An Efficient and Highly Available Read-One Write-All Protocol for Replicated Data Management *

Michael Rabinovich and Edward D. Lazowska

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

Abstract

A new read-one write-all (ROWA) protocol for replicated data is proposed that allows a system to adjust to failures dynamically in order to keep the data available. If failures arrive mostly sequentially, our protocol keeps the data available as long as there is at least one operational replica. This is achieved by making the epoch mechanism, previously applicable to non-ROWA schemes only, usable within the ROWA discipline. Also, adjusting the system to a new configuration in our protocol is done completely asynchronously with reads and writes. In contrast, in the existing dynamic schemes (both ROWA and non-ROWA), system reconfiguration may interfere with and delay user transactions.

1 Introduction

A read-one write-all (ROWA) replication scheme is commonly used in distributed systems to improve performance and availability of mostly-read data. In this scheme, the data is replicated in such a way that, for most clients, there is a nearby replica of the data item. A read operation can be done from any replica in the system and therefore is very efficient and available. To insure consistency of the data, a write operation is done on all replicas and hence is very expensive and has low availability. Low write availability is a major limitation of the ROWA scheme.

A read-one write-all algorithm is a particular case of a general quorum-based approach [8] in which every read (write) operation must obtain permission from all replicas from a read (write) quorum set. Read and write quorum sets are sets of replicas of a given data item defined in such a way that any read quorum and any write quorum, as well as any two write quorums, have at least one replica in common. This intersection property makes it possible to enforce a multiple-reader-single-writer model of logical read and write operations on replicated data and hence provide a one-copy image of the data to the applications. This general quorum-based approach reduces to ROWA if one defines a read quorum to be any single replica and a write quorum to be a set of all replicas of the data item. However, most proposals aimed at improving the availability of quorum-based protocols [10, 11, 1] do not apply to the ROWA case. Indeed, these schemes involve changing dynamically the set of nodes from which the quorums are drawn (the *distinguished partition* [10], or the current *epoch* [11]) to reflect failures and repairs occurring in the system. To guarantee consistency, these schemes require that, for the new epoch to be successfully formed, it must contain a write quorum of replicas drawn from an existing epoch. Since a write quorum in ROWA includes all replicas, any failure would make the epoch change impossible and therefore these dynamic schemes would not improve the data availability in the case of ROWA.

In this paper, we propose a scheme for dynamic system readjustment that, like the existing schemes, allows data to remain available for reads and writes as long as there is at least one operational replica (provided that failures are mostly sequential so that the system can adjust to failures as they arrive) but applicable to ROWA quorums. In addition, in our protocol, the system readjustment is done completely asynchronously with regard to read and write operations. This last property is equally significant for other quorums beside ROWA since in the existing (both ROWA and non-ROWA) replica control protocols that involve dynamic system readjustment, user operations are blocked while reconfiguration is in progress.

*Our research is supported in part by the National Science Foundation (Grants No. CCR-8907666, CDA-9123308, and CCR-9200832), Apple Computer, Boeing Computer Services, Digital Equipment Corporation, Hewlett-Packard Corporation, and the Washington Technology Center.

Our scheme is based on the notion of an epoch [11] which is a set of nodes in the system known at some point to be operational and connected. Originally, all nodes in the system form the current epoch. The system periodically runs a special operation, epoch checking, that polls all the nodes. 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 (possibly weighted) majority of nodes 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 all nodes from the new epoch. Then, due to the fact that any two majority sets have at least one node in common, 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, write operations can be allowed to succeed if they obtain permission from all replicas 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 epoch management scheme presented in this paper differs significantly from existing ones in several respects. First, we propose a way to define the quorums used to ensure uniqueness of the current epoch independently from the quorums used for replica control. This makes it possible to form new epochs even if ROWA quorums are used for replica control, while the existing protocols, where write quorums are employed for both replica control and epoch management, are not applicable to ROWA as explained above.

Second, forming a new epoch in our protocol is done completely asynchronously with user transactions and, therefore, cannot interfere with and delay the latter. In the existing dynamic protocols, the system reconfiguration is run as one or more special system transactions that compete for replica locks with user transactions.

Third, while the existing dynamic protocols (with the exception of the views and dynamic accessibility protocols discussed in the next section) manage partitions on a per data item basis, our epoch management is done for the whole system, with the overhead amortized.

