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, 12, 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

*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.

epoch [12]) 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 scheme is based on the notion of an epoch [12] 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, [14]) 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 [15]. The epoch checking operation that incorporates time-outs is presented in the Appendix.

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 Appendix for further discussion). However, no atomicity of switching to a new epoch with regards to read and write operations is implied.

```
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.

```
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.

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 precludes 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 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 timelapse 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 the next section.

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 [12] 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 Proof of Correctness

6.1 Background

We first give some preliminaries of the serializability theory from [2].

A *transaction* T_i is a partial order $(S_i, <_i)$, where (1) S_i is a set of logical read and write operations executed by T_i plus an abort a_i or commit c_i operation; (2) $<_i$ reflects the execution order of logical operations, i.e., if $op_1 <_i op_2$ then op_1 should be executed before op_2; (3) $a_i \in S_i$ iff $c_i \notin S_i$ (a transaction may either commit or abort but not both); (4) if $c_i \in S_i$, for any other operation $op \in S_i, op <_i c_i$ (if a transaction commits, commit is the last operation of the transaction); (5) if $a_i \in S_i$, for any other operation $op \in S_i, op <_i a_i$ (if a transaction aborts, abort is the last operation of the transaction); (6) any two operations on the same data item are ordered. We denote logical read and write operations on data item x issued by T_i as $w_i[x]$ and $r_i[x]$. We assume that a transaction reads and writes a data item x at most once.

In a replicated database, logical reads and writes on data items are translated into *physical* operations on replicas of these data items. This translation is formalized by a function h that maps each $r_i[x]$ into $r_i[x_p]$, where x_p is a replica of x; and each $w_i[x]$ into $w_i[x_{p_1}], \ldots, w_i[x_{p_m}]$ for some replicas $x_{p_1}, \ldots, x_{p_m}$ of x $(m > 0)$. It is also convenient to include commit and abort operations into the domain of h by defining $h(c_i) = c_i$ and $h(a_i) = a_i$.

A *history* H over transactions $\{T_0, \ldots, T_n\}$ is a partial order $(S_H, <_H)$, where (1) $S_H = h(\cup_{i=0}^{n} S_i)$ for some translation function h (a history records physical operations on replicas of data items); (2) $<_H$ reflects the execution order of physical operations, i.e., if $op_1 <_H op_2$ then op_1 was executed before op_2; (3) for each T_i and all operations op_1 and op_2 in S_i, if $op_1 <_i op_2$ then every operation in $h(op_1)$ is related by $<_H$ to every operation in $h(op_2)$ (a history preserves all orderings stipulated by transactions); (4) for every r_i in S_H, S_H contains some $w_j <_H r_i$ (a transaction may read only those replicas that have been previously initialized); (5) all pairs of conflicting operations in S_H are related by $<_H$, where two operations conflict if they operate on the same replica and at least on of them is a write; (6) if $w_i[x] <_i r_i[x]$ and $h(r_i) = r_i[x_p]$, then $w_i[x_p] \in h(w_i[x_i])$ (if a transaction T_i writes a data item x and then later reads x, then T_i must write into the same replica x_a that it subsequently reads)[1].

Transaction T_j *reads-x-from* T_i in H if there is a replica x_p of x such that $w_i[x_p] <_H r_j[x_p]$ and there is no $w_k[x_p]$ that falls between these operations, i.e., $w_i[x_p] <_H w_k[x_p] <_H r_j[x_p]$. Given history H, $w_i[x_p]$ is a *final write for* x_p in H if $a_i \notin S_H$ and for all $w_j[x_p]$ in H $(j \neq i)$, either $a_j \in S_H$ or $w_j[x_p] < w_i[x_p]$.

We will consider only *recoverable* histories, i.e., histories where aborted transactions cannot have any effect on committed transactions. To guarantee that, it is sufficient to require that a transaction can commit only after the commitment of all transactions (other than itself) from which it read. Formally, history H is *recoverable* if whenever a transaction T_i reads from T_j $(i \neq j)$ in H and $c_i \in S_H$, $c_j <_H c_i$. All practically useful concurrency control protocols allow only recoverable histories. Since we are interested in serializability of committed transactions only, and because aborted transactions in recoverable histories do not have any effect on committed transactions, we

[1]Transaction management mechanisms outside replica control are assumed responsible for enforcing the above properties of transactions and histories. For instance, commit protocols are used to make sure a transaction does not contain both commit and abort operations. However, often replica control protocols enforce some of these properties as side effects, so there is some duplication of work among different components of transaction management. As an example, in our protocol, the third and fifth conditions in the definition of histories are enforced by locking replicas prior to executing physical reads and writes. In real systems, this duplication is avoided by making various components cooperate closely.

can limit ourselves to considering histories that contain only committed transactions. *From now on, we assume that all transactions in H are committed.*

A history in which translation function h translates any logical operation on a data item x into a physical operation on a single replica x_p ($h(w_i[x]) = w_i[x_p]$ and $h(r_i[x]) = r_i[x_p]$ for any i) is called non-replicated history.

A *serial* history is a totally ordered history such that for every pair of transactions T_i and T_j, either all physical operations executed by T_i precede all physical operations executed by T_j or vice versa. A history H is called *conflict-serializable* if there is a serial history H_s over the same set of transactions and the same translation function h such that if (physical) operations op_1 and op_2 conflict and $op_1 <_H op_2$ then $op_1 <_{H_s} op_2$.

The serialization graph SG(H) for a history H is a directed graph where the nodes correspond to committed transactions in H and there is an edge $T_i \rightarrow T_j$ if there are conflicting operations op_1 in T_i and op_2 in T_j such that $op_1 <_H op_2$. A history is conflict serializable if and only if its serialization graph is acyclic (see Theorem 2.1 in [2]).

