Quorum Consensus in Nested Transaction Systems

KENNETH J. GOLDMAN
Washington University
and
NANCY LYNCH
Massachusetts Institute of Technology

Gifford's Quorum Consensus algorithm for data replication is studied in the context of nested transactions and transaction failures (aborts), and a fully developed reconfiguration strategy is presented. A formal description of the algorithm is presented using the Input/Output automaton model for nested transaction systems due to Lynch and Merritt. In this description, the algorithm itself is described in terms of nested transactions. The formal description is used to construct a complete proof of correctness that uses standard assertional techniques, is based on a natural correctness condition, and takes advantage of modularity that arises from describing the algorithm as nested transactions. The proof is accomplished hierarchically, showing that a fully replicated reconfigurable system "simulates" an intermediate replicated system, and that the intermediate system simulates an unreplicated system. The presentation and proof treat issues of data replication entirely separately from issues of concurrency control and recovery.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*concurrency*; *distributed systems*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Concurrency control, data replication, hierarchical proofs, I/O automata, nested transactions, quorum consensus

1. INTRODUCTION

In distributed database systems, data are often *replicated* in order to improve availability, reliability, and performance. Whenever replication is used, a *replication algorithm* is required to ensure that replication is transparent

K. J. Goldman was supported in part by the National Science Foundation under Grant CCR-91-10029. A preliminary version of this paper appeared in *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*. N. Lynch was supported in part by the National Science Foundation under Grant CCR-86-11442, in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, and in part by the Office of Naval Research under Contract N00014-85-0168.
Authors' addresses: K. J. Goldman, Department of Computer Science, Washington University, St. Louis, MO 63130; N. Lynch, Laboratory for Computer Science, MIT, Cambridge, MA 02139.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1994 ACM 0362-5915/94/1200-0537 $03.50

to the user programs. In understanding replication algorithms, it is convenient to think of each *logical object* as being implemented by a collection of *replicas* and *logical access programs*. The replicas retain state information, and the user programs invoke logical access programs in order to read or write the logical object. The logical access programs complete these read or write operations by accessing some subset of the replicas.

One of the best-known replication algorithms is the *quorum consensus* algorithm of Gifford [1979]. Gifford's work does not include transaction nesting, but permits the user to access the logical objects using single-level transactions. Based on the majority-voting algorithm of Thomas [1979], the ideas of this method underlie many of the more recent and sophisticated replication techniques (e.g., Eager and Sevcik [1983], Herlihy [1984], El Abbadi et al. [1985], and El Abbadi and Toueg [1989]). The goal of a replication algorithm is to ensure that each read operation sees enough replicas to ensure that it will return the value written by the "preceding" write operation. This can be guaranteed by a simple read-one/write-all approach (where every replica is updated on each write, and reading a single replica suffices), or by a read-majority/write-majority approach. The quorum consensus algorithm generalizes these approaches as follows. Each replica is assigned a number of *votes*, and retains in its state a data *value* with an associated *version number*. Each logical object X has an associated *configuration* consisting of two integers, called *read-quorum* and *write-quorum*. If k is the total number of votes assigned to replicas of X, then the configuration for X is constrained so that *read-quorum* + *write-quorum* $> k$. To read X, a logical access program collects the version numbers from enough replicas so that it has at least *read-quorum* votes; then it returns the value associated with the highest version number. To write X, a logical access program collects the version numbers from enough replicas so that it has at least *read-quorum* votes; then it writes its value with a higher version number to a collection of replicas with at least *write-quorum* votes.

In our discussion of quorum consensus algorithms for data replication, we use a configuration strategy that is slightly more general than the one described above. In this strategy, justified by Barbara and Garcia-Molina [1985], a configuration consists of a set of read quorums and a set of write quorums. Each quorum is a set of replica names; thus, the voting strategy above corresponds to the special case where any set of replicas is a read quorum if and only if the total votes among its members exceeds the threshold *read-quorum*. To read a logical object, a logical access program accesses all the replicas in some read quorum and chooses the value with the highest version number. To write a logical object, a logical access program first discovers the highest version number written so far by accessing all the replicas in some read quorum; then the logical access increments that version number by one and writes the new value and version number to all the replicas in some write quorum. Every read quorum must have a nonempty intersection with every write quorum. This ensures that each read of the logical object reads at least one of the replicas written by the most recent logical write.

In this paper, Gifford's quorum consensus algorithm is described and proven correct in the context of nested transaction systems that permit users to invoke accesses to logical objects using arbitrary nested transactions. Transaction nesting turns out to be useful not only for describing user programs, but also for describing and understanding the replication algorithms themselves (even if the user transactions themselves are not nested). This is because the logical access programs may be written as subtransactions of the user transactions, and the different tasks performed by a logical access may be written as subtransactions of the logical access transactions. We show how Gifford's algorithm can be structured in this way. Once this is done, it is very natural to allow nesting of user transactions as well.

The quorum consensus algorithm is well suited to systems in which transaction failures (aborts) may occur, and is therefore a natural choice for nested transaction systems in which one of the primary goals is to mask failures. For example, if the replica accesses performed by a logical access program are invoked as subtransactions of that program, then an operation to access a logical data item can complete even if some of its associated replica accesses abort. Our presentation includes a fully developed mechanism for changing the read- and write-quorums dynamically. This mechanism, known as *reconfiguration*, was outlined by Gifford, and has also been studied by Jajodia and Mutchner [1990]. Reconfiguration is useful for coping with site or link failures.

In this paper, we describe and prove the correctness of two algorithms for data replication using a general framework that is suitable for a wide range of distributed algorithms. Two principal algorithms are described in this paper. The first uses a *fixed* configuration, whereas the second supports reconfiguration. Both algorithms are presented in a nested transaction setting where transaction failures are possible. An interesting aspect of the reconfiguration algorithm is that the configuration information itself is replicated and managed by the data replication algorithm.

Separating the concerns of data replication from concurrency control and recovery has been an important goal in the formal treatment of replicated data management algorithms (e.g., El Abbadi and Toueg [1989]). Our presentation achieves this separation in the nested transaction setting. We present the replication issues solely in the context of systems without concurrency. We prove that a system that includes the new replication algorithm and that is serial at the level of the individual data copies looks the same, to the user transactions, as a system that is serial at the level of the logical objects. The fact that both systems involved in this simulation are serial systems helps to simplify the reasoning.

Of course, systems that are truly serial at the level of the data copies are of little practical interest. However, previous work on nested transaction concurrency control and recovery algorithms [Moss 1981; Reed 1983; Lynch and Merritt 1986; Fekete et al. 1987] has produced several interesting algorithms that guarantee that a system *appears to be serial*, as far as the transactions can tell. Combining any of these algorithms (at the replica level) with the replication algorithm we present yields a combined algorithm that appears unreplicated and serial (at the logical data item level), as far as the user transactions can tell. In fact, our results provide a careful proof of the fact that the replication algorithm can be combined with *any* algorithm that guarantees "serializability" at the copy level, to yield a system that is serializable at the logical data item level. Thus, our work formalizes, for the case of nested transaction systems, an important claim from classical concurrency control theory:

> quorum consensus works with any correct concurrency control algorithm. As long as the algorithm produces serializable executions, quorum consensus will ensure that the effect is just like an execution on a single copy database [Bernstein et al. 1987].

We are able to accomplish this clean separation in the case of nested transactions because our correctness conditions are stated from the point of view of the transactions, instead of from the point of view of the database interacting with a particular scheduler. Furthermore, the classical serializability theory correctness conditions are stated in terms of executions of the same system, while our correctness conditions are defined in terms of a separate specification of the allowable executions.

There have been some previous attempts at rigorous presentation and proof of replicated data algorithms. Most notable among these is the presentation and proof given by Bernstein et al. [1987] of Gifford's basic algorithm; this work is based on classical serializability theory. Their approach, however, does not appear to generalize easily to the case where nesting and failures are allowed. Also, Herlihy [1984] extends Gifford's algorithm to accommodate abstract data types and offers a correctness proof. Nested transactions are not considered.

The remainder of the paper is organized as follows. In Section 2 we summarize the system model we use for nested transaction systems, as presented by Fekete et al. [1990]. In Section 3 we give a formal definition of configurations. In Section 4 we describe the algorithm for a fixed configuration, followed by the reconfigurable algorithm in Section 5. Finally, in Section 6, we show that the correctness of interesting concurrent replicated systems follows directly from these results. Section 7 summarizes our results and suggests possible directions for further work.

2. BACKGROUND

We adopt the model of nested transaction systems constructed by Lynch and Merritt [1986] with the I/O automaton model of Lynch and Tuttle [1987] as a foundation. We begin with a review of the I/O automaton model and then describe its use in modeling nested transaction systems. Complete details may be found in Fekete et al. [1990].

We note that whereas classical serializability theory is designed specifically to deal with database concurrency and replica control, the I/O automaton model is well suited for studying a wide range of distributed algorithms. The model provides a foundation for precise problem specifications, detailed algorithm descriptions, and careful correctness proofs, as well as impossibility results.

2.1 The Input / Output Automaton Model

An I/O automaton A has a set of *states*, some of which are designated as *initial states*. Usually a state is given as an assignment of values to a collection of named typed variables. The automaton has *actions*, divided into *input actions*, *output actions*, and *internal actions*. We refer to both input and output actions as *external actions*. An automaton has a transition relation, which is a set of triples of the form (s', π, s), where s' and s are states, and π is an action. This triple means that in state s', the automaton can atomically perform action π and change to state s. An element of the transition relation is called a *step* of the automaton.

The input actions model the actions that are triggered by the environment of the automaton, the output actions model the actions that are triggered by the automaton itself and are potentially observable by the environment, and the internal actions model changes of state that are not directly detected by the environment.

Given a state s' and an action π, we say that π is *enabled* in s' if there is a state s for which (s', π, s) is a step. We require that each input action π be enabled in each state s', i.e., that an I/O automaton must be prepared to receive any input action at any time.

Rather than listing triples, we describe transition relations by associating a precondition and effect with each action. For a given automaton in state s', if the precondition for action π is true in s', then the automaton may perform π and go to state s, as defined by the effect. If π is enabled in all states, the precondition is omitted.

An *execution* of A is an alternating sequence $s_0 \pi_1 s_1 \pi_2 \cdots \pi_n s_n$ of states and actions of A such that s_0 is a start state and each triple (s', π, s) that occurs as a consecutive subsequence is a step of A. The *schedule* of an execution is the sequence of actions that occur in that execution. The *behavior* of an execution of A is the sequence of external actions of A in the execution. A behavior represents the information that the environment can detect about the execution. Since the same action may occur several times in an execution, schedule, or behavior, we refer to a single occurrence of an action as an *event*.

We say that a schedule (behavior) β *can leave A in state s* if there is some execution with schedule (behavior) α and final state s. If schedule (behavior) β' of A is a prefix of schedule (behavior) β of A, we say that state s is reachable from β' in β iff s is the final state in an execution α of A with schedule (behavior) $\beta'\gamma$, where $\beta'\gamma$ is a prefix of β.

We describe systems as consisting of interacting components, each of which is an I/O automaton. It is also convenient and natural to view systems as I/O automata. Thus, we define a composition operation for I/O automata, to yield a new I/O automaton. A collection of I/O automata is said to be *strongly compatible* if (1) every internal action of every automaton is not an action of any other automaton in the collection, (2) every output action of each automaton is not an output action of any other, and (3) no action is shared by infinitely many automata in the collection. A (possibly infinite) collection of strongly compatible automata may be composed to create a *system* $\mathscr{S}$. A state of the composed automaton is a tuple of states, one for which component automaton, and the start states are tuples consisting of start states of the components. An action of the composed automaton is an action of a subset of the component automata. It is an output of the system if it is an output of any component. It is an internal action of the system if it is an internal action of any component. During an action π of $\mathscr{S}$, each of the components that has an action π carries out the action, while the remainder stay in the same state. If β is a sequence of actions of a system with component A, then we denote by $\beta|A$ the subsequence of β containing all the actions of A. Clearly, if β is a finite behavior of the system, then $\beta|A$ is a finite behavior of A. There is an important converse result.

LEMMA 2.1.1. *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata, and let A be the composition of the collection. Suppose β is a finite sequence of external actions of A, such that $\beta|A_i$ is a finite behavior of A_i for every i. Then β is a finite behavior of A.*

Let A and B be automata with the same external actions. Then A is said to *implement* B if every finite behavior of A is a finite behavior of B. One way in which this notion can be used is the following: Suppose we can show that an automaton A is "correct," in the sense that its finite behaviors all satisfy some specified property. Then if another automaton B implements A, B is also correct.

2.2 Serial Systems and Correctness

Serial systems [Fekete et al. 1990] are used to characterize the correctness of a transaction-processing system. Serial systems consist of transaction automata and serial object automata communicating with a serial scheduler automaton.

Transaction automata represent code written by application programmers. Serial object automata serve as specifications for permissible behavior of data objects in the absence of concurrency. They describe the responses the objects should make to arbitrary sequences of operation invocations, assuming that later invocations wait for responses to previous invocations. The serial scheduler handles the communication among the transactions and serial objects, and thereby controls the order in which the transactions can take steps. It ensures that no two sibling transactions are active concurrently and decides whether each transaction commits or aborts. The serial scheduler can permit a transaction to abort only if its parent has requested its creation, but it has not actually been created. Thus, in a serial system, all sets of sibling transactions are run serially, and in such a way that no aborted transaction ever performs any steps.

A serial system allows no concurrency among sibling transactions, and has only a very limited ability to cope with transaction failures. However, serial systems provide useful specifications for correct behavior of other, more interesting systems, and as intermediate models for separating the concerns of replication algorithms from those of concurrency control.

We represent the pattern of transaction nesting, a *system type*, by a set $\mathcal{T}$ of transaction names, organized into a tree by the mapping *parent*, with T_0 as the root. In referring to this tree, we use traditional terminology, such as *child*, *leaf*, *ancestor*, and *descendant*. (A transaction is its own ancestor and descendant.) The leaves of the transaction-naming tree are called *accesses*. The accesses are partitioned so that each element of the partition contains the accesses to a particular object. Additionally, the system type specifies a set of *return values* for transactions (henceforth simply called *values*). If T is a transaction name that is an access to the object name X and v is a value, we say that the pair (T, v) is an *operation* of X.

The tree structure can be thought of as a predefined naming scheme for all possible transactions that might ever be invoked. In any particular execution, however, only some of these transactions will actually take steps. We imagine that the tree structure is known in advance by all components of a system. The tree will, in general, be infinite and have infinite branching.

The classical transactions of concurrency control theory (without nesting) appear in this model as the children of a "mythical" transaction T_0, the root of the transaction tree. Transaction T_0 models the environment in which the rest of the transaction system runs. It has actions that describe the invocation and return of the classical transactions. Only leaf transactions access the data, and so are distinguished as "accesses." Accesses may exist at any level below the root. The internal nodes of the tree model transactions whose function is to create and manage subtransactions, but not to access data directly. A serial system of a given system type is the composition of a transaction automaton for each nonaccess node of the transaction tree, a serial object automaton for each object name, and a serial scheduler. These are described below.

2.2.1 *Transactions*. A nonaccess transaction T is modeled as a *transaction automaton* A_T, an I/O automaton with the following external actions. (In addition to this, A_T may have arbitrary internal actions.)

Input: CREATE(T)
REPORT_COMMIT(T', v'), for every child T' of T, and every return value v' for T'
REPORT_ABORT(T'), for every child T' of T

Output: REQUEST_CREATE(T'), for every child T' of T
REQUEST_COMMIT(T, v), for every return value v for T.

The CREATE input action (an output of the scheduler automaton we describe later) "wakes up" the transaction. The REQUEST_CREATE output action is a request by T for the scheduler to create a particular child transaction. The REPORT_COMMIT input action reports to T the successful completion of one of its children, and returns a value recording the results of that child's execution. The REPORT_ABORT input action reports to T the unsuccessful completion of one of its children, without returning any other information. The REQUEST_COMMIT action is an announcement by T that it has finished its work, and includes a return value.

We leave the executions of particular transaction automata largely unconstrained; the choice of which children to create and what value to return will depend on the particular implementation. For the purposes of the systems studied here, the transactions are "black boxes." Nevertheless, it is convenient to require that all transaction automata preserve[1] certain syntactic conditions, called *transaction well-formedness*, as follows. A sequence β of external action of automaton A_T is *transaction well-formed* for T, provided the following conditions hold:

(1) The first event in β, if any, is a CREATE(T) event, and there are no other CREATE events.
(2) There is at most one REQUEST_CREATE(T') event in β for each child T' of T.
(3) Any report event for child T' of T is preceded by REQUEST_CREATE(T') in β.
(4) There is at most one report event in β for each child T' of T.
(5) If a REQUEST_COMMIT event for T occurs in β, then it is preceded by a report event for each child T' of T for which there is a REQUEST_CREATE(T') in β.
(6) If a REQUEST_COMMIT event for T occurs in β, then it is the last event in β.

