Virtual partition algorithm in a nested transaction environment and its correctness

Sanjay Kumar Madria a, S.N. Maheshwari b, B. Chandra c

a Department of Computer Science, University of Missouri – Rolla, Rolla, MO 65401, USA
b Department of CS & E, Indian Institute of Technology, Hauz Khas, New Delhi, India
c Department of Mathematics, Indian Institute of Technology, Hauz Khas, New Delhi, India

Abstract

In this paper, we present a formal description of the virtual partition algorithm in a nested transaction environment and prove its correctness. We model the virtual partition algorithm in a nested transaction environment using the I/O automaton model. The formal description is used to construct a complete correctness proof that is based on standard assertional techniques and on a natural correctness condition, and takes advantage of the modularity that arises from describing the algorithm as nested transactions. Our presentation and proof treat issues of data replication entirely separately from issues of concurrency control. Moreover, we have identified that the virtual partition algorithm cannot be proven correct in the sense of Goldman’s work [ACM Trans. Database Syst. 19(4) (1994) 537] on Gifford’s quorum consensus algorithm using the serializability theorem defined by Fekete et al. [Atomic Transactions, Morgan-Kaufmann, USA, 1994]. Thus, we have stated a weaker notion of correctness conditions, which we call the reorder serializability theorem. We have shown that not all classes of replication algorithms can be proven in the way Goldman has presented the proof of Gifford’s quorum consensus algorithm.

Keywords: Virtual partition algorithm; Nested transaction; Data replication; Quorum consensus; Reorder serializability
1. Introduction

The theory of nested transactions [18,23] allows the benefits of atomicity to be available within a transaction. In a nested transaction model [23], a subtransaction may contain operations to be performed concurrently, or operations that may be aborted independently of their invoking transaction. Such operations are considered as subtransactions (children) of the original transaction. This parent–child relationship defines a nested transaction tree and such transactions are termed as nested transactions [23]. Failure of subtransactions may result in invocation of alternate subtransactions that could replace the failed ones to accomplish the successful completion of the whole transaction. A child transaction has access to the data locked by its parent. It is atomic with respect to its parent and its siblings. It is serializable with its siblings. It becomes permanent only if its parent becomes permanent. If a parent aborts, all its descendants’ effects are undone. Therefore, a child’s scope is restricted to its parent only. Hence, this model is termed a closed nested transaction model. A parent commits only after all its children are terminated.

Replication is a key factor to improve availability. By replicating data at multiple sites, the distributed database system can operate even in the case of some site failures or network partitions and, therefore, provides increased availability. Replication of the data objects raises the issue of consistency among the replicas. The strongest and simplest notion of consistency is atomicity, which requires the replicas to collectively emulate a single centralized object. To achieve atomicity with replicated data, several methods have been proposed such as write-all/read-one [2], primary-copy [25], majority consensus [26], and quorum consensus [3,4,10]. All these methods ensure that a replicated database behaves like a single copy database for the user program, i.e., replication should be transparent to the user program. However, to achieve higher performance, a different notion of consistency, called sequential consistency [13], allows operations to be ordered as long as they remain consistent with the view of individual clients. There are some other systems that provide weaker guarantee of the clients [19] to get better performance. Improving performance by weaker notion of consistency can have more complicated semantics, and it may be more difficult to understand and reason the correctness of replicated systems.

In a replication algorithm, accesses to each logical data object can be implemented by a collection of data managers (DMs) and transaction managers (TMs). The collective state of the DMs defines the current state of the single copy data object, i.e., the logical data object. The user programs in order to access logical data objects invoke these TMs. A TM does this by accessing a group of DMs.

In the quorum consensus algorithm [10], every copy of a replicated object is assigned some number of votes. Every transaction must collect a read-quorum
of \( r \) votes to read a data object and a write-quorum of \( w \) votes to write a data object. A pair of the form \( (r, w) \) is called a configuration. Each configuration must satisfy two constraints. First, \( r + w \) should be \( > v \), where \( v \) is the total number of votes. This ensures that there is no non-null intersection between the set of sites contributing to \( r \) and \( w \), respectively. Second, \( w \) should be \( > r \). This condition ensures that two writes cannot happen in parallel on the same data object.

In the virtual partition algorithm of [3] and [4], each logical data object also maintains read- and write-quorums. The quorums, however, serve a different purpose in the virtual partition algorithm. The basic idea here is that each TM maintains a "view" of the DMs which the TM believes it can communicate with. In this algorithm, instead of reading a read-quorum of DMs, the TM can read from any one of the DMs in its view having a read-quorum of DMs. Similarly, the TM writes at all DMs in its view containing a write-quorum of DMs. Since the writing is performed at all the DMs in the view and reading can be done at any one of the DMs in the view, there is greater availability. Also, if a transaction incorrectly believes that a DM in its view has a read-quorum, it may read an old value. It will not be able to discover the new updates until an accessible update is performed in its view. Fortunately, these "correct" read operations could safely be run "as in the past". If a transaction believes that a DM in its view has a write-quorum but in fact does not, then its incorrect view will be discovered at the time write-all is attempted. However, this algorithm does not have enough details to enable complete understanding of its working and to reason about the correctness of transactions. This is due to the fact that transactions, objects and the scheduler are not modeled explicitly. The correctness proof of the algorithm is based on the notion of proving the absence of cycles in a graph representing dependencies among transactions and, hence, does give much insight into the operations of the different modules of the algorithm. The algorithm also ignores recovery from transaction aborts.

