e_1). In both cases, the quorum locked by w_1 includes a true quorum of the epoch formed by p_{e_1}. Hence, since p in its second phase collects a true quorum from this epoch, there is a common node

x included in the quorum locked by w_1 and the quorum collected by p in the second phase. Furthermore, w_1 locked x before p recorded its transitional or true parameters on x (otherwise, since p_{e_1} precedes p, parameters written by p carry a greater epoch number, and w_1 would have used them rather than those written by p_{e_1}).

Now, assume for the contradiction that w_1 has not released any locks by the time w_2 locks its quorum. Then, p will propagate w_1's lock from x to all members of the epoch it forms before recording its true parameters on any node. (Indeed, since by assumption w_1 has not released any locks, no node could have a record referring to w_1 in its log; therefore, unless a member replica is already locked by w_1, it will mark itself locked for w_1 during the fourth phase of p's execution.) Consider the epoch-changing operation q that directly follows p. During its second phase, q must collect a quorum of responses from epoch e_p (Corollary 3). If w_2 locked its quorum after q recorded its true parameters on some node, then by assumption w_1 had not released any locks by the time q recorded its true parameters on some node. Hence, responses from members of e_p to q's coordinator will carry the information that they are locked by w_1, and w_1's lock will be propagated to all members of the epoch that q forms.

Now, let s be the epoch changing operation that recorded true parameters of epoch e_2.[7] w_2 locked its quorum only after s recorded these parameters on some node (it could not have used a quorum from e_2 otherwise). Also, s is either the same as p or runs later than p. Then, by repeating the above reasoning about consecutive epoch-changing operations the necessary (may be zero) number of times, we conclude that s propagates w_1's lock to all members of the epoch it forms before recording its true parameters on any of these replicas.

Then, if w_2 uses a true quorum from e_2, all members of this quorum are locked by w_1 until the latter releases its lock on at least one replica. Thus this quorum cannot be locked by w_2. If w_2 uses a transitional quorum from e_2, then this quorum includes some true quorum from e_2. Again, the latter is locked by w_1 and cannot be locked by w_2 until w_1 releases some replicas. In both cases, we arrived at the contradiction with the assumption that w_2 locks its quorum before w_1 releases any locks. □

[7] Note that s may or may not be the same operation as p_{e_2}, depending on whether w_2 used true or transitional parameters of e_2.

Lemma 9 *If all successful writes are numbered according to the order in which they lock their quorums, then the i-th write w_i assigns version number i to the replicas on which it performs.*

Proof. By induction. The basis is trivial: initially, all replicas keep version number 0; so the first write to lock a quorum after initialization will obtain $version_m = 0$ in the first phase and distribute version number 1 in the second. Assume the lemma is correct for all writes numbered up to i $(w_1, \ldots, w_i)$ and consider the $(i+1)$th write w_{i+1}. Assume w_i used a quorum (true or transitional) from epoch number e_i, and w_{i+1} used a quorum (true or transitional) from epoch number e_{i+1}.

Case 1. $e_i = e_{i+1}$. Since any two true or transitional quorums from the same epoch intersect, the quorums used by both writes have a common node x. x could be locked by w_{i+1} only after w_i unlocked it, which could happen only after w_i recorded version i on x. Then, with its permission to w_{i+1}'s coordinator, x reported version number i, which was the maximum version number by inductive assumption. Therefore, w_{i+1} will assign version $i+1$ to replicas on which it performs.

Case 2. $e_i > e_{i+1}$. This case can happen only if w_i used a true quorum from e_i and w_{i+1} used a transitional quorum from $e_{i+1} = e_i - 1$. Indeed, if w_i used a true quorum from e_i, then, by the time w_i locked its quorum, the epoch operation that recorded true parameters of epoch $e_i - 1$ had finished (by Lemma 7), and the next epoch operation (the one that constructed true parameters of e_i) had recorded transitional parameters of $e_i - 1$ on a majority of nodes from epoch $e_i - 1$. Therefore, by Corollary 4, at any later time, a user operation cannot use a true quorum from $e_i - 1$ or any quorum from any earlier epoch. The only quorums from an epoch with the number smaller than e_i that can be used are transitional quorums from $e_i - 1$. If w_i used a transitional quorum from e_i, w_i could lock its quorum only after epoch operation q that recorded transitional parameters of e_i started, which happened after the previous epoch operation s that recorded true parameters of e_i finished. Thus, w_{i+1} locked its quorum after s finished, and, by claim 2 of Lemma 5 and Corollary 4, could not use quorums from epochs smaller than e_i.

Since any true quorum from e_i and any transitional quorum from $e_i - 1$ intersect, w_{i+1} will assign version $i+1$ to participating replicas as in Case 1.

Case 3. $e_i < e_{i+1}$. Here, as in the proof of Lemma 8, we will consider the following possibilities. (3a) w_i used a transitional quorum from e_i and $e_{i+1} = e_i + 1$. Then, by Corollary 2, quorums used by both writes intersect, and w_{i+1} will assign version $i+1$. (3b) w_i used a transitional quorum and $e_{i+1} > e_i + 1$, or w_i used a true quorum. Since epoch-changing operations propagate the maximum version number reported during their second phase to all members of the epoch being formed, the proof is completely similar to the proof of case (2) of Lemma 8. □

Lemma 10 *A read operation always returns the data written by the immediately preceding write operation (i.e., the write that locked its quorum directly before this read did).*

Assume that the immediately preceding write w recorded version number i on participating replicas. For the same reason as in the proof of Lemma 9, the read will correctly identify the maximum version i in the system. Then, it will read the data item from a replica that has version i and is marked non-stale. But any non-stale replica that has version number i either received its data from w or obtained via recovery the data that originated from some replica on which w performed. In both cases, the read returns the data that w wrote. □

Theorem 3 *If the S2PL concurrency control protocol is used in conjunction with our protocol, the overall execution of transactions is one-copy serializable.*

Proof. The lemmas proved above show that our protocol satisfies the conditions of Lemma 1. The claim of the theorem follows. □

It is interesting to note that not all correct concurrency control protocols executed on the level of physical replicas can be used in conjunction with our protocol. The following example shows a history that is allowed by our replica control protocol, serializable on the level of physical operations (so a concurrency control scheme allowing such a history would be correct), but not one-copy serializable.

Consider a system with data items x and y, where x is replicated on five nodes, $x_1, \ldots, x_5$ (with our protocol employed for replica control), and y is not replicated. Consider transactions T_1 and T_2, both of which write y and then write x: $w[y] \rightarrow w[x]$.

Assume that all replicas of x are initially members of epoch 0, whose true parameters ($ELIST = \{x_1, \ldots, x_5\}$ and *total-votes*$= 5$) are recorded on every replica. Consider a history with the following sequence of events: (1) T_1 writes y. (2) T_2 writes y. (Since we are not using two-phase locking, T_2 can do so without waiting for T_1 to terminate.) (3) T_2 writes x by performing physical write operations on replicas x_1, x_2, and x_3 (it is a majority in epoch 0). (4) The network partitions briefly, and replicas x_3, x_4, and x_5 form new epoch 1, after which the partitioning heals. (5) T_1 writes x by performing physical write operations on replicas x_4 and x_5, which are a majority in epoch 1. (6) Both transactions commit.

This history is serializable on the level of physical operations. Indeed, its serialization graph is $T_1 \rightarrow T_2$, so it is equivalent to the history where T_1 performs first and is followed by T_2. But it is not one-copy serializable: T_1 could not appear before T_2 in the equivalent serial history, because the final logical value of x is written by T_1. Also, T_2 could not appear before T_1, because the last value of y is written by T_2. □

5.2 Extensions

In this section, we describe how our protocol can be integrated with some other ideas in fault-tolerant distributed computing.

5.2.1 Arbitrary Vote and Quorum Assignments

Thus far, we assumed that every replica has a single vote, and both read and write quorums are majority sets. However, a simple modification of our protocol allows for arbitrary vote and quorum threshold assignments.

We will add two variables to the control state kept by each node: *votes*, the number of votes assigned to the node; and an *in-transition* flag, which indicates that epoch parameters stored by the node are transitional parameters. Also, instead of *total-votes*, epoch parameters will now include read and write quorum thresholds, (r, w). Thus, when a node responds to a request from a user or epoch operation, the state it reports includes its *votes* and quorum thresholds.

The algorithm for user operations is modified as follows. As before, the coordinator decides whether it has been able to collect a quorum based on the most recent epoch parameters obtained. However, since it is now not possible to easily distinguish transitional from true epoch parameters, the coordinator uses the explicit *in-transition* flag for this purpose. So, if any of the responses with the maximum epoch number has the *in-transition* flag set, the coordinator uses epoch parameters from such a response. Otherwise, it uses epoch parameters from any response with the maximum epoch number. Once the coordinator of a read (write) has identified the most recent epoch parameters, it assumes it has collected a quorum if the total number of votes it received from members of $ELIST_m$ is equal to or exceeds r_m (or w_m), where $ELIST_m$, r_m, and w_m are the epoch parameters chosen. The rest of the algorithm remains unchanged.

Some changes must also be made in the epoch operation algorithm. Let $ELIST_m$, r_m, w_m and $enumber_m$ be epoch parameters with the maximum epoch number among those obtained by the epoch operation coordinator during the first phase of the operation. First, when deciding whether it was able to collect a write quorum of responses from the current epoch, the coordinator compares the total number of votes that members of *CURRENT-NODES* have with w_m, rather than with $total\text{-}votes_m/2$. Second, when distributing (true or transitional) epoch parameters to participating replicas, the coordinator includes (true or transitional) quorum thresholds instead of *total-votes*. Given the list of members of the new epoch $ELIST_{new}$, the coordinator chooses true quorum thresholds of the new epoch (r_{new}, w_{new}) arbitrarily,[8] subject to the quorum conditions $r_{new} + w_{new} > v_{new}$ and $2w_{new} > v_{new}$, where v_{new} is the total number of votes in $ELIST_{new}$. The transitional quorum thresholds are calculated to be $r_{trans} = \max(r_m, r_{new})$ and $w_{trans} = \max(w_m, w_{new})$. Finally, when participants in the epoch operation adopt transitional epoch parameters, they also atomically set their *in-transition* flag to "*YES*". They reset this flag to "*NO*" upon adopting true parameters of the new epoch.

With these modifications, the inclusion property still holds, i.e., any transitional read (write) quorum contains both a true read (write) quorum from the current epoch and a

[8] Of course, the coordinator actually relies on application characteristics like the read/write ratio and the relative importance of read vs. write availability to choose quorum thresholds.

true read (write) quorum from the new epoch. Therefore, the proof of correctness given in Section 3.5.1 remains valid, with trivial rephrasing (replacing the words "majority" and *total-votes* with "write (read) quorum thresholds" in all occurrences).

5.2.2 Dynamic-linear Voting

In [30] and independently in [49], a method was proposed to relax quorum conditions by using the total order imposed on all replicas of the data item. In this approach, called *dynamic-linear voting*, the quorum conditions are $r + w \geq v$ and $2w \geq v$. A read (write) quorum is a set of replicas with a total number of votes either exceeding r (or w), or equal to r (or w) and such that it includes the greatest replica according to the total order. In terms of our protocol, when a user or epoch operation decides if it collected a quorum using, say, epoch parameters $ELIST$, r, and w, it concludes that a read (write) quorum has been collected if either the set of responses from members of $ELIST$ has more than r (or w) total votes, or this set has exactly r (or w) total votes and includes the response from the greatest replica among members of $ELIST$. This extension allows the protocol to form new epochs containing a single replica, and thus to keep the data item available for user operations even if only a single replica remains operational.

It is easy to see that both the intersection property of true quorums and the inclusion property of transitional quorums remain valid with this extension. Therefore, the proof of correctness given in Section 3.5.1 still holds.

5.3 Related Work

Previously, in [4], the problem of asynchronous system re-configuration was solved only for the specific method of quorum adjustment, in which a node independently modifies the number of votes it is assigned. As noted in [4], because each node's view of the system may be different, and because each node operates independently, the resulting quorums may not be optimal. In particular, the vote assignment of a failed node cannot be changed. The advantage of the method is that it is simple and fast. In our protocol, the coordinator distributes its view of the system to all other nodes; failed nodes are excluded from the system.

Another scheme, proposed in [25], involves pre-computing different quorum assignments in advance and storing these assignments at all replicas. A natural number (a *level*) is associated with each assignment. When nodes fail and a transaction cannot collect a quorum of level n, it can move to higher levels without *any* global reconfiguration procedure at all. However, when nodes repair, transactions are not allowed to move back to lower level quorums. Thus, to include the repaired nodes back in the voting process, new quorum assignments must be redistributed to replicas. This is done as a special operation, which is mutually exclusive with user operations. So, in this protocol, quorum re-adjustment after failures is asynchronous (in fact, no procedure is needed in this case), while re-adjustment after repairs is not. In our protocol, quorum re-adjustment after both failures *and* repairs is done asynchronously.

5.4 Summary

This chapter describes a new dynamic protocol for managing replicated data. Like the existing dynamic schemes, our protocol involves reconfiguring the system dynamically to reflect failures and repairs as they occur, so that data may be kept available for user operations even if only one replica of the data remains accessible. However, in existing schemes, it is required for the correctness of the protocol that system reconfiguration either be mutually exclusive with reads and writes (and thus can interfere with and delay user operations) or be done within the protocol for the write operation; the latter increases the cost of writes and makes fault tolerance of the system dependent on the rate of write operations. In contrast, our protocol allows system reconfiguration to be done asynchronously with and separately from read and write operations. Therefore, in our protocol, user operations can execute while system reconfiguration is in progress, with no interference. At the same time, the cost of write operations in our protocol remains low, and the fault tolerance of the system does not depend on the rate of writes.

We also showed that our protocol guarantees one-copy serializable executions of user transactions, provided strict two-phase locking is used as a concurrency control mechanism to ensure serializability at the level of physical replicas. However, it is easy to see that not all correct concurrency control protocols can be used. Indeed, one can pro-

vide examples of executions that are valid under our replica control protocol, serial at the level of physical replicas, and yet not one-copy serializable. Therefore, while strict two-phase locking is the most common concurrency control protocol, a more general characterization of concurrency control protocols that can be used in conjunction with the replica control protocol described here remains a question for future work.

Chapter 6

Fault-Tolerant Management of Read-One-Write-All Replicated Data

A read-one-write-all (ROWA) replication scheme can be used in distributed systems to improve performance and availability of accesses to mostly-read data. In this scheme, a read can be performed from any replica of a data item, while a write must execute on all replicas. This provides highly efficient and fault-tolerant read operations. A read is highly efficient because it does not require communication with multiple replicas, it is performed from a nearby server, and processing of a read request is carried by a single replica, which provides for very effective load sharing. A read is highly fault-tolerant because it can perform as long as there is a single operational data replica in the system.

However, write operations in the ROWA scheme are very expensive and vulnerable to a single replica failure. Low write availability is a major limitation of the ROWA scheme, one which has prevented this method from being extensively used in commercial wide-area networks. This chapter describes a way to improve the fault-tolerance of write operations without compromising the beneficial read properties that motivate the ROWA scheme.

ROWA quorums are a particular case of voting-based quorums, where every replica

has a single vote, the read quorum threshold is 1, and the write quorum threshold is equal to the total number of replicas of the data item. However, most proposals aimed at improving the availability of quorum-based protocols (including those described previously in Chapters 4 and 5) do not apply to the ROWA case. Indeed, these schemes involve dynamically changing the set of nodes from which quorums are drawn (the *distinguished partition* [29], or the current *epoch* [53, 58]) to reflect failures and repairs occurring in the system. To guarantee consistency, these schemes require that a new epoch contain a write quorum of replicas drawn from an existing epoch. Since a write quorum in ROWA includes *all* replicas, any failure would make an epoch change impossible. Therefore, these dynamic schemes would not improve data availability for ROWA quorums.

In this chapter, we propose a scheme for dynamic system reconfiguration that is applicable to ROWA quorums. Our scheme is, again, based on the notion of an epoch ([53, 54, 58], see also Chapters 4 and 5). However, the protocol of this chapter defines quorums used to ensure uniqueness of the current epoch independently of quorums used for replica control. This makes it possible to form new epochs even if ROWA quorums are used for replica control. Protocols in which write quorums are employed for both replica control and epoch management are not applicable to ROWA, as explained above. Also, the protocol described here manages epochs system-wide, unlike the protocols of Chapters 4 and 5, where epochs are maintained among replicas of a given data item.

In addition, forming a new epoch in our protocol is done asynchronously with reads and writes. Thus, epoch formation cannot interfere with and delay user transactions. The same feature is achieved in Chapter 5 for non-ROWA voting-based quorums. But, again, that approach is not applicable for the ROWA scheme. On the other hand, the protocol of this chapter cannot be used with non-ROWA quorums. So, we need a special solution in each case.

Finally, when forming a new epoch, our protocol is very selective about which replicas should run the initialization procedure. As a simple example, if all replicas of a data item failed, and if the replica that failed last is the first to become operational again, its information is guaranteed to be current without any additional work. Our protocol identifies these sorts of circumstances and does not require the replicas to re-initialize in such situations. Moreover, when replica initialization is performed, it is accomplished by

communicating with a single current replica, and replica locks are not held during this communication. This efficient initialization is one of the properties that distinguish our protocol from some of the existing alternatives (see Section 6.5).

6.1 The Protocol

Our protocol consists of the following asynchronous procedures: epoch operation, read, write, and replica recovery. Every node i in the system maintains the following state: *enumber*, the epoch number of node i's epoch; *ELIST*, a list of node names that are members of this epoch (the *epoch list*); and *stale(x)*, a stale replica flag, maintained by i for every replica x_i it stores. If set to "*YES*", this flag indicates that the replica is out of date and must recover before physical reads on it are allowed. Note that we do not need replica version numbers to identify the most recent replicas: any replica not marked stale is considered up-to-date. Finally, node i keeps the array *latest-epoch*. For every node p in the system, *latest-epoch(p)* contains the number of the latest epoch that included p. This information is accurate only if node i is a member of the latest epoch in the system.

Initially, all nodes are members of epoch 0, and so have epoch number 0 and an epoch list that includes all nodes in the system. All replicas are current, so all stale replica flags are initialized to "*NO*", and *latest-epoch(p)* = 0 for all nodes. As usual, we assume that local updates of the node's state are atomic.

6.1.1 Epoch-Changing Protocol

The algorithm for epoch checking and switching to a new epoch (if necessary) is shown in Figure 6.1. We call this procedure an *epoch operation*, or an *epoch-changing operation* when it actually forms a new epoch.

Any node can initiate an epoch operation. The initiating node is called the coordinator below. Epoch operation is done in three phases.

During the first phase, the coordinator sends a request to all nodes. Each node responds with its state, after which it ignores any messages from other epoch operations

```
EpochCheck
// This algorithm is run by the coordinator; actions of participants
// are given in the comments.
    multicast(all-nodes-in-the-system, ''epoch-check-request'');
    // A node that receives this message responds with a tuple of the form
    //(node-name, enumber, ELIST, latest-epoch), unless it is already
    // involved in an epoch operation.
    receive RESPONSES;
    let (node_m, version_m, elist_m, enumber_m) be a response with the
       maximum epoch number;
    let NEW-EPOCH be all-nodes-that-responded;
    if |NEW-EPOCH ∩ ELIST_m| > |ELIST_m|/2 and NEW-EPOCH ≠ ELIST_m {
       new-epoch-number := enumber_m + 1;
       new-latest-epoch := latest-epoch_m;
       for each node p in NEW-EPOCH do {
          new-latest-epoch(p) := new-epoch-number; }
       multicast(NEW-EPOCH,(''prepare-epoch'', NEW-EPOCH,
              new-epoch-number, new-latest-epoch, latest-epoch_m));
       // On receiving the above message, a participant records the obtained
       // information stably and responds with an acknowledgement.
       if acknowledgements from all members of NEW-EPOCH received {
          multicast(NEW-EPOCH, ''commit-epoch'');
          // On receiving this message, a node p updates its enumber, elist, and
          // latest-epoch to equal NEW-EPOCH, new-epoch-number, and
          // new-latest-epoch recorded from the ''prepare-epoch'' message. Also,
          // in the same atomic action, for each replica x_p, if there exists a
          // node s such that s has a replica of x and the original value of
          // enumber_p (before adopting new-epoch-number) is less than
          // latest-epoch_m(s), then stale(x) is set to "YES".
          // This completes p's transition to the new epoch.
       else
          multicast(NEW-EPOCH, ''abort-epoch''); }
          // A recipient rolls back to the state it had before the current
          // epoch operation began.
    else
       multicast(all-nodes-in-the-system, ''abort-epoch-operation''); }
       // This message tells the recipients that they can now participate in
       // other epoch operations.
end;
```

Figure 6.1: The algorithm for checking/changing an epoch in the ROWA scheme.

until the current epoch operation terminates. Upon receiving all responses, the coordinator decides whether epoch changing is needed and possible. Epoch changing is possible if the coordinator has been able to obtain a (possibly weighted) majority of responses from the current epoch, where the current epoch is represented by the epoch list from a response with the maximum epoch number. Epoch changing is needed if the set of nodes that responded is different from the current epoch. If both conditions are met, the coordinator attempts to form a new epoch that includes all nodes that responded.

To this end, the coordinator first obtains the new epoch number by incrementing the current epoch number by one. It then calculates the new value of the *latest-epoch* array by taking the current value and replacing the entries for the members of the new epoch with the new epoch number, and initiates the second phase of the operation.

In the second phase, the coordinator distributes a `prepare-epoch` message to all members of the new epoch. This message includes the parameters of the prospective epoch and the *latest-epoch* array of the current epoch. Recipients of this message record the information in the stable storage and respond with an acknowledgement. We will call the epoch parameters that participants record during the second phase their *prospective epoch parameters.*

In the third phase, if acknowledgements from all participants were received in response to the `prepare-epoch` message, the coordinator sends a `commit` message, instructing all participants to update their state according to the information recorded in the second phase. In addition, every node p checks whether it should mark any of its replicas stale. A replica x_p must be marked stale if node p is coming from an epoch, which is earlier than the latest epoch that contained any replicas of x. Indeed, in this case, writes to data item x might have succeeded in a later epoch where x_p was not present, thus making it obsolete. (In fact, only repaired nodes that are joining from epochs earlier than the current one need to check for staleness of their replicas: any non-stale replica on a node that is a member of the current epoch would pass the above test.)

If any participants failed to acknowledge the `prepare-epoch` message, the coordinator sends out an `abort` message, causing the participants to forget the recorded information from the `prepare-epoch` message. In both cases, the participants become ready

to participate in future epoch operations.

Similar to other epoch-management protocols described in this thesis, this algorithm embeds the two-phase commit protocol to ensure that epoch changing is done in an all-or-nothing fashion. Thus, epoch changing is vulnerable to failures of the coordinator that occur during the transition to the new epoch. Our earlier comment on the possibility of using three-phase quorum-based commit to address this problem applies here as well.

Note that the epoch operation never acquires any locks that may be needed by a user operation. Therefore, no interference with user operations is possible.

6.1.2 Write Operations

When a node initiates a transaction T_i, we assume that it associates with T_i the node's current epoch number and epoch list, $enumber(T_i)$ and $elist(T_i)$. These parameters never change during the lifetime of the transaction (even if the node's epoch changes) and are available to all reads and writes executed by the transaction. Similar to [16], we say that transaction T_i executes in epoch $enumber(T_i)$.

Figure 6.2 shows the algorithm for the write operation. To write a data item x, the coordinator (the node that initiated the write) requests the write lock for x from all participants (the nodes that have a replica of x and are members of the epoch associated with the owner transaction, T). With the request for the lock, the coordinator includes the epoch number of the owner transaction, $enumber(T)$.