A *transaction well-formed* sequence is always a prefix of a sequence that starts with CREATE(T), ends with REQUEST_COMMIT(T, v), and in between has some interleaving of a collection of two-element sequences REQUEST_CREATE(T')REPORT_COMMIT(T', v'), for various children T' of T.

2.2.2 *Serial Objects.* Recall that transaction automata are associated with nonaccess transactions only, and that access transactions model abstract operations on shared data objects. We associate a single I/O automaton with each object name. The external actions for each object are just the CREATE and REQUEST_COMMIT actions for all the corresponding access transactions. Although we give these actions the same kinds of names as the actions of nonaccess transactions, it is helpful to think of the actions of access transactions in other terms also: a CREATE corresponds to an invocation of an operation on the object, while a REQUEST_COMMIT corresponds to a response by the object to an invocation. Thus, we model the serial specification of an object X (describing its activity in the absence of concurrency and

[1]The concept of *preserving* a property of behaviors is formally defined in Fekete et al. [1990] and means that the automaton does not violate the property unless its environment has done so first.

failures) by a *serial object automaton* S_X with the following external actions. (In addition to this, S_X may have arbitrary internal actions.)

Input: CREATE(T), for every access T to X

Output: REQUEST_COMMIT(T, v), for every access T to X and every return value v for T.

As with transactions, while specific objects are left largely unconstrained, it is convenient to require that behaviors of serial objects satisfy certain syntactic conditions. Let α be a sequence of external actions of S_X. We say that α is *serial-object well-formed* for X if it is a prefix of a sequence of the form

CREATE(T_1) REQUEST_COMMIT(T_1, v_1) CREATE(T_2) REQUEST_COMMIT(T_2, v_2) $\cdots$

where $T_i \neq T_j$ when $i \neq j$. We require that every serial object automaton preserve serial-object well-formedness.

Read/Write Serial Objects. One especially important class of serial-object automata are the *read/write serial objects*. Each read/write serial object S_X has an associated domain of values, D, with a distinguished initial value d_0. S_X also has an associated function $kind : accesses(X) \rightarrow \{\text{"}read\text{"}, \text{"}write\text{"}\}$, and an associated function $data : T \in accesses(X) : kind(T) = \text{"}write\text{"} \rightarrow D$. The set of possible return values for each access T where $kind(T) = \text{"}read\text{"}$ is D, while an access T where $kind(T) = \text{"}write\text{"}$ has return value "*OK*". The state of S_X consists of two components: *active* (either "*nil*", or the name of an access to X), and *data* (an element of D). The start state s_0 has $s_0.active = \text{"}nil\text{"}$, and $s_0.data = d_0$. The transition relation is as follows:

```
CREATE(T)
  Effect: s.active = T
    s.active = T

REQUEST_COMMIT(T, v),
  for kind(T) = "write"
  Precondition:
    s'.active = T
    v = "OK"
  Effect:
    s.active = "nil"
    s.data = data(T)

REQUEST_COMMIT(T, v),
  for kind(T) = "read"
  Precondition:
    s'.active = T
    s'.data = v
  Effect:
    s.active = "nil"
```

2.2.3 *Serial Scheduler.* The third kind of component in a serial system is the serial scheduler. The transactions and serial objects have been specified to be any I/O automata whose actions and behavior satisfy simple restrictions. The serial scheduler, however, is a fully specified automaton, particular to each system type. It runs transactions according to a depth-first traversal of the transaction tree. The serial scheduler can choose nondeterministically to abort any transaction whose parent has requested its creation, as long as the transaction has not actually been created. Each child of T whose creation is requested must be either aborted or run to commitment with no siblings overlapping its execution, before T can commit. The result of a transition can be reported to its parent at any time after the commit or abort has occurred.

The actions of the serial scheduler are as follows:

Input: REQUEST_CREATE(T), $T \neq T_0$
REQUEST_COMMIT(T, v)

Output: CREATE(T)
COMMIT(T), $T \neq T_0$
ABORT(T), $T \neq T_0$
REPORT_COMMIT(T, v), $T \neq T_0$
REPORT_ABORT(T), $T \neq T_0$

The REQUEST_CREATE and REQUEST_COMMIT inputs are intended to be identified with the corresponding outputs of transaction and serial-object automata, and correspondingly for the CREATE, REPORT_COMMIT, and REPORT_ABORT output actions. The COMMIT and ABORT output actions mark the point in time where the decision on the fate of the transaction is irrevocable. Thus, for each transaction T' involved in an execution, we have a REQUEST_CREATE from the parent, followed by either (1) an ABORT and REPORT_ABORT from the scheduler or (2) a CREATE from the scheduler, a REQUEST_COMMIT from the transaction, and a COMMIT and REPORT_COMMIT from the scheduler.

Each state s of the serial scheduler consists of six sets, denoted via record notation: *s.create_requested*, *s.created*, *s.commit_requested*, *s.committed*, *s.aborted*, and *s.reported*. The set *s.commit_requested* is a set of operations. The others are sets of transactions. There is exactly one start state, in which the set *create_requested* is $\{T_0\}$, and the other sets are empty. The notation *s.completed* denotes *s.committed* $\cup$ *s.aborted*. Thus, *s.completed* is not an actual variable in the state, but rather a *derived variable* whose value is determined as a function of the actual state variables.

REQUEST_CREATE(T)
  Effect: $s.create\_requested$
    $= s'.create\_requested \cup \{T\}$

REQUEST_COMMIT(T, v)
  Effect: $s.commit\_requested$
    $= s'.commit\_requested \cup \{(T, v)\}$

ABORT(T)
  Precondition:
    $T \in s'.create\_requested -$
      s'.completed
    $T \notin s'.created$
      siblings(T) $\cap$ s'.created $\subseteq$
      s'.completed
  Effect:
    $s.aborted$
      $= s'.aborted \cup \{T\}$

REPORT_COMMIT(T, v)
  Precondition:
    $T \in s'.committed$
    $(T, v) \in$
      $s'.commit\_requested$

```
CREATE(T)
  Precondition:
    $T \in s'.create\_requested - s'.created$
    $T \notin s'.aborted$
    $siblings(T) \cap s'.created \subseteq s'.completed$
  Effect:
    $s.created = s'.created \cup \{T\}$

COMMIT(T)
  Precondition:
    $(T, v) \in s'.commit\_requested$ for some $v$
    $T \notin s'.completed$
  Effect:
    $s.committed = s'.committed \cup \{T\}$
    $T \notin s'.reported$
  Effect:
    $s.reported$
      $= s'.reported \cup \{T\}$
REPORT_ABORT(T)
  Precondition:
    $T \in s'.aborted$
    $T \notin s'.reported$
  Effect
    $s.reported$
      $= s'.reported \cup \{T\}$
```

2.2.4 *Serial Systems and Serial Behaviors.* A *serial system* is the composition of a strongly compatible set of automata consisting of a transaction automaton A_T for each nonaccess transaction name T, a serial-object automaton S_X for each object name X, and the serial-scheduler automaton for the given system type.

The discussion in the remainder of this paper assumes an arbitrary, but fixed, system type and serial system, with A_T as the nonaccess transaction automata, and S_X as the serial-object automata. We use the term *serial behaviors* for the system's behaviors. We give the name *serial actions* to the external actions of the serial system. The COMMIT(T) and ABORT(T) actions are called *completion* actions for T.

If T is a transaction name and π is one of the serial actions CREATE(T), REQUEST_CREATE(T'), REPORT_COMMIT(T', v'), REPORT_ABORT(T'), or REQUEST_COMMIT(T, v), where T' is a child of T, then we define *transaction*(π) to be T. If π is a serial action of the form CREATE(T) or REQUEST_COMMIT(T, v), where T is an access to X, then we define *object*(π) to be X. If β is a sequence of actions, T a transaction name, and X an object name, we define $\beta|T$ to be the subsequence of β consisting of those serial actions π such that *transaction*(π) $= T$, and we define $\beta|X$ to be the subsequence of β consisting of those serial actions π such that *object*(π) $= X$. We define *serial*(β) to be the subsequence of β consisting of serial actions.

If β is a sequence of actions and T is a transaction name, we say T is an *orphan* in β if there is an ABORT(U) action in β for some ancestor U of T. We say the T is *live* in β if β contains a CREATE(T) event but does not contain a completion event for T.

LEMMA 2.2.4.1. *Let β be a serial behavior.*

(1) *If T is live in β and T' is an ancestor of T, then T' is live in β.*
(2) *If T and T' are transaction names such that both T and T' are live in β, then either T is an ancestor of T' or T' is an ancestor of T.*

2.2.5 *Serial Correctness.* The serial system is used to specify the correctness condition that we expect other, more efficient systems to satisfy. We say that a sequence β of actions is *serially correct* for transaction name T provided that there is some serial behavior γ such that $\beta|T = \gamma|T$. We are interested in studying particular systems of automata representing replicated data objects, where each replica uses a concurrency control method and interacts with a controller that passes information between transactions and objects. We will show that all finite behaviors of these systems are serially correct for T_0, the mythical root transaction representing the environment.

We believe serial correctness to be a natural notion of correctness that corresponds precisely to the intuition of how nested transaction systems ought to behave. Serial correctness for T is a condition that guarantees to implementors of T that their code will encounter only situations that can arise in serial executions. Correctness for T_0 is a special case that guarantees that the external world will encounter only situations that can arise in serial executions.

3. CONFIGURATIONS

If S is an arbitrary set, we define a *configuration* c of S to be any pair, $(c.read, c.write)$, where each of $c.read$ and $c.write$ is a nonempty collection of subsets of S, and where every element of $c.read$ has a nonempty intersection with every element of $c.write$. (We will sometimes refer to $c.read$ and $c.write$ as the sets of *read quorums* and *write quorums*, respectively, of configuration c.) We write *configs*(S) for the set of all configurations of S.

For example, if the set S consists of three elements x, y, and z, then one configuration c has $c.read = \{\{x\},\{y\},\{z\}\}$ and $c.write = \{\{x, y, z\}\}$. This configuration corresponds to the read-one/write-all replication strategy. Another configuration has $c.read = \{\{x, y\},\{x, z\},\{y, z\}\}$ and $c.write = \{\{x, y\}, \{x, z\}, \{y, z\}\}$, corresponding to a read-majority/write-majority strategy.

4. FIXED-CONFIGURATION QUORUM CONSENSUS

In this section, we present and prove the correctness of a fixed-configuration quorum consensus algorithm. For simplicity, we carry out the formal development in this paper for the replication of a single object, which we call X. The same arguments apply to any finite number of objects.

First, Section 4.1 defines system $\mathscr{A}$, a serial system that has X as one of its object names. In system $\mathscr{A}$, the object automaton associated with X is a read/write serial-object automaton S_X, representing the logical object to be replicated. System $\mathscr{A}$ is used in order to define system correctness. Then Section 4.2 defines the replicated serial system $\mathscr{B}$. System $\mathscr{B}$ is identical to $\mathscr{A}$, except that logical object S_X is implemented as a collection, S_Y, $Y \in \mathscr{Y}$, of serial read/write objects that we call *replica objects*, and each logical read (or write) of X is replicated as a subtransaction that performs multiple read and write accesses to some subset of the replicas according to the quorum consensus algorithm. Section 4.3 shows that the algorithm is correct by demonstrating a relationship between $\mathscr{B}$ and $\mathscr{A}$; this proof is easy because both systems are serial.

4.1 System $\mathscr{A}$: The Unreplicated Serial System

We begin by defining unreplicated serial system $\mathscr{A}$. System $\mathscr{A}$ is an arbitrary serial system having a distinguished object name X, in which the object automaton S_X associated with X is a read/write serial object with domain D and initial value d_0.

We define la_r and la_w to be sets containing the respective names of the read and write accesses to X, in $\mathscr{A}$, and define $la = la_r \cup la_w$. (Here, la stands for "logical access.")

Since each transaction name $T \in la_r$ is a read access to S_X, the definition of a read/write serial object says that $kind(T) = read$ for each such T. Also, each transaction name $T \in la_w$ has $kind(T) = write$, and also has an associated value, $data(T) \in D$.

4.2 System $\mathscr{B}$: The Fixed-Configuration Quorum Consensus Algorithm

Here we describe system $\mathscr{B}$, which represents the quorum consensus algorithm with a fixed configuration. System $\mathscr{B}$ is a serial system in which object S_X is replicated; that is, it is implemented as several serial objects (replicas), S_Y, $Y \in \mathscr{Y}$, rather than just one. The accesses to X are implemented as subtransaction automata called *logical access automata* (LA's). The LA's use the quorum consensus algorithm to manage the replicas. We allow arbitrary quorums, as long as each read quorum has a nonempty intersection with every write quorum. In order to read the logical object, a read-LA accesses all the replicas in some read quorum, while in order to write the logical object, a write-LA writes to all the replicas in some write quorum. The intersection property guarantees that each read-LA receives at least one copy of the latest logical data value.

Since reads of different replicas could return different data values, a mechanism is needed to distinguish the most recent value from other possible values. Thus, each replica in the quorum consensus algorithm maintains a nonnegative integer *version-number* in addition to its data value. Successive write-LA's use successively larger version numbers, and a read-LA selects the data value associated with the largest version number it receives. This strategy requires each write-LA to learn what version number it is supposed to write, which in turn requires it to learn the largest version number previously used. In order to learn this number, each write-LA first does a preliminary read of a read-quorum of the replicas.

Since both the read-LA and the write-LA programs must perform reads of a read quorum of the replicas, it is helpful to have a subtransaction that performs such a read. This subroutine can be invoked by a read-LA to perform nearly all of its work, and can also be invoked by a write-LA to perform its preliminary read. We can describe such a subroutine in the nested transaction framework by introducing *two* extra levels of nesting. Thus, read-LA and write-LA transactions will have children known as *read-coordinator* transactions, which will be responsible for reading from a read quorum of replicas. Write-LA transactions will also have children known as *write-coordinator* transactions, which will be responsible for writing to a write quorum of replicas. Read-coordinators and write-coordinators will, in turn, have children that are individual read and write accesses to the replicas.

We begin by defining the type of $\mathcal{B}$, and then give the new automata that appear in $\mathcal{B}$ but not in $\mathcal{A}$. The new automata are the replica objects, and the coordinator and LA transaction automata. We define these automata "bottom up," starting with the replicas and the configuration automata.

4.2.1 *System Type.* The type of system $\mathcal{B}$ is defined to be the same as that of $\mathcal{A}$, with the following modifications. First, the object name X is now replaced by a new set of object names $\mathcal{Y}$, representing the replicas. (The replicas will store both a value and version number.) There are some new transaction names, co_r and co_w representing read- and write-coordinators, respectively, and acc_r and acc_w, which are the read and write accesses to the replicas, respectively. We let $co = co_r \cup co_w$ and $acc = acc_r \cup acc_w$.

The transaction names in la are accesses in $\mathcal{A}$, but in $\mathcal{B}$ they are not; each transaction name in la_r now has children that are in co_r, while each transaction name in la_w now has children in co_r and children in co_w. Also, each transaction name in co_r has children in acc_r, and each transaction name in co_w has children in acc_w. These are the only changes to the system type—for example, no transaction names other than those indicated are parents of the new transaction names. The naming structure for logical accesses and their descendants is shown in Figure 1.

In system $\mathcal{A}$, some information is associated with some of the transaction names; for example, each T in la (that is, each access to X) has $kind(T)$ indicating whether the access reads or writes, and write accesses have $data(T)$ indicating the value to be written. In $\mathcal{B}$, we keep all such associated information from $\mathcal{A}$ and add more information as follows. First, every transaction name $T \in co_w$ has an associated value $data(T) \in D$ and an associated version number $version\text{-}number(T) \in N$. These denote, respectively, the value for X and the associated version number to be written by the write-coordinator in the quorum consensus algorithm. Second, we associate some information with the accesses to the replicas. Namely, we assume that every transaction name $T \in acc_r$ has $kind(T) = read$, and that every transaction name $T \in acc_w$ has $kind(T) = write$ and $data(T) \in N \times D$, the version number and value to be written.[2]

Throughout the rest of this section, we let c denote a fixed configuration of $\mathcal{Y}$.

4.2.2 *Replica Automata.* A *replica automaton* is defined for each object name $Y \in \mathcal{Y}$. Each replica is a read/write serial object that keeps a version number and a value for X. More formally, the object automaton for each $Y \in \mathcal{Y}$ in system $\mathcal{B}$ is a read/write serial object for Y with domain $N \times D$

[2]The transaction names in acc_w name accesses to read/write objects. Hence, to agree with the definitions in Section 2.2.2, the version number and data values are combined into a single *data* attribute.

