Improving Fault Tolerance and Supporting Partial Writes in Structured Coterie Protocols for Replicated Objects *

Michael Rabinovich end Edward Lazowska

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

October 24, 1996

Abstract

This paper presents a new technique for efficiently controlling replicas in distributed systems. Conventional structured coterie protocols are efficient but incur a penalty of reduced availability in exchange for the performance gain. Further, the performance advantage can only be fully realized when write operations always *replace* the old data item with the new value instead of *updating a portion* of the data item. Our new approach significantly improves availability while allowing partial write operations.

After presenting our general approach, we apply it to an existing structured coterie protocol and analyze the availability of the resulting protocol. We also show that other classes of protocols can make use of our approach.

1 Introduction

Replication of data is commonly used in distributed systems to increase the availability of services. In most cases, the consistency of the data must be maintained despite node failures and/or network partitionings. This can be achieved by requiring that, in order to succeed, read and write operations obtain permission from certain sets of replicas. These sets, called read and write groups, are defined in such a way that any two write groups as well as any read and write groups 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.

Because the coordinator (the node that initiated the operation) must communicate with all nodes from at least one of the groups before returning to the user, there is some inherent performance penalty for providing

*This research was supported in part by the National Science Foundation (Grants No. CCR-8907666 and CCR-8619663), Digital Equipment Corporation, Apple Computer, and the Washington Technology Center.

strong consistency. To minimize this penalty, it is desirable that the groups be small. The structured coterie protocols [12, 3, 1] achieve this by defining groups based on a logical structure imposed on the network. For instance, in the grid protocol [3] considered in section 5, the nodes replicating the data item are viewed as arranged in a rectangular grid. A read group is defined to be any set of nodes that includes a representative from every column of the grid, and a write group is defined to include some read group plus an entire column of the grid. For square grids, the size of read groups is $\sqrt{N}$ and the size of write groups is $2\sqrt{N}-1$, where N is the total number of replicas. Ths is in contrast to the voting protocol [7], where the group size in the simplest case is $\lfloor \frac{N+1}{2} \rfloor$.

Since in the existing structured coterie protocols, the operations rely on their knowledge of statically predefined logical strusture of the network, these protocols are static by their nature. This means they cannot adjust the read and write groups to reflect failures and recoveries occurring in the system. On the other hand, the voting protocol allows such re-adgustment [11] because in this protocol the groups are defined based on the number of votes regardless of their identity. As a result, the dynamic voting protocol can keep the data item available as long as there is one accessible replica provided the failures are mostly suquential so that the protocol could adjust to the failures as they arrive. In contrast, in the existing structured coterie protocols, any read or write group of replicas being down makes the system unavailable, even if the failures are accumulated gradually over time.

In this paper, we propose a mechanism for group re-adjustment in the case of the structured coterie protocols. Moreover, we argue that our approach is a preferable way to adjust groups even in the voting protocol. We observe that, given an ordered set of nodes, one can usually devise a rule that unambiguously imposes a desired logical structure on this set. Then, the read and write operations can rely on this rule rather than on the knowledge of statical structure of the network in determining what replica sets include groups. If in addition, at any time, all operations can agree on a set of replicas from which the groups are drawn, then the protocol can adjust dynamically this set to reflect detected failures and repares and at the same time guarantee consistency.

In our protocol, we assume that each node is assigned a name and all names are linearly ordered. Among all nodes replicating the data item, we identify a set of nodes considered the current *epoch*. At any time, the data item may have only one current epoch associated with it. Originally all replicas of the data item form the current epoch. The system periodically runs a special operation, *epoch checking*, that polls all replicas of the data item. If any members of the current epoch are not accessible (failures detected), or any replicas outside the current epoch have been successfully contacted (repares detected), an attempt is made to form a new epoch. (Epochs are distinguished by their *epoch numbers*, with later epochs assigned greater epoch numbers) For this attempt to be successful, the new epoch must contain a write group of the previous epoch, and the list of the new epoch members (the *epoch list*) along with the new epoch number must be recorded on every member of the new epoch. Then, due to the intersection property of the groups, it is possible to guarantee that, if the network partitions, the attempt to form a new epoch will be successful in at most one partition and hence the uniqueness of the current epoch will be preserved. For the same reason, any successful read or write operation must contact at least one member of the current epoch and therefore obtain the current epoch list. Hence, the operation can reconstruct the logical structure of the current epoch and use it to identify read or write groups. Similarly to dynamic voting, the system will be available as long as some relatively small number of nodes (the number depends on the specific protocol) are up and connected.

Also, our protocol addresses the problem of supporting partial writes, i.e., the write operations that, on a given replica, update only a portion of the data item rather than replacing it entirely with a new value. File systems are an example of such systems. Supporting partial writes efficiently presents a difficult problem. Indeed, if all writes are total, the protocol does not have to worry about how current the version being replaced is. Hence, different write coordinators can perform the write of the members of different write groups of replicas. The coordinator will replace the data item with a new version on all nodes from the group 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 are normally served by nodes from different groups) and light network traffic. In systems with partial writes, a write operation can be applied to current replicas only. Therefore, it was thought [7], the write coordinator must collect permission from a write group of *current* replicas to perform the write. This means that requests from different coordinators cannot be served by different groups, and the coordinator must either perform the write on *all* replicas of the data item or face the possibility of synchronously bringing the obsolete replicas up-to-date whenever no full write group of current replicas is available.