In [11], Goldman and Lynch presented a quorum consensus algorithm [10] by incorporating the concept of transaction nesting [23] and transaction aborts. The proof technique in [11] proves that a replicated serial system is the same as a non-replicated serial system. It then uses the known fact that a concurrent replicated system, using the concurrency control algorithm given in [8, 24], is the same as a non-replicated serial system. This seems to work for a class of replicated serial systems but not for all. The virtual partition algorithm [4] has the property that an access in partition \( A \) may read a data object that was already modified by a different transaction in partition \( B \). Thus, the virtual partition algorithm allows some read-only transactions to "run as if in the past" and, therefore, it does not satisfy the external consistency condition since the apparent serial order has a different sequence when projected on the timestamp order. Hence, it does not satisfy the serial correctness condition as
defined in [8, 11]. That is, the virtual partition algorithm cannot be proved to be “serially correct” [18] as in [11]. Hence, it cannot be combined with algorithms such as those in [7] and [24] that provide “one-copy serializable schedules”. A weaker notion of correctness condition is needed that should allow reordering of transactions to bring the outdated read-only transactions at a point in the past consistent with the ordering of other transactions. Our nested transaction version of the virtual partition algorithm, therefore, can work with any of the algorithms that satisfy the weaker correctness condition. We have stated a weaker notion of correctness which we call reorder serially correct.

In this paper, we have generalized the virtual partition algorithm [3, 4] in a framework similar to Goldman’s analysis of the quorum consensus algorithm [11] by incorporating the concept of transaction nesting and transaction aborts. In our nested transaction version of the virtual partition algorithm, we have combined view formation and view update protocols in one module that is an improvement over the original algorithm [4]. We have placed the view formation and update protocols at the lower level in the transaction nesting structure to provide view transparency. We have also explained how view formation and update protocols are invoked and controlled by the TMs and not by the user-visible transactions.

In this paper, we show that the virtual partition algorithm under nested transaction environment satisfies the weaker notion of correctness. We prove that a serial replicated system is the same as a serial non-replicated system as far as the user is concerned. To argue the correctness of the concurrent replicated system being equal to the serial non-replicated system, we need to find a concurrent algorithm which satisfies the weaker notion of correctness condition so that our replication algorithm can be placed on the top of such algorithms. An open problem that still remains is to state and prove the reorder serializability theorem based on our reorder serial correctness. Another open problem is to find a concurrent algorithm (that provides the reading of old values) and to prove that it satisfies the weaker notion of correctness condition and the reorder serializability theorem analogous to the serializability theorem defined in [18].

1.1. Related work on the proof of correctness

Some more related work on formal correctness using the I/O automaton model [16] are as given below. Most of these algorithms are discussed using the I/O automaton model [16] and more details on some of these algorithms are available in [7].

- In [17], Lynch has presented a complete proof of the exclusive locking algorithm for nested transactions. Reed [24] has presented a multi-version timestamp concurrency control algorithm to provide nested-transaction-based data management. In [1], a formal analysis of the multi-version time-
stamp-based algorithm is given. The algorithm is proved to be correct by showing that the objects used in the algorithm are all static atomic. The atomicity theorem is used to show that if all the objects in the system are static atomic then the system guarantees atomicity. The proof techniques given are very general and can be applied to large classes of systems including those where different data objects are implemented independently, and where type of objects can be used to obtain increased concurrency.

- Moss [23] has extended two-phase locking with separate read/write locks to handle nesting. A formal version of this algorithm has appeared in [8]. In [9], the read-update locking algorithm [27] has been generalized and a new commutative locking algorithm has been introduced to handle nested transactions. The paper defines a local atomicity condition for data objects, called dynamic atomicity. The atomicity theorem is used to show that if all the objects in the system are dynamic atomic then the system guarantees atomicity. The dynamic atomicity provides modularity as it allows one to verify the implementations of individual objects independently. It also allows the use of different implementation techniques and different algorithms in different objects as long as all the objects are dynamic atomic.

- Fekete et al. [6] have presented a serialization graph construction for nested transactions. The proof technique has the same form as in the classical theory; one must show that a graph having transactions for nodes and edges representing ordering between transactions is acyclic. They have defined a new kind of serialization graph and proved that, under certain assumptions, the absence of cycles in this graph is a sufficient condition to ensure the serial correctness of a system. They have applied their technique to verify the correctness of Moss’s read/write locking algorithm for nested transactions, and an undo logging algorithm that has not been previously proved for the nested transaction system.

- The multi-granularity algorithm given in [12] has been extended to nested transaction systems in [14]. The algorithm considered file and record level granularity. The correctness proof shows that there is a possible mapping to the abstract algorithm for commutativity-based locking given in [9]. The paper shows that some objects can use multi-granularity locking while others use Moss’s two-phase locking and yet others use other user-defined dynamic atomic algorithms. Some more related work by the same authors on concurrency control using predicate locks appears in [13, 15].