Finally, for every node, our protocol keeps track of the number of the latest epoch that contained that node. Any time a new epoch is formed, this information is updated and recorded on all nodes of the new epoch. When a new node i is being included into the current epoch, this information is used to determine whether i has a replica of a data item x such that the latest epoch that contained any replica of x is more recent than i's epoch. If this is the case, the protocol infers that i's replica of x may not have seen some writes to x. Therefore, the protocol marks i's replica of x stale as part of the new epoch formation, thus prohibiting any reads from this replica until an asynchronous recovery procedure brings it up-to-date by copying information from any non-stale replica from the same epoch. Otherwise, the replica is made immediately available for accesses in the new epoch. This further improves the efficiency of our protocol since it often avoids initialization of replicas that are included in the newly-formed epoch, and when this initialization is necessary, it is performed by contacting a single current replica from the same epoch.

2 Related Work

Several protocols addressing the problem of availability of replicated data and applicable to the ROWA strategy have been proposed.

In the available copies protocol [2], a read is serviced by any operational replica and a write is performed on all *available*, i.e., currently operational replicas, with *validation* performed at the transaction commit time to make sure that the set of operational replicas of the data items involved in the transaction has not changed during the transaction execution. The available copies protocol provides very high level of data availability but guarantees data consistency only under node failures. In contrast, our protocol guarantees consistency under both node failures and network failures.

Consistency guarantees similar to those in our protocol are also provided by the views protocol ([4], generalized in [5] and [6]), the missing writes protocol [3], and the dynamic accessibility protocol [7].

A main limitation of the views protocol and the missing writes protocol is that a (possibly weighted) majority of replicas of the data item must be operational and connected for a logical write of this data item to be successful. In fact, in many cases, both these protocols require a majority of replicas to be available even for *reading* the data item. In addition, in the missing writes protocol, failures cause substantial extra logging of transaction information. Also, the missing writes protocol uses ROWA only when all replicas are operational and connected, and resorts to majority consensus otherwise. Hence, even a single replica failure causes a substantial read performance

penalty, which is incurred for the whole duration of the failure. Our protocol is free of these limitations. In particular, the data in our protocol may remain available for reads and writes even if only one replica of the data item is operational.

Another limitation of the views protocol is that in this protocol, all replicas that are to adopt a new view must be initialized, and, during initialization, must contact a majority of all replicas in the system to determine a current replica from which the data can be copied. Moreover, this replica initialization is run as a transaction that can interfere with the user transactions. In our protocol, initialization of replicas to be included into a new epoch is often avoided, and when this initialization is necessary, it is accomplished by contacting a single current replica from the same epoch.

In the dynamic accessibility protocol, as in the views protocol, every node s in the system maintains its *view*, which is a set of nodes that s assumes it can communicate with. For every data item x, the protocol keeps track of the latest view where x was *accessible*. Originally, all nodes have the view that includes all nodes in the system, and all data items are marked accessible in that view. When a new view is formed, a data item x is marked accessible there only if the new view includes a majority of the replicas of x from the latest existing view where it was accessible. If a data item is accessible in some view, a read of this data item can be serviced by any replica and a write must perform on all replicas of the data item from that view.

The dynamic accessibility protocol, like our protocol proposed here, can tolerate more than a majority of failed replicas. However, as in other existing dynamic protocols, the system reconfiguration in the dynamic accessibility protocol (a new view formation in this case) runs as an atomic transaction that may interfere and delay user transactions. Moreover, in the case of the dynamic accessibility protocol, this transaction involves communication with all sites to be included in the new view (not just sites that replicate a given data item), and the messages exchanged include the whole contents of databases stored on those sites. Thus, this reconfiguration transaction is very large, and even if it runs infrequently, the disruption of service caused by its execution may be prohibitively long. In our protocol, a new epoch formation is more efficient (the messages exchanged during epoch formation do not include contents of any data items) and, most importantly, runs asynchronously with user transactions.

Also, the policy for choosing partitions where the data will be available for accesses in these two protocols is different. The dynamic accessibility protocol makes a data item x available for both reads and writes in the partition that contains a "dynamic majority" of replicas of x and disallows reads and writes in other partitions. Our protocol allows the data item to be read in *any* partition that has a non-stale replica of x. Thus, our protocol provides better read availability. On the other hand, when replicas of x are split between partitions, we allow the writes to perform in the partition that has a "dynamic majority" of *nodes*, rather than replicas of x. So, in our case, we may disallow writes in a partition that has most of the replicas of x, and then, in the partition where writes into x are allowed, there may be few replicas of x. Should they fail, x would be not available for writes until one of those few replicas recovers. Thus, when the number of replicas of the data item is greater than two, the dynamic accessibility protocol provides better write availability. We believe the approach taken in our protocol is more in line with the general ROWA philosophy, which favors reads over writes.