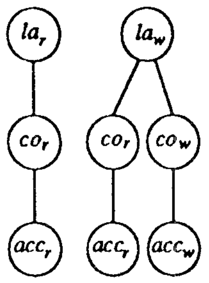

Fig. 1. Logical accesses and their descendants.

```
CREATE(T)
    Effect: $s.awake = true$
        $s.awake = true$

REQUEST_CREATE(T')
    Precondition:
        $s'.awake = true$
        $T' \notin s'.requested$
    Effect:
        $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT(T',v)
    Effect: $s.reported = s'.reported \cup \{T'\}$
        $s.read = s'.read \cup \{object(T')\}$
        if $v.version\text{-}number > s'.version\text{-}number$ then
            $s.version\text{-}number = v.version\text{-}number$
            $s.value = v.value$
        $s.reported = s'.reported \cup \{T'\}$
        $s.read = s'.read \cup \{object(T')\}$
        if $v.version\text{-}number > s'.version\text{-}number$ then
            $s\ version\text{-}number = v.version\text{-}number$
            $s.value = v.value$

REPORT_ABORT(T')
    Effect: $s.reported = s'.reported \cup \{T'\}$
        $s.reported = s'.reported \cup \{T'\}$

REQUEST_COMMIT(T,v)
    Precondition:
        $s'.awake = true$
        $s'.requested = s'.reported$
        $(\exists \mathcal{R} \in c.read)(\mathcal{R} \subseteq s'.read)$
        $v = (s'.version\text{-}number, s'.value)$
    Effect:
        $s.awake = false$
```

Fig. 2. Transition relation for Read-Coordinators.

and initial value $(0, d_0)$. For $v \in N \times D$, we use the record notation *v.version-number* and *v.value* to refer to the components of v.

4.2.3 *Coordinators.* In this section, we define the coordinator automata in $\mathscr{B}$, the transaction automata that are invoked by the LA's and access the replicas. We first define read-coordinators. The purpose of a read-coordinator is to determine the latest version number and value of X, on the basis of the data returned by the read accesses it invokes. A read-coordinator for transaction name $T \in co_r$ has state components *awake*, *value*, *version-number*, *requested*, *reported*, and *read*, where *awake* is a Boolean variable, initially *false*; *value* $\in D$, initially d_0; *version-number* $\in N$, initially 0; *requested* and *reported* are subsets of *children*(T), initially empty; and *read* is a subset of $\mathscr{Y}$, initially empty. The set of return values V_T is equal to $N \times D$.

The transition relation for read-coordinators is shown in Figure 2. Recall that c is a fixed configuration. This is used to determine when the coordinator has collected enough information to be sure of having seen the most recent value of X. A read-coordinator collects data from replicas for object names in $\mathscr{Y}$, and keeps track of the value from the replica with the highest version number seen so far. Whenever the read-coordinator reaches a state in which (1) some read quorum, according to c, is a subset of the replicas it has seen (i.e., those in $s'.read$) and (2) it has learned of the fates (i.e., commit or abort) of all of its requested child transactions, then the read-coordinator may request to commit and return its data.

We have written the read-coordinator with a high degree of nondeterminism. It may request any number of accesses to replicas, at any time while it is active. In practice, one would probably start at most one access to each replica at a time, and then request another one only after the failure of an earlier access to that replica. There is also a natural trade-off in practice between two methods of choosing which replicas to access at first: one could request initially only enough access to complete one read quorum, and then request others if these fail to commit in reasonable time, or else one could request accesses to all replicas concurrently. The first method minimizes communication cost in the best case (when all accesses complete successfully), and the second reduces latency in the bad case where some accesses fail. Each of these possibilities, and many others, could be modeled as an implementation of the automaton given here, obtained by restricting nondeterminism.

We next define write-coordinators. The purpose of a write-coordinator is to write a given value to a write quorum of replicas for object names in $\mathscr{Y}$. A write-coordinator for transaction name $T \in co_w$ has state components *awake*, *requested*, *reported*, and *written*, where *awake* is a Boolean variable, initially *false*; *requested* and *reported* are subsets of *children*(T), initially empty; and *written* is a subset of $\mathscr{Y}$, initially empty. The set of return values V_T is equal to {"*OK*"}.

The transition relation for a write-coordinator named $T \in co_w$ is shown in Figure 3. When created, a write-coordinator begins invoking write accesses to replicas for object names in $\mathscr{Y}$, overwriting the version numbers and values at the replicas with its own (*version-number*(T) and *data*(T), respectively). After writing to a write quorum, the write-coordinator may request to commit.

4.2.4 *Logical Access Automata.* Now we define logical access automata, beginning with read-LA's. The purpose of a read-LA for a transaction name $T \in la_r$ is to perform a logical read access to X. A read-LA has state components *awake*, *value*, *requested*, *reported*, and *value-read*, where *awake* and *value-read* are Boolean variables, initially *false*; *value* $\in D \cup \{nil\}$, initially *nil*; and *requested* and *reported* are subsets of *children*(T), initially empty. The set of return values V_T is equal to D.

The transition relation for read-LA's is shown in Figure 4. The read-LA invokes any number of read-coordinators. After receiving a REPORT_COMMIT from at least one of these coordinators, and receiving reports of the completion of all that were invoked, the read-LA may request to commit,

```
CREATE($T$)
  Effect: $s.awake = true$
    $s.awake = true$

REQUEST_CREATE($T'$)
  Precondition:
    $s'.awake = true$
    $T' \notin s'.requested$
    $data(T') = (version\text{-}number(T), data(T))$
  Effect:
    $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$)
  Effect: $s.reported = s'.reported \cup \{T'\}$
    $s.written = s'.written \cup \{object(T')\}$
    $s.reported = s'.reported \cup \{T'\}$
    $s.written = s'.written \cup \{object(T')\}$

REPORT_ABORT($T'$)
  Effect: $s.reported = s'.reported \cup \{T'\}$
    $s.reported = s'.reported \cup \{T'\}$

REQUEST_COMMIT($T$,$v$)
  Precondition:
    $s'.awake = true$
    $s'.requested = s'.reported$
    $(\exists \mathcal{W} \in c.write)(\mathcal{W} \subseteq s'.written)$
    $v = $ "$OK$"
  Effect:
    $s.awake = false$
```

Fig. 3. Transition relation for Write-Coordinators.

```
CREATE($T$)
  Effect: $s.awake = true$
    $s.awake = true$

REQUEST_CREATE($T'$)
  Precondition:
    $s'.awake = true$
    $T' \notin s'.requested$
  Effect:
    $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$)
  Effect: $s.reported = s'.reported \cup \{T'\}$
    if $s'.value\text{-}read = false$ then
        $s.value = v.value$
        $s.value\text{-}read = true$
    $s.reported = s'.reported \cup \{T'\}$
    if $s'.value\text{-}read = false$ then
        $s.value = v.value$
        $s.value\text{-}read = true$

REPORT_ABORT($T'$)
  Effect: $s.reported = s'.reported \cup \{T'\}$
    $s.reported = s'.reported \cup \{T'\}$

REQUEST_COMMIT($T$,$v$)
  Precondition:
    $s'.awake = true$
    $s'.requested = s'.reported$
    $s'.value\text{-}read = true$
    $v = s'.value$
  Effect:
    $s.awake = false$
```

Fig. 4. Transition relation for Read LA's.

returning the value component of the data returned by the first read-coordinator.

We next define write-LA's. The purpose of a write-LA for a transaction name in la_w is to perform a logical write access to X on behalf of a user transaction. A write-LA has state components *awake*, *value-read*, *value-written*, *version-number*, *requested*, and *reported*, where *awake*, *value-read*, and *value-written* are Booleans, initially *false*; *version-number* $\in N \cup \{nil\}$, initially *nil*; and *requested* and *reported* are subsets of *children*(T), initially empty. The set of return values V_T is equal to {"OK"}.

```
CREATE($T$)
    Effect: $s.awake = true$
        $s\ awake = true$

REQUEST_CREATE($T'$), $T' \in co_r$
    Precondition:
        $s'.awake = true$
        $T' \notin s'.requested$
    Effect:
        $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$), $T' \in co_r$
    Effect: $s.reported = s'.reported \cup \{T'\}$
        if $s'.value\text{-}read = false$ then
            $s.version\text{-}number = v.version\text{-}number$
            $s.value\text{-}read = true$
        $s.reported = s'.reported \cup \{T'\}$
        if $s'.value\text{-}read = false$ then
            $s.version\text{-}number = v.version\text{-}number$
            $s.value\text{-}read = true$

REQUEST_CREATE($T'$), $T' \in co_w$
    Precondition:
        $s'.awake = true$
        $s'.value\text{-}read = true$
        $data(T') = data(T)$
        $version\text{-}number(T') = s'.version\text{-}number + 1$
        $T' \notin s'.requested$
    Effect:
        $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$), $T' \in co_w$
    Effect: $s.reported = s'.reported \cup \{T'\}$
        $s'.value\text{-}written = true$
        $s.reported = s'.reported \cup \{T'\}$
        $s'.value\text{-}written = true$

REPORT_ABORT($T'$)
    Effect: $s.reported = s'.reported \cup \{T'\}$
        $s.reported = s'.reported \cup \{T'\}$

REQUEST_COMMIT($T$,$v$)
    Precondition:
        $s'.awake = true$
        $s'.requested = s'.reported$
        $s'.value\text{-}written = true$
        $v = \text{“}OK\text{”}$
    Effect:
        $s.awake = false$
```

Fig. 5. Transition relation for Write-LA's.

The transition relation for write-LA's is shown in Figure 5. A write-LA invokes any number of read-coordinators. After receiving a REPORT_COMMIT from one of the read-coordinators, the write-LA remembers the version number returned. The write-LA then invokes any number of write-coordinators, using a version number one greater than that returned with the first REPORT_COMMIT of a read-coordinator, along with its particular data value. In order for the write-LA to request to commit, it must receive at least one REPORT_COMMIT of a write-coordinator.

4.3 Correctness Proof

In this section, we prove the correctness of the quorum consensus algorithm given above as system $\mathscr{B}$. That is, we show that $\mathscr{B}$ is indistinguishable from $\mathscr{A}$, to the transactions and objects the two systems have in common. We begin by stating some definitions that are useful for reasoning about the sequence of logical accesses in an execution of $\mathscr{B}$. In each definition, β is a sequence of actions of $\mathscr{B}$.

First, the *logical access sequence* of β, denoted *logical-sequence*(β), is defined to be a subsequence of β containing the CREATE(T) and REQUEST_COMMIT(T, v) events, where $T \in la$. That is, the logical access sequence is the sequence of requests and responses for the logical accesses to X.

Next, if β is finite, then *logical-value*(β) is defined to be either *data*(T) if REQUEST_COMMIT(T, v) is the last REQUEST_COMMIT event for a transaction name in la_w that occurs in *logical-sequence*(β), or d_0 if no such REQUEST_COMMIT event occurs in *logical-sequence*(β). In other words, the logical value is the value of the last logical write (or the initial value of the logical object if no such write occurs).

Finally, if β is finite, then *current-vn*(β) is defined to be the highest *version-number* among the states of all replica automata in the global state led to by β.

The next lemma is the key to the proof of Theorem 4.3.7, the main correctness theorem. Condition (1) of the lemma, which enumerates some properties on the state led to by particular schedules of $\mathscr{B}$, is only needed for carrying out the inductive argument. These properties say that (a) there is a write quorum in which every replica has the current version number and (b) any replica with the current version number also has the logical value as its data. The more important part of the lemma is condition (2), which says that each read-LA returns the value associated with the previous logical write. (That is, each read-LA returns the *logical-value* of the logical object.) The schedules of $\mathscr{B}$ which we consider are those in which no logical access to X is active.[3] The fact that $\mathscr{B}$ is a serial system makes the reasoning simple (though detailed).

LEMMA 4.3.1. *Let β be a finite schedule of $\mathscr{B}$ such that no logical access to X is active in β.*

(1) *The following properties hold in any global state led to by β:*

  (*a*) *There exists a write quorum $\mathscr{W} \in c.write$ such that for any replica S_Y with object name $Y \in \mathscr{W}$, if v is the data component of S_Y, then $v.version\text{-}number = current\text{-}vn(\beta)$.*

  (*b*) *For any replica S_Y, if v is the data component of S_Y and $v.version\text{-}number = current\text{-}vn(\beta)$ then $v.value = logical\text{-}value(\beta)$.*

(2) *If β ends in REQUEST_COMMIT(T, v) with $T \in la_r$, then $v = logical\text{-}value(\beta)$.*

PROOF. Since β is a serial system, Lemma 2.2.4.1 shows that *logical-sequence*(β) consists of a sequence of pairs, each of the form CREATE(T) REQUEST_COMMIT(T, v), where $T \in la$. We proceed by induction on the number of such pairs. For the basis, suppose *logical-sequence*(β) contains no such pairs. Then *logical-value*$(\beta) = d_0$. Since β contains no CREATE(T) events for $T \in la$, it contains no REQUEST_COMMIT events for accesses in *acc*; therefore, since all replicas initially have *version-number* $= 0$ and *value* $= d_0$, this is also the case in any global state led to by β. It follows that

[3]Recall that a transaction name is active in a sequence if the sequence contains a CREATE action but no REQUEST_COMMIT action for that transaction.

current-vn$(\beta) = 0$. This implies condition (1), and condition (2) holds vacuously.

For the inductive step, suppose that *logical-sequence*(β) contains $k \geq 1$ pairs, and assume that the lemma holds for sequences with $k - 1$ pairs. That is, let $\beta = \beta'\gamma$, where *logical-sequence*(γ) begins with the last CREATE event in *logical-sequence*(β), and assume that the lemma holds for β'. (Note that Lemma 2.2.4.1 shows that no logical access to X is live in β'.) So, *logical-sequence*(γ) = CREATE(T_f)REQUEST_COMMIT(T_f, v_f) for some $T_f \in la$ and v_f of the appropriate type. The following observation enables us to argue only in terms of the transaction subtree rooted at T_f. □

Claim 4.3.2. Any access or coordinator for which a CREATE or REQUEST_COMMIT action appears in γ is a descendant of T_f.

Proof. This follows since $\mathscr{B}$ is a serial system. □

Now, since T_f has a REQUEST_COMMIT in γ, we know by the definition of T_f that there is at least one REPORT_COMMIT event for a transaction name in co_r in γ. Let T' be the read-coordinator in co_r with the first REPORT_COMMIT in γ; by Claim 4.3.2, $T' \in$ *children*(T_f). Let γ' be the portion of γ up to and including the REPORT_COMMIT event for T'. If T_f is in la_r, then the system type shows that it invokes no write-coordinator, while if T_f is in la_w, the code shows that T_f invokes no write-coordinator before receiving a REPORT_COMMIT for read-coordinator. Thus no actions of any write-coordinator or its descendants appear in γ'. Therefore, there are no REPORT_COMMIT events in γ' for descendants of T_f that are write accesses. We now give two claims about the states of the automata for T_f and T'.

Claim 4.3.3. Let s be the state of the transaction automaton associated with T' in any global state reachable from β' in $\beta'\gamma'$. Then *s.version-number* and *s.value* contain the highest *version-number* and associated *value* among the states led to by β' of the replicas whose names are in *s.value-read*.

Proof. This claim holds because T' retains the maximum *version-number* reported by a read access, together with its associated *value* upon each REPORT_COMMIT of a read access. Again, since no write accesses occur in γ', the *version-number* and *value* components of all replicas observed by T' must be the same during γ' as in the state led to by β'. □

Claim 4.3.4. Let s be the state of the automaton associated with T_f in any global state reachable from $\beta'\gamma'$ in β. Then *s.version-number* = *current-vn*(β') and *s.value* = *logical-value*(β').

Proof. Let s' be the state of T' just before it issues its REQUEST_COMMIT event in γ'. By the definition of a read-coordinator, s'*.read* must contain a read quorum $\mathscr{R} \in c.read$. By 1(a) of the inductive hypothesis, there is some write quorum $\mathscr{W} \in c.write$ such that the states of all replicas for object names in $\mathscr{W}$ in any global state led to by β' have *version-number* = *current-vn*(β'). Since c is a configuration, $\mathscr{R}$ and $\mathscr{W}$ have a nonempty intersection. So s'*.read* must contain at least one object name in $\mathscr{W}$. So, by Claim 4.3.3, *s'.version-number* = *current-vn*(β'). Therefore, by 1(b) of the inductive hypothesis, *s'.value* = *logical-value*(β').

When T' reports its commit to T_f, the *version-number* and *value* components of T_f are set equal to *s'.version-number* and *s'.value*, respectively. By definition of T_f, these components are never again modified. Therefore, Claim 4.3.4 is proved. □

We now return to the main proof of Lemma 4.3.1, and consider the two possible cases for T_f.

