Dynamic Quorum Adjustment for Partitioned Data

Maurice Herlihy
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213
18 September 1986

Abstract

A partition occurs when functioning sites in a distributed system are unable to communicate. This paper introduces a new method for managing replicated data objects in the presence of partitions. Each operation provided by a replicated object has a set of quorums, which are sets of sites whose cooperation suffices to execute the operation. The method permits an object's quorums to be adjusted dynamically in response to failures and recoveries. A transaction that is unable to progress using one set of quorums may switch to another, more favorable set, and transactions in different partitions may progress using different sets. This method has three novel aspects: (1) it supports a wider range of quorums than earlier proposals, (2) it scales up effectively to large systems because quorum adjustments do not require global reconfiguration, and (3) it systematically exploits the semantics of typed objects to support more flexible quorum adjustment.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs — *abstract data types; data types and structures*; D.4.3 [Operating Systems]: File System Management — *distributed file systems;* D.4.5 [Operating Systems]: Reliability — *fault-tolerance*; H.2.4 [Database Management]: Systems — *distributed systems; Transaction Processing*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Replication, reconfiguration, abstract data types

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

1. Introduction

A *replicated object* is a data item that is stored redundantly at multiple locations to enhance availability. A *replication method* is a technique for managing a physically distributed collection of data objects to appear as if it were a single, highly available data object. This paper introduces a new replication method that tolerates site crashes, timing failures, and communication link failures. Each operation provided by a replicated object has a set of *quorums*, which are sets of sites whose cooperation suffices to execute the operation. The method permits an object's quorums to be adjusted dynamically in response to failures and recoveries. A transaction unable to make progress using one set of quorums may switch to another, more favorable set, and transactions in different partitions may progress using different sets. This method has three novel aspects: (1) it supports a wider range of quorums than earlier proposals (e.g., [9, 10, 11]), (2) it scales up effectively to large systems because quorum adjustments do not require global reconfiguration, and (3) it systematically exploits the semantics of typed objects to support more flexible quorum adjustment.

Section 2 describes our model of computation. Section 3 describes our basic technique for *untyped* objects providing read and write operations, and Section 4 extends these methods to exploit the semantics of *typed* objects providing a richer set of operations. Section 5 presents correctness arguments, and Section 6 surveys and analyzes related work. Section 7 summarizes our results.

2. Terminology

A *distributed system* consists of multiple computers (called sites) that communicate through a network. Distributed systems are typically subject to faults such as site crashes, variations in timing, and communication link failures. Transient communication failures may be hidden by low level protocols, but longer-lived failures can cause *partitions*, in which functioning sites are unable to communicate. These classes of failures are usually indistinguishable; a site that fails to receive a response after a certain duration cannot tell whether the original message was lost, the reply was lost, the recipient has crashed, or whether the recipient is just slow to respond.

A *database* consists of a collection of *objects*. In this paper, we consider *untyped objects* that provide read and write operations, as well as *typed objects* such as counters, queues, or directories, that provide a richer set of operations. A *transaction* is an activity that observes and modifies a set of objects. A widely-accepted approach to preserving consistency in the presence of failures and concurrency is to ensure that transactions are *atomic*: that is, serializable and recoverable. *Serializable* means that transactions appear to execute in a serial order [26], and *recoverable* means that an activity either succeeds completely or has no effect. We assume that replicated objects display the same semantics as single-site objects, a property called *one-copy serializability* [2].

The techniques proposed in this paper are intended to enhance the availability of transactions that execute to completion within a single physical partition. We rely on standard commitment protocols (e.g., [17, 29]) to abort transactions interrupted by failures. Skeen [29] has shown that partitions can cause any commitment protocol to block, delaying the transaction's final commit or abort until the partition is

rejoined. Before a transaction's commitment protocol has reached the "prepared" state, however, a site can unilaterally abort the transaction and release its locks. We also rely on standard techniques to resolve deadlocks [24] and to recover from crashes [31].

3. Basic Techniques

Replicated objects are implemented by two kinds of modules: *repositories* and *front-ends*. Repositories provide long-term storage for the object's state, while front-ends carry out operations for clients. Different objects may have different sets of repositories. Because front-ends can be replicated to an arbitrary extent, perhaps placing one at each client's site, the availability of a replicated object is dominated by the availability of its repositories.

A *quorum* for an operation is any set of repositories whose cooperation suffices to execute that operation. A *quorum assignment* associates each operation with a set of quorums. An operation's quorums determine its availability, and the constraints governing an object's quorum assignments determine the range of availability properties realizable by a replication method. Gifford [16] has shown that one convenient way to characterize quorums is to assign weighted votes to repositories so that a collection of repositories is a quorum if and only if the sum of its votes exceeds a threshold value. Barbara and Garcia-Molina [15], however, have shown that not all valid quorum assignments can be characterized by weighted voting. Our examples all use voting schemes in which repositories have equal weight.

This section introduces two complementary techniques for dynamic quorum adjustment: *quorum inflation* shrinks write quorums, possibly expanding read quorums, and *quorum deflation* shrinks read quorums, possibly expanding write quorums. These two techniques can be combined to realize a wide range of adaptive replication methods, including generalizations of techniques in the literature [9, 10, 11]. This section considers only untyped objects providing read and write operations; extensions for typed objects are considered in Section 4.

3.1. Quorum Inflation

When a transaction begins execution, it chooses a natural number called its *level*. Transactions at different levels are serialized in the order of their levels, but transactions at the same level are serialized by standard techniques. Each object provides a sequence of quorum assignments, one for each level, and each transaction must use the quorum assignments for its level.

Concurrency control is accomplished by two separate mechanisms, one to ensure that transactions at the same level remain serializable, and one to ensure that higher-level transactions are serializable after lower-level transactions. Transactions at the same level are synchronized by the underlying system, using any mechanism capable of assigning logical timestamps [25] to transactions so that their timestamp ordering reflects their serialization ordering. For example, Reed's scheme [28] issues such a timestamp to each transaction when it begins execution, and ensures that transactions remain serializable in timestamp order. Alternatively, under strict two-phase locking [12], such a timestamp can be chosen at

any time in the interval after all the transaction's locks have been acquired, but before any have been released. The coordinator for the commitment protocol could choose the timestamp when it initiates the protocol, distributing the value during the first round of messages. We emphasize that the timestamp order is significant only for transactions at the same level.

We ensure that lower-level transactions are serialized before higher-level transactions by a mechanism similar to Reed's multiversion scheme [28]. Each repository has a *ratchet lock*, which is a counter that records the highest level of a transaction that has read the object at that repository. A repository rejects write requests from any transaction whose level is less than the repository's ratchet lock, ensuring that a value read by a higher-level transaction cannot be overwritten by a lower-level transaction. An attempt to read a value written by an uncommitted lower-level transaction is delayed until the latter commits or aborts. Writes are *enabled* for an object at level *n* if none of its ratchet locks exceeds *n*.

An object's quorum assignments must satisfy the following *quorum intersection invariant*:

> If writes to that object are enabled at level *n*, then each write quorum at level *n* must intersect each read quorum at levels greater than or equal to *n*.

For example, an object replicated among three identical repositories might provide either of the quorum assignments shown in Figure 3-1 or 3-2, where levels 3 and higher are bound to the same quorum assignment. In Figure 3-1, if the repositories partition into two groups, transactions at level 1 can read but not write in either partition, transactions at level 2 can read and write in the majority partition, and transactions at higher levels can write but not read in either partition. In Figure 3-2, if R3 is partitioned from the others, transactions at level 1 can read but not write in either partition, and transactions at higher levels can read or write in the majority partition. If, instead, R1 is partitioned from the other two, transactions at level 1 can read but not write in either partition, transactions at level 2 can read but not write in R1's partition, and transactions at higher levels can read and write in R1's partition.

