Asynchronous Epoch Management in Replicated Databases *

Michael Rabinovich and Edward D. Lazowska

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

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

1 Introduction

Replication of data is commonly used in distributed databases 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 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, no two write operations can succeed simultaneously thus excluding the possibility of write conflicts. In addition, if every write performs on all replicas from a quorum, a successful read operation is guaranteed to see at least one most recent replica of the data.

Numerous ways to define quorums have been proposed. In the voting-based quorums [7], each node replicating the data item is assigned some number of votes, and two numbers, read and write quorum thresholds r and w, are chosen such that $r + w > v$ and $2w > v$, where v is the total number of votes assigned to nodes that have a replica of the data item. A read quorum is defined to be any set of nodes with a total number

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

of votes equal to or exceeding r, and a write quorum is any set of nodes with a total number of votes at least w. In the simplest case, all nodes are given a single vote and read and write quorums are defined to be any set of nodes that includes a majority of replicas of the data item. In the structure-based quorums [1, 4, 10], all replicas agree on a logical structure imposed on all replicas of the data item, and define quorums based on that structure. When the number of replicas in the system is large, structure-based quorums contain fewer nodes than voting-based quorums and therefore exhibit better performance (since operations involve communicating with all nodes from a quorum). In replicated databases however, small numbers of replicas are typical, in which case the voting based approach is advantageous.

In all quorum based solutions to replica management, any quorum of replicas being down makes the data item unavailable, that is, even though there are up-to-date operational replicas available, access to the data is denied. Therefore, many protocols have been proposed in which quorums are adjusted to reflect failures and repairs occurring in the system. Then, if failures are mostly sequential so that the protocol can adjust to the failures as they arrive, the data item will be kept available as long as some small number of replicas (the number depends on the specific quorums used) are up and connected. In one such protocol, the dynamic voting protocol [9], the quorum re-adjustment is done within the protocol for the write operation. This approach eliminates the need for a separate mechanism for failure detection and quorum re-adjustment. On the other hand, failures and repairs are detected only at the time of write operations. Therefore, when the rate of writes gets low, failures may accumulate unnoticed by the quorum adjustment mechanism. Fixing this problem by running "dummy" writes as suggested in [9] can hurt performance since the "dummy" writes may interfere with and delay "normal" read and write operations. Also, this scheme involves communicating with all replicas on every write operation as opposed to a write quorum of replicas and thus makes writes less efficient.

In [5, 12] and in the group consensus protocol of [2], the check for changes in the system topology is done as a separate operation asynchronously with regard to user reads and writes. In addition, the views protocol [5] can be used in conjunction with read-one write-all quorums, and the epochs protocol [12] is suitable for structure-based quorums. However, while checking for changes in system topology in these protocols is asynchronous with reads and writes, actual system re-adjustment is not (i.e., it runs as a special operation constrained by the same concurrency control mechanism as user operations). Thus, it can interfere with user operations. This problem may become severe in the case of an unstable network where transient failures and repairs may cause frequent (and intrusive) system re-adjustments. On the other hand, user operations in these schemes may delay system reconfiguration. Sometimes, when the rate of user operations is high, or when failures occur during the execution of a user operation, this delay can be very long, which can have a negative effect on system availability.

In this paper, we propose a protocol in which both the detection of the changes in the system topology and the system re-adjustment are done completely asynchronously with reads and writes. Thus, user operations can be executed while the system reconfiguration is in progress, and no interference is possible. In addition to the traditional problem of system re-configuration in the aftermath of failures and repairs, the ability of our protocol to include/exclude nodes asynchronously with user operations allows it to be applied in other areas also. For example, in a mobile computing system, if a user moves from Seattle to New York, our protocol would allow the set of servers on which the user's data is replicated to change asynchronously, without any service interruption, to a set of servers that are close to the current user location. Thus, the user's data could smoothly follow the user as (s)he moves. Previously, such a change in the set of servers would require delaying service while the re-configuration is in progress. Another possible application is asynchronous file migration in a wide area shared-data environment, where data is moved close to the most frequent user, with data location following the access patterns. Again, the fact that such migration does not involve any service interruption allows for more flexible policies to be used.