On receiving a lock request, a participant first compares its epoch number with the epoch number received with the request. If its epoch number is lower, the participant is in the middle of switching to T's epoch: it has recorded parameters of T's epoch as its prospective epoch parameters but has not yet received the `commit-epoch` message. Indeed, the participant is a member of T's epoch and this epoch has committed on some nodes. This could have happened only after all members confirmed recording this epoch's parameters in the second phase of epoch changing. In addition, since the participant's epoch number is lower than $enumber(T)$, it could not yet have committed this epoch. Thus, the participant in this case commits T's epoch by adopting its prospective epoch parameters as its new epoch parameters. If the participant's epoch number exceeds the

```
Write(x /* data item */, new-value,
      elist(T)    /* epoch list of the owner transaction */,
      enumber(T)    /* epoch number of the owner transaction */);
// This algorithm is run by the coordinator; actions of participants are
// given within the comments.
   PARTICIPANTS := {node p such that p ∈ elist(T) and p has a replica of x};
   multicast(PARTICIPANTS, (''write-lock-request'', x, enumber(T)));
   // On receiving write-lock-request, a participant p compares its enumber
   // with enumber(T).  If enumber(p) < enumber(T) (in which case p has recorded
   // parameters of T's epoch as its prospective epoch but has not committed it
   // yet), p commits T's epoch by performing the same actions as upon
   // receiving the ''commit-epoch'' message.  If enumber(p) > enumber(T), p sends
   // back a negative acknowledgement.  Then, p locks its replica of x and, if
   // its epoch number still does not exceed enumber(T), sends a positive
   // acknowledgement to the coordinator.  If its epoch number has become greater
   // than enumber(T), p sends a negative acknowledgement to
   // the coordinator and releases the write lock for x.  */
   receive RESPONSES;
   if all nodes from PARTICIPANTS replied with positive response {
      multicast(PARTICIPANTS, (''do-update'', x, new-value, enumber(T)));
      // When a participant p receives this message, it atomically writes the
      // new-value into its replica of x and, if its epoch number still does not
      // exceed enumber(T), sets stale(x) = ''NO''.
   else
      multicast(PARTICIPANTS, (''unlock-replica'', x));
      // On receiving this message, a participant unlocks its replica of x.
      abort the owner transaction; }
end;
```

Figure 6.2: The algorithm for the write operation in the ROWA scheme.

one received, the participant responds with a negative acknowledgement, which causes the coordinator to abort the transaction.

The participant then acquires the write lock for its replica of x and compares its epoch number with $enumber(T)$ again. (The participant's epoch number might have changed due to asynchronous epoch formation. By repeating this comparison, the protocol ensures that the participant's epoch number is equal to the transaction's epoch number when x is locked.) If the participant's epoch number still does not exceed the epoch number received, the participant sends a positive acknowledgement back to the coordinator.[1] If its epoch number has become greater than $enumber(T)$, the participant sends a negative acknowledgement to the coordinator and releases the write lock for x.

If all participants responded with positive acknowledgements, the coordinator attempts to perform the write on all replicas. When a node receives a message requesting it to perform the write, it atomically writes the new value of the data item and, if the node's epoch number still does not exceed that of of the owner transaction, marks the replica non-stale.

If the epoch number of the node has become greater than the epoch number of the owner transaction (due to the asynchronous formation of a new epoch), the new data is still written, but the node's state remains unchanged.[2] Finally, the participant unlocks its replica.

Note that if two-phase locking is used for concurrency control, the same locks can be shared by both replica control and concurrency control protocols. Then, the policy for releasing locks would be a combination of policies used by each protocol, and a replica would not release the write lock immediately after performing the write. (The same goes for reads, considered in the next subsection.) However, since we do not make any assumptions about the nature of concurrency control mechanisms used in the system

[1] In fact, the participant's epoch number cannot be lower than $enumber(T)$ at this point, due to monotonicity of epoch numbers (see Corollary 5).

[2] The data should be written in this case due to the following reasoning. Since the replica was locked for the write operation while the node was in its previous epoch, no operations could read or update this replica after it switched to the new epoch. If this replica was found current when the node was included in the new epoch, the data should be written to prevent a transaction that executes in the new epoch from reading stale data. If the replica was marked stale when the node was included in the new epoch, it does not matter whether the data is written, since the replica cannot be read until it obtains a value considered current in the new epoch.

(e.g., these mechanisms may use separate locks or no locks at all), we describe the policy sufficient for the replica control protocol to be correct provided *any* correct concurrency control protocol is used.

In fact, if strict two-phase locking is used for concurrency control, with locks shared by the concurrency and replica control protocols (so that once a transaction locks a replica, it remains locked until transaction commit time), a one-phase write algorithm is possible. In this algorithm, the write coordinator would send the new value of x along with $enumber(T)$ to *PARTICIPANTS*. Each participant would first compare its epoch number with $enumber(T)$. If its epoch number is lower than $enumber(T)$, the participant would commit T's epoch from its prospective epoch parameters. If its epoch number is greater, the participant would send a negative acknowledgement to the coordinator, aborting the transaction. Then, the participant would acquire the write lock for x and, if its epoch number still does not exceed $enumber(T)$, perform the write. If its epoch number has become greater than $enumber(T)$, the participant would send a negative acknowledgement to the coordinator and release the write lock for x.

6.1.3 Read Operations

The algorithm for the read operation is simple. A node that initiates the read of data item x (the coordinator) chooses the closest node that has a replica of x (the participant) from the nodes represented in the owner transaction's epoch. The initiator then sends a request for the data to the chosen replica, together with the epoch number of the owner transaction.

If the participant's replica is marked stale, or its epoch number is different from the epoch number of the owner transaction, the participant rejects the request; this causes the initiator to repeat the operation with another participant (or abort if none is available). Otherwise, the participant obtains the local read lock for the replica, checks its epoch number again, and, if it is still equal to the one from the read request, sends the data to the coordinator and releases the read lock.

A few points in the read protocol are noteworthy. First, our protocol never needs more than a single current replica in the transaction's epoch to perform a read. This

contrasts favorably with the views protocol [16] and the dynamic accessibility protocol [17], where the transaction's view must include a majority (a "dynamic majority", in the latter protocol) of replicas of the data item for the read to be successful.

Second, even in the presence of failures, our protocol, unlike the missing writes protocol [13], does not need replica version numbers to determine stale replicas. It can always identify whether a replica is stale by using only the local state of the node on which the replica resides. Therefore, a read operation never requires more than one round of message exchange between the node issuing the read and the node with a replica.

Also, our protocol never contacts more than one current replica to perform a read. In contrast, the missing writes protocol uses a majority consensus scheme when failures are detected in the system, thus incurring a significant read performance penalty. Of course, if a replica contacted during the read turns out to be stale, our protocol must repeat the read with another replica. However, since epoch reconfiguration, and the subsequent replica recovery, occur only in the aftermath of failures or repairs in the system, no reconfiguration or recovery is going on in the system in the common case. Then, most of the time all replicas in the system are marked non-stale, unless a stale replica could not contact a current replica from the same epoch to recover. In the latter case, the read operation will most probably also fail, since it needs a current replica as well. Hence, a successful read operation executed by a transaction running in the current epoch in most cases is satisfied by contacting a single node.

Finally, when the owner transaction's epoch is not the most current one (the transaction runs "in the past" [13]), some nodes included in the owner transaction's epoch list may have moved to later epochs. The read initiator could then contact a node that would reject the read request, because it is now in a different epoch. Hence, in our protocol, even successful reads, if they "run in the past", may have to contact multiple nodes. We could have followed the approach taken by the views and dynamic accessibility protocols and aborted the transaction in this case. The way we have chosen gives the transaction an extra chance to complete successfully.

6.1.4 Recovery Operation

A node p runs a recovery operation for any of its replicas marked stale. The idea of the recovery algorithm is to ensure that both source and target replicas are members of the same epoch, and that the target replica remains stale by the time it receives the requested data. The last condition prevents incorrect executions like the following: (1) a source replica receives recovery request and sends its data; (2) a write runs, and the target replica executes it before receiving the data from the source replica; (3) the source replica's data (now obsolete) arrives and overrides the current information in the target replica. Also, to make sure a recovery operation does not propagate a transient value written by a transaction in progress that aborts later, a recovery operation is allowed to read a replica only if the replica is not written by an uncommitted transaction.

To recover a replica of data item x, node p (the target) finds the nearest node that has a replica of x (the source) from the nodes included in p's epoch list and sends a recovery request together with p's epoch number to the chosen node. The source rejects the recovery request by sending a negative acknowledgement if its replica is marked stale, or its epoch number is different from the epoch number received with the request (this may happen, e.g., if the participant moved to a later epoch without p's knowledge), or its replica has been written by an uncommitted transaction. Otherwise, the participant obtains the local read lock for the replica, checks again if its epoch number is still equal to that received from the target, sends the data together with its epoch number back to the target node, and releases the read lock.

Upon receiving the data, the target obtains the local write lock for its replica. The target then checks if its epoch number has not changed since the beginning of the recovery operation by comparing its current epoch number with that received from the participant; it also checks if its replica is still marked stale. (It may have become current due to an asynchronous write performed on the replica while it was waiting for an answer to its recovery request.) If these conditions are met, the target updates its replica, marks it non-stale, and releases the write lock.

Note that in the recovery operation, both the source and target replicas are locked only during local read and write operations, not while messages are being exchanged.

This is important because it means recovery cannot cause replicas to be locked for a long time due to network partitioning or overloading.

6.2 Examples

In this section, we discuss informally the correctness of the protocol and illustrate some of its more subtle points with examples. The formal proof of correctness follows in Section 6.3.

First, consider the case where logical read and write operations and epoch-changing operations do not execute at the same time. Since any new epoch must include a majority of nodes from an existing epoch, the uniqueness of epochs is guaranteed, i.e., all nodes that have the same epoch number also have the same epoch list. Then, due to the intersection property of ROWA quorums, all transactions executing in the same epoch are one-copy serializable. In addition, if transaction T_1 is successfully executed in an epoch with epoch number e_1, and transaction T_2 is successfully executed in an epoch with epoch number e_2, and $e_1 < e_2$, it can be shown similar to [16] that T_1 can always be serialized before T_2, regardless of the actual execution time of the physical operations contained in T_1 and T_2. The proof is based on the fact that none of the physical writes of a transaction T_i is allowed to perform on nodes with epoch number greater than $enumber(T_i)$, and physical reads of transaction T_i must execute only on nodes with epoch numbers equal to $enumber(T_i)$. Therefore, even though transactions are allowed to succeed while "running in the past" (i.e., in non-current epochs), one-copy serializability is enforced.

The most difficult case to consider is when a user operation and switching to a new epoch are both happening at the same time and involve the same nodes. In this case, the correctness of our protocol is ensured by maintaining the invariant that no matter how epoch changing, recovery, and physical read and write operations are interleaved, all replicas residing on nodes with the same epoch number are either identical or marked stale. We now give a few examples of how our protocol handles such situations.

Consider the situation where an epoch e_i includes two replicas of a data item x, x_1 and x_2, and a logical write $w[x]$ executing in epoch e_i (we will use this shortcut instead

of saying "a write executing on behalf of a transaction with associated epoch number e_i") has locked both replicas. Before it actually updates them, however, a change in the system occurs, and formation of a new epoch $e_j > e_i$ is attempted. Assume that the node that stores x_2 is to be included in the new epoch, while x_1 is not. Furthermore, another replica, x_3, joins epoch e_j coming from epoch e_k, and $e_i < e_k < e_j$. Then, during formation of epoch e_j, x_2 is marked stale, which initiates its recovery. However, x_2 cannot recover, since it is locked for a write. When write $w[x]$ tries to perform on x_2, it finds x_2's epoch number to have become greater than e_i and leaves x_2 marked stale. So, after $w[x]$ completes, x_1 remains in epoch e_i, is non-stale, and reflects the result of $w[x]$ (hence, x_1 continues to be available for transactions running "in the past" in epoch e_i); x_2 is in epoch e_j, marked stale, and has not recovered. Thus, the overall effect of these events is the same as if $w[x]$ had completed before the formation of epoch e_j began.

Next, consider a situation where an epoch e_1 includes a stale replica x_1 and two current replicas, x_2 and x_3, of a data item x. x_1 wants to recover from replica x_2. In the meantime, reconfiguration is initiated to form a new epoch with epoch number $e_2 = e_1 + 1$. Replica x_2 is to be included in e_2, while replicas x_1 and x_3 are not. A transaction T_i is initiated in epoch e_2 and executes $w_i[x]$ while x_2 still has not changed its state to indicate the transition from epoch e_1 to e_2.

If the recovery operation initiated by x_1 obtained x_2's read lock before x_2 switches to e_2, $w_i[x]$ could not have performed on x_2, and x_1 gets the value of x_2 that it had before reconfiguration started. Thus, all operations can be serialized as: (1) recovery executed in epoch e_1, (2) formation of e_2, and (3) write executed in e_2.

If the recovery request reached x_2 after it switched to e_2, there is no way to tell whether $w_i[x]$ already performed on x_2, since we do not keep version numbers. It would be wrong for x_1 to copy x_2's data *after* $w_i[x_2]$ performs, because replicas x_1 and x_3 would then diverge while being in the same epoch and marked non-stale. Therefore, we require that both source and target replicas be in the same epoch for a recovery operation to succeed. So, in our example, once x_2 switches to epoch e_2, the recovery initiated by x_1 will not succeed regardless of whether any writes performed on x_2 in epoch e_2. After x_1's attempt to recover from x_2 fails, it will probably try to recover from x_3. Barring additional reconfigurations, this recovery may well succeed while "running in the past"

in epoch e_1.

If $w_i[x]$ locks x_2 before the recovery, then, once it succeeds, the epoch number of x_2 becomes e_2, and x_2 will reject a recovery request from x_1. In both cases, the equivalent serial execution is: (1) formation of e_2, and (2) write executed in e_2. x_1 must recover from a replica other than x_2.

As the last example, consider a situation where epoch e_1 contains current replicas x_1 and x_2, and a write $w_i[x]$ is initiated by a transaction T_i in e_1. At about the same time, a new epoch $e_2 = e_1 + 1$ is being formed, x_2 is to be included in e_2, and a read $r_j[x]$ is initiated by a transaction T_j in e_2. Suppose that $r_j[x]$ is translated into a physical read on x_2. Suppose $w_i[x]$ locked x_1 and x_2, and then x_2 switched to e_2 by updating its epoch list and number. Since x_2 was current in e_1, and since e_2 is the next epoch in the system, x_2 will be found current when included in e_2. When a physical write executed on behalf of $w_i[x]$ reaches the node where x_2 resides, it will find x_2's epoch number equal $e_2 > enumber(T_i) = e_1$. However, it would be incorrect not to update x_2 since x_1 would be updated (it is still in epoch e_1) and, after x_2 is unlocked, $r_j[x]$ would read stale data. The way the protocol works, x_2 will be updated while its node's state will remain unchanged. Therefore, when $r_j[x]$ gets to read x_2, it will see the effect of $w_i[x]$. Thus, the operations are serialized as: (1) $w_i[x]$ executed in e_1, (2) formation of e_2, and (3) $r_j[x]$ executed in e_2.

6.3 Proof of Correctness

Similar to the protocol in Chapter 5, the epoch operation here is equivalent to the write operation in the standard dynamic voting protocol of [32, 29], where the write is performed on a special data item containing the epoch list and *latest-epoch* array, and the epoch number serves the role of this data's version number. Thus, the whole operation manages replicas of this special data item in exactly the same way as the dynamic voting protocol would have. In particular, the following facts are true (see [32, 29, 4]).

Lemma 11 (Uniqueness of epochs) *At all times, all nodes that have the same epoch number e also have the same epoch list* $elist_e$.

Lemma 12 (Mutual exclusiveness of epoch operations) *At most one epoch operation at a time can obtain enough responses in its first phase to proceed to the second phase.*□

Lemma 13 *The ith successful epoch-changing operation that runs since system initialization distributes epoch number i epoch parameters during its second phase.*

Corollary 5 (Monotonicity of epoch numbers) *Epoch numbers never decrease on any node.*□

Let H denote an arbitrary history produced by our protocol, in which all actions are successful. As before, $enumber(A_i)$ and $elist(A_i)$ denote the epoch number and epoch list of an action A_i. As for transactions, it is convenient to define the epoch number and epoch list of a recovery operation to be the epoch number and epoch list of the target node at the time the recovery is initiated. More precisely, it is the epoch number that the target sends to the source replica with the recovery request, and the corresponding epoch list.

Lemma 14 *If SG(H) has an edge from action A_i to action A_j then the epoch number of A_i is less than or equal to the epoch number of A_j.*

Proof. Since SG(H) has an edge $A_i \rightarrow A_j$, A_i and A_j physically conflict, e.g., on a replica x_s, with the conflicting operations $op_i[x_s]$ and $op_j[x_s]$. Let e_i and e_j be the epoch numbers of s at the time $op_i[x_s]$ and $op_j[x_s]$ lock x_s. By the definition of a serialization graph, $op_i[x_s]$ happens before $op_j[x_s]$. Then, by Corollary 5, $e_i \leq e_j$.

For a physical read or write executed by a recovery operation to succeed, the epoch number of the replica at the time it is locked must be equal to the epoch number of the recovery operation. The same is true for a physical read executed by a transaction. For a physical write run by a transaction, the condition for successful execution is that the epoch number of the replica by the time it is locked must be less than or equal to the epoch number of the transaction. Hence, in all cases, $e_i \leq enumber(A_i)$ and $e_j \leq enumber(A_j)$. If $e_i = enumber(A_i)$, then $enumber(A_i) = e_i \leq e_j \leq enumber(A_j)$, as was to be proved.

Consider the case where $e_i < enumber(A_i)$. As shown above, $op_i[x_s]$ can only be a write issued by a transaction. In the write protocol, every participating replica with epoch number less than the epoch number of the owner transaction adopts the owner transaction's epoch number before obtaining the write lock. Thus, when the write lock is released, x_s's epoch number e_i' cannot be smaller than $enumber(A_i)$. Also, by Corollary 5, $e_i' \leq e_j$. Thus, we have $enumber(A_i) \leq e_i' \leq e_j \leq enumber(A_j)$, as was to be proved. □

Lemma 15 *SG(H) is acyclic.*

Proof. The proof is by contradiction. Let C be a cycle. By Lemma 14, the epoch numbers of all actions in C must be the same. Let e be this epoch number. First, note that C must include at least one recovery. Indeed, no cycle involving only transactions is possible, since execution of transactions is regulated by a concurrency control protocol that prevents such cycles.

Now consider the case where all vertices in C are recoveries, so that C is of the form $R_{u_1} \rightarrow R_{u_2} \rightarrow \ldots R_{u_n} \rightarrow R_{u_1}$. Consider two neighboring recoveries, R_{u_i} and R_{u_j}. For the edge $R_{u_i} \rightarrow R_{u_j}$ to exist, one of these two conditions must be satisfied: (1) R_{u_i} and R_{u_j} write to the same replica x_i, and $w_{u_i}[x_i]$ happens before $w_{u_j}[x_i]$. (2) R_{u_i} reads the same replica x_i that R_{u_j} writes to, and $w_{u_i}[x_i]$ happens before $r_{u_j}[x_i]$. Condition (1) is impossible, since after $w_{u_i}[x_i]$ performs, x_i becomes non-stale and will never be stale again in epoch e, which is necessary for $w_{u_j}[x_i]$ to succeed. Therefore, for all neighboring vertices in C, only (2) is possible. Every recovery consists of a physical read of a replica followed by a physical write to another replica. Then, by applying condition (2) to all pairs of neighboring recoveries in C, we conclude that $r_{u_1}[x_0]$ happens before $w_{u_1}[x_1]$, which happens before $r_{u_2}[x_1]$, which happens before $w_{u_2}[x_2]$ ..., which happens before $r_{u_n}[x_n]$, which happens before $w_{u_n}[x_0]$, which happens before $r_{u_1}[x_0]$, for some replicas $x_0, \ldots, x_n$. Thus, we arrived at the conclusion that $r_{u_1}[x_0]$ happens before itself, which is impossible.

Finally, consider the case where C contains both recoveries and transactions. Then, C must include an arrow from some transaction to some recovery. Let T_i and R_u be such a transaction and recovery. Since there is an edge $T_i \rightarrow R_u$ in SG(H), T_i contains

a physical operation that conflicts with and precedes some physical operation in R_u. Because a recovery consists of a read of a replica, $r_u[x_p]$, followed by a write to another replica, $w_u[x_q]$, only three cases are possible:

Case 1. T_i contains $w_i[x_p]$, which precedes $r_u[x_p]$. Since a recovery requires both the source and target replicas to be in the same epoch, x_p and x_q are both included in epoch list $elist_e$. Then, the logical write $w_u[x]$, on behalf of which $w_i[x_p]$ is executed, also includes $w_i[x_q]$. Moreover, all replicas of x in epoch e (including x_q) must be locked by $w_i[x]$ before $w_i[x_p]$ can perform. Therefore, since $w_u[x_q]$ happens after $r_u[x_p]$, which happens after $w_i[x_p]$, which happens after $w_i[x]$ acquires a write lock for x_q, $w_u[x_q]$ can acquire the write lock for x_q and perform only after $w_i[x_q]$ performs and releases the lock. The write protocol on Figure 6.2 provides that, after a physical write executed by a transaction performs on a replica, the replica is either marked non-stale or its epoch number is greater than the epoch number of the transaction. Hence, upon completion of $w_i[x_q]$, x_q is either non-stale or its epoch number is greater than e. In both cases, $w_u[x_q]$ could not succeed, since a recovery performs the write only if the target replica is not stale and is in the same epoch as it was when the recovery was initiated. This contradicts the fact that R_u is included in SG(H), since SG(H) contains only successful actions.

Case 2. T_i contains $r_i[x_q]$, which precedes $w_u[x_q]$. For $r_i[x_q]$ to succeed, x_q must be non-stale at the time $r_i[x_q]$ performs. Once a replica is non-stale in some epoch, it can never become stale in this epoch again (the only time a non-stale replica can be marked stale is when it is included in a new epoch). Then, x_q is non-stale by the time $w_u[x_q]$ begins, and $w_u[x_q]$ will fail. Thus, we arrive at the same contradiction with the fact that R_u is in SG(H).

Case 3. T_i contains $w_i[x_q]$, which precedes $w_u[x_q]$. After $w_i[x_q]$ performs, x_q is either non-stale or its epoch number is greater than e. In both cases, $w_u[x_q]$ could not succeed, and R_u could not be in SG(H). □

We next intend to add certain edges to SG(H) and prove that the resulting graph is RDSG(H) and has no cycles. First, we define a relation that includes any two actions that write to the same data item, provided one of the actions is a transaction.

Definition 1 *Let A_i and A_j be two actions that write to replicas of the same data item x, and at least one of these actions is a transaction. If A_i and A_j both write to a common replica x_p, then $A_i \Rightarrow_x A_j$ iff $w_i[x_p] <_H w_j[x_p]$. If there is no common replica of x to which A_i and A_j write, $A_i \Rightarrow_x A_j$ iff the epoch number of A_i is smaller than the epoch number of A_j.*

The following lemma establishes some properties of relation $\Rightarrow_x$.

Lemma 16 *(1) Any two actions that write to the same data item x, with one of these actions a transaction, are related by $\Rightarrow_x$; (2) $A_i \Rightarrow_x A_j$ precludes $A_j \Rightarrow_x A_i$; (3) if $A_i \Rightarrow_x A_j$ then the epoch number of A_i does not exceed the epoch number of A_j.*

Proof. Let A_i and A_j be two actions that write to replicas of data item x, and one of these actions (e.g., A_i) is a transaction.