Two histories H_1 and H_2 are *equivalent* if (1) they are defined over the same set of transactions (note that translation functions may be different); (2) T_i reads-x-from T_j in H_1 iff T_i reads-x-from T_j in H_2; and (3) for each data item x, if $w_i[x_p]$ is a final write in H_1 on a replica of x then there is a final write $w_i[x_q]$ in H_2 on a replica of x (note that the final writes are executed by the same transaction T_i in both histories). A history is *one-copy serializable* if it is equivalent to a serial non-replicated history.

In a directed graph, node n *precedes* node m, denoted $n \ll m$, if there is a path from n to m. Given a history H, a *replicated data serialization graph (RDSG)* for H is SG(H) with enough edges added so that, for every data item x, (1) if T_i and T_j write x, then either $T_i \ll T_j$ or $T_j \ll T_i$; and (2) if T_j reads-x-from T_i, T_k writes some replica of x ($k \neq i, k \neq j$), and $T_i \ll T_k$ then $T_j \ll T_k$. Note that an RDSG for H is not unique. The main theorem is that if H has an acyclic RDSG, then H is one-copy serializable (see Theorem 8.4 in [2]).

6.2 Extension to the Serializability Theory

Similarly to [6], we need to extend the standard serializability theory to include recovery operations. While our recovery operations are significantly different from update transactions in [6] (the main difference being that, unlike update transactions, our recovery operations are not controlled by the concurrency control protocol that ensures serializability of transactions on the level of physical operations; thus our recovery operations are *not* transactions), the extension described in this subsection is the same, with update transactions replaced by recovery operations (or recoveries, for short).

Below, when we need to refer to transactions and recoveries together, we will call them *actions*. Histories and serialization graphs are now defined over sets of actions, including recoveries.

Action A_j *directly reads-x-from* action A_i in history H if there is a replica x_p such that $w_i[x_p] <_H r_j[x_p]$ and there is no $w_k[x_p]$ that falls between these operations, i.e., $w_i[x_p] <_H w_k[x_p] <_H r_j[x_p]$.

Let $R_{u_1}, \ldots, R_{u_n}$ be a sequence of recoveries and A_i and A_j be two actions. A_j *indirectly reads-x-from* A_i if R_{u_1} *directly reads-x-from* A_i, R_{u_2} *directly reads-x-from* R_{u_1}, ..., and A_j *directly reads-x-from* R_{u_n}. Action A_j *reads-x-from* A_i if A_j *directly* or *indirectly reads-x-from* A_i.

Note that unsuccessful recoveries do not change state of any node in the system. Therefore, unsuccessful recoveries do not have any effect on committed transactions or successful recoveries. Also, as we discussed in the previous subsection, since we consider only recoverable histories, aborted transactions cannot affect in any way committed transactions. Moreover, because a recovery reads the source replica only if it has not been written by any uncommitted transaction, aborted transactions cannot have any effect on successful recoveries either. Hence, when we study serializability of committed transactions, we can ignore both aborted transactions and unsuccessful recoveries and consider only histories where all transactions are committed and all recoveries are successful. From now on, *we assume that history H contains only successful actions, that is, committed transactions and successful recoveries.*

A serialization graph SG(H) is a directed graph whose nodes are (successful) actions from H, and there is an edge $A_i \rightarrow A_j$ if there are conflicting operations op_1 in A_i and op_2 in A_j such that $op_1 <_H op_2$. Note that if action A_j *reads-x-from* A_i (directly or indirectly) then SG(H) has a path from A_i to A_j. A replicated data serialization graph RDSG(H) for history H is SG(H) with enough edges added so that, for every data item x, (1) if transactions T_i and T_j write x, then either $T_i \ll T_j$ or $T_j \ll T_i$ (note that recoveries are not required to be ordered); and (2) for any three transactions T_i, T_j, and T_k, if T_j *reads-x-from* T_i, T_k writes x, and $T_i \ll T_k$, then $T_j \ll T_k$.

The proof of Theorem 8.4 from [2] still holds with the extension we described. So, we have the following theorem.

Theorem 1 *If RDSG(H) is acyclic, H is one-copy serializable.*

6.3 The Proof

In this subsection, H denotes an arbitrary history produced by our protocol, in which all actions are successful. As before, *enumber*(A_i) and *elist*(A_i) denote the epoch number and epoch list of an action A_i.

Lemma 1 (Uniqueness of epochs) *(1) At all times, all nodes that have the same epoch number e have also the same epoch list $elist_e$, which includes all nodes with epoch number e in the system (and possibly other nodes as well). (2) If p is an epoch checking operation and e_m is the maximum epoch number present in the system, then, by the time p obtained the lock for epoch checking, all members of epoch list $elist_{e_m}$ have epoch number e_m, and for any epoch number $e < e_m$, a majority of members of epoch list $elist_e$ have epoch numbers greater than e, and none have epoch numbers smaller than e.*