3 The Model and Notation

We assume that the database is distributed among N nodes, with data items x, y, z, ... replicated on $n(x), n(y), n(z)$, ... number of nodes. Each node is assumed to have a unique name. No particular ordering is imposed on these names. Nodes and/or 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. Finally, in this paper, we assume that logical write operations are total, i.e., they always install a whole new value of the data item rather than updating a portion of the information contained in the data item.

The following is standard terminology and notation. We distinguish logical operations on data items from physical operations on the replicas of the data items. A logical read (write) operation on a data item x issued by a transaction T_i is denoted $r_i[x]$ ($w_i[x]$) and is translated by the protocol into a set of physical operations $r_i[x_p]$ ($w_i[x_p]$), where x_p is a replica of x that resides on node p. We call transaction T_i the *owner transaction* of logical and physical operations executed by T_i.

As a correctness criterion, we use one-copy serializability [2], which requires that concurrent execution of

transactions on replicated data be equivalent to some serial execution of these transactions on non-replicated data. We assume that some existing concurrency control protocol is used to ensure that the execution of transactions is serializable on the level of physical operations (see [2] for concurrency control protocols), 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, epoch checking, and replica recovery. Every node i in the system maintains the following state: *enumber*, the epoch number of node i's epoch; *ELIST*, a list of node names that are members of this epoch (the *epoch list*); *echeck-in-progress*, a flag indicating that node i is in the middle of switching to a new epoch and used by i to determine when to initiate recovery operations for i's stale replicas; for every replica x_i stored on node i, node i maintains a stale replica flag *stale(x)*, which, if set to "*YES*", indicates that this replica is out of date and must be recovered before physical reads on this replica are allowed. It is interesting to note that we do not need replica version numbers to identify most recent replicas: any replica not marked stale is considered up-to-date. Finally, a node keeps an array *latest-epoch*. For every node p in the system, *latest-epoch(p)* contains the number of the latest epoch that included p. This information is accurate only if the node is a member of the latest epoch in the system.

Originally, all nodes are members of epoch 0, and so have epoch number 0 and an epoch list that includes all nodes in the system. All replicas are current, so all stale replica flags are initialized to "*NO*", and *latest-epoch(p)*$= 0$ for all nodes. On all nodes, *echeck-in-progress* is initially set to "*NO*".

We assume that local updates of the node's state are atomic.

4.1 Epoch Checking

The algorithm for epoch checking and switching to a new epoch (if necessary) is shown in Figure 1. For simplicity, we assume that epoch checking operations are mutually exclusive, with any existing distributed mutual exclusion algorithm (for example, [13]) used to enforce this condition. In reality, a time-out mechanism specific to our protocol can be used to avoid a situation where a node that fails while holding the lock for epoch checking blocks any future epoch checking operations. This mechanism is actually a specialization of the three phase commit protocol [14] and is presented in [12].

The epoch checking operation can be initiated by any node. This node is called the coordinator below. The operation is done in three phases. During the first phase, the coordinator 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 (possibly weighted) majority of responses from the current epoch, where the current epoch is represented by the epoch list from a response with the maximum epoch number. If this is the case, the coordinator constructs a prospective 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 attempts to form a new epoch by initiating the second phase of the protocol. In this phase, the coordinator identifies new nodes that are to be added to the current epoch and for each such node i and each replica x_i, checks if i's epoch is less recent than the latest epoch that contained a replica of x, in which case x_i is marked stale. (The information on the latest epoch containing a replica of x is derived from array *latest-epoch* reported by any member of the current epoch). Then, in the third phase, the coordinator updates the prospective epoch list to include only those nodes that acknowledged marking their replicas stale, generates a new epoch number by incrementing the current epoch number by 1, and distributes the new epoch number and epoch list to all members of the prospective epoch. To ensure all-or-nothing execution, the new epoch number and epoch list are written on all members of the new epoch atomically using, for example, the two-phase commit protocol [9] (but see [12] for further discussion). However, no atomicity of switching to a new epoch with regards to read and write operations is implied.