If A_i and A_j write to a common replica x_p, $w_i[x_p]$ and $w_j[x_p]$ are related by $<_H$ by the definition of histories. Hence, A_i and A_j are related by $\Rightarrow_x$, accordingly. Moreover, by Lemma 15, physical writes executed by these actions on *all* common replicas of x will perform in the same order. Thus, $A_i \Rightarrow_x A_j$ precludes $A_j \Rightarrow_x A_i$ in this case.

If A_i and A_j do not write to any common replicas, then they execute in different epochs (because A_i writes to all replicas of x from its epoch). Therefore, these actions are related by $\Rightarrow_x$, and since $enumber(A_i) < enumber(A_j)$ precludes $enumber(A_j) < enumber(A_i)$, $A_i \Rightarrow_x A_j$ precludes $A_j \Rightarrow_x A_i$.

Finally, if A_i and A_j do not write to any common replicas, the third claim follows directly from the definition of $\Rightarrow_x$. If there is a replica x_p such that $w_i[x_p] <_H w_j[x_p]$, then $enumber(A_i) \leq enumber(A_j)$ by Lemma 14. □

Now we can construct RDSG(H).

Lemma 17 (Construction of RDSG(H)) *Let $G(H)$ be $SG(H)$ with edges $T_i \rightarrow T_j$ added for any two transactions T_i and T_j that run in different epochs and such that (1) $T_i \Rightarrow_x T_j$ for some data item x; or (2) there is an action A_k and a data item x such that T_i reads-x-from A_k and $A_k \Rightarrow_x T_j$. Then, $G(H)$ is an $RDSG(H)$.*

Proof. To show that G(H) is an RDSG(H), we need to prove that (1) for any two transactions T_i and T_j that write the same data item x, G(H) has a path either from T_i

to T_j or vice versa; and (2) for any three transactions T_i, T_j, and T_k and some data item x, if T_i *reads-x-from* T_k, T_j writes x and $T_k \ll T_j$, then $T_i \ll T_j$.

By Lemma 16, any two transactions T_i and T_j that write the same data item x are related by $\Rightarrow_x$. Hence, if they run in different epochs, there is a path between them by the definition of G(H). If they run in the same epoch, they execute physical writes on the same set of replicas of x, and hence physically conflict. Then, by the definition of history and serialization graph, there is an edge between T_i and T_j in SG(H) and therefore in G(H). So, the first condition required for G(H) to be an RDSG(H) is satisfied.

Now, let T_i, T_j, and T_k be any three transactions, such that for some data item x, T_i *reads-x-from* T_k, T_j writes x, and $T_k \ll T_j$. If T_i and T_j run in different epochs, there is an edge $T_i \rightarrow T_j$ in G(H) by the definition of G(H). If T_i and T_j both run in the same epoch e, then, since T_j writes to all replicas of x in e and T_i reads from one replica of x in e, there is a replica x_p from which T_i reads and to which T_j writes. Then, T_i and T_j physically conflict, and SG(H) contains an edge between them. Assume for contradiction that $T_i \not\rightarrow T_j$. Then, $T_j \rightarrow T_i$. By assumption that any transaction contains at most one logical read of a given data item, and because a logical read is translated into a single physical read, x_p is the only replica of x that T_i reads. Consider two cases.

If T_i *directly reads-x-from* T_k, then T_k writes into x_p. Then, by the definition of histories, either $w_j[x_p] <_H w_k[x_p]$ or $w_k[x_p] <_H w_j[x_p]$. Because the assumption that $T_j \rightarrow T_i$ is in SG(H) entails $w_j[x_p] <_H w_i[x_p]$, $w_k[x_p] <_H w_j[x_p]$ would contradict the assumption that T_i *directly reads-x-from* T_k. Thus, we are left with the condition that $w_j[x_p] <_H w_k[x_p]$, which, by the definition of $\Rightarrow_x$, entails $T_j \Rightarrow_x T_k$ and contradicts claim (2) of Lemma 16.

If T_i *indirectly reads-x-from* T_k, there is a sequence of recoveries $R_{u_1}, \ldots, R_{u_n}$ in H such that R_{u_1} *directly reads-x-from* T_k, R_{u_2} *directly reads-x-from* R_{u_1}, ..., and T_i *directly reads-x-from* R_{u_n}. Then, as in the preceding case, R_{u_n} writes to x_p and $w_j[x_p] <_H w_{u_n}[x_p] <_H r_i[x_p]$. Therefore, by Corollary 5, R_{u_n} runs in epoch e, the same as T_j and T_i. But this would mean that R_{u_n} could not be successful, since after $w_j[x_p]$ completes, x_p can never be stale in epoch e, which is necessary for $w_{u_n}[x_p]$ to succeed. Hence, we have arrived at a contradiction with the assumption that R_{u_n} is included in H.

In both cases, we have shown that the assumption that $T_i \rightarrow T_j$ is not in SG(H) leads to contradictions. Therefore, G(H) has an edge from T_i to T_j. So, the second condition required for G(H) to be an RDSG(H) is satisfied. □

Now, to show that our protocol produces one-copy serializable executions of committed transactions, we need only prove that G(H) is acyclic.

Lemma 18 *If recovery R_u reads-x-from action A_i, and $A_i \Rightarrow_x T_j$ for some transaction T_j, then $R_u \Rightarrow_x T_j$.*

R_u reads-x-from A_i iff there is a (possibly empty) sequence of recoveries such that *R_u directly reads-x-from* the last recovery in the sequence (or from A_i, if the sequence is empty), every recovery in the sequence except the first *directly reads-x-from* the previous recovery in the sequence, and the first recovery in the sequence *directly reads-x-from* A_i. We will prove the lemma by induction on the number of recoveries in the sequence.

Basis: *R_u directly reads-x-from A_i*. Let R_u consist of $r_u[x_p]$ followed by $r_u[x_q]$. Since *R_u directly reads-s-from A_i*, A_i includes a physical write into x_p and $w_i[x_p] <_H r_u[x_p]$. Also, since $A_i \Rightarrow_H T_j$, T_j includes a logical write of x.

Case 1. x_p is included in the epoch list of T_j. Then, T_j executes a physical write into x_p. Thus, both A_i and T_j write to x_p. By claim (2) of Lemma 16, $w_i[x_p] <_H w_j[x_p]$ (since $w_j[x_p] <_H w_i[x_p]$ would entail $T_j \Rightarrow_H A_i$). Therefore, because *R_u directly reads-x-from A_i*, $r_u[x_p] <_H w_j[x_p]$. Hence, by Lemma 14, $enumber(R_u) \leq enumber(T_j)$. If $enumber(R_u) < enumber(T_j)$, then $R_u \Rightarrow_x T_j$ follows directly from the definition of $\Rightarrow_x$. If $enumber(R_u) = enumber(T_j)$, then, by Lemma 11, R_u and T_j run in the same epoch. Since T_j executes a logical write of x translated into physical writes on all replicas of x in T_j's epoch, and R_u executes a physical write on a replica of x in the same epoch, both R_u and T_j execute a physical write on the same replica of x, say, x_q. $w_j[x_q]$ could not happen before $w_u[x_q]$, since in this case, after $w_j[x_q]$ completes, x_q would be marked non-stale, and $w_u[x_q]$ (and thus R_u) could not succeed. So, $w_j[x_q]$ happens after $w_u[x_q]$ and, by the definition of $\Rightarrow_x$, $R_u \Rightarrow_x T_j$.

Case 2. x_p is not included in epoch list of T_j $clist(T_j)$. We first prove that in this case, $enumber(R_u) \leq enumber(T_j)$. Assume to the contrary that $enumber(R_u) > enumber(T_j)$. Since x_p is included in epoch list of A_i and not in epoch list of T_j, by

Lemma 11, $enumber(A_i) \neq enumber(T_j)$. In addition, by Lemma 16, $enumber(A_i) \leq enumber(T_j)$. So, we have $enumber(A_i) < enumber(T_j) < enumber(R_u)$. Consider the epoch with the smallest epoch number e such that $enumber(T_j) < e$ and $x_p \in elist_e$. (e is the first epoch after epoch $enumber(T_j)$ that includes x_p. We know e exists, since $x_p \in elist(R_u)$). Since $A_k \Rightarrow_x T_j$, there is a replica x_q of x that T_j writes. Therefore, $x_q \in elist(T_j)$. Because $elist(T_j)$ includes some replicas of x but not x_p, when an epoch-changing operation includes x_p in epoch e, it marks x_p stale. Since $x_p \in elist(R_u)$, $e \leq enumber(R_u)$ and, by Corollary 5, the node where x_p resides adopted epoch number e and marked x_p stale before $r_u[x_p]$ performed. Hence, before R_u could read x_p, x_p had to recover, or some write executed on x_p by a transaction different than A_i (since $e > enumber(A_i)$) had to perform. Both cases contradict the assumption that R_u *directly reads-x-from* A_i. Thus, we conclude that $enumber(R_u) \leq enumber(T_j)$. Moreover, because x_p is included in the epoch list of R_u and not in the epoch list of T_j, their epoch numbers are different by Lemma 11. Therefore, $enumber(R_u) < enumber(T_j)$, and by the definition of $\Rightarrow_x$, $R_u \Rightarrow_x T_j$, as was to be proved.

Induction. Assume the lemma is correct for sequences of recoveries whose length does not exceed i and consider a sequence $\{R_{u_1}, \ldots, R_{u_i}, R_{u_{i+1}}\}$ that includes $i+1$ recoveries. So, R_u *directly reads-x-from* $R_{u_{i+1}}$, which *directly reads-x-from* R_{u_i}, ... , which *directly reads-x-from* R_1, which *directly reads-x-from* from A_i. By the inductive assumption, since $A_i \Rightarrow_x T_j$ and $R_{u_{i+1}}$ *reads-x-from* A_i, $R_{u_{i+1}} \Rightarrow_x T_j$. Then, by the basis of the induction, $R_u \Rightarrow_x T_j$. □

Lemma 19 *If G(H) has an edge* $A_i \rightarrow A_j$, *then* $enumber(A_i) \leq enumber(A_j)$.

Proof. If edge $A_i \rightarrow A_j$ is in G(H), then either this edge is in SG(H), or $A_i \Rightarrow_x A_j$ for some x, or A_i and A_j are transactions, and there is an action A_k and a data item x such that A_i *reads-x-from* A_k and $A_k \Rightarrow_x A_j$. In the first case, $enumber(A_i) \leq enumber(A_j)$ by Lemma 14. In the second case, the same it true by the definition of $\Rightarrow_x$. Consider the third case. Since A_i and A_j are transactions, we will denote them T_i and T_j below. If T_i *directly reads-x-from* A_k, it can be shown completely similarly to to the proof of the induction basis from Lemma 18 that $T_i \Rightarrow_x T_j$. Then, by Lemma 16, $enumber(T_i) \leq enumber(T_j)$. If T_i *indirectly reads-x-from* A_k, then there is a recovery R_u such that T_i

directly reads-x-from R_u, and R_u *reads-x-from* A_k. By Lemma 18, $R_u \Rightarrow_x T_j$. Then, again by Lemma 18, $T_i \Rightarrow_x T_j$ and, by Lemma 16, $enumber(T_i) \leq enumber(T_j)$. □

We are now ready to prove the main theorem.

Theorem 4 *H is one-copy serializable.*

Proof. We first prove that G(H) is acyclic. Indeed, suppose for contradiction that G(H) has a cycle C of the form $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow A_1$. Then, by Lemma 19, $enumber(A_1) \leq \ldots \leq enumber(A_n) \leq enumber(A_1)$. Therefore, all actions in cycle C have the same epoch number. By Lemma 15, C cannot have all its edges included in SG(H). Thus, at least one edge in this cycle is in G(H) but not in SG(H). By definition (see Lemma 17), any such edge connects actions that have different epoch numbers. Thus, we have a contradiction and conclude that G(H) is acyclic. But, by Lemma 17, G(H) is an RDSG(H). Hence, H has an acyclic RDSG and is therefore one-copy serializable. □.

6.4 A Variant of the Protocol for Non-Blind Writes

A write operation performed by a transaction is called *blind* if it is not preceded by a read operation of the same data item in the same transaction. Otherwise, a write is called *non-blind.*

An example of a system with non-blind writes is a system where replication is done at the physical page level and where a logical data item may occupy just a portion of a page. Indeed, in this case, pages are read and written as a whole. So, when an application writes a logical data item on the page, this page is first read, the corresponding portion of it is overwritten by the new value of the data, and then the whole page is written back to disk.

If our protocol (as described so far) is used in a system where non-blind writes are prevalent, an unfortunate situation may occur, in which a data item cannot be accessed without manual intervention from the system administrator. The following example illustrates this problem.

Consider a system with six nodes, three of which replicate data item x, (x_1, x_2, and x_3), and three others replicate data item y (y_1, y_2, y_3). Assume that transactions always

read data items before writing them (all writes are non-blind). Initially, all nodes form epoch 0. Consider the following chain of events. (1) The system partitions into regions $A_1 = \{x_1, x_2\}$ and $B_1 = \{x_3, y_1, y_2, y_3\}$. Nodes in partition B_1 form epoch 1. (2) The system re-partitions, and the new regions are $A_2 = \{x_1, x_3\}$ and $B_2 = \{x_2, y_1, y_2.y_3\}$. Partition B_2 forms epoch 2, in which x_2 is marked stale. Since epoch 2 contains no current replicas of x, x cannot be read (and, hence, written) in this epoch. For the same reason, x_2 cannot recover. Thus, x_2 will remain stale in epoch 2 forever. (3) Partitioning of the system heals. Epoch 3 is formed to include all nodes. x_1 and x_3 are marked stale, because they came from epochs that are earlier than epoch 2. In a system where blind writes are possible, this would be justified, since x_2 might have been written in epoch 2. In the absence of blind writes, however, x_3 is guaranteed to be current even though it is coming from an earlier epoch, so marking it stale is unnecessary. Worse, all replicas of x in epoch 3 are now stale, and only intervention from the system administrator to reset x's stale data flag would allow further access to x.

Clearly, in this situation, we need to recognize that x_3 is current and not mark it stale during the formation of epoch 3. This can be done (conceptually) if, for every data item x, all nodes maintain information on the number of the latest epoch that contained current replicas of this data item, *latest-current-epoch(x)*. This data structure would replace the *latest-epoch* array. Then, during new epoch formation, a replica x_p kept by node p is marked stale if p comes from an epoch with a smaller epoch number than *latest-current-epoch(x)*. As part of the control state, the *latest-current-epoch* array would be maintained on all nodes using the dynamic voting discipline embedded in the algorithm for epoch changing. Therefore, members of the most recent epoch will have the correct value of the *latest-current-epoch* array, and a successful epoch-changing operation will be guaranteed to have used this correct value.

The only problem with this approach is that maintaining a data structure with an entry for every data item on all nodes (even those without replicas of this data item) does not scale well. Obviously, this data structure can be compressed by not keeping entries for the data items x such that *latest-current-epoch(x)*$= e_m$, where $= e_m$ is the most recent epoch number in the system,[3] and by specifying information on the latest epoch

[3]The protocol would assume that *latest-current-epoch(x)*$= e_m$ for any data item not found in the list.

number for ranges of data items with the same value of *latest-current-epoch*. However, there exists a more radical and elegant solution, which we describe next.

To make a system administratively manageable, data are usually replicated across nodes in large chunks (sometimes called *volumes* [66, 27]). All data items belonging to the same volume are replicated on the same set of nodes. To avoid possible confusion, we will distinguish *volume replicas*, which are replicas of the whole volumes, from *item replicas*, which are replicas of individual data items that form volumes.

While the number of data items in the system may be high, the number of volumes is usually very small. (In fact, often server nodes are replicated as a whole. In this case, there is only a single volume on every node, and the total number of volumes is smaller than the number of nodes, because each volume is replicated on several nodes.) Then, instead of maintaining the *latest-current-epoch* array on a per-data-item basis, we can maintain it on a per-volume basis. For a volume X, *latest-current-epoch*(X) gives the number of the latest epoch that contained a replica of volume X with *all* current item replicas.

Thus, every node maintains an array *latest-current-epoch* for every volume, even for those it does not keep. Initially, *latest-current-epoch*(X) = 0 for all volumes. We will call a replica of volume X_p kept by node p *dormant in epoch enumber*(p) if *latest-current-epoch*$_p$(X) < *enumber*(p). A dormant volume has current replicas in epochs that are earlier than *enumber*(p). To guarantee that this is always the case, writes to a dormant item replica are prohibited regardless of the value of its stale flag.

Also, every node p maintains an array *stale-counter* with an entry for every volume *kept on* p. For volume X, *stale-counter*$_p$(X) gives the number of item replicas in this volume that are marked stale on p. In particular, if *stale-counter*$_p$(X) = 0, all item replicas in volume X are non-stale on p.

Figure 6.3 shows the modified algorithm for epoch operation. When nodes respond to an `epoch-check-request` message, they include the *latest-current-epoch* array (instead of *latest-epoch* array in the algorithm of Section 6.1.1) and *stale-counter*. Also, if a node has a dormant volume with all item replicas marked non-stale, it includes an *epoch-change-demand* flag with its response to indicate that a new epoch should be formed even if no changes in system topology are detected. The formation of a new epoch is

```
EpochCheck
// This algorithm is run by the coordinator; actions of participants are
// given within the comments.
   multicast(all-nodes-in-the-system, ''epoch-check-request'');
   // A node that receives this message responds with a tuple of the form
   // (node, enumber, ELIST, latest-current-epoch, stale-counter). Also,
   // if there is a dormant volume X such that stale-counter(X) = 0,
   // the node includes epoch-change-demand flag in its response.
   receive RESPONSES;
   let (node_m, version_m, elist_m, enumber_m) be a response with the
      maximum epoch number;
   let NEW-EPOCH be all-nodes-that-responded;
   if |NEW-EPOCH ∩ ELIST_m| > |ELIST_m|/2 and NEW-EPOCH ≠ ELIST_m or
      any response with the maximum epoch number includes epoch-change-demand {
      new-epoch-number := enumber_m + 1;
      new-latest-current-epoch := latest-current-epoch_m;
      for each volume X do {
         if there exists node p ∈NEW-EPOCH such that
               enumber_p ≥latest-current-epoch_m(X) and stale-counter_p(X) = 0 {
            new-latest-current-epoch(X) := new-epoch-number; } }
      multicast(NEW-EPOCH,(''prepare-epoch'', NEW-EPOCH, new-epoch-number,
         new-latest-current-epoch, latest-current-epoch_m));
      // On receiving the above message, a participant records the obtained
      // information stably and responds with an acknowledgement.
      if acknowledgements from all members of NEW-EPOCH received {
         multicast(NEW-EPOCH, ''commit-epoch'');
         // On receiving this message, a node p updates its enumber, elist, and
         // latest-current-epoch to be equal to the parameters of the prospective
         // epoch recorded from ''prepare-epoch'' message. Also, in the same
         // atomic action, for each volume X stored on p, if the original
         // value of enumber(p) (before adopting new-epoch-number) is less
         // than latest-current-epoch_m(X), p sets stale(x) to "YES" for all item
         // replicas in X and sets stale-counter(X) to the number of data
         // items in volume X. This completes p's transition to the new epoch.
      else
         multicast(NEW-EPOCH, ''abort-epoch'');
         // A recipient rolls back to the state it had before the current
         // epoch operation began. }
   else
      multicast(all-nodes-in-the-system, ''abort-epoch-operation'');
      // This message tells the recipients that they can now participate in
      // other epoch operations. }
end;
```

Figure 6.3: The algorithm for epoch checking/changing in the ROWA scheme with non-blind writes.

desirable in this case to make this volume non-dormant again, so that writes to its data will no longer be prohibited.

Let *latest-current-epoch*$_m$ be the *latest-current-epoch* array from a response with the maximum epoch number.

If the epoch operation coordinator decides to form a new epoch, it checks for every volume X whether the new epoch will include a node p on which all item replicas from volume X are current. An item replica x_p can be assumed current only if it is marked non-stale and $enumber_p \geq latest\text{-}current\text{-}epoch_m(X)$, where $enumber_p$ is the epoch number reported by p in the first phase of the operation. Indeed, for any node p with an epoch number less than $latest\text{-}current\text{-}epoch_m(X)$, all item replicas from X must be assumed stale even if marked non-stale, since these data items were represented in later epochs, of which node p was not a member. Hence, these data may have been modified there, making p's data obsolete. If the prospective epoch contains a replica of volume X with all item replicas current (according to the above criterion), X's entry in the new value of the *latest-current-epoch* array is set to the new epoch number.

When a participant p commits the new epoch, it marks data items stale on a per-volume basis: it marks stale *all* item replicas from any volume X for which $enumber(p) < latest\text{-}current\text{-}epoch_m(X)$, where $enumber(p)$ is the epoch number that p had before joining the new epoch and *latest-current-epoch*$_m$ is the *latest-current-epoch* array from the current epoch. Indeed, this condition means that p was not a member of an epoch where X was represented by a current volume replica. Hence, p's copy of X may not have seen some updates to data items from X.

The write and recovery algorithms are modified slightly. Whenever stale flag of a replica x_p is changed from stale to non-stale, the entry in $stale\text{-}counter(X)$ is decremented by one. In addition, a write to a data item belonging to a dormant volume is prohibited. In other words, when node p receives a write lock request for data item x and $latest\text{-}current\text{-}epoch(X) < enumber(p)$, where X is the volume to which x belongs, p responds with a negative acknowledgement, causing the transaction to abort. This is done to ensure that the epoch indicated in $latest\text{-}current\text{-}epoch(X)$ indeed contains the current version of all of X's data items.

The read algorithm remains the same.

Finally, as the result of recovery, if a dormant volume on some node p becomes completely non-stale (i.e., $stale\text{-}counter_p(X) = 0$), node p may initiate a new epoch formation (even if no failures/repairs have been detected) to make this volume non-dormant. As a result, writes to the data in this volume will no longer be prohibited.

The proof of correctness from Section 6.3 holds for this modified version of the protocol, with one exception. For case (2) of the induction basis proof in Lemma 18, different reasoning is needed to show that x_p is marked stale when included in epoch e. Namely, we can state now that, because T_j wrote x, X was not dormant in T_j's epoch. Hence, in this epoch, $latest\text{-}current\text{-}epoch(X) = enumber(T_j)$. Thus, when node p is included in epoch e, because its epoch number is less than $enumber(T_j)$, x_p is marked stale. The rest of the proof remains the same.

6.5 Related Work

Several protocols applicable to the ROWA strategy have been proposed to address the problem of availability of replicated data.

6.5.1 The Dynamic Accessibility Protocol

In the *dynamic accessibility protocol* [17], every node s in the system maintains its *view*, or a set of nodes with which s assumes it can communicate. Unlike our epochs, which require that a new epoch contain a majority of members from the current epoch, new views can be formed with no restrictions (except that every view must have a unique identifier, with all identifiers totally ordered). However, to avoid replica divergence, for every data item x, the protocol keeps track of the latest view where x was *accessible*. Initially, all nodes belong to the view that includes all nodes in the system, and all data items are marked accessible in that view. When a new view is formed, a data item x is marked accessible there only if the new view includes a majority of the replicas of x from the latest existing view where x was accessible. If a data item is accessible in some view, a read of this data item can be serviced by any replica; a write must perform on all replicas of the data item from that view.

