Efficient Support for Partial Write Operations in Replicated Databases *

Michael Rabinovich and Edward D. Lazowska

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

*In this paper, we present a new replica control technique targeted at replicated systems in which write operations update a portion of the information in the data item rather than replacing it entirely. The existing protocols capable of supporting partial writes must either perform the writes on all accessible replicas of the data item, or always apply the writes to the same group (*quorum set*) of replicas. In the former case, the system incurs high message overhead. In the latter case, if any of the replicas in this group fail, additional replicas must be synchronously brought up-to-date during the write operation causing delay to the operation. Also, in both cases, the system loses the advantage of load sharing provided by replication. Our protocol avoids performing the write on all nodes while preserving load sharing and reducing greatly the risk of having to propagate updates synchronously.*

The paper describes the protocol, proves it correct, and presents a comparative performance study of our protocol and the existing alternatives.

1 Introduction

Distributed systems often keep multiple copies of data to improve their resiliency to node and communication failures. Numerous algorithms have been proposed for managing replicated data. The consistency guarantees provided as well as the costs incurred by these protocols vary widely. In this paper, we concentrate on *quorum-based protocols*, which require that, in order to succeed, read and write operations must obtain permission from certain sets of nodes called read and write quorums. Quorums are defined in such a way that any two write quorums as well as any read and write quorums have at least one node in common. Then, a read operation is guaranteed to see at least one most recent version of the data, and no two write operations can succeed simultaneously thus excluding the possibility of write conflicts [7]. Quorum-based protocols guarantee consistency despite both node failures and network partitionings. This is unlike *available copies protocols* [2], which provide better availability than quorum-based protocols but do not guarantee consistency in the presence of network partitionings.

Because the coordinator (the node that initiated the operation) must communicate with multiple nodes before returning to the user, there is some inherent performance penalty incurred, especially in the case of write operations. The degree of the performance penalty depends on the semantics of write operations in the system. If the write always replaces the data item with an entirely new value, then the protocol does not have to worry about how current the version being replaced is. Hence, in the common case of absence of failures, the write coordinator can communicate only with nodes from an arbitrary write quorum. The coordinator will replace the data item with a new version on all nodes from the quorum therefore guaranteeing that a subsequent read will see the latest version of the data (if successful). This scheme allows good load sharing (because requests from different coordinators can be served by nodes from different quorums) and light network traffic. The performance of these systems depends on the size of read and write quorums, which motivated a search for small quorums [1, 4, 10].

Unfortunately, in information systems, it is often impractical to manage replication at such a small data granularity that all modifications of data items become total. Instead, operations in these systems update just a portion of the information in the data item. For example, in object-oriented databases, since

*This work is supported by the National Science Foundation under Grant No. CCR-9200832.

objects are typically large, and operations associated with them are typically complex [3], it would be prohibitively expensive to break the object into many components and execute a replica control protocol every time the operation accesses a new component, rather than once for the operation. Also, in many existing replicated file systems, (e.g., [15], [8]), to save on management and replica control costs, replication is maintained at the granularity of files or even larger units of data, so these systems have partial writes semantics as well.

In systems with partial writes, a write can be applied to current replicas only. Therefore, it was thought, the write coordinator must collect permission from a write quorum of *current* replicas in order to perform the write [6]. This implies that the coordinator must either apply the write to *all* operational nodes having the replica, or always perform all writes on the same write quorum of nodes. In the former case, the system incurs high message overhead. In the latter case, if any of the replicas in this quorum fail, additional replicas must be synchronously brought up-to-date during the write operation, which can have a significant impact on response time. Also, in both cases, the system loses an advantage of load sharing provided by replication.

The protocol proposed in this paper supports partial writes while preserving load sharing and reducing greatly the risk of having to propagate updates synchronously. The protocol uses the *stale data flag* introduced in our previous work [14]. It was observed there that while the requirement that the write operation changes the state of at least a write quorum of replicas is fundamental for the protocol to be correct, it does not mean that the write itself must be applied to all these replicas. Instead, the coordinator can apply the write to the current replicas only, while marking the other replicas from the write quorum stale. Then, a subsequent operation (if successful in obtaining responses from a quorum) is guaranteed to see either a current replica or a replica marked "stale" and thus identify the current replicas correctly.

The distinction between good and stale replicas allows different operations to use different quorums while still guaranteeing strict consistency. However, in the protocol of [14], if an operation fails to contact a current replica after obtaining a quorum of permissions, it must perform extra rounds of message exchange with additional replicas in an attempt to contact current replicas, thus incurring an additional delay even in the absence of failures. Also in that protocol, there may be a vulnerability window during which the current data is stored on a single replica.

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

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

There is also a way to guard against the case where all current replicas become partitioned from the rest of the system. It involves making good replicas keep track of the total number of good replicas in the system. (Originally, the number of good replicas can be recorded by the write operation; then, a good replica would increment the good replica counter whenever it propagates its data to any stale replica, and stale replicas would obtain this counter along with other state from the source replica during propagation.) Then, the nodes in a partition can collectively decide whether the partition includes all current replicas in the system, in which case the partition forms a new *epoch* [14]. (In the epoch-based, or dynamic protocols (for example, see [9]), the read and write quorums are determined dynamically based on the latest epoch; it is guaranteed that only operations that run in the latest epoch can succeed.) This condition for forming a new epoch can be added to the existing "write quorum containment" criterion, so that the resulting criterion says that a new epoch can be formed in a partition if it either contains a write quorum from the existing epoch and at least one current replica, or it contains all current replicas.

In the next section, we list the assumptions and definitions used in this paper. Section 3 describes the protocol, some properties of which are given in section 4. Section 5 gives the performance comparison of our protocol with the existing alternatives. We summarize our results in section 6.

2 Model and terminology

The protocol presented here suits both the client-server model of distributed computation and the model in which operation coordinators are among the nodes having a replica of the data item. We refer to nodes that can initiate operations as "potential coordinators", while "node", by default, will mean "a node having a replica". All considerations below apply on a per-data-item basis, so we will often use shortcuts and say "all nodes" instead "all nodes that have a replica of the same data item", "all writes" instead "all writes applied to the same data item", and so on.

