Reducing Storage for Quorum Consensus Algorithms

Divyakant Agrawal
Department of Computer Science
University of California
Santa Barbara, CA 93106

Amr El Abbadi
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

In this paper, we first develop a fragmentation method that reduces the storage overhead of replicated objects. We then present a data management protocol for these fragmented objects, and show that this protocol is a generalisation of quorum consensus algorithms for replicated data in which objects are not fragmented. Although this protocol reduces storage requirements, it does not achieve the same level of resiliency for both read and write operations. By integrating a log-based propagation mechanism with our protocol, we are able to achieve the same level of resiliency for both read and write operations as other quorum consensus protocols, while reducing the storage cost.

1 Introduction

In a distributed database system data is replicated to achieve fault-tolerance. One of the most important advantages of replication is that it masks and tolerates failures in the network gracefully. In particular, the system remains operational and available to the users despite failures. Another advantage is that recovery from catastrophic failures, such as a loss of storage media, becomes possible due to the presence of redundant information in the system. However, the increase in fault-tolerance in a replicated database has several underlying costs: storage and communication. In this paper, we present an algorithm that reduces the storage cost of replication, while achieving the same degree of data availability as previous replicated data management protocols [7].

We consider a distributed system consisting of a set of sites connected by a communication network. Sites communicate with each other through messages only. We assume that sites are either *fail-stop* [13], or may fail to send or receive messages. Communication links may fail by crashing, or by failing to deliver messages. Combinations of such failures may lead to *partitioning failures*[4], where sites in a *partition* may communicate with each other, but no communication can occur between sites in different partitions. We also assume that the failures in the systems are temporary. That is, a site does not fail permanently and the network does not remain partitioned forever.

Gifford [7] presents a simple quorum consensus protocol to manage replicated objects in a distributed environment that suffers from such failures. In this protocol an object may be read by reading a *read quorum* number of copies, and it

may be written by writing a *write quorum* number of copies. The restriction on the choice of quorum assignment is that the sum of the read and write quorums must exceed the total number of copies of the object in the system. Also, the size of write quorum must be such that the sum of two write quorums exceeds the total number of copies of the object.

Since Gifford's protocol does not require that write operations be executed at all copies of an object, it becomes necessary to be able to identify the current copy in any read or write quorum number of copies. This is achieved by associating a *version number* with each copy. The version number is updated every time a copy is modified. The copy with the largest version number is *current*. The new version number assigned to each copy is one more than the version associated with the current copy. The read and write quorums in this protocol are such that the read and write operations are always performed on a current copy.

The read and write quorums in [7] determine the number of copies that may become inaccessible without rendering the object unavailable for reading or writing. For example, consider an object x with $N[x]$ copies, and read quorum $Q_r[x]$ and write quorum $Q_w[x]$. A read operation can be executed even when $N[x] - Q_r[x]$ copies are inaccessible. Similarly, a write operation can be executed even when $N[x] - Q_w[x]$ copies are inaccessible. The main disadvantage of this protocol, however, is that in order to achieve such a degree of availability the object as a whole must be replicated at N sites, hence, requiring N times the storage of one copy.

Paris [12] presents an interesting protocol that addresses the problem of storage requirements in Gifford's protocol. Instead of storing an object x at $N[x]$ sites, this protocol stores the object at $m[x]$ sites ($m[x] \leq N[x]$) and stores *witnesses* at the remaining $N[x] - m[x]$ sites. A witness stores only a version number, and hence requires nominal storage. An analysis shows that when read and write quorums correspond to a majority of copies, and under very general assumptions, the reliability of a replicated object with $N[x]$ copies is the same as the reliability of an object with $m[x]$ copies and $N[x] - m[x]$ witnesses. A crucial assumption made in [12] to achieve this degree of reliability is the existence of a *repair process* that ensures the continuous availability of a valid quorum, *i.e.*, all copies residing on a site that has failed will be brought up-to-date when the site recovers. The repair process must be executed atomically at recovery time; otherwise certain timing and failure sequences may result in an available quorum that does not contain a current copy of the object.

In this paper we propose a protocol that ensures the same degree of data availability as that attained by Gifford's quorum consensus protocol, while, in general, requiring less storage. Our protocol reduces storage costs when write quorums are less than all copies of an object. In order to make write operations fault-tolerant, most systems satisfy this property, and therefore our protocol can be used to reduce storage costs. Furthermore, the protocol does not require any special recovery process to handle failures, and the updating of information on recovering sites is not subject to special timing constraints.