For example, consider a system with ten nodes, five of which replicate data item x and five others keep replicas of data item y. Denote the nodes $x_1, \ldots, x_5, y_1, \ldots, y_5$. Initially, all nodes are in view 0, and both data items are accessible there. If the network partitions into two regions, $A = \{x_1, x_2, x_3, y_1\}$ and $B = \{x_4, x_5, y_2, \ldots, y_5\}$, both partitions can form new views. Assume, for example, that partition A forms view 1, and partition B forms view 2. In view 1, x is marked accessible (because it contains a majority of replicas of x from the previous view where x was accessible, i.e., view 0), and y is marked inaccessible. Similarly, in view 2, y is marked accessible, and x is not.

A transaction initiated from nodes that are members of view 1 can read x by accessing any of $\{x_1, x_2, x_3\}$ and write x by writing all these replicas, but it can neither read nor write y. A transaction in view 2 can read and write y (by reading from any of $\{y_2, \ldots, y_5\}$ and writing all of them), but it can neither read nor write x.

The dynamic accessibility protocol, like our protocol proposed here, can tolerate more than a majority of failed replicas. (In our example, if x_3 fails in view 1, the remaining nodes would form view 3, where transactions could still read and write x). However, there are important differences.

First, the policy for choosing partitions where data will be available for accesses in these two protocols is different. The dynamic accessibility protocol makes a data item x available for both reads and writes in the partition that contains a "dynamic majority" of replicas of x and disallows reads and writes in other partitions. This decision is made individually for every data item. It can therefore cause a situation where accessibility rights to different data items are scattered across different partitions. Then, a transaction that accesses multiple data items could fail no matter in which partition it runs, because it would require all its data to be simultaneously available. Continuing our example, if transaction T accesses x and y, it will fail whether it runs in partition A or B. We call this effect the *access scatter* problem.

Our protocol does not suffer from the access scatter problem. When replicas of some data items are split among partitions, our protocol concentrates write access rights on all data items in one (the largest) partition, the one with a "dynamic majority" of *nodes*, rather than replicas of any particular data items. In the example, users can write both x and y in partition B, so transaction T will succeed if it runs in this partition.

Second, the dynamic accessibility protocol improves the fault-tolerance of write operations at the expense of the fault-tolerance of reads. Indeed, unless a data item is represented in a partition by a "dynamic majority" of replicas, both writes *and* reads to this data item are disallowed. For instance, if transaction T above just reads x and y, it would still fail in either partition. Thus, one of the two motivating properties of the ROWA scheme (high fault-tolerance and efficiency of reads) is compromised. (The high efficiency property is preserved, because if a data item is accessible, it can be read by contacting any single replica in the view.) In contrast, our protocol allows the data item to be read in *any* partition that has a non-stale replica of x. In particular, transaction T in our protocol would succeed in either partition.

Third, as in other existing dynamic protocols, system reconfiguration in the dynamic accessibility protocol (new view formation in this case) runs as an atomic transaction that may interfere with and delay user transactions. Moreover, this transaction involves communication with all sites to be included in the new view (not just sites that replicate a given data item), and messages exchanged include the entire contents of databases stored at those sites. Thus, the reconfiguration transaction is very large. Even if it runs infrequently, the disruption of service caused by its execution may be prohibitively long. In our protocol, new epoch formation is more efficient (messages exchanged during epoch formation do not include the contents of any data items) and, most importantly, runs asynchronously with user transactions.

An advantage of the dynamic accessibility protocol over ours is that the former directly incorporates different voting-based quorums for different data items. For instance, some data items may use majority quorums while others may use ROWA quorums. Our protocol can be used only when all data items utilize ROWA quorums.

A detailed performance comparison of the two protocols is presented in Section 6.6.

6.5.2 The Dynamic Data Distribution Protocol

The *dynamic data distribution (D^3)* protocol is based on *partition functions (PF)*. A partition function is a mapping of data items to nodes replicating them. The corresponding *participation set (PS)* is the set of all nodes on which any data replicas are stored under

this PF. The partition function is stored on every node in the system. In the absence of system reconfiguration, a read is accomplished from any replica, and a write performs on all replicas of the data item (as defined by the current PF).

Partition functions serve roughly the same purpose as our epochs. When system reconfiguration is needed (e.g., failures are detected, or the load on some servers changes significantly), a new PF is generated and distributed to all members of its PS. Different PFs are distinguished by identifiers, with more recent PFs receiving greater IDs. If the network partitions, at most one region is allowed to form a new PF. This region is chosen as follows. Let n be the number of replicas of the data item replicated on the fewest number of nodes under the current PF. A region is allowed to form a new PF if it contains at least n nodes from the current participation set, and at most $n - 1$ nodes from the current participation set are left out. This rule ensures that at most one region can form a new PF. In addition, the region chosen according to this rule has a property that it (and thus any valid PF) always contains at least one replica of every data item.

New PFs are distributed using the two-phase commit protocol. Once nodes receive the new PF, they start redistributing replicas according to the new mapping. When all members of the new PF's participation set complete this task, the previous PF is forgotten. During the transitional period, when some data items are mapped to nodes according to the old PF and some according to the new PF, a write operation is required to contact all nodes on which a replica of the data item is stored under either the old or new PF. A read can perform from any one of these replicas. In fact, there may be even more than two active PFs in the system at a time, i.e., the protocol does not have to wait for a current reconfiguration to complete before initiating a new one. In this case, a write would need to contact all nodes that have a replica under any of the active PFs.

When a node p recovers from a failure (or gets re-connected), its data may have become stale. Node p infers conservatively that all its replicas are stale if there are any newer PFs in the system from which p was excluded. Indeed, since every PF contains replicas of every data item, a write to any data item could have occurred using a newer PF that does not include p, making p's replica of this data item obsolete.

This protocol shares some advantageous properties with our protocol. To our knowledge, this is the only protocol (besides ours) that guarantees correctness under network

partitionings and does not suffer from access scatter. Also, as in our protocol, system reconfiguration in the D^3 protocol is asynchronous with user operations.

The main difference between our protocol and the D^3 protocol is that our protocol is better suited for partition-prone wide-area networks. For instance, if all data items in the system are represented by three replicas, ($n = 3$), the D^3 protocol can only tolerate a partitioning where at most two ($n - 1$) nodes get disconnected from the rest of the system. Any other partitioning will cause the protocol to shut down the system. In contrast, our protocol would continue to provide full access to any data in the majority partition, and some access to data in minority partitions.[4]

Another limitation of the D^3 protocol, even on networks that rarely partition, is that the number of simultaneous node failures it tolerates is determined by the smallest number of replicas that any data item has. Thus, to make one data item more available, one would have to increase the number of replicas of *every* data item in the system. Our protocol has more flexibility in this respect: one can improve the availability of data items selectively, by increasing the number of replicas of these data items only. In fact, with our protocol, some data may not be replicated at all. In the D^3 protocol, this would mean the complete loss of fault-tolerance.

Finally, the requirement of the D^3 protocol that the operational region must always contain replicas of every data item is sometimes too restrictive. A partition may not contain some data item but continue providing useful service to the applications that do not need this data.

On the other hand, the fact that every PF in the D^3 protocol contains a replica of every data item makes determining stale replicas very simple. A recovering node marks all its replicas stale if there is a newer PF in the system that does not include this node. So, there is no need to maintain the *latest-epoch* array and dormant data lists, as in our protocol. Also, unlike our scheme, system reconfiguration operations do not have to be mutually exclusive with each other.

[4] If a minority partition contains all replicas of a data item, our protocol allows both read and write access to this data item in this partition; if a minority partition contains some but not all replicas of a data item, the data item can be read but not written.

6.5.3 Primary Copy Protocols

We next discuss the *primary copy* protocol, another approach that allows reads to be accomplished from a single replica. This scheme is very different from the ROWA approach. Therefore we compare it not so much to our specific ROWA scheme, but to ROWA-based protocols in general.

In primary copy protocols (e.g., Lis91 and Man89), every data item has a distinguished replica, called the *primary copy* (or just primary). All reads and writes to data items are directed to nodes that have primary copies for these data. Reads are serviced by the primary alone, while writes are performed (by the primary) on a majority of all replicas before control returns to the user. When the primary fails, a new primary is elected from the remaining replicas. A majority of replicas must agree on the new primary for it to be elected.

As the ROWA scheme, primary copy protocols allow reads to perform by contacting a single replica (which is why these protocols are discussed here). However, the primary copy approach lacks several advantages of the ROWA scheme. Read requests from geographically distributed clients cannot be serviced by nearby replicas: all requests must go to the primary. For the same reason, workload due to reads cannot be shared among all replicas. Compared to our specific ROWA scheme, the primary copy protocols suffer from the access scatter problem. Also, failure of the primary replica triggers the election process, during which access to the data item is suspended; in our protocol, system reconfiguration is done asynchronously.

On the other hand, the primary copy approach has some advantages over ROWA. Locks have to be maintained only by primary replicas, because these replicas see all operations. This may result in earlier detection of inter-transaction conflicts. Indeed, in a distributed system using the ROWA scheme, different transactions can hold conflicting locks on different replicas of the same data item at the same time. (Of course, they will eventually come into collision on some replica.) Under the primary copy approach, all locks on the same data item are obtained at the same site, so conflicts are detected immediately. Also, the primary copy protocols may be less prone to deadlocks, since fewer nodes have to be locked.

The above differences suggest situations where one of these two families of protocols would be preferable. In general, if the system involves data access from geographically distributed clients, or some data items are potentially so heavily read that the server overloading is a concern, the ROWA approach would be beneficial. On the other hand, the primary copy would be a better choice if the system is confined to a local-area network (so that the primary copy is close to all clients), and if server overloading can be solved by placing primary replicas of different data items on different nodes. Also, high-contention workload suggests considering the primary copy approach. However, separate research is needed to investigate in detail the various tradeoffs between the primary copy and ROWA.

6.5.4 Other Protocols

Among earlier protocols, the available copies protocol [5], the views protocol ([14], generalized in [15] and [16]), and the missing writes protocol [13] address the fault-tolerance of writes in the ROWA scheme.

In the available copies protocol [5], a read is serviced by any operational replica, and a write is performed on all *available* (i.e., currently operational) replicas. *Validation* is performed at transaction commit time to ensure that the set of operational replicas of the data items involved in the transaction has not changed during transaction execution. The available copies protocol provides a very high level of data availability, but guarantees data consistency only under node failures. In contrast, our protocol guarantees consistency under both node failures and network failures.

The views and missing writes protocols can tolerate both node and network failures. A main limitation of these protocols is that a (possibly weighted) majority of replicas of a data item must be operational and connected for a logical write of this item to be successful. In fact, in many cases, both protocols require a majority of replicas to be available even for *reading* the data item. Thus, the data availability for read operations is significantly reduced (as compared to the basic ROWA scheme). Also, the missing writes protocol uses ROWA only when all replicas are operational and connected, and resorts to majority consensus otherwise. Hence, even a single replica failure causes a

substantial read performance penalty, which is incurred for the whole duration of the failure. Our protocol is free of these limitations. In particular, the data in our protocol may remain available for reads and writes even if only one replica of the data item is operational.

An additional limitation of the views protocol is that all replicas that are to adopt a new view must be initialized; during initialization, they must contact a majority of all replicas in the system to determine a current replica from which the data can be copied. Moreover, replica initialization is run as a transaction that can interfere with user transactions. In our protocol, the initialization of replicas to be included in a new epoch is often avoided. When initialization is necessary, it is accomplished by contacting a single current replica from the same epoch.

6.6 Performance

In this section, we compare the performance of our protocol with the basic ROWA scheme (a protocol with ROWA quorums and no system reconfiguration) and with the dynamic accessibility (DA) protocol, the most closely related existing alternative to our protocol. We created a simulation model of a distributed replicated system, implemented the three protocols, and measured their performance. We used the event-driven simulation package MODSIM, from CACI, Inc., to implement our models.

6.6.1 The General System Model

In our model, there are clients, acting as transaction coordinators, and servers replicating the data. Each client issues a transaction, waits for the result, and then issues the next transaction after some *think time*. We model both nodes and point-to-point communication links. One node can communicate with another if there is a path between them such that all intermediate nodes and links are up. Every node or link can fail and repair independently according to the Poisson stochastic processes. The mean time to failure and repair are parameters of the model. A node can fail or be disconnected at any time, even if it is in the middle of executing an operation. When this occurs during the execution of a commit protocol, some participants can be stuck in the waiting state,

not knowing the outcome of the operation[5]. To determine the outcome, they repeatedly ping the other participants. This can also be required in the case of a recovering node if it finds itself in the midst of commit protocol executions.

We consider a system consisting of several LANs connected via a wide-area network. Each LAN has a server and an equal number of clients connected to it. Clients and their connections to their server are considered fully reliable. The server provides access to the data items it stores and also serves as a gateway to the WAN. Servers keep multiple data items and maintain replica locks. Strict two-phase locking is used for concurrency control. System reconfiguration is performed by servers. Each server polls its peers periodically to detect changes in system topology (To provide a relatively steady rate of node-polling, time slots for node-polling are assigned to each server based on the number on nodes in the server's epoch.)

6.6.2 The Server Scheduler

A server processes requests one at a time. All incoming messages are placed in a queue. Whenever the server is free, the scheduler scans the queue in FIFO order and takes the first message that can be processed. (Some messages cannot be processed if, for example, they request data locked by other transactions.) To reduce the possibility of deadlocks, if any data items requested by a message are locked, the message does not lock any data items at all, waiting for the full set of needed data to become unlocked. The only exception is the messages issued by reconfiguration transactions in the dynamic accessibility protocol. These messages lock *all* data items on the server. So, when we followed the above discipline, we observed that these messages were being starved. We then modified the scheduler to allow these messages to obtain locks gradually as they become free. Thus, such a message locks all replicas that are not locked at the time the message is first scanned by the scheduler; then, any time a lock is released, it is given to this message until all replicas are locked.

Once a message is taken off the queue for processing, the server becomes busy for

[5]Note that commit protocols are employed not only by user transactions, but also by the operations that perform system reconfiguration in our protocol. The dynamic accessibility protocol uses a specialized commit protocol with the possibility of timeouts.

a period of time equal to the service demand for this message type. The overhead of scheduling is assumed to be negligible.

6.6.3 The Transaction Model

We use the model of transactions from [19]. Transaction execution consists of the following steps.

1. **Read step.** T first reads all data items it needs. A transaction reads a data item by accessing a site that has a replica of that data item subject to the constraints of the protocol used. If several sites can be used, one is chosen at random.

   We assume that transactions know in advance all the data they will want to read. Therefore, all read requests are issued at once in parallel. If several data items are being read from the same node, all requests to this node are sent in a single message. If some read requests could not be satisfied (due to failures or replicas being stale), these reads are attempted on alternative replicas, according to the protocol used. When all alternative replicas have been tried with no success, the transaction is aborted.

2. **Compute step.** The coordinator decides which of the data items obtained in the read step must be modified and computes their new values. Thus, we do not have blind writes: all data written by the transaction are first read. The time spent in the compute step is assumed to be negligible compared to other steps, which require disk access.

3. **Write step.** New values of the modified data are recorded on all replicas, according to the protocol.

6.6.4 Simulation Parameters

The simulation parameters are listed in Table 6.1. All three protocols behave the same in a fully operational system. Since the goal of this study is to compare the behavior of the protocols when failures and repairs do occur, we exaggerated the probability of

Table 6.1: Simulation parameters (ROWA protocols).

Parameter	Value
Mean time to fail	72000
Mean time to repair	8000
Service demand for: Read request Write request Abort Commit	30
Inter-polling time	50
Service demand for any messages issued by the reconfiguration operation	0
Think time	Exponential with mean 200
Timeout	200

failures (which is $\frac{8000}{8000+72000} = 0.1$), as well as the mean times to failure and repair, as compared with service demands.

However, we needed to preserve the fact that system reconfiguration occurs quickly relative to the periods between failures and repairs. Otherwise, we would be measuring mostly the transient behavior occurring after a change in system topology and before the protocol reconfigures quorums accordingly. Thus, we assume short inter-polling time, based on which servers determine when they should poll all other servers for the purpose of failure/repair detection. Also, all service demands for messages issued by reconfiguration operations are assumed to be 0. This correctly reflects the reality that failures/repairs are detected quickly, and the service demands of the reconfiguration operation are negligible relative to the rate of failures and repairs in the system.

In addition, reducing the service demands of a reconfiguration operation versus those of user operations favors the dynamic accessibility protocol over ours. Indeed, in the DA protocol, reconfiguration operations compete for locks with user operations. Hence, reducing service demands for reconfiguration reduces the lock contention in the system. In our protocol, reconfiguration is done asynchronously and does not affect lock contention regardless of the service demands.

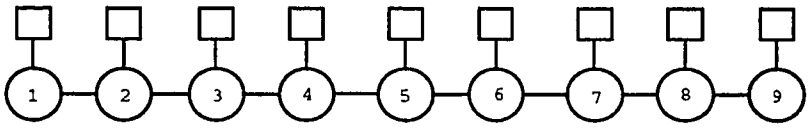


Figure 6.4: The weakly connected ROWA system.

6.6.5 Experiments

We concentrate on the behavior of the protocols under network partitions: when nodes fail, our protocol and DA protocol reconfigure the system in a similar way. The only difference is that system reconfiguration in DA protocol is done as a special transaction that competes for locks with user operations.

Thus, in this study, we consider a weakly connected system where only links fail. The system is shown in Figure 6.4. Nine servers are located in different geographical areas. Each has one client issuing transactions. Each server keeps replicas of ten data items, and each data item is replicated on three servers. Data are distributed uniformly among the servers, so that no client is too far from any data. Specifically, data items 1-10 are kept on servers 1, 4, 7; data items 11-20 are replicated on servers 2, 5, 8; and data items 21-30 are located on servers 3, 6, 9.

In all experiments, the simulation was run until the main measure – the percent of aborted transactions – was obtained with 90% confidence level and a confidence interval not exceeding 10% of the value measured. (A method described in [39] was used to determine the duration of simulation.) We assume that by then, the system reached a steady state, and other measures obtained in the same run were representative of the system properties. Typically, the simulation extended over ten million time units and was always longer than 6 million units. The number of attempted transactions during simulation runs was usually between ten and twenty thousand. To reduce the initial skew (all experiments started when the system was fully operational), we started collecting statistics after the simulation clock reached 400,000 time units.

In the first series of experiments, we consider update transactions, to compare the levels of fault-tolerance for write operation provided by different protocols. Specifically, the workload consists of transactions that read a fixed number of data items and then

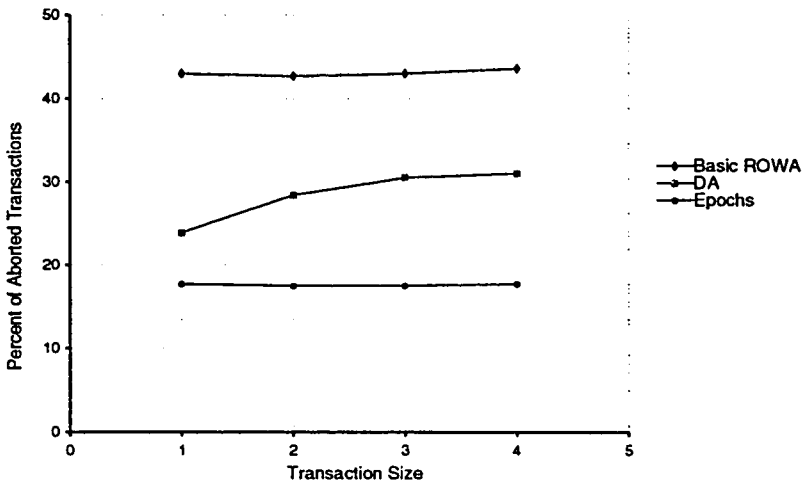


Figure 6.5: The percent of aborts of update transactions in the ROWA system.

write one of them. (The number of data items involved in the transaction is called the transaction size.)

Figure 6.5 shows the percent of aborted transactions under the three protocols for different transaction sizes. As expected, the basic ROWA scheme shows the worst results. This is not surprising, because if any of the three replicas of the data item written by a transaction is disconnected from the client, the transaction is aborted.

Our protocol provides better availability than the DA protocol, even for transactions of size 1. The difference for transactions of size 1 is caused by the interference of reconfiguration operations with user operations in the DA protocol: since reconfiguration transactions are very large (recall that they locks every replica of every data item in the system), they are likely to cause deadlocks and delays that lead to timeouts of user transactions. The access scatter effect does not come into play here because none of the transactions accesses multiple data items. Also, transactions in this case always write to the data item that had been read. Therefore, the fact that the DA protocol often requires multiple replicas of the data item for a successful read is not a disadvantage: even if our protocol successfully read a data item in a situation where the DA protocol fails,

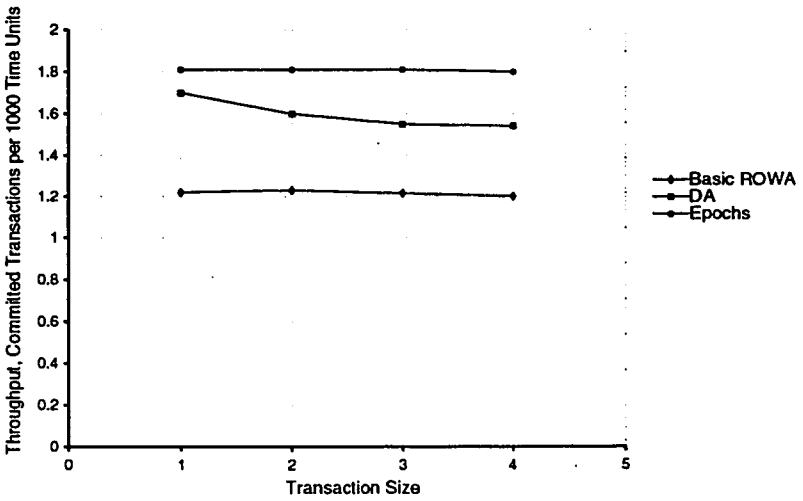


Figure 6.6: The throughput of committed update transactions in the ROWA system.

ours will abort the transaction when attempting the write. However, when the size of the transactions increases, both these problems start playing an increasingly important role (in addition to the remaining interference from reconfiguration transactions, in the DA case). Consequently, the advantage of our protocol increases.

On the other hand, when the DA protocol aborts transactions, it does so very quickly. Indeed, in many cases, aborting a transaction (specifically, because reading the data is impossible) only involves checking the accessibility of all data items involved. In our protocol, this often means several tries and timeouts. We observed that the average response time of aborted transactions in the DA protocol was often an order of magnitude less than in our protocol. So, the question arises as to whether our protocol has any advantage in terms of *committed* transactions. Figure 6.6 gives the throughput of committed transactions. It shows the same trends as the previous graph, although the advantage of our protocol is somewhat smaller. Thus, relative to our protocol, the DA protocol does not simply abort more transactions more quickly: it does so at the expense of the number of committed transactions.

In the next series, the workload consists of read-only transactions. The objective is to

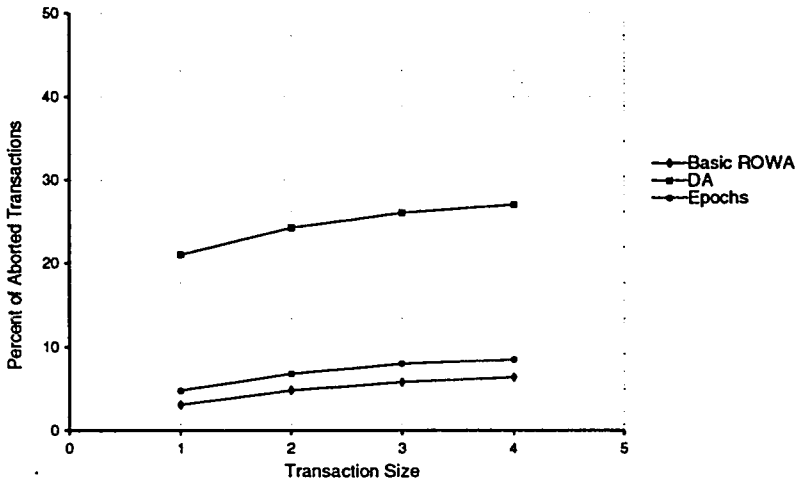


Figure 6.7: Percent of aborted read-only transactions in the ROWA system.

see whether the improvements in write fault-tolerance that DA and our protocols achieve over the basic ROWA scheme come at the cost of compromised fault-tolerance for reads. Again, we begin by measuring the percent of aborted transactions for various transactions sizes (Figure 6.7). Obviously, the basic ROWA protocol gives the best results in this case, because it always allows reads to succeed without any restrictions as long as there is a single reachable replica. But the reduction in read fault-tolerance in our protocol is very small. On the other hand, the DA protocol exhibits much worse read availability than both other protocols, due to the combination of the three factors described earlier. These results are also confirmed by Figure 6.8, which shows the throughput of committed transactions.