We assume that write operations update only a portion of information in the corresponding data items. Hence, a write can be applied to current replicas only. We assume RPC-style communication in which the notification `RPC.CallFailed` is returned to the sender if the message cannot be delivered. Nodes and communication links are *fail-stop*, that is, they fail by crashing and do not behave maliciously. The protocol as described here does not address the deadlock problem. For ways to handle deadlocks that can easily be integrated with our protocol, see for example [2]. We assume that a commit protocol [5] that ensures all-or-nothing execution of transactions begins at the end of the transaction's execution: the systems with "early prepare" commit protocols [16] are expressly excluded[1].

We use one-copy serializability as the criterion for consistency: the system is consistent if the concurrent execution of operations on replicated data is equivalent to a serial execution of those operations on non-replicated data [2]. It can be shown that in the case of systems with partial writes, this criterion is satisfied if (a) neither two write operations nor read and write operations can be performed concurrently, and (b) a write is always applied to the most recent replicas of the data and a read always returns the most recent version of the data.

Finally, we give the formal definition of quorums [7]. Let V be a set of nodes that have a replica of the data item. A coterie over V is a pair of sets $W = \{w_1, \ldots, w_k\}$ and $R = \{r_1, \ldots, r_l\}$ such that their elements are subsets of V; $w_i \cap w_j \neq \emptyset$; $r_s \cap w_j \neq \emptyset$; $w_i \not\subseteq w_j$; $r_s \not\subseteq r_t$ $(1 \leq i, j \leq k; 1 \leq s, t \leq l)$. W is called a write coterie and R is called a read coterie. The elements of W and R are called write and read quorums over V.

3 The protocol

Each node maintains the following state: a version number, a stale-data flag, and a *good replica list* which is the list of current replicas found during the last write operation. The meaning of the version number depends on the state of the stale-data flag. For replicas marked "current", it is equal to the number of write operations applied to the replica. For replicas marked "stale", it stands for the *desired version number*, i.e. the maximum version number in the system the replica knows of. Also, each replica maintains three separate locks. The *read and write permission locks* ensure that the replica does not issue permission to multiple writes or a read and a write at the same time. The *commit lock* is used by the atomic commit protocol. Figure 1 gives the compatibility table for these locks. Note that a replica is allowed to participate in the commit protocol even if it did not issue permission to the operation, or even if it issued permission to a different operation. This not only eliminates some rounds of message exchange, but (as we will see in section 5) also increases concurrency in the system. Originally, all nodes have identical replicas with version numbers all 0, their good replica lists include all nodes, and all replicas are marked "current". There is a system parameter `SAFETY-THRESHOLD` that defines the number of simultaneous node failures the protocol is supposed to tolerate.

The algorithm for the write operation performs the following steps.

1. The coordinator requests permission from a randomly chosen write quorum of nodes. Each node that received the request obtains the write permission lock for its replica and responds with its state. If the coordinator receives all non-`RPC-CallFailed` responses

[1] In these systems, replicas enter the *prepared* state of the commit protocol at the time when they issue permission for the operation. Then, at commit time, one less round of message exchange is required. However, these systems significantly extend the "vulnerability period" during which replicas are in the prepared state and hence may be blocked for accesses indefinitely if the coordinator fails.

	Read perm	Write perm.	Commit
Read perm.	yes	no	yes
Write perm.	no	no	yes
Commit	yes	yes	no

Figure 1: The compatibility table of replica locks.

(which means it has collected a write quorum of permissions; this is the common case of the absence of failures), it returns control to the application and, in the background, executes step 3.

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

3. Once the coordinator has the state information from a write quorum of replicas, it is able to identify at least some current replicas in the system (these are the members of the good replica list from any response with the maximum version number; good replica lists from all such responses are identical and will be called the current good replica list below; if `SAFETY-THRESHOLD` is not greater than the size of any write quorum, there are at least `SAFETY-THRESHOLD` members in the current good replica list)[2]. The coordinator also can tell which of the responses it received came from the current replicas (these are the responses with the maximal version number marked "current", or the ones from the members of the current good replica list). Then, the coordinator constructs a `good-list` which includes all current replicas contacted so far plus additional replicas from the current good replica list as needed for the `good-list` to contain at least `SAFETY-THRESHOLD` members. It also constructs a `stale-list` which includes all non-current replicas from the write quorum contacted. The maximal reported version number will be denoted `max-version` below.

[2]The statements in parentheses are proven in the next section.

At transaction commit time, the following steps are executed for every data item written by the transaction.

4. Let `quorum-list` be a set of replicas that are members of the quorum collected in step 1 or 2. The coordinator sends a "prepare" message to replicas from `good-list`$\cup$`stale-list` (note that this set includes `quorum-list`). Each replica that receives this message obtains its commit lock and responds with a prepare acknowledgement, which also includes the replica's version number and stale data flag. If the coordinator receives all acknowledgements (the common case), it executes step 6.

5. If the coordinator failed to receive some acknowledgements (i.e., it received `RPC.CallFailed` instead), its actions depend on the identities of the failed replicas. If any of the members of `quorum-list` failed to acknowledge the prepare message, the transaction is aborted. Otherwise (i.e., the failed replicas are among the additional current replicas included in step 3), the prepare message is sent to all remaining replicas. Once the prepare acknowledgement is received from a total of `SAFETY-THRESHOLD` current replicas (the ones whose acknowledgements carry version number `max-version` and stale data flag "current"), the protocol moves to step 6, with the rest of acknowledgements received and dismissed in the background. (The coordinator sends a "prepare dismissed" message to these extra replicas so that they can release their commit lock.)

If, upon receiving all responses to its prepare message, the coordinator has received the acknowledgement from some current replicas, but their number is less than `SAFETY-THRESHOLD`, it brings necessary number of stale replicas up-to-date using one of the current replicas as the source of propagation, and then moves to step 6.

Finally, if no current replicas acknowledged the prepare message, or the above update propagation failed, the transaction is aborted.

6. The coordinator updates its `good-list` and `stale-list` based on the version number and stale data flag from the prepare acknowledgements received. Replicas that reported version number `max-version` and stale data flag "current" are put into `good-list`, and the rest are included into `stale-list`. Updating these lists is required, first of all, to exclude replicas

that were added to good-list in step 3 but failed to acknowledge the prepare message. Also, additional current replicas found in step 5 can be included into good-list, and some replicas that were found stale in step 3 may have become current due to asynchronous update propagation.