In the next section we present our fragmentation method for objects and describe a simple replicated data management protocol. We show that the protocol cannot attain the same level of read and write resiliency as [7], while reducing the storage requirements. In section 3 we extend this simple protocol to achieve the same level of

resiliency for read and write operations as that achieved by Gifford's protocol, while still requiring less storage. We conclude the paper with a discussion of our results.

2 A Simple Data Management Protocol

We consider a set of *sites* connected by bidirectional *links*. A *distributed database* consists of a set of *objects* that may reside at different sites. Users execute *transactions* that read and write the objects in the database. The execution of a transaction is *atomic*, *i.e.*, before a transaction terminates it either *commits* or *aborts* all changes it made to the database. We also assume that transaction execution is synchronized by an underlying concurrency control mechanism, e.g, two-phase locking protocol[5] or timestamp ordering protocol [2].

2.1 The Protocol

In a distributed system, a high level of data availability can be achieved by storing several copies of each object at different sites. The standard quorum consensus approach [7] would require replicating the object, *i.e.*, storing the whole object, at several sites. We say that the implementation of an object requires l *storage* if that implementation uses l times as much storage as a single copy of the object would use. Hence, the standard replication approach uses n storage, if an object is replicated at n sites. We now propose a different approach to distribute an object at n different sites. This approach requires m storage, where $m \leq n$. In this section we propose a simple protocol for managing a replicated object at n sites, where each site stores a fraction of the whole object, thus requiring low storage requirements.

Given an object x, we distribute it on $n[x]$ sites so that the overall storage used is $m[x] \leq n[x]$. This is done by dividing the object x into $n[x]$ *fragments* and storing them on $n[x]$ sites such that:

1. Each fragment is stored at $m[x]$ different sites.
2. Each site has $m[x]$ distinct fragments.

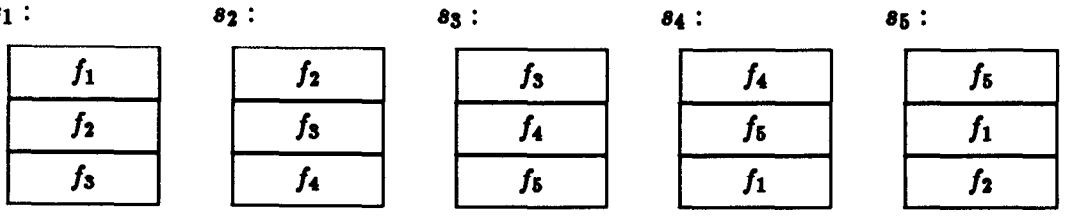
The $m[x]$ fragments of x stored at a site are called a *segment*. Since each fragment is replicated at $m[x]$ sites, the total storage used in our scheme is $m[x]$ storage. We define the *Full Copy Equivalent* ($FCE[x]$) of an object x to be the least number of segments necessary in the worst case to reconstruct the object, *i.e.*, the least number of segments containing all $n[x]$ distinct fragments of x. Since each fragment exists in $m[x]$ segments, any $n[x] - m[x] + 1$ segments must contain at least one copy of each fragment of x. Furthermore, if $FCE[x]$ is less than $n[x] - m[x] + 1$ then in the worst case $FCE[x]$ segments may not contain any copy of a particular fragment. Thus, the fragmentation technique satisfies the following property:

F1. For an object x, $FCE[x]$ is at least $n[x] - m[x] + 1$ distinct segments.

Figure 1 depicts one possible storage scheme for an object x with 3 copies at 5 sites, *i.e.*, $m[x] = 3$ and $n[x] = 5$. In this example, $FCE[x]$ is 3 segments. Note that the entire object could be constructed from two segments: s_1 and s_4; however, not any two segments would suffice, e.g., segments s_1 and s_2 do not contain fragment f_5. In contrast, any three segments would suffice to reconstruct the entire object x.

We associate with each segment a *version number*, which is initialized to 1, and with each object x, a *read quorum*, $q_r[x]$, and a *write quorum*, $q_w[x]$. A read operation, $r[x]$, is executed as follows:

1. Select $q_r[x]$ segments of x, and determine the



f_i: i^{th} fragment of the object s_j: j^{th} segment of the object

Figure 1: A storage scheme for an object x with 3 copies at 5 sites

maximum version number, vn_{max}, of the selected segments.