(1) $T_f \in la_r$. Then *logical-value*(β) = *logical-value*(β') by definition. Also, since T_f invokes only read-coordinators, which in turn invoke only read accesses, the *version-number* and *value* components of the states of the replicas in any global state led to by β are the same as in any global state led to by β', and so *current-vn*(β) = *current-vn*(β'). Therefore, condition (1) holds for β.

By definition, T_f cannot request to commit until at least one of its read-coordinators commits. Since T' is the first read-coordinator child that reports its commit, the REQUEST_COMMIT for T_f must occur at some point after $\beta'\gamma'$. When T_f requests to commit, it returns the *value* component of its state. By Claim 4.3.4, this value is *logical-value*(β') = *logical-value*(β). Thus, condition (2) holds for β.

(2) $T_f \in la_w$. Then *logical-value*(β) = *data*(T_f). We first give two claims about the information associated with the descendants of T_f invoked during γ.

Claim 4.3.5. If T is a write-coordinator invoked by T_f, then *version-number*(T) = *current-vn*(β') + 1 and *data*(T) = *data*(T_f).

PROOF. Let s be the state of T_f just before the REQUEST_CREATE(T) event that occurs in γ. By definition, *version-number*(T) = *s.version-number* + 1 and *data*(T) = *data*(T_f). Also by definition, T_f cannot invoke a write-coordinator until at least one of its read-coordinators reports its commit. So, all REQUEST_CREATE events for write-coordinators in γ occur after $\beta'\gamma'$. By Claim 4.3.4, *s.version-number* = *current-vn*(β'). Thus, Claim 4.3.5 holds. □

Claim 4.3.6. If T is a write access for an object in $\mathscr{Y}$ invoked in γ, then *data*(T) = (*current-vn*(β') + 1, *data*(T_f)).

PROOF. The type of $\mathscr{B}$ is constrained so that T is invoked by some write-coordinator. The result follows from Claim 4.3.5 and the definition of a write-coordinator. □

By definition, T_f cannot request to commit until it receives a REPORT_COMMIT from at least one of its write-coordinators. Let T_w be the write-coordinator child of T_f that has the first COMMIT event in γ, and let δ be the portion of γ up to and including COMMIT(T_w). By definition, T_w cannot request to commit until it has received REPORT_COMMIT events for write accesses to a write quorum of replicas. Therefore, by Claim 4.3.6, it follows that *current-vn*$(\beta'\delta)$ = *current-vn*$(\beta')+1$, so condition (1) holds in any global state led to by $\beta'\delta$.

We now show that condition (1) still holds in any global state led to by β. By Claim 4.3.6, any write-coordinators that may execute in γ after δ merely propagate the new value and version number. Any read-coordinators that may execute in γ after δ cannot change the values at the replicas, since they do not invoke write accesses. Therefore, condition (1) holds after β.

Since T_f does not name a read-LA, condition (2) holds vacuously.

Thus in both cases, the lemma holds. (This ends our proof of Lemma 4.3.1.)
□

Now we prove that $\mathscr{B}$ is correct by showing that transactions cannot distinguish between replicated serial system $\mathscr{B}$ and unreplicated serial system $\mathscr{A}$. Here, let $\mathscr{T}_A$ and $\mathscr{X}_A$ denote the transaction and object names of $\mathscr{A}$, respectively.

THEOREM 4.3.7. *Let β be a finite schedule of $\mathscr{B}$. Then there exists a schedule γ of $\mathscr{A}$ such that the following two conditions hold*:

(1) *$\gamma|T = \beta|T$ for each transaction name $T \in \mathscr{T}_{\mathscr{A}} - la$, and*
(2) *$\gamma|Y = \beta|Y$ for each object name $Y \in \mathscr{X}_A$.*

PROOF. We construct γ by removing from β all REQUEST_CREATE(T), CREATE(T), REQUEST_COMMIT(T, v), COMMIT(T), ABORT(T), REPORT_COMMIT(T, v), and REPORT_ABORT(T) events for all transaction names T in $co \cup acc$. Clearly, the two conditions in the statement of the theorem hold. It remains to show that γ is a schedule of $\mathscr{A}$. By Lemma 2.1.1, it suffices to show that γ projects to yield a schedule of each component of $\mathscr{A}$. It is easy to see that, for each nonaccess transaction name T of $\mathscr{A}$, $\gamma|T$ is a schedule of the transaction automaton for T, and for each object name $Y \in \mathscr{X}_A - \{X\}$, $\gamma|Y$ is a schedule of the object automaton for Y. Since we construct γ by removing all the actions associated with a specific set of transactions, and since none of these transactions have children with actions in γ, the fact that β is a schedule of the serial scheduler implies that γ is as well.

It remains to consider the automaton for object name X: we must show that $\gamma|X$ is a schedule of S_X, where S_X is the serial object associated with X in $\mathscr{A}$. We proceed by induction on the length of β. The basis, where β is of length 0, is trivial. Suppose that $\beta = \beta'\pi$, where π is a single event and the result is true for β'. The only case that is not immediate is the case where π is a REQUEST_COMMIT event for a transaction name in la. So suppose that π = REQUEST_COMMIT(T, v), where $T \in la$. Let $\gamma = \gamma'\pi$; by the inductive hypothesis, $\gamma'|X$ is a schedule of S_X. Let s' be the state of S_X led to by $\gamma'|X$.

One precondition for REQUEST_COMMIT(T, v) as an output of S_X is that $T \in s'.created$. By transaction well-formedness of β, CREATE(T) occurs in β'. By the construction, CREATE(T) occurs in γ', so that this precondition is satisfied. We consider the two possible cases for T in order to show that the remaining clauses of the precondition for REQUEST_COMMIT(T, v) are satisfied.

(1) $T \in la_w$. Then the definition of a write-LA implies that $v =$"*OK*", which satisfies the remaining precondition.

(2) $T \in la_r$. Then by the construction, $\gamma'|X =$ *logical-sequence*(β'). If there is a REQUEST_COMMIT for a write access to S_X in γ', let T' be the last such write access. Then $s'.data$ is equal to $data(T')$. By the construction, we know that T' is also the name of the logical write access in $\mathscr{B}$ having the last REQUEST_COMMIT in β'. Therefore, $data(T') =$ *logical-value*(β'). So the *data* component of the state of S_X led to by γ' is *logical-value*(β').

On the other hand, if there is no REQUEST_COMMIT for a write access to S_X in γ', then $s'.data$ is d_0, which is *logical-value*(β'). Therefore, the *data* component of the state of S_X led to by γ' is again *logical-value*(β').

Thus, in either case, *logical-value*(β') is the *data* component of the state of S_X in any global state led to by γ'. By Lemma 4.3.1, we know that the return value $v =$ *logical-value*(β'). Thus, v is the *data* component of the state, s', of S_X led to by γ', which implies that the remaining precondition for REQUEST_COMMIT(T, v) in S_X is satisfied.

Therefore, γ is a schedule of $\mathscr{A}$. □

5. RECONFIGURABLE QUORUM CONSENSUS

Now we present the algorithm with reconfiguration. For the concurrent implementations that are our ultimate goal, the activity of changing the configuration must interact correctly with reading the configuration during processing of a logical access. Thus each reconfiguration needs to take its place as a transaction in the transaction nesting structure, accessing a configuration object that holds the current configuration. This might be done with reconfiguration transactions as children of T_0; however, the correctness arguments work equally well no matter which existing transaction is taken as parent of each reconfiguration transaction. Thus we allow each reconfiguration to occur at an arbitrary location in the transaction tree.

The presentation and proof are carried out in stages. First, in Section 5.1, we define system $\mathscr{A}$, a serial system that has X as one of its object names[4] and that also has another special object name Z. As in Section 4, the object automaton S_X associated with X is a read/write serial object automaton. The object automaton S_Z associated with Z is a special *dummy* object, which simply receives requests from transactions to change the current configuration and responds to them with a trivial acknowledgment. As before, system $\mathscr{A}$ is used to define correctness.

Then in Section 5.2, we define replicated serial system $\mathscr{B}$. System $\mathscr{B}$ is identical to $\mathscr{A}$, except that logical object S_X is replicated as in system $\mathscr{B}$ of Section 4, and also dummy object S_Z is replaced by a *configuration object automaton* $S_{\bar{X}}$, which is a read/write object containing the current configura-

[4]In this section, we redefine certain notation from Section 4, such as the system names $\mathscr{A}$ and $\mathscr{B}$.

tion of replicas of X. In $\mathscr{B}$, reconfiguration requests involve a single write of object $S_{\bar{X}}$. In the course of performing each logical read or write access to X, the configuration object $S_{\bar{X}}$ is read to determine the current configuration. In Section 5.3, we show that the new algorithm is correct by demonstrating a relationship between $\mathscr{B}$ and $\mathscr{A}$. The proof is analogous to that in Section 4.3; the changes involve the handling of the new configuration object.

Next, in Section 5.4, we define serial system $\mathscr{C}$, which is the same as $\mathscr{B}$ except that the configuration object is replicated. That is, the configuration object $S_{\bar{X}}$ from $\mathscr{B}$ is replaced in $\mathscr{C}$ by a collection of serial read/write objects that we call *configuration replica objects*, and each access to a configuration object is implemented by a subtransaction that performs accesses to a subset of the configuration replicas. The configuration replica management is based in an interesting way on the data replica management scheme. Finally, in Section 5.5, we prove a simulation relationship between $\mathscr{C}$ and $\mathscr{B}$, which in turn implies a similar connection between $\mathscr{C}$ and $\mathscr{A}$.

5.1 System $\mathscr{A}$: The Unreplicated Serial System with Dummy Reconfiguration

We begin by defining the unreplicated serial system $\mathscr{A}$. System $\mathscr{A}$ is an arbitrary serial system having two distinguished object names, X and Z, in which the object automaton S_X associated with X is a read/write serial object with domain D and initial value d_0, and in which the object automaton S_Z associated with Z is a special *dummy* object automaton, defined below.

As before, we define la_r and la_w to be the respective names of the read and write accesses to X. Now we also define la_{rec} to be the set of names of accesses to the dummy object Z. We define $la = la_r \cup la_w \cup la_{rec}$.[5] As before, each transaction name $T \in la_r$ has $kind(T) = read$, and each transaction name $T \in la_w$ has $kind(T) = write$ and also has an associated value, $data(T) \in D$.

Recall that system $\mathscr{A}$ is used to define correctness, and that correctness is defined at the transaction boundary. Thus, it is desirable that we use the same transaction interfaces in $\mathscr{A}$ as we do in the later systems, $\mathscr{B}$ and $\mathscr{C}$. Since those systems will permit arbitrary transaction automata to invoke reconfiguration operations, we also allow them to do so in $\mathscr{A}$. However, since X is not replicated in $\mathscr{A}$, these operations will not do anything interesting in $\mathscr{A}$. We simply make them operations on the dummy object, which merely responds "OK" in all cases.[6]

[5]Note that this is different from the definition in Section 4, where la was just defined to be the union of la_r and la_w.

[6]It may seem artificial to place the invocations of the reconfiguration transactions in the transaction automaton interface in $\mathscr{A}$; an alternative approach would be to make these reconfigurations invisible to users and application programmers by *not* having them appear explicitly in $\mathscr{A}$. In this approach, $\mathscr{A}$ would be defined just as it is in Section 4. An auxilliary system $\mathscr{A}'$ would be defined, which would represent an augmentation of $\mathscr{A}$ with added reconfiguration operations, where the augmentation is presumably made by programmers of the database system rather than application programmers. We leave the details of this approach to interested readers, and settle instead for the simpler alternative of including reconfiguration requests in $\mathscr{A}$

```
CREATE(T)                  REQUEST_COMMIT(T,v)
  Effect: s.active = T       Precondition:
    s.active = T               T = s'.active
                               v = "OK"
                             Effect:
                               s.active = nil
```

Fig. 6. Dummy object automaton.

5.1.1 *Dummy Object Automata.* More precisely, we define a single *dummy object automaton* S_Z, which is a serial-object automaton having a single state component, $active \in accesses(Z) \cup \{nil\}$, initially *nil*. The set of return values V_T for accesses to Z is equal to {"*OK*"}. The (trivial) transition relation for S_Z is shown in Figure 6.

5.2 System $\mathcal{B}$: The Reconfigurable Quorum Consensus Algorithm with a Centralized Configuration

In this section, we define replicated serial system $\mathcal{B}$. This is similar to system $\mathcal{B}$ of Section 4 in that object S_X is implemented by replicas S_Y, $Y \in \mathcal{Y}$, and the accesses to X are implemented as subtransaction automata called read-LA's and write-LA's, using the same basic strategy as in the fixed-configuration algorithm. Additionally, in the new $\mathcal{B}$, the dummy object S_Z is implemented by a *configuration object* $S_{\bar{X}}$, which is a read/write serial object that maintains a configuration, and the accesses to Z are implemented as subtransactions called *reconfigure-LA*'s that invoke accesses to $S_{\bar{X}}$. (In $\mathcal{C}$, the configuration objects themselves will be replicated, and the reconfigure-LA's will invoke coordinator subtransactions to read and write them.)

We first define the type of $\mathcal{B}$, and then give the new automata that appear in $\mathcal{B}$ but not in $\mathcal{A}$. The new automata are the replica objects, the configuration object, and the coordinator and LA transaction automata. We again define these automata "bottom up."

5.2.1 *System Type.* The type of system $\mathcal{B}$ is defined to be the same as that of $\mathcal{A}$, with the following modifications. As in Section 4, the object name X is now replaced by a new set of object names $\mathcal{Y}$, and there are new transaction names, co_r and co_w, representing read- and write-coordinators, respectively, and acc_r and acc_w, which are the read and write accesses to the replicas, respectively. Additionally, object name Z is now replaced by a new object name $\bar{X}$, and there are new transaction names co_{rc} and co_{wc}, which are the read and write accesses to object name $\bar{X}$, respectively. (We denote the configuration accesses in this way, because in $\mathcal{C}$ they will be replaced by coordinator automata.) We let $co = co_r \cup co_w \cup co_{rc} \cup co_{wc}$.

The transactions names in la are accesses in $\mathcal{A}$, but in $\mathcal{B}$ they are not; each transaction name in la_r now has children that are in co_{rc} and co_r; each transaction name in la_w now has children in co_{rc}, co_r, and co_w; and each transaction name in la_{rec} now has children in co_{rc}, co_{wc}, co_r, and co_w. Also, as in Section 4, each transaction name in co_r has children in acc_r, and each transaction name in co_w has children in acc_w. These are the only changes to

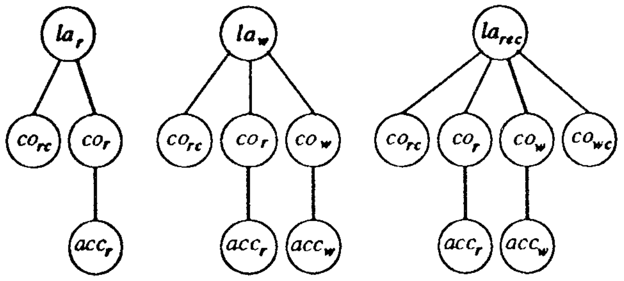

Fig. 7. Logical accesses and their descendants.

the system type—for example, no transaction names other than those indicated are parents of the new transaction names. The naming structure for logical accesses and their descendants is shown in Figure 7.

In defining $\mathscr{A}$, we associated information with some of the transaction names. In $\mathscr{B}$, we keep all such information from $\mathscr{A}$ and add more information as follows.

First, to denote the new configuration for each logical reconfigure access, we associate with every transaction name $T \in la_{rec}$ a configuration *config*(T) in *configs*$(\mathscr{Y})$, the set of all configurations of $\mathscr{Y}$.

Second, to denote the configuration that is used by each read-coordinator, we assume that every transaction name $T \in co_r$ has an associated configuration *config*$(T) \in$ *configs*$(\mathscr{Y})$. Similarly, every transaction name $T \in co_w$ has an associated value *data*$(T) \in D$, an associated version number *version-number*$(T) \in N$, and an associated configuration *config*$(T) \in$ *configs*$(\mathscr{Y})$. These denote the value for X written by the write-coordinator, the associated version number in the quorum consensus algorithm, and the configuration to be used for writing the new value and version number.

Third, we associate some information with the accesses to both the configuration objects and the replicas. We assume that every transaction name $T \in co_{rc}$ has *kind*(T) = *read*, and that every transaction name $T \in co_{wc}$ has *kind*(T) = *write* and *data*$(T) \in$ *configs*$(\mathscr{Y})$, representing the new configuration. Also, as before, we assume that every transaction name $T \in acc_r$ has *kind*(T) = *read* and that every transaction name $T \in acc_w$ has *kind*(T) = *write* and *data*$(T) \in N \times D$.