In our approach, we note that the write coordinator can avoid having to apply the write to every replica in a write group. Instead, it can mark some replicas "stale". Then, the subsequent operation is guaranteed to see either a current replica or a replica marked stale, as long as it can obtain responses from a group. Therefore, the write coordinator avoids having to collect permission from a write group of current replicas (even stale replicas will do), different coordinators can communicate with different write groups, and synchronous reconciliation of obsolete replicas is never needed. In addition, the distinction between good and stale data allows a very effective asynchronous update propagation. When the write coordinator discovers it will have to mark some replicas stale, it sends a list of these to the good replicas so that they can bring the stale ones up to date.

2 Related Work

Our epochs are in essence analogous to distinguished partitions in dynamic voting [11] except the members of an epoch know the full list of members of their epoch as opposed to just the cardinality of the ditinguished partition. This allows the structured coterie protocols to become dynamic. Besides, our epoch management is different from the mechanism used in dynamic voting [11] in that we separate the read and write operations from the operation that checks whether epoch changing is needed. There are many benefits in doing this. First, epoch checking must be done much less frequently than reads and writes. It makes sense then to separate frequent operations from less frequently needed work. Second, epoch checking requires an attempt to communicate with all nodes whereas we want to avoid that in reads and writes. Third, while a stream of reads and writes is beyond the control of the system and usually dries out during off-hours, we want a steady (albeit infrequent) pulse of epoch checking operations to avoid the accumulation of failures. Fixing this problem by running "dummy" writes as suggested in [11] can hurt the performance since the "dummy" writes may interfere and hence delay the "normal" reads and write. In contrast, epoch checking does not interfere with reads and writes in the absence of failures. Finally, if several data items are replicated on the same set of nodes, the epoch management can be done per this whole group of data. Thus, the overhead is amortized over several data items, whereas if epoch management is bundled with writes it must be done separately for each data item.

The idea of changing the nodes' states asynchronously with read and write operations to reflect the dynamically changing topology of the network was previously used in the virtial partitions protocol ([4], generalized in [5] and [6]). Our approach is different in that we require at least a write group of nodes from an existing epoch to be included in the new epoch. In the virtual partitions protocol, new views are formed regardless of the node membership in the earlier views. Therefore, in order to ensure that the data object can be updated in at most one view, the accessibility threshold is introduced. Assuming the accessibility threshold is at least half of the total number of replicas and the write quorum is at least half of the replicas in the view, one can infer that at least a quarter of the total number of replicas need be operational and connected for the data object to be available for update. Our protocol is free of such limitation. On the other hand, the advantage of the accessible copies protocol is that it can use the read-one, write-all discipline and still allow up to half of the nodes in the system to fail. Our protocol is not suitable for using this discipline because in this case, a single failure would make the epoch change impossible and the data object unavailable for update.

The distinction between the good and stale replicas was previously used in the dynamic voting protocol with its logical and physical version numbers [11]. However, this idea was not employed there to reduce the number of replicas that must be contacted during the write operation, since in the absence of failures, all replicas of the data item must be contacted in [11].

3 Model and Terminology

Two different models are common for distributed systems with data replication. The client-server model is popular within the distributed file system community. In this model, the system consists of two distinct sets of nodes: servers that actually store the data and clients that initiate operations. In the other model, common in the database community, all nodes are equal: each keeps a full-rights replica, and each can initiate operations. The difference between these models, from the replica control point of view, is whether the node that initiates the operation has a full-rights replica of the data and whether it can be trusted. In the client-server model the coordinator is often not trusted and therefore may not always use message broadcast to communicate with the servers [14]. Most replica control protocols, including the ones presented in this paper, can be relatively easily translated from one model to the other. We choose the "database" model for the current presentation.

We assume that write operations update only a portion of a data item. 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. Multicast capability is not required of the network but is desirable for better performance. We will assume that the nodes in the system have loosely synchronized clocks, i.e., the difference between any two clock readings does not exceed some fixed amount. Although helpful, this assumption can be removed easily. Finally, we assume that nodes and communication links are *fail-stop*, that is, they fail by crashing and do not behave maliciously.

All algorithms and considerations below are on per-data-item basis. Therefore, we will often use shortcuts and say "all nodes" meaning "all nodes that have the replica of the same data item", "all writes" meaning "all writes applied to the same data item", and so on.

In this paper, 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]. 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 will use the term *coterie* [10]. 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[1].

4 General Protocol

We assume that all nodes agree on a *coterie rule* which defines a coterie over an arbitrary ordered set of nodes. Given two sets of nodes V and S, `coterie-rule(`V`,`S`)` is `true` if S includes a write (read) quorum over V, and `false` otherwise. We also assume that there is a *quorum function* that, given a set of nodes V and a node name, yields a list of nodes representing some quorum over V. It is desirable for better load sharing that the quorum function yield different quorums for different node names and that those quorums contain as few common elements as possible. The algorithms in this section use the procedure `multicast(`V`, message)` to send an identical message to a set of nodes V. We do not make any assumptions about either the implementation of this procedure or the existence of a multicast facility for the network.