Our scheme uses the notion of an *epoch* [12], which is a set of replicas of the data item known at some point to be operational and connected. Originally, all replicas of the data item form the current epoch. The system periodically runs a special operation, epoch checking, that polls all the replicas. If any members of the current epoch are not accessible (failures detected), or any nodes outside of the current epoch are contacted (repairs detected), an attempt is made to form a new epoch. (Epochs are distinguished by their epoch numbers, with later epochs assigned greater epoch numbers). For this attempt to be successful, the new epoch must contain a write quorum of replicas drawn from the previous epoch. Then, due to the intersection property of write quorums, it is possible to guarantee that, if the system partitions, the attempt to form a new epoch will be successful in at most one partition. Therefore, read and write operations can be allowed to succeed if they obtain permission from a read or write quorum of replicas drawn from the current epoch rather than from all replicas in the system, and data items will remain available as long as they have at least one replica included in the current epoch.

The novelty of our approach is in the method of switching to a new epoch that allows reads and writes to proceed while the epoch switching is in progress. In our scheme, all replicas keep control information that restricts the ways in which quorums may be formed. When switching from the current epoch to a new epoch, the protocol specifies transitional values of the control information in such a way that any quorum formed according to the transitional control information includes a quorum from the current epoch as well as a quorum from the new epoch. Then, the epoch change is done in two stages. First, transitional control information is asynchronously recorded on all members of the new epoch. During this stage, some read and write operations may use quorums from the current epoch while others may used quorums defined by the transitional control state. However, since the latter quorums always include the former ones, all operations will in effect execute in the current epoch. Once a write quorum of replicas from the current epoch confirm switching to the transitional state, the second stage of the protocol begins, during which replicas asynchronously switch to the new epoch. During the second stage, some reads and writes may use "transitional" quorums while some may already use quorums from the new epoch. But since any "transitional" quorum includes a quorum from the new epoch, all operations will in effect execute in the new epoch. Thus, the overall effect of the protocol execution is as if the system switched from the old to the new epoch instantaneously at the moment when the second stage of the protocol begins. Then, with a little care in a situation where a user operation obtains permission from a quorum of replicas before and executes after "instantaneous" epoch switching, the consistency of replicated data can be guaranteed.

2 Related Work

Previously, in [2], the problem of asynchronous system re-configuration was solved only for a specific method of quorum adjustment where a node independently modifies the number of votes it is assigned. As pointed out in [2], because the view of the system at each node may be different and because each node operates independently, the resulting quorums may not be optimal. In particular, the vote assignment of a failed node cannot be changed. The advantage of the method is that it is simple and fast.

Another scheme, proposed in [8], involves pre-computing different quorum assignments in advance and storing these assignments at all replicas. A natural number (a *level*) is associated with each assignment. When nodes fail and a transaction cannot collect a quorum of level n, it can move to higher levels, without *any* global reconfiguration procedure at all. However, when nodes repair, transactions are not allowed to move back to lower level quorums. Thus, to include the repaired nodes back in the voting process, new quorum assignments must be redistributed to replicas. This is done as a special operation, which is mutually exclusive with user operations. So, in this protocol, quorum re-adjustment after failures is asynchronous (in fact, no procedure is needed in this case), while re-adjustment after repairs is not. In our protocol, quorum re-adjustment after both failures and repairs is done asynchronously.

3 The Model

The protocol presented here suits both the client-server model of distributed computation and the "symmetrical" model in which nodes that initiate operations are among the nodes having a replica of the data item. We refer to nodes that initiate operations as "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.