Proof. The proof is by induction on the maximum epoch number present in the system. (i) If the maximum epoch number in the system is zero, the lemma is obviously correct: initially all nodes have the same epoch list that includes all nodes in the system. (ii) Assume that the lemma is correct for the maximum epoch number equal to some $e_m \geq 0$. We must prove that in all situations where the maximum epoch number becomes $e'_m > e_m$, the lemma remains correct during all time while the maximum epoch number remains e'_m. Since reads, writes and recoveries do not introduce new epoch numbers in the system, the maximum epoch number present in the system can increase only as a result of an epoch checking operation. Consider the first epoch checking operation p that forms a new epoch after the maximum epoch number present in the system has been e_m. First, consider the case where write operations do not modify epoch lists and numbers, so that the only operations that can do that are epoch checking operations. The p's coordinator attempts to form a new epoch only if, during the first phase, it received responses from a majority of nodes from an epoch list reported by a node with the greatest epoch number among those that responded. We now show that it can be only the epoch list $elist_{e_m}$ kept by nodes with epoch number e_m. By the inductive assumption, when p obtained the lock for epoch checking, any majority set of members of any other epoch list $elist_l$, $l < e$, had at least one node with epoch number greater than l, and no members with epoch numbers less than l. Therefore, if l were the greatest epoch number reported to p' coordinator during the first phase, the coordinator would not be able to obtain responses from a majority of nodes from $elist_l$, which would contradict the assumption that p forms a new epoch. Thus, the maximum epoch number among responses p obtained can only be e_m, and the new epoch number p generates will be $e_m + 1$ and therefore unique. Let $elist_{e_m+1}$ be epoch list NEW-EPOCH formed by p' coordinator during the first phase of p. Since p records the same epoch list $elist_{e_m+1}$ on all nodes that receive the new epoch number, and since this epoch list includes all nodes receiving the new epoch number, the first claim of the lemma remains correct during and after the execution of p, at least until some later epoch checking operation changes epoch lists and numbers on some nodes. Also, for p to be successful, it must record the new epoch number on all nodes from $elist_{e_m+1}$, which includes a majority of members of $elist_{e_m}$. Therefore, once p completes, list $elist_{e_m}$ will no longer have a majority of its members keeping epoch numbers that do not exceed e_m. Moreover, since an epoch checking operation never writes earlier epoch numbers on any nodes, the sets of nodes with epoch numbers less than e_m can only shrink. Hence, after p completes, it is still true that any node with an epoch number $l < e_m$ has an epoch list in which a majority of members store epoch numbers greater than l. So, by the time the next epoch checking operation after p can obtain the lock for epoch checking, all members of epoch list $elist_{e_m+1}$ have epoch number $e_m + 1$, which is the maximum epoch number in the system, and for any epoch number $e < e_m + 1$, a majority of members of epoch list $elist_e$ have epoch numbers greater than e, and none have epoch numbers smaller than e. Let q be the first successful epoch checking operation (i.e., the operation that forms a new epoch) after p. Since unsuccessful epoch checking operations do not change epoch lists and numbers on any nodes in the system, the first claim of the lemma is satisfied until q updates epoch list or number on the first node, and the second claim of the lemma is satisfied for all epoch checking operations after p and up to (and including) q. But similarly to p, on all nodes where q modifies an epoch list or number, it records the epoch number equal to the maximum epoch number in the system at the time q obtained the lock for epoch checking plus one. Therefore, the lemma statement is correct any time while the maximum epoch number in the system is $e_m + 1$. Hence, by induction, under the assumption of the absence of modification of epoch state information by writes, the lemma is correct. So, all we have to show is that modification of epoch lists and numbers by writes cannot violate the lemma. Indeed, a write w_i issued by a transaction T_i can change epoch lists and numbers only on nodes that are members of $elist(T_i)$. In addition, if w_i changes epoch list or number on some nodes, it makes the new nodes' epoch list and number equal to $elist(T_i)$ and $enumber(T_i)$. Therefore, if before w_i changes a node's epoch list and number the first claim of the lemma was correct, it will also be correct after the change as well. Since initially (when all epoch numbers are 0) the lemma is correct, we conclude that no modification of epoch lists or numbers by write operations can violate the first claim of the lemma. Finally, since modification of the epoch number of a node by a write can only increase the node's epoch number (indeed, a write changes the epoch number on a node only if the node's current epoch number is smaller than the prospective epoch number), this modification cannot violate the second claim of the lemma either. □

Lemma 2 (Monotonicity of epoch numbers) *Epoch numbers never decrease on any node.*

Proof. Epoch number on a node may be changed by an epoch checking operation or a write operation. In the case of a write operation, a node adopts a new epoch number only if its old epoch number does not exceed the new one, with the check and the update done atomically. In the case of an epoch checking, since epoch checking operations are mutually exclusive, only one such operation at a time can change nodes' epoch numbers. Let p be an epoch checking operation that successfully forms a new epoch (unsuccessful epoch checking operations so not change epoch numbers on any nodes), and let e_m be the maximum epoch number present in the system by the time p's coordinator obtains the lock for epoch checking. Since no operations that may run in parallel with p can increase e_m and because, by Lemma 1, for any $e < e_m$, a majority of members of $elist_e$ have epoch numbers greater than e, it is easy to show similarly to the proof of Lemma 1 that the new epoch number generated by p will be $e_m + 1$, which is greater than the existing epoch number of any node in the system. □

Similarly to transactions, it is convenient to define the epoch list and epoch number of a recovery operation to be the epoch list and epoch number of the recovery initiator by the time the recovery is initiated. More precisely, it is the epoch number that the initiator sends to the source replica with the recovery request, and the corresponding epoch list.

Lemma 3 *If $SG(H)$ has an edge from action A_i to action A_j then the epoch number of A_i is less than or equal to the epoch number of A_j.*

Proof. Since SG(H) has an edge $A_i \rightarrow A_j$, they physically conflict, say on a replica x_s, with the conflicting operations $op_i[x_s]$ and $op_j[x_s]$. Let e_i and e_j be the epoch numbers of s by the time when $op_i[x_s]$ and $op_j[x_s]$ lock x_s. By the definition of a serialization graph, $op_i[x_s]$ happens before $op_j[x_s]$. Then, by Lemma 2, $e_i \leq e_j$.