Because read operations cannot cause inconsistency, we concentrate on writes here. The read protocol is similar to the write protocol except it does not update any replicas. From now on, unless stated otherwise, quorums will mean write quorums. The protocol consists of three asynchronous procedures: write (and read), propagate, and epoch checking. Each node maintains the following state: a version number which is increased any time a write is applied to the replica; an epoch number; a stale-data flag, a *desired version number* which is meaningful only if the stale-data flag is set; and a list of node names representing a current epoch (the *epoch list*) [2]. Originally all nodes have identical replicas; version numbers and epoch numbers are all 0; stale-data flags are all 0 (none of the replicas are stale); and epoch lists include all nodes. Each node also maintains as a hint a list of nodes (*quorum list*) representing some quorum over the node's epoch list. Originally it is the quorum yielded by the quorum function. The protocol as described here does not address the deadlock problem. For ways to handle deadlocks see for example [2].

[1]There is some confusion in terminology here. Most papers call the elements of W and R write and read groups as we did in the introduction, reserving the term quorum for the voting protocols. From now on, we follow [1] in this matter and use quorums in the general meaning as defined above. The reason for using *quorums* in place of *groups* is that the term *group* is already overloaded with different colloquial and mathematical meanings.

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

4.1 Write Protocol

An algorithm for the write operation in pseudo-code is given in the Appendix. The coordinator sends out a request for permission to the members of its quorum list. Each node that receives the request, obtains a lock for its replica and responds with its state. Upon receiving all responses, the coordinator faces the following cases:

1. The coordinator has collected a set `RESPONSES` consisting of responses other than `RPC.CallFailed` that includes some quorum over the epoch list from a response with the maximum epoch number. (This is the branch taken in the common case of absence of failures.) If `RESPONSES` include a response with the stale-data flag set and the desired version number from this response is greater than all version numbers from the "non-stale" responses (which means that the coordinator failed to contact an up-to-date replica of the data), then the coordinator tries to execute the write on the other nodes in the system (see Case 2). Otherwise, it performs the write on the non-stale replicas with the maximum version number (among those that responded) and marks the other ones stale. Along with the update data to "good" replicas, the coordinator piggybacks the list of nodes it is marking stale thus initiating the propagation of the update. Along with the mark-stale signal sent to replicas being marked stale, the coordinator piggybacks the desired version number equal to the version number that the up-to-date replicas will have after performing the current write operation. The two-phase commit protocol [8] is used to ensure all-or-nothing execution. If the whole action performs successfully, the coordinator returns to the user. Otherwise it tries to execute the write on the other nodes in the system (see Case 2 below).

2. Otherwise (i.e., the coordinator failed to collect a quorum of non-`RPC.CallFailed` responses), the coordinator sends the request for permission to *all* nodes (except perhaps those polled before). After receiving all responses, it checks if it has been able to obtain a quorum of non-`RPC.CallFailed` responses over the epoch list from the response with the maximum epoch number [3], and whether this quorum includes a "non-stale" response with the version number greater or equal to the desired version numbers of all "stale" responses. If this is the case, the coordinator performs the write on the non-stale replicas with the maximum version number (among those that responded) and marks the other ones stale. Similarly to Case 1, the coordinator also sends the list of stale nodes to "good" replicas, and the desired version number to the nodes being marked stale. Finally, the coordinator applies the quorum function to the epoch list from the response with the maximum epoch number and stores the new quorum list for future use. If no quorum of non-`RPC.CallFailed` responses is obtained or some "stale" response has the desired version number greater than the version numbers of all "non-stale" responses, the coordinator aborts the operation and returns "failure" to the user. There is no reason to wait for possible epoch change because such an operation can succeed only if it can obtain a quorum as well.

Note that if the coordinator has found only one "good" replica, then this replica is given full responsibility for the update propagation. Thus, if this replica fails before it completes the propagation to at least one other replica, the data object will be unavailable for the consequent write operations. In order for a single failure to make the system unavailable, the following combination of events must happen: only one "good" replica has been found by the write coordinator; the failure occurs while no propagation procedure is finished; and the failed node is the one responsible for the propagation. The likelihood of such combination of events is very low. Nonetheless, if one desires to completely avoid it, there is a way to do that. In short, it requires that the list of "good" replicas is recorded in every node participating in a write operation. Then, the coordinator of a (successful) write operation always knows the list of "good" replicas. If the number of "good" replicas contacted is less than a predefined safety threshold, the coordinator includes additional "good" replicas in the set of nodes on which it performs the write. Surprisingly, no locking of these additional replicas is needed, so there are no additional rounds of message exchange involved. This approach provides the unconditional resilience to any number of simultaneous failures less than the safety threshold. For a detailed treatment of this problem, the reader is referred to [16].

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

4.2 Propagation Protocol

A node begins the propagation protocol upon receiving a non-empty list of stale replicas. The algorithm design (pseudo-code is given in the Appendix) reflects the fact that we expect many nodes will receive requests to propagate their data whereas it need be done only once for each stale replica. (It would be wrong for the coordinator of the write operation to send the stale list to just one node among those with the current data; this would make the propagation vulnerable to a single node failure.) The duplicate conditional statements are caused by the desire to minimize unnecessary locking.