5.2.2 *Replica and Configuration Automata.* As in Section 4, we define a *replica automaton* for each object name $Y \in \mathscr{Y}$. This is a read/write serial object for Y with domain $N \times D$ and initial value $(0, d_0)$. As before, for $v \in N \times D$, we write *v.version-number* and *v.value* to refer to the components of v.

For the rest of this paper, let C = *configs*$(\mathscr{Y})$, and let $c_0 \in C$ be a distinguished *initial configuration.* The *configuration automaton* is defined to be a read/write serial object for $\bar{X}$, with domain C and initial value c_0. (Recall that its read accesses are the elements of co_{rc}, and its write accesses are the elements of co_{wc}.)

```
CREATE($T$)
    Effect: $s.awake = true$
        $s.awake = true$

REQUEST_CREATE($T'$)
    Precondition:
        $s'.awake = true$
        $T' \notin s'.requested$
    Effect:
        $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$)
    Effect: $s.reported = s'.reported \cup \{T'\}$
        $s.read = s'.read \cup \{object(T')\}$
        if $v.version\text{-}number > s'.version\text{-}number$ then
            $s.version\text{-}number = v.version\text{-}number$
            $s.value = v.value$
        $s.reported = s'.reported \cup \{T'\}$
        $s.read = s'.read \cup \{object(T')\}$
        if $v.version\text{-}number > s'.version\text{-}number$ then
            $s.version\text{-}number = v.version\text{-}number$
            $s.value = v.value$

REPORT_ABORT($T'$)
    Effect: $s.reported = s'.reported \cup \{T'\}$
        $s.reported = s'.reported \cup \{T'\}$

REQUEST_COMMIT($T$,$v$)
    Precondition:
        $s'.awake = true$
        $s'.requested = s'.reported$
        $(\exists \mathcal{R} \in config(T).read)(\mathcal{R} \subseteq s'.read)$
        $v = (s'.version\ number, s'.value)$
    Effect:
        $s.awake = false$
```

Fig. 8. Transition relation for Read-Coordinators.

5.2.3 *Coordinators.* In this section, we define the coordinator automata in $\mathcal{B}$. These transaction automata are invoked by the LA's and access the replicas. We first define read-coordinators; these are the same as those in Section 4, except that instead of using the fixed configuration, the new read-coordinator uses its own associated configuration. A read-coordinator for transaction name $T \in co_r$ has state components *awake*, *value*, *version-number*, *requested*, *reported*, and *read*, where *awake* is a Boolean variable, initially *false*; $value \in D$, initially d_0; $version\text{-}number \in N$, initially 0; *requested* and *reported* are subsets of *children*(T), initially empty; and *read* is a subset of $\mathcal{Y}$, initially empty. The set of return values V_T is equal to $N \times D$.

The transition relation for read-coordinators is shown in Figure 8. This code is identical to that in Figure 2, except that *config*(T) is used to determine when the coordinator has collected enough information to be assured of having seen the most recent value of X.

We next define write-coordinators. Again, these are the same as those in Section 4, except that instead of using the fixed configuration, the new write-coordinator uses its own associated configuration. A write-coordinator for transaction name $T \in co_w$ has state components *awake*, *requested*, *reported*, and *written*, where *awake* is a Boolean variable, initially *false*; *requested* and *reported* are subsets of *children*(T), initially empty; and *written* is a subset of $\mathcal{Y}$, initially empty. The set of return values V_T is equal to {"*OK*"}.

The transition relation for a write-coordinator named $T \in co_w$ is shown in Figure 9. This code is identical to that in Figure 3, except that *config*(T) is used to determine when the coordinator has written to enough replicas.

```
CREATE($T$)
    Effect: $s.awake = true$
        $s.awake = true$

REQUEST_CREATE($T'$)
    Precondition:
        $s'.awake = true$
        $T' \notin s'.requested$
        $data(T') = (version\text{-}number(T), data(T))$
    Effect:
        $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$)
    Effect: $s.reported = s'.reported \cup \{T'\}$
        $s.written = s'.written \cup \{object(T')\}$
        $s.reported = s'.reported \cup \{T'\}$
        $s.written = s'.written \cup \{object(T')\}$

REPORT_ABORT($T'$)
    Effect: $s.reported = s'.reported \cup \{T'\}$
        $s.reported = s'.reported \cup \{T'\}$

REQUEST_COMMIT($T$,$v$)
    Precondition:
        $s'.awake = true$
        $s'.requested = s'.reported$
        $(\exists \mathcal{W} \in config(T).write)(\mathcal{W} \subseteq s'.written)$
        $v = \text{"}OK\text{"}$
    Effect:
        $s.awake = false$
```

Fig 9. Transition relation for Write-Coordinators.

5.2.4 *Logical Access Automata.* Now we define logical access automata, beginning with read-LA's. These are the same as those in Section 4, except that the new read-LA first invokes a read-configuration-coordinator in order to determine the current configuration. It then invokes only read-coordinators whose associated configuration is the same as the determined current configuration.

A read-LA has state components *awake*, *config*, *value*, *requested*, *reported*, *value-read*, and *config-read*, where *awake*, *value-read*, and *config-read* are Boolean variables, initially *false*; $config \in C \cup \{nil\}$, initially *nil*; $value \in D \cup \{nil\}$, initially *nil*; and *requested* and *reported* are subsets of *children*(T), initially empty. The set of return values V_T is equal to D.

The transition relation for read-LA's is shown in Figure 10. The changes to the code in Figure 4 are as follows. First, there are additional REQUEST_CREATE and REPORT_COMMIT actions for invoking and obtaining responses from the configuration-coordinators. There are two additional state components: *config*, which keeps track of the configuration returned, and *config-read*, a flag that records that a configuration has been read. Also, there are extra clauses in the precondition for the REQUEST_CREATE events for read-coordinators, which ensure that only read-coordinators with the current configuration are invoked.

We next define write-LA's. These are the same as those in Section 4, except that the new write-LA first invokes a read-configuration-coordinator in order to determine the current configuration, and then invokes only read-coordinators and write-coordinators having the proper associated configuration. A write-LA has state components *awake*, *config*, *config-read*, *value-read*, *value-written*, *version-number*, *requested*, and *reported*, where *awake*, *config-read*, *value-read*, and *value-written* are Booleans, initially *false*; $config \in C \cup \{nil\}$, initially *nil*; $version\text{-}number \in N \cup \{nil\}$, initially *nil*; and *requested* and *reported* are subsets of *children*(T), initially empty. The set of return values V_T is equal to $\{\text{"}OK\text{"}\}$.

```
CREATE(T)
    Effect: s.awake = true
        s.awake = true

REQUEST_CREATE(T'), T' ∈ co_rc
    Precondition:
        s'.awake = true
        T' ∉ s'.requested
    Effect:
        s.requested = s'.requested ∪ {T'}

REPORT_COMMIT(T',v), T' ∈ co_rc
    Effect: s.reported = s'.reported ∪ {T'}
        if s'.config-read = false then
            s.config = v
            s.config-read = true
        s.reported = s'.reported ∪ {T'}
        if s'.config-read = false then
            s.config = v
            s.config-read = true

REQUEST_CREATE(T'), T' ∈ co_r
    Precondition:
        s'.awake = true
        s'.config-read = true
        config(T') = s'.config
        T' ∉ s'.requested
    Effect:
        s.requested = s'.requested ∪ {T'}

REPORT_COMMIT(T',v), T' ∈ co_r
    Effect: s.reported = s'.reported ∪ {T'}
        if s'.value-read = false then
            s.value = v.value
            s.value-read = true
        s.reported = s'.reported ∪ {T'}
        if s'.value-read = false then
            s.value = v.value
            s.value-read = true

REPORT_ABORT(T')
    Effect: s.reported = s'.reported ∪ {T'}
        s.reported = s'.reported ∪ {T'}

REQUEST_COMMIT(T,v)
    Precondition:
        s'.awake = true
        s'.requested = s'.reported
        s'.value-read = true
        v = s'.value
    Effect:
        s.awake = false
```

Fig. 10. Transition relation for Read LA's.

The transition relation for write-LA's is shown in Figure 11. The changes to the code in Figure 5 are as follows. First, there are additional REQUEST_CREATE and REPORT_COMMIT actions for invoking and obtaining responses from the configuration-coordinators, and associated state components for keeping track of the results. There are also extra clauses in the precondition for the REQUEST_CREATE events for read-coordinators and write-coordinators, ensuring that only coordinators with the current configuration are invoked.

Finally, we define reconfigure-LA's. The purpose of a reconfigure-LA is to change the current configuration to a given target configuration. The main thing it does in order to accomplish this change is to invoke a write access to the configuration object. However, this alone is not enough; the reconfigure-LA transaction must also maintain the crucial property that *all the members of some write quorum, according to the current configuration, must have the latest version number and data value.* (The importance of this property can be seen in the proof of Lemma 4.3.1.) An arbitrary configuration change might cause this property to become violated.

Thus, extra activity is required on the part of the reconfigure-LA to ensure that the latest data gets propagated to some write quorum of the new configuration. In particular, the reconfigure-LA must first obtain this latest

```
CREATE($T$)
  Effect: $s.awake = true$
    $s.awake = true$

REQUEST_CREATE($T'$), $T' \in co_{rc}$
  Precondition:
    $s'.awake = true$
    $T' \notin s'.requested$

  Effect:
    $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$), $T' \in co_{rc}$
  Effect: $s.reported = s'.reported \cup \{T'\}$
    if $s'$ $config\text{-}read = false$ then
        $s$ $config = v$
        $s.config\text{-}read = true$
    $s.reported = s'.reported \cup \{T'\}$
    if $s'.config\text{-}read = false$ then
        $s$ $config = v$
        $s.config\text{-}read = true$

REQUEST_CREATE($T'$), $T' \in co_r$
  Precondition:
    $s'.awake = true$
    $s'.config\text{-}read = true$
    $config(T') = s'.config$
    $T' \notin s'.requested$
  Effect·
    $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$), $T' \in co_r$
  Effect: $s.reported = s'.reported \cup \{T'\}$
    if $s'.value\text{-}read = false$ then
        $s.version\text{-}number = v.version\text{-}number$
        $s.value\text{-}read = true$
    $s.reported = s'.reported \cup \{T'\}$
    if $s'.value\text{-}read = false$ then
        $s.version\text{-}number = v.version\text{-}number$
        $s.value\text{-}read = true$

REQUEST_CREATE($T'$), $T' \in co_w$
  Precondition:
    $s'.awake = true$
    $s'.value\text{-}read = true$
    $config(T') = s'.config$
    $data(T') = data(T)$
    $version\text{-}number(T') = s'.version\text{-}number + 1$
    $T' \notin s'.requested$
  Effect:
    $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$), $T' \in co_w$
  Effect: $s.reported = s'.reported \cup \{T'\}$
    $s'.value\text{-}written = true$
    $s.reported = s'.reported \cup \{T'\}$
    $s'.value\text{-}written = true$

REPORT_ABORT($T'$)
  Effect: $s.reported = s'$ $reported \cup \{T'\}$
    $s$ $reported = s'.reported \cup \{T'\}$

REQUEST_COMMIT($T$,$v$)
  Precondition:
    $s'.awake = true$
    $s'.requested = s'$ $reported$
    $s'.value\text{-}written = true$
    $v = \text{"}OK\text{"}$
  Effect:
    $s$ $awake = false$
```

Fig. 11. Transition relation for Write LA's.

data, which requires that it first obtain the value of the old configuration from a read-configuration-coordinator and then the latest data value from a read-coordinator with the *old* configuration. Once it has the latest data, the reconfigure-LA must cause this data to be written to some write quorum, according to the *new* configuration. It accomplishes this by invoking write-coordinators whose corresponding configuration is the new configuration. (Note that the reconfigure-LA can *invoke* the write-coordinators and the write accesses to the configuration object concurrently, even though in the serial systems considered here, the subtransactions themselves will be run serially. This concurrent invocation will be useful in the concurrent systems we study later, where these activities can run in parallel to reduce latency.)

A reconfigure-LA for transaction name $T \in la_{rec}$ has state components *awake*, *config*, *value*, *version-number*, *requested*, *reported*, *config-read*, *value-read*, *value-written*, and *config-written*, where *awake*, *config-read*, *value-read*, *value-written*, and *config-written* are Boolean variables, initially *false*; *config* is in $C \cup \{nil\}$, initially *nil*; *value* $\in D \cup \{nil\}$, initially *nil*; *version-number* $\in N \cup \{nil\}$, initially *nil*; and *requested* and *reported* are subsets of *children*(T), initially empty. The set of return values V_T is equal to $\{\text{“}OK\text{”}\}$.

The transition relation for reconfigure LA's is shown in Figure 12. Just as for the read- and write-LA's, a reconfigure-LA first determines the current configuration with a read access to the configuration object. Then the reconfigure-LA invokes read-coordinators using that configuration; when the first read-coordinator reports its commit, the reconfigure-LA remembers the value and version number returned. Then the reconfigure-LA may invoke any number of write-coordinators, using its new configuration, along with the value and version number returned by the read-coordinator. This propagates the current value to every replica in a write quorum of the new configuration, as needed. Concurrently, the LA invokes at least one write access to the configuration object in order to record the new configuration. In order to request to commit, the reconfigure-LA must receive REPORT_COMMIT responses for at least one write-coordinator and at least one write access to $\bar{X}$.

5.3 Correctness Proof

In this section, we prove the correctness of system $\mathscr{B}$. As in Section 4, we begin with some definitions. In each definition, β is a sequence of actions of $\mathscr{B}$. First, the *logical access sequence* of β, *logical-sequence*(β), is defined exactly as in Section 4 to be the subsequence of β containing the CREATE(T) and REQUEST_COMMIT(T, v) events, where $T \in la$. (But now this definition is based on the new definition of la, which includes the names in la_{rec}.) Also, if β is finite, then *logical-value*(β) and *current-vn*(β) are defined as before, to be the value of the last logical write (or the initial value of the logical object if no such write occurs), and the highest *version-number* among the local states of all replica automata in any global state led to by β, respectively.

We require one new definition, analogous to *logical-value*(β). Namely, if β is finite, then *logical-config*(β) is defined to be either *config*(T) if REQUEST_COMMIT(T, v) is the last REQUEST_COMMIT event for a transaction name in la_{rec} that occurs in *logical-sequence*(β), or c_0 if no such REQUEST_COMMIT event occurs. In other words, the logical configuration is the value of the last logical reconfigure access (or the initial configuration if no such reconfigure access occurs).

The next lemma is the key to the proof of Theorem 5.3.11, the main correctness theorem. As in Lemma 4.3.1, condition (1) is only needed for the inductive argument. Parts (a) and (b) of condition (1) are as before, and part (c) says that the configuration object holds the logical configuration. As

```
CREATE($T$)
    Effect: $s'.awake = true$
        $s'.awake = true$

REQUEST_CREATE($T'$), $T' \in co_{rc}$
    Precondition:
        $s'.awake = true$
        $T' \notin s'.requested$

    Effect:
        $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$), $T' \in co_{rc}$
    Effect: $s.reported = s'.reported \cup \{T'\}$
        if $s'\ config\text{-}read = false$ then
            $s\ config = v$
            $s.config\text{-}read = true$
        $s.reported = s'.reported \cup \{T'\}$
        if $s'.config\text{-}read = false$ then
            $s.config = v$
            $s.config\text{-}read = true$

REQUEST_CREATE($T'$), $T' \in co_r$
    Precondition:
        $s'\ awake = true$
        $s'.config\text{-}read = true$
        $config(T') = s'\ config$
        $T' \notin s'.requested$
    Effect:
        $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$), $T' \in co_r$
    Effect: $s.reported = s'.reported \cup \{T'\}$
        if $s'.value\text{-}read = false$ then
            $s.value = v.value$
            $s.version\text{-}number = v.version\text{-}number$
            $s.value\text{-}read = true$
        $s.reported = s'.reported \cup \{T'\}$
        if $s'.value\text{-}read = false$ then
            $s.value = v\ value$
            $s.version\text{-}number = v.version\text{-}number$
            $s.value\text{-}read = true$

REQUEST_CREATE($T'$), $T' \in co_w$
    Precondition:
        $s'.awake = true$
        $s'\ value\text{-}read = true$
        $data(T') = s'.value$
        $version\text{-}number(T') = s'\ version\text{-}number$
        $config(T') = config(T)$
        $T' \notin s'.requested$
    Effect:
        $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$), $T' \in co_w$
    Effect: $s.reported = s'.reported \cup \{T'\}$
        $s'.value\text{-}written = true$
        $s.reported = s'.reported \cup \{T'\}$
        $s'.value\text{-}written = true$

REQUEST_CREATE($T'$), $T' \in co_{wc}$
    Precondition:
        $s'.awake = true$
        $s'.value\text{-}read = true$
        $data(T') = config(T)$
        $T' \notin s'\ requested$
    Effect:
        $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT($T'$,$v$), $T' \in co_{wc}$
    Effect: $s\ reported = s'.reported \cup \{T'\}$
        $s.config\text{-}written = true$
        $s\ reported = s'.reported \cup \{T'\}$
        $s.config\text{-}written = true$

REPORT_ABORT($T'$)
    Effect: $s\ reported = s'.reported \cup \{T'\}$
        $s.reported = s'.reported \cup \{T'\}$

REQUEST_COMMIT($T$,$v$)
    Precondition:
        $s'.awake = true$
        $s'.requested = s'.reported$
        $s'.value\text{-}written = true$
        $s'.config\text{-}written = true$
        $v = \text{"}OK\text{"}$
    Effect:
        $s.awake = false$
```

Fig. 12. Transition relation for Reconfigure LA's.

before, the important part of the lemma is condition (2), which tells us that each read-LA returns the value associated with the previous logical write.

