Quorum Consensus in Nested Transaction Systems

Kenneth J. Goldman and Nancy A. Lynch

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract: Gifford's basic Quorum Consensus algorithm for data replication is generalized to accommodate nested transactions and transaction failures (aborts). A formal description of the generalized algorithm is presented using the new Lynch-Merritt input-output automaton model for nested transaction systems. This formal description is used to construct a complete (yet simple) proof of correctness that uses standard assertional techniques and is based on a natural correctness condition. The presentation and proof treat issues of data replication entirely separately from issues of concurrency control and recovery.

1 Introduction

In distributed database systems, logical data items are often *replicated* in order to improve availability, reliability and performance. Whenever replication is used, a *replication algorithm* is required in order to ensure that the replication is transparent to the user programs. In understanding replication algorithms, it is convenient to think of each logical data item as being implemented by a collection of data managers (DMs) and transaction managers (TMs). The DMs retain state information, and the collective state of the DMs defines the current state of the logical data item. The user programs invoke TMs in order to read or write the logical data item; the TMs accomplish this by physically accessing some subset of the DMs.

One of the most well-known replication algorithms is Gifford's algorithm [10], which we call *Quorum Consensus*. Based on Thomas [21], the ideas of this method underlie many of the more recent and sophisticated replication techniques (e.g., [1,2,8,12]). In Gifford's algorithm, each DM is assigned a certain number of *votes* and keeps as part of its state a data *value* with an associated *version number*. Each logical data item x has an associated *configuration* that consists of a pair of integers called *read-quorum* and *write-quorum*. If v is the total number of votes assigned to DMs for x, then the configuration is constrained so that read-quorum + write-quorum $> v$. To read x, a TM collects the version-numbers and values from enough DMs so that it has a read-quorum of votes; then it returns the value associated with the highest version-number. To write x, a TM first collects the version-numbers from enough DMs so that it has a read-quorum of votes; then, it writes its value with a higher version number to a collection of DMs with a write-quorum of votes. This method generalizes both the read-one/write-all and the read-majority/write-majority algorithms.

Here, we adopt a slightly more general configuration strategy, which is justified by Barbara and Garcia-Molina in [3]: A configuration consists of a set of read-quorums and a set of write-quorums. Each quorum is a set of DM names, and every read-quorum must have a non-empty intersection with every write-quorum. To read a data item, a TM accesses all the DMs in some read-quorum and chooses the value with the highest version number. To write a data item, a TM first discovers the highest version number written so far by accessing all the DMs in some read-quorum; then the TM increments that version number by one and writes the new value and version number to all

The full version of this paper is available as MIT Technical Report *MIT/LCS/TR-390*.

This work was supported in part by the Office of Naval Research under Contract N00014-85-K-0168, by the Office of Army Research under Contract DAAG29-84-K-0058, by the National Science Foundation under Grants DCR-83-02391 and CCR-8611442, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125. The first author was supported by an Office of Naval Research graduate fellowship.

the DMs in some write-quorum.

In this paper, we generalize Gifford's algorithm in two fundamental ways. First, we incorporate the concept of *transaction nesting* into the algorithm. Transaction nesting is useful in its own right (for instance, as the basis of the distributed programming language ARGUS [14,15,19,22]). In addition, it turns out that nested transactions provide a useful way of understanding replication algorithms even if user transactions are not nested (as in Gifford [10]). This is because the TM's themselves can be regarded as *subtransactions* of the user transactions. Once one sees how to understand the algorithm in this way, it is very natural to generalize the algorithm to allow nesting of user transactions as well. Second, we extend the algorithm to accommodate transaction failures (aborts). Thus, for example, an operation to access a logical data item can complete even if some of its accesses to DMs abort.

We present our algorithm using the new framework of Lynch and Merritt [16] for modelling nested transaction concurrency control and recovery. The descriptions are clear, simple, and unambiguous. A complete correctness proof is also described; it is short, natural, and intuitive, yet completely rigorous.

An important reason for the simplicity of the proof is the fact that we are able to separate the treatment of replication entirely from the treatment of concurrency control and recovery. That is, we are able to consider the replication issues solely in the context of serial systems. We prove that a system which includes the new replication algorithm and which is serial at the level of the individual data copies "simulates" (in a strong sense) a system which is serial at the level of the logical data items. In particular, it "looks the same" to the user transactions. Since both systems involved in this simulation are serial systems, the simulation proof is very simple, and is based on standard assertional techniques.

Of course, systems which 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 [19,20,16,9] has produced several interesting algorithms which guarantee that a system *appears to be serial*, as far as the transactions can tell. Combining any of these algorithms (at the copy level) with the new replication algorithm yields a combined algorithm which appears to be nonreplicated and serial (at the logical data item level), as far as the user transactions can tell.

In fact, our results show that the replication algorithm can be combined with *any* algorithm which guarantees serializability at the copy level, to yield a system which is serializable at the logical item level. Thus, our work formalizes a frequently stated informal claim that "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" [7].

Related work, in addition to the papers already mentioned, includes some previous attempts at rigorous presentation and proof of replicated data algorithms. Most notable among these is the presentation and proof given by Bernstein, Hadzilacos, and Goodman [7] of Gifford's basic algorithm. This work is based on *serializability theory*, a theory which has made a significant contribution to the understanding of concurrency control. This approach, however, does not appear to generalize easily to the case where nesting and failures are allowed. Also, Herlihy [12] extends Gifford's algorithm to accommodate abstract data types and offers a correctness proof. Again, nesting is not considered. This paper is part of a larger effort to unify the work in concurrency control and recovery, as well as extend it to permit nesting [16,9,13].

The remainder of the paper is organized as follows. In Section 2, we introduce the computation model. Then, in Section 3, we describe the generalized version of Gifford's algorithm with fixed configurations and prove its correctness. Then we show that the correctness of interesting non-serial replicated systems follows directly from these results. In Section 4, we describe how these methods are expanded to give a correctness proof when we permit configurations to be changed dynamically (reconfiguration). Section 5 contains a summary of our results and a brief discussion of possible further research.

For the algorithm without reconfiguration, we present the statements of all the lemmas and theorems, and describe the highlights of the more interesting proofs. For complete detailed proofs, including reconfiguration, see [11].

2 The Model

We use the I/O automaton model, due to Lynch-Merritt [16] and Lynch-Tuttle [18], as the formal foundation for our work. We model components of a system with (possibly infinite-state) nondeterministic automata that have operation names associated with their state transitions. Communication among automata is described by identifying their operations. We only prove properties of finite behavior, so a simple special case of the general model is sufficient. Sections 2.1 and 2.2 provide a brief introduction to I/O automata and systems that includes the definitions from [16] and [18] that are relevant to this work. Then, in Section 2.3, we extend the model with some new definitions that are particularly useful for modeling replicated data management algorithms.

2.1 I/O Automata and Systems

The basic components of the model are *I/O automata.* An I/O automaton $\mathcal{A}$ has components *states*($\mathcal{A}$), *start*($\mathcal{A}$), *out*($\mathcal{A}$), *in*($\mathcal{A}$), and *steps*($\mathcal{A}$). Here, *states*($\mathcal{A}$) is a set of states, of which a subset *start*($\mathcal{A}$) is designated as the set of start states. The next two components are disjoint sets: *out*($\mathcal{A}$) is the set of *output operations*, and *in*($\mathcal{A}$) is the set of *input operations.* The union of these two sets is the set of *operations* of the automaton. Finally, *steps*($\mathcal{A}$) is the transition relation of $\mathcal{A}$, which is a set of triples of the form (s', π, s), where s' and s are states, and π is an operation. This triple means that in state s', the automaton can atomically perform operation π and change to state s. An element of the transition relation is called a *step* of $\mathcal{A}$. If (s', π, s) is a step of $\mathcal{A}$, we say that π is *enabled* in s'.