Each node is assumed to have a unique name. Nodes and communication links may fail, and this may cause partitioning of the system. Failures are of *fail-stop* type, that is, nodes and communication links may crash but do not behave maliciously. Messages can be delivered out of order, or may not be delivered at all. However, we assume RPC-style communication in which the notification `RPC.CallFailed` is returned to the sender if the message cannot be delivered. We rely on every node having a local clock, with no assumptions about network-wide clock synchronization.

In this paper, we assume that write operations are *total*, that is, they always replace the old value of the data item entirely with a new value rather than updating a portion of the information contained in the data item[1].

[1]The idea described in this paper can be equally used for a variant of the protocol that would support partial writes. We have chosen the total writes semantics in order to avoid irrelevant details.

We use one-copy serializability as the criterion for consistency: the system is consistent if the concurrent execution of transactions on replicated data is equivalent to a serial execution of these transactions on non-replicated data [3]. We assume that strict two-phase locking [3] is used to ensure that the execution of transactions is serializable on the level of physical operations on replicas, that is, it is equivalent to some serial execution. So, the purpose of a replica control protocol is to ensure *one-copy* serializability.

4 The Protocol

Our protocol consists of three asynchronous procedures: read/write (referred to as user operations), epoch checking and switching to a new epoch if necessary (called below the epoch operation or epoch changing operation, in the case when it actually changes epochs), and recovery. To highlight the main idea and simplify the proofs, we describe here a bare-boned protocol where each node is assumed to have a single vote and read and write quorums are majority sets of nodes. Also, for the same reason, we do not address here the vulnerability of the epoch changing mechanism to failures that occur during the transition to the new epoch. The extension that addresses the tolerance of epoch changing operation to failures during the operation's execution as well as the extensions that generalize our protocol to weighted voting and non-equal read and write quorum thresholds are described in [13].

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

Finally, i maintains a log where it records identities of operations that locked at some point but then released i's replica. The log records referring to an operation p are purged by a distributed garbage-collector when all nodes ever locked by or on behalf of p have been released. Hence, a node keeps the record for p only from the time its lock is released by p till the time when all locks held by p are released. Since we assume a strict two-phase locking where all locks acquired by a transaction are released at once after the transaction commits, this period of time is typically very short, and the log contains very few (if any) records.

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

The protocol as described here does not address the deadlock problem. One simple way to handle deadlocks is to use timeouts and Ethernet-style exponential backoffs approach [11]. For other ways that can easily be integrated with our protocol, see for example [3].

4.1 Overview

We begin by describing informally the ideas behind our protocol.

As in the existing dynamic voting protocols, a new epoch is required to contain a majority from the current epoch, which ensures that, should the system partition, at most one new epoch can be constructed. The method of forming the new epoch will be discussed shortly.

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

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

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

The epoch changing operation will first update *ELIST* and *total-votes* kept by the members of the prospective epoch to the transitional values. Thus, in this stage, replicas x_1, x_2, and x_3 will get transitional *ELIST* and *total-votes* and keep their old epoch number. While this is being done, some reads and writes may still use quorums from the old epoch, and some may already use transitional quorums. However, since the only possible transitional quorum includes also a quorum from the old epoch, all operations in effect continue using quorums from the old epoch. Only after the transitional values are reflected in a quorum of replicas from the old epoch (in our case, after x_1, x_2, and x_3 confirm adopting the transitional state), the second stage of the protocol begins, during which the state of the members of the new epoch is updated to the new epoch parameters ($ELIST = \{x_1, x_2, x_3\}$, $enumber = 1$, and *total-votes*$=3$). Any time during this stage, all nodes from some write quorum from the old epoch have either transitional or new epoch state. Thus, any successful user operation that executes after this stage begins is guaranteed to see either transitional or new values of epoch parameters, and therefore use either transitional or new quorums. Since any (or, in our case, the only) transitional quorum includes a quorum from the new epoch, all user operations effectively use new quorums during this stage. The only difference is that operations using transitional quorums must obtain one extra permission.