2. Read $FCE[x]$ segments with the version number vn_{max} to correctly construct the whole object x.

A write operation, $w[x]$, is executed as follows:

1. Select $q_w[x]$ segments of x and determine the maximum version number, vn_{max}, of the selected segments.

2. Write all fragments in the selected segments and update their version numbers to $vn_{max}+1$.

Read and write quorums must satisfy the following requirements:

$$n[x] - m[x] + 1 \leq q_r[x] \leq n[x] \quad (2.1)$$

$$\max\left(n[x] - m[x] + 1, \lceil \tfrac{n[x]+1}{2} \rceil\right) \leq q_w[x] \leq n[x] \quad (2.2)$$

$$n[x] + (n[x] - m[x] + 1) \leq q_r[x] + q_w[x] \leq 2 \cdot n[x] \quad (2.3)$$

Equation (2.1) captures the requirement that each read operation must access at least $FCE[x]$ segments, *i.e.*, at least $n[x] - m[x] + 1$ segments, otherwise some fragments of the object may not be in the read quorum (property F1), and hence, the object cannot be reconstructed completely.

Equation (2.2) places two restrictions on the lower bound of the write quorums. First, a write operation on object x must write at least $FCE[x]$ segments. Second, any two write operations of an object x must have a non-empty intersection, *i.e.*, there must be at least one segment written by both operations. This restriction is imposed because every write operation must assign a new version number greater than the version numbers assigned to any segment.[1]

Finally, Equation (2.3) imposes the restriction that for an object x, any two sets of sizes $q_r[x]$ and $q_w[x]$ must contain at least $FCE[x]$ segments in common. Since a read operation intersects with every write operation, it can determine the highest version number written. Furthermore, the entire object x can be constructed by using $FCE[x]$ segments with the highest version number. The following lemma formalizes these arguments and shows the necessity of the lower bound in Equation (2.3).

Lemma 1 For a read operation $r[x]$ to read the entire object x with the highest version number, $q_r[x]+q_w[x]$ must be greater than or equal to $n[x]+(n[x] - m[x] + 1)$.

Proof. If $q_r[x] + q_w[x] < n[x] + 1$, then in the worst case $r[x]$ may not access any segment with

[1]Equations (2.1) and (2.2) assume that operations must access the entire object. We are investigating possible optimizations that can be made when operations access a subset of the fragments of the object.

the highest version number. Hence, we only have to consider the case where:

$$n[x] < q_r[x] + q_w[x] < n[x] + (n[x] - m[x] + 1)$$

i.e., the case where a read and write operation have a non-empty intersection that does not contain $FCE[x]$ segments. In this case, $r[x]$ intersects with every write operation at least at one segment, and hence, $r[x]$ can calculate the value of the highest version number vn_{max}. Let $w_{max}[x]$ be the write operation that writes x with the version number vn_{max}. Let $q_r[x] + q_w[x] = n[x] + (n[x] - m[x] + 1) - 1$ then, in the worst case, operations $w_{max}[x]$ and $r[x]$ have at most $n[x] - m[x]$ segments in common. But from F1, this implies that $r[x]$ in the worst case may not be able to construct the entire object x with vn_{max}. Thus, $q_r[x] + q_w[x]$ must be greater that or equal to $n[x] + (n[x] - m[x] + 1)$. □

We next compare Equations (2.1), (2.2), and (2.3) with Gifford's Equations relating the read and write quorums of replicated objects. In Gifford's protocol, a read operation can be performed by accessing as few as one copy of an object. On the other hand, in our protocol, one segment does not represent the entire object, and hence from Equation (2.1) a read operation for an object x must access at least $FCE[x]$ segments. It must be noted, however, that in order to increase the availability of write operations in Gifford's protocol, the read quorum, in general, is chosen greater than one. We make use of this fact to reduce the storage cost of replication and will further address this issue in the next section.

A similar distinction exists for the write operations in the two protocols. In Gifford's protocol, a write operation can be executed with as few as $\lceil \frac{n[x]+1}{2} \rceil$ copies an object x, while our protocol requires the maximum of $n[x] - m[x] + 1$ and $\lceil \frac{n[x]+1}{2} \rceil$ segments. This restriction, however, can be made void by always choosing a value of $m[x]$ such that $\frac{n[x]}{2} < m[x] \leq n[x]$. Hence, Equation (2.2) requires that two write operations must have at least one segment in common, which is the same as in Gifford's protocol. This is due to the observation that in order to execute a write operation the current value of the object is not necessary, rather, the highest version number associated with any of its segments must be available.