The output operations are intended to model the actions that are triggered by the automaton itself, while the input operations model the actions that are triggered by the environment of the automaton. We require the following condition, which says that an I/O automaton must be prepared to receive any input operation at any time.

Input Condition: For each input operation π and each state s', there exist a state s and a step (s', π, s).

An *execution* of $\mathcal{A}$ is a finite alternating sequence $s_0, \pi_1, s_1, \pi_2, \ldots, s_n$ of states and operations of $\mathcal{A}$, where s_0 is in *start*($\mathcal{A}$) and each subsequence $(s_i, \pi_{i+1}, s_{i+1})$ is in *steps*($\mathcal{A}$). From any execution, we can extract the *schedule*, which is the subsequence of the execution that contains only the operations (e.g., $\pi_1, \pi_2, \ldots, \pi_n$). Because transitions to different states may be labeled with the same operation, different executions may have the same schedule.

If S is any set of schedules (or *property* of schedules), then automaton $\mathcal{A}$ is said to *preserve* S provided that the following holds. If $\alpha = \alpha'\pi$ is any schedule of $\mathcal{A}$, where π is an output operation and α' is in S, then α is in S. That is, $\mathcal{A}$ is not the first to violate the property described by S.

We model a *system* as a set of interacting components, each of which is an I/O automaton. It is convenient and natural to view systems as I/O automata as well. Thus, we define an operation that composes a set of I/O automata to yield a new I/O automaton.

A set of I/O automata may be composed to create a *system* S, provided that the sets of output operations for the automata are disjoint. This ensures that every output operation in S will be triggered by exactly one component. The system S is itself an I/O automaton. A state of the composed automaton is a tuple of states, one for each component, and the start states are tuples consisting of start states of the components. The set of *operations* of S, *ops*(S), is the union of the sets of operations of the component automata. The set of *output operations* of S, *out*(S), is likewise the union of the sets of output operations of the component automata. Finally, the set of *input operations* of S, *in*(S), is *ops*(S)$-$*out*(S), the set of operations of S that are not output operations of S. The output operations of a system are intended to be exactly those that are triggered by components of the system, while the input operations of a system are those that are triggered by the system's environment.

The triple (s', π, s) is in the transition relation of S if and only if for each component automaton $\mathcal{A}$, one of the following two conditions holds. Either π is an operation of $\mathcal{A}$, and the projection of the step onto $\mathcal{A}$ is a step of $\mathcal{A}$, or else π is not an operation of $\mathcal{A}$, and the state corresponding to $\mathcal{A}$ in tuple s' is identical to the state corresponding to $\mathcal{A}$ in tuple s. Thus, each operation of the composed automaton is an operation of a subset of the component automata. During an operation π of S, each of the components which has operation π carries out the operation, while the remainder stay in the same state. Again, the operation π is an output operation of the composition if it is the output operation of a component — otherwise, π is an input operation of the composition.

An *execution* of a system is defined to be an execution of the automaton composed of the individual automata of the system. If σ is a sequence of operations of a system S with component $\mathcal{A}$, then $\sigma|\mathcal{A}$ (read "σ restricted to $\mathcal{A}$") is the subsequence of σ consisting of the operations of $\mathcal{A}$. Clearly, if σ is a schedule of S, then $\sigma|\mathcal{A}$ is a schedule of $\mathcal{A}$.

The following lemma, known as the Composition Lemma, expresses formally the notion that an operation is under the control of the component of which it is an output.

Lemma 1 Let σ' be a schedule of a system S, and let $\sigma = \sigma'\pi$, where π is an output operation of component $\mathcal{A}$. If $\sigma|\mathcal{A}$ is a schedule of $\mathcal{A}$, then σ is a schedule of S.

Proof: In [16]. ■

Let σ be a schedule of system S. We say that property P *holds after* σ iff property P holds for the final state of every execution of S whose schedule is σ. We say that property P *holds forever after* σ iff property P holds for the final state of every execution of S whose schedule has σ as a prefix.

Let $\mathcal{A}$ be an automaton whose transition relation is restricted so that if (s', π, s_1) and (s', π, s_2) are both in steps($\mathcal{A}$), then $s_1 = s_2$. If $\mathcal{A}$ has a unique initial state, then we say that $\mathcal{A}$ is a ***state-deterministic*** automaton. That is, $\mathcal{A}$ is deterministic in the sense that its state is a function of its schedule.

All of the automata that we define explicitly are state-deterministic. For such automata, we will freely use the words "*state s of $\mathcal{A}$ after schedule σ*" to denote the unique state of $\mathcal{A}$ resulting from the execution of $\mathcal{A}$ whose schedule is σ.

2.2 Nested Transaction Systems

To model nested transaction systems we use a ***system type***, which is a tuple ($\mathcal{T}$,parent,$\mathcal{O}$,V). $\mathcal{T}$ is the set of transaction names organized into a tree by the mapping parent:$\mathcal{T} \rightarrow \mathcal{T}$, with T_0 as the root. In referring to this tree, we use traditional terminology, such as child, leaf, least common ancestor (lca), ancestor and descendant. (A transaction is its own ancestor and descendant.) The leaves of $\mathcal{T}$ are called ***accesses***. The set $\mathcal{O}$ is a partition of the set of accesses, where each element (class) of the partition contains the accesses to a particular object; each element of $\mathcal{O}$ denotes its corresponding object. Finally, V is the set of ***values*** that may be returned by transactions. The tree structure is know in advance by all the components of the system and can be thought of as a predefined naming scheme for all possible transactions that might ever be invoked. In general, the tree is an infinite structure, and only some of the transactions will take steps in any given execution.

The root transaction T_0 plays a special role in this theory. The root models the environment of the nested transaction system (the "external world") from which requests for transactions originate and to which the results of these transactions are reported. Since it has no parent, T_0 may neither commit nor abort. The classical transactions of concurrency control theory (without nesting) appear in our model as the children of T_0. (In other work on nested transactions, such as Argus, the children of T_0 are often called "top-level" transactions.) Even in the context of classical theory (with no additional nesting) it is convenient to introduce the root transaction to model the environment in which the rest of the transaction system runs, with operations that describe the invocation and return of the classical transactions. It is natural to reason about T_0 in the same way as about all of the other transactions.

The internal nodes of the tree model transactions whose function is to create and manage subtransactions, but not to access data directly. The only transactions which actually access data are the leaves of the transaction tree, and thus they are called "accesses". The partition $\mathcal{O}$ simply identifies those transactions which access the same object.

The systems we describe are ***serial systems***. A serial system is the composition of a set of I/O automata. This set contains a ***transaction*** for each internal node of the transaction tree, a ***basic object*** for each element of $\mathcal{O}$, and a ***ser'al scheduler*** for the given system type. The system ***primitives*** are the transaction automata and the basic objects; these describe user programs and data, respectively. The serial scheduler controls communication between the primitives, and thereby defines the allowable orders in which the primitives may take steps. All three types of system components are modelled as I/O automata. These automata are described below. (If X is a basic object associated with an element $\mathcal{X}$ of the partition $\mathcal{O}$, and T is an access in $\mathcal{X}$, we write T∈*accesses*(X) and say that "T is an access to X".)