To each node from the stale list, the coordinator of the propagation operation sends a propagation offer containing its version number. The target node responds with an "already-recovering" signal if propagation is already underway with some other source node; an "i-am-current" signal if it has already been brought up-to-date or the version number from the propagation offer is less than the desired version number of the target replica; or a "propagation-permitted" signal otherwise. In the last case, the target node locks its replica prior to responding.

On receiving permission, the coordinator locks its replica and propagates missing updates to the target node. The concrete way of doing this depends on the data organization in the system. After propagation is completed (or fails), both the coordinator and the target node unlock their replicas.

In this protocol, the source and the target nodes lock their replicas to perform the propagation. Hence, the propagation can interfere with write operations. This is done only for simplicity of presentation. In reality, various logging techniques can be employed to avoid using the same lock for propagation and write operations.

4.3 Epoch Checking Protocol

To avoid accumulating failures, it is desirable to have a relatively steady rate of epoch checking operations. The first question to ask is which node is to initiate them. A simple solution exists if loosely-synchronized clocks are maintained in the system. Each node has its epoch list providing a hint about how many nodes are in the epoch and what position in the ordered epoch list the node itself occupies. The desirable rate of epoch checking operations is a parameter of the system known to all nodes. Given this, each node can calculate times when it should do epoch checking. Because all clocks are loosely synchronized, in a system with all nodes having correct identical epoch lists, the intervals between consequent operations will not exceed $2d$, where d is the maximum clock skew. As we will see, any epoch checking operation brings the system into a state in which all nodes within the current epoch have correct epoch lists and all nodes outside the current epoch are either down or partitioned and hence it does not matter what they do. Therefore, except for intervals between failures or reconnections and the next epoch checking, the rate is steady. In any case, an execution of epoch checking may be lost only if the node whose turn is next to run the operation, fails or partitions. This can be easily fixed, e.g., by assigning the i-th time slot to nodes occupying the $(2i-1)$-th and the $2i$-th positions in the epoch list (modulo epoch list size). The $2i$-th node would first check if the $(2i-1)$-th node is accessible and start epoch checking if not.

This is the only place where we use any assumptions about clocks. An alternative would be to elect a site responsible for initiating all epoch checkings. A new election would be started by any node noticing that epoch checking has not run for a while. (See [9] for election protocols.)

The algorithm for epoch checking is given in pseudo-code in the Appendix. The initiator sends a request to all nodes. Each node responds with its state. Upon receiving all responses, the coordinator checks if it has been able to obtain a quorum of responses (not counting `RPC.CallFailed`) over the epoch list from a response with the maximum epoch number. If this is the case, the coordinator constructs a new epoch list that includes all nodes that responded, and checks if the new list is different from the current epoch. If the new list is different, the coordinator sends the new epoch list and epoch number to all members of the new epoch. Atomic commit [8] is used to ensure all-or-nothing execution as well as the atomicity of the epoch change with regard to the read and write operations. (It includes locking, but because the epoch checking operation changes the epoch only if any failures or repairs have occurred since the previous epoch check, it happens very infrequently.) The coordinator also determines what replicas are out-of-date and initiates the propagation procedure at the nodes that are current.

4.4 Proof of Correctness

We are going to prove that the protocol above indeed provides one-copy serializability. As mentioned in section 3, it is sufficient to show that (a) neither two write operations nor a read and a write operation can perform 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. As a preliminary note, it is easy to see that nodes with the same epoch number all have the same epoch lists of which they all are members: the epoch checking operation always sends the same pair (epoch number, epoch list) to all nodes from the new epoch list, and the pair is updated atomically at each node. So, we will say interchangeably that a node has epoch number e, or a node is a member of epoch e, or it is a member of epoch list e. Similarly, we will say "quorum from epoch e" meaning a quorum of nodes with epoch number e over the epoch list stored at those nodes. We will split the proof into several lemmas.

Lemma 1. At all times, only nodes with the maximum epoch number can form a quorum over their epoch.

Proof. By induction on the maximum epoch number e present in the system. (i) If $e = 0$, the lemma is obvious (all nodes are in the epoch e). (ii) Assume the lemma is correct for some $e > 0$. Then, there does not exist a set of nodes with some epoch number $l < e$ such that it includes a quorum over epoch l. Let the next epoch checking operation create a new epoch with epoch number $e + 1$. The epoch checking operation succeeds in forming a new epoch only if it successfully writes a new epoch number and epoch list on a set of nodes that includes a quorum over some earlier epoch. By the inductive assumption, it can only be epoch e. Then, due to the intersection property of quorums, any quorum of nodes over epoch e would have to include at least one node with epoch number $e + 1$. Thus, no quorum of nodes with epoch number e over epoch list e exits. Because an epoch checking operation never writes earlier epoch numbers on any nodes, the sets of nodes with epoch numbers less than e can only shrink. Hence, after the epoch checking operation completes, it is still true that no quorum of nodes with epoch number less than e over the corresponding epoch exists. So, the only epoch that may contain a quorum is epoch $e + 1$. □

Lemma 2. Write operations as well as read and write operations cannot perform in parallel.