This is equivalent to instantaneous epoch switching at the time when the second stage begins. Hence, if at this time there are no user operations in progress, the consistency of the replicated data is guaranteed, as in the existing schemes where epoch switching is mutually exclusive with user operations. The situation where the second phase of epoch switching begins while a user operation is underway is considered next[2].

Assume that the epoch changing operation from the previous example finds x_3 to be locked for a write operation, i.e., x_3 has given its permission for a write operation to some write coordinator and is waiting for the message from that coordinator instructing x_3 to perform the update (the *update* message). Furthermore, replicas x_1 and x_2 are found to have version number 1, while x_3 has version number 0. Several scenarios could have led to this situation. For example, in one scenario, a write w_1 executed on replicas x_1, x_2, x_5 and gave them version number 1, then another write, w_2, collected permission from a quorum $\{x_3, x_4, x_5\}$, performed on x_4 and x_5 (giving them version number 2), and then, while the update message from w_2's coordinator to x_3 is in transit, x_4 and x_5 get disconnected and the epoch operation runs. In another scenario, a write w_1 collected permission from a quorum $\{x_1, x_2, x_3\}$, performed on x_1 and x_2 (giving them version number 1), and then, while the update message to replica x_3 is in transit, x_4 and x_5 get disconnected and the epoch operation runs.

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

Note that in this approach, the user operations never have to wait for the epoch operations (hence, epoch changing never interferes with user operations). Moreover, since the epoch changing operation must wait for only those user operations on a given replica that locked the replica *before* it adopted the transitional state (and replicas adopt the transitional state immediately when the message carrying this information arrives), there is no possibility of starvation of epoch changing operations. The advantage of this approach is in its simplicity: the protocol using this approach requires only three phases [13]. The drawback is that if a user operation coordinator fails before unlocking some nodes, epoch changing operations may be blocked for a long time.

---

[2]Due to the lack of space, we do not consider an example where newly repaired nodes are included in the new epoch. The reader is encouraged to work through such an example as well as the case where some repaired nodes are being included into the new epoch while some newly failed nodes are being excluded at the same time using the general description of the protocol in section 4.2.

The other way to handle this problem, first sketched in [2] to deal with a similar situation and used in our protocol here, is to propagate x_3's lock to x_1 and x_2 and assign x_3 the responsibility to unlock x_1 and x_2 after its lock is released. Also, x_1 and x_2 can unlock themselves by contacting a node that committed the operation holding their locks. In this approach, one should be careful to avoid excessive lock propagation. In our example, if the system evolved according to the second scenario, x_1 and x_2 already finished the write that is locking x_3. Therefore, propagating x_3's lock to them is is not only unnecessary, but harmful, since if x_3 fails before unlocking either of the remaining replicas, the data item will be permanently blocked for future accesses until x_3 recovers. We use the logs to distinguish between the two scenarios. In the second scenario, x_1 and x_2 will find a record referring to w_1 in their logs, and will not accept lock propagation. In the first scenario, there will be no record referring to w_2 in the logs, and both x_1 and x_2 will mark themselves as locked for w_2.

4.2 Epoch Changing

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

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

During the first phase, the coordinator sends a request to all nodes. Each node, if it is not already participating in another epoch operation, responds with its epoch parameters. (Otherwise it ignores the request.) Upon receiving all responses, the coordinator checks if it has been able to obtain responses from a write quorum as defined by the epoch parameters from a response with the maximum epoch number. (We denote these epoch parameters as $enumber_m$, $ELIST_m$, and $total\text{-}votes_m$ below.) If this is the case, $ELIST_m$ and $total\text{-}votes_m$ are in fact the epoch parameters of the current (most recent) epoch. Once the coordinator obtains the current epoch parameters, it checks whether a set of responded nodes, $ELIST_{new}$, is different from the current epoch, $ELIST_m$, in which case the coordinator attempts to form a new epoch by initiating the second phase of the protocol. Note that since nodes respond to an epoch operation coordinator only if they are not participating in other epoch operations, at most one epoch operation coordinator at a time can obtain enough responses to proceed to the second phase. In effect, our protocol uses a standard dynamic voting approach (see, e.g., [2]) to ensure that epoch changing operations are mutually exclusive.