- Gifford’s basic quorum consensus algorithm for data replication [10] is generalized by Goldman and Lynch in [11] to accommodate nested transactions and transaction aborts. The presentation separates the treatment of replication entirely from concurrency control, which helps simplify the reasoning. The paper shows that any correct concurrency control mechanism may be used on the copies, considered as separate objects; the whole system will then appear to be atomic and non-replicated.
• Nested transactions have also been discussed in the context of B-trees [5] and linear hash structures [21]. In [5], the notion of "strongly-serially correct" behavior has been defined and used as the correctness criterion. The drawbacks of [5] have been removed in [21]. In the algorithm, the locks have been considered at both key and vertex level. These locks have been implemented in nested transaction environment using Moss's two-phase locking algorithm and the locking protocols of the linear hash structure algorithm with lock-coupling technique. The linear hash structure algorithm in nested transaction environment is proved to be "serially correct".

• An open and safe nested transaction model and outline of correctness has appeared in [20]. The nested transaction model uses a prewrite operation before an actual write operation to increase the concurrency. The concurrency control is provided by extending Moss's two-phase locking algorithm. The model also takes into account the buffer management operations, which are controlled by the concurrency control algorithm. Non-access subtransactions, objects and the scheduler are modeled as I/O automata with the help of some pre- and postconditions. The concurrency control algorithm has been proved serially correct in the sense of [18].

• A short version of this paper will appear in [22], where most of the details of modeling and proofs are omitted.

1.2. Paper organization

The rest of the paper is organized as follows. In Section 2, we discuss some preliminaries required to understand the ideas presented in this paper. We discuss the virtual partition algorithm under nested transaction environment in Section 3. Reorder serial correctness and some properties of replicated systems under the virtual partition algorithm are given in Section 4. We discuss correctness of the algorithm in Section 5. Open problems are discussed in Section 6. We conclude in Section 7.

2. Preliminaries

2.1. Virtual partition algorithm

The virtual partition algorithm of [3] and [4] is designed so that a transaction never has to access more than one copy to read a data item. The basic idea of the virtual partition algorithm is that each site $A$ maintains a view consisting of sites with which $A$ can communicate. The view of a transaction is the view of its home site at the time $T$ starts. Initially, each view is associated with a unique vid (view identifier). Read- and write-quorums are maintained at each copy.
Read operation: A transaction translates a read($x$) into reading any of the copy of $x$ in its view having a read-quorum.

Write operation: A transaction translates a write($x$) into writes on all copies of $x$ in its view having a write-quorum.

View formation protocol: When a site wishes to form a new view, say $v$, it generates a unique vid greater than its present vid. The site $A$ sends a "join view messages" to each site in $v$ and sites in $v$ accept the invitation only if its present vid is less than the new vid. After receiving the acknowledgments, site $A$ either aborts the creation of the new view and starts the protocol again using a greater vid or may ignore rejected invitations and proceed to form a new view consisting only of the set of sites $v' \subseteq v$ that accepted the invitation. The site $A$ then sends the "view formed messages" to each site in $v'$ and attaches the set $v'$ to that message. All the sites in $v'$ adopt $v'$ as their present view.

View update protocol: When a site $A$ creates a new view, it reads a read-quorum of each data item $x$. It then writes all copies of $x$ in its view using the most recent value it read. This is because any such copies might be read in the view.

Reading of old values: If a site $A$ incorrectly believes that it has a read-quorum, it may read an old value and not discover the mistake until an inaccessible update is performed in its view. Fortunately, these "incorrect" read operations can safely be "run in the past" and read an old value of a data item during a failure. This ability to run in the past allows a site that has become isolated from the rest of the network to execute read-only transactions even if updates are being performed on remote copies of the data items stored at that site.

2.2. I/O automaton model

An I/O automaton $A$ as defined by Lynch and Merrit [18] is a five-tuple (states($A$), start($A$), out($A$), in($A$), steps($A$)). States($A$) is the set of states of $A$, start($A$) is the set of start states and a subset of states($A$). Out($A$) and in($A$) are the sets of output and input operations, respectively. Steps($A$) is the transition relation of $A$, which is a set of triples of the form ($s'$, $\pi$, $s$) where $s'$ and $s$ $\in$ states($A$), $\pi$ $\in$ in($A$) $\cup$ out($A$), i.e., the automaton changes its state from $s'$ to $s$ on operation $\pi$. An element of the transition is called a step of $A$.

The finite alternating sequence $s_0, \pi_1, s_1, \ldots, \pi_n, s_n$ of states and operations of $A$ is called an execution of $A$. A schedule of $A$ is the subsequence of an execution of $A$ consisting only of the operations of $A$.

A set of the I/O automaton may be composed to create a system $S$ such that the sets of output operations are disjoint. Thus, a state of the composed I/O automaton is a tuple of states, one for each component, and the start states are tuples consisting of start states of the components. Let $\alpha$ be a schedule of a system with a component $A$; then $\alpha \mid A$ is the subsequence of $\alpha$ containing exactly the operations of $A$. Clearly, $\alpha \mid A$ is a schedule of $A$. The reverse holds by the
composition lemma [18] which is formally stated as follows: Let $\sigma'$ be a schedule of a system $S$ and let $\sigma = \sigma' \pi$, where $\pi$ is an output operation of component $A$. If $\sigma |_A$ is a schedule of $A$, then $\sigma$ is a schedule of $S$.