Each repository stores a sequence of timestamped versions, associating with each level its most recently written version, if any. A level *n* transaction reads an object by sending a read request to a front-end, which forwards the request to a level *n* read quorum. Each repository responds with its *closest preceding* version, which is the version associated with the highest level less than or equal to *n*. The front-end returns to the client the version with the highest level and latest timestamp. When the reader commits, each ratchet lock is set to the maximum of its current value and the reader's level. A level *n* transaction writes an object by sending the new version to a front-end. The front-end appends the transaction's timestamp[1] and forwards the version to a level *n* write quorum.[2] After checking the ratchet lock, the repository tentatively installs the new version and responds with a confirmation. The tentative version

---

[1]As discussed in more detail elsewhere [23], if strict two-phase locking is used, the front-end creates a timestamp *template* to be filled in at commit.

[2]For brevity, we assume that each transaction writes to an object no more than once. Multiple writes can be accommodated simply by appending to each version's timestamp a low-order field that reflects the value of the front-end's logical clock when the write occurred.

Level	Read	Write
1	1 of R1,R2,R3	3 of R1,R2,R3
2	2 of R1,R2,R3	2 of R1,R2,R3
3	3 of R1,R2,R3	1 of R1,R2,R3
.	.	.
.	.	.
.	.	.

Figure 3-1: First Multi-Level Quorum Assignment

Level	Read	Write
1	1 of R1,R2,R3	3 of R1,R2,R3
2	1 of R1,R2	2 of R1,R2
3	1 of R1	1 of R1
.	.	.
.	.	.
.	.	.

Figure 3-2: Second Multi-Level Quorum Assignment

becomes permanent when the writer commits.

A transaction that cannot locate a write quorum because of failures or ratchet lock conflicts may abort and restart at a higher level, a process called *quorum inflation*. To increase the likelihood the desired quorum is available at the new level, the underlying system should provide hints about which sites are accessible. To decrease the likelihood of further ratchet lock conflicts, the transaction should restart at a level at least as high as any it has observed in use.

Quorum inflation is illustrated by a simple example. Consider a database in which objects are partially replicated, and object *x* has repositories R1, R2, and R3. Initially, *x* binds level 1 to the "read one, write all" quorum assignment, and levels 2 and higher are bound to majority quorum assignments.

Level	Read	Write
1	1 of R1,R2,R3	3 of R1,R2,R3
2	2 of R1,R2,R3	2 of R1,R2,R3
3	2 of R1,R2,R3	2 of R1,R2,R3
.	.	.
.	.	.
.	.	.

Object *x* is represented schematically as follows. The ratchet lock at each repository is shown at the top of each column. Each version is tagged with its level and timestamp, separated by a colon. In this simplified, schematic representation, logical timestamps are represented by two digits, and new timestamps are generated sequentially. In practice, however, logical timestamps would be much longer, and successive timestamps would separated by arbitrary gaps. Transaction *A* creates and initializes *x* with value *a* at level 1.

R1	R2	R3
Ratchet: 1	Ratchet: 1	Ratchet: 1
1:00 a	1:00 a	1:00 a

If a failure partitions R1 from the others, a transaction *B* executing in the majority partition at level 1 can no longer write to *x*. *B* inflates to level 2, the lowest level at which a write quorum is available, and writes the new value *b*.

R2	R3
Ratchet: 1	Ratchet: 1
1:00 a 2:01 b	1:00 a 2:01 b

Level 1 transactions can still read the object's old version in either partition.

This example illustrates both the advantages and the limitations of quorum inflation. If quorum assignments were fixed, the object could be read and written in at most one partition. Quorum inflation, however, permits transactions in one partition to read and write the object, and transactions in the other partition to read it. Nevertheless, quorum inflation by itself has three important limitations. First, once the object has been read at a higher level, it can no longer be written at lower levels, hence there is no way to restore the original quorum assignment when the partition is repaired. Second, a transaction restarting at a higher level must use the higher-level quorum assignments for all objects it accesses, including objects whose repositories are unaffected by failures. Finally, reading is more expensive at higher levels, since read quorums are larger. These limitations are addressed in the next section.

3.2. Quorum Deflation

This section introduces *quorum deflation*, a complementary technique used primarily to shrink read quorums. Each object maintains a binding between level numbers and quorum assignments. So far, we have assumed that this binding is fixed, but it need not be. An object's bindings are kept in a *quorum assignment table* replicated among its repositories and cached at its front-ends. Quorum deflation is a protocol that alters the quorum assignment bound to a particular level by updating the replicated quorum assignment table, redistributing data as necessary to preserve the quorum intersection invariant.

A *coquorum* for an operation is any set of repositories that has a non-empty intersection with every quorum for that operation. The quorum deflation protocol for level *n* consists of the following steps, which are executed as a transaction, and which must preserve the quorum intersection invariant.

1. From each repository in an old read quorum, read the closest preceding version for level *n*, its timestamp, and the repository's ratchet lock. If there exists an *m*, $n > m$, such that writes are enabled at level *m* but some new read quorum does not intersect some level *m* write quorum, disable writes at level *m* by advancing the repositories' ratchet locks beyond *m*.
2. Copy the latest version and its timestamp to a new write quorum, and the highest ratchet lock value to a new read quorum.
3. Update the quorum assignment tables at an old coquorum for read and write.

Step 1 ascertains the current version for level *n*, advancing ratchet locks as necessary to preserve the

quorum intersection invariant. Because Step 2 copies an existing version, it does not generate a new timestamp, and it ignores existing ratchet locks.

Step 3 ensures the consistency of the cached quorum assignment tables by the following "tripwire" mechanism. Each binding in an object's quorum assignment table has a logical timestamp that is updated each time the binding is changed. Whenever a front-end sends a message to a repository on behalf of a level *n* transaction, it includes its cached timestamp for level *n*'s binding. If a repository receives a message with an out-of-date timestamp, it refuses the request, and notifies the front-end of the new binding. When a front-end receives notification of the new binding, it updates its cache and retries the operation. Step 3 ensures that any level *n* quorum chosen by a front-end whose cache is out of date will include at least one repository whose binding is more current.

Objects decide when to deflate. Each object has a preferred quorum assignment, initially bound to level 1. Front-ends for the object can keep track of the set of accessible repositories, responding to recoveries by rebinding levels to more preferable quorum assignments. In the following example, we assume that reads are much more common than writes, and that partitions are long-lived. Whenever a front-end observes a transaction executing at a level whose read quorums encompass multiple repositories, it tries to rebind that level to single-site read quorums. Deflation could be triggered by the first read attempt, or it could be undertaken asynchronously by a background process. Less aggressive strategies are appropriate if writes are relatively common, or if partitions are expected to be short-lived. For example, deflation might be used only to restore earlier quorum assignments when enough repositories rejoin the partition.

For brevity, we abbreviate the following quorum assignments:

Abbreviation	Read	Write
Q13	1 of R1,R2,R3	3 of R1,R2,R3
Q22	2 of R1,R2,R3	2 of R1,R2,R3
Q12	1 of R2,R3	2 of R2,R3
Q12'	1 of R1,R2	2 of R1,R2

Each repository's quorum assignment table is shown at the top of its column. As a notational convenience, entries in quorum assignment tables are tagged with level 0. The first entry is the binding for level 1, the second for level 2, and so on. The last entry in the quorum assignment table implicitly binds higher levels to the same quorum assignment.

Recall that R2 and R3 are partitioned from R1, and transaction *B* has just inflated to level 2 and written *b* to *x*.

R2	R3
0:00 Q13	0:00 Q13
0:00 Q22	0:00 Q22
Ratchet: 1	Ratchet: 1
1:00 a	1:00 a
2:01 b	2:01 b

When *B* attempts to read from *y*, an object unaffected by the partition, *y*'s front-end uses the deflation protocol to rebind level 2 to its preferred quorum assignment.