Finally, the coordinator sends the commit message to good-list∪stale-list. With this message, it includes good-list, stale-list, new version number equal to max-version + 1, and the update data. Upon receiving this message, nodes that find themselves among good-list, increment their version number and install the update data. Replicas that are members of stale-list mark themselves "stale" and adopt the new version number received. Besides, every replica records good-list as its new good replica list, releases its commit lock, and, if the current transaction also holds its permission lock, releases this lock as well. (Note that the permission lock may be held by another transaction, in which case it remains locked.)

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

The read protocol is similar to the write protocol. During the first phase, the coordinator identifies the current replicas by contacting a read quorum of nodes and learning the maximum version number among responses. The current replicas are the ones marked "current" and with the maximum version number. Also current are all members of the good replica list from any response with maximum version number. In the second phase, the coordinator reads the data from one of the current replicas [3].

The good replica lists allow a very efficient asynchronous update propagation. The initiators are the stale nodes which use their good replica lists to locate provably more recent replicas.

First, the initiator of the propagation tries to contact any replica from its good replica list. If successful, the initiator obtains the version number, stale data flag and good replica list from the node contacted. If the source replica is marked "current", the initiator also obtains the missing updates so that its replica becomes identical to the source replica. If the newly obtained stale data flag is "current", the propagation is done. Otherwise, the algorithm is run again with the new good replica list.

If no node from the good replica list is available (they all fail or partition), then the initiator tries to contact any other node with a more recent replica than its own. (The hope is that some node might have done the propagation before all members of the initiator's good replica list failed). The more recent replicas are the ones with greater version numbers or the ones with the same version numbers but marked "current". As before, the propagation involves obtaining the new state if the source node is marked "stale", and both the new state and the missing updates if the source is marked "current". As soon as the initiator succeeds in obtaining a newer state, it goes back to the "pointed" search for a current replica using its new good replica list.

Note that the source and destination nodes do not need to hold any locks during their communication. All that is needed is that the source replica send and the destination replica install a consistent snapshot of the data and the control state.

[3] If the nodes from the read quorum contacted that are marked "current" would respond not only with their state but with the requested data as well, the read operation could be done in one phase in most cases. The design choice depends on the specifics of a particular system.

4 Properties of the protocol

We prove two useful lemmas first and then give a few important properties of the protocol.

Lemma 1. No two write operations nor a read and a write operations can can obtain their quorum of permissions at the same time: one must unlock a node before another can complete its quorum.
Proof. Follows directly from the definition of quorums, which says that any two write quorums as well as any read and write quorum have at least one node in common. □

The consequence of Lemma 1 is that we can assign sequential numbers to successful write operations according to the order in which they obtain their quorums of permission. In particular, one can use induction on these numbers. Below, we refer to the write quorum used by the i-th write operation as the i-th write quorum, and to the good replica list constructed by the i-th write operation during step 6 of the protocol as the i-th good replica list.

We will say that a write operation has performed if it released the write permission lock on at least one node from its quorum. (A write releases the permission lock on nodes from its quorum during step 6 of the protocol by sending them the commit message.) We will say a node has performed a write if it received and processed the commit message from the coordinator of that write operation.

Lemma 2. In the system state after the m-th write performed and before the $(m + 1)$-st write obtained its quorum, the following is true: (1) all nodes that performed the m-th write have version number m; (2) m is the maximum version number among the replicas in the system; (3) all nodes with version number m and stale data flag "current" are indeed current; (4) all nodes with version number m have the m-th good replica list; and (5) all nodes from the m-th good replica list that performed the m-th write have their stale data flag "current" (and hence, are current due to conditions (1) and (3)); all other nodes from the m-th good replica list have their commit lock acquired by the m-th write.

Proof. Induction on m. After the first write, the lemma is correct: initially all replicas are current, so the first write coordinator finds all replicas from its write quorum current. Also current are any additional replicas it may put into the 1-st good replica list. Hence, the write is applied at all participating replicas, and they all increase their version numbers from 0 to 1 which becomes the maximum version number among all replicas. Also, all replicas that performed the write remain current after that. Because the only nodes with version number 1 are the ones that participated in the write, they all receive the 1-st good replica list. Finally, no replica is marked "stale" after the 1-st write.

Assume that the lemma is correct after i writes perform, and consider the $(i + 1)$-st write. By the definition of quorums, the $(i + 1)$-st write quorum has at least one node in common with the i-th write quorum. This node can give permission to the $(i + 1)$-st write only after it performs the i-th write (since a write releases the permission lock on the members of its quorum only in step 6 of the protocol, when the commit message is sent). Therefore, by assumption, this node reports to the $(i + 1)$-st operation version number i, which is the maximum version number in the system. Hence, the nodes that the $(i+1)$-st write puts into the $(i + 1)$-st good replica list in step 3 are:

- all replicas from the $(i + 1)$-st write quorum with version number i and stale data flag "current". By assumption, they are all current.

- some additional replicas from the good replica list taken from one of the replicas with version number i. By assumption, it can only be the i-th good replica list. Since i-th operation obtains the commit lock on all these nodes before performing, for every such node, either its commit lock is held by the i-th write, or it performed the i-th write. Therefore, by the inductive assumption, all replicas from this list whose commit lock is not held by the i-th write have version number i, stale data flag "current" and hence are current.

Therefore, since replicas send their prepare acknowledgement to a write only after obtaining the commit lock on behalf of this write, it is guaranteed that all above replicas that respond with the prepare acknowledgement in step 5 of the $(i+1)$-st write will be found current. In addition, in step 6, the $(i + 1)$-st write may include into the good replica list some additional replicas that, during step 5, reported version number i and stale-data flag "current" in their prepare acknowledgement. By assumption, these additional replicas are also current.

Therefore, all replicas put into the $(i + 1)$-st good replica list are current and have version number i. Hence, on receiving the commit message from the $(i + 1)$-st write, they apply the write (thus they will be current after they perform $(i + 1)$-st write) and increment their version numbers (thus they will all have version number $i + 1$ after the write), while their stale replica flags remain "current". (Thus, we have shown that (5) holds.)