Finally, in Gifford's protocol, a read and a write operation on an object x need only one copy in common. However, in our protocol, for a transaction to read an object x, it must be able to access at least $n[x] - m[x] + 1$ segments with the highest version number (Equation (2.3)). Note that for the purpose of correctness, *i.e.*, to ensure ***one-copy serializability*** [3], the sum of read and write quorum of $n[x]+1$ segments would have been sufficient. This is due to the fact that to ensure one-copy serializability, all our protocol has to guarantee is that two conflicting operations[2] must physically conflict on at least one segment. Since our protocol imposes stronger restrictions on read and write quorums, it must ensure one-copy serializability.

2.2 Resiliency of the Protocol

The simple protocol is a generalization of Gifford's quorum consensus protocol in which objects are not fragmented, or, equivalently, in which for object x, $m[x] = n[x]$. In this section we compare the levels of resiliency achieved by both protocols: we show that, in general, our protocol can achieve the same resiliency level for at least one operation at reduced storage cost. We start by formalizing

[2]Two operations conflict if they operate on the same object and at least one of them is a write operation.

the notion of resiliency as follows: an implementation of an object x has *read resiliency*, $R_r[x]$, if a read operation on x can be executed even after $R_r[x]$ segments are inaccessible due to site or partitioning failures. *Write resiliency*, $R_w[x]$, for an object x is defined similarly.

For purposes of comparison, let $Q_w[x]$ and $Q_r[x]$ be the read and write quorums associated with an object x according to Gifford's protocol, and let $N[x]$ be the total number of copies implementing object x. This implementation has a read resiliency $R_r[x] = N[x] - Q_r[x]$ and a write resiliency $R_w[x] = N[x] - Q_w[x]$. We now present two implementations using our protocol, one that achieves the same write resiliency as that achieved by Gifford's protocol (but a lower degree of read resiliency), and another that achieves the same read resiliency (but a lower write resiliency). Both implementations use less storage than that required by Gifford's protocol.

We implement our protocol using $n[x] = N[x]$ segments, and any value of $m[x]$ such that $\frac{n[x]}{2} < m[x] \leq n[x]$. The fragmentation approach can achieve the same write resiliency for write operations by assigning $q_w[x] = Q_w[x]$; since $n[x] = N[x]$, $q_w[x] > \frac{N[x]}{2} = \frac{n[x]}{2}$, and since $\frac{n[x]}{2} < m[x]$, $q_w[x] \geq n[x] - m[x] + 1$. Hence, our protocol allows the implementation of an object x using as low as half the storage requirements of Gifford's protocol, while achieving the same degree of write resiliency, and the same communication costs per write operations (since $q_w[x] = Q_w[x]$). Unfortunately, to achieve this degree of write resiliency and communication cost with less storage, read operations in our protocol become more expensive and less resilient to failures. More specifically, $q_r[x] = Q_r[x] + (n[x] - m[x])$, i.e., the read quorum has increased in size by $(n[x] - m[x])$, and hence the read resiliency has decreased by that amount too.

Our second implementation achieves the same read resiliency as Gifford's protocol while using less storage. However, this improved performance is at the expense of write operations. Let $n[x] = N[x]$ and $q_r[x] = Q_r[x]$. Since $q_r[x] \geq n[x] - m[x] + 1$, we require that $m[x] \geq n[x] - Q_r[x] + 1$. Hence, the greater the read quorum, the smaller $m[x]$ may be, thus achieving the same read resiliency while requiring less storage. However, write quorums are larger in size than in the corresponding implementation using Gifford's protocol, specifically, $q_w[x] = Q_w[x] + (n[x] - m[x])$, *i.e.*, the write quorum has increased in size by $(n[x] - m[x])$, thus lowering the write resiliency of x.

In conclusion, we note that since the read and write quorum for an object x must contain $FCE[x]$ segments, *i.e.*, $n[x] - m[x] + 1$ segments, both read and write operations cannot achieve the same degree of resiliency and communication cost as Gifford's protocol using less storage. In the next section we present a special mechanism that overcomes this problem, and then we show that we can attain comparable cost and resiliency performance to Gifford's protocol using less storage.