Non-access Transactions: Transactions are modelled as I/O automata. In modelling transactions, we consider it very important not to constrain them unnecessarily; thus, we do not want to require that they be expressible as programs in any particular high-level programming language. Modelling the transactions as I/O automata allows us to state exactly the properties that are needed, without introducing unnecessary restrictions or complicated semantics. A non-access transaction T is modelled as an I/O automaton, with the following operations:

Input Operations:
CREATE(T)
COMMIT(T',v), where T'∈children(T) and $v \in V$
ABORT(T'), where T'∈children(T)
Output Operations:
REQUEST-CREATE(T'), where T'∈children(T)
REQUEST-COMMIT(T,v), where $v \in V$

The CREATE input operation "wakes up" the transaction. The REQUEST-CREATE output operation is a request by T to create a particular child transaction [1]. The COMMIT input operation reports to T the successful completion of one of its children, and returns a value recording the results of that child's execution. The ABORT input operation reports to T the

[1] Note that there is no provision for T to pass information to its child in this request. In a programming language, T might be permitted to pass parameter values to a subtransaction. Although this may be a convenient descriptive aid, it is not necessary to include in it the underlying formal model. Instead, we consider transactions that have different input parameters to be different transactions.

unsuccessful completion of one of its children, without returning any other information. We call COMMIT(T',v), for any v, and ABORT(T') *return* operations for transaction T'. The REQUEST-COMMIT operation is an announcement by T that it has finished its work, and includes a value for reporting the results of that work to its parent.

It is convenient to use two separate operations, REQUEST-CREATE and CREATE, to describe what takes place when a subtransaction is activated. The REQUEST-CREATE is an operation of the transaction's parent, while the actual CREATE takes place at the subtransaction itself. In actual distributed systems such as Argus [15], this separation does occur, and the distinction will be important in our results and proofs. Similar remarks hold for the REQUEST-COMMIT and COMMIT operations, which occur at at transaction and its parent, respectively.

We leave the executions of particular transaction automata largely unspecified; the choice of which children to create, and what value to return, will depend on the particular implementation. However, it is convenient to assume that schedules of transaction automata obey certain syntactic constraints. Thus, transaction automata are required to preserve well-formedness, as defined below.

We recursively define *well-formedness* for sequences of operations of transaction T. Namely, the empty schedule is well-formed. Also, if $\alpha = \alpha'\pi$ is a sequence of operations of T, where π is a single operation, then α is well-formed provided that α' is well-formed, and the following hold:

If π is CREATE(T), then
  (i) there is no CREATE(T) in α'.
If π is COMMIT(T',v) or ABORT(T') for a child T' of T,
  (i) REQUEST-CREATE(T') appears in α' and
  (ii) there is no return operation for T' in α'.
If π is REQUEST-CREATE(T') for a child T' of T, then
  (i) there is no REQUEST-CREATE(T') in α'
  (ii) there is no REQUEST-COMMIT for T in α' and
  (iii) CREATE(T) appears in α'.
If π is a REQUEST-COMMIT for T, then
  (i) there is no REQUEST-COMMIT for T in α' and
  (ii) CREATE(T) appears in α'.

These restrictions are very basic; they simply say that a transaction is created at most once, does not receive repeated (or conflicting) notification of the fates of its children, and does not receive information about the fate of any child whose creation it has not requested. Also, a transaction performs output operations neither before it is created nor after it has requested to commit, and a transaction does not request the creation of any given child more than once.

Except for these minimal conditions, there are no restrictions on allowable transaction behavior. For example, the model allows a transaction to request to commit without discovering the fate of all subtransactions whose creation it has requested. Also, a transaction can request creation of new subtransactions at any time, without regard to its state of knowledge about subtransactions whose creation it has previously requested. Particular programming languages may choose to impose additional restrictions on transaction behavior. (An example is Argus, which suspends activity in transactions until subtransactions complete.) However, our results do not require such restrictions.

Basic Objects: Since access transactions model abstract operations on shared data objects, we associate a single I/O automaton with each object, rather than one with each access. The operations of a *basic object* automaton X are the invocation and return operations of the its access transactions:

Input Operations:
  CREATE(T), for T $\in$ accesses(X)
Output Operations:
  REQUEST-COMMIT(T,v), for T$\in$accesses(X), v$\in V$

Let α be a sequence of operations of basic object X. Then an access T to X is said to be *pending* in α provided that there is a CREATE(T) but no REQUEST-COMMIT for T in α.

It is convenient to require that schedules of basic objects satisfy certain syntactic conditions. Thus, each basic object is required to preserve well-formedness, which is defined recursively as follows.

The empty schedule is well-formed. If $\alpha = \alpha'\pi$ is a sequence of operations of basic object X, where π is a single operation, then α is well-formed provided that α' is well-formed, and the following conditions hold.

If π is CREATE(T) then
  (i) there is no CREATE(T) in α', and
  (ii) there are no pending accesses in α'.
If π is a REQUEST-COMMIT for T then
  (i) there is no REQUEST-COMMIT for T in α', and
  (ii) CREATE(T) appears in α'.

That is, the schedules of basic objects are restricted to consist of alternating CREATE and REQUEST-COMMIT operations, starting with a CREATE, and with each (CREATE, REQUEST-COMMIT) pair having the same access transaction, where each access transaction has at most one CREATE.

Serial Scheduler: The serial scheduler is a fully specified automaton. The serial scheduler can choose nondeterministically to abort any transaction T after parent(T) has issued a REQUEST-CREATE(T) operation, as long as T has not actually been created. Thus, the "semantics" of an abort(T) operation are that T was never created. Furthermore, a transaction can only be created if (1) it has not already been created, (2) its parent has requested its creation, and (3) all of its created siblings have returned. In other words, the scheduler runs transactions according to a depth-first traversal of the transaction tree.

Finally, the scheduler cannot commit a transaction until all of the transaction's children have returned. The formal definition of the serial scheduler, adapted from [16,9], is as follows.

The state of the serial scheduler has components create-requested, created, commit-requested, committed, aborted, and returned. Commit-requested is a set of (transaction,value) pairs, and the rest are sets of transaction names. Initially, create-requested = $\{T_0\}$, and the other sets are empty.

The steps of the transition relation for each automaton we define are exactly those triples (s', π, s) satisfying the pre- and postconditions listed, where π is the indicated operation. If a component of s is not mentioned in the postcondition, then it is taken to be the same in s as in s'.

Input Operations:
REQUEST-CREATE(T)
REQUEST-COMMIT(T,v)
Output Operations:
CREATE(T)
COMMIT(T,v)
ABORT(T)

•REQUEST-CREATE(T)
Postconditions:
create-requested(s) = create-requested(s') ∪ {T}

•REQUEST-COMMIT(T,v)
Postconditions:
commit-requested(s)=commit-requested(s')∪{(T,v)}

•CREATE(T)
Preconditions:
T ∈ create-requested(s') − (created(s') ∪ aborted(s'))
siblings(T) ∩ created(s') ⊆ returned(s')
Postconditions:
created(s) = created(s') ∪ {T}

•COMMIT(T,v)
Preconditions:
(T,v) ∈ commit-requested(s')
T ∉ returned(s')
children(T) ∩ create-requested(s') ⊆ returned(s')
Postconditions:
committed(s) = committed(s') ∪ {(T,v)}
returned(s) = returned(s') ∪ {T}

•ABORT(T)
Preconditions:
T ∈ create-requested(s') − (created(s') ∪ aborted(s'))
siblings(T) ∩ created(s') ⊆ returned(s')
Postconditions:
aborted(s) = aborted(s') ∪ {T}
returned(s) = returned(s') ∪ {T}

Let S be a serial system, and let σ be a sequence of operations of S. We say that σ is *well-formed* iff its projection at every primitive is well-formed. If σ is a schedule of S, then σ is a *serial schedule*. In [16], it is shown that all serial schedules are well-formed.

Let S be a serial system, and let γ be an arbitrary sequence of operations. We say that γ *is serially correct with respect to S for transaction* T provided that $\gamma|T = \sigma|T$ for some schedule σ of S.

2.3 Model Extensions for Replicated Data Systems

In this section, we add to the model some definitions that are useful for formalizing and understanding replicated data management algorithms.

In order to understand why these particular definitions are useful, it is helpful to keep in mind the general proof strategy we use. As explained in Section 1, for each algorithm considered we first construct a serial system in which database items are implemented as multiple replicas, where access to the replicas is controlled by the replication algorithm. Then, we construct a serial system with the same user transactions in which each database item is implemented as a single replica. Finally, we prove that each user transaction[2] in the replicated system has the same execution as its corresponding transaction in the non-replicated system.