Thus, we conclude that our protocol provides fault-tolerance to writes in the read-one-write-all scheme without compromising the fault-tolerance of reads. On the other hand, the DA protocol provides better write fault-tolerance than the basic ROWA scheme (yet the improvement is less significant than in our protocol), but it trades off read fault-tolerance for achieving this.

To our knowledge, the protocol proposed in this chapter is the first protocol designed

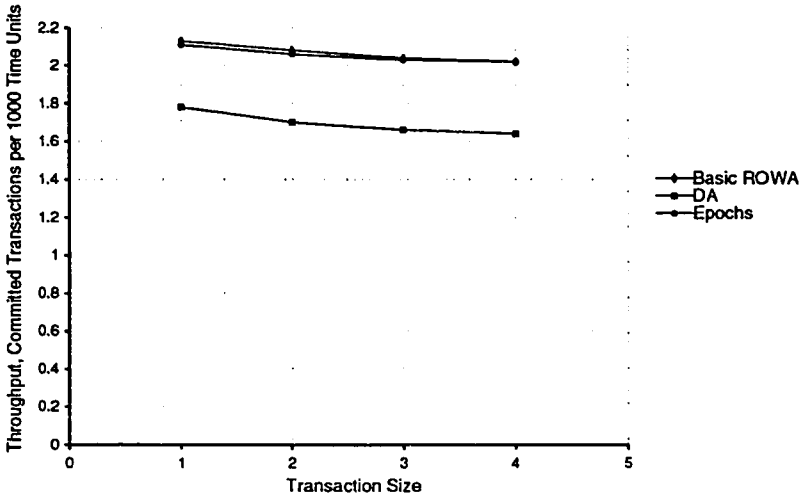


Figure 6.8: The throughput of committed read-only transactions in the ROWA system.

for a partition-prone network that provides fault-tolerance to writes in the ROWA scheme without compromising read fault-tolerance or efficiency. We hope this protocol will encourage wider use of ROWA-based replication in wide-area networks.

6.7 Summary

We have presented a new read-one-write-all (ROWA) replica control protocol that adjusts to failures and repairs as they occur in the system to keep the data available. This is achieved by making the epoch mechanism, previously applicable to non-ROWA schemes only, usable within the ROWA discipline. In our protocol, a write to a data item can succeed if it can perform on all replicas of the data item included in the transaction's *epoch*. A read of a data item can succeed if it can perform on a single current replica of the data item from the transaction's epoch. Therefore, data is available for both read and write accesses even if only one operational replica of it remains in the system, provided not too many failures occur undetected so that the protocol could switch to new epochs as failures occur.

At the same time, our protocol retains both major benefits of the original static

ROWA protocol. First, if communication failures divide non-stale replicas of a data item among different partitions, the data item remains available for reads in any partition that contains at least one such replica. Second, even in the presence of failures, our protocol never needs to contact more than one current replica to perform a logical read, and then it needs only one round of message exchange with this replica. It can also always determine whether a replica is stale using only the local state of the node on which the replica resides.

Another new property of our protocol is that system readjustment is done asynchronously with user transactions. Therefore, no interference with user transactions is possible. In addition, when switching to a new epoch, our protocol is very careful about which replicas should run the initialization procedure. As a simple example, if all replicas of a data item failed, and if the replica that failed last is the first to become operational again, its information is guaranteed to be current without additional work. Our protocol identifies these sorts of circumstances and does not require replicas to run initialization in such situations. Running excessive initializations is undesirable, because affected replicas are unavailable (at least for reads) while initialization is in progress, .

Finally, all messages exchanged in our protocol during the execution of logical reads and writes have constant lengths independent of the number of nodes in the system. Therefore, our protocol should scale well.

The read-one-write-all scheme would be very attractive (because of its high fault-tolerance and efficiency of read operations) except for extremely low fault-tolerance of writes. Thus, our protocol, by improving write fault-tolerance without compromising beneficial read properties, should encourage a more widespread use of the ROWA scheme in practical systems.

Chapter 7

Replication Management in Systems with Partial Writes

This chapter studies the effects of write operation semantics on the performance of replica control protocols and describes a protocol designed specifically for systems with *partial* write semantics. The method proposed is equally applicable to structure- and voting-based (non-ROWA) quorums.

Systems exhibit two kinds of write operation semantics, *total* and *partial.* In systems with total writes, a write always replaces the entire contents of the data item. In systems supporting partial writes, a write is allowed to update just a portion of the information in the data item.

If the write always replaces the data item with an entirely new value, it can execute on stale as well as on current replicas (since all previous contents of the replica are completely overwritten anyway). Hence, in the common case of the absence of failures, the write coordinator can communicate only with the nodes belonging to an arbitrary write quorum: the coordinator will replace the data item with a new version on all nodes from the quorum, thereby guaranteeing that a subsequent read (if successful) will see the latest version of the data. This scheme allows good load sharing (because requests from different coordinators can be served by nodes from different quorums) and light network traffic (since a write involves communication with only replicas from a quorum).

Unfortunately, it is often impractical in information systems to manage replication at such a small data granularity that all modifications of data items become total. Instead, operations in these systems update just a portion of the information in the data item. For example, in object-oriented databases, objects are typically large, and operations associated with them are typically complex. Therefore, it would be prohibitively expensive to break the object into many components and execute a replica control protocol every time the operation accesses a new component, rather than doing so once for the entire operation. Also, in many existing replicated file systems, (e.g., [66], [27]), to save on management and replica control costs, replication is maintained at the granularity of files or even larger units of data, so these systems have partial write semantics as well.

In systems with partial writes, a write can be applied to current replicas only. Therefore, it was thought that the write coordinator must collect permission from a write quorum of *current* replicas in order to perform the write [18]. This implies that the coordinator must either apply the write to *all* operational nodes having the replica, or always perform all writes on the same write quorum of nodes. In the former case, the system incurs high message overhead. In the latter case, if any replicas in this quorum fail, additional replicas must be synchronously brought up-to-date during the write operation, which can have a significant impact on response time. Also, in both cases, the system loses the advantage of load sharing provided by replication.

The protocol described here (originally proposed in [59]) supports partial writes while preserving load sharing and greatly reducing the risk of having to propagate updates synchronously. We observe that while the requirement that the write operation must change the state of at least a write quorum of replicas is fundamental for the protocol to be correct, it does not mean that the write itself must be applied to all these replicas. Instead, the coordinator can apply the write to current replicas only, while marking the other replicas from the write quorum stale. Then, a subsequent operation (if successful in obtaining responses from a quorum) is guaranteed to see either a current replica or a replica marked "stale," and thus to identify the current replicas correctly.

The distinction between good and stale replicas, while guaranteeing correctness, does not by itself make it practical to service different operations by accessing different quorums. Indeed, if an operation fails to contact a current replica after obtaining a quorum

of permissions, it must perform extra rounds of message exchange with additional replicas in an attempt to contact current replicas. The system thus incurs an additional delay even in the absence of failures. In addition, if a write operation obtained only a single response from a current replica among its quorum of permissions, the operation will perform on this current replica only. Therefore, the system will be vulnerable to a single node failure (at least until other nodes "catch up" by copying the latest version).

To address these problems, we add a special data item, a *good replica list*, to the control state of the replicas. The good replica list of replica x records (to the best of x's knowledge) the identities of current replicas in the system; it is updated as part of the execution of the write operation. The idea is that, unlike actual data, the good replica list is always overwritten completely. Hence, a write can always record the latest value of the good replica list on all participating replicas, regardless of whether they are current or stale. Similar to the quorum-based protocols supporting total writes only, the coordinator of a (successful) write operation always learns the latest value of the good replica list after having contacted any write quorum of replicas. Hence, the coordinator knows the identities of at least some current replicas. Then, if the number of "good" replicas already contacted is less than *safety threshold* t (a parameter that specifies how many *simultaneous* failures should be tolerated by the system), the coordinator includes additional "good" replicas in the set of nodes on which it performs the write. Moreover, we prove that no permission from these additional replicas is needed, so there are no additional rounds of message exchange involved.

Another benefit of keeping the good replica list is that it enables stale replicas to obtain missing updates very effectively by contacting a node with a newer replica directly, without searching. Because stale replicas can recover from any of at least t replicas, our protocol can tolerate up to t arbitrary simultaneous node failures. Thus, there is no possibility of a vulnerability window.

All considerations in this chapter apply on a per-data-item basis, so we will often use shortcuts and say "all nodes" instead of "all nodes that have a replica of the same data item," "all writes" instead of "all writes applied to the same data item," and so on. Moreover, we will use "node" and "replica" interchangeably, as if nodes replicated a single data item.

We assume that a commit protocol that ensures all-or-nothing execution of transactions begins at the end of the transaction's execution: the systems with "early prepare" commit protocols [68] (see Section 1.3.1) are expressly excluded.

As usual, we also assume that strict two-phase locking is used for concurrency control, i.e., all locks the transaction obtains during its execution are released only after the transaction terminates.

7.1 The Protocol

Each replica maintains the following state: a version number, a stale-data flag, and a *good replica list* which is the list of current replicas found during the last write operation. The meaning of the version number depends on the state of the stale-data flag. For replicas marked "current," it is equal to the number of write operations applied to the replica. For replicas marked "stale," it stands for the *desired version number*, i.e., the maximum version number in the system known to the replica.

Also, each replica maintains three separate locks. The *read and write permission locks* ensure that the replica does not issue permission to multiple writes, or a read and a write, at the same time. The *commit lock* is used by the atomic commit protocol. Table 7.1 shows the compatibility table for these locks. Note that a replica is allowed to participate in the commit protocol even if it did not issue permission to the operation, or even if it issued permission to a different operation. This not only eliminates some rounds of message exchange, but (as we will see in Section 7.3) also increases concurrency in the system. Initially, all nodes have identical replicas with version numbers 0, their good replica lists include all nodes, and all replicas are marked "current".

The protocol uses a system parameter `SAFETY-THRESHOLD`, which determines the number of simultaneous node failures the protocol can tolerate.

The algorithm for the write operation performs the following steps.[1]

1. The coordinator requests permission from a randomly chosen write quorum of nodes.

[1] An example clarifying the execution of this protocol will be given in the next section, after the proof of correctness.

Table 7.1: The compatibility table of replica locks.

	Read Permission	Write Permission	Commit
Read Permission	yes	no	yes
Write Permission	no	no	yes
Commit	yes	yes	no

Each node that received the request obtains the write permission lock for its replica and responds with its state. If the coordinator receives all non-`RPC-CallFailed` responses (this is the common case of the absence of failures), it returns control to the application and, in the background, executes step 3.

2. If the coordinator failed to obtain a write quorum of permissions in step 1 (i.e., some of the responses were `RPC.CallFailed`), it requests permission from all remaining replicas. Once enough non-`RPC.CallFailed` responses arrive to complete a write quorum, the coordinator returns control to the application and, in the background, moves to step 3. The rest of the permission messages are received and dismissed in the background. (The coordinator sends a "permission dismissed" message to these extra replicas, so that they can release their permission lock.) Upon receiving all responses, if the coordinator fails to collect a write quorum of non-`RPC.CallFailed` responses, the transaction is aborted.

3. Once the coordinator has permission and the state information from a write quorum of replicas, it can identify at least some current replicas in the system. These are members of the good replica list from any response with the maximum version number. (As we will see in the next section, good replica lists from all such responses are identical; they are called the *current* good replica list below. We will also show in the next section that, there are at least `SAFETY-THRESHOLD` members in the current good replica list.) Also current are replicas (even if they are not members of the current replica list) whose responses carried the maximum version number and stale-data flag "current". The coordinator then constructs a `good-list`, which includes all current replicas con-

tacted thus far plus additional replicas from the current good replica list if needed for the `good-list` to contain at least `SAFETY-THRESHOLD` members. It also constructs a `stale-list`, which includes all non-current replicas from the obtained write quorum. The maximum reported version number will be denoted `max-version` below.

At transaction commit time, the following steps are executed for every data item written by the transaction:

4. Let `quorum-list` be a set of replicas that are members of the quorum collected in step 1 or 2. The coordinator sends a "prepare" message to all replicas that are members of `good-list` or `stale-list` (note that these replicas include `quorum-list`). Each replica that receives this message obtains its commit lock and responds with a prepare acknowledgement, which also includes the replica's version number and stale-data flag. If the coordinator receives all acknowledgements (the common case), it executes step 6.

5. If the coordinator failed to receive some acknowledgements (i.e., it received `RPC.CallFailed` instead), its actions depend on the identities of the failed replicas. If any members of `quorum-list` failed to acknowledge the prepare message, the transaction is aborted. Otherwise (i.e., if the failed replicas are among the additional current replicas included in step 3), the prepare message is sent to all remaining replicas. Once the prepare acknowledgement is received from a total of `SAFETY-THRESHOLD` current replicas (the ones whose acknowledgements carry version number `max-version` and stale-data flag "current"), the protocol moves to step 6. Remaining acknowledgements are received in the background. The coordinator sends a "prepare dismissed" message to these extra replicas enabling them to release their commit lock.

If the coordinator has received the acknowledgement to its prepare message from some current replicas, but their number is less than `SAFETY-THRESHOLD`, it brings the necessary number of stale replicas up-to-date using one of the current replicas as the source of propagation, and then moves to step 6. If no current replicas acknowledged the prepare message, or the above update propagation failed, the transaction is aborted.

6. The coordinator updates its `good-list` and `stale-list` based on the version number

and stale-data flag from the prepare acknowledgements received. Replicas that reported version number `max-version` and stale-data flag "current" are put into `good-list`. The rest are included into `stale-list`. Updating these lists is required, mainly, to exclude replicas added to `good-list` in step 3 that failed to acknowledge the prepare message. Also, additional current replicas found in step 5 can be included into `good-list`, and some replicas found stale in step 3 could have become current due to asynchronous update propagation.

Finally, the coordinator sends the commit message to members of `good-list` and `stale-list`. With this message, it includes `good-list`, `stale-list`, a new version number equal to `max-version + 1`, and the updated portion of the data item. Upon receiving this message, nodes that find themselves in `good-list`, increment their version number and install the updated data. Replicas that are members of `stale-list` mark themselves "stale" and adopt the new version number received. In addition, every replica records `good-list` as its new good replica list and releases its commit lock and permission lock (if held by the current transaction). Note that the permission lock may be held by another transaction, in which case it remains locked.

The read protocol is similar to the write protocol. During the first phase, the coordinator identifies the current replicas by contacting a read quorum of nodes and learning the maximum version number among all responses. Current replicas are those whose responses carry stale-data flag "current" and the maximum version number. Also current are all members of the good replica list from any response with the maximum version number. In the second phase, the coordinator reads the required portion of the data from one of the current replicas.[2]

Good replica lists allow very efficient asynchronous update propagation. The initiators are the stale replicas, which use their good replica lists to locate provably more recent replicas.

First, the initiator of the propagation tries to contact any replica from its good replica list. If successful, the initiator obtains the version number, stale-data flag, and

[2]If replicas that are marked "current" would respond not only with their state but with the requested data as well, the read operation could be done in one phase in most cases. The design choice depends on the specifics of a particular system.

good replica list from the node contacted. If the source replica is marked "current", the initiator also obtains the missing updates, to make its replica identical to the source replica. If the newly obtained stale-data flag is "current", the propagation is done. Otherwise, the algorithm is run again with the new good replica list.

If no node from the good replica list is available (they all fail or partition), then the initiator tries to contact any other node with a more recent replica than its own. (The hope is that some node might have done the propagation before all members of the initiator's good replica list failed). The more recent replicas are the ones with greater version numbers, or those with the same version numbers that are marked "current". As before, the propagation involves obtaining the new state if the source node is marked "stale", and both the new state and the missing updates if the source is marked "current". As soon as the initiator succeeds in obtaining a newer state, it goes back to the "pointed" search for a current replica using its new good replica list.

Note that the source and destination nodes need not hold any locks during their communication. All that is needed is that the source replica send, and the destination replica install, a consistent snapshot of the data and control state. Also, as in the recovery algorithm of our ROWA scheme (Chapter 6), the source replica should not send the data written by an uncommitted transaction. This prevents the propagation of transient data written by a transaction that later aborts. If strict two-phase locking is used for concurrency control, the last condition is easily enforced by the source replica acquiring its read lock prior to sending its data to the target replica.

7.2 Correctness and Other Properties

Lemma 20 *Neither two write operations, nor a read and a write operation, can obtain their quorum of permissions at the same time: one must unlock a replica before another can complete its quorum.*

Proof. Follows directly from the definition of quorums, which says that any two write quorums, as well as any read and write quorum, have at least one replica in common. □

As a consequence of Lemma 20, we can assign sequential numbers to successful write operations according to the order in which they obtain their quorums of permission. In particular, one can use induction on these numbers. Below, we refer to the write quorum used by the i-th write operation as the i-th write quorum, and to the good replica list constructed by the i-th write operation during step 6 of the protocol as the i-th good replica list.

We will say that a write operation has performed if it released the write permission lock on at least one node from its quorum. (A write releases the permission lock on nodes from its quorum during step 6 of the protocol by sending them the commit message.) We will say a node has performed a write if it received and processed the commit message from the coordinator of that write operation.

We are going to use Lemma 1 to prove the correctness of the protocol. However, the additional requirement for correctness imposed by the partial write semantics is that a write installs new data on current replicas only, and a read obtains information from a current replica.[3] Thus, in addition to the conditions of Lemma 1, we need to prove that these requirements are satisfied as well. Informally, a replica is current if it either performed all write operations executed in the system in the right order, or if it copied the data and control state information from another current replica. Formally, the following replicas are called current.

Definition 2 *A replica p is current with respect to version number i if its version number is i, it is not marked "stale", and: (1) $i = 0$, or (2) the first, ..., the i-th write operation has performed on p in this order, or (3) p copied the full state (both data and control information) from a current replica w.r.t. version number $j \leq i$, after which the $(j+1)$st, ..., the ith write performed on p in this order.*

Replicas that are current w.r.t. the maximum version number in the system are called current replicas.

Lemma 21 *In the system state after the m-th write performed and before the $(m+1)$-st write obtained its quorum, the following is true: (1) all nodes that performed the m-th*

[3]For the total writes case, the former condition is not necessary, and the latter condition follows from Lemma 1 itself. Indeed, this lemma requires a read operation to read from a replica on which the previous write performed, and any such replica will be current in a system with total writes.

write have version number m; (2) m is the maximum version number among the replicas in the system; (3) all nodes with version number m and stale-data flag "current" are indeed current; (4) all nodes with version number m have the m-th good replica list; and (5) all nodes from the m-th good replica list that performed the m-th write have their stale-data flag "current" (and, hence, are current due to conditions (1) and (3)); all other nodes from the m-th good replica list have their commit lock acquired by the m-th write.

Proof. By induction on m. After the first write, the lemma is correct: initially all replicas are current, so the first write coordinator finds all replicas from its write quorum to be current. Also current are any additional replicas it may put into the first good replica list. Hence, the write is applied to all participating replicas, and they all increase their version numbers from 0 to 1, which becomes the maximum version number among all replicas. Also, all replicas that performed the write remain current after that. Because the only nodes with version number 1 are those that participated in the write, they all receive the first good replica list. Finally, no replica is marked "stale" after the first write.

Assume that the lemma is correct after i writes perform, and consider the $(i+1)$-st write. By the definition of quorums, the $(i+1)$-st write quorum has at least one node in common with the i-th write quorum. This node can give permission to the $(i+1)$-st write only after it performs the i-th write (since a write releases the permission lock on the members of its quorum only in step 6 of the protocol, when the commit message is sent). Therefore, by assumption, this node reports to the $(i+1)$-st operation version number i, which is the maximum version number in the system. Hence, the nodes that the $(i+1)$-st write puts into the $(i+1)$-st good replica list in step 3 are:

- All replicas from the $(i+1)$-st write quorum with version number i and stale-data flag "current". By assumption, they are all current.
- Some additional replicas from the good replica list taken from one of the replicas with version number i. By assumption, it can only be the i-th good replica list. Since the i-th operation obtains the commit lock on all these nodes before performing, for every such node, either its commit lock is held by the i-th write, or

it performed the i-th write. Therefore, by the inductive assumption, all replicas from this list whose commit lock is not held by the i-th write have version number i and stale-data flag "current", and hence are current.

Therefore, since replicas send their prepare acknowledgement to a write only after obtaining the commit lock on behalf of this write, it is guaranteed that all above replicas that respond with a prepare acknowledgement in step 5 of the $(i+1)$-st write will be found current. In addition, in step 6, the $(i+1)$-st write may include into the good replica list some additional replicas that, during step 5, reported version number i and stale-data flag "current" in their prepare acknowledgement. By assumption, these additional replicas are also current.

Therefore, all replicas put into the $(i+1)$-st good replica list are current and have version number i. Hence, on receiving the commit message from the $(i+1)$-st write, they apply the write (thus, they will be current after they perform the $(i+1)$-st write) and increment their version numbers (thus, they will all have version number $i+1$ after the write). Their stale replica flags remain "current". We have therefore shown that condition 5 of the lemma holds.

All other replicas that perform the $(i+1)$-st write are given version number $i+1$ and the good replica list constructed above. After the $(i+1)$-st write, these nodes are marked "stale". Hence, they run the propagation protocol. They can obtain a new state either from a member of their good replica list (which has been shown above to have version number $(i+1)$ and stale-data flag "current") or from a node outside their good replica list with a version number no less than their own. In the latter case, it can only be a node with version number $i+1$, because a write operation can increase the maximum version number by at most 1, and the maximum version number before the $(i+1)$-st write was i by assumption. It can also be only a node marked "current", since this is the only case when propagation is done between two nodes having the same version number, with a source not in the recipient's good replica list. Hence, a node that performed the $(i+1)$-st write can obtain the new state only from replicas with version number $i+1$ and stale-data flag "current". Therefore, after obtaining the new state, recipients will have version number $i+1$ (as before) and be marked "current", which means they will

no longer run the propagation protocol. Thus, we have shown that, after the $(i+1)$-st write, despite possible propagations, all nodes that performed the $(i+1)$-st write have version number $i+1$ (condition 1).

As already mentioned, a write can increase the maximum version number in the system by at most 1; any propagation only transfers version numbers between nodes and cannot increase them. Hence, $i+1$ is the maximum version number in the system after the $(i+1)$-st write (condition 2).

Before the $(i+1)$-st write arrived, no replica had version number $i+1$, so the first replica that receives version number $i+1$ does so during the $i+1$-st write operation. Hence, this replica receives the $(i+1)$-st good replica list. Other replicas can obtain version number $i+1$ either during the $(i+1)$-st write or by propagation. In either case, they get the $(i+1)$-st good replica list, together with the $(i+1)$-st version number. (For writes, all replicas are given the same good replica list. In the case of propagation, a replica always obtains both the version number and the good replica list from the source node.) Therefore, condition 4 holds.