3 A Modified Data Management Protocol

In the previous section we showed that in order to reduce the storage cost of a replicated object x from $n[x]$ copies to $m[x]$ copies, the size of the intersection between read and write operations increases from one copy of x to $n[x] - m[x] + 1$ segments of x. The larger size of intersection between read and write operations results in increased communication costs for read and/or write operations. In this section we provide an underlying mechanism, which ensures that the information written by a write operation on an object is eventually propagated to all segments of the object

in the system. Although all segments of an object x are updated as a result of a write operation on x, this does not mean that $q_w[x]$ in this protocol is $n[x]$. By using this underlying mechanism, we will show how to decrease the size of intesection between read and write operations from $n[x] - m[x] + 1$ segments to one segment while maintaining the reduced level of storage for x. First, we describe the underlying mechanism to propagate write operations to all segments of an object. Next, we explain how to integrate our protocol with the propagation mechanism. Finally, we compare our modified protocol with Gifford's protocol and demonstrate that we achieve the same level of resiliency and communication cost for read and write operations in our protocol.

3.1 The Propagation Mechanism

A common technique to propagate information efficiently in a network and, thus, synchronize various components of a distributed application is to construct a *log* of certain application specific events that have occurred in the network [14]. In the case of a replicated database such events include reading or writing a copy of an object at the coordinator site of a transaction. Each site maintains a local copy of the log, which is organized as an ordered sequence of event records, and a propagation mechanism is employed to keep the copies of the log up-to-date. The mechanism makes use of communication operations, *send* and *receive*, to exchange portions of the copies of the log for this purpose [6,15,9,11,8]. The background messages used in the propagation mechanism to bring all the copies of the log up-to-date are also referred to as gossip messages in [11]. We have chosen the algorithm proposed by [15] to integrate the propagation mechanism with our protocol.

The algorithm described by Wuu and Bernstein [15] is an efficient implementation of the propagation mechanism. Each site, S_i, maintains a time-table, T_i, which is an $N \times N$ array of timestamps of events that have occurred in the network, where N is the total number of sites. A site uses the time-table to place a bound on how out-of-date other sites are about events that have happened in the network. The time-table allows a site to decide what portion of its copy of the log it should send to another site, and when all sites have learned about a particular event. This information is used by a site to determine when certain portions of its copy of the log can be discarded. Hence a site retains a particular event record in its copy of the log only if it is not certain that all other sites have learned of that event. The *happened before relation*, "$\rightarrow$" [10], relates the application specific events and the communication operations employed by the propagation mechanism. Periodically a site sends its time-table and a portion of its copy of the log to another site. On receiving such a message a site updates its copy of the log by including event records of which it was unaware and updates its time-table using information in the received time-table. The following two properties are guaranteed by the algorithm:

P1. Every site eventually learns of each event.

P2. If e_1 and e_2 are two events such that $e_1 \rightarrow e_2$, then if a site knows of e_2, it must also know of e_1.

P1 is dependent on the assumption that site failures and network partitions are not permanent. It follows from P2 that a site can process events in the *happened-before* order.

It must be noted that in the model of the system discussed above, all communication among sites is performed implicitly by exchanging the copies of

logs among the sites. That is, explicit communication operations, *send* and *receive*, are not available to application programs. Instead, an application program relies on the underlying propagation mechanism to inform other sites about its operation request (for example, a find operation on a distributed dictionary). The responses of other sites (for example, the results of a find operation on a distributed dictionary) are also communicated to the application program through the log. Although the propagation mechanism has an overhead of maintaining copies of the log, it has several advantages that offset this extra overhead: it can be easily implemented in an unreliable network, and the number of messages in the system can be reduced at the expense of the size of messages. Furthermore, the size of the copies of the log is bounded since sites discard event records from their copies as soon as they discover that all sites have learned about the events corresponding to these event records. Several optimizations have been proposed in [15] that reduce the overhead associated with this mechanism.

3.2 The Integrated Protocol

We now integrate the propagation mechanism introduced in the previous subsection with the execution of read and write operations of transactions. The fragmentation approach described earlier is used to store the segments of objects at different sites. The model of the system remains the same as that developed in Section 2.1 except for the distinction that application programs, transactions in our case, do not explicitly communicate with *remote* sites in the system. All communication is achieved by modeling operations and the results of the operations as events in the system and then exchanging the copies of the log among the sites. The site where a transaction originates is designated as the *coordinator* of the transaction. Read and write operations of a transaction are recorded as *events* in the copy of the log at the coordinator; other sites in the network learn about these events as a result of the propagation. Furthermore, sites agreeing to be in the quorum of an operation do not communicate explicitly with the coordinator. Instead, their decision to be in the quorum is also recorded as an *event* in their copy of the log; they too rely on the underlying communication operations to propagate these events to the coordinator.