We have already discussed serial systems and provided formal definitions for transactions, accesses, and executions. However, in order to give a more precise meaning to the above description of our proof strategy, we need formal definitions for "database item," "replica," and "corresponding transaction."

Logical Data Items: We refer to database items as "logical data items" to distinguish them from their physical counterparts, the replicas.

[2] For each system, we will define formally what is meant by a user transaction in terms of the system type. In general, however, one may think of user transactions as all the non-access transactions that do not model part of the replication algorithm. As a rule, user transactions are those transactions which we do not describe with fully-specified automata.

A *logical data item* x is a variable, whose type is the tuple $\langle V_x, i_x \rangle$. The set V_x is the domain of possible values for x, and $i_x \in V_x$ is the initial value of x. We require that a special undefined value, *nil*, be an element of V_x.

Read-write Objects: Each replica is modelled as a fully specified basic object called a *read-write object*, where the domain and initial value depend upon the particular data replication management algorithm and the type of the logical data item.

The state of a read-write object O with domain D with initial value $d \in D$ consists of two components, active and data. The variable active (initially nil) holds the name of the current access to O. Data holds an element of D (initially d). Every read-write object has a set of accesses, denoted *accesses*(O). Each access T to a read-write object has the attributes *kind*(T) $\in$ {read,write} and *data*(T) $\in$ D. When kind(T) = write, data(T) is the data to be written.

Input Operations:
CREATE(T), where T $\in$ accesses(O)
Output Operations:
REQUEST-COMMIT(T,v), where T $\in$ accesses(O)

•CREATE(T), for T $\in$ accesses(O)
Postconditions:
active(s) = T
•REQUEST-COMMIT(T,v), where kind(T) = read
Preconditions:
active(s') = T
v = data(s')
Postconditions:
active(s) = nil
•REQUEST-COMMIT(T,v), where
kind(T) = write and data(T) = d
Preconditions:
active(s') = T
v = nil
Postconditions:
data(s) = d
active(s) = nil

A read-write object accepts read and write accesses. For read accesses, it returns the value in the data component of its state. For write accesses, it records the new data value.

If T$\in$accesses(O), we say that O(T)=O. That is, we use O(T) to denote the read-write object to which T is an access.

Lemma 2 Read-write objects are basic objects.

Extensions of Systems: We want to define formally the notion of "corresponding transactions" so that we can be precise in our comparisons of each pair of replicated and non-replicated systems. In order to do so, we must impose certain restrictions on the system types of the two systems.

Let S' and S be two systems with system types Σ' and Σ, respectively. System type Σ' *is an extension of* system type Σ if the transaction tree of Σ is a subgraph of the transaction tree of Σ' and both trees have the same root. If Σ' is an extension of Σ, then we say that system S' *is an extension of* system S.

If system S' is an extension of system S, relating the transactions in the two systems is easy. We define function $\mathcal{F}_{SS'} : \mathcal{T}_S \rightarrow \mathcal{T}_{S'}$ to map transactions in S to their same-named transactions in S'. The inverse, $\mathcal{F}_{S'S}$, is a partial function unless S and S' have the same transaction tree.

Configurations: As a final addition to the model, we introduce the following general definitions, which are central to the algorithms we study.

Let S be any arbitrary set, and let Q be the power set 2^S. We define *configurations*(S) to be the set of all pairs of the form $\langle r, w \rangle$, where $r, w \subseteq Q$. (We sometimes refer to r and w as sets of read-quorums and write-quorums, respectively.) The set *legal*(S) is defined to be the set of all elements $\langle r, w \rangle$ of configurations(S) such that every element of r has a non-empty intersection with every element of w.

We say that every element of configurations(S) is a *configuration* of S, and that every element of legal(S) is a *legal configuration* of S.

Notation: We let N denote the set of non-negative integers (i.e., {0,1,2,...}).

3 Fixed Quorum Consensus

In this section, we formalize and prove the correctness of a generalized version of Gifford's algorithm without reconfiguration, as described in the introduction. In Section 3.1, we define system B, a *replicated serial system* that uses the fixed quorum consensus algorithm to manage replicas, and prove some properties of its schedules. Then, in Section 3.2, we define a corresponding *non-replicated serial system*, named system A. We prove the correctness of the fixed quorum consensus algorithm in Section 3.3 by showing that system B simulates system A in a strong sense. Finally, in Section 3.4, we show that non-serial replicated systems are correct.

3.1 Replicated Serial System

The replicated serial system defined in this section is an ordinary serial system in which certain logical data items are replicated. That is, they are implemented as several basic objects (replicas), rather than just one. We impose a restriction on the transaction tree so that all accesses to the replicas are the children of transaction manager automata (TMs), which we define explicitly. The TMs model the Quorum Consensus algorithm itself. We model the read and write operations of the algorithm by providing two kinds of TMs, read-TMs and write-TMs. We place no restrictions on the remaining automata, except that they preserve well-formedness. The system is formally defined as follows.

Fix $\mathcal{I}$, a set of logical data items. We define system B to be a serial system of type $\langle \mathcal{T}, \text{parent}, \mathcal{O}, \mathcal{V} \rangle$. With each element x of $\mathcal{I}$, we associate:

- $dm(x)$, a subset of $\mathcal{O}$
- $acc(x)$, a subset of the accesses in $\mathcal{T}$
- $tm_r(x)$ and $tm_w(x)$, disjoint subsets of the non-accesses in $\mathcal{T}$, and
- $config(x)$, a legal configuration of $dm(x)$.

Let $tm(x) = tm_r(x) \cup tm_w(x)$. We require that $acc(x)$ is exactly the set of all accesses to objects in $dm(x)$. In our replicated serial system, the replicas for x will be associated with the members of $dm(x)$, and the logical accesses to x will be managed by automata associated with the members of $tm(x)$. Since we want all accesses to replicas for x to be controlled by the replication algorithm, we require that T$\in acc(x)$ iff parent(T)$\in tm(x)$. Finally, for all pairs $x, y \in \mathcal{I}$, we require that $dm(x) \cap dm(y) = \emptyset$.

We define the *user transactions* in B to be the set of non-access transactions in $\mathcal{T}$ that are not in $tm(x)$ for all $x \in \mathcal{I}$. We refer to accesses in $acc(x)$ for all $x \in \mathcal{I}$ as *replica accesses*, and to the remaining accesses in $\mathcal{T}$ as *non-replica accesses*.

Figure 1 provides an example of a possible transaction tree for system B.

In system B, each member of $dm(x)$ has a corresponding data manager automaton (DM) for x, each member of $tm_r(x)$ has an associated read-TM automaton for x, and each member of $tm_w(x)$ has an associated write-TM automaton for x. From the restrictions on the system type, then, the members of $acc(x)$ are the accesses to the DMs for x. Furthermore, the accesses to DMs for x are exactly the children of the TMs for x. DMs and TMs for x are described below.

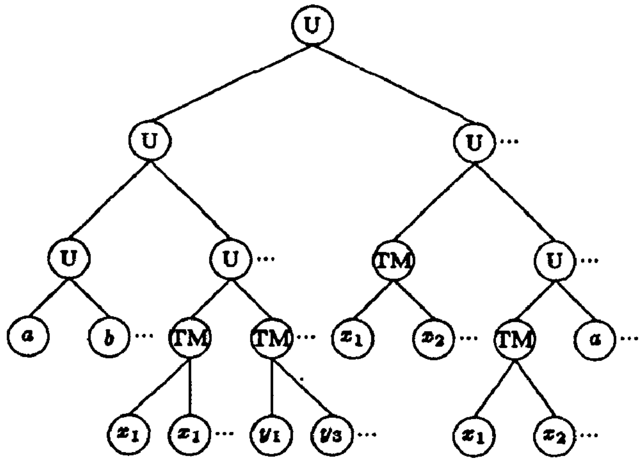

Figure 1: A possible transaction tree for system B. Transactions are labeled as follows: U = user transaction; TM = transaction manager; a, b = non-replica accesses; x_1 = accesses to replica 1 of logical data item x, etc.