When *B* writes to *x* at level 2, the front-end schedules a deflation transaction that eventually rebinds level 2 to Q12. Deflation provides two benefits: it restores the ability to read from a single site, and it ensures that the object will be readable at level 2 if, say, R2 is partitioned from R3. The quorum assignment table is updated at R2 and R3, an old coquorum for read and write.

R2	R3
0:00 Q13	0:00 Q13
0:02 Q12	0:02 Q12
0:00 Q22	0:00 Q22
Ratchet: 1	Ratchet: 1
1:00 a	1:00 a
2:01 b	2:01 b

Because each read quorum in Q12 intersects each write quorum in Q13, the deflation transaction does not need to to advance the ratchet locks. If a level 2 transaction attempts to read the object from a front-end whose cache is out of date, at least one repository will detect a timestamp mismatch, and respond with the up-to-date binding.

R2 is suddenly partitioned from R3 and rejoined to R1. At levels 2 and higher in the majority partition, the object can be read and written, but at levels 1 and 2, it can only be read. (For clarity, a missing version is indicated by a dash.)

R1	R2
0:00 Q13	0:00 Q13
0:00 Q22	0:02 Q12
0:00 Q22	0:00 Q22
Ratchet: 1	Ratchet: 1
1:00 a	1:00 a
———	2:01 b

Transaction *C*, intending to read and write *x*, inflates to level 3. *C*'s front-end rebinds level 3 to Q12' by reading the current version, writing it to R1 and R2, and updating the quorum assignment tables at R1 and R2. Because Q12' includes a read quorum (R1) that does not intersect a write quorum in Q12 (R2 and R3), the ratchet locks at R1 and R2 must be advanced to 3. *C* reads *x* and writes a new value *c*.

R1	R2
0:00 Q13	0:00 Q13
0:00 Q22	0:02 Q12
0:03 Q12'	0:03 Q12'
0:00 Q22	0:00 Q22
Ratchet: 3	Ratchet: 3
1:00 a	1:00 a
2:01 b	2:01 b
3:04 c	3:04 c

Although deflation is primarily useful for shrinking read quorums, it can also be used to move objects from one set of repositories to another. The deflation protocol is easily extended to permit a single deflation transaction to rebind the quorum assignments for multiple levels. In Step 1, the deflation protocol chooses a set of repositories that encompasses an old read quorum for all levels being rebound, and reads the closest preceding versions for all levels. Even if the number of levels being rebound is infinite, there will be only a finite number of versions, and a finite number of distinct quorum assignments. In Step 2, these versions are written to a new write quorum for all levels, and in Step 3 the quorum assignment tables are updated at old coquorums for all levels.

For example, if it becomes apparent that R3 will be inaccessible for a long time, the deflation protocol can be used to replace R3 with a new repository R4 at levels 3 and higher.

Abbreviation	Read	Write
Q13'	1 of R1,R2,R4	3 of R1,R2,R4
Q22'	2 of R1,R2,R4	2 of R1,R2,R4

To rebind level 3 to Q13' and levels 4 and higher to Q22', the front-end reads the one version current at levels 3 and higher, writes it to R4, and updates the quorum assignment tables at R1, R2, and R4. To save storage, repositories eventually discard versions and quorum information for levels 1 and 2. Later requests from transactions at those levels will be refused.

R1	R2	R4
0:05 Q13'	0:05 Q13'	0:05 Q13'
0:05 Q22'	0:05 Q22'	0:05 Q22'
Ratchet: 3	Ratchet: 3	Ratchet:3
3:04 c	3:04 c	3:04 c

3.3. Remarks

Inflation and deflation are complementary techniques. Inflation shrinks a transaction's write quorums for multiple objects, while deflation shrinks an object's read quorums for multiple transactions. Inflation can enhance availability, since a transaction restarting at a higher level may be able to locate a write quorum unavailable at lower levels. Although deflation cannot make unavailable operations available again, it can

enhance efficiency since smaller read quorums require fewer messages, and it can enhance long-term availability, since smaller read quorums are more likely to survive subsequent failures. For both techniques, quorum adjustment is a local process; inflation is undertaken by individual transactions, and deflation by individual objects.

Inflation and deflation can be combined to implement a wide range of adaptive replication techniques. In Section 6, we show that inflation and deflation can provide at least as much availability as certain techniques in the literature. Choosing the best adaptive strategy for a particular application is a complex optimization problem that must take into account the frequency and distribution both of failures and of operations. For example, the quorum assignment shown in Figure 3-1 has the advantage that transactions in the majority partition can both read and write the object, and transactions in a minority partition can read or write the object, but not both. It has the disadvantage that levels 3 and higher can be deflated only when all three repositories are available. In the quorum assignment used in the example, deflation at levels 2 and higher requires only a majority, but minority write quorums are not supported. In the quorum assignment shown in Figure 3-2, the object can always be read and written in R1's partition, but it can never be written elsewhere.

A typical application of inflation and deflation can be summarized as follows. Each object's read quorums are smaller at lower levels, and write quorums are smaller at higher levels. A transaction unable to locate a write quorum for an object inflates to a level where such a quorum is still available. As illustrated in the example above, the object may then use the deflation protocol to "customize" that level's quorum assignment to the current physical partition. The inflated transaction advances ratchet locks as it reads, which in turn provoke other transactions to inflate, and the higher level gradually propagates through the system. As soon as it can, each object accessed by a higher-level transaction rebinds that level to its preferred quorum assignment, and in the absence of additional failures, the system stabilizes in a state where all transactions execute at the higher level, but using the original, preferred quorum assignments. The accuracy of the underlying failure detection mechanism does not affect the correctness of our replication method, although unnecessary quorum adjustments would harm performance.

A transaction executing at a level where write quorums are large might reduce write latency by spawning an asynchronous process to update repositories in parallel with the transaction's other activities. Such asynchronous writes would have to be integrated with the underlying synchronization mechanism to ensure serializability, perhaps using a nested transaction mechanism. Failure to complete the updates at a write quorum for the transaction's current level would force the parent transaction to restart.

No transaction that can locate a read quorum is ever prohibited from reading the object (provided that level's version has not been discarded), but a transaction may be prohibited from writing if the object has been read by a higher-level transaction. As discussed in Section 6, related quorum adjustment techniques [9, 10, 11] impose equivalent limitations on writes, and somewhat stronger limitations on reads.

4. Extensions for Typed Objects

Elsewhere [23, 20], we have shown that replication techniques that exploit the semantics of typed objects can realize a wider range of quorum assignments than techniques that classify operations only as reads and writes. This section shows how those techniques can be extended to support quorum inflation and deflation. The discussion here is informal, focusing on examples and motivation. A formal treatment is given in Section 5.

An object's *type* defines a set of possible *states* and a set of primitive *operations* that provide the only means to manipulate objects of that type. For example, a bank account might be modeled by an *account* object that provides *Balance*, *Credit* and *Debit* operations. Balance returns the current account balance:

```
Balance = Operation() Returns (Dollar)
```

Credit increments the balance:

```
Credit = Operation(sum: Dollar).
```

Debit attempts to decrement the balance:

```
Debit = Operation(sum: Dollar) Signals (Overdrawn).
```

If the amount to be debited exceeds the balance, the operation signals an exception, leaving the balance unchanged. For brevity, a Debit operation that terminates normally is simply called a *debit*; otherwise it is called an *overdraft*.

The conventional approach to replicating typed objects is to treat all operations as combinations of reads and writes. For example, both Credit and Debit would read the current balance, perform some arithmetic, and write out the altered balance. Clearly, quorum inflation and deflation can do little to enhance the availability of such an implementation, since Credit and Debit each require both a read and a write quorum. This section describes an alternative approach that permits Credit and Debit to occur in distinct partitions.