Finally, we have shown that all replicas from the $(i+1)$-st good replica list are current and are marked "current". All other replicas participating in the $(i+1)$-st write are originally marked "stale" by the write operation. Therefore, any replica outside the $(i+1)$-st good replica list can obtain the state (version number $i+1$, stale data flag "current") only by a propagation that ultimately originated at one of the nodes from the $(i+1)$-st good replica list. But any propagation involving a source replica marked "current" results in the recipient replica having the same data item as the source replica. Because the source replicas are current, the recipients are also current (condition 3). □

Based upon these lemmas, the first theorem below says that the protocol is correct.

Theorem 5 *The protocol defined in the previous section provides one-copy serializability if S2PL is used for concurrency control at the level of individual replicas.*

Proof. From Lemma 21, it immediately follows that conditions 1, 2 and the part of condition 3 about write operations of Lemma 1 are satisfied. It also follows from Lemma

21 that writes are always applied to current replicas. With respect to read operations, any read quorum has at least one node in common with the write quorum used by the preceding write operation. By Lemma 21, this common node has the maximum version number of all replicas, which is the version number recorded by the preceding write. Also by Lemma 21, any replica with this version number and the stale data flag "current" is current, as is any member of the good replica list reported by the common node. Since these are the only replicas from which a read operation can read data, it is guaranteed to read from a current replica. □

From the proofs of Lemma 20 and 21, one can see why write coordinators can include additional replicas into the commit protocol regardless of who holds their permission locks, even if doing so renders incorrect responses these replicas sent to other operations. (Similar considerations hold for reads.) Obtaining permission locks from a write quorum of nodes is sufficient for serializability, so locking additional replicas is not necessary. As to identifying the most recent replicas on which to apply the update, the write protocol relies exclusively on responses with the maximum version number, and these responses never come from the nodes whose permission locks are bypassed by other operations. Thus, the responses from these replicas are always correct. Because it is this observation that makes our approach viable, we would like to emphasize it with an example.

Example. Consider a system S with 5 replicas ($x1$, $x2$, $x3$, $x4$, $x5$), majority-defined quorums, and safety threshold 2. Initially, all replicas are current, have version number 0, and a good replica list that includes all replicas.

Assume a write runs, collecting quorum $\{x3, x4, x5\}$ in step 1 of the protocol. Since these replicas are all current and their number exceeds the safety threshold, no additional replicas need be contacted. Then, in step 3, all replicas from the quorum are put in the new good replica list. Since no stale replicas have been found, the empty stale list is constructed. Finally, at commit time, the write performs on all replicas from the quorum, bringing the system to the state shown in Figure 7.1.

In this system state, assume that coordinators A and B initiate write operations at the same time, using the quorums $\{x1, x2, x3\}$ and $\{x3, x4, x5\}$, respectively. A has

	Version Number	Good Replica List	Stale-Data Flag
x1	0	{x1 - x5}	"current"
x2	0	{x1 - x5}	"current"
x3	1	{x3, x4, x5}	"current"
x4	1	{x3, x4, x5}	"current"
x5	1	{x3, x4, x5}	"current"

Figure 7.1: Example: the state of system S with 5 replicas.

	Version Number	Good Replica List	Stale-Data Flag
x1	2	{x3, x4}	"stale"
x2	2	{x3, x4}	"stale"
x3	2	{x3, x4}	"current"
x4	2	{x3, x4}	"current"
x5	1	{x3, x4, x5}	"current"

Figure 7.2: Example: the state of system S after the write operation A.

obtained permissions (and hence responses) from $x1, x2, x3$; B has obtained permissions from $x4, x5$ and is blocked, waiting for permission from $x3$. A is going to perform first. During the quorum-collecting phase, A is able to contact only one current replica, $x3$. It therefore needs one more current replica to reach the safety threshold. So, when constructing the new good replica list in step 3, A puts there $x3$ (it reported the maximum version number and stale-data flag "current"), and adds a replica from $x3$'s good replica list, say, $x4$. At commit time, A performs on $x1$, $x2$, $x3$, and $x4$. It marks $x1$ and $x2$ stale and applies the update to $x3$ and $x4$, getting around the permission lock on $x4$ and rendering incorrect the response that B obtained from $x4$ (Figure 7.2). After A is finished, B obtains permission from $x3$ and proceeds with the operation. The response from $x3$ carries the version number 2, which is the maximum version number that B has collected. The rest of the algorithm depends only on the response from $x3$, which makes irrelevant the fact that some other responses are incorrect. In fact, B will apply the update to $x4$ because it is in the good replica list of $x3$, even though the $x4$'s response that B has still carries version number 1. □

The following two theorems deal with tolerance to simultaneous node failures and the effectiveness of the propagation algorithm. Given Lemma 20 and 21, they can be proved by simple induction on the sequential number of write operations.

Theorem 6 *Let t be the safety threshold. After any successful write operation, there are at least t current replicas in the system, and there are at least t replicas in this write's good replica list.* □

Theorem 7 *Any node running the propagation protocol makes progress whenever it successfully obtains a new state from any member of its good replica list. Formally, let $v(a), s(a)$, and $G(a)$ be the version number, the stale-data flag, and the good replica list, respectively, of node a. Then, for any node a, if $b \in G(a)$, then either $v(a) < v(b)$ or $v(a) = v(b)$ and $s(b) =$ "current".* □

Finally, the proof of Lemma 21 showed that members of the current good replica list will always be found current during step 5 of the write operation. Moreover, by Theorem 6, this list contains at least t replicas. Hence, in step 3, the coordinator always finds t replicas to put into the new good replica list, and, if these replicas are operational, they will be found current in step 5. Therefore, in the absence of failures, our protocol never requires more than two rounds of message exchange before committing a write.

7.3 Performance

In this section, we compare the performance of our protocol with three variations of Gifford's protocol that can support partial writes.

In the first variation, the one actually described by Gifford in [18] and denoted G1 below, all writes to a given data item normally perform on the same quorum of replicas. When a write fails to obtain permission from a quorum of current replicas (a current quorum for short), it sends the permission requests to all remaining replicas. The operation proceeds as soon as a current quorum of permissions is obtained. If the operation fails to obtain permission from a current quorum even after requesting

all replicas, the necessary number of stale replicas are synchronously brought up-to-date to conclude a current quorum. After completion of the operation, the coordinator remembers the current quorum collected and uses it for future operations.

The second variation, denoted G2, is similar to G1. The only difference is that G2 attempts to reduce the risk of synchronous update propagation by propagating updates asynchronously after every write, from the replicas on which the write was performed to the remaining replicas. As in G1, all writes to a given data item normally perform on the same quorum of replicas. (Otherwise, if a write occurs before the previous propagation completes, the quorum collected may not contain all current replicas. Then, the write would either have to wait for the propagation to complete, or request permission from additional replicas, in both cases incurring a delay even in the absence of failures.) Risking being unfair to our protocol, we assume for simplicity that a single current replica is assigned the responsibility for propagating updates to the stale replicas, and that this "propagator" never fails in the process. We assume that the propagator offers the propagation to every stale replica only once. If a stale replica was found failed at that time, no further attempt to bring it up-to-date is made until the next write operation spawns a new propagation process.

Finally, variation G3 attempts to perform every write on all replicas of the data item. The write coordinator requests permission from all replicas. If a quorum is collected, the coordinator performs the write on all accessible replicas. This variation practically never performs synchronous propagation, since the only case where a replica could be stale is when it is recovering from a failure.

7.3.1 Message Overhead

First, consider message traffic generated by the protocols. Let N be the total number of replicas of a given data item, w be the number of nodes in write quorums,[4] and t be the safety threshold in our protocol.

In the normal case of the absence of failures, protocol G1 sends w permission requests,

[4] We limit our discussion to *fully distributed* quorums, where all write quorums are of the same size, all read quorums are of the same size, and every replica participates in an equal number of read and write quorums.

Table 7.2: Number of messages per operation in different protocols.

Protocol	G1	G2	G3	Our
Message Overhead	$O(w)$	$O(N)$	$O(N)$	$O(w)$

receives w permissions, sends w prepare messages, receives w acknowledgements, and sends w commit messages, for a total of $5w$ messages per operation. G2 generates the same messages, followed by $N - w$ propagation messages from the propagator to stale replicas, for a total of $4w + N$. G3 obviously generates $5N$ messages per operation (N in every phase above). Our protocol needs $2w$ messages to obtain a quorum of permissions (as in G1 and G2); then, in the best case, if the obtained quorum contains at least t current replicas, it generates $3w$ messages during the commit stage involving w nodes, for a total of $5w$. In the worst case, when the collected quorum contains no current replicas, the commit stage involves $w + t$ replicas and generates $3(w + t)$ messages. In addition, w replicas found stale will request and receive missing updates, generating $2w$ more messages. Overall, the total number of messages in the worst case is $2w+3(w+t)+2w = 7w+3t$. (Note that, as in G2, we could also assign the propagation job to a single good replica reducing the number of messages in the worst case to $6w+3t$.)

The message overhead in the absence of failures is summarized in Table 7.2. If we assume the grid-based quorums of [9] described in Section 2.3.2 and Chapter 4, where $w = 2\sqrt{N} - 1$, Table 7.2 shows that G1 and our protocol normally generate $O(\sqrt{N})$ messages per operation, while G2 and G3 generate $O(N)$ messages per operation. The number of messages in G2 could be reduced if the propagator is required to propagate the update to only w stale replicas. Then, the generated traffic for G2 would be $O(\sqrt{N})$, as well.

In the presence of failures, all protocols require $O(N)$ messages per operation.

7.3.2 Response Time and Utilization

To estimate the response time and server utilization for our protocol and the alternatives, we conducted a simulation study. We used a *closed* model of a computer system [40], in which there are several clients acting as coordinators of replica control and commit

protocols, and servers replicating the data. Each client issues a transaction. If the transaction aborts, the client immediately resubmits it; if the transaction commits, the client issues the next transaction after some *think time*.[5] We assume the availability of a multicast facility for the network, so that clients can issue a message to a group of servers in parallel. Network delays are considered negligible compared to disk access delays, which dominate service demands at the servers. Also, we assume that only nodes can fail, and the network is reliable.

In modeling node failures, we follow [29] and [50] in assuming that a node participating in an operation cannot repair or fail during the operation execution (i.e., operations are "instantaneous" with regard to failures and repairs). In other words, when a server is first contacted by an operation, it can be found either up or down (with the probability p of being up). If the node is found up, no subsequent messages from the same operation will find this node down. If the node is found down, the operation coordinator assumes it is going to be down for the duration of the operation and never contacts this node again.

The simulation is event-driven. Deadlocks are detected by the fact that the event queue is empty and resolved by aborting a random transaction that holds any locks. This simple way of handling deadlocks may not be the most appropriate for all the protocols considered. Since it turns out that our protocol is less prone to deadlocks than G1 and G2, we present the results both with and without the overhead caused by transactions that were aborted due to deadlocks. Results with deadlock overhead excluded can be considered as the case where an ideal zero-overhead deadlock-handling method is employed.

Simulation parameters include service demands for processing various messages, a timeout parameter after which `RPC.CallFailed` is returned to a message sender, and think time. In response to permission request, prepare, and abort messages, servers must acquire or release some locks in a main memory cache. In addition, processing a

[5]Note that this is slightly different from the model used in Chapter 6. In Chapter 6, our goal was to improve availability, and our focus was on the percent of transactions that are aborted. So, we assumed that a failed transaction is not immediately retried from the same site: if it is resubmitted, it is counted as a separate transaction. In this chapter, our protocol's goal it to improve the response time, despite the possibility of slightly *lowered* data availability. Thus, we want to take into account the availability aspect by including all retries of the transaction in the transaction response time.

permission request requires a disk access to obtain the replica state. Processing prepare and abort messages involves writing a log record. Commit processing requires a different amount of work depending on whether the replica is stale or current. (The case where stale replicas "commit" is relevant only in our protocol; we refer to this case as *stale commit* below). To process a commit message, stale replicas release their locks, update their version number and stale-data flag (we assume that version numbers and stale-data flags are updated in main memory cache most of the time), and write a log record. In addition, current replicas must install the updated data itself. (We call the commit by current replicas *current commit*).

Thus, all of the above processing, except for the current commit, requires one disk access, which dominates processing time. The current commit requires an additional disk access to install data. We assume that servers do not have a dedicated disk drive for the log file. Hence, the time to write a log is a random disk access time. Therefore, service demands for all messages except the current commit are assumed to be a uniformly distributed random variable with a mean of 1 time unit. The service demand for the current commit is assumed to be a uniformly distributed random variable with a mean of 2 time units.

The time to perform propagation generally depends on the number of missing updates to be installed. We assume that, on average, propagation takes three disk accesses on the receiving side and no accesses on the sending side (the recent updates are cached in main memory there). This assumption favors G1 over G2 and our protocol: in G1, stale replicas are likely to miss many updates, while in G2 and our protocol, stale replicas are usually only slightly behind because of asynchronous update propagation. Therefore, propagation in G1 is likely to require more disk accesses on both sides. Note that this assumption says nothing about the overall number of disk accesses due to propagation activity in the system. Rather, it concerns the number of disk accesses performed in a *single instance* of propagation involving a single stale and a single current replica.

Finally, timeout is assumed to be 10 time units, and think time of the client is an exponentially distributed random variable with a mean of 15 time units.

The simulation parameters are summarized in Table 7.3. Since we are interested only in the relative performance of the protocols in this study, we measure time in abstract

Table 7.3: Simulation parameters (partial writes protocols).

Parameter	Value
Service demand for: Permission Request Prepare Abort Stale Commit	Uniform with mean 1
Service demand for Current Commit	Uniform with mean 2
Service demand for Propagation	Uniform with mean 3 for the receiving server; 0 for the sending server
Think time	Exponential with mean 15
Timeout	10

time units.

In all experiments, unless specified otherwise, the number of replicas in the system is 7, all quorums are majority-based (i.e., any 4 replicas form a quorum), and the multiprogramming level is 3 (i.e., there are 3 clients issuing transactions). All transactions are assumed to be single-step and involve writing to the same data item. The reason for choosing this workload is that, as explained below, it is the most unfavorable to our protocol, and because it is the simplest to model.

For this study, we implemented our protocol (referred to as Our-t, where t is the safety threshold), G1 and G2. We did not feel it was necessary to implement G3, since the tradeoffs between it and our protocol are sufficiently clear.

The average response time of successful transactions (including all retries due to deadlocks and failures) versus p is shown in Figure 7.3. The figure shows a rather dramatic improvement in the response time of our protocol over G1 and G2. However, it turns out that a significant part of this improvement is due to our protocol's being less prone to deadlocks. The reason for lower likelihood of deadlocks is that in our protocol, once an operation collects a quorum of permissions, it will never be a party to deadlock. On the other hand, in G1 and G2, only collecting a *current* quorum guarantees the operation will not be deadlocked. If the quorum the operation first collected is not current, it requests permission from additional replicas. These additional replicas may

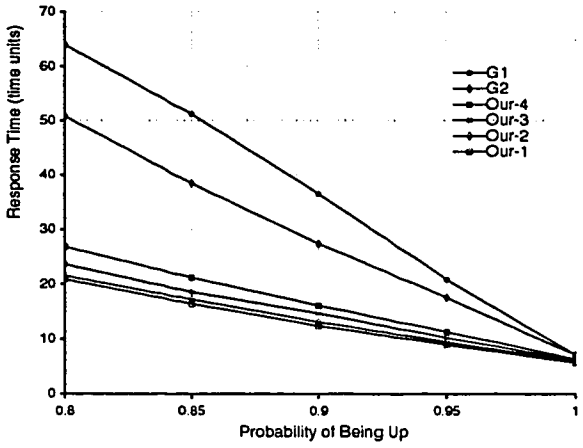


Figure 7.3: Partial writes protocols: average response time including all overhead.

well be locked by other transactions, in which case deadlock will occur.

Because many ways of handling deadlocks exist, and existing protocols benefit from lower-overhead deadlock handling more than ours, we now consider response times with deadlock overhead factored out. This is equivalent to applying an ideal deadlock-handling method with no overhead.

Figure 7.4 shows the average response time of successful transactions. which includes retries due to failures but not to deadlocks. While less dramatic than in Figure 7.3, the improvement our protocol achieves is still significant. This improvement results from a combination of the following factors. First, our protocol in most cases performs many fewer synchronous update propagations than G1 and, for the safety thresholds smaller than 4, even G2. Second, if the quorum collected in the first phase of a protocol is not current, G1 and G2 require one extra round of message exchange to obtain permission from additional replicas; our protocol in this case moves directly to the commit stage. Third, when a permission request is sent to all replicas (which happens if the coordinator failed to collect a quorum of permissions in the first phase, or, for G1 and G2, also when the collected quorum is not current), G1 and G2 wait until the *current* quorum is collected or until timeout before moving to the next stage. On the other hand, our protocol moves

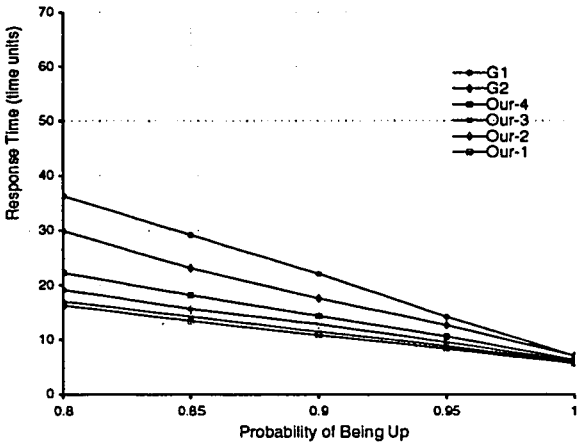


Figure 7.4: Partial writes protocols: average response time with deadlock overhead excluded.

to the next stage as soon as *any* quorum is collected. Therefore, our protocol waits for the quorum of responses that are the fastest to arrive.

It is interesting to note that our protocol performs slightly better even when all nodes are completely reliable ($p = 1$), in which case none of the above factors exist. In fact, as Figure 7.5 shows, this improvement becomes rather significant for higher multiprogramming levels. The reason for this improvement is that our protocol finds some parallelism in processing the commit stage of one transaction and the permission-collecting stage of another, as the following example illustrates.

Consider a system with 3 nodes, initially all current, and three clients. Assume each client has issued a transaction, so that there are three outstanding transactions in the system, T1, T2, and T3. Since we consider a fully reliable system, for G1 and G2, all three transactions will execute sequentially on the same quorum of replicas. Assuming that service demands are deterministic and equal to the mean values from Table 7.3, the total execution time for these transactions is $3 \cdot (1 + 1 + 2) = 12$ time units. In our protocol, each transaction chooses a random quorum. Then, the execution depicted in Figure 7.6 is possible. There, transaction T1 chooses quorum $\{1,2\}$ and succeeds in obtaining permission from both replicas. In the meantime, T2 chooses quorum $\{2,3\}$,

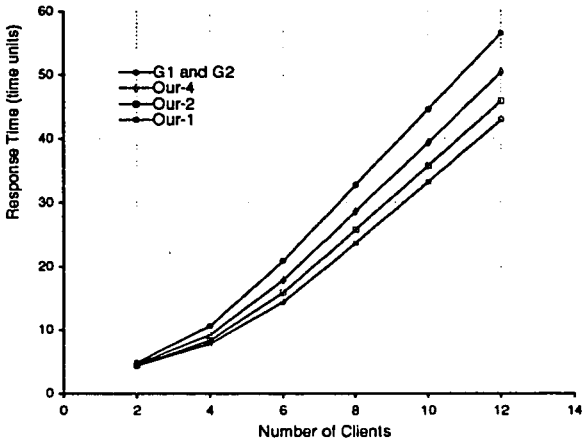


Figure 7.5: Partial writes protocols: average response time for a fully reliable system.

obtains permission from replica 3, and is blocked waiting for replica 2. T3 chooses quorum $\{1,3\}$ and is blocked waiting for both. At time 4, T1 finishes; T2 can now acquire permission from node 2, and T3 can get permission from node 1. T2 can now proceed, while T3 is still blocked waiting for node 3. T2 finds that it contacted only one current replica (node 3 became stale after T1 finished) and adds replica 1. Thus, all three replicas participate in the commit stage of T2. Note that replica 1 participates even though its permission lock was acquired by T3. When T2 sends out the commit message, nodes 1 and 2 take 2 time units to process it, while node 3 takes only 1 time unit, because it is stale. As soon as node 3 finishes with T2, it can give its permission to T3, which can now proceed. Therefore, the total time to execute the three transactions is only 11 time units. This effect is magnified when service demands are non-deterministic and increases with their dispersion.

We now turn to server utilization. Figure 7.7 shows maximum average server utilization for systems using grid quorums with different numbers of replicas of the data item. The size of a write quorum in these systems is $w = 2\sqrt{N} - 1$. Here, we assume fully reliable servers and multiprogramming level 3. Utilization counts all work done at the server, including update propagation. Figure 7.7 shows a significant improvement of our

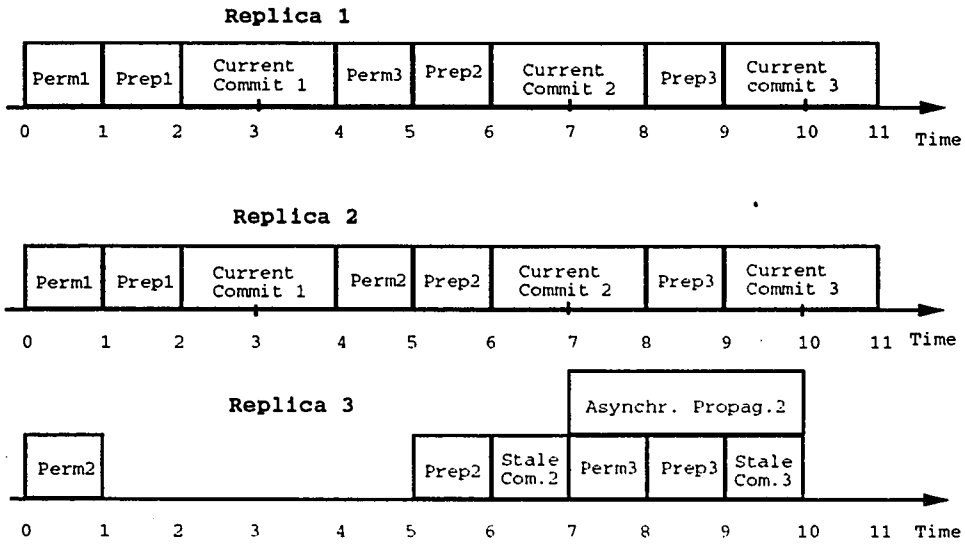


Figure 7.6: A sample execution of our partial writes protocol.

protocol over G1 and G2. This is because in G1 and G2, all operations are performed by the same servers, while in our protocol, the load is shared more evenly among all servers. As shown in [9], higher server utilization negatively affects the response time of transactions that involve multiple data items. Hence, our protocol should reduce the response time even further when the workload includes access to multiple data items.