We associate with each object x, a *read quorum*, $q_r[x]$, and a *write quorum*, $q_w[x]$. A read operation, $r[x]$, in this protocol is executed as follows:

1. A read operation, $r[x]$, results in an event, $r[x]$-*event*, at the coordinator. An $r[x]$-*event* record is placed in the coordinator's copy of the log. When the coordinator's copy of the log is propagated to other sites, the effect is the same as the transaction sending a read request to other sites in the system.

2. When a site, S, learns of $r[x]$-*event* and decides[3] to be in the quorum for $r[x]$, an event, $ok_S(r[x])$-*event*, occurs at that site. The event record corresponding to $ok_S(r[x])$-*event* in the copy of the log includes the value of the segment of x at that site. When the site's copy of the log is eventually propagated to the coordinator, the effect is the same as the transaction receiving a reply to the read request from a site in the quorum.

3. The operation, $r[x]$, is not completed until the coordinator can determine that $q_r[x]$ segments of x have been accessed. The events,

[3]This decision is based on the concurrency control mechanism employed.

ok$_S(r[x])$-*event*, are observed at the coordinator for this purpose. After accessing $q_r[x]$ segments of x, the coordinator returns the value of x to the requesting transaction.

A write operation, $w[x]$, is executed as follows:

1. A write operation, $w[x]$, results in an event, $v[x]$-*event*, at the coordinator. A $v[x]$-*event* record is placed in the coordinator's copy of the log. When the coordinator's copy of the log is propagated to other sites, the effect is same as the transaction sending a version request to other sites in the system.

2. When a site, S, learns of a $v[x]$-*event* and decides to be in the quorum for $w[x]$, an event, *ok*$_S(v[x])$-*event*, occurs at that site. The event record corresponding to *ok*$_S(v[x])$-*event* in the copy of the log includes the version of the segment of x at that site. When the site's copy of the log is eventually propagated to the coordinator, the effect is same as the transaction receiving a reply to the version request from a site in the quorum.

3. The operation, $w[x]$, is not completed until the coordinator can determine that $q_w[x]$ segments of x have been accessed. The events, *ok*$_S(v[x])$-*event*, are observed by the coordinator for this purpose. The operation, $w[x]$, is completed when an event, $w[x]$-*event*, occurs at the coordinator. A $w[x]$-*event* record is placed in the coordinator's copy of the log, and the event record includes the new value of x and its version number that will be used to update the segments of x at various sites in the network. When the coordinator's copy of the log is propagated to other sites, the effect is same as the transaction executing a write at other sites. Note that the concurrency control mechanism ensures no other transactions can access x until the transaction executing $w[x]$ commits or aborts at its quorum.

The modified protocol must satisfy the following requirements:

$$n[x] - m[x] + 1 \leq q_r[x] \leq n[x] \quad (3.1)$$

$$\max\left(n[x] - m[x] + 1, \lceil \tfrac{n[x]+1}{2} \rceil\right) \leq q_w[x] \leq n[x] \quad (3.2)$$

$$n[x] + 1 \leq q_r[x] + q_w[x] \leq 2 \cdot n[x] \quad (3.3)$$

Equations (3.1) and (3.2) are the same as Equations (2.1) and (2.2) from the previous section; this is due to the identical considerations. On the other hand, Equation (3.3) is different and states that the read and write quorum intersection of one segment is sufficient. This is a significant improvement from the previous section where it was required that a read and a write quorum for an object x must have an intersection of $n[x] - m[x] + 1$ segments. This is due to the fact that the event record for $w[x]$ in the copy of the log at a site contains the entire information about x, and not only the information concerning the segment of x at that site. Since $q_r[x] + q_w[x] \geq n[x] + 1$, there will always be at least one segment with the highest version number in $q_r[x]$ corresponding to some write operation $w_{max}[x]$. If $r[x]$ collects $n[x] - m[x] + 1$ segments or more with the highest version number, it can reconstruct the entire object x. In the case when $r[x]$ collects fewer than $n[x] - m[x] + 1$ segments with the highest version number, then $w_{max}[x]$-*event* has not been propagated to all segments of x in the network. Since an event record is discarded by a site only when all sites learn about the event, therefore $w_{max}[x]$-*event* record still exists in the log. Hence, the coordinator can always construct the entire object x by using its copy of the log. This is formally proved in the following lemma.