2.3. Nested transaction system and correctness

A nested transaction system is modeled by a four-tuple $(\tau, \text{parent}, O, V)$, where $\tau$ is a set of transaction names organized into a tree by the mapping parent $\tau \rightarrow \tau$, where $T_0$ acts as the root. The set $O$ denotes the set of objects; it partitions the set of accesses, where each partition block contains accesses to the particular objects. $V$ is the set of return values. We can relate two nested transaction systems as follows: A nested transaction system $P = (\tau_P, \text{parent}_P, O_P, V_P)$ is called a structural extension of the nested transaction system $Q = (\tau_Q, \text{parent}_Q, O_Q, V_Q)$ if $\tau_P \supseteq \tau_Q$, $O_P \supseteq O_Q$, respectively, $V_P = V_Q$, and parent$_P$, restricted to $\tau_Q$, = parent$_Q$.

A nested transaction processing system is said to be modeled using the I/O automaton model when each component of the system, namely each non-access transactions, objects and the scheduler is modeled as an automaton. Each of these automata is specified with the help of some pre- and postconditions. These pre- and postconditions are used to prove the properties describing the behavior of the system. Formulation of nested transaction systems as I/O automata permits precise correctness conditions to be satisfied by the algorithm. These correctness conditions can be stated at the transaction interface that does not contain explicit information about object representation.

The correctness of a transaction processing system is defined in terms of a serial execution of the “same” system. That is, it requires an execution of the same system to exist in which transactions run one at a time without interleaving of steps of different transactions. Correctness is defined by first giving a separate specification of permissible serial executions as seen by a user of the system, and then defining how executions of a transaction processing system must relate to this specification. The permissible serial execution for a transaction processing system is defined by introducing the notion of a scheduler, which executes transactions serially. Such systems, called serial systems, are not constrained by the issues of concurrency control, recovery and transaction aborts.

Serial correctness. Formally, a schedule $\alpha$ of a system is serially correct for a transaction $T$ if its projection on $T$, $\alpha |_T$, is identical to $\beta |_T$ for some serial schedule $\beta$ [16]. In other words, $T$ sees the same things in $\alpha$ that it would see in some serial schedule. $\alpha$ is serially correct if it is serially correct for every non-orphan (transactions with no parents), non-access transaction.

The principal notion of correctness for a transaction processing system is that of serial correctness of the root transaction $T_0$ of all finite schedules. This says that the “outside world” cannot distinguish between the given system and the serial system. A fairly strong and possible interesting correctness condition
is the serial correctness of all non-access transactions. In this case, neither the outside word nor any other individual user transaction can distinguish between the given system and the serial system. Note that the definition of serial correctness, relative to all non-access transactions, does not require that all transactions see schedules that are a part of the same execution of the serial system; rather each could see schedules arising in a different serial execution.

Serial correctness for all non-orphan transactions implies serial correctness for \( T_0 \) because the serial scheduler does not have the action \( \text{ABORT}(T_0) \) so \( T_0 \) cannot be an orphan. Note that each correctness condition discussed here can be applied to many different kinds of transaction processing systems. All that is needed is that the system be modeled as an I/O automaton with appropriate named actions.

Transaction automata. We model each transaction as an I/O automaton that is specified with the help of the following operations [7]:

- **Input operations:**
  - \( \text{CREATE}(T) \)
  - \( \text{REPORT-COMMIT}(T', v) \), where \( T' \in \text{children}(T) \) and \( v \) is the return value
  - \( \text{REPORT-ABORT}(T') \), where \( T' \in \text{children}(T) \)

- **Output operations:**
  - \( \text{REQUEST-CREATE}(T') \), where \( T' \in \text{children}(T) \)
  - \( \text{REQUEST-COMMIT}(T, v) \), where \( v \) is the return value

The \( \text{CREATE} \) operation wakes up the transaction. The \( \text{REQUEST-CREATE} \) is a request by \( T \) to create a particular child transaction. The \( \text{REPORT-COMMIT} \) operation reports to \( T \) the successful completion of one of its children and returns a value depending upon the operation performed. The \( \text{REPORT-ABORT} \) operation reports to \( T \) the unsuccessful completion of one of its children, without returning any value. The \( \text{REQUEST-COMMIT}(T, v) \) is a request by \( T \) to commit and returns a value \( v \).

Serial object automata. The serial object automaton [7] serves as the specifications of the concurrent behavior of the operations on the data objects. The operations for each object are the \( \text{CREATE} \), \( \text{REQUEST-COMMIT} \) operations for all the corresponding access transactions. The \( \text{CREATE} \) operation is an invocation of an access to the object, while the \( \text{REQUEST-COMMIT} \) is a return of value in response to such an invocation.