4.2 Write Operation

We assume that when a node initiates a transaction T_i, it associates with it the node's current epoch number and epoch list, $enumber(T_i)$ and $elist(T_i)$. This epoch number and epoch list never change during the lifetime of the transaction (even if the node's epoch changes) and are available to all reads and writes executed by the transaction. Similarly to [6], we say that transaction T_i executes in epoch $enumber(T_i)$. We also assume that every replica has local read and write locks with the multiple-readers-single-writer conflict rule. (Locking a replica for write precludes any other operations on the replica, while locking for read pre-

```
EpochCheck
/* This algorithm is run by the coordinator; the actions of participants are given within the comments */
    obtain the lock for epoch checking;   /* Ensures that epoch checking operations are mutually exclusive */
    multicast(all-nodes-in-the-system, "epoch-check-request");
    /* A node that receives this message responds with a tuple of the form (node, enumber, ELIST, latest-epoch). */
    receive RESPONSES[1:k];
    /* find a response with the maximum epoch number. */
    m := an index j such that enumber_j = max_{i=1,...,k}{enumber_i};
    if |{node_1, ..., node_k} ∩ ELIST_m| ≥ ⌈|ELIST_m|/2⌉ and {node_1, ..., node_k} ≠ ELIST_m then
        NEW-EPOCH := {node_1, ..., node_k};
        NEW-NODES := NEW-EPOCH\ELIST_m;
        for each node i ∈ NEW-NODES do in parallel
            STALE-REPLICAS_i := ∅;   /* a working variable */
            for each replica x_i residing on node i do
                if there exists a node p such that p has a replica of x and enumber_i < latest-epoch_m(p) then
                    STALE-REPLICAS_i := STALE-REPLICAS_i ∪ {x_i};
                endif;
            endfor;
            if STALE-REPLICAS_i ≠ ∅ then
                send-message(i, ("mark-stale", STALE-REPLICAS_i));
                /* On receiving the above message node i atomically sets stale(x) to "YES" for all data items x
                from STALE-REPLICAS_i, sets echeck-in-progress to "YES" to prevent recovery operations to perform
                until i switches to the new epoch, and sends an acknowledgement back to the coordinator. */
                if i fails to acknowledge that it marked the specified replicas stale then
                    NEW-EPOCH := NEW-EPOCH\{i};
                endif;
            endif;
        endfor;
        new-epoch-number := enumber_m + 1;
        do-atomically   /* ensures all-or-nothing execution; no atomicity with regards to reads and writes is implied. */
            multicast(NEW-EPOCH, ("switch-epoch", NEW-EPOCH, new-epoch-number));
            /* On receiving the above message, a participant updates its enumber and elist to be equal to
            the values received; sets latest-epoch(p) to be equal to new-epoch-number for each node p
            represented in NEW-EPOCH; sets echeck-in-progress to "NO" to indicate that it completed switching to
            the new epoch; and responds with an acknowledgement */
        end-do-atomically;
        release the lock on epoch checking;
    endif;
end;
```

Figure 1: The algorithm for checking an epoch and switching to a new epoch.

cludes writes and allows reads.) In the algorithm below, we do not address the deadlock problem. For ways to handle deadlocks, see for example [2].

The algorithm for the write operation is shown in Figure 2. To write a data item x, the coordinator (the node that initiated the write) requests the write lock for x from all participants (the nodes that have a replica of x and are members of the epoch associated with the owner transaction). With the request for the lock, the coordinator includes the epoch number of the owner transaction. On receiving a lock request, a participant first compares its epoch number with the epoch number received with the request. If the participant's epoch number does not exceed the epoch number received, the participant locks its replica of x and sends a positive acknowledgement back to the coordinator. (The epoch number comparison and locking the replica is done as one atomic action.) If the participant's epoch number is greater than the epoch number received, the participant responds with a negative acknowledgement, which causes the coordinator to abort the transaction.