Proof. Due to Lemma 1, only one epoch may contain quorums at any time. Hence, during the time when epochs do not change, any operation has to lock a quorum of the same epoch. But by the intersection property, any two write quorums as well as read and write quorums have at least one node in common. Thus, the lemma is correct during the time when epochs do not change. In addition, because changing the epoch involves locking at least a quorum of some existing epoch (which can only be the same epoch in which reads and writes can succeed, by Lemma 1), changing the epoch cannot be done in parallel with any read or write. Therefore, read or write operations certainly cannot perform in parallel during epoch changing. In fact, they cannot perform during that time at all. □

Lemma 3. Writes are always applied to and reads always return the most recent version of the data.

Proof. We will give the proof for writes only. The proof for reads is similar. We showed in Lemma 2 that writes are serializable. Therefore, one can use induction on the sequential number of the write operation. (i). Before the 1-st write, the lemma is trivially correct (all replicas are current). (ii) Assume that the lemma is correct before the i-th write and consider the state of the system after this write. We need to show that the $(i + 1)$-st write will apply to the current replica. Case 1: no epoch change occurred between the i-th and $(i+1)$-st writes. Then, the $(i+1)$-st write must collect a quorum of the same epoch as the i-th write. Hence, these quorums will have at least one node in common. By the induction hypothesis, the i-th write correctly identifies the current replicas of the data and marks all other replicas from the quorum stale. Because the replicas marked stale by the i-th write are given the desired version number equal to the the version number the current replicas would have after the i-th write, the former replicas can accept propagation from the current replicas only. Hence, by the time the $(i + 1)$-st write arrived, the common node is either current or is marked stale and has the desired version number equal to the version number of the current replicas. If the common node is not stale, it has the greatest version number and is correctly identified by the $(i + 1)$-st write as current; if the common node is stale, the $(i + 1)$-st write can succeed only if the quorum it collected includes a non-stale replica with the version number greater or equal to the desired version number of the common node. Because the desired version number of the common node is equal to the version number of the current replicas, any non-stale replica with a version number not less than that is current.

Case 2: The epoch change occurred between the i-th and $(i+1)$-st writes. After the epoch change, all nodes from the new epoch are either current or marked stale. To succeed, the $(i+1)$-st write must collect a quorum over the new epoch that includes at least one non-stale replica. Because the $(i + 1)$-st write is the first write

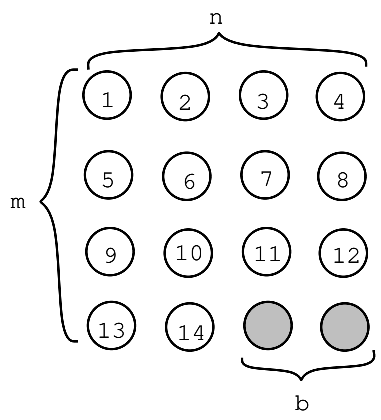

Figure 1: The grid for N = 14.

to occur in the new epoch, any non-stale replica is current. □

5 Example: Dynamic Grid Protocol

The grid protocol [3] arranges all nodes having a replica of the data item into a logical $m \times n$ grid (see Figure 1). To succeed, a read operation must collect permissions from a set of nodes such that one node is selected from each column of the grid. The write operation must collect permissions from a set as above plus from all nodes from some column. For example, in the grid in Figure 1a, a set of nodes $\{1, 6, 3, 7, 11, 4\}$ is a write quorum because it includes a set $\{1, 6, 3, 4\}$ of representatives from each column and a set $\{3, 7, 11\}$ that covers all nodes from some column. The nodes from V are assigned positions in the grid in the increasing order (columns first, see Figure 1). It is easy to see that the set of above sets satisfies the definition of coterie, so those sets are indeed read and write quorums. All we have to do to make this protocol dynamic and applicable for use in systems in which writes update only a portion of the data item is come up with a coterie rule and quorum function and stick them into the protocol from section 4. As a first step, we show how a grid with m rows and n columns can be constructed given an ordered set of nodes V. Let N be the number of nodes in V. The following are desirable (and contradicting) properties of the grid: (1) $m + n$ is to be as small as possible. This is the size of the write quorums. The fewer nodes it includes, the better load sharing and message traffic. (2) $\frac{m}{n}$ must be as close as possible to some k which is a parameter of the system. This ratio determines relative performance and availability of read and write operations. Increasing k, one makes reads more efficient and writes less available [3]. In practice, it is desirable to keep it around 1 and use k just to choose between cases like 5×6 and 6×5 grids. (3) If $m \times n \geq N$, we want $m \times n$ to be as small as possible because the unused $(m \times n - N)$ positions in the grid are equivalent to unoperational nodes with regard to collecting a set of representatives from each column. (4) If $m \times n \leq N$ we want $m \times n$ to be as big as possible, because $(N - m \times n)$ nodes in the resulting grid will not be used.

The various tradeoffs introduced by these requirements deserve a separate study ([13] is an example of research in this area). All we are concerned about here is to propose a reasonable rule that constructs the grid unambiguously, so that all nodes agree on the same grid given the same V. We construct the grid in which $m \times n$ is always greater or equal to N; requirement (1) takes precedence over the requirement that $m \times n$ be the smallest possible; maintaining $\frac{m}{n}$ close to 1 takes precedence over the requirement that $m \times n$ be minimal (we allow m and n to differ by at most 1); when choosing between $n \times (n+1)$ and $(n+1) \times n$ grids, the rule chooses the former. The *DefineGrid* subroutine in Figure 2 returns dimensions of the grid m, n and the number of unoccupied positions b. It uses the fact that, among all m, n such that $m \times n = N$, $m + n$ is minimum when $m = n = \sqrt{N}$.