Finally, compared to G3, our protocol has lower message overhead ($O(w)$ vs. $O(N)$). Server utilization in our protocol is also lower (the maximum server utilization in G3 should be the same as in G1 and G2, since these protocols perform all writes on the same replicas). As shown in [9], when server nodes keep multiple data items (which is usually the case in practice), systems with lower server utilization have better response time and greater overall capacity. Thus, under a moderate to heavy load, our protocol should have a performance advantage over G3. However, under a light load and in the presense of failures, G3 should outperform all other protocols, including ours, since these protocols first request permission from only a quorum of replicas; if any of them fail, the protocols have to timeout and then request permission from the remaining replicas. In contrast, G3 always requests permission from all replicas. If some of them fail, it can

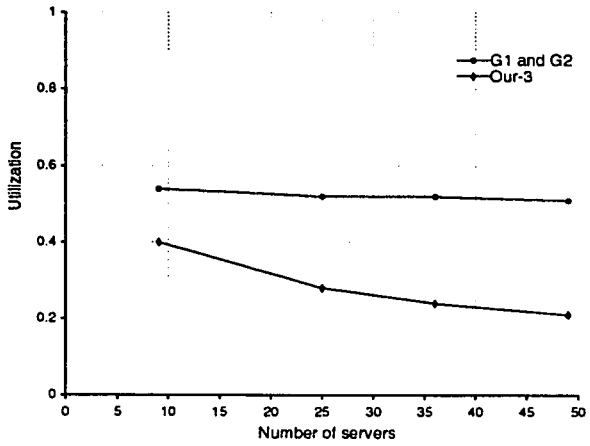


Figure 7.7: Partial writes protocols: maximum server utilization for a fully reliable system using grid quorums.

still collect a quorum in one phase.

Thus, our overall conclusion is that if message overhead is of no concern, and the load is expected to be light, consider using G3. Otherwise, our protocol is a better choice.

7.3.3 Choosing the Safety Threshold: Availability

When the safety threshold in our protocol is equal to the size of a write quorum, the protocol maintains the same number of current replicas as G1 and G2. Hence, it can tolerate the same number of arbitrary simultaneous replica failures as G1, G2, and G3.[6] As we saw in Section 7.3.2, for all values of the safety threshold, our protocol has a performance advantage over G1 and G2. However, this advantage increases when the safety threshold is smaller. It is believed that simultaneous failures are unlikely to occur, especially if a battery-backed memory is employed [29, 41]. Then, small values of the safety threshold can be chosen. To quantify this conjecture, Table 7.4 gives the fraction of aborted transactions for different values of the safety threshold, assuming independent

[6]It may appear that since G3 maintains *all* operational replicas current, it should provide better availability than the other protocols. In fact, it does not, because if a write quorum of replicas fails, the system becomes unavailable regardless of whether the remaining replicas are current or not.

Safety threshold	4	3	2	1
Percent of aborted transactions	0.00027	0.00085	0.0011	0.0021

Table 7.4: Fraction of aborted transactions for $p = 0.95$ and $N = 7$.

node failures. (The system is assumed to have 7 nodes, with probability 0.95 of a node being up.)

Note that in a protocol with majority-based quorums, t should never be chosen to be greater than the quorum size. Maintaining more than a majority of replicas current does not improve data availability (if a majority of replicas fail, the data is unavailable regardless of how many remaining replicas are current). Thus, increasing t beyond the quorum size only increases overhead with no effect on availability.

7.4 Summary

In this chapter, we draw attention to write semantics as an important factor in the performance of replicated systems. We describe a protocol based on good replica lists that supports partial writes efficiently. Compared to three variations of Gifford's protocol also capable of supporting partial writes, our protocol provides better response time than two of them, lower message overhead than the third, and better load sharing than any of the three.

An important observation that made our approach viable is that if one wants to maintain at least t current replicas at any time to guard against simultaneous node failures, updates can be applied to additional replicas without obtaining their permission first. This is true regardless of whether these replicas already gave their permission to other operations, even if their earlier responses are invalidated by the current operation. This observation allowed us to cut the number of nodes to be contacted without adding more phases to the protocol. The good replica lists used in our protocol also allow very effective propagation of missing updates by providing stale replicas with a strong hint about the location of current replicas.

Our approach blends well with other ideas in replica control, such as dynamically-adjusted quorums and protocols with witnesses. Dynamic quorum schemes, like those described in Chapters 4 and 5, are orthogonal to this protocol and can be combined with it directly. Witnesses can be incorporated into our protocol simply by excluding them from the initial good replica lists. The lists themselves, as part of the node state, should be replicated on witnesses as well as on the first-class replicas that have actual data.

Chapter 8

Fault-Tolerant Commit Protocols in Replicated Systems

Finally, we discuss how the presense of replication in a distributed system influences the design of the commit protocol. The commit protocol ensures that a transaction is consistently done (committed) or not done (aborted) across all sites and data items involved. When failures occur during the execution of a distributed commit protocol, the protocol may block in some partitions to avoid inconsistent termination of the transaction. Under strict two-phase locking, locks obtained by a transaction are released only after the transaction terminates. Thus, blocking of a commit protocol makes participating replicas unavailable for access.

On the other hand, a replica control protocol may also block access to data in a partition to prevent the possibility of replica divergence. Obviously, if the replica control and commit protocols are unaware of each other, their blocking of data accesses may have a cumulative negative effect on data availability. This chapter describes how a commit protocol may take into account the way replicas are managed to avoid such cumulative blocking.

The most common commit protocol is two-phase commit, outlined in Section 1.3.1. However, this protocol is not fault-tolerant: a failure or disconnection of the coordinator when all participants are in the waiting state causes blocking of all data items involved.

It has been shown that in order to tolerate failures, a commit protocol must have three phases [65]. In the three-phase quorum-based commit protocol proposed by Skeen [64], participants in the transaction go through an additional *prepare-to-commit (PC)* state on their way to commitment. Thus, the coordinator of the three-phase commit protocol begins by sending out a request for commitment to nodes participating in the transaction. The participants enter the waiting state W and respond with an acknowledgement; after the coordinator receives acknowledgements from all participants that they entered the waiting state, the coordinator sends a *prepare-to-commit* message, causing the participants to enter the PC state. Only after a (possibly weighted) majority of participants has acknowledged entering this state will the coordinator send out the commit message. The recipients of this message enter the *commit* state C, thereby committing the transaction locally.

The purpose of the prepare-to-commit state is to de-couple the waiting state and the commit state, which a participant enters upon committing the transaction locally. Then, it is guaranteed that, if there is a participant in the waiting state, no one could have entered the commit state. Likewise, if there is a participant in the PC state, no one could have aborted. Then, if the system partitions during the execution of the protocol, or the coordinator fails, nodes in a partition can attempt to terminate the transaction by counting the number of nodes in various states. Obviously, if a partition contains a node in the commit or abort state, everyone else can terminate the transaction accordingly. In addition, if some participant has been found to have not yet entered the waiting state, then the coordinator has not completed even the first phase of the protocol, and the transaction can be aborted.

Otherwise, if the partition contains a majority of nodes in the waiting or PC state, the transaction can be committed in this partition provided there is at least one participant in the PC state, ensuring that no participant outside the partition could have aborted. (Nodes in the waiting state still need to move to the PC state first.) If the partition contains a majority of nodes in the waiting state and none in the PC state, the transaction can be aborted in this partition. (For a reason that will become clear later, participants must move from the waiting state to a special *prepare-to-abort* state before aborting the transaction.)

Thus, in the quorum-based commit protocol, if the network partitions while all the participants are in intermediate states, the transaction is terminated in the partition that contains a majority of participating nodes. Nodes in other partitions will remain blocked until the partition heals.

It has been shown that a commit protocol that *never* blocks under communication failures does not exist [65]. Given this fact, the goal of fault-tolerant commit protocols is to reduce the negative effect on data availability caused by failures. In this chapter, we describe a commit protocol that incorporates two independent ideas to improve data availability in replicated systems [55].

First, our protocol is based on a two-level weighted voting scheme. When failures occur during the execution of the commit protocol, two-level elections are held to choose partitions where the transaction in progress can be terminated. Nodes with a replica of a given data item form an electoral district. Each district holds local elections, according to the replica control protocol used for maintaining consistency of the corresponding data item. Specifically, a partition wins in district-level elections if it contains enough replicas of the corresponding data item to keep it available under the replica control protocol used. During upper-level elections, the voting entities are districts. Each district has a number of votes equal to some weight pre-assigned to the corresponding data item. A partition receives all votes of the districts in which it won local elections, and none from the districts where it lost. The transaction can be committed (aborted) in a partition that obtained a minimum number of upper-level votes, called a *commit quorum* V_c *(abort quorum* V_a*)*. The sum of the commit and abort quorums must exceed the total number of votes, which ensures that the transaction will not terminate inconsistently in different partitions.

The main advantage of two-level voting is that when choosing a partition for terminating the transaction, the protocol counts data items available *under the replica control mechanism used in the system*, rather than nodes. In particular, if all data items have one vote, and a majority of the total number of votes must be collected in upper-level elections for both committing and aborting a transaction, then the partition that wins the elections is the one that keeps a majority of data items available. In a non-replicated system, districts consist of just one node and local elections become trivial. Thus, in

both replicated and non-replicated cases, a partition in which a majority of data items is available is chosen for terminating the transaction. Besides being a sensible strategy from the data availability standpoint, this property specifies a well-defined and easily understood behavior of the system under failures.

As an example, consider a transaction that updates data items x, y, z, replicated on sites x_1, x_2, x_3; y_1, y_2, y_3; and z_1, z_2, z_3 respectively. Let commit and abort quorums both be equal to a majority of nodes. Assume the replica control protocol uses majority quorums. Assume that the system partitions into $A = \{x_1, x_2, x_3, y_1, z_1\}$ and $B = \{y_2, y_3, z_2, z_3\}$. Skeen's protocol will terminate the transaction in partition A and block the transaction in partition B. However, the replica control protocol makes only one data item x available in partition A, as opposed to two available data items in partition B. Hence, data availability for further accesses would be improved if the transaction would terminate in partition B. In contrast, our protocol counts the number of data items available in a partition, not the number of nodes. In our example, partition B wins in two local elections (districts y and z). Hence, this is where the transaction terminates, even though A's total number of votes is greater.

Another advantage is that, if certain data items are known to be heavily used or otherwise especially important, our protocol can give preference to keeping these items available by assigning them multiple upper-level votes (heavier weight). Finally, our scheme can work with a wide variety of replica control protocols. In fact, it can work when different replica control protocols are used for different data items. Continuing our analogy, each district can independently establish its electoral law. The party winning in a majority of districts (by the local laws in each case) wins the general election. Thus, a system designer has the flexibility to choose a replica control protocol best suited to each individual data item.

With our second idea, we propose a way to reduce the harmful effect of accumulating network failures on data availability. In distributed commit protocols, a transaction can be committed only if the coordinator of the protocol (usually the node that initiated the transaction) receives permission for commitment from all participating sites. Upon issuing permission for commitment, a participant enters a chain of vulnerable states in which it relies on instructions from the coordinator to make further progress. Should it

become partitioned from the coordinator, blocking may occur. In existing protocols, the likelihood of blocking increases if the system has already been fragmented due to prior failures: in this case, partitions will tend to contain fewer nodes and be less likely to win the elections necessary for transaction termination.

To address this problem, we observe that if a set of nodes involved in a transaction is disjoint at the time the commit protocol starts, then with high probability the coordinator will not receive permissions for commitment from all sites. Hence, the transaction will not commit. Therefore, if the individual sites could tell with high probability that the set of participants is disjoint, there would be no reason for them to issue permission for commitment. Instead, they could abort immediately and send a demand for aborting to the coordinator. This way, they avoid being in a state where blocking is possible.

In our scheme, every node in the system maintains a periodically refreshed hint about its partition, called a *view*.[1] The commit coordinator piggybacks the list of participants with the request for commitment. If a participant discovers that some members of this list are outside its view, it aborts the transaction locally and sends a demand for aborting to the coordinator. This causes all other participants who are in the same partition or can communicate with the coordinator to abort as well, freeing their data items for future accesses. Although this idea has been developed in conjunction with the protocol proposed here, it is independent of replication and can be used in non-replicated distributed systems as well.

8.1 The Protocol

In this chapter, we reserve the word "participant" for nodes participating in the transaction. Following [63], we describe commit protocols using finite state machine (FSM) diagrams, one for each participating site. The structure of the state diagrams of our protocol is similar to that of existing protocols [64, 28]; the novelty is in the transition functions.

Below, a set of nodes each storing a replica of a data item x is called a *district* x. A subset of a district x is *commit-sufficient* if it includes a write quorum of replicas of x;

[1] Views were originally introduced in [14] in the context of replica control.

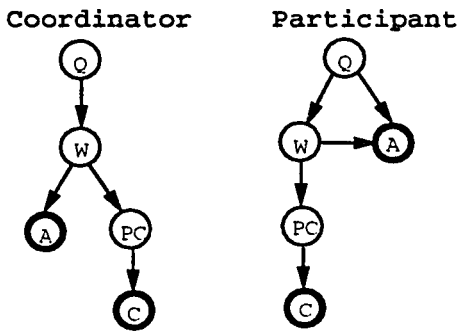


Figure 8.1: A state diagram for the commit protocol (normal operation).

it is *abort-sufficient* if it includes a write *or* read quorum of such replicas.[2] Each district (data item) is assigned a (positive) weight. Two numbers, commit quorum V_c and abort quorum V_a, are associated with every transaction, such that $V_c + V_a > V$, where V is the total weight of all data items participating in this transaction.

We assume in this chapter that the network is symmetrical (if p can talk to s then s can talk to p) and transitive (if p can talk to s and s can talk to t, then p can talk to t). We also assume that strict two-phase locking is used for concurrency control. While not needed for correctness, these assumptions motivate our design choices.

8.1.1 Normal Operation

In this section, we describe the behavior of the protocol when no additional failures occur during protocol execution. As in [28], we assume that a replica control protocol is responsible for choosing the replicas on which to perform operations that comprise the transaction. At commit time, the whole transaction should be aborted if any participating site demands to abort.

Our protocol is based on the three-phase commit protocol [63] and is shown in Figure 8.1, with the transition function given in Table 8.1. Each node maintains a periodically refreshed list of all nodes in the system with which it can communicate. Following [14], we call this list a node's *view*.

[2]The dual and equally possible convention is to consider a subset commit-sufficient if it includes a write or read quorum of replicas, and abort-sufficient if it includes a write quorum of replicas.

Table 8.1: Transition function for the commit protocol (normal operation).

State	Messages Received	Action
	COORDINATOR	
Q	Start protocol.	Enter state W; broadcast request for commitment to all participants, along with participation set PS.
W	**permission-to-commit** received from all participants.	Enter state PC; broadcast **prepare-to-commit** message to all participants.
	demand-to-abort received from some participants (inaccessible sites considered to respond with demand to abort).	Enter state A; broadcast **abort** message to all participants; stop.
PC	**PC-ack**'s received from all nodes from commit-sufficient subsets of districts with total weight at least V_c.	Enter state C; broadcast **commit** message to all participants; stop.
	PARTICIPANTS	

Q	Request for commitment together with participation set PS received.	If PS $\not\subseteq$ VIEW, where VIEW is the node's view, then enter state A; send **demand-to-abort** to coordinator; stop.
		else
		if transaction can be committed locally, enter state W; send **permission-to-commit** to coordinator.
		else enter state A; send **demand-to-abort** to coordinator; stop.
		endif
		endif

W	**abort** message received.	Enter state A; abort the transaction; stop.
	commit message received.	Enter state C; commit the transaction; stop.
	prepare-to-commit message received.	Enter PC state; send **PC-ack** to coordinator.
PC	**commit** message received.	Enter state C; commit the transaction; stop.

During the first phase, the commit coordinator broadcasts the request for commitment to all nodes participating in the transaction. With this request, the coordinator piggybacks a *participation set*, which is the list of all participating sites.

Upon receiving the request for commitment, a participant first checks if any members of the participation set received are outside its view. If this is the case, the participant immediately aborts the transaction (even if there are no local reasons for this) and sends a `demand-to-abort` message back to the coordinator. Otherwise, the participant either sends its permission for committing to the coordinator and enters state `W` where it waits to be informed by the coordinator about the global decision on how to terminate the transaction, or it responds with the demand for aborting and aborts the transaction without waiting.

The rest of the protocol under normal operation is the same as the three-phase commit protocols of [63, 64]. After receiving all responses (timeouts being considered as demands for aborting), the coordinator initiates the second phase by broadcasting a `prepare-to-commit` message if all participants permitted the commitment, or an `abort` message otherwise. On receiving an `abort` message, a participant aborts the transaction, completing the protocol. On receiving a `prepare-to-commit` message, a participant enters a `prepare-to-commit` state (`PC`) and responds with acknowledgement.

The third phase starts when the coordinator receives acknowledgements from all participants from commit-sufficient subsets of a commit quorum of districts. In the third phase, the coordinator broadcasts a `commit` message to all participants, causing them to commit the transaction.

Consider the rationale behind this modification of the three-phase commit protocol. We make participants maintain a view and demand aborting a transaction if any participants are outside the view. Alternatively, the coordinator itself could maintain its view and abort the transaction without even sending a request for commitment if any participants are outside its view. However, there are many more potential coordinators (which are client nodes) than participants (which are server nodes). This alternative therefore would result in heavy background message traffic and server load for maintaining views.

A participant's view is a hint on the partition to which the participant belongs. So, if any members of the participation set do not belong to the node's view, the node infers

that, with high probability, the set of sites involved in the transaction is disjoint. Then, by the assumption of the transitivity of the network, the coordinator will not receive permission for commitment from all participants and will abort the transaction. Hence, there is no harm if the participant aborts the transaction immediately. In our protocol, the participant does just that, freeing its data item for future accesses. Given the same situation in the existing protocols, the participant could well decide to issue permission for commitment and enter the waiting state. If additional failures occur during execution of the protocol, voting will be initiated to find partitions where the transaction could be safely terminated. However, because the network has already been fragmented, the chances that any partition will have enough operational nodes to win the elections will be slim.

Let Δt be the maximum time needed for all nodes in the system to notice a change in network topology and update their views to reflect the change [14]. Then, if a set of participants in the transaction was disjoint at Δt before the commit protocol begins, and stays disjoint through the first phase of the protocol, then all participants will correctly identify the participation set as disjoint and abort the transaction. Hence, no blocking will take place at any site, even if additional node and communication failures occur during execution of the protocol.

If a hint about network topology is optimistically incorrect at some sites, i.e., it identifies the participation set as connected when in fact it is disjoint (this may happen if the network partitions within Δt before the protocol starts), then our protocol may show at worst the same blocking behavior as a similar protocol that does not use views. Indeed, a protocol without views can be considered as a protocol with views that contain all nodes in the system (i.e., views are always optimistic).

Finally, if the communication failure that caused fragmentation was repaired just before the protocol starts (within Δt), the views of some participants may be pessimistically incorrect, i.e., they may identify the participation set as disjoint when in fact it is connected. In this case, the transaction will be unnecessarily aborted. Because changes in network topology are infrequent compared to the transaction rate, this is unlikely to happen. But even if it does, it is a small price to pay for avoiding the harmful effect of network fragmentation on blocking. The consequence of aborting a transaction is

that it will have to be repeated, while the consequence of blocking is that data become unavailable.

Why could not the coordinators themselves maintain their views and abort transactions immediately if any participants are outside their views?

8.1.2 Operation under Failures (Termination Protocol)

A node concludes that a failure occurred if an expected message from another node does not arrive for a sufficiently long time and a subsequent ping determines that this other node is not accessible. The only exception is the coordinator in state W, which interprets the missing messages as demands for abortion. If a node detects a failure during the commit protocol described in the previous subsection, it initiates a *termination protocol* to either commit or abort the transaction consistently and to unblock the participating data items for future accesses. Note that each partition runs a separate instance of the termination protocol.

In the initial stage of the termination protocol, a termination coordinator is elected. Any existing election algorithm is suitable (see, for example, [21]). Once the termination coordinator is chosen, it runs the protocol shown in Figure 8.2 and Table 8.2. First, it polls participants in the transaction about their states. If any node is in the commit state (C), the transaction is immediately committed, no matter how many nodes are in the partition. (In this case, the transaction can only be either blocked or committed in all partitions, so consistency is not violated.) Similarly, the transaction is immediately aborted if a node in the initial (Q) or abort (A) state is found.

If the partition does not contain sites in states A, C, or Q, the termination coordinator will count votes to decide how to terminate a transaction. It is possible to commit if: (1) a partition contains at least one node in the PC state, and (2) there are districts with a total weight of at least V_c, such that each is represented in the partition by a commit-sufficient subset of nodes in states PC or W. It is possible to abort if there are districts with a total weight of at least V_a, such that each is represented by an abort-sufficient subset of nodes in states PA or W. Note that while it is quite possible that some partition satisfies the conditions for both committing and aborting, a situation where

Table 8.2: Transition function for the termination protocol.

State	Messages Received	Action
	TERMINATION COORDINATOR	
P1	Start protocol.	Enter state P2; request local states from all participants.
P2	At least one participant reported state C.	Broadcast **commit** message to all participants; stop.
	At least one participant reported state A or Q.	Broadcast **abort** message to all participants; stop.
	None of the above combinations of messages received, and at least one participant reported state PC, and all nodes from commit-sufficient subsets of districts with a total weight of at least V_c reported states PC or W.	Enter state P3; broadcast **prepare-to-commit** message to all participants.
	None of the above combinations of messages received, and all nodes from abort-sufficient subsets of districts with a total weight of at least V_a reported states PA or W.	Enter state P3; broadcast **prepare-to-abort** message to all participants.
P3	**PC-ack**'s received from all nodes from commit-sufficient subsets of districts with a total weight of at least V_c.	Broadcast **commit** message to all participants; stop.
	PA-ack's received from all nodes from abort-sufficient subsets of districts with a total weight of at least V_a.	Broadcast **abort** message to all participants; stop.
	PARTICIPANTS	
Q, W, PC, PA, C, A	Request for a local state.	Stop executing the normal commit protocol; report the local state to the coordinator.
W, PC, PA	**commit**	Enter C state; commit transaction; stop.
Q, W, PC, PA	**abort**	Enter A state; abort transaction; stop.
W	**prepare-to-commit**	Enter PC state; send **PC-ack** to the coordinator.
	prepare-to-abort	Enter PA state; send **PA-ack** to the coordinator.
PC	**prepare-to-commit**	Send **PC-ack** to the coordinator.
PA	**prepare-to-abort**	Send **PA-ack** to the coordinator.

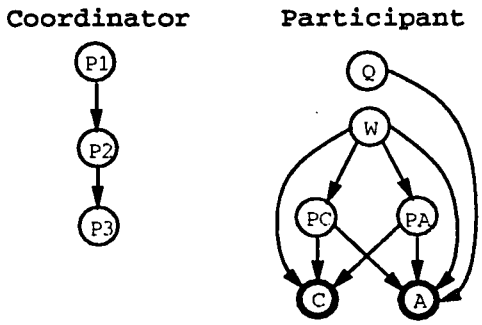


Figure 8.2: A state diagram for the termination protocol.

the condition for committing is satisfied in one partition and the condition for aborting is satisfied in another cannot occur.

If committing is possible, the termination coordinator moves the participants in state W into state PC and then commits the transaction. Moving the sites into the PC state prior to committing is necessary to ensure that, should additional failures cause the termination process to be restarted from scratch, no partition will ever satisfy the condition for aborting. Similarly, if aborting is possible, the termination coordinator moves the participants in state W into state PA and then aborts the transaction. If both committing and aborting are possible, the coordinator chooses to commit. Finally, if neither committing nor aborting is possible, the transaction has to be blocked.

If additional failures are detected during the execution of the termination protocol, then a new termination coordinator is elected, and the same protocol is restarted.

8.2 Proof of Correctness

We now prove that the proposed protocol is correct. From now on, we will refer to the protocol in Table 8.1 as the normal commit protocol and to the protocol in Table 8.2 as the termination protocol. We will say that a commit quorum is established in the system if all nodes from commit-sufficient subsets of the districts with a total weight of at least V_c have visited state PC during the execution of the normal commit and termination protocols. Similarly, we will say that an abort quorum is established if all nodes from

abort-sufficient subsets of districts with a total weight of at least V_a have visited state PA.