- **Input operation:**
  - \( \text{CREATE}(T) \) for every access to an object \( X \)

- **Output operation:**
  - \( \text{REQUEST-COMMIT}(T, v) \) for every access of \( T \) to \( X \) and \( v \) is the return value

Serial scheduler automata. We model the serial scheduler [7] as an I/O automaton. It passes requests for the creation of subtransactions to the appropriate recipient, makes decisions about the completion of children and reports
back to their parents, and informs objects of the fate of transactions. The operations are as follows:

**Input operations:**
- REQUEST-CREATE(T)
- REQUEST-COMMIT(T, v)

**Output operations:**
- CREATE(T)
- COMMIT(T)
- ABORT(T)
- REPORT-COMMIT(T, v)
- REPORT-ABORT(T)

The REQUEST-CREATE and REQUEST-COMMIT inputs are identified with the corresponding output operations of transactions and object automata, and correspondingly for the CREATE, REPORT-COMMIT and REPORT-ABORT output operations. The COMMIT and ABORT operations are internal, marking the point of time where the decision on the fate of the transaction is irreversible. The COMMIT(T) and ABORT(T) are called return operations; for details see [7].

2.4. Non-replicated serial system

In a non-replicated environment, only one copy of each logical data object exists. The non-replicated serial system has, for each logical data object, a corresponding DM. To perform read and write accesses on a logical data object \(x\), a transaction has to access a DM for \(x\).

The nested transaction system of a non-replicated serial system \(A = (\tau_A, \text{parent}_A, O_A, V_A)\) has the following structure. In the system \(A\), all non-access transactions are user-visible. The nested transaction tree of the system \(A\) has access subtransactions at the leaf node and the user-visible transactions at one level above the access transactions. \(\text{parent}_A\) is a mapping which gives the parent-child relationships in the transaction tree. \(O_A\) is the set of logical data objects that partitions the set of accesses in system \(A\) such that each partition contains accesses to one particular data object. All the data objects of system \(A\) are user-visible. \(V_A\) is a set of return values. Each non-access transaction and DM is modeled as an I/O automaton. These automata issue requests to and receive replies from a serial scheduler, which is simply another I/O automaton [16].

3. Virtual partition algorithm and nested transaction system

In this section, we build a nested transaction tree (Fig. 1) in the case of a replicated environment under the virtual partition algorithm. We will observe
that the nested transaction trees up to the level of user-visible transactions are the same in the non-replicated and the replicated serial systems.

In a replicated database, data objects are modeled with the help of DMs. The read and write operations of the virtual partition algorithm are modeled by providing two kinds of TMs, namely read- and write-TMs. These TMs are introduced one layer below the user-visible transactions in the nested transaction tree. The read and write accesses are accomplished using two types of coordinators (COs), namely read- and write-COs. In the algorithm, each transaction maintains its operational view of DMs. To keep the user-visible transactions transparent to the view formed, we have modeled view formation and update protocols as view-coordinator (view-CO) rather than introducing it at the level of TMs. We have kept view-COs at lower levels in the transaction nesting structure so that their invocations and returns are not to be seen by the user-visible transactions. The COs are accommodated in the transaction tree at the intermediate layer between the TMs and the access subtransactions situated at the leaves of the transaction tree. That is, COs are placed as children of TMs. TMs and COs are user-invisible transactions. The nested transaction tree in the case of the replicated serial system, therefore, has two more levels of nesting after the level of user-visible transactions as compared to the non-replicated environment.

The virtual partition algorithm of [3,4] in the nested transaction environment works as follows: The set of all DMs for the logical data object $x$ models the set of physical replicas for $x$. DM keeps two types of data items, viz., the user data item $\{UD\}$ and the control data item $\{CD\}$ maintained in the form of the tuple $\{\{UD\}, \{CD\}\}$. $\{UD\}$ and $\{CD\}$ keep $\{\text{version-no}, \text{value}\}$ and $\{\text{vid}, \text{view-no}, \text{view}\}$, respectively. The collective form of $\{\{UD\}, \{CD\}\}$ is defined over the domain $D_x = \{\{\text{version-no}, \text{value}\}, \{\text{vid}, \text{view-no}, \text{view}\}\}$ for $x$ with initial data $\{\{0, 0\}, \{0, 0, 0, 0\}\}$. The user's data item $\{UD\}$ can be read and written by subtransactions invoked by any of the COs. The control data item $\{CD\}$ is written by subtransactions invoked by only view-COs. view, a part of the control data item, is associated with a unique vid. vid is used to form a new view. Therefore, each view-CO also maintains a vid. The vid of the view-CO is the same as that of the site where view-CO is initiated. We use $o(T)$ to denote the DM to which $T$ is an access.
The user-visible transactions invoke a TM in order to perform read and write operations. Each TM maintains its operational view of the DMs.

When a read-TM performs a logical read of \( x \), it invokes a read-CO that in turn invokes a read access to read one of the DMs for \( x \) in its view having some read-quorum. To achieve the non-deterministic nature of the algorithm, the read-CO may invoke more than one read access. The read-CO will accept the value returned by the first committed read access. This value is returned to the user-visible transaction by the read-TM.

To perform a logical write of \( x \) with a new value \( v' \), a write-TM again invokes a read-CO which reads a DM for \( x \) as in the case of a logical read. The read returns the value \( v \) along with the version-no to the write-TM. The write-TM increases the version-no by one and invokes a write-CO which further initiates write accesses to write the value \( v' \) along with the new version-no at all the DMs in the view of the write-TM containing some write-quorum.