It can be shown that the highest availability is achieved if the fewest possible number of rows is "spoiled" by unoccupied positions. It is also easy to see that, for the parameters returned by *DefineGrid*, b is always less than n and therefore at most one row in the grid is "spoiled". We assume that the unoccupied positions are all in the bottom row and right-justified. Now we are ready to give the coterie rule that, given sets

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

Figure 2: The algorithm for defining the parameters of the grid.

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

Figure 3: The write coterie rule.

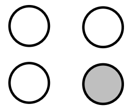

Figure 4: The grid for N = 3.

of nodes V and S, tells if S includes a quorum from the coterie over V. The algorithms *IsReadQuorum* and *IsWriteQuorum* first determine parameters of the grid defined by V. Then, *IsReadQuorum* checks if S includes a representative from each column of the grid. *IsWriteQuorum* does the same plus checks if S covers completely the nodes from one of the columns. *IsWriteQuorum* algorithm is shown on Figure 3. *IsReadQuorum* can be obtain by disregarding the part that involves the variable COLUMNS. Due to the lack of space, we will assume that quorum function returns random quorums (recall that the choice of quorum function affects load sharing only).

6 Availability

We now give an analysis of the write availability provided by the dynamic grid protocol as compared to the static one. Because of space limitations, we omit the analysis for read availability which is completely analogous. We will use the *site model* of availability [11] because a similar model is used in [3] for the static grid protocol with which we compare our results. The assumptions in the site model are [15]: (1) communication links are reliable, so only sites can fail; (2) the failures and repairs at the various nodes are independent Poisson processes with rates λ and μ respectively; (3) no failures or repairs can occur during any operation (operations are instantaneous). In addition, we assume that (4) epoch checking operations run frequently compared to failures and repairs (after any failure or repair, epoch checking always runs before the next failure or repair). The last assumption replaces the more restrictive assumption that write operations arrive frequently, which was used in [11] to analyze the availability of the dynamic voting protocol. Finally, for fairness of comparison, we assume in this section that, like the static grid protocol in [3], our protocol is to support total writes only. Hence, update propagation is not needed as all replicas participating in a write operation receive the new value regardless of how current they were before the operation.

The analysis uses Markov chains and goes along the lines of [11]. Initially, all N nodes are in the latest epoch. As nodes fail and get repaired, the epoch checking operation, according to the site model assumptions, instantaneously updates the latest epoch. Due to the assumption (4) of the site model of availability, only one failure may occur between two consecutive epoch checking operations. Because any grid constructed in our protocol that contains at least four nodes tolerates a single failure, the above process of epoch changes continues successfully unless the system comes to the point when there are only three nodes in the latest epoch and one of them fails. Subsequent epoch checking operations will fail to collect a quorum over the latest epoch until all three nodes become simultaneously available again. (The grid for $N = 3$ is shown in Figure 4. One can see that all three nodes are needed to collect a quorum.) So, no matter how many other nodes get repaired, the epoch cannot change and the system stays unavailable. Once all three nodes from the latest epoch are repaired, the new epoch can be formed to include all other nodes that are up at the moment. The state diagram is shown in Figure 5. State (x, y, z) is the state in which the latest epoch contains y nodes; x of the y nodes from that epoch are up; z of the $N - y$ other nodes are up. The system is available if it is in one of the states in the upper row. (We assume that operations can be initiated by the operational nodes only. This assumption is also used in [3] which is the basis for the comparison of our results. In cases where this assumption does not hold, it makes no sense to replicate data because at best one can achieve an availability equal to the probability that a single node is up. But that is the availability of a non-replicated system.) We use the classical *global balance* technique (see, for example, [17]) to solve the diagram, i.e., to find the probabilities of the system being in particular states. Then, the availability of the system, which is equal to the probability that the system is in one of the states in the upper row of the diagram, is calculated as the sum of the individual state probabilities. For compatibility with [3], the diagram has been solved for the same probability that a node is up $p = 0.95$ which is achieved when $\mu/\lambda = 19/1$. Table 1 shows the best

Figure 5: The state diagram for the dynamic grid protocol.

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

Table 1: Unavailability of conventional and dynamic grid with p = 0.95

write *unavailability* (i.e., 1 - availability) achieved by the conventional grid protocol for various number of nodes taken from [3] (it can vary depending on the dimensions of the grid) and the unavailability provided by our protocol. The improvement is several orders of magnitude and is achieved while preserving the good load sharing and message traffic characteristics of the conventional grid protocol. Additional traffic caused by periodic epoch checkings should not be noticeable because epoch checking is an infrequent operation.

7 Summary

There are two main results in this paper. First, the applicability of structured coterie protocols is expanded to include systems in which write operations update only a portion of a data item. Hence, these high performance protocols now can be used with any file or database system. Second, the availability of such protocols is improved by dynamically adjusting set of nodes over which the coterie is defined. Therefore, the approach developed in this paper addresses two major limitations of the previous structured coterie protocols without compromising performance in the common case of absence of failures and light sharing. (The only additional work needed in this case is periodic non-locking polling of all nodes by the epoch checking operation and asynchronous update propagation.)