Lemma 2 For a read operation, $r[x]$, to read the current value of an object x, it is sufficient that

$q_r[x] + q_w[x]$ be greater than or equal to $n[x] + 1$.

Proof. Since $q_r[x] + q_w[x] \geq n[x] + 1$, therefore $r[x]$ intersects with every write operation at least at one segment, and hence, can determine the highest version number vn_{max}. Let $w_{max}[x]$ be the write operation that writes x with version number vn_{max}. Since $q_r[x] > n[x] - m[x] + 1$, there are two possible cases. If $r[x]$ collects $n[x] - m[x] + 1$ segments with vn_{max}, it can construct current value of x. Otherwise, $r[x]$ collects fewer than $n[x] - m[x] + 1$ segments with vn_{max} and $w_{max}[x]$-*event* must exist in the copies of the log at the sites that have the segments of x with vn_{max} (This is from [15] in which an event record is discarded by a site if it can determine that all sites have learned about the event). At least one of these sites, S, with vn_{max} must be in $q_r[x]$. Since at S, $r[x]$ will read from $w_{max}[x]$, then:

$$w_{max}[x]\text{-}event \rightarrow r[x]\text{-}event \rightarrow ok_S(r[x])\text{-}event$$

Hence, when the coordinator of $r[x]$ learns about $ok_S(r[x])$-*event*, from P2, it must also become aware of $w_{max}[x]$-*event*. Thus, the coordinator will always return the current value of x. □

Although we have described the protocol in which $ok_S(r[x])$-*event* includes the value of the segment of x at S, this is not necessary. A possible optimization is not to include the values in the events. Instead, the coordinator (after collecting the required number of $ok_S(r[x])$-*event*) can explicitly read the value of the segments from the respective sites in the quorum. This will result in a significant reduction in the size of the log. An extension of this optimization is related to query processing. If objects are fragmented in such a way that queries reference objects on a fragment basis, the coordinator can reduce communication by executing a query at the site where the up-to-date fragment resides.

3.3 Resiliency of the Protocol

The quorum intersection requirement in our protocol is identical to that in Gifford, and therefore, we achieve the same level of resiliency with m copies that is achieved by Gifford with n copies. However, the range of assignment for read and write quorums in our protocol is narrower than the ranges in the Gifford's scheme. As in Section 2.2, let $Q_w[x]$ and $Q_r[x]$ be the read and write quorums associated with an object x according to Gifford's protocol, and let $N[x]$ be the total number of copies implementing object x. This implementation has a read resiliency $R_r[x] = N[x] - Q_r[x]$ and a write resiliency $R_w[x] = N[x] - Q_w[x]$. In our implementation, we use $n[x] = N[x]$, $q_r[x] = Q_r[x]$, and $q_w[x] = Q_w[x]$. We choose $m[x]$ such that:

$$n[x] - m[x] + 1 \leq q_r[x] \leq n[x] \quad (3.1)$$

$$\max\left(n[x] - m[x] + 1, \lceil \tfrac{n[x]+1}{2} \rceil\right) \leq q_w[x] \leq n[x] \quad (3.2)$$

If we choose a value of $m[x]$ such that $\frac{n[x]}{2} < m[x] \leq n[x]$ then Equation (3.2) reduces to:

$$\lceil \tfrac{n[x]+1}{2} \rceil \leq q_w[x] \leq n[x] \quad (3.2)$$

which imposes the same restriction on write operations as in Gifford's protocol. Thus the only restriction imposed by our scheme is that the read quorum must be greater than or equal to $n[x] - m[x] + 1$. This implies that the range of read quorum assignment is restricted at the lower end in our protocol. However, this is not a serious shortcoming, since any fault-tolerant implementation of an object x will rarely use the lower end of the read quorum assignments (for write operations to tolerate the failures of t copies, the read quorum must be greater than t).

We now show how to reduce storage for a given

read and write resiliency constraint so that it is minimal with respect to the fragmentation approach. Given $q_r[x]$ and $q_w[x]$ for an object x corresponding to the desired level of resiliency, Equation (3.3) requires that:

$$q_r[x] \geq n[x] + 1 - q_w[x]$$

But from Equation (3.1), we know that $q_r[x] \geq n[x] - m[x] + 1$. Thus, in order to minimize storage (in our protocol) and still satisfy the resiliency constraints, we must choose:
$q_r[x] = n[x] - m[x] + 1$, and $q_r[x] = n[x] + 1 - q_w[x]$. Hence, $m[x] = q_w[x]$. This result indicates that for any quorum assignment other than read-one/write-all approach, we can reduce storage in our protocol and still maintain the same level of availability as in other quorum consensus algorithms.