Phase 2. In this phase, the coordinator distributes the *transitional values* of the *total-votes* and *ELIST* to all members of the prospective epoch. The transitional *total-votes* includes votes from all nodes from the current epoch (both operational and failed) as well as the new nodes outside the current epoch that will be members of the prospective epoch, while the transitional value of *ELIST* includes only those nodes from the current epoch that will also be members of the prospective epoch and does not include either failed nodes from the current epoch or the new nodes. Formally, let *CURRENT-NODES* = $ELIST_m \cap ELIST_{new}$ and *NEW-NODES* = $ELIST_{new} \setminus$ *CURRENT-NODES*. Then, $ELIST_{trans}$ = *CURRENT-NODES*, and *total-votes*$_{trans}$ = *total-votes*$_m$ + (the number of nodes in *NEW-NODES*).

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

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

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

$$FILTERED\text{-}OP\text{-}LIST_i = COMBINED\text{-}OP\text{-}LIST \setminus LOG_i,$$

where LOG_i is the replica's log.

Phase 4. The fourth phase begins when the coordinator gets filtered lists from all members of the prospective epoch. If some operation was filtered by any participant, then that operation is in the lock releasing stage. Hence, its locks should not be propagated. Therefore, to find operations whose locks must be propagated, the coordinator computes the intersection of all the filtered lists it received: *FINAL-OP-LIST* = $\cap_i$ $FILTERED\text{-}OP\text{-}LIST_i$. Then, the coordinator distributes *FINAL-OP-LIST* to all members of the prospective epoch $ELIST_{new}$. For every operation from the list received, a participant checks if its replica is not already locked for the operation, and if there is still no record of this operation in its log, which would mean that a replica is done with the operation. If both conditions are met, the replica marks itself locked for the operation. Otherwise, the replica makes a note that when its lock is no longer held by the operation, the lock release should be propagated to other members of the new epoch. The participants conclude this phase by sending an acknowledgement to the coordinator.

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

No log records should be garbage-collected on a node from the time it sent the filtered list of current operations to the coordinator until the time it finishes its participation in the protocol [3]. Also, during phases 3 or 4, if the coordinator fails to receive response from some of the participants, it distributes an abort message, which causes all participants to terminate the protocol and restore their original state. Thus, in effect, the epoch changing algorithm embodies the two-phase commit protocol [6] to ensure that epoch switching is done in an all-or-nothing manner. (However, no atomicity of switching to a new epoch with regard to read and write operations is implied.) As already mentioned, this makes the epoch changing mechanism vulnerable to failures of the coordinator that occur during the transition to the new epoch, since such a failure may leave the system in a state where no further epoch operations are possible. To deal with this problem, one can incorporate the fault-tolerant *three-phase commit* [14] protocol into the epoch changing algorithm above. (See [13] for implementation details of fault-tolerant epoch changing.)

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

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

4.3 User Operations

Read and write coordinators determine whether they collected a quorum of permissions using the most recent values of epoch parameters obtained along with permissions from replicas. The required number of permissions is determined by the most recent

[3] Otherwise, the situation may occur when, e.g., in the first scenario from section 4.1, w_2 was holding x_3's lock at the time x_3 constructed a list of current operations in the second phase of epoch switching, but finished and was purged from x_3's log by the time x_3 enters the final phase of epoch switching. Then, x_3 would mark itself locked for an operation it already performed.

total votes number (more than half the total votes is needed), while replicas whose permissions may count towards quorum formation are determined by the most recent epoch list (only members of this epoch list are allowed to participate).