Although structured coterie protocols are the direct concern of this paper, the dynamic voting protocol can also benefit from the proposed approach. The approach makes it possible to separate read and write operations from the epoch changing operation. Thus, work is decoupled from reads and writes and the rate of changing view operations become steady. Also, the proposed approach allows reads and writes in the dynamic voting protocol to poll only quorums of nodes rather than all nodes in the normal case, therefore improving load sharing and message traffic in the system.

Acknowledgements

We would like to thank Tom Anderson, Irina Bam, and Clifford Neuman for useful discussions of the issues presented in this paper. In particular, C. Neuman suggested that the desired version numbers be taken into account in deciding whether the quorum of responses contains the most recent replica. This suggestion improves performance of the write protocol. He also noted that the write quorums in the grid protocol need include only the part of a grid column that corresponds to physical nodes.

References

[1] D. Agraval and A. El Abbadi. An efficient solution to the distributed mutual exclusion problem. In *Proc. of ACM Symp. on Principles of Distributed Computing*, pp. 193-200, 1989.

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

[3] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The Grid protocol: a high performance scheme for maintaining replicated data. In *Proc. of the IEEE 6th Int. Conf. on Data Engineering*, pp. 438-445, 1990.

[4] A. El Abbadi, D. Skeen, and F. Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proc. of the 4th ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pp. 215-229, 1985.

[5] A. El Abbadi and S. Toueg. Availability in partitioned replicated databases. In *Proc. of the 5th ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pp. 240-251, 1986.

[6] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Trans. Database Syst.* 14(2), pp. 264-290, June 1989.

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

[8] H. Garcia-Molina. Reliability issues for fully replicated distributed databases. *IEEE Comput.* 15(9), pp. 34-42, September 1982.

[9] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Computers,* C-31(1), pp. 48-59, January 1982.

[10] 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.

[11] 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.

[12] A. Kumar. Performance analysis of a hierarchical quorum consensus algorithm for replicated objects. In *Proc. of 10th Symp. on Distributed Computing Sys.,* pp. 378-385, IEEE, 1990.

[13] A. Kumar and K. Malik. Generalizing and optimizing hierarchical quorum consensus algorithms for replicated data. Graduate School of Management, Cornell University, Tech. Report, October 1991.

[14] T. Mann, A. Hisgen, and G. Swart. *An algorithm for data replication.* Research Report 46, DEC Systems Research Center, June 1989.

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

[16] M. Rabinovich and E. Lazowska. Efficient support for partial write operations in a replicated data storage system. Submitted for publication.

[17] K. Trivedi. *Probability and statistics with reliability, queueing, and computer science applications.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