For a physical read or write executed by a recovery operation to be successful, the epoch number of the replica at the time it is locked must be equal to the epoch number of the recovery operation; the same is true for a physical read executed by a transaction; the condition for successful execution of a physical write run by a transaction is that the epoch number of the replica by the time it is locked must be less or equal to the epoch number of the transaction. Hence, in all cases, $e_i \leq enumber(A_i)$ and $e_j \leq enumber(A_j)$. If $e_i = enumber(A_i)$ then $enumber(A_i) = e_i \leq e_j \leq enumber(A_j)$ as was to be proved. Consider the case where $e_i < enumber(A_i)$. As shown above, $op_i[x_s]$ can only be a write issued by a transaction. Since in the write protocol, every participating replica with epoch number less than the epoch number of the owner transaction adopts the owner transaction's epoch number before releasing the write lock, by the time $op_i[x_s]$ completes, x_s's epoch number e_i' cannot be smaller than $enumber(A_i)$. Also, by Lemma 2, $e_i' \leq e_j$. Thus, we have $enumber(A_i) \leq e_i' \leq e_j \leq enumber(A_j)$ as was to be proved. □

Lemma 4 *SG(H) in acyclic.*

Proof. The proof is by contradiction. Let C be a cycle. By Lemma 3, the epoch numbers of all actions in C must be the same. Let e be this epoch number. First, note that C must include at least one recovery. Indeed, no cycle involving only transactions is possible since execution of transactions is regulated by a concurrency control protocol that prevents such cycles. Now consider the case where all vertices in C are recoveries, so that C is of the form $R_{u_1} \rightarrow R_{u_2} \rightarrow \ldots R_{u_n} \rightarrow R_{u_1}$. Consider two neighboring recoveries, R_{u_i} and R_{u_j}. For the edge $R_{u_i} \rightarrow R_{u_j}$ to exist, one of the two conditions must be satisfied: (a) R_{u_i} and R_{u_j} write to the same replica x_i, and $w_{u_i}[x_i]$ happens before $w_{u_j}[x_i]$. (b) R_{u_i} reads the same replica x_i that R_{u_j} writes to, and $w_{u_i}[x_i]$ happens before $r_{u_j}[x_i]$ Condition (a) is impossible since after $w_{u_i}[x_i]$ performs, x_i becomes non-stale and it will never be stale again in epoch e, which is necessary for $w_{u_j}[x_i]$ to succeed. Therefore, for all neighboring vertices in C, only (b) is possible. Hence, by applying condition (b) to all pairs of neighboring recoveries in C, and since every recovery consists of a physical read of a replica followed by a physical write to another replica, we conclude that $r_{u_1}[x_0]$ happens before $w_{u_1}[x_1]$ that happens before $r_{u_2}[x_1]$ that happens before $w_{u_2}[x_2]$ ... that happens before $r_{u_n}[x_n]$ that happens before $w_{u_n}[x_0]$ that happens before $r_{u_1}[x_0]$, for some replicas $x_0, \ldots, x_n$. Thus, we arrived at the conclusion that $r_{u_1}[x_0]$ happens before itself, which is impossible.

Finally, consider the case where C contains both recoveries and transactions. Then, C must include an arrow from some transaction to some recovery. Let T_i and R_u be such a transaction and a recovery. Since there is an edge $T_i \to R_u$ in SG(H), T_i contains a physical operation that conflicts with and precedes some physical operation in R_u. Because a recovery consists of a read of a replica, $r_u[x_p]$, followed by a write to another replica, $w_u[x_q]$, only three cases are possible:
(1) T_i contains $w_i[x_p]$ that precedes $r_u[x_p]$. Since a recovery requires both the source and the target replicas to be in the same epoch, x_p and x_q are both included in epoch list $elist_e$. Then, the logical write $w_u[x]$, on behalf of which $w_i[x_p]$ is executed, also includes $w_i[x_q]$. Moreover, all replicas of x in epoch e (including x_q) must be locked by $w_i[x]$ before $w_i[x_p]$ can perform. Therefore, since $w_u[x_q]$ happens after $r_u[x_p]$, which happens after $w_i[x_p]$, which happens after $w_i[x]$ acquires a write lock for x_q, $w_u[x_q]$ can acquire the write lock for x_q and perform only after $w_i[x_q]$ performs and releases the lock. The write protocol on Figure 2 provides that, after a physical write executed by a transaction performs on a replica, the replica is either marked non-stale, or its epoch number is greater than the epoch number of the transaction. Hence, upon completion of $w_i[x_q]$, x_q is either non-stale or its epoch number is greater than e. In both cases, $w_u[x_q]$ could not succeed since a recovery performs the write only if the target replica is not stale and in the same epoch as it was when the recovery was initiated. This contradicts the fact that R_u is included in SG(H) since the latter contains only successful actions.
(2) T_i contains $r_i[x_q]$ that precedes $w_u[x_q]$. For $r_i[x_q]$ to succeed, x_q must be non-stale at the time $r_i[x_q]$ performs. Once a replica in non-stale in some epoch, it can never become stale in this epoch again (the only time when a non-stale replica can be marked stale is when it is included into a new epoch). Then, x_q is non-stale by the time $w_u[x_q]$ begins and $w_u[x_q]$ will fail. Thus, we arrive at the same contradiction with the fact that R_u is in SG(H).
(3). T_i contains $w_i[x_q]$ that precedes $w_u[x_q]$. After $w_i[x_q]$ performs, x_q is either non-stale, or its epoch number is greater than e. In both cases, $w_u[x_q]$ could not succeed, and R_u could not be in SG(H). □

Now we intend to add certain edges to SG(H) and prove that the resulting graph is RDSG(H) and that it has no cycles. First, we define a relation that includes any two actions that write to the same data item provided one of the actions is a transaction.