Lemma 22 *Once a commit quorum is established in the system, an abort quorum can never be established, and vice versa.*

Proof. Assume that a commit quorum is established, and let S_c be the set of nodes in the system that have visited state PC. By definition, S_c includes all nodes from commit-sufficient subsets of districts with a total weight of V_c. Let S_a be any set of nodes that must visit state PA for the abort quorum to be established. By definition, any such set must include all nodes from abort-sufficient subsets of districts with a total weight of V_a. Since $V_c + V_a > V$, where V is the total weight of all districts, there exists a district x such that S_c includes a commit-sufficient subset of x and S_a includes an abort-sufficient subset of x. Due to the intersection property of read and write quorums, any commit-sufficient subset and abort-sufficient subset of the same district have at least one node in common. This common node belongs to both S_c and S_a. Since this node belongs to S_c, it has visited state PC. Hence, because state PA is not reachable from state PC, this node can never visit state PA. Therefore, not all nodes from S_a can ever visit state PA. Because S_a was assumed to be any set of nodes that must visit state PA for an abort quorum to be established, we conclude that establishing an abort quorum is impossible. A symmetric argument can be used to show that the dual claim of the lemma is also correct. □

Lemma 23 *If any participant has visited state* PC, *then there is a participant that was moved to state* PC *during execution of the normal commit protocol.*

Proof. The proof by contradiction is trivial, given the fact that in order to move any node to state PC, the termination protocol must find at least one node that has already been moved to this state. □

Theorem 8 *The proposed commit and termination protocols will terminate a transaction consistently (if at all) in the presence of any number of fail-stop node and communication failures.*

Proof. First, assume that some participant terminates the transaction by committing. We will show that no participant can ever abort. Since no site can commit before a commit quorum is established, a commit quorum has been established in the system. In particular, there are nodes that have visited state PC. Hence, by Lemma 23, there is a node that was moved to state PC by the normal commit protocol. Before the normal commit coordinator moves any node into state PC, it makes sure that all participants have left the initial state Q and that no participant can ever abort during execution of the normal commit protocol. Hence, all we need to show to prove that consistency is not violated in this case is that no participant can abort during execution of the termination protocol. Indeed, before the first site can abort via the termination protocol, one of the following conditions must be satisfied. (a) There is a partition containing participants in states Q or A. Since we are considering the prerequisites necessary for the *first* participant to abort, this condition translates into the requirement that there be a partition containing a participant in state Q. But this is impossible, because all participants have left state Q, and this state is not reachable from any other state. (b) An abort quorum is established. This is also impossible, due to Lemma 22.

Now assume that there is a participant that terminates by aborting during the execution of the normal commit protocol. The normal commit coordinator issues prepare-to-commit message only after making sure that no participant ever aborts during the execution of the normal commit protocol. Therefore, the above assumption means that no participant will ever be moved into state PC by the normal commit protocol. Hence, by Lemma 23, no participant will ever move to state PC, and a commit quorum will never be established. Since establishing a commit quorum is necessary before any node can commit, no participant will ever commit.

Finally, consider the case where no participant aborts during the execution of the normal commit protocol, but there is a participant aborted by the termination protocol. Let x be the first such participant.[3] We must show that no participant can ever commit. Indeed, x could be aborted by the termination protocol only if there is a participant in state Q or A, or if an abort quorum is established. In the former case, because we are considering the *first* participant to abort, the condition reduces to the requirement that

[3] More precisely, x is a node such that no other node aborts before x does.

there is a participant in state Q. Then, since no node is moved to state PC by the normal commit protocol unless all participants leave state Q, no node moves to state PC during execution of the normal commit protocol. As we showed in the previous paragraph, this means that no participant can ever commit. In the latter case, due to Lemma 22, establishing a commit quorum is impossible, and no participant will ever commit.

In all cases, we conclude that the proposed protocol terminates transactions consistently. □

8.3 A Refinement of the Protocol

The protocol as described in Section 8.1 works best only when transactions write to all their data items. Otherwise, as the following example illustrates, a transaction may be aborted even if all participating nodes agree to commit. Consider a system with three data items, x, y, z, replicated on three nodes each, $x1$, $x2$, $x3$, $y1$, $y2$, $y3$, $z1$, $z2$, $z3$. Assume that read-one-write-all quorums are employed for replica control. Consider a transaction T that reads x and y and writes z. Assume that $V_c = V_a = 2$. Then, even if all replicas reported PC-ack to the normal commit coordinator, the transaction cannot be committed, because these responses include a commit-sufficient subset only from district z.

However, the following simple refinement eliminates this problem. The coordinator of the normal commit protocol from Table 8.1 can broadcast the commit message as soon as there are districts involved in the transaction with a total weight of at least V_c, such that all nodes from commit-sufficient subsets of these districts either responded with PC-ack, or did not participate in the transaction.

Similarly, the termination coordinator can broadcast a prepare-to-commit message as soon as there are districts involved in the transaction with a total weight of at least V_c, such that all nodes from commit-sufficient subsets of these districts either reported state PC or did not participate in the transaction.

Finally, the termination coordinator does not have to wait for PC-ack's from all nodes from commit-sufficient subsets of districts with a total weight of V_c before broadcasting the commit message. All it must ensure is that the nodes that sent PC-ack together with

the replicas that do not participate in the transaction include commit-sufficient subsets of above districts.

Returning to our example, when all nodes have sent a PC-ack to the normal commit coordinator, districts x and y have commit-sufficient subsets that do not participate in the transaction, and district z has a commit-sufficient subset that responded with PC-acks. The total number of these districts, three, exceeds V_c. Hence, the transaction is committed.

No modification to messages exchanged in the protocol is required to implement this refinement. Indeed, only the participation set of a transaction is needed to tell whether a commit-sufficient subset of a district is involved in the transaction. The normal commit coordinator knows the participation set when starting the protocol. When the termination coordinator is elected, it either has received the participation set with a request for commitment, or is still in the initial state, in which case the transaction is aborted.

The proof of correctness from the previous section still holds for the refined protocol. The only difference is that we will now say that a commit quorum is established if all nodes from commit-sufficient subsets of districts with a total weight of at least V_c either have visited state PC or do not participate in the transaction.

8.4 Setting Parameters of the Protocol

In this section, we discuss how to choose parameters of the protocol. Weights for data items should be assigned depending on the importance of the individual data item's availability. The more important it is to have a data item available, the more weight it is assigned. The commit and abort quorums are chosen depending on how much time the nodes spend in states W and PC during execution of the normal commit protocol [64]. Should a failure occur, the more time the nodes spend in state W (as compared to state PC), the more likely the termination coordinator will find all participants in the partition to be in state W, in which case the transaction can only be aborted or blocked. Hence, the easier it should be to abort, which means a smaller abort quorum (and larger commit quorum) must be chosen.

A node in state W must log the update and state information (a site must keep its state in stable storage for recovery). In modern systems, this can usually be done in one disk operation. A node in state PC must log only its state. This also requires one disk operation. In most cases, the time to execute both operations will be roughly the same. Then, assuming that communication delays are negligible compared to disk operations, the termination coordinator will equally likely find all participants in state W or PC. Hence, the most likely choice is to make commit and abort quorums equal. However, these parameters can be fine-tuned to match specific performance characteristics of the system.

8.5 Related Work

In [28], two dual protocols were proposed that are extreme particular cases of our two-level voting scheme. Our scheme reduces to these protocols if one requires that a partition win in all districts to commit, and in at least one district to abort (or vice versa). However, these protocols lack the flexibility provided by our approach. In particular, if certain data items are known to be especially important, our protocol can be tuned to give preference to keeping them available. Also, by varying the relative sizes of commit and abort quorums, we can tune the protocol to reflect specific performance and failure characteristics of the system. In fact, as discussed in Section 8.4, in modern systems, the most likely choice is to make abort and commit quorums equal, which is the farthest deviation from both dual protocols of [28].

8.6 Summary

In this chapter, we described a fault-tolerant distributed commit protocol that incorporates two new ideas. First, when choosing a partition in which to terminate a transaction, we propose a way to count the number of data items available in the partition under the replica control protocol used in the system, as opposed to counting individual nodes. In replicated systems, the partition that contains the most nodes is not necessarily the one that contains the most available data items. Thus, choosing the partition with the

most available data items reduces the extent of blocking in the system. In non-replicated systems, our scheme converges with the pre-existing one. Also, by associating weights with data items rather than with individual nodes, our scheme allows for a very straightforward interpretation of weights: they are the measure of the comparative importance of maintaining various data items unblocked (i.e., available for accesses).

Second, regardless of whether there is replication in the system, we propose a way to significantly reduce the harmful effect of network fragmentation on blocking. In essence, we eliminate the effect of fragmentation accumulated before the protocol begins. The price for achieving this is that transactions may be unnecessarily aborted if the fragmentation has been repaired just before the commit protocol starts. Because the rate of failures and repairs is much lower that the transaction rate, such unnecessary aborting is unlikely. But even if it happens, it is a small price to pay to avoid blocking: the consequence of aborting a transaction is that it will have to be repeated, while the consequence of blocking is that data become unavailable.

Chapter 9

Conclusions

In a large wide-area distributed system, some of the components are going to be non-operational or disconnected most of the time. In addition, latency of access to remote components, as well as overloading of services, can seriously degrade performance. Addressing these issues is critical for distributed systems to scale well. A common approach is to replicate data and services.

This thesis concentrates on the quorum-based approach to replication management. Numerous ways to define quorums have been proposed. The choice of a particular type of quorum depends on application requirements, performance and reliability parameters of system components, etc. In other words, different quorums work best in different circumstances. Therefore, this thesis addresses the following general question: given a quorum definition, how can we manage replication efficiently?

A wide variety of quorum types are considered. Chapters 3 and 4 address the problems of structure-based quorums. Chapter 5 is devoted to voting-based quorums. Chapter 6 concerns read-one-write-all (ROWA) quorums. Finally, Chapters 7 and 8 discuss general issues relevant to all types of quorums. The protocols proposed in these chapters place slightly different restrictions on the environment. Table 9.1 summarizes the underlying assumptions of these protocols. The main contributions of the thesis are:

- We show that it is possible to provide high data availability using structure-based quorums. These quorums are attractive because they result in low-overhead replica

Table 9.1: Summary of the protocols proposed in the thesis.

Protocol	Chapter	Concurrency Control	Commit Protocol	Epoch Management
Dynamic Structure-based Protocol	4	Any	Any	Per data item
Asynchronous System Reconfiguration	5	Strict 2-phase locking	Any	Per data item
Fault-tolerant ROWA Protocol	6	Any	Any	System-wide
Partial Writes Protocol	7	Any	Any except "Early Prepare"	-
Commit in the Presence of Replication	8	Strict 2-phase locking	3-phase commit	-

management when the number of replicas is high. (Chapters 3 and 4.)

- We show that is possible to include new/repaired replicas in the system and exclude failed/disconnected/decommissioned replicas without any service interruption. Thus, system reconfiguration can be done more freely for various purposes (e.g., improving data availability, migrating data, redistributing load). (Chapter 5.)

- We show that it is possible to provide fault-tolerance to updates in the read-one-write-all (ROWA) replication scheme without compromising the high efficiency and availability of read operations, even in a wide-area, partition-prone network. The ROWA scheme has the potential to be widely used, since many services and significant amount of data in a distributed system fall under the mostly read category. By removing the main limitation of the ROWA scheme, this result should allow the potential of the ROWA scheme to be realized. (Chapter 6.)

- We call attention to the fact that the semantics of write operations can significantly affect the performance of a replicated system. Thus, replica control protocols must often be designed for particular semantics of writes exhibited in the system.

(Chapter 7.)

- We propose a method for atomic commitment in a replicated system that improves data availability by minimizing the blocking of data in the event of network partitioning. (Chapter 8.)

In summary, this dissertation contains a number of proposals for improving performance and data availability in replicated distributed systems that require strong consistency. It addresses a broad range of problems, from theoretical to very practical ones that promise immediate benefits to current systems. Thus, we hope that this dissertation will be helpful in building more efficient and fault-tolerant replicated distributed systems and will contribute to a better understanding of the properties and limitations of these systems in general.

Bibliography

[1] D. Agrawal and A. El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. on Comp. Sys.*, 9(1), pp. 1-20, February 1991.

[2] D. Agrawal and A. J. Bernstein. A non-blocking quorum consensus protocol for replicated data. *IEEE Trans. on Parallel and Distr. Sys.*, 2(2), pp. 171-179, April 1991.

[3] D. Agrawal and A. El Abbadi. Resilient Logical Structures for Efficient Management of Replicated Data. In *Proc. of the 18th Int. Conf. on Very Large Data Bases*, pp. 151-162, 1992.

[4] D. Barbara, H. Garcia-Molina, and A. Spauster. Increasing availability under mutual exclusion constraints with dynamic vote reassignment. *ACM Trans. on Comp. Sys.*, 7(4), pp. 394-426, November 1989.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, Reading, Mass., 1987.

[6] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. on Computer Systems*, 2(1), pp. 39-59, February 1984.

[7] D. D. Chamberlin and F. B. Schmuck. Dynamic data distribution (D^3) in a shared-nothing multiprocessor data store. In *Proc. of the 18th Int. Conf. on Very Large Data Bases*, pp. 163-174, 1992.

[8] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The Grid protocol: a high performance scheme for maintaining replicated data. In *Proc. of the IEEE 6th Int. Conf. on Data Engineering*, pp. 438-445, 1990.

[9] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The Grid protocol: a high performance scheme for maintaining replicated data. *IEEE Trans. on Knowledge and Data Eng.*, 6(4) pp. 582-592, December 1992.

[10] F. Cristian, B. Dancey, and J. Dehn. High availability in the advanced automation system. In *20th Int. Symp. on Fault-tolerant Computing*, invited paper, Newcastle-upon-Tyne, UK, July 1990.

[11] S.B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3), pp. 341-370, September 1985.

[12] E. W. Dijkstra. *A Discipline of Programming.* Englewood Cliffs, NJ:Prentice-Hall, 1976.

[13] D. Eager and K. Sevcik. Achieving robustness in distributed database systems. *ACM Trans. on Database Sys.* 8(3), pp. 354-381, 1983.

[14] A. El Abbadi, D. Skeen, and F. Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proc. of the 4th ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pp. 215-229, 1985.

[15] A. El Abbadi and S. Toueg. Availability in partitioned replicated databases. In *Proc. of the 5th ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pp. 240-251, 1986.

[16] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Trans. on Database Sys.* 14(2), pp. 264-290, June 1989.

[17] A. El Abbadi and S. N. Dani. A dynamic accessibility protocol for replicated databases. *Data and Knowledge Engineering*, 6, pp. 319-332, 1991.

[18] D. K. Gifford. Weighted voting for replicated data. In *Proc. of the Seventh ACM Symposium on Operating Systems Principles*, pp. 150-159, December 1979.

[19] H. Garcia-Molina. *Performance of Update Algorithms for Replicated Data.* UMI Research Press, Ann Arbor, Michigan, 1981.

[20] H. Garcia-Molina. Reliability issues for fully replicated distributed databases. *IEEE Trans. on Computers* 15(9), pp. 34-42, September 1982.

[21] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. on Computers,* C-31(1), pp. 48-59, January 1982.

[22] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *J. of the ACM,* 32(4), pp. 841-860, October 1985.

[23] H. Garcia-Molina. The future of data replication. In *Proc. of the 5th IEEE Symp. on Reliability in Distr. Software and Database Sys.*, pp. 13-19, 1986.

[24] J. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course*, New York, Springer-Verlag, 1979.

[25] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Trans. on Database Sys.* 12(2), pp. 170-194, 1987.

[26] A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. Availability and consistency tradeoffs in the Echo distributed file system. In *Proc. of the IEEE Second Workshop on Workstation Operating Systems*, pp. 49-54, 1989.

[27] A. Hisgen, A. Birrell, C. Jerian, T. Mann, M. Schroeder, and G. Swart. Granularity and semantic level of replication in the Echo distributed file system. In *Proc. of the IEEE Workshop on Management of Replicated Data*, pp. 2-4, 1990.

[28] C.-L. Huang and V. Li. A quorum-based commit and termination protocol for distributed database systems. In *Proc. of the 4th IEEE Int. Conf. on Data Eng.*, pp. 136-143, 1988.

[29] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. on Database Sys.* 15(2), pp. 230-280, June 1990.

[30] S. Jajodia and D. Mutchler. Dynamic voting. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 227-238, 1987.

[31] S. Jajodia and D. Mutchler. Integrating static and dynamic voting protocols to enhance file availability. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pp. 144-153, 1988.

[32] S. Jajodia and D. Mutchler. A pessimistic consistency control algorithm for replicated files that achieves high availability. *IEEE Trans. on Software Engineering*, 15(1), pp. 39-45, January 1989.

[33] A. Kumar. Performance analysis of a hierarchical quorum consensus algorithm for replicated objects. In *Proc. of 10th Symp. on Distributed Computing Sys.*, pp. 378-385, IEEE, 1990.

[34] A. Kumar. Hierarchical quorum consensus: a new algorithm for managing replicated data. *IEEE Trans. on Computers*, 40(9), pp. 996-1004, September 1991.

[35] A. Kumar and K. Malik. Generalizing and optimizing hierarchical quorum consensus algorithms for replicated data. Graduate School of Management, Cornell University, Tech. Report, October 1991.

[36] A. Kumar and S. Y. Cheung. A $\sqrt{N}$ high availability hierarchical grid algorithm for replicated data. *Information Processing Letters* 40, pp. 311-316, 1991.

[37] A. Kumar, M. Rabinovich, and R. Sinha. A performance study of general grid structures for replicated data. In *Proc. of the 13th IEEE Int. Conf. on Distributed Computing Systems*, pp. 178-185, May 1993.

[38] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), pp. 558-565, July 1978.

[39] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*, pp. 305-306, McGraw-Hill, New York, 1982.

[40] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*, Prentice-Hall, Englewood Cliffs, 1984.

[41] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pp. 226-238, October 1991.

[42] D. D. E. Long. *The Management of Replication in a Distributed System.* Ph.D. Dissertation, University of California, San Diego, 1988. Also available as a Tech. Report UCSC-CRL-88-07 from Computer Research Laboratory, University of California, Santa Cruz, 1988.

[43] D. D. E. Long and J.-F. Pâris. Regeneration protocols for replicated objects. In *Proc. of the 5th IEEE Int. Conf. on Data Eng.*, pp. 538-545, 1989.

[44] M. Maekava. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. on Comp. Sys.* 3(2), May 1985.

[45] T. Mann, A. Hisgen, and G. Swart. *An algorithm for data replication.* Research Report 46, DEC Systems Research Center, June 1989.

[46] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Trans. on Database Sys.*, 11(4), pp. 378-396, December 1986.

[47] M. L. Neilsen. *Quorum structures in distributed systems*, Ph.D. thesis, Kansas State University, May 1992.

[48] M. L. Neilsen and M. Mizuno. Decentralized consensus protocols. In *Proc. of the 10th Int. Phoenix Conf. on Comp. and Comm.*, pp. 257-262, 1991.

[49] J.-F. Pâris and D. D. E. Long. Efficient dynamic voting algorithms. In *Proc. of the IEEE 4th Int. Conf. on Data Engineering*, pp. 268-275, 1988.

[50] J.-F. Pâris. Voting with witnesses: A consistency scheme for replicated files. In *Proc. of the IEEE Int. Conf. on Distr. Comp*, pp. 606-621, 1986.

[51] J.-F. Pâris and P. K. Sloope. Dynamic Management of Highly Replicated Data. In *Proc. of the IEEE 11th Symp. on Reliable Distributed Systems*, pp. 20-28, 1992.

[52] G. J. Popek, R. G. Guy, T. W. Page, Jr., and J. S. Heidemann. Replication in Ficus distributed file systems. In *Proc. of the IEEE Workshop on Management of Replicated Data*, pp. 5-10, 1990.

[53] M. Rabinovich and E. D. Lazowska. Improving fault-tolerance and supporting partial writes in structured coterie protocols for replicated objects. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 226-235, June 1992.

[54] M. Rabinovich and E. D. Lazowska. The dynamic tree protocol: avoiding "graceful degradation" in the tree protocol for distributed mutual exclusion. In *IEEE Int. Phoenix Conf. on Computers and Communication*, April 1992.

[55] M. Rabinovich and E. D. Lazowska. A fault-tolerant commit protocol for replicated databases. In *Proc. of the 11th ACM Symp. on Principles of Database Systems*, June 1992.

[56] M. Rabinovich and E. D. Lazowska. An efficient and highly available read-one write-all protocol for replicated data management. In *IEEE Int. Conf. on Parallel and Distributed Information Systems*, January 1993.

[57] M. Rabinovich and E. D. Lazowska. Asynchronous epoch management in replicated databases. Technical Report 92-12-02. University of Washington Department of Computer Science, January 1993.

[58] M. Rabinovich and E. D. Lazowska. Asynchronous epoch management in replicated databases. In *Proc. of the 7th Int. Workshop on Distributed Algorithms*, pp. 115-128, Springer-Verlag, September 1993.

[59] M. Rabinovich and E. D. Lazowska. Efficient support for partial write operations in replicated databases. In *Proc. of the 10th IEEE Int. Conf. on Data Engineering*, pp. 43-53, February 1994.

[60] S. Rangarajan, S. Setia, and S. K. Tripathi. A fault-tolerant algorithm for replicated data management. In *Proc. of the 8th IEEE Int. Conf. on Data Engineering*, pp. 230-237, 1992.

[61] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pp. 520-529, 1993.

[62] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and R. Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. of 1985 USENIX Summer Conference*, pp. 119-130, 1985.

[63] D. Skeen. Nonblocking commit protocols. In *Proc. of SIGMOD Int. Conf. on Management of Data*, pp. 133-142, 1981.

[64] D. Skeen. A quorum-based commit protocol. In *Proc. of the 6th Berkeley Workshop on Distr. Data Managenemt and Comp. Networks*, pp. 69-80, 1982.

[65] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Trans. on Software Eng.*, 9(3), pp. 219-228, May 1983.

[66] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. on Computers*, 39(4), 1990.

[67] M. Stonebraker and G. A. Schloss. Distributed RAID - a new multiple copy algorithm. In *Proc. of the IEEE Int. Conf. on Data Eng.*, pp. 430-437, 1990.

[68] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. on Software Eng.*, 5(3), pp. 188-194, 1979.

[69] P. Triantafillou and D. Taylor. Using multiple replica classes to improve performance in distributed systems. In *Proc. of the IEEE Int. Conf. on Distr. Comp. Sys.*, pp. 420-428, 1991.

[70] K. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

[71] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Sys.*, 4(2), pp. 180-209, June 1979.

[72] C. Wu and G. G. Belford. The triangular lattice protocol: a highly fault tolerant and highly efficient protocol for replicated data. In *Proc. of the IEEE 11th Symp. on Reliable Distributed Systems*, pp. 66-73, 1992.

Vitae

Michael (Misha) Rabinovich was born in St. Petersburg (then Leningrad), Russia on July 16, 1955. He graduated with distinction from Leningrad Ulyanov-Lenin Electrotechnical Institute with an Engineer Diploma in Computer Engineering in 1979. He also graduated from College of Music of Leningrad State Conservatory with a diploma in Music Education and Piano Performance in 1976. From 1989 to 1994 he attended the University of Washington, receiving his M.S. degree in Computer Science in 1991 and his Ph.D. degree in Computer Science in 1994. Upon completion of his Ph.D., he joined the AT & T Bell Labs at Murray Hill, New Jersey.