Appendix. The algorithms for write, propagate, and epoch checking operations

```
Write(input:  update-data)
/* This algorithm is run by the coordinator.  The
actions taken by the other nodes are given within
the comments.  */
   multicast(quorum-list, ``write-request'');
/* each node that receives the write-request obtains the
lock for its replica and responds with a tuple of the
form (node, version, dversion, stale, elist, enumber),
where the elements in the tuple are the node name,
version number, desired version number, stale data flag,
epoch list, and epoch number respectively.  */
   receive RESPONSES[1:n]; /* a set of
non-RPC.CallFailed responses in the form of (node_i,
version_i, dversion_i, stale_i, elist_i, enumber_i), i = 1,...,n.  */
/* Find a response with the maximum epoch number.  */
   m := an index j such that
      enumber_j = max_{i=1,...,n}{enumber_i};
   if coterie-rule(elist_m, {node_1,...,node_n}) then
/* find the greatest version number among responses from
non-stale replicas and the greatest desired version
number among responses from stale replicas.  */
      max-version := max_{i=1,...,n}{version_i|stale_i = 0};
      max-dversion := max_{i=1,...,n}{dversion_i|stale_i = 1};
      if max-dversion > max-version then /* RESPONSES do
not contain the response from a current replica */
         HeavyProcedure;
      else
/* GOOD and STALE are a set of the current replicas
and a set of replicas to be marked stale respectively */
         GOOD := {node_i|stale_i = 0
            and version_i = max-version; i = 1,...,n};
         STALE := {node_i; i = 1,...,n } \ GOOD;
         try-atomically /* ensures all-or-nothing
execution */
            multicast(GOOD, (``do-update'', STALE));
/* Upon receiving this message, each node from
the GOOD set performs the write and increments its
version number.  If the action is committed, the nodes
from the GOOD set unlock their replicas and, if STALE
is not empty, start the propagation protocol.  */
            multicast(STALE,
               (``mark-stale'', max-version + 1));
/* Upon receiving this message, each node sets its
stale-data flag to 1 and updates its desired version
number to be equal to the version number from the
message.  */
         if-failed
            HeavyProcedure;
         end-try-atomically;
      endif;
   else
      HeavyProcedure;
   endif;
end;
HeavyProcedure
   multicast(all-nodes-that-have-a-replica,
      ``write-request'');
   receive RESPONSES[1:k]; /* a set of
non-RPC.CallFailed responses in the form of (node_i,
version_i, dversion_i, stale_i, elist_i, enumber_i), i = 1,...,k.  */
   m := an index j such that
      enumber_j = max_{i=1,...,k}{enumber_i}
   max-version := max_{i=1,...,k}{version_i|stale_i = 0};
   max-dversion := max_{i=1,...,k}{dversion_i|stale_i = 1};
   if coterie-rule(elist_m, {node_1,...,node_k}) and
```

```
      max-version >= max-dversion then
/* GOOD and STALE are a set of the current replicas
and a set of replicas to be marked stale.  */
      GOOD := {node_i|stale_i = 0 and
         version_i = max-version; i = 1,...,k};
      STALE := {node_i; i = 1,...,k } \ GOOD;
      try-atomically
         multicast(GOOD, (``do-update'', STALE))
/* The actions of a node upon receiving ``do-update''
and ``mark-stale'' messages are described above.  */
         multicast(STALE, (``mark-stale'',
            max-version + 1));
      if-failed
         abort;
      end-try-atomically;
   else
      abort
   endif;
   quorum-list := quorum-function(elist_m , my-node-name);
end;



Propagate(input:  STALE-NODES)
/* Let v be the version number of the coordinator's
replica */
while STALE-NODES ≠ ∅ do
   foreach node ∈ STALE-NODES do
      send-message(node, (``propagation-offer'', v);
/* Each node that receives this message, runs
PropagateResponse algorithm below */
      receive response;
      case (response):
      ``i-am-current'':
         STALE-NODES := STALE-NODES \ {node};
      ``already-recovering'':
         pause(some-time); /* delay and repeat the
propagation offer later to make sure the recovery was
successful.  */
      ``propagation-permitted'':
         perform propagation; /* The specific way of
doing propagation depends on data organization in the
system.  At the end of the propagation, both the
coordinator and the target node unlock their
replicas.  */
      endcase;
   endforeach;
endwhile;
end;
PropagateResponse(input:  v /* the version number of the
            source replica */)
/* This algorithm is executed by a node upon receiving a
propagation-offer message.  Assume each node has a
locked-for-propagation bit, originally 0.  */
   if locked-for-propagation = 1 then
      reply(``already-recovering'');
   else
      if local-replica is locked then
         lock(local-replica);
         if (stale-data = 1 and
            desired-version-number <= v then
            locked-for-propagation := 1;
            reply (``propagation-permitted'');
         else
            unlock(local-replica);
            reply(``i-am-current'');
         endif;
      else
         if stale-data = 1 and
            desired-version-number <= v then
            lock(local-replica);
```

```
            if stale-data = 1 and
               desired-version-number <= $v$ then
               locked-for-propagation := 1;
               reply(``propagation-permitted'');
            else
               unlock(local-replica);
               reply(``i-am-current'');
            endif;
         else
            reply(``i-am-current'');
         endif;
      endif;
   endif;
end;
```

```
CheckEpoch
/* The epoch checking operations must be mutually
exclusive.  For simplicity, the corresponding locking
is not shown.  This algorithm is run by the initiator of
the epoch checking.  The actions taken by the other
nodes are given in the comments.  */
   multicast(all-nodes-that-have-a-replica,
      ``epoch-checking-request'');
/* each node that receives the epoch-checking-request
responds with a tuple of the form
$(node, version, dversion, stale, elist, enumber)$, where the
elements in the tuple are the node name, version number,
desired version number, stale data flag, epoch list,
and epoch number respectively.  */
   receive RESPONSES[1:k];
/* The above is a set of non-RPC.CallFailed responses in
the form of $(node_i, version_i, dversion_i, stale_i,$
$elist_i, enumber_i)$, $i = 1, \ldots, k$.  */
/* find a response with the maximum epoch number.  */
   m := an index j such that
      $enumber_j = \max_{i=1,\ldots,k}\{enumber_i\}$;
   if coterie-rule($elist_m, \{node_1, \ldots, node_k\}$) then
      NEW-EPOCH := $\{node_1, \ldots, node_k\}$;
      if NEW-EPOCH $\neq$ $elist_m$ then
         new-epoch-number := $enumber_m + 1$;
/* find a set of the current replicas (GOOD set) and a
set of replicas to be marked stale (STALE set) */
         max-version := $\max_{i=1,\ldots,k}\{version_i | stale_i = 0\}$;
         GOOD := $\{node_i | version_i$ = max-version and
            $stale_i = 0$; $i = 1, \ldots, k\}$;
         STALE := NEW-EPOCH $\setminus$ GOOD;
         try-atomically /* ensures all-or-nothing
execution and atomicity with regard to read and
write operations */
            multicast(NEW-EPOCH, (``new-epoch'',
               NEW-EPOCH, new-epoch-number, GOOD, STALE));
/* Upon receiving this message, each node updates its
epoch list and epoch number to be equal to NEW-EPOCH and
new-epoch-number respectively.  The node also checks if
it is a member of the GOOD set.  If the action is
committed and the STALE set is not empty, the nodes
from the GOOD set start the propagation protocol.
            multicast(STALE,
               (``mark-stale'', max-version));
         end-try-atomically;
      endif;
   endif;
end;
```