All other replicas that perform the $(i + 1)$-st write are given version number $i + 1$ and the good replica list constructed above. After the $(i+1)$-st write, these nodes are marked "stale" and hence run the propagation protocol. They can obtain a new state either from a member of their good replica list (which has been shown above to have version number $(i + 1)$ and stale data flag "current") or from some node which has a version number no less than their own. In the latter case, it can only be a node with version number $i + 1$, because a write operation can increase the maximum version number by at most 1 and before the $(i + 1)$-st write the maximum version number was i by assumption. It can also be only a node marked "current", because this is the only case when propagation is done between two nodes having the same version number with a source node not being in the recipient's good replica list. Hence, any replica from which a node that performed the $(i + 1)$-st write may obtain the new state must have version number $i + 1$ and stale data flag "current". Therefore, after obtaining the new state, the recipients will have version number $i + 1$ (as before) and marked "current" which means they will no longer run the propagation protocol. Thus, (together with the previous paragraph) we have shown that, after the $(i + 1)$-st write, despite possible propagations, all nodes that performed the $(i + 1)$-st write have version number $i + 1$ (condition (1)).

As already mentioned, a write can increase the maximum version number in the system by at most

1, and any propagation only transfers version numbers between nodes and cannot increase them. Hence, $i+1$ is the maximum version number in the system after the $(i+1)$-st write (condition (2)).

Until the $(i+1)$-st write arrived, no replica had version number $i+1$, so the first replica that receives version number $i+1$ does it during the $i+1$-st write operation. Hence, this replica receives the $(i+1)$-st good replica list. Other replicas may obtain version number $i+1$ either during the $(i+1)$-st write or by propagation. In either case, they get the $(i+1)$-st good replica list together with the $(i+1)$-st version number. (In the case of writes, all replicas are given the same good replica list. In the case of propagation, a replica always obtains both the version number and the good replica list from the source node.) Therefore, condition (4) holds.

Finally, we have shown that all replicas from the $(i+1)$-st good replica list are current and marked "current". All other replicas participating in the $(i+1)$-st write are originally marked "stale" by the write operation. Therefore, any replica outside the $(i+1)$-st good replica list can obtain the state (version number $i+1$, stale data flag "current") only by propagation ultimately originated at one of the nodes from the $(i+1)$-st good replica list. But any propagation involving the source replica marked "current" results in the recipient replica having the same data item as the source replica. Because the source replicas are current, the recipients are also current (condition (3)). □

Based upon these lemmas, the first theorem below says that the protocol is correct.

Theorem 1. The protocol defined in the previous section provides one-copy serializability.

Proof. As mentioned in section 2, it is sufficient to show that (a) write operations as well as read and write operations cannot perform in parallel; and (b) a write is always applied to the current replicas and a read always returns the current data. Condition (a) is given by Lemma 1. To show that a write operation is always applied to the current replicas, one needs only to slightly re-phrase the proof of condition (5) from Lemma 2. We leave it to the reader. As to read operations, any read quorum has at least one node in common with the write quorum used by the preceding write operation. By Lemma 2, this common node has the maximum version number among all replicas. Also by Lemma 2, any replica with this version number and the stale data flag "current" is current, and so is any member of the good replica list reported by the common node. □

	Version number	Good replica list	Stale data flag
x1	0	{x1 - x5}	"current"
x2	0	{x1 - x5}	"current"
x3	1	{x3, x4, x5}	"current"
x4	1	{x3, x4, x5}	"current"
x5	1	{x3, x4, x5}	"current"

Figure 2: Example: the state of a system with 5 replicas.

One can see from the proofs of Lemma 1 and 2 why the write coordinators can include additional replicas into the commit protocol regardless of who holds their permission locks, even if that renders responses these replicas had sent to other operations incorrect. (Similar considerations hold for reads.) Obtaining permission locks from a write quorum of nodes is sufficient for serializability, so locking additional replicas is not necessary. As to identifying the most recent replicas on which to apply the update, the write protocol relies exclusively on responses with the maximum version number, and these responses never come from the nodes whose permission locks are bypassed by other operations. Then, the responses from these replicas are always correct. Because it is this observation that makes our approach viable, we would like to re-emphasize it with an example.

Example. Consider a system with 5 replicas $x1$, $x2$, $x3$, $x4$, $x5$, majority-defined quorums, and safety threshold 2. Originally all replicas are current, have version number 0 and good replica list that includes all replicas.

Assume a write runs, collecting quorum $\{x3, x4, x5\}$ in step 1 of the protocol. Since all these replicas are current and their number exceeds the safety threshold, no additional replicas need be contacted. Then, in step 3, all replicas from the quorum are put in the new good replica list, and, as no stale replicas were found, the empty stale list is constructed. Finally, at the commit time, the write performs on all replicas from the quorum bringing the system to the state shown in Figure 1.

Now assume that, in this system state, coordinators A and B initiate the write operations at the same time, using the quorums $x1, x2, x3$ and $x3, x4, x5$ respectively; A has obtained permissions (and hence responses) from $x1, x2, x3$; B has obtained permissions from $x4, x5$ and is blocked waiting for the permission from $x3$. A is going to perform first. Since during the

	Version number	Good replica list	Stale data flag
x1	2	{x3, x4}	"stale"
x2	2	{x3, x4}	"stale"
x3	2	{x3, x4}	"current"
x4	2	{x3, x4}	"current"
x5	1	{x3, x4, x5}	"current"

Figure 3: Example: the state of the system from Figure 1 after the write operation A.

quorum collecting phase A was able to contact only one current replica ($x3$), it needs one more current replica to reach the safety threshold. So, when constructing the new good replica list in step 3, A puts there $x3$ (it reported the maximum version number and stale data flag "current") and adds a replica from $x3$'s good replica list, say, $x4$. At the commit time, A will perform on $x1$, $x2$, $x3$, and $x4$. It will mark $x1$ and $x2$ stale and apply the update to $x3$ and $x4$, getting around the permission lock on $x4$ and rendering the response that B obtained from $x4$ incorrect (Figure 2). After A is finished, B obtains permission from $x3$ and proceeds with the operation. The response from $x3$ carries the version number 2 which is the maximum version number that B has collected. The rest of the algorithm depends only on the response from $x3$ which makes it irrelevant that some other responses are incorrect. In fact, B will apply the update to $x4$ because it is in the good replica list of $x3$, even though the response from $x4$ that B has still carries version number 1. $\square$

The following two theorems deal with tolerance to simultaneous node failures and effectiveness of the propagation algorithm. Given Lemma 1 and 2, they can be proved by simple induction on the sequential number of write operations.