4.1. Representing Typed Objects

A replicated object's state is represented as a *log*, which is a sequence of *entries*, each consisting of a level, a logical timestamp, and an operation. An operation is written *op(args*)/term(res*)*, where *op* is an operation name, *args** is a sequence of argument values, *term* is a termination condition name, and *res** is a sequence of result values. The operation name and argument values constitute the *invocation*, and the termination condition and result values constitute the *response*. We use "Ok" for normal termination.

Log entries are partially replicated among the repositories. For example, the following is a schematic representation of an account replicated among three repositories.

R1	R2	R3
1:01 Credit($5)/Ok()	1:01 Credit($5)/Ok()	
	1:03 Credit($10)/Ok()	1:03 Credit(10)
2:02 Credit($15)/Ok()		2:02 Credit(15)

The account has been credited three times, twice at level 1 and once at level 2, but no single repository has an entry for all three credits.

It is convenient to divide an operation's quorum into two parts: a front-end executing the invocation reads from an *initial quorum* of repositories, performs a local computation to choose a response, and records the new event at a *final quorum*. Initial quorums depend on the operation name, and final quorums also depend on the termination condition name. (Initial and final quorums may also depend on argument and result values, but we do not use them in this paper.) For untyped objects, read quorums are really initial quorums for Read, and write quorums are really final quorums for Write. Final Read quorums and initial Write quorums are empty.

To execute an operation, a level n transaction sends the invocation to a front-end, which collects the logs for levels less than or equal to n from a level n initial quorum for that invocation. The front-end merges the logs by level and by timestamp, discarding duplicates, to construct a history called the *view*. It chooses a response to the invocation consistent with the object's state as reconstructed from the view. (The view may not completely reflect the object's current state, but it will contain enough information to choose a correct response.) The front-end records the operation by appending a new entry to the view, and sends the updated view to a level n final quorum. Each repository in the final quorum merges the updated view with its resident log. The front-end returns the response to the client transaction when it has received acknowledgments from the final quorum.

We emphasize that logs represent a conceptual model for the replicated data, not a literal design for an implementation. The advantage of the log formalism is that it allows us to reason abstractly about the fundamental constraints governing availability without having to consider *ad hoc* techniques for data representation. In practice, the best representation for an object is application-specific. For example, if a repository has a prefix of the object's complete log, it may replace that prefix with a version whose timestamp is that of the last entry it replaces. For an untyped object, the version is just the last value written at that level, and for an account, the version is the account balance. A more detailed discussion of log and message compaction appears elsewhere [23].

4.2. Quorum Inflation

The fundamental constraint governing quorum assignments, and hence realizable levels of availability, is a property called *serial dependency* [19, 23]. A precise definition of dependency is given in Section 5. Informally, however, an operation p depends on an operation q if the front-end executing the invocation for p must observe earlier entries for q to choose a valid response. For example, reads depend on earlier writes, and debits and balance inquiries depend on earlier credits and debits, but not on earlier overdrafts. Writes and credits do not depend on any earlier operations.

Each repository maintains a *ratchet lock* for each invocation. The ratchet lock is a counter that records the highest level of a transaction that has included that repository in an initial quorum for that invocation. The ratchet lock ensures that operations executed by higher-level transactions cannot be invalidated by later operations of lower-level transactions. Let p and q be operations, and let *inv(p)* denote the invocation part of p. If p depends on q, the repository will not participate in a final quorum for q for any transaction whose level is less than the ratchet lock for *inv(p)*. An attempt to execute an invocation that

depends on an uncommitted lower-level operation is delayed until the lower-level transaction commits or aborts. An operation *q* is *enabled* at level *n* if no conflicting ratchet locks exceed *n*. The ratchet locks for untyped objects are really ratchet locks for read invocations; ratchet locks for writes are unnecessary because writes cannot be invalidated by lower-level operations.

Quorum assignments must satisfy the following *generalized quorum intersection invariant*:

> If *p* depends on *q* and *q* is enabled at level *n*, then each final quorum for *q* at level *n* must intersect each initial quorum for *p* at levels greater than or equal to *n*.

If all transactions are restricted to a single level, this constraint is equivalent to that imposed by *general quorum consensus* [23].

An account replicated among three repositories might support the following quorum assignments, where an entry of the form (*m*,*n*) means that the operation has initial quorums of any *m* repositories and final quorums of any *n* repositories.

Abbreviation	Credit	Debit	Overdraft/Balance
Q1	(0,3) of R1,R2,R3	(1,3) of R1,R2,R3	(1,0) of R1,R2,R3
Q2	(0,2) of R1,R2,R3	(2,2) of R1,R2,R3	(2,0) of R1,R2,R3
Q3	(0,1) of R1,R2,R3	(3,1) of R1,R2,R3	(3,0) of R1,R2,R3

Initially, level 1 is bound to Q1, level 2 to Q2, and levels 3 and higher to Q3.

The account is initialized with a balance of zero. Transaction *A* at level 1 credits ten dollars.

R1	R2	R3
0:00 Q1	0:00 Q1	0:00 Q1
0:00 Q2	0:00 Q2	0:00 Q2
0:00 Q3	0:00 Q3	0:00 Q3
Debit: 1	Debit: 1	Debit: 1
Balance: 1	Balance: 1	Balance: 1
1:01 Credit($10)/Ok()	1:01 Credit($10)/Ok()	1:01 Credit($10)/Ok()

The repositories are partitioned into a group containing R1 and a group containing R2 and R3. In the minority partition, transaction *B* discovers it cannot locate a level 1 final credit quorum, restarts at level 3, and credits five dollars at R1.

R1
Debit: 1
Balance: 1

1:01 Credit($10)/Ok()
3:02 Credit($5)/Ok()

Meanwhile, in the other partition, transaction *C* discovers it cannot locate a level 1 initial debit quorum, inflates to level 2, and debits ten dollars at R2 and R3, advancing the ratchet locks for debit.

R2	R3
Debit: 2	Debit: 2
Balance: 1	Balance: 1
1:01 Credit($10)/Ok()	1:01 Credit($10)/Ok()
2:03 Debit($10)/Ok()	2:03 Debit($10)/Ok()

The partitions are rejoined, and *C* executes a Balance at R1 and R2, observing that the account is empty and advancing the ratchet locks. *C* does not observe the five dollars credited by *B* because *B* is serialized "in the future."

R1	R2	R3
Debit: 1	Debit: 2	Debit: 2
Balance: 2	Balance: 2	Balance: 1
1:01 Credit($10)/Ok()	1:01 Credit($10)/Ok()	1:01 Credit($10)/Ok()
3:02 Credit($5)/Ok()	2:03 Debit($10)/Ok()	2:03 Debit($10)/Ok()

4.3. Quorum Deflation

The quorum assignment for level *n* is rebound in the following steps, which must be executed atomically.

1. Merge the level *n* entries from an old initial quorum for each invocation, advancing ratchet locks as necessary to preserve the generalized quorum intersection invariant.
2. Write out the entries to a new final quorum for each operation, and the ratchet locks to a new initial quorum for each invocation.
3. Update the quorum assignment table at an old coquorum for each operation.

Step 1 ensures that each level *n* entry appears in the merged log, and Step 2 ensures that each level *n* entry appears at its new final quorum. Step 3 preserves cache consistency, as discussed above.

To rebind level 2 to Q1, the first step is to collect the entries at level 2 and lower from an initial quorum in Q2 for Credit and Debit, say R1 and R2. These entries are then written out to a final quorum in Q1 for Credit and Debit, here consisting of all three repositories. The final step is to update the quorum assignment table at a coquorum in Q2 for each operation, say R1 and R2.