If all participants responded with positive acknowledgements, the coordinator attempts to perform the write on all replicas. When a participating node receives a message requesting it to perform the write, it atomically writes the new value of the data item and, if the node's epoch number still does not exceed the epoch number of the owner transaction, marks the replica non-stale and sets its epoch list and number to be equal to the epoch list and number of the owner transaction. If the epoch number of the node has become greater than the epoch number of the owner transaction (due to asynchronous formation of a new epoch), the new data is still written but the node's state remains unchanged. (The data should be written in this case due to the following reasoning. Since the replica was locked for write operation while the node was in its previous epoch, no operations could read or update this replica after it switched to the new epoch. If this replica was found current when the node was included into the new epoch, the new data should be written to prevent reading stale data by a transaction executing in the new epoch. If the replica was marked stale at the time the node was included into the new epoch, it does not matter whether the new data is written or not since the replica cannot be read until it obtains a value considered current in the new epoch.) After performing the atomic action above, the participant unlocks its replica.

It should be noted that if *two-phase locking* [2] is used for concurrency control, the same locks can be shared by both replica control and concurrency control protocols. Then, the policy for releasing the locks would be the combination of policies used by each protocol, and a replica would not release the write lock immediately after performing the write. (The same goes for reads considered in the next subsection.) However, as we do not make any assumptions about the nature of concurrency control mechanisms used in the system (e.g., these mechanisms may use separate locks or no locks at all), we describe the policy sufficient for the replica control protocol to be correct provided any correct concurrency control protocol is used.

Also note that in the normal case of absence of changes in the system topology, *OLD-EPOCH-NODES* in the write algorithm is empty and all messages exchanged during the write operation have constant lengths independent of the number of nodes in the system. The desire to keep the length of all messages constant and to avoid relying on any specific concurrency control mechanism leads to the two-phase write algorithm described. If the length of messages is of no concern, and two-phase locking is used for concurrency control, with locks shared by the concurrency and replica control protocols (so that once a transaction locks a replica the replica remains locked until the transaction commit time) a one-phase write algorithm is possible. In this algorithm, the write coordinator would send the new value of x along with $enumber(T)$ and $elist(T)$ to *PARTICIPANTS*, and each participant would atomically check if its epoch number does not exceed $enumber(T)$ and, if so, acquire the write lock for x, perform the write, and adopt the epoch list and number received.

4.3 Read Operation

The algorithm for the read operation is simple. A node that initiates the read of data item x (the coordinator) chooses the closest node that has a replica of x (the participant) among the nodes represented in the epoch of the owner transaction. The initiator then sends a request for the data to the chosen replica together with the epoch number of the owner transaction. If the participant's replica is marked stale, or if its epoch number is different from the epoch number of the owner transaction, the participant rejects the request which causes the initiator to repeat the operation with another participant (or abort if none is available).

Otherwise, the participant obtains the local read lock for the replica, checks its epoch number again and, if it is still equal to the one from the read request, sends the data to the coordinator and releases the read

```
Write(x /* data item */, new-value,
      elist(T)   /* epoch list of the owner transaction */,
      enumber(T)   /* epoch number of the owner transaction */)
/* This algorithm is run by the coordinator; the actions of participants are
given within the comments. */
   PARTICIPANTS := {node p such that p ∈ elist(T) and p has a replica of x};
   multicast(PARTICIPANTS, ("write-lock-request", x, enumber(T)));
   /* On receiving write-lock-request, a participant atomically compares its epoch number
   with enumber(T) received and, if the participant's epoch number does not exceed enumber(T),
   locks its replica of x and sends a positive acknowledgement to the coordinator.
   Otherwise, the participant sends back a negative acknowledgement. */
   receive RESPONSES[1:k];
   if all nodes from PARTICIPANTS replied with positive response then
      CURRENT-EPOCH-NODES := {participants that reported the epoch number equal to enumber(T)};
      OLD-EPOCH-NODES := {participants that reported epoch numbers smaller than enumber(T)};
      multicast(OLD-EPOCH-NODES, ("do-update-and-adopt-new-epoch", x, new-value, enumber(T), elist(T)));
      /* When a participant p receives "do-update-and-adopt-new-epoch" message, it atomically writes
      the new-value into its replica of x and, if its epoch number still does not exceed
      enumber(T), sets stale(x) = "NO" and updates its epoch list and number to
      be equal to elist(T) and enumber(T). Then it unlocks its replica. */
      multicast(CURRENT-EPOCH-NODES, ("do-update", x, new-value, enumber(T)));
      /* On receiving"do-update" message, a participant p atomically writes
      the new-value into its replica of x and, if its epoch number still does not exceed
      enumber(T), sets stale(x) = "NO". Then it unlocks its replica. */
   else
      if some node from PARTICIPANT replied with negative acknowledgement
         multicast(PARTICIPANTS, ("unlock-replica", x));
         /* On receiving this message a participant unlocks its replica of x */
         abort the owner transaction;
      else   /* no negative acknowledgement received but some participants failed to respond. */
         multicast(PARTICIPANTS, ("unlock-replica", x));
         repeat the whole operation later or abort the owner transaction;
      endif
   endif
end;
```

Figure 2: The algorithm for the write operation.

lock.