LEMMA 5.3.1. *Let β be a finite schedule of $\mathscr{B}$ such that no logical access to X is active in β.*

(1) *The following properties hold in any global state led to by β:*

  (*a*) *There exists a write quorum $\mathscr{W} \in$ logical-config(β).write such that for any replica S_Y for an object name $Y \in \mathscr{W}$, if v is the data component of S_Y, then v.version-number = current-vn(β).*

(*b*) *For any replica S_Y, if v is the data component of S_Y and v.version-number = current-vn*(β) *then v.value = logical-value*(β).

(*c*) *The data component of the configuration object is logical-config*(β).

(2) *If β ends in REQUEST_COMMIT(T, v) with $T \in la_r$, then v = logical-value*(β).

Proof. As before, *logical-sequence*(β) consists of a sequence of pairs, each of the form CREATE(T) REQUEST_COMMIT(T, v), where $T \in la$. We proceed by induction on the number of such pairs. For the basis, suppose *logical-sequence*(β) contains no such pairs. Then *logical-config*(β) = c_0. Parts (a) and (b) follow as before, with *logical-config*(β) used in place of the fixed configuration. For part (c), since β contains no CREATE(T) events for $T \in la$, it contains no REQUEST_COMMIT events for accesses to the configuration object; therefore, since the *data* component of the configuration object is initially c_0, this is also the case in any global state led to by β. This yields part (c), and condition (2) holds vacuously.

For the inductive step, let $\beta = \beta'\gamma$, where *logical-sequence*(γ) begins with the last CREATE event in *logical-sequence*(β), and assume that the lemma holds for β'. So, *logical-sequence*(γ) = CREATE(T_f) REQUEST_COMMIT(T_f, v_f) for some $T_f \in la$ and v_f of the appropriate type. As before, the fact that the system is serial shows that only descendants or ancestors of T_f have actions in γ; in particular:

Claim 5.3.2. If T is in $acc_r \cup acc_w$ and if CREATE(T) or REQUEST_COMMIT(T, v) occurs in γ, then $T \in$ *descendants*(T_f). Also, if T is in *co* and if CREATE(T) or REQUEST_COMMIT(T, v) occurs in γ, then $T \in$ *children*(T_f).

There is at least one REPORT_COMMIT event for a transaction name in co_r in γ; let T' be the read-coordinator in co_r with the first REPORT_COMMIT in γ. By Claim 5.3.2, $T' \in$ *children*(T_f). Let γ' be the portion of γ up to and including the REPORT_COMMIT event for T'. Then, because the code shows that no LA requests the creation of any write-coordinator (for replicas or configuration objects) until after a read-coordinator has returned a value, we see that there are no REPORT_COMMIT events in γ' for descendants of T_f that are write accesses.

Also, since γ must contain a REQUEST_CREATE event for a transaction name in co_r, the definition of T_f implies that there is at least one REPORT_COMMIT event in γ' for an access in co_{rc}; let T'' be the access in co_{rc} with the first REPORT_COMMIT in γ', and let γ'' be the portion of γ up to and including this REPORT_COMMIT event. We now give several claims about the states of the automata for T_f and T'.

Claim 5.3.3. Let s be the state of the transaction automaton associated with T_f in any global state reachable from $\beta'\gamma''$ in β. Then *s.config* = *logical-config*(β').

Proof. We have argued that there are no REPORT_COMMIT events in γ' for write accesses that are descendants of T_f. So, by Claim 5.3.2, no write accesses occur in γ'. Therefore, by part (c) of the inductive hypothesis, T'' returns as its value *logical-config*(β'). By definition of T_f, *s.config* is permanently set to the value returned by T''. Therefore, the claim holds. □

Claim 5.3.4. Let s be the state of the transaction automaton associated with T' in any global state reachable from β' in $\beta'\gamma'$. Then *s.version-number* and *s.value* contain the highest *version-number* and associated *value* among the states led to by β' of the replicas whose names are in *s.value-read.*

Claim 5.3.5. Let s be the state of the transaction automaton associated with T_f in any global state reachable from $\beta'\gamma'$ in β. Then *s.version-number* = *current-vn*(β') and *s.value* = *logical-value*(β').

PROOF. Let s' be the state of T' just before it issues its REQUEST_COMMIT event in γ'. By the definition of a read-coordinator, s'*.read* must contain a read quorum $\mathcal{R} \in$ *config*(T')*.read*. Claim 5.3.3 and the definition of T_f imply that *config*(T') = *logical-config*(β'), so the read quorum $\mathcal{R}$ is in *logical-config*(β')*.read*. By part (a) of the inductive hypothesis, there is some write quorum $\mathcal{W} \in$ *logical-config*(β')*.write* such that the states of all replicas for object names in $\mathcal{W}$ in any global state led to by β' have *version-number* = *current-vn*(β'). Since *logical-config*(β') is a configuration, $\mathcal{R}$ and $\mathcal{W}$ must have a nonempty intersection. So s'*.read* must contain at least one object name in $\mathcal{W}$. So, by Claim 5.3.4, s'*.version-number* = *current-vn*(β'). Therefore, by part (b) of the inductive hypothesis, s'*.value* = *logical-value*(β').

When T' reports its commit to T_f, the *version-number* and *value* components of T_f are set equal to s'*.version-number* and s'*.value*, respectively. By definition of T_f, these components are never again modified. Therefore, Claim 5.3.5 is proved. □

We now return to the main proof. We consider the three possible cases for T_f.

(1) $T_f \in la_r$. The *logical-value*(β) = *logical-value*(β'). Since T_f invokes only read-configuration-coordinators and read-coordinators, which in turn invoke only read accesses, the *version-number* and *value* components of the states of the replicas in any global state led to by β are the same as in any global state led to by β', so *current-vn*(β) = *current-vn*(β'). Likewise, we have *logical-config*(β) = *logical-config*(β'), and the *data* component of the configuration object is the same in any global state led to by β and β'. Therefore, condition (1) holds for β. The proof for condition (2) is as before, this time based on Claim 5.3.5.

(2) $T_f \in la_w$. Then *logical-value*(β) = *data*(T_f) and *logical-config*(β) = *logical-config*(β'). We prove two claims about the information associated with the descendants of T_f invoked in γ.

Claim 5.3.6. If T is a write-coordinator invoked by T_f then *version-number*(T) = *current-vn*(β') + 1, *data*(T) = *data*(T_f), and *config*(T) = *logical-config*(β').

PROOF. Let s be the state of T_f just before the REQUEST_CREATE(T) event that occurs in γ. By definition, *version-number*(T) = s.*version-number* + 1, *data*(T) = *data*(T_f), and *config*(T) = s.*config*. Also, by definition, T_f cannot invoke a write-coordinator until at least one of its read-coordinators reports its commit. So all REQUEST_CREATE events for write-coordinators in γ occur after $\beta'\gamma'$. Therefore, by Claim 5.3.3, s.*config* = *logical-config*(β'), and by Claim 5.3.5, s.*version-number* = *current-vn*(β'). Thus, Claim 5.3.6 holds. □

Claim 5.3.7. If T is a write access for an object in $\mathscr{Y}$ invoked in γ, then *data*(T) = (*current-vn*(β') + 1, *data*(T_f)).

PROOF. By Claim 5.3.6, and the definition of a write-coordinator. □

As before, T_f cannot request to commit until it receives a REPORT_COMMIT from at least one of its write-coordinators. Let T_w be the write-coordinator child of T_f that has the first COMMIT events in γ, and let δ be the portion of γ up to and including COMMIT(T_w). T_w cannot request to commit until it has received REPORT_COMMIT events for write accesses to a write quorum of replicas, according to *config*(T_w). By Claim 5.3.6, *config*(T_w) = *logical-config*(β'), so T_w must receive REPORT_COMMIT events for a write quorum in *logical-config*(β'). Therefore, by Claim 5.3.7, it follows that *current-vn*$(\beta'\delta)$ = *current-vn*(β') + 1, so condition (1) holds in any global state led to by $\beta'\delta$. As before, but this time using Claim 5.3.7, condition (1) still holds for β. Condition (2) holds vacuously.

(3) $T_f \in la_{rec}$. Then *logical-value*(β) = *logical-value*(β') and *logical-config*(β) = *config*(T_f). We first give three claims about the information associated with descendants of T_f invoked in γ.

Claim 5.3.8. If T is a write-coordinator invoked by T_f then *version-number*(T) = *current-vn*(β'), *data*(T) = *logical-value*(β'), and *config*(T) = *config*(T_f).

PROOF. Analogous to that of Claim 5.3.6. □

Claim 5.3.9. If T is a write access invoked by a write-coordinator in γ, then *data*(T) = (*current-vn*(β'), *logical-value*(β')).

PROOF. By Claim 5.3.8 and the definition of a write-coordinator. □

Claim 5.3.10. If T is a write access in co_{wc} invoked by T_f in γ, then *data*(T) = *config*(T_f).

PROOF. By the definition of T_f. □

It follows from Claim 5.3.9 that *current-vn*(β) = *current-vn*(β'). By definition, T_f cannot request to commit until a REPORT_COMMIT for at least one of its write-coordinators occurs. Let T_w be any write-coordinator child of T_f that has a COMMIT event in γ. By Claim 5.3.8, *version-number*(T_w) = *current-vn*(β'); *data*(T_w) = *logical-value*(β'); and *config*(T_w) = *config*(T_f). By definition of a write-coordinator and Claim 5.3.9, just prior to the

REQUEST_COMMIT for T_w in γ, there must be a write quorum $\mathscr{W} \in$ *config*(T_w).*write* such that all replicas for object names in $\mathscr{W}$ have *data.version-number* = *version-number*(T_w). By the equalities above, $\mathscr{W}$ is a write quorum in *logical-config*(β), and all replicas for object names in $\mathscr{W}$ have *data-version-number* = *current-vn*(β). This shows part (a).

Claim 5.3.9 implies that all write accesses in γ have *data* = (*current-vn*(β'), *logical-value*(β')), which is equal to (*current-vn*(β), *logical-value*(β)). Since part (b) holds for β', it also holds for β. Claim 5.3.10 and the definition of a reconfigure-LA imply that the data component of the state of the configuration object in any global state led to by β is equal to *config*(T_f), and hence to *logical-config*(β). This shows part (c). Since T_f does not name a read-LA, condition (2) holds vacuously.

Thus in all three cases, the lemma holds. The end of the proof for Lemma 5.3.1. □

Now we give the main correctness theorem for $\mathscr{B}$. Again, let $\mathscr{T}_A$ and $\mathscr{X}_A$ denote the transaction names and object names of $\mathscr{A}$, respectively.

THEOREM 5.3.11. *Let β be a finite schedule of $\mathscr{B}$. Then there exists a schedule γ of $\mathscr{A}$ such that the following two conditions hold:*

(1) *$\gamma|T = \beta|T$ for each transaction name $T \in \mathscr{T}_A - la$ and*
(2) *$\gamma|Y = \beta|Y$ for each object name $Y \in \mathscr{X}_A$.*

PROOF. The proof is analogous to that for Theorem 4.3.7. We construct γ by removing from β all REQUEST_CREATE(T), CREATE(T), REQUEST_COMMIT(T, v), COMMIT(T), ABORT(T), REPORT_COMMIT(T, v), and REPORT_ABORT(T) events for all transaction names T in $co \cup acc_r \cup acc_w$. Clearly, the two conditions hold. It remains to show that γ is a schedule of $\mathscr{A}$; to do this, it suffices to show that γ projects to yield a schedule of each component of $\mathscr{A}$. It is easy to see that, for each nonaccess transaction name T of $\mathscr{A}$, $\gamma|T$ is a schedule of the transaction automaton for T, and for each object name $Y \in \mathscr{X}_A - \{X, Z\}$, $\gamma|Y$ is a schedule of the object automaton for Y. The sequence $\gamma|Z$ is exactly the sequence of CREATE(T) and REQUEST_COMMIT(T, v) actions in γ for $T \in la_{rec}$. Since reconfigure-LA's preserve transaction well-formedness and always return value v = "*OK*", $\gamma|Z$ is a schedule of the dummy object automaton. Also, as before, γ is a schedule of the serial scheduler.

It remains to show that $\gamma|X$ is a schedule of S_X. The proof is by induction on the length of β. The basis, when β is of length 0, is trivial. Suppose that $\beta = \beta'\pi$, where π is a single event and the result true for β'. The only case that is not immediate is the case where π is a REQUEST_COMMIT event for a transaction name in $la_r \cup la_w$, so suppose that π = REQUEST_COMMIT(T, v), where $T \in la_r \cup la_w$. The proof that the preconditions of π in S_X are satisfied as in Theorem 4.3.7, but this time using Lemma 5.3.1 in place of Lemma 4.3.1. Therefore, γ is a schedule of $\mathscr{A}$. □

5.4 System $\mathscr{C}$: The Reconfigurable Quorum Consensus Algorithm with Replicated Configurations.

It is possible to manage configurations in a centralized fashion, directly implementing the algorithm described by system $\mathscr{B}$ above. However, it is also interesting to consider replicating the configuration information. This may avoid the risk of the configuration storage becoming a bottleneck or a single point of vulnerability in the system. One way of doing this is using the fixed-configuration quorum consensus algorithm: define a fixed *metaconfiguration* that describes read quorums and write quorums of replicas of the configuration. A *read-configuration-coordinator* reads a read quorum of configuration replicas, and returns both the latest configuration and a *generation number*, which is analogous to a version number. A *write-configuration-coordinator* writes the new configuration to a write quorum of configuration replicas; it does this using the generation number obtained from a read-configuration-coordinator. Except for the need to manage the generation-number information, the LA's are the same in this algorithm as in $\mathscr{B}$. It is also possible to manage the configuration replicas using changing metaconfigurations, i.e., to reconfigure the configuration replicas! But then similar issues arise for the implementation of metaconfigurations.

All of this could continue for any number of steps. However, in order to avoid an infinite regress, we must stop at some point and use an implementation that does not require further configuration. Such a stopping point might be either a centralized implementation or a fixed-quorum algorithm. (Note that a centralized implementation is just a special case of using fixed quorums, where there is only one replica.)

There is another interesting alternative, however, in which there is a one-to-one correspondence between the replicas at the last two stages, and the same configurations are used for both stages. That is, at the last two stages, the same configurations are used to manage the data replicas and the copies of the configurations themselves. This means that the read-configuration-coordinators will need to read a set of replicas of the configuration objects that form a read-quorum, without first knowing what the current read-quorum is! It turns out that they can simply start reading configuration replicas and use the generation numbers and configurations stored in them to determine when a read-quorum for the current configuration has been read.

In this subsection, we will describe this strategy. For simplicity, we consider the special case of which there are only two kinds of replicas—of logical objects and configurations—and both are managed using the same configurations.

The system we define is called $\mathscr{C}$. We first define the type of $\mathscr{C}$, and then give the new automata that appear in $\mathscr{C}$ but not in $\mathscr{B}$. The new automata are the configuration replica objects and the read-configuration-coordinator and write-configuration-coordinator transaction automata. Also, the LA's are slightly modified from those of $\mathscr{B}$.

5.4.1 *System Type.* The type of system $\mathscr{C}$ is defined to be the same as that of $\mathscr{B}$, with the following modifications: The object name $\overline{X}$ is now replaced by

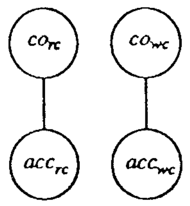

Fig. 13. Reconfiguration coordinators and their children.

a new set of object names, $\overline{\mathscr{Y}}$, representing the configuration replicas. We assume that there is a bijection i, from the names in $\overline{\mathscr{Y}}$ to those in $\mathscr{Y}$. There are new transaction names, acc_{rc} and acc_{wc}, which are the read and write accesses to the configuration replicas, respectively. We let $acc = acc_r \cup acc_w \cup acc_{rc} \cup acc_{wc}$. The transaction names in co_{rc} and co_{wc} are accesses in $\mathscr{B}$, but in $\mathscr{C}$ they are not; each transaction name in co_{rc} now has children in acc_{rc}, and each transaction name in co_{wc} has children in acc_{wc}. These are the only changes to the system type—for example, no transaction names other than those indicated are parents of the new transaction names. The naming structure for the read and write configuration coordinators and their children are shown in Figure 13.