R1	R2	R3
0:00 Q1	0:00 Q1	0:00 Q1
0:03 Q1	0:03 Q1	0:00 Q2
0:00 Q3	0:00 Q3	0:00 Q3
Debit: 2	Debit: 2	Debit: 2
Balance: 2	Balance: 2	Balance: 2
1:01 Credit($10)/Ok()	1:01 Credit($10)/Ok()	1:01 Credit($10)/Ok()
2:03 Debit($10)/Ok()	2:03 Debit($10)/Ok()	2:03 Debit($10)/Ok()
3:02 Credit($5)/Ok()		

If a level 2 transaction attempts to credit the account from a front-end whose cache is out of date, it will send the credit entry to two repositories, at least one of which will detect a mismatch, and respond with the up-to-date binding. The front-end updates its cache and retries the operation.

4.4. Remarks

We have assumed an independent atomicity mechanism provided by a lower level of the system. An alternative approach is to provide an integrated mechanism that handles concurrency control and recovery as well as replication. Integrated mechanisms are more complex, but we have shown elsewhere [20, 21] that they can provide better concurrency because they can exploit more information. For example, *consensus locking* [20] is a generalization of strict two-phase locking in which each operation must acquire *initial locks* at its initial quorum, and *final locks* at its final quorum. Certain initial and final locks conflict. A necessary and sufficient condition to ensure atomicity is that the intersection of the lock conflict relation and the quorum intersection relation be a serial dependency relation, thus atomicity and replication mechanisms are bound by a common constraint. Consensus locking can be extended to support multiple levels by observing that initial locks need not conflict with higher-level final locks [22]. Even more concurrency can be achieved if scheduling decisions use state information as well as operation conflicts, although such mechanisms may impose additional constraints on quorum assignments [20, 21].

5. Correctness Arguments

This section gives formal correctness arguments for quorum inflation and deflation. Our model extends techniques used to to prove the correctness of simpler quorum-consensus techniques for typed objects [23]. Weihl [32] has used a similar formalism to prove correctness for atomicity mechanisms.

5.1. Serial Computations

In the absence of concurrency and failures, a computation is modeled by a *history*, which is a finite sequence of object/operation pairs. For example,

x Credit(20)/Ok()
x Debit(15)/Ok()
x Debit(10)/Overdraft()
x Balance()/Ok(5)

is a history for account object *x*. Initially, *x*'s balance is zero. The account is credited $20, debited $15, and an attempt to debit $10 for *x* is rejected. Finally, *x*'s balance is observed to be $5.

An object's *specification* defines a set of legal histories for that object. For example, the specification for an Account object includes only histories in which the balance covers each debit and fails to cover each overdraft. A history is *legal* if the subhistory associated with each object is legal for that object.

Constraints on quorum intersection are governed by the notion of a *serial dependency relation* between invocations and operations. Let >> be any relation between invocations and operations. (For brevity, we write "$p >> q$" for "$inv(p) >> q$.") Informally, a subhistory g of h is *closed* under >> if whenever it contains an operation p it also contains every earlier operation q of h such that $p >> q$. More precisely, let $h(i)$ denote the i-th operation of h:

Definition 1: A history g is a *closed subhistory* of h with respect to >> if there exists an injective order-preserving map s such that $g(i) = h(s(i))$ for all i in the domain of g, and if $p >> q$, $j > j'$, $h(j) = p$, $h(j') = q$, and $s(i) = j$, then there exists i' such that $s(i') = j'$.

Informally, >> is a *serial dependency relation* if a response to an invocation is legal for a complete history whenever it is legal for a closed subhistory of h that includes the operations on which the invocation depends. More precisely, let " · " denote concatenation:

> **Definition 2:** A relation >> is a *serial dependency* relation if, for all operations p and all legal histories h, whenever g is a closed subhistory containing all operations q such that $p >> q$:
>
> $g \cdot p$ is legal $\Rightarrow h \cdot p$ is legal.

Of principal interest are *minimal* serial dependency relations, having the property that no smaller relation is a serial dependency relation. As discussed elsewhere [23], a data type may have multiple minimal serial dependency relations, and concurrency control and replication methods based on serial dependency provide more concurrency and more quorum choices than conventional techniques based on commutativity.

5.2. Concurrent Computations

A *schedule* is a sequence of object/operation/transaction triples. Operations of different transactions are not interleaved; we assume that individual operations are serialized and recovered by an independent underlying atomicity mechanism. The order induced by this mechanism is called the *physical serialization* order.

Each transaction has an associated level. We define the *logical serialization order* as follows: A precedes B in the logical serialization order if A's level is lower than B's, or if their levels are equal, and A is physically serialized before B. If S is a schedule, let reorder(S) be the history constructed by reordering the operations of S in the logical serialization order of their transactions. S is *logically serializable* if reorder(S) is a legal history. Logical serializability plays the same role as *one-copy serializability* in graph-oriented models [2]; it states that a replicated object satisfies the same specification as a single-site object.

Since all objects agree on the relative physical ordering of transactions and on their levels, a system's schedules are logically serializable if and only if each individual object's schedules are logically serializable. Henceforth, we consider only schedules for individual objects, omitting object names from schedules and histories. A replicated object is modeled as an automaton that accepts certain schedules. We show that all schedules accepted by the automaton are logically serializable if and only if it satisfies the generalized quorum intersection invariant.

5.3. Multi-Level Automata

We use the following primitive domains: INV is the set of invocations, RES is the set of responses, REPOS is the set of repositories, NAT is the set of natural numbers, and TIMESTAMP is the set of timestamps. We also use the following derived domains: OP = INV × RES is the set of operations, QUORUM = 2^{REPOS} is the set of quorums, and a *log* is a map from a finite set of timestamps to operations: LOG = TIMESTAMP → OP.

Two logs L and M are *coherent* if they agree for every value where they are both defined. The *merge* operation $\cup$ is defined on pairs of coherent logs by:

$(L \cup M)(t)$ =if $L(t)$ is defined then $L(t)$ else $M(t)$.

Every log corresponds to a history constructed by ordering its operations in timestamp order. For brevity, we sometimes refer to a log L in place of its history, e.g., "L is legal" instead of "the history represented by L is legal." The exact meaning should be clear from context.

A replicated object is modeled as a non-deterministic automaton that accepts certain schedules. A *multi-level* automaton has the following state components:

- Log: REPOS × NAT → LOG
- Clock: TIMESTAMP
- Ratchet: REPOS × INV → NAT
- Initial: INV × NAT → 2^{QUORUM}
- Final: OP × NAT → 2^{QUORUM}

The *Log* component associates a log (initially empty) with each repository and each level. The *Clock* component models a system of logical clocks; it may have an arbitrary initial value. *Ratchet(R,i)* is the value of the ratchet lock for invocation *i* at repository R, initially 1. *Initial(inv(p),n)* and *Final(p,n)* are the sets of initial and final quorums for *p* at level *n*. For brevity, we write "Initial(p,n)" and "Ratchet(R,p)" for "Initial(inv(p),n)" and "Ratchet(R,inv(p))."

An operation *p* is *enabled* at level *n* if Ratchet(R,p) ≤ *n* at all repositories R. Initial and Final induce a *quorum intersection relation* $>>_Q$:

$p >>_Q q \equiv$
If *q* is enabled at level *n* then each final quorum for *q* at level *n* intersects each initial quorum for *p* at levels *n* and higher.

If Q is a set of repositories, define:

$\text{Log}(Q,n) = \cup_{R \in Q} \text{Log}(R,n)$,

$\text{View}(Q,A) = \text{Log}(Q,1) \cup \ldots \cup \text{Log}(Q,\text{Level}(A))$, and

$\text{Ratchet}(Q,p) = \max_{R \in Q} (\text{Ratchet}(R,p))$.

A's view of Q is the log of operations recorded at Q that are logically serialized before *A*.