A few points in the read protocol are noteworthy. First, our protocol never needs more than a single current replica in the transaction's epoch to perform a read. This contrasts favorably with the views protocol [6] and the dynamic accessibility protocol [7] where the transaction's view must include a majority (a "dynamic majority" in the latter protocol) of replicas of the data item for the read to be successful.

Second, unlike the missing writes protocol [3], even in the presence of failures, our protocol does not need replica version numbers to determine stale replicas. It can always identify whether a replica is stale by using only local state of the node on which the replica resides. Therefore, a read operation never requires more than one round of message exchange between the node issuing the read and the node having a replica. Also, our protocol never contacts more than one current replica to perform a read. In contrast, the missing writes protocol uses a majority consensus scheme when failures are detected in the system, thus incurring a significant read performance penalty. Of course, if a replica contacted during the read turns out to be stale, our protocol must repeat the read with another replica. However, since epoch reconfiguration and the subsequent replica recovery occurs only in the aftermath of failures or repairs in the system, in the common case no reconfiguration or recovery is going on in the system. Then, most of the time all replicas in the system are marked non-stale, unless some stale replica could not contact a current replica from the same epoch to copy the data during recovery. In the latter case, the read operation will most probably also fail since it also needs a current replica. Hence, a successful read operation executed by a transaction running in the current epoch in most cases is satisfied by contacting a single node.

Finally, when the epoch of the owner transaction is not the most current one (the transaction runs "in the past" [3]), some nodes included into the owner transaction's epoch list may have moved to the later epochs. Then, the read initiator may contact a node that will reject the read request because it is now in a different epoch. Hence, in our protocol, even successful reads, if they "run in the past", may have to contact multiple nodes. We could have followed the approach taken by the views and dynamic accessibility protocols and abort the transaction in this case. The way we have chosen gives the transaction an extra chance to complete successfully.

4.4 Recovery Operation

A node p runs a recovery operation for any of its replicas that is marked stale. The recovery can be initiated only if the node is not in the middle of epoch switching (i.e., if *echeck-in-progress* is "*NO*"). The reason for this restriction is that there is some time-lapse between the moment an epoch checking operation marks a replica x_p stale and the moment when node p switches to the new epoch. If recovery were allowed to perform during this period, x_p could "recover" from a member of its old epoch, mark itself non-stale, and *then* switch to the new epoch. In this case, x_p would be obsolete in the new epoch while not marked stale, thus allowing read operations to perform on x_p and return obsolete data.

The idea of the recovery algorithm is to make sure that both source and target replicas are members of the same epoch, and that the target replica remains stale by the time it actually gets updated. (Otherwise, if the message with the new version from the source replica gets delayed so that, by the time it arrives, the target replica has been written by a later write then the version from the delayed message, now obsolete, would override the current information in the target replica.) Also, to make sure a recovery operation does not propagate a transient value written by a transaction in progress that aborts later, a recovery operation is allowed to read a replica only if this replica is not written by an uncommitted transaction.

To recover a replica of data item x, node p (the coordinator) finds the nearest node that has a replica of x (the participant) among the nodes included into p's epoch list and sends a recovery request together with p's epoch number to the chosen node. If the participant replica is marked stale, or if its epoch number is different from the epoch number received with the request, or if its replica has been written by an uncommitted transaction, the participant rejects the recovery request by sending a negative acknowledgement. Otherwise, the participant obtains the local read lock for the replica, checks again if its epoch number is still equal to the epoch number received from initiator, sends the data together with its epoch number back to the coordinator, and releases the read lock.

Upon receiving the data, the coordinator obtains the local write lock for its replica, checks if its epoch number has not changed since the beginning of the recovery operation by comparing it with the epoch number received from the participant and if its replica is still marked stale. If that is the case, the coordinator updates its replica, marks it non-stale, and releases the write lock.

Note that in the recovery operation, both source and target replicas are locked only during local read and write operations and not while messages are being exchanged.