Theorem 2. Let t be the safety threshold. If t does not exceed the size of any write quorum, then, after any successful write operation op, there are at least t current replicas in the system, and there are at least t replicas in op's good replica list. $\square$

Theorem 3. Any node running the propagation protocol makes progress whenever it successfully obtains a new state from any member of its good replica list. Formally, let $v(a), s(a)$, and $G(a)$ be the version number, the stale data flag and the good replica list of a node a. Then, for any node a, if $b \in G(a)$, then either $v(a) < v(b)$ or $v(a) = v(b)$ and $s(b) =$ "current". $\square$

Finally, we showed in the proof of Lemma 2 that the members of the current good replica list will always be found current during step 5 of the write operation. Moreover, by Theorem 2, this list contains at least t replicas. Hence, in step 3, the coordinator always finds t replicas to put into the new good replica list, and, if these replicas are operational, they will be found current in step 5. Therefore, in the absence of failures, our protocol never requires more than two rounds of message exchange before committing the operation.

5 Performance

In this section, we compare the performance of our protocol with three variations of the Gifford's protocol that can support partial writes.

In the first variation, the one actually described in [6] and denoted G1 below, all writes to a given data item normally perform on the same quorum of replicas. When a write fails to obtain permission from a quorum of current replicas (a current quorum for short), it sends the permission requests to all remaining replicas, and the operation proceeds as soon as a current quorum of permissions is obtained. If the operation fails to obtain permission from a current quorum even after requesting all replicas, necessary number of stale replicas are synchronously brought up-to-date to conclude a current quorum. After completion of the operation, the coordinator remembers the current quorum collected and uses it for future operations.

The second variation, denoted G2, is very similar to G1. The only difference is that G2 attempts to reduce the risk of synchronous update propagation by propagating updates asynchronously after every write from the replicas on which the write was performed to the rest of replicas. As in G1, all writes to a given data item normally perform on the same quorum of replicas. (Otherwise, if a write occurs before previous propagation completes, the quorum collected may not contain all current replicas, and the operation would either have to wait for the propagation to complete, or request permission from additional replicas, in both cases incurring a delay even in the absence of failures.) Risking being unfair to our protocol, we assume for simplicity that a single current replica is assigned the responsibility for propagating updates to the stale replicas, and that this "propagator" never fails in the process. We assume that the propagator offers the propagation to every stale replica only once; if a stale replica was found failed at that time, no further attempt to bring it up-to-date is maid until the next write operation spawns a new propagation process.

Protocol	G1	G2	G3	Our
Msg overhead	$O(w)$	$O(N)$	$O(N)$	$O(w)$

Table 1: Number of messages per operation in different protocols.

Finally, variation G3 attempts to perform every write on all replicas of the data item. The write coordinator requests permission from all replicas. If a quorum is collected, the coordinator performs the write on all accessible replicas. This variation practically never performs synchronous propagation, since the only case where a replica may be stale is when it is recovering from a failure.

5.1 Message overhead

First, consider message traffic generated by the protocols. Let N be the total number of replicas of a given data item, w be the number of nodes in write quorums[4], and t be the safety threshold in our protocol.

In the normal case of the absence of failures, protocol G1 sends w permission requests, receives w permissions, sends w prepare messages, receives w acknowledgements, and sends w commit messages, for the total of $5w$ messages per operation. G2 generates the same messages followed by $N - w$ propagation messages from the propagator to stale replicas, for the total of $4w + N$. G3 obviously generates $5N$ messages per operation (N in every phase above). Our protocol needs $2w$ messages to obtain a quorum of permissions (as G1 and G2), and then, in the best case, if the obtained quorum contains at least t current replicas, generates $3w$ messages during the commit stage involving w nodes, for the total of $5w$. In the worst case, when the collected quorum contains no current replicas, the commit stage involves $w + t$ replicas and generates $3(w + t)$ messages. In addition, w replicas found stale will request and receive the missing updates generating $2w$ more messages. Overall, the total number of messages in the worst case is $2w + 3(w + t) + 2w = 7w + 3t$. (Note that like in G2, we could also assign the propagation job to a single good replica reducing the number of messages in the worst case to $6w + 3t$.)

The message overhead in the absence of failures is summarized in Table 1. If we assume the grid-based quorums of [4] where $w = 2\sqrt{N} - 1$, Table 1 shows that G1 and our protocol normally generate $O(\sqrt{N})$ messages per operation while G2 and G3 generate $O(N)$ messages per operation. The number of messages in G2 could be reduced if the propagator is required to propagate the update to only w stale replicas. Then, the generated traffic for G2 would be $O(\sqrt{N})$ as well.

In the presence of failures, all protocols require $O(N)$ messages per operation.

5.2 Response time and utilization

To estimate the response time and server utilization in our and the alternative protocols, we have conducted a simulation study. We used a *closed* model of a computer system [11], in which there are several clients acting as coordinators of replica control and commit protocols, and servers replicating the data. Each client issues a transaction and, depending on its outcome, either immediately resubmits it (if the transaction aborted), or issues the next transaction after some *think time* (if the transaction committed). We assume the availability of a multicast facility for the network, so that clients can issue a message to a group of servers in parallel. The network delays are considered negligible compared with disk access delays which dominate service demands at the servers. Also, we assume that only nodes can fail, and the network is reliable.

In modeling node failures, we follow [9] and [13] in assuming that a node participating in the operation cannot repair or fail during the operation execution (operations are "instantaneous" with regard to failures and repairs). In other words, when a server is first contacted by an operation, it can be found either up or down (with the probability p of being up). If a node was found up, no subsequent messages from the same operation will find this node down. If a node was found down, the operation coordinator assumes it is going to be down for the duration of the operation and never contacts this node again.

The simulation is event-driven. Deadlocks are detected by the fact that the event queue is empty and resolved by aborting a random transaction that holds any locks. This simple way of handling deadlocks may not be the most appropriate for all the protocols considered. Since it turns out that our protocol is less prone to deadlocks than G1 and G2, we present the results both with and without the overhead caused by transactions that were aborted due to deadlocks. The results with deadlock overhead excluded can be considered as the case where an ideal zero-overhead deadlock handling method is employed.