An operation/transaction pair <*p*,*A*> is accepted only in states satisfying the following preconditions:

- There exists IQ ∈ Initial(p,Level(A)) such that View(IQ,A) · p is a legal history.
- There exists FQ ∈ Final(p,Level(A)) such that Ratchet(FQ,q) ≤ Level(A)

These preconditions state that that the new operation must produce a legal history when appended to the transaction's view, and there are no conflicting ratchet locks at the final quorum. This transition produces the following postcondition, where *x'* denotes the new value of component *x*.

- Clock' > Clock
- Log'(R,m)(t) = If R ∉ FQ then Log(R,m)(t)
  elseif m = n and t = Clock' then p
  else (Log(R,m) ∪ Log(IQ,m))(t)
- Ratchet'(R,p) = if R ∉ IQ then Ratchet(R,p)

else max(Ratchet(R,p),Level(A))

These postconditions state that the clock is advanced, the new operation is appended to the transaction's view, the view is merged with the logs at the final quorum, and the ratchet locks are advanced at the initial quorum.

The automaton also undergoes spontaneous *deflation transitions*, which have the following postconditions. There exists n, and sets of repositories I, I', and F' such that:

- For all p, I ∈ Initial(p,n), I' ∈ Initial'(p,n), and F' ∈ Final'(p,n).
- For all R ∈ F', Log'(R,k) = Log(R,k) ∪ Log(I,k).
- For all R ∈ I', Ratchet'(R,p) ≥ Ratchet(I,p).
- For all $k \geq j$, if $p >>_Q q$, IQ ∈ Initial'(p,k), FQ ∈ Final'(q,j), and Ratchet'(IQ,p) ≤ j, then IQ ∩ FQ ≠ ∅.

These postconditions state that for some level, deflation reads the logs and ratchet locks from an old initial quorum for all operations, writes out the merged logs to a new final quorum for all operations, and writes out the current ratchet lock value to a new initial quorum. Finally, the new quorum assignments and ratchet lock values satisfy the generalized quorum intersection invariant.

5.4. The Proof

We use the following technical lemmas. Let S be the schedule accepted by an automaton in a given state.

Lemma 3: If >> is an arbitrary relation between invocations and operations, the result of merging logs closed under >> produces a closed subhistory of reorder(S).

Lemma 4: If p $>>_Q$ q and IQ ∈ Initial(p,Level(A)), then View(IQ,A) includes all entries for q in S executed by transactions logically serialized before A.

Proof: The property clearly holds for the original quorum assignments, and it is preserved by deflation.

Lemma 5: Let >> be a serial dependency relation, p an operation, and h_1 and h_2 histories such that $h_1 \cdot h_2$ and $h_1 \cdot p$ are legal. If h_2 contains no q such that $q >> p$, then $h_1 \cdot p \cdot h_2$ is legal.

Proof: The result is immediate if h_2 is empty. Otherwise, assume h_2 has a prefix $h_3 \cdot q$ such that $h_1 \cdot p \cdot h_3$ is legal, but $h_1 \cdot p \cdot h_3 \cdot q$ is not. The history $h_1 \cdot h_3$ is a closed legal subhistory of $h_1 \cdot p \cdot h_3$ containing all operations r such that $q >> r$. Because >> is a serial dependency relation, and $h_1 \cdot p \cdot h_3$ is legal, so is $h_1 \cdot p \cdot h_3 \cdot q$, a contradiction.

We now identify an invariant property of multi-level automata.

Lemma 6: If the automaton has accepted S, then View(Q,A) is a closed subhistory of reorder(S) for any set of repositories Q and any transaction A.

Proof: The argument is by induction on the number of transitions. The base case is trivial, and the induction hypothesis asserts the result in the current state. The induction step shows that the result is preserved across both kinds of transitions: deflation and accepting an operation/transaction pair. It suffices to show that View(Q,R) remains closed for any single repository R; the more general result follows from Lemma 3.

Following a deflation transition that merges the logs from Q and sends them to R, View'(R,A) = View(R,A) ∪ View(Q,A) = View(Q ∪ {R},A) which is closed by the induction hypothesis.

Suppose the automaton accepts <p,B>, where R is in B's initial quorum, IQ. If Level(A) < Level(B), then View'(R,A) = View(R,A) ∪ View(IQ,A), which is closed by the induction hypothesis. Otherwise,

View'(R,A) = View(R,A) ∪ (View(IQ,B) · p)

View(IQ,B) and View(R,A) are closed by the induction hypothesis, and View(IQ,B) · p is closed by construction, therefore View'(R,A) is closed by Lemma 3.

Corollary 7: If IQ ∈ Initial(p,Level(A)), then View(IQ,A) is a closed subhistory of reorder(S).

We are now ready to present the basic correctness result:

Theorem 8: If $>>_Q$ is a serial dependency relation, then every schedule accepted by a quorum consensus automaton is logically serializable.

Proof: The argument proceeds by induction on the number of transitions. The result holds trivially in the automaton's initial state. Assume as the induction hypothesis that the automaton has accepted the logically serializable schedule S. The result is trivially preserved across deflation transitions. We show the result is preserved if the automaton accepts the operation/transaction pair <p,A> with an initial quorum IQ. Let reorder(S · <p,A>) = $h_1 \cdot p \cdot h_2$. View(IQ,A) is a closed subhistory of h_1 (Lemma 6), it is legal (because View(IQ,A) · p is legal), and it contains every q such that $p >>_Q q$ (Lemma 4). Because $>>_Q$ is a serial dependency relation and View(IQ,A) · p is legal by construction, $h_1 \cdot p$ is legal. Reorder(S) = $h_1 \cdot h_2$ is legal by the induction hypothesis, and the ratchet locks and Lemma 4 ensure that there is no q in h_2 such that $q >>_Q p$, thus $h_1 \cdot p \cdot h_2$ is legal by Lemma 5.

If all transactions execute at the same level, then the multi-level automaton reduces to a *quorum consensus automaton.* Elsewhere [23; Theorem 9], we have shown that no constraint on quorum intersection weaker than serial dependency guarantees correctness for quorum consensus automata, a result that translates into the basic optimality result for quorum inflation/deflation:

Theorem 9: For any $>>_Q$ that is not a serial dependency relation, there exists a multi-level automaton that accepts a schedule that is not logically serializable.

Theorem 8 shows that multi-level automata accept only logically serializable histories, but the following lemma is need to show that they accept any histories at all. The proof is omitted since it is a straightforward generalization of a proof given elsewhere [23; Lemma 6].

Lemma 10: If the quorum intersection relation $>>_Q$ is a serial dependency relation, then View(Q,A) is a legal history for all sets of repositories Q.

This lemma ensures, for example, that a Credit or Debit will succeed in choosing a response whenever it can locate an appropriate quorum. This lemma also reveals the following *fail-safety* property: if a catastrophic failure makes it permanently impossible to assemble a quorum for certain operations, the result of merging the logs from any set of surviving repositories yields a logically serializable subschedule of the true (lost) schedule.

6. Related Work

The first part of this section surveys background material, and the remainder of the section gives a detailed comparison of our method with similar methods proposed by Eager and Sevcik [9], by El-Abbadi, Skeen, and Cristian [10], and by El-Abbadi and Toueg [11]. Some differences are apparent: our techniques support more flexible quorum adjustment for typed objects, and our use of multiple versions forces fewer restarts. Even for untyped objects, however, we show that our scheme provides greater flexibility: every quorum adjustment permitted by any of the alternatives can be achieved by some combination of quorum inflation and deflation, but not vice-versa. Moreover, our techniques scale up more effectively to large distributed systems because they do not require reconfiguration transactions that span multiple objects.

6.1. Background

Early replication methods for untyped objects [1, 30] did not preserve one-copy serializability, nor do more recent replication methods for directories [4, 13]. Replication methods that preserve one-copy serializability in the presence of site crashes but not partitions include SDD-1 [18] and the *available copies* scheme [3].