Definition 1 *Let A_i and A_j be two actions that write into some replicas of the same data item x, and at least one of these actions is a transaction. If A_i and A_j both execute in the same epoch then $A_i \Rightarrow_x A_j$ iff there is a common replica x_p of x such that $w_i[x_p] <_H w_j[x_p]$. If A_i and A_j execute in different epochs then $A_i \Rightarrow_x A_j$ iff epoch number of A_i is smaller than epoch number of A_j.*

The following lemma establishes some properties of relation $\Rightarrow_x$.

Lemma 5 *(1). Any two actions that write to the same data item x and such that one of these actions is a transaction, are related by $\Rightarrow_x$; (2) $A_i \Rightarrow_x A_j$ precludes $A_j \Rightarrow_x A_i$; (3) If $A_i \Rightarrow_x A_j$ then epoch number of A_i does not exceed epoch number of A_j.*

Proof. Let A_i and A_j be two actions that write into some replicas of data item x, and one of these actions (say, A_i) is a transaction. If A_i and A_j are executed in different epochs, they are related by $\Rightarrow_x$ according to the order of their epoch numbers, with $A_i \Rightarrow_x A_j$ precluding $A_j \Rightarrow_x A_i$. Assume A_i and A_j run in the same epoch. Since a transaction that writes into one replica of a data item always writes into all replicas of this data item in the transaction's epoch list, A_i writes to all replicas of x in its epoch list. Therefore, because, due to Lemma 1, both A_i and A_j use identical epoch lists, A_i and A_j write to at least one common replica x_p of x. Then, by the definition of histories, $w_i[x_p]$ and $w_j[x_p]$ are related by $<_H$ and hence A_i and A_j are related by $\Rightarrow_x$ accordingly. Moreover, $A_i \Rightarrow_x A_j$ precludes $A_j \Rightarrow_x A_i$ in this case as well. Indeed, if A_j is a recovery, $w_j[x_p]$ is the only write it contains. Also, by the assumption that a transaction contains at most one logical write to a data item, and since a logical write executes at most one physical write on every replica of the data item, $w_i[x_p]$ is the only physical write executed by A_i on x_p. Then, since $w_i[x_p] <_H w_j[x_p]$ precludes $w_j[x_p] <_H w_i[x_p]$, $A_i \Rightarrow_x A_j$ precludes $A_j \Rightarrow_x A_i$. If A_j is a transaction then, because it runs in the same epoch as A_i, it performs physical writes on the same set of replicas of x. (Both transactions perform physical writes on all replicas of x from their epoch lists.) Since both transactions lock all replicas of x that they intend to write before performing a write on any of these replicas, either all physical writes performed by A_i precede all physical writes performed by A_j or vice versa. Hence, either $A_i \Rightarrow_x A_j$ or and $A_j \Rightarrow_x A_i$, but not both. The third claim of the lemma follows directly from the definition of $\Rightarrow_x$. □

Now we can construct RDSG(H).

Lemma 6 (Construction of RDSG(H)) *Let $G(H)$ be $SG(H)$ with edges $T_i \rightarrow T_j$ added for any two transactions T_i and T_j that run in different epochs and such that (1) $T_i \Rightarrow_x T_j$ for some data item x; or (2) there is an action A_k and a data item x such that T_i reads-x-from A_k and $A_k \Rightarrow_x T_j$. Then, $G(H)$ is an $RDSG(H)$.*

Proof. To show that G(H) is an RDSG(H), we need to prove that (1) for any two transactions T_i and T_j that write the same data item x, G(H) has a path either from T_i to T_j or vice versa; and (2) for any three transactions T_i, T_j, and T_k and some data item x, if T_i *reads-x-from* T_k, T_j writes x and $T_k \ll T_j$ then $T_i \ll T_j$.

By Lemma 5, any two transactions T_i and T_j that write the same data item x are related by $\Rightarrow_x$. Hence, if they run in different epochs, there is a path between them by the definition of G(H). If they run in the same epoch, they execute physical writes on the same set of replicas of x, and hence physically conflict. Then, by the definition of history and serialization graph, there is an edge between T_i and T_j in SG(H) and therefore in G(H). So, the first condition required for G(H) to be an RDSG(H) is satisfied.

Now let T_i, T_j, and T_k be any three transactions such that for some data item x, T_i *reads-x-from* T_k, T_j writes x and $T_k \ll T_j$. If T_i and T_j run in different epochs, there is an edge $T_i \rightarrow T_j$ in G(H) by the definition of G(H). If T_i and T_j both run in the same epoch e then, since T_j writes to all replicas of x in e and T_i reads from one replica of x in e, there is a replica x_p such that T_i reads from and T_j writes to. Then, T_i and T_j physically conflict and SG(H) contains an edge between them. Assume for contradiction that $T_i \not\rightarrow T_j$. Then, $T_j \rightarrow T_i$. By assumption that any transaction contains at most one logical read of a given data item, and because a logical read is translated into a single physical read, x_p is the only replica of x that T_i reads. Consider two cases.

If T_i *directly reads-x-from* T_k then T_k writes into x_p. Then, by the definition of histories, either $w_j[x_p] <_H w_k[x_p]$ or $w_k[x_p] <_H w_j[x_p]$. Because the assumption that $T_j \rightarrow T_i$ is in SG(H) entails $w_j[x_p] <_H w_i[x_p]$, $w_k[x_p] <_H w_j[x_p]$ would contradict the assumption that T_i *directly reads-x-from* T_k. Thus, we are left with the condition that $w_j[x_p] <_H w_k[x_p]$, which, by the definition of $\Rightarrow_x$, entails $T_j \Rightarrow_x T_k$ and contradicts to the claim (2) of Lemma 5.