Data Managers: The set of data managers for logical data item x models the set of physical replicas of x. Each DM is a read-write object that keeps a version-number and a value for x. The formal definition follows.

If x is a logical data item, a *DM* for x is a read-write object over domain $D_x = N \times V_x$ with initial data $\langle 0, i_x \rangle$. We refer to each member of D_x as a $\langle$version-number,value$\rangle$ pair. (For v$\in D_x$, we use the record notation v.version-number and v.value to refer to the components of v.)

Lemma 3 DMs are basic objects.

Recall that we have restricted the system type of B so that accesses to DMs for x are invoked only by TMs for x. We now define read-TMs and write-TMs for x.

Read TMs: Let x be a logical data item in $\mathcal{I}$. The purpose of a read-TM for x is to perform a logical read access to x. A read-TM for x invokes read accesses to multiple DMs for x. It then returns the "current" value of x, which it calculates from the information returned by the read accesses. In Lemma 8, we show that read-TMs in system B do, in fact, return the proper value of x. That is, a read-TM returns the value that would be expected, given the sequence of logical write accesses to x that precedes its invocation.

A read-TM T for x has state components awake, data, requested, and read, where awake is a boolean value, data is a value in the domain D_x, requested is a subset of $acc(x)$, and read is a subset of $dm(x)$.

Initially, data is $\langle 0, i_x \rangle$, awake is false, and requested and read are both empty.

Note: Whenever an undefined variable (for example, q in the REQUEST-COMMIT operation of the following automaton) appears in the pre- and/or postconditions for an operation, then that variable has an implicit existential quantifier (i.e., there exists a q such that...).

```
Input Operations:
  CREATE(T)
  COMMIT(T',v), where T'∈children(T) and v∈$D_x$
  ABORT(T'), where T' ∈ children(T)
Output Operations:
  REQUEST-CREATE(T'), where T' ∈ children(T)
  REQUEST-COMMIT(T,v), where v ∈ $D_x$
```

```
•CREATE(T)
 Postconditions:
  awake(s) = true
•REQUEST-CREATE(T'), where kind(T') = read
 Preconditions:
  awake(s') = true
  T' ∉ requested(s')
 Postconditions:
  requested(s) = requested(s') ∪ {T'}
•COMMIT(T',d)
 Postconditions:
  read(s) = read(s') ∪ {O(T')}
  if d.version-number>data(s').version-number,
   data(s) = d
•ABORT(T')
 Postconditions:
  (no change)
•REQUEST-COMMIT(T,v)
 Preconditions.
  awake(s') = true
  $q \in config(x).r$
  $q \subseteq$ read(s')
  v = data(s').value
 Postconditions:
  awake(s) = false
```

A read-TM collects data from some number of DMs for x, always keeping the data from the DM with the highest version number seen so far. When a read-quorum of DMs has been seen, the read-TM may request to commit and return its data.

It is interesting to note the extensive use of nondeterminism in this algorithm. For example, the read-TM does not set out to access any particular read-quorum in the configuration. Rather, the read-TM simply invokes any number of accesses to any of the DMs until it happens to notice that COMMIT operations have been received from some read-quorum of DMs. Also, since it is not necessary for correctness (as opposed to efficiency) for the read-TM to remember which of its children have aborted, the ABORT(T') operation has no postconditions.

The nondeterminism allows for greater generality of our results. However, one would not want to implement read-TMs this loosely in a real system. For the sake of efficiency, one would want to limit the number of accesses invoked by a read-TM. For example, one would want the read-TM to invoke accesses with some particular read-quorum in mind. The important point, however, is that all of our results apply even if such heuristics are added. Our proofs depend only upon the fact that all operations performed satisfy the preconditions and postconditions we define.

Write TMs: Let x be a logical data item in I. The purpose of a write-TM for x is to perform a logical write access to x. The formal description of a write-TM automaton follows.

A write-TM T for x has state components awake, data, read-requested, write-requested, read and written, where awake is a boolean variable, data is an element of D_x, read-requested and write-requested are subsets of $acc(x)$, and read and written are subsets of $dm(x)$. Initially, data = $\langle 0, i_x \rangle$, awake is false, and the sets are empty. Every write-TM T for x has an associated value $value$(T) $\in V_x$.

```
Input Operations:
  CREATE(T)
  COMMIT(T',v), where T'∈children(T) and v∈$D_x$
  ABORT(T'), where T' ∈ children(T)
Output Operations:
  REQUEST-CREATE(T'), where T' ∈ children(T)
  REQUEST-COMMIT(T,v), where v = nil
```

```
•CREATE(T)
 Postconditions:
  awake(s) = true
•REQUEST-CREATE(T'), where kind(T') = read
 Preconditions:
  awake(s') = true
  T' ∉ read-requested(s')
 Postconditions:
  read-requested(s) = read-requested(s') ∪ {T'}
•COMMIT(T',d), where kind(T') = read
 Postconditions:
  if write-requested(s') = {},
   read(s) = read(s') ∪ {O(T')}
   if d.version-number>data(s').version-number,
    data(s).version-number = d.version-number
•REQUEST-CREATE(T'), where
    kind(T') = write and data(T') = d
```

Preconditions:
awake(s') = true
$q \in config(x).r$
$q \subseteq$ read(s')
d = ⟨data(s').version-number+1,value(T)⟩
T' $\notin$ write-requested(s')
Postconditions:
write-requested(s) = write-requested(s') ∪ {T'}

•COMMIT(T',v), where kind(T') = write
Postconditions:
written(s) = written(s') ∪ {O(T')}

•ABORT(T')
Postconditions:
(no change)

•REQUEST-COMMIT(T,v)
Preconditions:
awake = true
v = nil
$q \in config(x).w$
$q \subseteq$ written(s')
Postconditions:
awake = false

A write-TM invokes read accesses to some number of DMs for x, keeping track of the highest version number returned. Once information from a read-quorum of DMs has been collected, the write-TM may begin invoking write accesses. (See the REQUEST-CREATE(T') operation.) The version-number of each write access invoked is one greater than the version-number in the data component of the write-TM's state, and the value of each write access invoked is value(T). Once COMMIT operations have been received from a write-quorum of DMs, the write-TM may request to commit.

It is possible that some read accesses to the DMs may not commit until after the write-TM has already invoked one or more write accesses. Thus, some read accesses may actually return the data that was written to the DMs on behalf of the write-TM itself. Therefore, in order to prevent the write-TM from seeing the data it wrote and incorrectly increasing its version-number, the COMMIT operation for read accesses is defined so that the state of the write-TM is modified only if no write accesses have been invoked.

Our discussion of the nondeterminism in read-TMs also applies to write-TMs, as well as to all other automata we define.

Lemma 4 TMs are transactions.

Lemma 5 Schedules of system B are well-formed.

Proof: By Lemmas 3 and 4, DMs are basic objects and TMs are transactions. Therefore, system B is a serial system. In [16], it is proved that all schedules of serial systems are well-formed. ■

The following definitions are useful for describing the logical accesses to the logical data items in system B and for setting up inductive arguments about these logical accesses.

Access sequence: This definition formalizes the intuitive notion of a sequence of logical accesses to x.

Let β be a sequence of operations of system B, and let x be a logical data item in I. Then the *access sequence* of x in β, denoted $access(x, \beta)$, is defined to be the subsequence of β containing the CREATE and REQUEST-COMMIT operations for the members of $tm(x)$.

Logical state: The following definition formalizes the intuitive notion of the "current state" of a logical data item, the expected return value of a logical read.