Optimistic replication methods permit inconsistencies to develop during partitions, but these inconsistencies are detected and reconciled when communication is restored. Reconciliation methods may be *ad hoc*, as in Locus [27] or Data-patch [14], or systematic, as in proposals by Davidson [7] and Wright [33].

Pessimistic replication methods prevent inconsistencies from developing. Wright [33] has proposed a pessimistic scheme in which transactions taken from a predefined set are classified by their read and write sets. A dependency analysis determines which classes can execute concurrently in distinct partitions. Unlike the scheme proposed here, Wright's scheme requires that each transaction's data dependencies be known in advance.

Quorum consensus methods are pessimistic; an operation invocation will succeed in any partition containing an appropriate quorum. Quorum consensus replication methods have been proposed for files by Gifford [16], for directories by Bloch, Daniels, and Spector [5], and for arbitrary data types by the author [19, 23, 20].

Davidson, Garcia-Molina, and Skeen [8] have published a survey of techniques for preserving consistency in partitioned networks, and Coan, Oki, and Kolodner [6] have published a quantitative analysis of some of these techniques.

6.2. Missing Writes

In Eager and Sevcik's *missing write* scheme [9], transactions execute in one of two modes: normal or partitioned. In *normal* mode, transactions read from any copy of an object and write to all copies. In *partitioned* mode, transactions use Gifford's quorum consensus method to read and write a majority of copies. Each partitioned-mode transaction posts *missing write information* at each site it visits. When a normal-mode transaction attempts to read or write an object with posted missing write information, it must restart in partitioned mode, or execute a restoration protocol as discussed below.

Quorum inflation is a straightforward generalization of Eager and Sevcik's protocol. The quorum assignments for normal and partitioned modes are the same as those for levels 1 and 2 in Figure 3-1. Missing write information serves the same function as ratchet locks; it ensures that transactions in normal mode are never serialized after transactions in partitioned mode. Quorum inflation extends this technique by incorporating more than two levels, multiple versions, timestamps instead of version numbers, and the exploitation of data type semantics.

More fundamental distinctions arise when we turn our attention to deflation. In Eager and Sevcik's method, a normal-mode transaction that becomes aware of missing writes may avoid restarting in partitioned mode if it can carry all the missing writes. A complex protocol ensures that the missing writes are executed in the correct order, and that any other missing writes discovered during the protocol are themselves executed correctly. Once a missing write has been executed, it need no longer be posted. Using our terminology, Eager and Sevcik's restoration protocol retroactively lowers the levels of the transactions whose missing writes are carried out.

Although this protocol has the advantage that a normal-mode transaction that becomes aware of missing writes may still execute to completion if it is able to carry out the missing writes, it has the disadvantage that it imposes a high level of interdependence among distinct objects. A repository for one object may have to keep track of an arbitrary amount of missing write information for other objects. Propagation of missing write information may force a transaction to use partitioned-mode quorums even when all repositories for all objects it uses are available. Restoring normal quorums for one object may require visiting an arbitrary number of other sites, making it difficult or impossible to predict the amount of work required, or the likelihood that the necessary sites will be available. Finally, restoration may become progressively more difficult as missing write information propagates through the system.

The principal advantage of our protocol is that deflation is accomplished on a per-object basis instead of a per-transaction basis. Each object manages its quorum assignments independently of the others. One object's preferred quorum assignment can be restored without affecting other objects. Because object boundaries are fixed in advance, while transaction boundaries are not, this independence makes it possible to predict both the amount of work required to restore a quorum assignment, and the likelihood that the necessary repositories will be available. Repositories for one object do not keep track of information about other objects, and deflation does not become progressively more difficult.

6.3. Virtual Partitions

In a scheme proposed by El-Abbadi, Skeen, and Cristian [10], levels are called *virtual partitions*. Each virtual partition is associated with a set of sites. An object is *accessible* in virtual partitions that include a majority of its sites. A transaction may read or write only those objects that are accessible in its virtual partition. A read quorum for a virtual partition is any site in that partition, and a write quorum is all sites in that partition. Virtual partitions are totally ordered by logical timestamps, and each site keeps track of the latest virtual partition to which it belongs. Sites use these timestamps as ratchet locks, rejecting requests from transactions in earlier virtual partitions.

The principal advantage of the virtual partition scheme is that any accessible object can be reconfigured to be read at any accessible repository. Quorum inflation and deflation can be combined as follows to achieve the same property. Each object binds level 1 to the "read one, write all" quorum assignment, and higher levels to majority quorums, just as in the example in Section 3. Transactions respond to failures by inflating to higher levels. Whenever a front-end for an object receives a read or write request at a level bound to majority quorums, it ascertains which repositories are accessible, and starts a deflation transaction to rebind that level to read from any accessible site and to write to all accessible sites. The deflation transaction reads from a majority of sites, advancing their ratchet locks, and writes the current version to all accessible sites, updating (at least) a majority of that object's quorum assignment tables.

As illustrated by the examples in Section 3, our techniques also permit quorums and quorum adjustments not permitted by virtual partitions. For example, when a site rejoins a partition, objects may respond by executing small deflation transactions instead of a single large transaction to create a new virtual partition. Our techniques support multi-site read quorums, minority write quorums, and objects that may be moved from one set of repositories to another.

The principal disadvantage of the virtual partition scheme is that it does not appear to scale up effectively to large decentralized systems. While our scheme adjusts quorums lazily, inflating and deflating individual transactions and objects as the opportunity arises, the virtual partition scheme adjusts quorums eagerly, responding to a failure or recovery by executing a single transaction that must lock, read, and update every object accessible in the new virtual partition, effectively shutting down the database while reconfiguration is in progress. Eager quorum adjustment takes longer to respond to failures and recoveries, and there may be a substantial likelihood that such a large transaction will be aborted by failures or synchronization conflicts, perhaps causing much work to be lost. If partitions are long-lived, both schemes reconfigure each accessible object once, but if partitions are short-lived, the lazy approach may avoid some unnecessary quorum adjustment. As the number of sites and objects grows, the lifetimes of virtual partitions shrink, the cost and latency of creating new virtual partitions increases, and the overall availability of the system diminishes.

6.4. Accessibility Thresholds

El-Abbadi and Toueg [11] have proposed an extension to the virtual partition scheme that replaces the "read one, write all" mechanism with a more flexible quorum consensus mechanism. In this scheme, levels are called *views*, and each view is associated with a set of sites. Each object x is assigned *read* and *write accessibility thresholds*, which are integers denoted $A_r[x]$ and $A_w[x]$, such that $A_r[x]+A_w[x]$ exceeds the number of copies of x, and $A_w[x]$ exceeds half the number of copies of x. An object is read (write) accessible in a view only if $A_r[x]$ ($A_w[x]$) copies of x are associated with that view. In a view v, each accessible object x is assigned *read* and *write quorums*, which are integers denoted $q_r[x,v]$ and $q_w[x,v]$, such that $q_r[x,v]+q_w[x,v]$ exceeds the number of copies of x in v, $q_r[x,v]$ is less than the number of copies of x in v, and $q_w[x,v]$ exceeds $A_w[x]$.

The principal advantage of the threshold scheme over the virtual partition scheme is that it permits a wider range of trade-offs between quorum sizes for read and write operations. An analysis similar to that undertaken for virtual partitions shows that quorum inflation and deflation can be combined to realize every quorum assignment permitted by the threshold scheme. If v is the first view, each x binds level 1 to read and write quorums of sizes $q_r[v,x]$ and $q_w[v,x]$, and each higher level to read and write quorums of sizes $A_r[x]$ and $A_w[x]$. A transaction creates a new view v' by restarting at a higher level. When a front-end for x receives a read or write request at a new level, it ascertains which repositories are accessible, and executes a deflation transaction to rebind that level to read and write quorums of sizes $q_r[v',x]$ and $q_w[v',x]$. Here, too, quorum inflation and deflation is more flexible, since quorum sizes are not constrained by fixed read and write thresholds.