4 Conclusion

In this paper, we presented a simple and storage efficient protocol for managing replicated data. We first developed a fragmentation approach to reduce storage requirements in a replicated environment, and described a generalization of Gifford's protocol to maintain data in this approach. However, this protocol can not achieve the same level of resiliency for both read and write operations as Gifford's protocol, while reducing the storage. In order to overcome this drawback, we integrated a propagation technique proposed by Wuu and Bernstein [15] with our protocol. This propagation mechanism does have an extra storage overhead of maintaining the copies of the log in the system. In particular, when an object is updated, the storage requirements of the copies of the log increase temporarily. We are currently developing a model to analyze the storage behavior of our protocol and compare it with other replicated data management protocols. We would also like to integrate our fragmentation method with other propagation mechanisms[9,11,8], which may be more suitable for different environments. In this paper, we demonstrated that for any read quorum greater than one, our protocol can reduce the storage while maintaining the same resiliency for read and write operations. In most fault tolerant systems based on the quorum approach, the write quorum is generally less than all copies of an object, and hence the read quorum is greater than one. Such fault tolerant systems can benefit from the approach proposed in this paper to reduce storage. Furthermore in [1], it has been argued that the optimal read and write quorums are majority assignments. Given such an assignment of read and write quorums, we can reduce the storage cost by as much as half of that used in Gifford's protocol and still achieve the same level of resiliency and communication cost for both read and write operations.

Acknowledgements

We would like to thank John Bruno and Abdelsalam Heddaya for suggesting improvements to an earlier draft of the paper. We are also grateful to the anonymous refrees for their comments on the paper.

References

[1] Ahamad M., Ammar M. H., "Performance Characterization of Quorum Consensus Algorithms for Replicated Data", Proceedings of Sixth Symposium on Reliability in Distributed Software and Database Systems", pp 161-168, 1987.

[2] Bernstein P. A., Goodman N., "Concurrency Control in Distributed Database Systems",

ACM Computing Surveys, Vol. 13, No. 2, pp 185-221, June 1981.

[3] Bernstein P. A., Goodman N., "The Failure and Recovery Problem for Replicated Databases", Proceedings of the Second Annual ACM Symposium of Principles of Distributed Computing, August, 1983.

[4] Davidson S., Garcia-Molina, H., and Skeen, D., "Consistency in Partitioned Networks", ACM Computing Surveys, Vol. 17, No. 3, pp 341-370, September 1985.

[5] Eswaran K. P., Gray J. N., Lorie R. A., Traiger I. L., "The Notion of Consistency and Predicate Locks in Database System" Communications of the ACM 11, pp 624-633, 1976.

[6] Fischer M. J., Michael A., "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network", ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1982.

[7] Gifford D. K., "Weighted Voting for Replicated Data", Proceedings of the Seventh Symposium on Operating Systems Principles, December 1979.

[8] Heddaya A., Hsu M., Weihl W., "Two Phase Gossip: Managing Distributed Event Histories", *To Appear* in Information Sciences: An International Journal, 1988.

[9] Joseph T. A., Birman K. P., "Low Cost management of Replicated Data in Fault-Tolerant Distributed Systems", ACM Transactions on Computer Systems, Vol. 4, No.1, February 1986.

[10] Lamport L., "Time, Clocks, and the Ordering of events in a Distributed System", Communication of the ACM, July 1978.

[11] Liskov B., R. Ladin, "Highly Available Services in Distributed Systems", Proceedings of the Fifth ACM Symposium on Priniciples of Distributed Computing, August 1986.

[12] Paris J. F., "Voting with Witnesses: A Consistency Scheme for Replicated Files", Proceedings of the 6th Conference on Distributed Computing Systems, 1986.

[13] Schlicting R., Schneider F. B., "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems", ACM Transactions on Computing Systems, Vol. 1, No. 3, pp 222-238, August 1983.

[14] Schneider F. B., "Synchronization in Distributed Programs", ACM Transactions on Programming Languages and Systems, April 1982.

[15] Wuu G. T. J., Bernstein A. J., "Efficient Solutions to he Replicated Log and Dictionary Problems", Proceedings of the Third ACM Symposium on Priniciples of Distributed Computing, August 1984.