5 Examples

In this section, we discuss informally the correctness of the protocol and illustrate some of the more subtle points with examples. The formal proof of correctness is given in [12].

First, consider the case where logical read and write operations and epoch changing operations do not execute at the same time. Since any new epoch must include a majority of nodes from an existing epoch, the uniqueness of epochs is guaranteed, i.e., all nodes that have the same epoch number have also the same epoch list (see Lemma 1 from [11] for a similar proof). Then, due to the intersection property of ROWA quorums, all transactions executing in the same epoch are one-copy serializable. In addition, if transaction T_1 is successfully executed in an epoch with epoch number e_1 and transaction T_2 is successfully executed in an epoch with epoch number e_2, and $e_1 < e_2$, it can be shown similarly to [6] that T_1 can be always serialized before T_2, regardless of actual execution time of the physical operations contained in T_1 and T_2. The proof is based on the fact that none of the physical writes of a transaction T_i is allowed to perform on nodes with epoch number greater than $enumber(T_i)$, and physical reads of transaction T_i must execute only on nodes with epoch numbers equal to $enumber(T_i)$. Therefore, even though transactions are allowed to succeed while "running in the past" (i.e., in non-current epochs) one-copy serializability is enforced.

The most difficult case to consider is when a user operation and switching to a new epoch are both happening at the same time and involve the same nodes. The correctness in this case is ensured in our protocol by maintaining the invariant that no matter how epoch changing, recovery, and physical read and write operations are interleaved, all replicas residing on nodes with the same epoch number are either identical or marked stale. We now give a few examples of how our protocol handles such situations.

Consider the situation where an epoch e_i includes two replicas of a data item x, x_1 and x_2, a logical write $w[x]$ executing in epoch e_i (we will use this shortcut instead of saying "a write executing on behalf of transaction with associated epoch number e_i") has locked both replicas, but before it actually updates them, a change in the system occurs and formation of a new epoch $e_j > e_i$ is attempted. Assume that the node that stores x_2 is to be included into the new epoch while x_1 is not. Furthermore, during formation of epoch e_j, another replica x_3 was found to be in epoch e_k, and $e_i < e_k < e_j$. Then, the coordinator of e_j formation marks x_2 stale (updating the epoch number of x_2's node to e_j at the same time) and thus initiates its recovery. However, x_2 cannot recover since it is locked for write. When write $w[x]$ tries to perform on x_2, it finds the epoch number of x_2 to have become greater than e_i and leaves x_2 marked stale. So, after $w[x]$ completes, x_1 remains in epoch e_i, is non-stale, and reflects the result of $w[x]$ (hence, x_1 continues to be available for transactions running "in the past" in epoch e_i), while x_2 is in epoch e_j, marked stale, and has not recovered. Thus, the overall effect of all these events is the same as if $w[x]$ had completed before formation of epoch e_j began.

Next, consider a situation where an epoch e_1 includes a stale replica x_1 and two current replicas, x_2 and x_3, of a data item x, and x_1 wants to recover the current data from replica x_2. In the meantime, reconfiguration is initiated to form a new epoch with epoch number $e_2 = e_1 + 1$. Replica x_2 is to be included into e_2 while replicas x_1 and x_3 are not. A transaction T_i is initiated in epoch e_2 and executes $w_i[x]$ while x_2 still has not changed its state to indicate the transition from epoch e_1 to e_2. Now, if the recovery operation initiated by x_1 obtained x_2's read lock before x_2 switches to e_2, $w_i[x]$ could not have performed on x_2 and x_1 gets the value of x_2 that it had before reconfiguration started. Thus, all operations can be serialized as recovery executed in epoch e_1, reconfiguration, write executed in e_2. If the recovery request reached x_2 after it switched to e_2, there is no way to tell whether $w_i[x]$ already performed on x_2 or not, since we do not keep version numbers. It would be wrong for x_1 to copy x_2's data *after* $w_i[x_2]$ performs because then replicas x_1 and x_3 would diverge while both being in the same epoch and marked non-stale. Therefore, we require that both source and target replicas be in the same epoch for a recovery operation to be successful. So, in our example, once x_2 switches to epoch e_2, the recovery initiated by x_1 will not succeed no matter whether any writes have performed on x_2 in epoch e_2 or not. After x_1's attempt to recover from x_2 fails, it will probably try to recover from x_3. Barring additional reconfigurations, this recovery may well succeed while "running in the past" in epoch e_1. If $w_i[x]$ locks x_2 before the recovery, then once it succeeds, the epoch number of x_2 becomes e_2, and x_2 will reject recovery request from x_1. In this case, the equivalent serial execution is reconfiguration, write executed in e_2, and x_1 must recover

from a replica other than x_2.