The simulation parameters include service demands

[4] We limit our discussion to *fully distributed* quorums where all write quorums are of the same size, all read quorums are of the same size, and every replica participates in an equal number of read and write quorums.

for processing various messages, timeout parameter after which `RPC.CallFailed` is returned to a message sender, and think time.

In response for permission request, prepare, and abort messages, the servers must acquire or release some locks in main memory cache and write a log record. Commit processing requires different amount of work depending on whether the replica is stale or current. (The case where stale replicas "commit" is relevant only in our protocol. We refer to this case as "stale commit" below). To process a commit message, stale replicas release their locks, update their version number and stale-data flag (we assume that version numbers and stale-data flags are updated in the main memory cache most of the time), and write a log record. In addition, current replicas must install the update data itself. (We call the commit by current replicas "current commit").

Thus, all of the above, except for the current commit processing, requires one disk access to write a log record, which dominates processing time. The current commit requires an additional disk access to install data. We assume that servers do not have a dedicated disk drive for the log file, and hence the time to write a log is a random disk access time. Therefore, service demands for all above messages except the current commit are assumed to be a uniformly distributed random variable with the mean of 1 time unit, and the service demand for the current commit is assumed to be a uniformly distributed random variable with the mean of 2 time units.

The time to perform propagation generally depends on the number of missing updates to be installed. We assume that on average it takes three disk accesses on the receiving side and no accesses on the sending side (the recent updates are cached in the main memory there) to perform the propagation. This assumption favors G1 over G2 and our protocol since in G1, stale replicas are likely to miss many updates, while in G2 and our protocol, stale replicas are usually only slightly behind because of asynchronous update propagation. Note that this assumption says nothing about the overall number of disk accesses due to propagation activity in the system. Rather, it concerns the number of disk accesses performed in a *single instance* of propagation involving a single stale and a single current replica.

Finally, timeout is assumed to be 10 time units, and think time of the client is exponentially distributed random variable with the mean of 15 time units. The simulation parameters are summarized in Table 2. Since we are interested only in relative performance of the protocols, in this study, we measure time in abstract time units.

Parameter	Value
Service demand for Perm. Request Prepare Abort Stale Commit	Uniform with mean 1
Service demand for Current Commit	Uniform with mean 2
Service demand for Propagation	Uniform with mean 3 for the receiving server; 0 for the sending server
Think time	Exponential with mean 15
Timeout	10

Table 2: Simulation parameters.

In all experiments, unless specified otherwise, the number of replicas in the system is 7, all quorums are majority-based (i.e., any 4 replicas form a quorum), and the multiprogramming level is 3 (i.e., there are 3 clients issuing transactions). All transactions are assumed to be single-step and involve writing to the same data item. The reason for choosing this workload is that, as explained below, it is the most unfavorable to our protocol, and because it is the simplest to model.

For this study, we implemented our protocol (referred to as Our-t, where t is the safety threshold), and G1 and G2. We did not feel it was necessary to implement G3 since the tradeoffs between it and our protocol are clear enough.

The average response time of successful transactions (including all retries due to deadlocks and failures) versus p is shown in Figure 4. The figure shows rather dramatic improvement in response time of our protocol over G1 and G2. However, it turns out that a significant part of this improvement is due to the fact that our protocol is less prone to deadlocks. The reason for lower likelihood of deadlocks is that in our protocol, once an operation collected a quorum of permissions, this operation will never be a party to deadlock. On the other hand, in G1 and G2, only collecting a *current* quorum guarantees the operation will not be deadlocked. If the quorum the operation first collected is not current, it requests permission from additional replicas. These additional replicas may well be locked by other transactions, in which case deadlock will occur.

Because many ways of handling deadlocks exist, and the existing protocols benefit from lower-overhead deadlock handling more than ours, we now consider

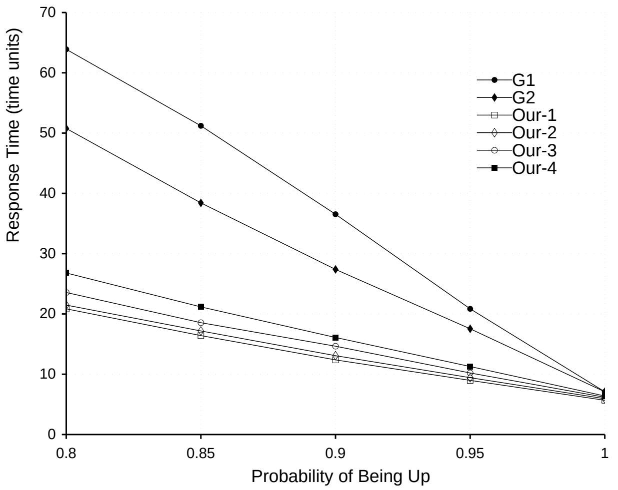

Figure 4: Average response time including all overhead.

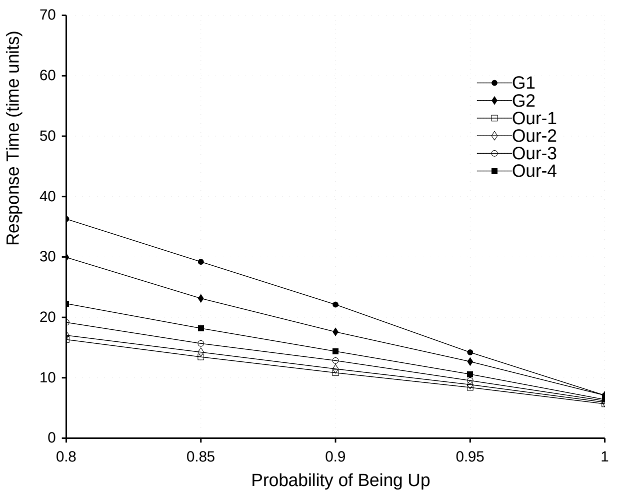

Figure 5: Average response time with deadlock overhead excluded.

response times with the deadlock overhead factored out. This is equivalent to applying an ideal deadlock handling method with no overhead. Figure 5 shows the average response time of successful transactions that includes retries due to failures but not deadlocks. While less dramatic than in Figure 4, the improvement our protocol achieves is still significant. The reason for this improvement is a combination of the following factors. First, our protocol in most cases performs many fewer synchronous update propagations than G1 and, for the safety thresholds smaller than 4, even G2. Second, if the quorum collected in the first phase of a protocol is not current, G1 and G2 require one extra round of message exchange to obtain permission from additional replicas, while our protocol in this case moves directly to the commit stage. Third, when permission request is sent to all replicas (which happens if a protocol failed to collect a quorum of permissions in the first phase, or, in the case of G1 and G2, also when the collected quorum is nor current), G1 and G2 wait until the *current* quorum is collected or until the timeout before moving to the next stage. On the other hand, our protocol moves to the next stage as soon as *any* quorum is collected. Therefore, our protocol waits for the quorum of responses that are the fastest to arrive.