Let β be a sequence of operations of system B, and let x be a logical data item in I. The *logical state of x after β*, denoted $logical\text{-}state(x, \beta)$, is defined to be either *value*(T) if REQUEST-COMMIT(T,v) is the last REQUEST-COMMIT operation for a write-TM in access(x, β), or i_x if no REQUEST-COMMIT operation for a write-TM occurs in access(x, β).

Current version number: Let β be a sequence of operations of system B, and let x be a logical data item in I. Let $last(x, \beta)$ denote the subset of $acc(x)$ such that for each member T of last(x, β), REQUEST-COMMIT for T is the last REQUEST-COMMIT operation for a write access to O(T) in β. The *current version number of x after β*, denoted *current-vn*(x, β), is defined as follows. If last(x, β) is nonempty, then current-vn(x, β) is the maximum over all T∈last(x, β) of data(T).version-number. Otherwise, current-vn(x, β) = 0.

The next pair of lemmas follow immediately from the above definitions and Lemma 5.

Lemma 6 If β is a schedule of B and x is a logical data item in I, then access(x, β) begins with a CREATE operation for some TM in $tm(x)$ and continues alternately with REQUEST-COMMIT and CREATE operations for TMs in $tm(x)$ such that each REQUEST-COMMIT for T is preceded immediately by a CREATE(T) operation.

Lemma 7 Let x be a logical data item, and let β be a schedule of B. Then the following property holds after β: The highest version number among the states of all DMs in $dm(x)$ is current-vn(x, β).

The following lemma is the key to the proof of Theorem 10. Condition 1 is only needed for carrying through the inductive argument. The important part of the lemma is Condition 2, which tells us that each read-TM returns the value expected as dictated by the previous logical write operations. That is, each read-TM returns the logical-state of the data item. Because the system is serial, we are able to carry out a simple inductive proof using standard assertional techniques.

Lemma 8 Let x be a logical data item in $\mathcal{I}$. Let β be a schedule of B such that access(x, β) is of even length.

1. The following properties hold after β:
   (a) There exists a write-quorum $q \in \mathit{config}(x).w$ such that for all DMs $O \in q$, if d is the data component of O, then d.version-number = current-vn(x, β).
   (b) For all DMs $O \in \mathit{dm}(x)$, if d is the data component of O, then d.version-number = current-vn(x, β) implies that d.value = logical-state(x, β).
2. If β ends in REQUEST-COMMIT(T,v) with T$\in$ $\mathit{tm}_r(x)$, then v = logical-state(x, β).

Proof: By induction on the length of β. The base case (β the empty schedule) is trivial. Let $\beta = \beta'\tau$, where access(x, τ) begins with the last CREATE operation in access(x, β). Assume that the Lemma holds for β'. By Lemma 6 and the fact that access(x, β) is of even length, access(x, τ) = (CREATE(T$_f$), REQUEST-COMMIT(T$_f$,v$_f$)) for some T$_f \in \mathit{tm}(x)$ and v$_f \in V_x$. The following assertions about T$_f$ may be easily proved:

Fact 1: All accesses in τ to DMs in $\mathit{dm}(x)$ are descendants of T$_f$.

Fact 2: Let s be the state of T$_f$ after any prefix of β. If read(s) is non-empty, then data(s).version-number and data(s).value contain the highest version-number and associated value among the states of the DMs in read(s) after β'.

Fact 3: Let s be the state of T$_f$ after any prefix of β. If read(s) contains some read quorum $r \in \mathit{config(x).r}$, then data(s).version-number = current-vn(x, β') and data(s).value = logical-state(x, β').

Fact 4: If T$_f$ is a write-TM, then all write accesses T' invoked by T$_f$ have data(T') = ⟨current-vn(x, β')+1, value(T$_f$)⟩.

By Fact 1, in order to prove that the induction hypothesis holds for β, we merely need to demonstrate that T$_f$ preserves the properties stated. There are two possibilities for T$_f$; it may be either a read-TM or a write-TM:

If T$_f$ is a read-TM, then logical-state(x, β) = logical-state(x, β') by definition. Also, since T$_f$ invokes only read accesses, the version-number and value components of the states of the DMs in $\mathit{dm}(x)$ after β are the same as after β', and current-vn(x, β) = current-vn(x, β'). Therefore, part 1 of the Lemma holds for β.

Let s$_f$ be the state of T$_f$ when T$_f$ issues its REQUEST-COMMIT operation. The preconditions for REQUEST-COMMIT require that read(s$_f$) contain some read-quorum $r \in \mathit{config}(x).r$. Therefore, by Fact 3, data(s$_f$).value = logical-state(x, β'), which equals logical-state(x, β). By definition, v$_f$ = data(s$_f$).value, so part 2 of the Lemma holds for β.

If T$_f$ is a write-TM, then logical-state(x, β) = value(T$_f$) by definition. Let s$_f$ be the state of T$_f$ when T$_f$ issues its REQUEST-COMMIT operation. The preconditions for REQUEST-COMMIT require that written(s$_f$) contain some write-quorum $w \in \mathit{config}(x).w$. Furthermore, no DM is added to the written component of the state of T$_f$ unless a write access to that DM has committed to T$_f$. So, τ must contain a REQUEST-COMMIT operation for a write access to each DM in w. After a COMMIT of a write access T' to a DM, the data component of that DM is equal to data(T'). Therefore, by Fact 4, the states after β of all the DMs in w must have value = value(T$_f$) and version-number = current-vn(x, β')+1. (By Fact 1, T$_f$ is the only transaction that issues write accesses to DMs in $\mathit{dm}(x)$ in τ.)

By Lemma 7, current-vn(x, β') is the highest version-number among the states of DMs in $\mathit{dm}(x)$ after β'. Since every write access in τ to DMs in $\mathit{dm}(x)$ has version-number = current-vn(x, β')+1, we know that this is the highest version-number among DMs in $\mathit{dm}(x)$ after β. That is, current-vn(x, β')+1 = current-vn(x, β). Therefore, since value(T$_f$) = logical-state(x, β), part 1 of the Lemma holds. Since T$_f$ is not a read-TM, β does not end with a REQUEST-COMMIT of a read-TM for x, so part 2 holds vacuously.

Thus, the Lemma holds in both cases. ■

3.2 Non-replicated Serial System

As the basis of our correctness condition, we define non-replicated serial system A of type ($\mathcal{T}_A$,parent$_A$,$\mathcal{O}_A$,$\mathcal{V}_A$) in terms of replicated serial system B of type ($\mathcal{T}_B$,parent$_B$,$\mathcal{O}_B$,$\mathcal{V}_B$).[3] System A is identical to System B, except that logical accesses to objects in $\mathcal{I}$ (which are implemented as TMs in system B) are implemented as accesses in system A, and

[3] We introduce the subscripts to distinguish the components of A from the components of B.