When several invocations to the read- or write-COs fail to read a DM in their view, a read- or write-TM may invoke a view-CO in order to form a new view. The view-CO first generates a vid greater than its present vid and then initiates a process of reading data components from all the DMs in the system. However, only those DMs having a vid < new vid of the view-CO are eligible to respond. The view-CO keeps in its state all the data components it receives from the DM with highest view-no seen, and the set of the names of the DMs read denoted by \( d \). Once the view-CO is able to read access the minimum number of DMs requires for a read-quorum, it decides about the new view. Then it writes data components at all the DMs in its view or else it could repeat the above procedure to form the new view again.

Formally, we capture the replicated serial system discussed above as a four-tuple \( B = (\tau_B, \text{parent}_B, O_B, V_B) \), where \( \tau_B \) is a set of transactions in the system \( B \), \( \text{parent}_B \) defines the parent-child relationships discussed above, \( O_B \) is a set of physical data objects and \( V_B \) is the set of values returned. For each \( x \in I \), where \( I \) is a set of logical data objects, we define the following:

- \( \text{dm}(x) \) is a subset of \( O_B \).
- \( \text{acc}(x) \), a subset of the accesses in \( \tau_B \), is exactly the set of all accesses to objects in \( \text{dm}(x) \).
- \( \text{tm}_a(x), \text{tm}_u(x) \) are disjoint subsets of the non-accesses in \( \tau_B \), and \( \text{tm}(x) = \text{tm}_a(x) \cup \text{tm}_u(x) \).
- \( \text{co}(x) \) is a subset of the non-accesses in \( \tau_B \).
- \( \text{config}(x) \) is a legal configuration of \( \text{dm}(x) \).

The replicas for \( x \) will be associated with the members of \( \text{dm}(x) \) and the logical accesses to \( x \) will be managed by the automata associated with the members of \( \text{tm}(x) \). Also, in the transaction tree, \( T \in \text{acc}(x) \) iff \( \text{parent}(T) \in \text{co}(x) \), and \( T \in \text{co}(x) \) iff \( \text{parent}(T) \in \text{tm}(x) \). That is, the accesses to DMs for \( x \) are exactly the children of the COs for \( x \). Finally, we require that \( \text{dm}(x) \cap \text{dm}(y) = \emptyset \) for any two logical data objects \( x \) and \( y \).
The user-visible transaction in $B$ is the set of non-access transactions in $\tau_B$ that are neither in $\text{tm}(x)$ nor in $\text{co}(x)$ for all $x \in I$. We refer to accesses in $\text{acc}(x)$ for all $x \in I$ as replica accesses and to the remaining accesses (accesses to the non-replicated data objects) in $\tau_B$ as non-replica accesses.

In system $B$, each member of $\text{dm}(x), \text{tm}_r(x), \text{tm}_w(x)$ has a corresponding DM automaton, read-TM automaton, write-TM automaton for $x$. Each member of $\text{co}(x)$ has an associated read-CO, write-CO or view-CO automaton for $x$.

### 3.1. Transaction automata

The TMs and COs, modeled as I/O automata, are specified with the help of the operations given in Section 2. Here, we give the details of each I/O operation of read-TM, write-TM, view-CO, read-CO and write-CO automata.

The transition relation, denoted by $(s', p, s)$, have pre- and postconditions given separately for each I/O operation $p$ where $s'$ is the state before and $s$ is the state after $p$.

Each state of an automaton has a subset of the following components:
1. $\text{active}$ is a Boolean variable used to control the initiation of the CREATE operation.
2. $\text{view-change}$ is a Boolean variable used to control the initiation of read-COs in the changed view.
3. $\text{read-1}$ is a Boolean variable used to control the commit of read-TMs.
4. $\text{written-1}$ is a Boolean variable used to control the commit of write-COs.
5. $\text{read}$ is a Boolean variable used to control the commit of read-COs. All of the above Boolean variables are initially false.
6. $\text{read-req}$, $\text{write-req}$ and $\text{view-req}$ are the sets of requested read-, write- and view-COs, respectively.
7. $\text{aborted}$ is the set of aborted transactions.
8. $\text{read-act}$ and $\text{write-act}$ are the sets of requested read and write access subtransactions, respectively.
9. $\text{written-2}$ is the set of data objects updated.
10. $\text{read-2}$ is the set of data objects read. All of the above sets are initially empty.
11. $\text{data}$ consists of all the components of the data object. Initially, $\text{data}$ is undefined.

We refer to the view associated with the transaction $T$ by $\text{view}(T)$, similarly, vid as $\text{vid}(T)$ and version-no as $\text{version-no}(T)$.

#### 3.1.1. Read-TM

A read-TM for $x$ performs a logical read access to $x$ on behalf of the user-visible transactions.
Each state of a read-TM has components active, view-change, view-req, read, read-req, aborted and data used to define pre- and postconditions for any I/O operation. The I/O operations are defined as follows:

- **CREATE(T)**: To initiate a CREATE(T) operation, the initial state $s'$ in the precondition must have the Boolean variable active set to value false. When a read-TM $T$ is activated by a CREATE operation on the given precondition, the state of the automaton changes from $s'$ to $s$ and in the postcondition active becomes true.