It is interesting to note that our protocol performs slightly better even when all nodes are completely reliable ($p = 1$), in which case none of the above reasons exist. In fact, as Figure 6 shows, for higher multi-programming levels this improvement becomes rather significant. The reason for this improvement is that our protocol finds some parallelism in processing the commit stage of one transaction and the permission collecting stage of another, as the following example illustrates.

Consider a system with 3 nodes, initially all current, and three clients. Assume each client has issued a transaction, so that there are three outstanding transactions in the system, T1, T2, and T3. Since we consider a fully reliable system, in G1 and G2, all three transactions will execute sequentially on the same quorum of replicas. Assuming that service demands are deterministic and equal to the mean values from Table 2, the total execution time for these transactions is $3 \cdot (1 + 1 + 2) = 12$ time units. In our protocol, each transaction chooses a random quorum. Then, the execution depicted in Figure 7 is possible. There, transaction T1 chooses quorum $\{1, 2\}$ and succeeds in obtaining permission from both replicas. In the mean time, T2 that chose quorum $\{2, 3\}$, obtains permission from replica 3 and is blocked waiting for replica 2. T3 chooses quorum $\{1, 3\}$ and is blocked waiting for both.

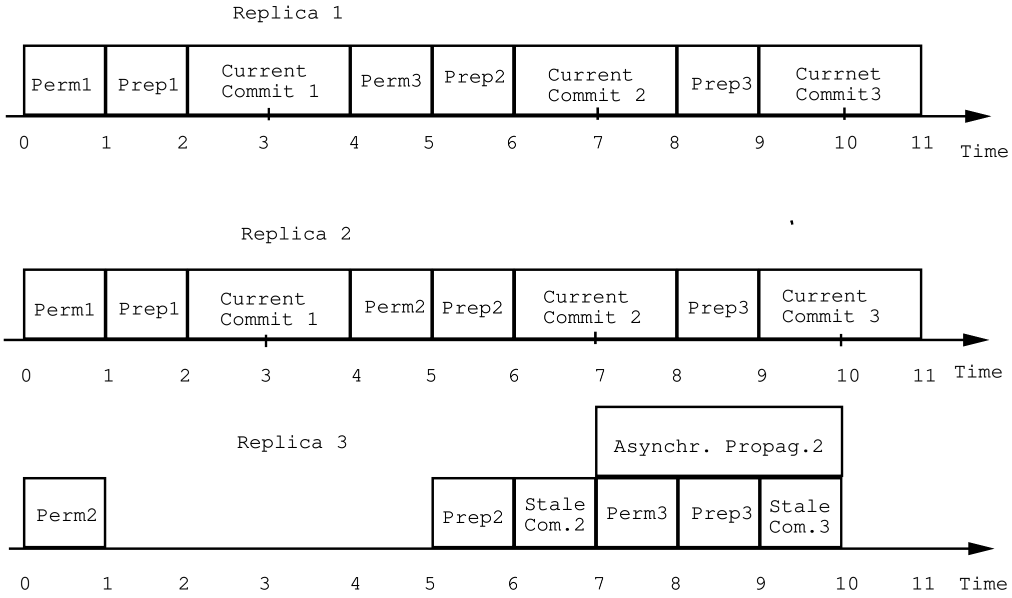

Figure 7: An example execution of our protocol.

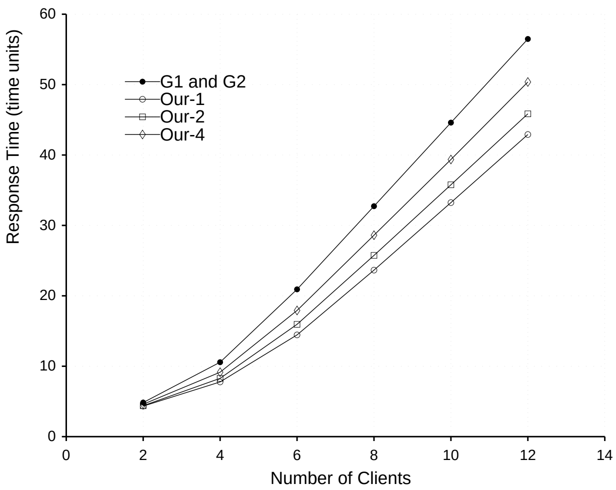

Figure 6: Average response time for a fully reliable system.

At time 4, T1 finishes, and T2 can acquire permission from node 2, and T3 gets permission from node 1. T2 can now proceed, while T3 is still blocked waiting for node 3. T2 finds that it contacted only one current replica (node 3 became stale after T1 finished), and adds replica 1. Thus, all three replicas participate in the commit stage of T2. Note that replica 1 participates even though its permission lock was acquired by T3. When T2 sends out the commit message, nodes 1 and 2 take 2 time units to process it while node 3 takes only 1 time unit, because it is stale. But as soon as node 3 finishes with T2, it can give its permission to T3, which now can proceed. Therefore, the total time to execute the three transactions is only 11 time units. This effect is magnified when the service demands are non-deterministic and increases with their dispersion.

Now we turn to server utilization. Figure 8 shows maximal server utilization for systems with different number of replicas of the data item using *grid* quorums [4], in which the size of a write quorum $w = 2\sqrt{N} - 1$. Here, we assume fully reliable servers and multiprogramming level 3. The utilization includes all work done at the server, including update propagation. Figure 8 shows a significant improvement of our protocol over G1 ad G2. The reason is that in G1 and G2, all operations are performed by the same servers while in our protocol, load is shared more evenly among all servers. As shown in [4], higher server utilization affects negatively the response time of transactions that involve multiple data items. Hence, our protocol should reduce the response time even further when the workload includes multi-step transactions.