If T_i *indirectly reads-x-from* T_k, there is a sequence of recoveries $R_{u_1}, \ldots, R_{u_n}$ in H such that R_{u_1} *directly reads-x-from* T_k, R_{u_2} *directly reads-x-from* R_{u_1}, ..., and T_i *directly reads-x-from* R_{u_n}. Then, as in the case from the previous paragraph, R_{u_n} writes into x_p and $w_j[x_p] <_H w_{u_n}[x_p] <_H r_i[x_p]$. Therefore, by Lemma 2, R_{u_n} runs in epoch e, the same as T_j. But this would mean that R_{u_n} could not be successful, since after $w_j[x_p]$ completes, x_p can never be stale in epoch e, which is necessary for $w_{u_n}[x_p]$ to succeed. Hence, we have arrived at contradiction with the assumption that R_{u_n} is included in H.

In both cases, we have shown the assumption that $T_i \to T_j$ is not in SG(H) leads to contradictions. Therefore, G(H) has an edge from T_i to T_j. So, the second condition required for G(H) to be an RDSG(H) is satisfied. □

Now all we need to show that our protocol produces one-copy serializable executions of committed transactions is to prove that G(H) is acyclic.

Lemma 7 *If recovery R_u reads-x-from action A_i, and $A_i \Rightarrow_x T_j$ for some transaction T_j then $R_u \Rightarrow_x T_j$.*

R_u *reads-x-from* A_i iff there is a (possibly empty) sequence of recoveries such that R_u *directly reads-x-from* the last recovery in the sequence (or from A_i if the sequence is empty), every recovery in the sequence except for the first one *directly reads-x-from* the previous recovery in the sequence, and the first recovery in the sequence *directly reads-x-from* A_i. We will prove the lemma by induction on the number of recoveries in the sequence.

Basis: R_u *directly reads-x-from* A_i. Let R_u consist from $r_u[x_p]$ followed by $r_u[x_q]$. Since R_u *directly reads-s-from* A_i, A_i includes a physical write into x_p and $w_i[x_p] <_H r_u[x_p]$. Also, since $A_i \Rightarrow_H T_j$, T_j includes a logical write of x.

Case (1). x_p is included into the epoch list of T_j. Then, T_j executes a physical write into x_p. If A_i is a recovery, then the only replica of x it writes to is x_p. Then, by the definition of $\Rightarrow_x$, $w_i[x_p] <_H w_j[x_p]$. If A_i is a transaction then, because A_i and T_j lock all replicas of x they intend to write into before performing any physical writes on these replicas, and since $A_i \Rightarrow_x T_j$ (and thus, there is at a replica of x on which A_i performs the write before T_j does), all physical writes into replicas of x performed by A_i happen before those performed by T_j. In particular, $w_i[x_p] <_H w_j[x_p]$. Therefore in both cases, because R_u *directly reads-x-from* A_i, $r_u[x_p] <_H w_j[x_p]$. Hence, by Lemma 3, $enumber(R_u) \leq enumber(T_j)$. If $enumber(R_u) < enumber(T_j)$ then $R_u \Rightarrow_x T_j$ follows directly from the definition of $\Rightarrow_x$. If $enumber(R_u) = enumber(T_j)$ then, by Lemma 1, R_u and T_j run in the same epoch. Since T_j executes a logical write of x translated into physical writes on all replicas of x in T_j's epoch, and R_u executes a physical write on a replica of x in the same epoch, both R_u and T_j execute a physical write on the same replica of x, say x_q. $w_j[x_q]$ could not happen before $w_u[x_q]$, since in that case, after $w_j[x_q]$ completes, x_q would be marked non-stale and $w_u[x_q]$ (and thus R_u) could not succeed. So, $w_j[x_q]$ happens after $w_u[x_q]$ and, by the definition of $\Rightarrow_x$, $R_u \Rightarrow_x T_j$.

Case (2). x_p is not included into epoch list of T_j $elist(T_j)$. We first prove that in this case $enumber(R_u) \leq enumber(T_j)$ Assume to the contrary that $enumber(R_u) > enumber(T_j)$. Since x_p is included into epoch list of A_i and not into epoch list of T_j, by Lemma 1, $enumber(A_i) \neq enumber(T_j)$. In addition, by Lemma 5, $enumber(A_i) \leq enumber(T_j)$. So, we have $enumber(A_i) < enumber(T_j) < enumber(R_u)$. Consider the epoch with the smallest epoch number e such that $enumber(T_j) < e$ and $x_p \in elist_e$. (e is the first epoch after epoch $enumber(T_j)$ which includes x_p. We know e exists since $x_p \in elist(R_u)$). Since $A_k \Rightarrow_x T_j$, there is a replica x_q of x that T_j writes. Therefore, $x_q \in elist(T_j)$. Because $elist(T_j)$ includes some replicas of x but not x_p, when epoch checking operation includes x_p into epoch e, it marks x_p stale. Since $x_p \in elist(R_u)$, $e \leq enumber(R_u)$ and, by Lemma 2, the node where x_p resides adopted epoch number e and marked x_p stale before $r_u[x_p]$ performed. Hence, before R_u could read x_p, x_p had to recover or some write executed on x_p by a transaction different from A_i (since $e > enumber(A_i)$) had to perform. Both cases contradict with the assumption that R_u *directly reads-x-from* A_i. Thus, we conclude that $enumber(R_u) \leq enumber(T_j)$. Moreover, because x_p is included into epoch list of R_u and not into epoch list of T_j, their epoch numbers are different by Lemma 1. Therefore,

$enumber(R_u) < enumber(T_j)$ and by the definition of $\Rightarrow_x$, $R_u \Rightarrow_x T_j$ as was to be proved.