The principal disadvantage of the threshold scheme is similar to that of the virtual partition scheme: the cost and latency of view creation increases with system size. The threshold scheme aborts any transaction that attempts to read or write a version residing at a site belonging to a different view. A site joins a view by executing an *update transaction* that updates the local copy of each read accessible object x after reading $A_r[x]$ copies. Creating a new view requires more message traffic than creating a new virtual partition, since each of x's sites independently reads $A_r[x]$ copies, but individual update transactions will be smaller than virtual partition creation transactions if objects are partially replicated, since a site's update transaction ignores objects that have no copies at that site. Unlike deflation transactions, update transactions may require locking multiple replicated objects, and our earlier remarks about cost and latency apply here as well. Even in the extreme case where each site stores a version for only one object, x's new quorum assignment is fully installed only after every one of x's sites has completed its update transaction. Our simulation of the threshold scheme achieves the same effect by a deflation transaction that spans only $A_w[x]$ sites. If the view's quorums and thresholds are the same, our scheme does not need to deflate at all. In the threshold scheme, the minimum delay before a transaction can read (write) x in a new view v is the time needed to execute update transactions at $q_r[x,v]$ ($q_w[x,v]$) sites. In our scheme, the transaction can read or write immediately if it postpones deflation and uses read and write quorums of sizes $A_r[x]$ and $A_w[x]$.

7. Conclusions

This paper has proposed new techniques for managing replicated data in the presence of crashes and partitions. Like similar methods, we exploit the observation that transactions serialized "in the future" may use different quorum assignments from transactions serialized "in the past." We provide two basic techniques for quorum adjustment: quorum inflation shrinks write quorums, and quorum deflation shrinks read quorums. These two techniques can be combined to implement a wide range of adaptive strategies for enhancing the availability of replicated data, generalizing several techniques in the literature. This approach scales up effectively to large systems because quorum adjustment occurs locally at transactions and objects, and its effects propagate lazily through the system as circumstances require. Finally, this approach systematically exploits the semantics of typed objects to enhance the flexibility of quorum adjustment.

Acknowledgments

I am grateful to Joshua Bloch, Elliot Kolodner, Sharon Perl, and the anonymous referees for their comments and suggestions.

References

[1] P.A. Alsberg, and J.D. Day.
A principle for resilient sharing of distributed resources.
In *Proceedings, 2nd Annual Conference on Software Engineering*. 1976.
San Francisco, CA.

[2] P.A. Bernstein, and N. Goodman.
The failure and recovery problem for replicated databases.
In *Proceedings, 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. 1983.
Montreal, Quebec.

[3] P. Bernstein and N. Goodman.
An algorithm for concurrency control and recovery in replicated distributed databases.
ACM Transactions on Database Systems 9(4):596-615, December, 1984.

[4] A.D. Birrel, R. Levin, R. Needham, and M. Schroeder.
Grapevine: an exercise in distributed computing.
Communications of the ACM 25(14):260-274, April, 1982.

[5] J.J. Bloch, D.S. Daniels, and A.Z. Spector.
Weighted Voting for Directories: a Comprehensive Study.
Technical Report CMU-CS-84-114, Carnegie-Mellon University, April, 1984.

[6] B.A. Coan, B.M. Oki, and E.K. Kolodner.
Limitations on database availability when networks partition.
In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 187-194. 1986.
Calgary, Alberta.

[7] S.B. Davidson.
Optimism and consistency in partitioned distributed database systems.
ACM Transactions on Database Systems 9(3), September, 1984.

[8] S.B. Davidson, H. Garcia-Molina, and M.D. Skeen.
Consistency in a partitioned network: a survey.
ACM Computing Surveys 17(3):341-370, September, 1985.

[9] D.L. Eager and K.C. Sevcik.
Achieving robustness in distributed database systems.
ACM Transactions on Database Systems 8(3):354-381, September, 1983.

[10] A. El-Abbadi, D. Skeen, and F. Cristian.
An efficient, fault-tolerant protocol for replicated data management.
In *Conference on Principles of Database Systems.* ACM SIGACT/SIGMOD, December, 1985.

[11] A. El-Abbadi and S. Toueg.
Availability in Partitioned Replicated Databases.
Technical Report TR 85-721, Dept. of Computer Science, Cornell University, December, 1985.

[12] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.
The notion of consistency and predicate locks in a database system.
Communications of the ACM 19(11):624-633, November, 1976.

[13] M. Fischer and A. Michael.
Sacrificing serializability to attain high availability of data in an unreliable network.
In *Proceedings, ACM SIGACT-SIGMOD Symp. on Principles of Database Systems.* March, 1982.

[14] H. Garcia-Molina, T. Allen, B. Blaustein, R.M. Chilenskas, and D. Ries.
Data-Patch: integrating inconsistent copies of a database after a partition.
In *Proceedings, 3rd Symposium on Reliability in Distributed Software and Database Systems.* October, 1983.

[15] H. Garcia-Molina and D. Barbara.
How to assign votes in a distributed system.
Journal of the ACM 34(2):841-861, October, 1985.

[16] D. K. Gifford.
Weighted voting for replicated data.
In *Proceedings of the Seventh Symposium on Operating Systems Principles.* ACM SIGOPS, December, 1979.

[17] J. Gray.
Notes on database operating systems.
Lecture Notes in Computer Science 60.
Springer-Verlag, Berlin, 1978, pages 393-481.

[18] M.M. Hammer, and D.W. Shipman.
Reliability mechanisms in SDD-1, a system for distributed databases.
ACM Transactions on Database Systems 5(4):431-466, December, 1980.

[19] M.P. Herlihy.
Replication Methods for Abstract Data Types.
Technical Report MIT/LCS/TR-319, MIT Laboratory for Computer Science, May, 1984.
Ph.D. Thesis.

[20] M.P. Herlihy.
Availability vs. Atomicity: Concurrency Control for Replicated Data. .
Technical Report CMU-CS-85-108, Carnegie-Mellon University, February, 1985.

[21] M.P. Herlihy.
Comparing how atomicity mechanisms support replication.
In *Fourth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* August, 1985.

[22] M.P. Herlihy.
Using Type Information to Enhance the Availability of Partitioned Data..
Technical Report CMU-CS-85-119, Carnegie-Mellon University, April, 1985.

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

[24] W.H. Kohler.
A survey of techniques for synchronization and recovery in decentralized, computer systems.
ACM Computing Surveys 13(2):149-185, June, 1981.

[25] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7):558-565, July, 1978.

[26] C.H. Papadimitriou.
The serializability of concurrent database updates.
Journal of the ACM 26(4):631-653, October, 1979.

[27] G.J. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel.
Locus: a network transparent high reliability distributed system.
In *Proceedings, Eighth Symposium on Operating Systems Principles.* December, 1981.

[28] D.P. Reed.
Implementing atomic actions on decentralized data.
ACM Transactions on Computer Systems 1(1):3-23, February, 1983.

[29] M.D. Skeen.
Crash Recovery in a Distributed Database System.
PhD thesis, University of California, Berkeley, May, 1982.

[30] R.H. Thomas.
A majority consensus approach to concurrency control.
ACM Transactions on Database Systems 4(2):180-209, June, 1978.

[31] J.S.M. Verhofstad.
Recovery techniques for database systems.
ACM Computing Surveys 10(2):167-196, June, 1978.

[32] W.E. Weihl.
Specification and implementation of atomic data types.
Technical Report TR-314, M.I.T. Laboratory for Computer Science, March, 1984.
Ph.D. Thesis.

[33] D.D. Wright.
Managing Distributed Databases in Partitioned Networks.
Technical Report 83-572, Cornell University, September, 1983.