In defining $\mathscr{B}$, we associated information with some of the transaction names. In $\mathscr{C}$, we keep all such information from $\mathscr{B}$ and add more information as follows. We now assume that every transaction name $T \in co_{wc}$ has associated values *old-config*$(T) \in C$ and *generation-number*$(T) \in N$, in addition to *data*(T) as before. Also, we assume that every transaction name $T \in acc_{rc}$ has *kind*(T) = *read*, and that every transaction name $T \in acc_{wc}$ has *kind*(T) = *write* and also *data*$(T) \in N \times C$.[7]

5.4.2 *Configuration Replica Automata.* We define a *configuration replica automaton* for each object name $Y \in \overline{\mathscr{Y}}$. This is a read/write serial object for Y with domain $N \times C$ and initial value $(0, c_0)$. For $v \in N \times C$, we write v.*generation-number* and v.*config* to refer to the components of v.

5.4.3 *Coordinators.* In this section, we define the coordinator automata in $\mathscr{C}$. The read- and write-coordinators are as in $\mathscr{B}$, so we need only define the read-configuration-coordinators and write-configuration-coordinators. We first define read-configuration-coordinators. The purpose of a read-configuration-coordinator is to determine the latest generation number and configuration, on the basis of the data returned by the read accesses it invokes on configuration replica objects.

A read-configuration-coordinator for transaction name $T \in co_{rc}$ has state components *awake*, *config*, *generation-number*, *requested*, *reported*, and *read*, where *awake* is a Boolean variable, initially *false*; *config* $\in C$, initially c_0; *generation number* $\in N$, initially 0; *requested* and *reported* are subsets of

[7]As with the transaction names in acc_w, since the transaction names in acc_{wc} name accesses to read/write objects, agreement with that definition compels the combination of the generation number and configuration into a single *data* attribute.

```
CREATE(T)
  Effect: $s.awake = true$
    $s.awake = true$

REQUEST_CREATE(T')
  Precondition:
    $s'.awake = true$
    $T' \notin s'.requested$
  Effect:
    $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT(T',v)
  Effect: $s.reported = s'.reported \cup \{T'\}$
    $s.read = s'.read \cup \{object(T')\}$
    if $v.generation\text{-}number$
        $> s'.generation\text{-}number$ then
          $s.generation\text{-}number$
            $= v.generation\text{-}number$
          $s.config = v.config$
    $s.reported = s'.reported \cup \{T'\}$
    $s.read = s'.read \cup \{object(T')\}$
    if $v.generation\text{-}number$
        $> s'.generation\text{-}number$ then
          $s.generation\text{-}number$
            $= v.generation\text{-}number$
          $s.config = v.config$

REPORT_ABORT(T')
  Effect: $s.reported = s'.reported \cup \{T'\}$
    $s.reported = s'.reported \cup \{T'\}$

REQUEST_COMMIT(T,v)
  Precondition:
    $s'.awake = true$
    $s'.requested = s'.reported$
    $(\exists \mathcal{R})(\iota(\mathcal{R}) \in s'.config.read$
        $\wedge \mathcal{R} \subseteq s'.read)$
    $v = (s'.generation\text{-}number, s'.config)$
  Effect:
    $s.awake = false$
```

Fig. 14. Transition relation for Read-Configuration-Coordinators.

children(T), initially empty; and *read* $\subseteq \overline{\mathscr{Y}}$, initially empty. The set of return values V_T is equal to $N \times C$.

The transition relation for read-configuration-coordinators is shown in Figure 14. A read-configuration-coordinator invokes accesses to configuration replicas. On receiving a REPORT_COMMIT, the coordinator compares the returned generation number with the *generation-number* component of its own state. If the returned generation number is larger, the coordinator updates its own *generation-number* and *configuration* components to the returned values. The interesting part of a read-configuration-coordinator is the set of preconditions for its REQUEST_COMMIT. When the coordinator reaches a state s' in which $s'.read$ contains a set of configuration replica names that corresponds (using the correspondence i between $\overline{\mathscr{Y}}$ and $\mathscr{Y}$) to a read quorum, according to $s'.config$, then it may request to commit, returning the highest generation number it has seen, along with the associated configuration. One should note the similarity of the read-configuration-coordinators and the read-coordinators in the way the most current configuration and generation number are obtained, and also the seemingly "circular" use of replicated configuration data in order keep track of the current configuration of the configuration replicas themselves.

Of course, it remains to show that this method of determining the current configuration is correct. The key intuition is that the write-configuration coordinators, defined next and as in system $\mathscr{B}$, write the new configuration and generation number to a write-quorum of the old configuration, which of

```
CREATE(T)
  Effect: $s.awake = true$
    $s\ awake = true$

REQUEST_CREATE(T')
  Precondition:
    $s'.awake = true$
    $T' \notin s'.requested$
    $data(T') = (generation\text{-}number(T), data(T))$
  Effect:
    $s.requested = s'.requested \cup \{T'\}$

REPORT_COMMIT(T',v)
  Effect: $s.reported = s'.reported \cup \{T'\}$
    $s.written = s'.written \cup \{object(T')\}$
    $s.reported = s'.reported \cup \{T'\}$
    $s.written = s'.written \cup \{object(T')\}$

REPORT_ABORT(T')
  Effect: $s.reported = s'.reported \cup \{T'\}$
    $s\ reported = s'.reported \cup \{T'\}$

REQUEST_COMMIT(T,v)
  Precondition:
    $s'.awake = true$
    $s'.requested = s'.reported$
    $(\exists \mathcal{W})(\imath(\mathcal{W}) \in old\text{-}config(T).write$
        $\wedge \mathcal{W} \subseteq s'.written)$
    $v = \text{"OK"}$
  Effect:
    $s.awake = false$
```

Fig. 15. Transition relation for Write-Configuration-Coordinators.

course will intersect all read-quorums of the old configuration. Hence, if a configuration replica's generation number is the largest found in reading a quorum according to that replica's configuration, then no such write has occurred and the given configuration is the current one.

As promised, we next define write-configuration-coordinators. A write-configuration-coordinator for transaction name $T \in co_{wc}$ has state components *awake*, *requested*, *reported*, and *written*, where *awake* is a Boolean variable, initially *false*; *requested* and *reported* are subsets of *children*(T), initially empty; and *written* is a subset of $\overline{\mathscr{Y}}$, initially empty. The set of return values V_T is equal to {"*OK*"}.

The transition relation for write-configuration-coordinators is shown in Figure 15. Recall that a write-configuration-coordinator T has an associated old configuration, *old-config*(T), and a new configuration, *data*(T), as well as a generation number, *generation-number*(T). The purpose of a write-configuration-coordinator is to write its generation number and new configuration to a write quorum in its old configuration. It does this by invoking write accesses to the configuration replicas, and can only request to commit after receiving REPORT_COMMIT actions for accesses to all configuration replicas in some set corresponding to a write quorum, according to *old-config*(T).

5.4.4 *Logical Access Automata.* A read-LA in $\mathscr{C}$ is identical to a read-LA in $\mathscr{B}$, except for the REPORT_COMMIT input action for children that are read-configuration-coordinators. The only difference is that the read-configuration-coordinator returns not only a configuration, but a pair consisting of a generation number and a configuration. The read-LA simply ignores the generation number. Formally, we have:

```
REPORT_COMMIT(T', v), $T' \in co_{rc}$
  Effect: $s.reported = s'.report \cup \{T'\}$
    if $s'.config\text{-}read = false$ then
      $s.config = v.config$
      $s.config\text{-}read = true$
```

```
$s.reported = s'.reported \cup \{T'\}$
if $s'.config\text{-}read = false$ then
  $s.config = v.config$
  $s.config\text{-}read = true$
```

A write-LA in $\mathscr{C}$ is identical to a write-LA in $\mathscr{B}$, except for the same change described above for read-LA's.

Unlike the read- and write-LA's, the reconfigure-LA's of $\mathscr{C}$ make use of the generation numbers returned by the read-configuration-coordinator, to manage the configuration replicas. Thus, the reconfigure-LA's of $\mathscr{C}$ differ more from those of $\mathscr{B}$ than do the read- and write-LA's. Here, a read-configuration-coordinator returns a pair consisting of a generation number and a configuration, both of which are saved by the reconfigure-LA. These are used to determine, for the write-configuration-coordinators T' invoked by the LA, the allowable values for *old-config*(T') and *generation-number*(T').

A reconfigure-LA for transaction name $T \in la_{rec}$ has the same state components as it does in $\mathscr{B}$, except for the addition of the component *generation-number*, which takes on values in $N \cup \{nil\}$ and has initial value *nil*. The signature is the same, except that the type of the return values for read-configuration-coordinators is now $N \times C$. The actions that change are:

```
REPORT_COMMIT($T', v$), $T' \in co_{rc}$
  Effect: $s.reported = s'.reported \cup \{T'\}$
    if $s'.config\text{-}read = false$ then
      $s.config = v.config$
      $s.generation\text{-}number$
        $= v.generation\text{-}number$
      $s.config\text{-}read = true$
    $s.reported = s'.reported \cup \{T'\}$
    if $s'.config\text{-}read = false$ then
      $s.config = v.config.$
      $s.generation\text{-}number$
        $= v.generation\text{-}number$
      $s.config\text{-}read = true$
```

```
REQUEST_CREATE($T'$), $T' \in co_{wc}$
  Precondition:
    $s'.awake = true$
    $s'.value\text{-}read = true$
    $data(T') = config(T)$
    $old\text{-}config(T') = s'.config$
    $generation\text{-}number(T')$
      $= s'.generation\text{-}number + 1$
    $T' \notin s'.requested$
  Effect:
    $s.requested = s'.requested \cup \{T'\}$
```

5.5 Correctness Proof

In this section, we prove the correctness of system $\mathscr{C}$. Once again, we begin with definitions, this time for a sequence β of actions of $\mathscr{C}$. Specifically, we extend the definitions of *logical-sequence*(β) and *logical-config*(β) to apply the sequences of actions of $\mathscr{C}$. We also require one new definition, analogous to *current-vn*(β). Namely, if β is finite, then *current-gn*(β) is defined to be the highest *generation-number* among the states of all configuration replica automata in any global state led to by β.

We prove correctness of $\mathscr{C}$ by means of a correspondence between $\mathscr{C}$ and $\mathscr{B}$. If β is a sequence of actions of $\mathscr{C}$, then we define $f(\beta)$ to be the sequence of actions of $\mathscr{B}$ that results from

(1) removing all REQUEST_CREATE(T), CREATE(T), REQUEST_COMMIT(T, v), COMMIT(T), ABORT(T), REPORT_COMMIT(T, v), and RE-

PORT_ABORT(T) events for all transaction names $T \in acc_{rc} \cup acc_{wc}$, and

(2) replacing all REQUEST_COMMIT(T, v) and REPORT_COMMIT(T, v) where $T \in co_{rc}$, with REQUEST_COMMIT(T, $v.config$) and REPORT_COMMIT(T, $v.config$), respectively.

The following lemma is the key to the proof of Theorem 5.6, the main correctness theorem for $\mathscr{C}$. As in Lemma 5.3.1, condition (1) is only needed for the inductive argument. Part (a) says that any configuration replica $S_{\overline{Y}}$ either has the current generation number, or has a configuration such that all configuration replicas in some write quorum of that configuration hold generation numbers higher than the one held by $S_{\overline{Y}}$. Part (b) says that every configuration replica holding the current generation number also holds the logical configuration. Condition (2), the important part of the lemma, says that our construction yields a schedule of $\mathscr{B}$.

LEMMA 5.5.1. *Let β be a finite schedule of $\mathscr{C}$ such that no logical access is active in β.*

(1) *The following properties hold in any global state led to by β:*

(a) *For any configuration replica automaton $S_{\overline{Y}}$, if v is the data component of $S_{\overline{Y}}$ and $v.generation\text{-}number < current\text{-}gn(\beta)$, then there exists a write quorum $\mathscr{W} \in v.config.write$ with the following property: for any configuration replica $S_{\overline{Y}}$ for an object name $\overline{X}$ such that $i(\overline{X}) \in \mathscr{W}$, if v' is the data component of $S_{\overline{Y}}$ then $v'.generation\text{-}number > v.generation\text{-}number$.*

(b) *For any configuration replica $S_{\overline{Y}}$, if v is the data component of $S_{\overline{Y}}$ and $v.generation\text{-}number = current\text{-}gn(\beta)$ then $v.config = logical\text{-}config(\beta)$.*

(2) *The sequence $f(\beta)$ is a schedule of $\mathscr{B}$.*

PROOF. We proceed by induction on the number of pairs of the form CREATE(T) REQUEST_COMMIT(T, v) in *logical-sequence*(β). For the basis, suppose *logical-sequence*(β) contains no such pairs. Then *logical-config*(β) $= c_0$. Since all configuration replicas initially have *generation-number* $= 0$ and *config* $= c_0$, the same is true in any global state led to by β. Thus, *current-gn*(β) $= 0$. This implies that condition (1), and condition (2) hold because the two systems behave identically if no LA's are invoked.

For the inductive step, let $\beta = \beta'\gamma$, where *logical-sequence*(γ) begins with the last CREATE event in *logical-sequence*(β), and assume that the lemma holds for β'. Then *logical-sequence*(γ) = CREATE(T_f) REQUEST_COMMIT(T_f, v_f) for some $T_f \in la$ and v_f of the appropriate type. As before, we can restrict attention to T_f and its descendants.

For any LA to issue a REQUEST_COMMIT, it must first receive a REPORT_COMMIT for a read-configuration-coordinator and a read-coordinator. Let T' be the read-coordinator in co_r with the first REPORT_COM-

MIT in γ, and let γ' be the portion of γ up to and including the given REPORT_COMMIT event. Also, let T'' be the read-configuration-coordinator in co_{rc} with the first REPORT_COMMIT in γ, and let γ'' be the portion of γ up to and including this REPORT_COMMIT event.

Claim 5.2.2. If REQUEST_COMMIT(T'', v) occurs in γ, then $v =$ (*current-gn*(β'), *logical-config*(β')).

PROOF. Let s be the state of the automaton associated with T'' when that automaton issues its REQUEST_COMMIT event. Using an argument similar to the proof of Claim 4.3.3, we can show that *s.generation-number* and *s.config* contain the highest *generation-number* and associated *config* among the configuration replicas whose names are in *s.read*. Suppose (for contradiction) that this highest *generation-number* is not equal to *current-gn*(β'); then part (a) of the inductive hypothesis implies that there is some set $\mathscr{W}$ with $i(\mathscr{W}) \in$ *s.config.write* such that all configuration replicas for object names in $\mathscr{W}$ have *generation-number* > *s.generation-number*. By the definition of a read-configuration-coordinator, *s.read* must contain a set $\mathscr{R}$ with $i(\mathscr{R}) \in$ *s.config.read*. Since *s.config* is a configuration, and i is one-to-one and onto, $i(\mathscr{R})$ and $i(\mathscr{W})$ have a nonempty intersection; therefore, some configuration replica for an object name in $\mathscr{R}$, and hence in *s.read*, lies in $\mathscr{W}$ and hence has its *generation-number* component strictly greater than *s.generation-number*. This is a contradiction to the claim that *s.generation-number* contains the highest generation number among the configuration replica automata whose names are in *s.read*. It follows that *s.generation-number* = *current-gn*(β'). Then it follows by part (b) of the inductive hypothesis that *s.config* = *logical-config*(β').

Now we consider three cases.

(1) $T_f \in la_r$. It is easy to see that condition (1) is preserved, since no descendants of T_f are write accesses to configuration replicas. Claim 5.5.2 implies that the value returned by T'' in γ has as its configuration component the value *logical-config*(β'). By the definition of *logical-config*, this is the same value that would be returned by the configuration automaton in $\mathscr{B}$. Also, the automaton associated with T_f in $\mathscr{C}$ behaves identically to that associated with T_f in $\mathscr{B}$ except that, in $\mathscr{C}$, it receives and ignores the *generation-number* component of the return value of T'', exactly as the correspondence f requires. Since β is a schedule of $\mathscr{C}$, it follows that $f(\beta)$ is a schedule of $\mathscr{B}$, which shows condition (2).

(2) $T_f \in la_w$. The argument is similar to the one for the previous case.