As the last example, consider a situation where epoch e_1 contains current replicas x_1 and x_2, and a write $w_i[x]$ is initiated by a transaction T_i in e_1. At about the same time, a new epoch $e_2 = e_1 + 1$ is being formed, x_2 is to be included into e_2, and a read $r_j[x]$ is initiated by a transaction T_j in e_2. Suppose that $r_j[x]$ is translated into a physical read on x_2. Suppose $w_i[x]$ locked x_1 and x_2, and then x_2 switched to e_2 by updating its epoch list and number. Since x_2 was current in e_1, and since e_2 is the next epoch in the system, x_2 will be found current when included in e_2. When a physical write executed on behalf of $w_i[x]$ reaches the node where x_2 resides, it will find x_2's epoch number equal $e_2 > enumber(T_i) = e_1$. However, it would be incorrect not to update x_2 since x_1 would be updated (it is still in epoch e_1) and, after x_2 is unlocked, $r_j[x]$ would read stale data. The way the protocol works, x_2 will be updated while its node's state will remain unchanged, so when $r_j[x]$ gets to read x_2, it will see the effect of $w_i[x]$. Thus, the operations are serialized as $w_i[x]$ executed in e_1, reconfiguration, $r_j[x]$ executed in e_2.

6 Conclusion

We have presented a new read-one-write-all (ROWA) replica control protocol that adjusts to failures and repairs as they arrive in the system so that the data remains available. This is achieved by making the epoch mechanism, previously applicable to non-ROWA schemes only, usable within the ROWA discipline. In our protocol, the write can succeed if it can perform on all replicas of the data item included into the transaction's *epoch*. The read can succeed if it can perform on a single current replica of the data item from the transaction's epoch. Therefore, the data may be available for both read and write accesses even if only one operational replica of the data item remains in the system provided that failures occurred mostly sequentially so that the protocol could switch to new epochs as failures arrive. At the same time, our protocol retains both major benefits of the original static ROWA. First, if media failures divide non-stale replicas of a data item among different partitions, the data item remains available for reads in any partition that contains at least one such replica. Second, even in the presence of failures, our protocol never needs to contact more than one current replica to perform a logical read, and then it needs only one round of message exchange with this replica. It can also always determine whether a replica is stale using only local state of the node on which the replica resides.

Another new property of our protocol as compared to the existing dynamic protocols is that system readjustment is done asynchronously with user transactions and, therefore, cannot interfere with or delay the latter.

Also, when switching to a new epoch, our protocol is very careful as to what replicas to make run the initialization procedure. As a simple example, if all replicas of a data item failed, and if the replica that failed last is the first to become operational again, its information is guaranteed to be current without any additional work. Our protocol identifies these sorts of circumstances and does not require the replicas to run initialization in such situations. Running excessive initializations is undesirable since while initialization is in progress, the affected replicas are unavailable (at least for reads).

Finally, in the normal case of the absence of changes in the system topology, all messages exchanged in our protocol during the execution of logical reads and writes have constant lengths independent of the number of nodes in the system. Therefore, our protocol should scale well.

The read-one-write-all scheme is commonly employed to manage replicated data in distributed systems. We believe that the protocol proposed here can find practical use in these systems.

Acknowledgements

We would like to thank Bruce Lindsay, Amr El Abbadi, and the referees for their helpful comments.

References

[1] 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.

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

[3] D. Eager and K. Sevcik. Achieving robustness in distributed database systems. *ACM Trans. Database Sys.* 8(3), pp. 354-381, 1983.

[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] A. El Abbadi and S. N. Dani. A dynamic accessibility protocol for replicated databases. *Data and Knowledge Engineering*, 6(1991), pp. 319-332, 1991.

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

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

[10] 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.

[11] 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.

[12] M. Rabinovich and E. Lazowska. An efficient and highly available read-one write-all protocol for replicated data management. Technical Report 92-07-04, University of Washington Department of Computer Science and Engineering, July 1992.

[13] M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, 1986.

[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.