Finally, compared to G3, our protocol has lower message overhead ($O(w)$ vs. $O(N)$). Also, server utilization in our protocol is lower (maximum server utilization in G3 should be the same as in G2 and G3 since all these protocols perform all writes on the same replicas). As shown in [4], when server nodes keep

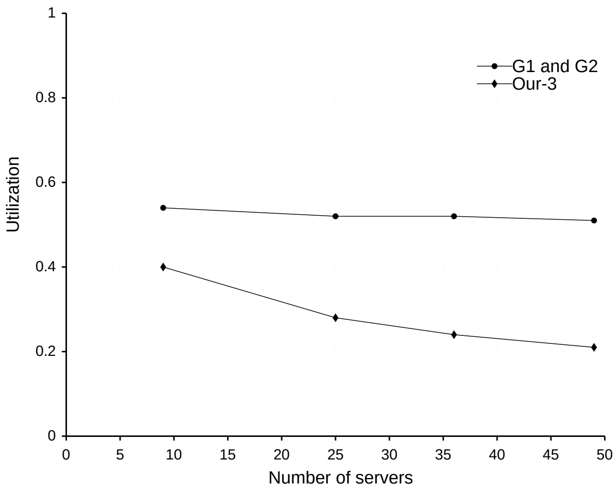

Figure 8: Maximum server utilization for a fully reliable system using grid quorums.

multiple data items (which is usually the case in practice), systems with lower server utilization have better response time and greater overall capacity. Thus, under moderate to heavy load, our protocol should have a performance advantage over G3. However, under light load, in the presense of failures, G3 should outperform all other protocols including ours, since these protocols first request permission from only a quorum of replicas, and if any of them fail, the protocols have to timeout and then request permission from the remaining replicas. In contrast, G3 always requests permission from all replicas, so if some of them fail, it can still collect a quorum in one phase.

Thus, the overall conclusion is that if the message overhead is of no concern, and the load is expected to be light, consider using G3. Otherwise, our protocol is a better choice.

5.3 Choosing the safety threshold: availability

When the safety threshold in our protocol is equal to the size of a write quorum, the protocol guarantees to maintain the same number of current replicas as G1 and G2. Hence, it can tolerate the same number of arbitrary simultaneous replica failures as G1, G2, and G3[5]. As we saw in section 5.2, for all values

Safety threshold	4	3	2	1
% of aborted transactions	0.00027	0.00085	0.0011	0.0021

Table 3: Fraction of aborted transactions for $p = 0.95$ and $N = 7$.

of the safety threshold, our protocol has performance advantages over G1 and G2. However, this advantage increases when the safety threshold is smaller. It is believed [9, 12] that simultaneous failures are unlikely to occur, especially if a battery-backed memory is employed. Then, small values of the safety threshold can be chosen. To quantify this conjecture, Table 3 gives the the fraction of aborted transactions for different values of the safety threshold under the assumption of independent node failures. (The system is assumed to have 7 nodes, with the probability of a node being up 0.95.)

6 Conclusion

In this paper, we propose a protocol based on good replica lists that supports partial write operations. Compared to three variations of Gifford's protocol also capable of supporting partial writes, our protocol provides better response time than two of them, lower message overhead than the third, and better load sharing than any of the three.

An important observation which made our approach viable is that if one wants to maintain at least t current replicas at any time to guard against simultaneous node failures, the updates can be applied to additional replicas without obtaining their permission first, even though these replicas may have previously given their permission to other operations, and their earlier responses are invalidated by the current operation. This observation allowed us to cut the number of nodes to be contacted without adding more phases to the protocol. The good replica lists used in the protocol also allow very effective propagation of missing updates by providing stale replicas with a strong hint about the location of current replicas.

Finally, our approach blends well with other ideas in replica control like dynamically-defined quorums (see, e.g., [9]) and protocols with *witnesses* proposed in [13]. Witnesses are the nodes that keep state information but no actual data. They participate in read and write quorums, update their version numbers, etc., but never apply actual operations. The idea is to cut disk space overhead without compromising availability of the system. One can incorporate this idea in our protocol by simply excluding the witnesses from the initial

[5] It may appear that since G3 maintains *all* operational replicas current, it should provide better availability than the other protocols. However, it does not, because if a write quorum of replicas fail, the system becomes unavailable no matter whether the remaining replicas are current or not.

good replica lists. The lists themselves, as part of the node state, should be replicated on witnesses as well as on the first class nodes that have actual data.

References

[1] D. Agrawal and A. El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. on Comp. Sys.*, 9(1), pp. 1-20, February 1991.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, Reading, Mass., 1987.

[3] A. W. Brown. *Object-Oriented Databases: Applications in Software Engineering*, McGraw-Hill, New York, 1991.

[4] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The Grid protocol: a high performance scheme for maintaining replicated data. *IEEE Trans. on Knowledge and Data Eng.*, 4(6), pp. 582-592, 1992.

[5] J. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course*, New York, Springer-Verlag, 1979.

[6] D.K. Gifford. Weighted voting for replicated data. In *Proc. of the Seventh ACM Symposium on Operating Systems Principles*, pp. 150-159, December 1979.

[7] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *J. of the ACM,* 32(4), pp. 841-860, October 1985.

[8] A. Hisgen, A. Birrell, C. Jerian, T. Mann, M. Schroeder, and G. Swart. Granularity and semantic level of replication in the Echo distributed file system. In *Proc. of the IEEE Workshop on Management of Replicated Data*, pp. 2-4, 1990.

[9] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. Database Syst.* 15(2), pp. 230-280, June 1990.

[10] A. Kumar. Hierarchical quorum consensus: a new algorithm for managing replicated data. *IEEE Trans. on Computers,* 40(9), pp. 996-1004.

[11] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*, Prentice-Hall, Englewood Cliffs, 1984.

[12] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pp. 226-238, October 1991.

[13] J.-F. Pâris. Voting with witnesses: A consistency scheme for replicated files. In *Proc. of the IEEE Int. Conf. on Distr. Comp*, pp. 606-621, 1986.

[14] M. Rabinovich and E. D. Lazowska. Improving fault-tolerance and supporting partial writes in structured coterie protocols for replicated objects. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 226-235, June 1992.

[15] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. on Computers*, 39(4), 1990.

[16] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. on Software Eng.*, 5(3), pp. 188-194, 1979.