(3) $T_f \in la_{rec}$. We first give several claims about the state of T_f and the information associated with the descendants it invokes.

Claim 5.5.3. If s is a state of T_f in a global state reachable from $\beta'\gamma''$ in β, *s.generation-number* = *current-gn*(β') and *s.config* = *logical-config*(β').

PROOF. By Claim 5.5.2, since γ'' ends with the first REPORT_COMMIT of a read configuration coordinator. □

Claim 5.5.4. If T is a write-configuration-coordinator invoked by T_f in γ then $data(T) = config(T_f)$, $generation\text{-}number(T) = current\text{-}gn(\beta') + 1$, and $old\text{-}config(T) = logical\text{-}config(\beta')$.

PROOF. By Claim 5.5.3 and the definition of T_f. □

Claim 5.5.5. If T is a write access in acc_{wc} invoked by a write-configuration-coordinator in γ then $data(T) = (current\text{-}gn(\beta') + 1, config(T_f))$.

PROOF. By Claim 5.5.4 and the definition of a write-configuration coordinator. □

By definition, T_f cannot request to commit until at least one of its write-configuration-coordinators has committed; let T_w be any write-configuration-coordinator child of T_f that has a COMMIT event in γ. Moreover, at least one write access to a configuration replica must commit before a write-configuration-coordinator can request to commit. From Claim 5.5.5, we know that any write-configuration access T that is a descendant of T_f has $data(T) = (current\text{-}gn(\beta') + 1, config(T_f))$. Therefore, $current\text{-}gn(\beta) = current\text{-}gn(\beta') + 1$. Furthermore, by the definition of *logical-config*, $config(T_f) = logical\text{-}config(\beta)$. Therefore, we can conclude that all write accesses T in acc_{wc} invoked by a write-configuration-coordinator in γ have $data(T) = (current\text{-}gn(\beta), logical\text{-}config(\beta))$.

To show that part (a) of the inductive hypothesis holds for β, we consider three cases for each configuration replica $\overline{Y}$ with data v, in any global state led to by β. (By the preceding arguments, these cases are exhaustive.)

(1) $v.generation\text{-}number < current\text{-}gn(\beta')$. Then $\overline{Y}$ is not updated by any descendant of T_f in β. That is, v is the data in $\overline{Y}$ in any global state led to by β'. Therefore, it follows immediately from part (a) of the inductive hypothesis that there exists a set $\mathscr{W}$ with $i(\mathscr{W}) \in v.config.write$ such that for any configuration replica $S_{\overline{Y}}$, for an object name $\overline{X}' \in \mathscr{W}$, if v' is the *data* component of $S_{\overline{Y}}$, in any global state led to by β', then $v'.generation\text{-}number > v.generation\text{-}number$. During γ, either the *generation-number* in the data held in $S_{\overline{Y}}$, is unchanged, or else it is set to $current\text{-}generation(\beta)$ (which is greater than $v.generation\text{-}number$) by a descendant of T_f. In either case, part (a) holds for β.

(2) $v.generation\text{-}number = current\text{-}gn(\beta')$. Then by part (b) of the inductive hypothesis, we know that $v.config = logical\text{-}config(\beta')$. By definition of a write-configuration-coordinator, just prior to the REQUEST_COMMIT for T_w in γ, there must be a set $\mathscr{W}$ with $i(\mathscr{W}) \in old\text{-}config(T_w).write$ such that write accesses to all configuration replicas for object names in $\mathscr{W}$ have committed, so by our arguments above, all of these configuration replicas have $generation\text{-}number = current\text{-}gn(\beta)$ in any global state led to by β. By Claim 5.5.4, $old\text{-}config(T_w) = logical\text{-}config(\beta')$. Therefore, $\mathscr{W}$ satisfies $i(\mathscr{W}) \in v.config.write$, and all configuration replicas for object names in $\mathscr{W}$ have $generation\text{-}number = current\text{-}gn(\beta) > v.generation\text{-}number$ in any global state led to by β. Therefore, part (a) holds.

(3) $v.generation\text{-}number = current\text{-}gn(\beta)$. Then part (a) is trivial.

So in all three cases, part (a) holds for β.

We now show part (b). By the definition of *current-gn*, we know that if v' is the data component of the state of a configuration replica S_Y in any global state led to by β' then $v'.generation\text{-}number \le current\text{-}gn(\beta')$. Therefore, any configuration replica that has, in any global state led to by β, a generation number equal to $current\text{-}gn(\beta)$ (and so larger than $current\text{-}gn(\beta')$) must be written after β'. The writer of such a configuration replica must be a descendent of T_f, and therefore must be a child of some write configuration-coordinator $T_{w'}$ invoked by T_f. Since all such $T_{w'}$ have $data(T_{w'}) = config(T_f)$, which by definition is $logical\text{-}config(\beta)$, part (b) holds.

Condition (2) is straightforward. □

The lemma above immediately yields a relationship between $\mathscr{C}$ and $\mathscr{B}$. Let $\mathscr{T}_B$ and $\mathscr{X}_B$ denote the transaction names and object names of $\mathscr{B}$, respectively.

THEOREM 5.5.6. *Let β be a finite schedule of $\mathscr{C}$. Then there exists a schedule of γ of $\mathscr{B}$ such that the following two conditions hold:*

(1) *$\gamma|T = \beta|T$ for each transaction name $T \in \mathscr{T}_B - (co_{rc} \cup co_{wc})$ and*
(2) *$\gamma|Y = \beta|Y$ for each object name $Y \in \mathscr{X}_{\mathscr{B}}$.*

Now we can combine Theorems 5.3.11 and 5.5.6 to prove a relationship between $\mathscr{C}$ and $\mathscr{A}$. Let $\mathscr{T}_A$ and $\mathscr{X}_A$ denote the transaction names and object names of $\mathscr{A}$, respectively.

THEOREM 5.5.7. *Let β be a finite schedule of $\mathscr{C}$. Then there exists a schedule γ of $\mathscr{A}$ such that the following two conditions hold:*

(1) *$\gamma|T = \beta|T$ for each transaction name $T \in \mathscr{T}_A - la$ and*
(2) *$\gamma|Y = \beta|Y$ for each object name $Y \in \mathscr{X}_A$.*

6. CONCURRENT REPLICATED SYSTEMS

So far, this paper has dealt exclusively with serial systems. However, a useful nested transaction system must allow concurrency and the possibility of aborting a transaction after it has begun running. In order to simplify the programming effort, it is best for a system not to permit arbitrary concurrency, but rather to make it seem (to the programs) as if the system were serial. Because we have been discussing several different serial systems, we extend the concept of serial correctness given earlier to mention explicitly which serial system is considered to define correctness: if $\mathscr{S}$ is a serial system, we say that a sequence β of actions is *serially correct with respect to* $\mathscr{S}$ for transaction name T, provided that there is some behavior γ of $\mathscr{S}$ such that $\beta|T = \gamma|T$.

Many concurrency control mechanisms are known that enable a concurrent system to appear like a serial one. For example, in Fekete et al. [1990] a *generic system* is defined as a system containing transaction automata (just as in the serial system), generic objects that accept concurrent accesses and also receive information about the commit or abort of transactions, and a controller that passes information between them. In that paper several algorithms are given for constructing the generic objects from the serial objects of $\mathscr{S}$, in ways that ensure that every execution of the generic system is serially correct with respect to $\mathscr{S}$ for each nonorphan transaction.

We use $\mathscr{A}$ and $\mathscr{C}'$ to denote the systems defined in Section 5. A concurrent system may be constructed by applying the methods discussed in Fekete et al. [1990] to the system $\mathscr{C}$. This can be described as applying concurrency control to each copy separately. Thus, for example, we consider a transaction-processing system $\mathscr{D}$ that uses Moss' read-update locking algorithm [Moss 1981] to give a generic object for each object of $\mathscr{C}$. The results of Fekete et al. show that all behaviors of $\mathscr{D}$ are serially correct with respect to $\mathscr{C}'$, for all nonorphan transaction names, in particular, for all nonorphan transaction names in $\mathscr{T}_A - la$. That is, $\mathscr{D}$ looks like a serial replicated system. However, the goal of building a transaction-processing system is that both concurrency and replication should be transparent. That is, one wants a system to have behaviors that are serially correct with respect to $\mathscr{A}$, the serial unreplicated system. This is in fact the case for $\mathscr{D}$, a fact that follows from the above by the results of this paper.

To be precise, we have the following general result:

COROLLARY 6.1. *Let β be a sequence of actions that is serially correct with respect to $\mathscr{C}$, for transaction name $T \in \mathscr{T}_A - la$. Then β is also serially correct with respect to $\mathscr{A}$ for T.*

PROOF. By Theorem 5.5.7. □

Corollary 6.1 implies that all behaviors of $\mathscr{D}$ are serially correct with respect to $\mathscr{A}$ for all nonorphan transaction names in $\mathscr{T}_A - la$. This says that system $\mathscr{D}$, which combines concurrency control techniques based on locking with the replication strategies of this paper, looks the same as the serial, unreplicated system $\mathscr{A}$, to the nonorphan transaction names in $\mathscr{T}_A - la$ (in particular, to T_0).

Corollary 6.1 also shows that if a concurrent, replicated transaction-processing system is constructed by (1) using the reconfigurable quorum consensus protocol to manage the copies and (2) applying any of the concurrency control algorithms verified in Fekete et al. [1990] to each copy individually, then the concurrent replicated system appears serial and unreplicated.

Similar conclusions can be drawn for a system using the multiversion timestamp algorithms of Reed [1983] and Herlihy [1987], as modeled in Aspnes et al. [1988]. Also, similar conclusions can be drawn when $\mathscr{C}$ is replaced by system $\mathscr{B}$ of Section 5, or by $\mathscr{B}$ of Section 4. In general, any concurrency control algorithm that provides serial correctness at the level of the replicas may be combined with any of our replication algorithms to produce a correct system.

Finally, we note that our techniques allow combination of algorithms for orphan management (as described in Herlihy et al. [1987]) with algorithms for concurrency control and for replication. For example, consider the concurrent system $\mathscr{D}$ described just above, and let $\mathscr{E}$ be another system that is constructed from $\mathscr{D}$ by adding one of the orphan management algorithms. The results of Herlihy et al. [1987] imply that $\mathscr{E}$ is serially correct with respect to $\mathscr{C}$, for *all* transaction names (including orphans), in particular, for all transaction names in $\mathscr{T}_A - la$. Then Corollary 6.1 implies that $\mathscr{E}$ is serially correct with respect to $\mathscr{A}$ for all transaction names in $\mathscr{T}_A - la$. Thus, system $\mathscr{E}$, which combines concurrency control techniques and orphan management techniques with the replication strategies of this paper, looks like $\mathscr{A}$ to all the transaction names in $\mathscr{T}_A - la$, and in particular, to T_0.

7. CONCLUSION

We have presented a precise description and rigorous correctness proof for Gifford's data replication algorithm in the context of nested transactions and transaction failures. The algorithm was decomposed into simple modules that were arranged naturally in a tree structure. This use of nesting as a modeling tool enabled us to use standard assertional techniques to prove properties of transactions based upon the properties of their children.

Each module was described in terms of an automaton that made extensive use of nondeterminism. Although an actual implementation would not be nondeterministic, the nondeterminism adds a degree of generality to our proof. That is, the correctness proof holds for any implementation that further restricts the nondeterministic choices.

The modularity of the proof strategy permitted us to separate the concerns of replication from those of concurrency control and recovery. We could deal exclusively with serial systems in order to simplify our reasoning. The proof was accomplished hierarchically, showing that the fully replicated system simulated an intermediate system and that the intermediate system simulated an obviously correct unreplicated system. Then, to complete the proof, we presented a theorem stating that the combination of any correct concurrency control algorithm with the replication algorithm yields a correct system.

This work has identified a general framework for proving the correctness of data replication algorithms in nested transaction systems. One begins by constructing a formal description of the algorithm in terms of a nested transaction system built from I/O automata. Then, one uses the appropriate definitions to show that each logical read access returns the proper value. Next, one constructs a corresponding serial system without replication, and proves that the user transactions in that system have the same executions as the user automata in the replicated system. Finally, one proves separately the correctness of the concurrency control algorithm, and applies a result analogous to Corollary 6.1 to show that the combined system is correct.

It may be possible to use this general technique to add transaction nesting to other, more complicated, data replication schemes, and to prove the resulting algorithms correct. Such algorithms include the "Virtual Partition" approach of Abbadi and Toueg [1989], and Herlihy's "General Quorum Consensus" [1984]. An interesting question is whether the techniques presented here can be extended to accommodate these algorithms when transaction nesting is added. Several of these algorithms do not provide atomicity as we define it. Rather, they allow the serialization order to differ from the true order, between transactions that run in separate partitions of the network. Therefore, additional definitions and theory will be required before these algorithms can be verified using our techniques.

ACKNOWLEDGMENTS

We thank Alan Fekete, Michael Merritt, William Weihl, and the anonymous referees for their useful comments on this material and for their help in improving its presentation.

REFERENCES

ASPNES, J., FEKETE, A., LYNCH, N., MERRITT, M., AND WEIHL, W. 1988. A theory of timestamp-based concurrency control for nested transactions. In *Proceedings of the 14th International Conference on Very Large Data Bases* VLDB Endowment, 431–444.

BARBARA, D., AND GARCIA-MOLINA, H. 1985. Mutual exclusion in partitioned distributed systems. Tech. Rep. TR-346, Dept. Computer Science, Princeton Univ., Princeton, N.J

BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. 1987 *Concurrency Control and Recovery in Database Systems* Addison-Wesley, Reading, Mass.

EAGER, D., AND SEVCIK, K. 1983 Robustness in distributed database systems. *ACM Trans. Database Syst* 8, 3 (Sept.) 354–381.

EL ABADDI, A., TOUEG, S. 1989. Maintaining availability in partitioned replicated databases. *ACM Trans Database Syst. 14*, 2 (June) 264–290.

EL ABBADI, A., SKEEN, D., AND CRISTIAN, F. 1985 An efficient fault-tolerant protocol for replicated data management In *Proceedings of the 4th ACM Symposium on Principles of Database Systems* (Portland, Oregon, Mar.). ACM, New York, 215–229

FEKETE, A., LYNCH, N., MERRITT, M., AND WEIHL, W. 1990 Commutativity-based locking for nested transactions. *J Comput. Syst Sci.* (Aug.) 65–156.

FEKETE, A., LYNCH, N., MERRITT, M., AND WEIHL, W. 1987. Nested transactions and read/write locking. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems* (San Diego, CA, Mar.). ACM, New York, 97–111 Expanded version available as Tech. Memo MIT/LCS/TM-324, Laboratory for Computer Science, MIT, Cambridge, Mass., April

GIFFORD, D 1979. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating System Principles*. ACM, New York, 150–162.

HERLIHY, M. 1987. Extending multiversion time-stamping protocols to exploit type information *IEEE Trans Comput C-36*, 4 (Apr.).

HERLIHY, M. 1984. Replication methods for abstract data types. Tech. Rep. MIT/LCS/TR-319, MIT Laboratory for Computer Science, Cambridge, Mass., May.

HERLIHY, M., LYNCH, N., MERRITT, M., AND WEIHL, W. 1987. On the correctness of orphan elimination algorithms In *Proceedings of the 17th IEEE Symposium on Fault-Tolerant Computing* IEEE, New York, 8–13 Also, MIT/LCS/TM-329, MIT Laboratory for Computer Science, Cambridge, Mass., May. To appear in *J. ACM*.

JAJODIA, S., AND MUTCHLER, D. 1990 Dynamic voting algorithms for maintaining the consistency of replicated databases *ACM Trans. Database Syst. 15* (June), 230–280.

LYNCH, N., AND MERRITT, M. 1986. Introduction to the theory of nested transactions. *Theoret. Comput. Sci. 62,* 123–185. Also in *International Conference on Database Theory* (Rome, Italy, Sept.) 278–305. Expanded version in MIT/LCS/TR-367 July.

LYNCH, N., AND TUTTLE, M. 1987. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computation*. ACM, New York, 137–151 Expanded version available as Tech Rep. MIT/LCS/TR-387, Laboratory for Computer Science, MIT Cambridge, Mass., Apr

MOSS, J. E. B. 1981. Nested transactions: An approach to reliable distributed computing. Ph.D. thesis, MIT, Cambridge, Mass. Tech. Rep. MIT/LCS/TR-260, Laboratory for Computer Science, MIT, Apr. Also, published by MIT Press, Mar. 1985.

REED, D. 1983. Implementing atomic actions on decentralized data. 1983. *ACM Trans. Comput. Syst. 1*, 1 (Feb.) 3–23.

THOMAS, R. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June) 180–209.

Received August 1992; revised March 1993 and September 1993; accepted February 1994