  Precondition: $s'.active = false$
  Postcondition: $s.active = true$

- **REQUEST-CREATE(T')** where $T'$ is a view-CO: A view-CO $T'$ issues a REQUEST-CREATE only when there is a change in the communication network or some sites have failed or previous invocations to some read-COs have failed. The component view-req specifies all the view-CO transactions which are invoked by the read-TMs, and, therefore, is a subset of C0s. This component checks that the same transaction has not been requested before. The variable active = true indicates that CREATE(T) has occurred before. view-change = false acts as a control variable in the initiation of some read-COs at a later stage. On these preconditions, the automaton changes its state from $s'$ to $s$ and the component view-req becomes view-req $\cup \{T'\}$, i.e., it simply records the request.

  Precondition: a change in the communication network topology or some sites have failed or previous invocations to some read-COs have failed, i.e., some read-COs $\in s'.aborted$
  $s'.active = true$
  $T' \notin s'.view-req$
  $s'.view-change = false$
  Postcondition: $s.view-req = s'.view-req \cup \{T'\}$

- **REPORT-COMMIT(T', v)** where $T'$ is a view-CO: This I/O operation informs $T$ of the successful completion of its view-CO subtransaction $T'$. It has no preconditions. In the postconditions, data is the component used to specify the data returned. The variable view-change = true in the postcondition says that commit of $T'$ has occurred. It also controls the activation of a read-CO if it is to be invoked after the view-CO $T'$.

  Postcondition: if $s'.view-change = false$ then $s.data = v$
  $s.view-change = true$

- **REPORT-ABORT(T')** where $T'$ is a view-CO: It has no postconditions as it is not necessary for correctness purpose to remember the aborted view-COs.
Postcondition: no change

There are two REQUEST-CREATE($T'$) where $T'$ is a read-CO transaction but only one of them is initiated depending on whether a read-CO has been initiated without involving the initiation of a view-CO or after a view-CO has been committed under the read-CO.

- REQUEST-CREATE($T'$) where $T'$ is a read-CO: This I/O operation is to invoke a read-CO. The precondition no view-CO $T'' \in s'.\text{view-req}$ captures the fact that no view-CO $T''$ has been request-created before. active = true informs that CREATE($T$) has occurred before. read-req checks that REQUEST-CREATE operation for $T'$ has not been initiated before. read = false is used as a control variable which informs that read-CO $T'$ has been invoked but has not committed. view($T$) is assigned to view the component of $T'$ so that the read-CO can read any one of the DMs in its associated view. In the postcondition, read-req in state $s$ is updated to include $T'$.

Precondition: there is no view-CO $\in s'.\text{view-req}$
$s'.\text{active} = \text{true}$
$T' \notin s'.\text{read-req}$
$s'.\text{read} = \text{false}$
view($T'$) = view($T$)

Postcondition: $s'.\text{read-req} = s'.\text{read-req} \cup \{T'\}$

The following REQUEST-COMMIT operation is for a read-CO after the new view is formed.

- REQUEST-CREATE($T'$) where $T'$ is a read-CO: All the pre- and post conditions are as explained before except $s'.\text{view-change} = \text{true}$, which gives information about when a new view is formed.

Precondition: $s'.\text{active} = \text{true}$
$T' \notin s'.\text{read-req}$
$s'.\text{read} = \text{false}$
$s'.\text{view-change} = \text{true}$
view($T'$) = view($T$)

Postcondition: $s.\text{read-req} = s'.\text{read-req} \cup \{T'\}$

- REPORT-COMMIT($T'$, $v$) where $T'$ is a read-CO: This operation returns value $v$ to the parent transaction if $s'.\text{read} = \text{false}$, i.e., a REQUEST-CREATE of $T'$ has occurred but no REQUEST-COMMIT has occurred. $s.\text{read} = \text{true}$ in the postcondition says that REPORT-COMMIT of $T'$ has occurred.
Postcondition:  \( \text{if } s'.\text{read} = \text{false then } s'.\text{data} = v \)  
\( s'.\text{read} = \text{true} \)

- \text{REPORT-ABORT}(T') where \( T' \) is a read-CO: The postcondition simply records all the aborted read-COs.

Postcondition:  \( s.\text{aborted} = s'.\text{aborted} \cup \{T'\} \)

- \text{REQUEST-COMMIT}(T, v): This I/O operation returns the required value \( v \) to the user-visible transaction. \( \text{read} = \text{true} \) in the precondition informs that a COMMIT of a read-CO has occurred before. \( v = s'.\text{data}.\text{value} \in D_x \), the domain of \( x \), is the returned value. The other components are as defined before.