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

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

The algorithm for user operations is simple. To perform a read or write, the operation coordinator requests permission from all replicas (see however the end of this subsection for an easy way of avoiding having to contact all replicas on every operation). Upon receiving this request, a node obtains the local read or write lock for its replica and responds with its epoch parameters, stale flag, and version number. After getting all responses (recall that `RPC.CallFailed` is returned from failed or disconnected nodes), the coordinator identifies the maximum epoch number reported, $enumber_m$. Due to asynchronous epoch changing, epoch lists and total votes numbers from responses with the same epoch number may be not identical: some may contain true values while some may contain transitional values. Responses with transitional values can be identified by the fact that their total votes number is greater than the number of replicas in their epoch list. From the epoch changing algorithm, it is clear that the transitional values of epoch list and total votes number are more recent than true values from responses with the same epoch number. Let $ELIST_m$ and *total-votes*$_m$ be either transitional values from some response with epoch number $enumber_m$ (if any such response contains transitional values), or the epoch list and *total-votes* from any response with epoch number $enumber_m$ (otherwise, i.e., all such responses contain the same epoch parameters). The coordinator is considered to have collected a quorum if it obtained more than half of *total-votes*$_m$ non-`RPC.CallFailed` responses from members of $ELIST_m$.

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

Note that we do not specify when replicas are unlocked since it is done according to strict two-phase locking, at the commit or abort time of user transactions. Further, we are not concerned with a *commit* mechanism that ensures that transactions are done in an all-or-nothing manner[4]. We note however that in a real implementation, messages used by replica control, concurrency control, and commit protocols are often combined to improve an overall efficiency of the system.

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

4.4 Recovery operation

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

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

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

The reason for the target replica to compare the version numbers before adopting the data received is twofold. First, an asynchronous write operation w may have performed on the target replica x_i since the recovery request was sent making x_i non-stale again. Copying data from the source replica x_j in this case could violate the correctness. Indeed, if the data sent from x_j to x_i is older than the data recorded on the x_i by the write w, then the effect of adopting x_j's data by x_i would be as if the write w finished without performing on x_i. But if x_i happens to be part of w's quorum, that would mean that w did not perform on all nodes from its quorum, which is an essential condition

[4] See [3] for two-phase locking and commit protocols.

for the correctness of the protocol. The second reason for comparing version numbers is that the version number kept by x_i might have increased due to an epoch changing operation performed after the recovery was initiated, in which case x_i should not accept data with earlier versions.

4.5 Proof of Correctness

Theorem 1 *If strict two-phase locking is used for concurrency control at the level of physical operations on replicas, then the protocol above allows only one-copy serializable executions of user transactions.*

Proof. See [13].

5 Conclusion

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

Theorem 1 establishes that our protocol guarantees one-copy serializable executions of user transactions provided strict two-phase locking is used as a concurrency control mechanism to ensure serializability at the level of physical replicas. However, not all correct concurrency control protocols can be used. Indeed, one can provide examples of executions that are legal under our replica control protocol, serial at the level of physical replicas, and yet not one-copy serializable. Therefore, while strict two-phase locking is the most common concurrency control protocol, a more general characterization of concurrency control protocols that can be used in conjunction with our replica control protocol remains a question for future work.

References

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

[2] D. Barbara, H. Garcia-Molina, and A. Spauster. Increasing availability under mutual exclusion constraints with dynamic vote reassignment. *ACM Trans. on Comp. Sys.*, 7(4), pp. 394-426, November 1989.

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

[4] 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.

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

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

[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] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Trans. on Database Syst.* 12(2), pp. 170-194, 1987.

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

[10] 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.

[11] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *CACM* 19(7) pp. 395-404, 1976.

[12] 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.

[13] M. Rabinovich and E. D. Lazowska. Asynchronous epoch management in replicated databases. Technical Report 92-12-02. University of Washington Department of Computer Science, January 1993.

[14] D. Skeen. A quorum-based commit protocol. In *Proc. of the 6th Berkeley Workshop on Distr. Data Management and Comp. Networks*, pp. 69-80, 1982.