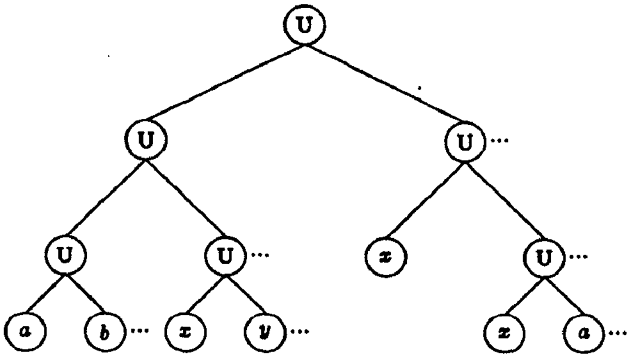

Figure 2: The transaction tree for system A that corresponds to the transaction tree for B shown in Figure 1. Transactions are labeled as follows: U = user transaction; a, b, x, y = accesses.

the logical data items in $\mathcal{I}$ (which are implemented as collections of DMs in system B) are implemented as single read-write objects in system A. These changes are reflected in the system type, which is formally defined as follows:

- $\mathcal{T}_A = \mathcal{T}_B - \left(\bigcup_{x \in \mathcal{I}} acc(x)\right)$
- parent$_A$ = parent$_B$ restricted to $\mathcal{T}_A$
- $\mathcal{O}_A = \mathcal{O}_B - \left(\bigcup_{x \in \mathcal{I}} dm(x)\right) \cup \{tm(x) | x \in \mathcal{I}\}$
- $\mathrm{V}_A = \mathrm{V}_B$

Informally, to construct the type of system A from that of system B, we first remove from $\mathcal{T}$ all the accesses to the DMs for objects in $\mathcal{I}$. As a result, all the TMs for objects in $\mathcal{I}$ become leaves in $\mathcal{T}$ and are therefore accesses. Next, we remove from $\mathcal{O}$ all the DMs for objects in $\mathcal{I}$. Also, we partition all the accesses that were formerly TMs according to their logical data item. Each class of this partition is a new object in $\mathcal{O}$. Thus, each logical data item is implemented by a single object.

Figure 2 illustrates the transaction tree for system A that corresponds to the transaction tree for system B given in Figure 1.

We would like to relate transactions in system B to those in system A. Recall that the function $\mathcal{F}_{AB}$ is well-defined, provided that system B is an extension of system A. Thus, we state the following easy lemma.

Lemma 9 System B is an extension of system A.

We define ***user transactions*** in system A to be all non-access transactions in $\mathcal{T}_A$. We note that T is a user transaction in system B iff $\mathcal{F}_{BA}$(T) is a user transaction in system A. This is because if T is a TM in system B, then $\mathcal{F}_{BA}$(T) is an access transaction.

Transactions and objects in system A have the same corresponding automata as in system B, except that for all $x \in \mathcal{I}$, the following hold:

1. The object corresponding to $tm(x)$ is modelled as a read-write object O over domain V_x with initial value i_x. (We refer to this particular read-write object as O(x).)
2. For each transaction T$\in tm(x)$, $\mathcal{F}_{BA}$(T) is an access to O(x) such that
   (a) if T is a read-TM, then $\mathcal{F}_{BA}$(T) is a read access, and
   (b) if T is a write-TM, then $\mathcal{F}_{BA}$(T) is a write access with data($\mathcal{F}_{BA}$(T)) = value(T).

3.3 Correctness

In this section, we prove that system B is correct by showing that user transactions cannot distinguish between replicated serial system B and non-replicated serial system A.

Theorem 10 Let β be a schedule of replicated serial system B. There exists a schedule α of non-replicated serial system A such that the following two conditions hold.

1. For all objects O in system B that are not in $dm(x)$ for any x, $\alpha|\mathrm{O} = \beta|\mathrm{O}$.
2. For all user transactions T in system B, $\alpha|\mathcal{F}_{BA}(\mathrm{T}) = \beta|\mathrm{T}$.

Proof: We construct α by removing from β all the REQUEST-CREATE(T), CREATE(T), REQUEST-COMMIT(T,v), COMMIT(T,v), and ABORT(T) operations for all transactions T in $acc(x)$ for all $x \in \mathcal{I}$. Clearly, the two conditions hold. What needs to be proved is that α is a schedule of A. We proceed by induction on the length of β.

The base case (β empty) is trivial. Let $\beta = \beta'\pi_\beta$, where the claim holds for β'. Let $\alpha = \alpha'\pi_\alpha$, where α' is the schedule of A corresponding to β'. We show, by a simple case analysis, that for each possible operations π_β, the claim holds for β.

The interesting case is when π_β is a REQUEST-COMMIT(T,v), where T$\in tm(x)$ for some $x \in \mathcal{I}$. By the construction $\pi_\alpha = \pi_\beta$. By the definition of system A, $\mathcal{F}_{BA}$(T) is an access to a read-write object. The only precondition for a REQUEST-COMMIT of T, then, is that T has been created. By the construction and the fact that β is a well-formed schedule, CREATE(T) occurs in α'. Therefore, the precondition for REQUEST-COMMIT(T,v') is satisfied in A for some v'.

If T is a write-TM, then $v = v' = nil$. We need to show that $v = v'$ if T is a read-TM. By Lemma 8, we know that $v =$ logical-state(x, β'). By definition of a read-write object, v' is the value in the state of O(x) after α'. We observe that, by the construction, $\alpha'|$O$(x) =$ access(x, β'). So, by definition of system A, the last write access in α' to O(x) has the same value as the last write-TM in β'. Hence, the value in the state of O(x) after α' is logical-state(x, β'). Therefore, $v = v'$. ∎

3.4 Concurrent Replicated Systems

So far, we have been able to deal exclusively with serial systems in order to simplify our reasoning. We now complete the correctness proof by showing that non-serial replicated systems are correct. Recall the definition of serial correctness: Let S be a serial system, and let γ be an arbitrary sequence of operations. We say that γ *is serially correct with respect to S for transaction* T provided that $\gamma|$T $= \sigma|$T for some schedule σ of S.

With the following theorem, we show that given a correct concurrency control algorithm, combining that algorithm with our replication algorithm yields a correct system. This theorem allows us to achieve a complete separation of the issues of concurrency control and recovery from the issues of replication. In other words, one may prove a concurrency control algorithm correct, then separately prove a replication algorithm correct for serial systems, and finally apply this theorem to show that the (combined) concurrent replicated system is correct. The modularity of this proof method permits us to ignore all the complicated interactions of the two algorithms that one would need to consider in a direct proof that the concurrent replicated system simulates a non-replicated serial system.

Theorem 11 Let C be any system that has the same type as system B, and let the set of user transactions in C be the same as in B. Assume that all schedules γ of C are serially correct with respect to serial system B for all non-orphan[4] non-access transactions. Then all schedules γ of C are serially correct with respect to system A for all non-orphan user transactions.

Proof: Immediate from Theorem 10. ∎

So, any concurrency control algorithm that provides serializability at the level of the copies may be combined with the Fixed Quorum Consensus replica management algorithm to produce a correct system. Interesting concurrency control algorithms that satisfy this condition include Reed's multi-version timestamp concurrency control algorithm [20] and Moss' two phase locking algorithm with separate read and write locks [19]. (See also the correctness proof given by Fekete et al. [9].)

4 Reconfiguration

In this section, we describe how the results of Section 3 are extended to systems that permit reconfiguration. By *reconfiguration*, we mean that read- and write-quorums are permitted to change dynamically, rather than being fixed for the entire execution. This flexibility is important for coping with site and link failures in practical systems. For example, if some DMs are down, we may want to change the quorums so that logical accesses can be processed in spite of the failures.

To prove the reconfiguration algorithm correct, systems A and B are redefined. Then, proofs analogous to those for the fixed configuration systems are constructed. In doing so, some interesting new considerations arise: As before, the logical accesses are described in terms of read- and write-TMs. However, we also need a new kind of TM, called a reconfigure-TM, to effect changes in the quorums. We would like the reconfigure-TMs to be modelled as transactions for the sake of uniformity, and to be positioned in the tree as children of the user transactions in order to model the correct atomicity requirements. For instance, if T and T' are TMs for x that are invoked by the same user transaction, we would like to permit reconfiguration of x to take place between the COMMIT of T and the CREATE of T'. However, the reconfigure-TMs are special in that their invocations and returns are not to be controlled, or even seen, by the user transactions. Rather, they are intended to run spontaneously and transparently from the user's point of view. So, we want the reconfigure-TMs to be positioned in the tree as children of the user transactions, but we do not want the user programs to be aware of their invocations and returns.