Induction. Assume the lemma is correct for sequences of recoveries of length not exceeding i and consider a sequence $\{R_{u_1}, \ldots, R_{u_i}, R_{u_{i+1}}\}$ that includes $i+1$ recoveries. So, R_u *directly reads-x-from* $R_{u_{i+1}}$, which *directly reads-x-from* R_{u_i}, ... , which *directly reads-x-from* R_1, which *directly reads-x-from* from A_i. By the inductive assumption, since $A_i \Rightarrow_x T_j$ and $R_{u_{i+1}}$ *reads-x-from* A_i, $R_{u_{i+1}} \Rightarrow_x T_j$. Then, by the basis of the current lemma, $R_u \Rightarrow_x T_j$. □

Lemma 8 *If G(H) has an edge $A_i \rightarrow A_j$, then $enumber(A_i) \leq enumber(A_j)$.*

Proof. If edge $A_i \rightarrow A_j$ is in G(H) then this edge is either in SG(H), or $A_i \Rightarrow_x A_j$ for some x, or A_i and A_j are transactions and there is an action A_k and a data item x such that A_i *reads-x-from* A_k and $A_k \Rightarrow_x A_j$. In the first case, $enumber(A_i) \leq enumber(A_j)$ by Lemma 3. In the second case, the same it true by the definition of $\Rightarrow_x$. Consider the third case. Since A_i and A_j are transactions, we will denote them T_i and T_j below. If T_i *directly reads-x-from* A_k, it can be shown completely similarly to to the proof of the basis of induction from Lemma 7 that $T_i \Rightarrow_x T_j$. Then, by Lemma 5, $enumber(T_i) \leq enumber(T_j)$. If T_i *indirectly reads-x-from* A_k then there is a recovery R_u such that T_i *directly reads-x-from* R_u and R_u *reads-x-from* A_k. By Lemma 7, $R_u \Rightarrow_x T_j$. Then, again by Lemma 7, $T_i \Rightarrow_x T_j$ and, by Lemma 5, $enumber(T_i) \leq enumber(T_j)$. □

Now we are ready to prove the main theorem.

Theorem 2 *H is one-copy serializable.*

Proof. We first prove that G(H) is acyclic. Indeed, suppose for contradiction that G(H) has a cycle C of the form $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow A_1$. Then, by Lemma 8, $enumber(A_1) \leq \ldots \leq enumber(A_n) \leq enumber(A_1)$. Therefore, all actions in cycle C have the same epoch number. By Lemma 4, C cannot have all its edges included in SG(H). Thus, at least one edge in this cycle is in G(H) but not in SG(H). By definition (see Lemma 6), any such edge connects actions that have different epoch numbers. Thus, we have contradiction and conclude that G(H) is acyclic. But, by Lemma 6, G(H) is an RDGS(H). Hence, H has an acyclic RDSG and, by Theorem 1, is one-copy serializable. □.

7 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 and Amr El Abbadi 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] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), pp. 558-565, July 1978.

[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. The dynamic tree protocol: avoiding "graceful degradation" in the tree protocol for distributed mutual exclusion. In *Proc. of the IEEE Int. Phoenix Conf. on Computers andCommunication*, pp. 101-107, April 1992.

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

[15] 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.

Appendix. Fault-Tolerant Epoch Checking

In the algorithm for epoch checking operation presented in the paper we assumed that these operations are always mutually exclusive. This means that if a node fails or gets disconnected while holding the lock for epoch checking, no further epoch checking is possible. We can use some known techniques for fault-tolerance to avoid that. The idea is to use the permission-based dynamic majority consensus protocol [1] for arbitrating among competing epoch checking operations, and the quorum-based three-phase commit protocol [15] to ensure that the nodes to be included into the new epoch update their epoch lists and numbers in the quorum-or-nothing manner. The algorithm as shown here does not address the deadlock problem. For ways to handle deadlocks, see for example [2].

We add two variables to a node's state. One variable, *echecker*, records a unique identity of the epoch operation in which the node is currently participating or has participated last. When an epoch operation coordinator initiates the epoch operation, it generates a unique operation ID using, e.g., Lamport's timestamps [11]. Then, the coordinator includes this ID with all messages it sends to the participants. Once a node agreed to participate in epoch operation and registered its identity in its *echecker*, it ignores messages from any other epoch operations until it completes the current one. Another variable, *state*, indicates the node's stage in the execution of the epoch checking protocol in which it participates. Initially (when a node is not participating in any epoch checking operation), the node is in "*INITIAL*" state.