This conflict introduces a modelling problem. We solve the problem by associating a *spy* automaton with each user transaction. The spy wakes up with the associated transaction and nondeterministically invokes reconfigure-TMs until the associated transaction requests to commit. In this way, we capture

[4] A a transaction T is an orphan in γ if ABORT(T') occurs in γ for some ancestor T' of T.

formally the notions of spontaneity and transparency while at the same time modelling the proper atomicity requirements.

Gifford's reconfiguration algorithm[5] works as follows. In addition to a value and a version number, each replica of x contains a configuration and a generation number. The value and version number are initialized as in the non-reconfiguration case, and all replicas of x initially hold the same configuration and generation number.

To perform a logical read of x, a TM reads DMs for x, keeping in its state the value v and version number t from the DM with the highest version number seen, the configuration c and generation number g from the copy with the highest generation number seen, and the set d of the names of the DMs read. If the TM reaches a state in which c has a read-quorum that is a subset of d, then the TM returns v.

To perform a logical write of x with new value v', a TM again reads DMs for x, keeping in its state the version number t from the DM with the highest version number seen, the configuration c and generation number g from the DM with the highest generation number seen, and the set d of the names of the DMs read. If the TM reaches a state in which c has a read-quorum that is a subset of d, then the TM computes the new version number $t' = t + 1$ and writes v' along with t' to some write-quorum of DMs in c.

To reconfigure x with new configuration c', a TM first reads DMs for x and computes $v, t, c, g,$ and d, just as for a logical read. If the TM reaches a state in which c has a read-quorum that is a subset of d, then the TM does the following. It writes v and t to a write-quorum in c', and it writes c' and $g' = g + 1$ to a write-quorum in c.[6]

We generalize Gifford's reconfiguration algorithm in the same ways that we generalized the fixed quorum consensus algorithm in the previous section. To simplify our reasoning, we separate the read, write, and reconfigure tasks of the TMs into modules called *coordinators*. This is done most naturally by introducing another level of nesting, providing additional evidence of the power of nesting as a modelling tool.

The formalisms and proofs follow the same pattern as those of the previous section. The complete proof may be found in [11].

5 Conclusion

We have presented a precise description and rigorous correctness proof for a generalization of Gifford's data replication algorithm that accommodates nested transactions and transaction failures. The algorithm was described using the new Lynch-Merritt input-output automaton model for nested transaction systems, and the correctness proof was constructed directly from this description.

The algorithm was decomposed into simple modules that were arranged naturally in a tree structure. This use of nesting as a modelling 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 one would not actually implement a system in this way, the nondeterminism permitted us to construct a correctness proof that was independent of any particular programming language or implementation.

The modularity of the proof strategy permitted us to separate the concerns of replication from those of concurrency control and recovery. Our arguments were simple, in part, because of this separation. That is, we were able to deal exclusively with serial systems in order to simplify our reasoning. Then, to complete the proof, we presented a simple theorem which stated that combining any correct concurrency control algorithm with our replication algorithm yields a correct system.

One possible direction for further work involves using this general technique to add transaction nesting to other, more complicated, data replication schemes, and prove the resulting algorithms correct. Some interesting examples include the "Virtual Partition" approach of Abbadi and Toueg [2], and Herlihy's "General Quorum Consensus" [12].

Some replication algorithms guarantee weaker correctness conditions than the one presented here for Gifford's algorithm. It would be interesting to see what impact these weaker correctness conditions would have on the proof structure that we have presented.

Acknowledgments

We thank Alan Fekete, Sally Goldman, Mark Tuttle, and Jennifer Welch for their helpful comments on various drafts of this paper. Also, we thank Bill Weihl and Sharon Perl for useful discussions during the early stages of this work.

[5] In [10], Gifford describes the algorithm in terms of *votes*. However, we substitute the more general configuration definition.

[6] The description in [10] actually requires that the new configuration be written to both an old and a new write-quorum. However, we find that it is only necessary to write this information to an old write-quorum.

References

[1] El Abbadi, A., Skeen, D. and Cristian, F., "An Efficient Fault-Tolerant Protocol for Replicated Data Management", *Proc. 4th ACM Symposium on Principles of Database Systems*, Portland, Oregon, March 1985, pp. 215-229.

[2] El Abbadi, A. and Toueg, S., "Maintaining Availability in Partitioned Replicated Databases", *Proc. 5th ACM Symposium on Principles of Database Systems*, 1986.

[3] Barbara, D., and Garcia-Molina, H., "Mutual Exclusion in Partitioned Distributed Systems," Technical Report, Department of Computer Science, Princeton University, TR-346, July 1985.

[4] Beeri, C., Bernstein, P. A., and Goodman, N., "A Model for Concurrency in Nested Transaction Systems," Technical Report, Wang Institute TR-86-03, March 1986.

[5] Beeri, C., Bernstein, P. A., Goodman, N., Lai, M. Y., and Shasha, D. E., "A Concurrency Control Theory for Nested Transactions," *Proc. 1983 Second Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, Canada, August 17-19, 1983, pp. 45-62.

[6] Bernstein, P. A., and Goodman, N., "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys* 13,2 (June 1981), pp. 185-221.

[7] Bernstein, P. A., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1986.

[8] Eager, D. and Sevcik, K., "Robustness in Distributed Database Systems", *Transactions on Database Systems*, 8,3 (September 1983), pp. 354-381.

[9] Fekete, A., Lynch, N., Merritt, M., and Weihl, W., "Nested Transactions and Read/Write Locking," *Proc. 6th ACM Symposium on Principles of Database Systems*, March, 1987.

[10] Gifford, D., "Weighted Voting for Replicated Data", *Proc. of the 7th Symposium on Operating Systems Principles*, December, 1979.

[11] Goldman, K., and Lynch, N., "Data Replication in Nested Transaction Systems," Technical Report MIT/LCS/TR-390, MIT Laboratory for Computer Science, Cambridge, MA., May 1987.

[12] Herlihy, M., "Replication Methods for Abstract Data Types," Technical Report MIT/LCS/TR-319, MIT Laboratory for Computer Science, Cambridge, MA, May 1984.

[13] Herlihy, M., Lynch, N., Merritt, M., and Weihl, W., "On the Correctness of Orphan Elimination Algorithms," submitted for publication.

[14] Liskov, B., Herlihy, M., Johnson, P., Leavens, G., Scheifler, R., and Weihl, W., "Preliminary Argus Reference Manual," Programming Methodology Group Memo 39, October 1983.

[15] Liskov, B., and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *ACM Transactions on Programming Languages and Systems* Vol. 5, No. 3, July 1983, pp. 381-404.

[16] Lynch, N., and Merritt, M., "Introduction to the Theory of Nested Transactions," Technical Report MIT/LCS/TR-367, MIT Laboratory for Computer Science, Cambridge, MA., July 1986.

[17] Lynch, N. A., "Concurrency Control For Resilient Nested Transactions," *Advances in Computing Research* 3, 1986, pp. 335-373.

[18] Lynch, N. and Tuttle, M., "Hierarchical Correctness Proofs for Distributed Algorithms," Technical Report MIT/LCS/TR-387, MIT Laboratory for Computer Science, Cambridge, MA., April 1987. Also appeared in *Proc. of the 6th Symposium on Principles of Distributed Computing*, August 1987.

[19] Moss, J. E. B., "Nested Transactions: An Approach To Reliable Distributed Computing," Ph.D. Thesis, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, MA., April 1981. Also, published by MIT Press, March 1985.

[20] Reed, D. P., "Implementing Atomic Actions on Decentralized Data", *ACM Trans. on Computing Systems*, Vol. 1, No. 1, pp. 3-23.

[21] Thomas, R. H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. on Database Systems*, Vol. 4, No. 2, June 1979, pp. 180-209.

[22] Weihl, W. E., "Specification and Implementation of Atomic Data Types," Ph.D Thesis, Technical Report/MIT/LCS/TR-314, MIT Laboratory for Computer Science, Cambridge, MA., March 1984.