The algorithm for fault-tolerant epoch checking is shown in Figure 3. A node may initiate epoch checking at an arbitrary moment. The operation consists of four phases. The first phase serves a dual role: (1) it informs the coordinator about the set of nodes with which the coordinator can currently communicate (and which should be included in the new epoch), and (2) it arbitrates among possibly conflicting parallel epoch checking operations. In the first phase, the coordinator broadcasts the request for epoch checking to all nodes in the system, and enters the *waiting* state. Upon receiving the request, a participant first checks if it is already participating in another epoch checking operation, in which case the participant responds with `permission-denied` message. Otherwise, it responds with permission (which also includes the participant's epoch list and epoch number), records the identity of the epoch checking operation in which it agreed to participate in

```
EpochCheck
/* This algorithm is run by the coordinator; the actions of participants are given within the comments.
The coordinator generates a unique ID for the epoch checking operation and includes this ID with every message to participants.
A participant accepts a message only if either its is "epoch-check-request" and the participant is
in "INITIAL state, or the operation ID from the message matches the ID stored in the participant's vechecker.
The corresponding code is not shown in the algorithm below. */
    multicast(all-nodes-in-the-system, "epoch-check-request");
    /* If a node that receives this message is in "INITIAL" state then it records the ID of the epoch
    checking operation received in its echecker, sets its state to "WAITING" and responds with a permission message of the form
    (node, enumber, ELIST, latest-epoch). Otherwise, the node responds with "permission-denied". */
    state := "WAITING";
    receive RESPONSES;
    PERMISSIONS[1:k] := RESPONSES\{"permission-denied" responses};
    /* 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;
        state := "PREPARED";
        multicast(NEW-EPOCH, ("prepare-to-switch", NEW-EPOCH, new-epoch-number));
        /* On receiving the above message, if a participant's state is "WAITING", the participant stores the prospective
        epoch list and number, sets its state to "PREPARED", and responds to the coordinator with the
        acknowledgement. */
        when acknowledgements are received from a majority of members of ELIST_m
            multicast(NEW-EPOCH, "commit");
            /* When a participant receives a "commit" message, it atomically updates its epoch list and number
            to be equal to the prospective epoch list and number recorded in the previous stages on the protocol,
            sets its state to "INITIAL" to indicate that it completed the current epoch checking, and sets echeck-in-progress
            to "NO". */
            state := "INITIAL";
        endwhen;
    endif;
end;
```

Figure 3: The fault-tolerant algorithm for checking an epoch and switching to a new epoch.

its *echecker*, and enters the waiting state. After receiving all responses (the timeouts are counted as `permission-denied`'s), the coordinator checks if it has been able to obtain permission from a majority of members of 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 with permission and, if this list is different from the current epoch, attempts to form a new epoch by initiating the second phase of the protocol. In this phase, as in the algorithm in Figure 1, the coordinator identifies the nodes in the prospective epoch list that have replicas to be marked stale, multicasts a `mark-stale` signal to all these nodes, and updates the prospective epoch list to exclude the nodes that were sent the `mark-stale` signal but failed to acknowledge marking their replicas stale. Then, in the third phase, the coordinator generates the new epoch number by incrementing the current epoch number by 1, broadcasts a `prepare-to-switch` message that contains the new epoch number and epoch list to all members of the prospective epoch, and enters the *prepared* state. Upon receiving acknowledgements from a majority of the current epoch members that they registered the prospective epoch list and number and entered the prepared state, the coordinator executes the final phase of the operation. It sends a `commit` message to all participants. When a participant receives this message, it atomically updates its epoch number and list to be equal to those of the prospective epoch, resets its *echecker* to "*NIL*", and enters the initial state.

When a participant detects a failure or disconnection of the coordinator[2], it initiates a *termination protocol* to releave itself and its fellow participants from their promise to participate only in the current epoch checking operation and thus make it possible for the future epoch checking operations to perform. This protocol is a variant of the termination protocol from [15], with the following termination conditions. Let a node p be called a member of epoch e if it belongs to epoch list kept at some node with epoch number e, and p's epoch number is also e. The epoch checking operation with ID a can be committed by all nodes in a partition only if (1) the partition contains a majority of members of some existing epoch, which are participating in operation a (i.e., they keep a in their *echecker*) and are in the waiting or prepared states, and at least one node participating in epoch checking a is in the prepared state (a majority of members of the existing epoch participating in epoch checking a must be moved to "*PREPARED*" state before committing the operation in this case), or (2) there is a node in "*INITIAL*" state that has a in its *echecker* (i.e., this epoch checking operation committed on some node). Epoch switching can be aborted on all nodes in a partition that contains a majority of members of some existing epoch that are in the initial or waiting state, regardless of the contents of their *echecker*. Unless all members of some majority set of that epoch are in the initial state, a majority of members of that epoch must be moved to a "*PREPARE-TO-ABORT*" state [15] prior to aborting on any node). Aborting epoch checking on a node means reseting its state to the values it had before the current epoch checking operation began on that node.

Note that, as in the standard protocol, once a node chooses to participate in a termination operation, it may not continue participating in the "normal" epoch operation. Moreover, if a node participating in a termination operation receives the request for an epoch operation, it responds with "permission-denied" message even if it is in "*INITIAL*" state.

As shown in [15], there are still situations where some participants may not be able to unlock themselves. However, since permission from only a majority of members of the current epoch is required for an epoch checking operation to proceed, in most cases, subsequent epoch checking/changing operations will not be blocked. To further reduce the possibility of such blocking, one can require that, in the prospective epoch list, the number of nodes from non-current epochs be less

[2]The participant infers that the coordinator is not available if it does not receive an expected message from the coordinator for a long time and a subsequent ping to the coordinator is not successful.

than half the number of nodes that the current epoch contains. In other words, in the algorithm, we would limit the number of elements in *NEW-NODES* to be less than $\frac{|ELIST_m|}{2}$. This would preclude a situation where the transition to the prospective epoch commits in a partition that contains only a minority of members of the new epoch and blocks in a partition with a majority of the new epoch, which would block any future epoch checking operations until the partitioning of the new epoch is fixed.

The described modification of the epoch checking algorithm does not affect read, write and recovery operations. The changes to the proof of correctness would involve only modification of the proofs of Lemmas 1 and 2. Given the proofs of correctness of the dynamic voting protocol for mutual exclusion [1, 13] and the three-phase commit protocol [15], these modifications are a technical matter and not presented here.