Precondition:  \( s'.\text{active} = \text{true} \)  
\( s'.\text{read} = \text{true} \)  
\( v = s'.\text{data}.\text{value} \)

Postcondition:  \( s.\text{active} = \text{false} \)

3.1.2. Write-TM

A write-TM performs logical write accesses on behalf of the user-visible transactions. Each state of a write-TM has active, view-change, view-req, read-1, read-req, write-req, written-1, aborted and data as state components used to define pre and post conditions for the following I/O operations:

- \text{CREATE}(T):

Precondition:  \( s'.\text{active} = \text{false} \)

Postcondition:  \( s.\text{active} = \text{true} \)

- \text{REQUEST-CREATE}(T') where \( T' \) is a view-CO:

Precondition:  a change in the communication network topology or some sites have failed or previous invocations to the read-COs have failed, i.e., some read-COs \( \in s'.\text{aborted} \) or some write-COs \( \in s' \) aborted  
\( s'.\text{active} = \text{true} \)  
\( T' \notin s'.\text{view-req} \)  
\( s'.\text{view-change} = \text{false} \)

Postcondition:  \( s.\text{view-req} = s'.\text{view-req} \cup \{T'\} \)
- REPORT-COMMIT\((T', v)\) where \(T'\) is a view-CO:

Postcondition: 
\[
\begin{align*}
\text{if } s'.\text{view-change} = \text{false} & \quad \text{then } s.\text{data} = v \\
& \quad s.\text{view-change} = \text{true}
\end{align*}
\]

- REPORT-ABORT\((T', v)\) where \(T'\) is a view-CO:

Postcondition: 
no change

- REQUEST-CREATE\((T')\) where \(T'\) is a read-CO:

Precondition: 
\[
\begin{align*}
\text{there is no view-CO} & \in s'.\text{view-req} \\
s'.\text{active} = \text{true} \\
T' & \not\in s'.\text{read-req} \\
s'.\text{read-l} = \text{false} \\
\text{view}(T') & = \text{view}(T)
\end{align*}
\]

Postcondition: 
\[
s.\text{read-req} = s'.\text{read-req} \cup \{T'\}
\]

- REQUEST-CREATE\((T')\) where \(T'\) is a read-CO:

Precondition: 
\[
\begin{align*}
s'.\text{active} & = \text{true} \\
T' & \not\in s'.\text{read-req} \\
s'.\text{read-l} & = \text{false} \\
s'.\text{view-change} & = \text{true} \\
\text{view}(T') & = \text{view}(T)
\end{align*}
\]

Postcondition: 
\[
s.\text{read-req} = s'.\text{read-req} \cup \{T'\}
\]

- REPORT-COMMIT\((T', v)\) where \(T'\) is a read-CO:

Postcondition: 
\[
\begin{align*}
\text{if } s'.\text{read-l} = \text{false} & \quad \text{then } s.\text{data} = v \\
& \quad s.\text{read-l} = \text{true}
\end{align*}
\]

- REPORT-ABORT\((T')\) where \(T'\) is a read-CO:

Postcondition: 
\[
s.\text{aborted} = s'.\text{aborted} \cup \{T'\}
\]

- REQUEST-CREATE\((T')\) where \(T'\) is a write-CO and \(\text{data}(T) = d\):

Precondition: 
\[
\begin{align*}
s'.\text{active} & = \text{true} \\
\text{view}(T') & = \text{view}(T) \\
T' & \not\in s'.\text{write-req} \\
s'.\text{read-l} & = \text{true} \\
\text{version-no}(T') & = s'.\text{data}.\text{version-no} + 1
\end{align*}
\]

Postcondition: 
\[
s.\text{write-req} = s'.\text{write-req} \cup \{T'\}
\]
• REPORT-COMMIT($T'$, $V$) where $T'$ is a write-CO:
  Postcondition: $s$.written-1 = true

• REPORT-ABORT($T'$) where $T'$ is either a write- or view-CO:
  Postcondition: $s$.aborted = $s'$.aborted $\cup \{T'\}$

• REQUEST-COMMIT($T$, $V$):
  Precondition: $s'$.active = true
               $v = \text{nil}$
               $s'$.written-1 = true
  Postcondition: $s$.active = false

3.1.3. View-CO
The purpose of a view-CO is to form a new view and update the value, version-no, vid, view-no and view at all the DMs in its view. That is, a view-CO models the view formation and update protocols.

A view-CO for $x$ has components active, read-act, write-act, data, read-2 and written-2.
• CREATE($T$):
  Postcondition: $s$.active = true

• REQUEST-CREATE($T'$) where $T'$ is a read access:
  Precondition: $s'$.active = true
               $T' \notin s'$.read-act
               $\text{vid}(T') = \text{vid}(T)$
  Postcondition: $s$.read-act = $s'$.read-act $\cup \{T'\}$

• REPORT-COMMIT($T'$, $V$) where $T'$ is a read access: The first postcondition says that $o(T')$ is included in the set read-2. The following postconditions say that the value and view in state $s$ are those associated with $v$.version-no and $v$.view-no if they are greater than what they were in state $s'$.
  Postcondition: $s$.read-2 = $s'$.read-2 $\cup \{o(T')\}$
                 if $v$.version-no $\geq s'$.data.version-no
                    then $
                       \{s$.data.version-no = $v$.version-no
                       s$.data.value = $v$.value\}$
                 if $v$.view-no $\geq s'$.data.view-no
                    then $
                       \{s$.data.view-no = $v$.view-no
                       s$.data.view = $v$.view\}$
• REQUEST-CREATE($T'$) where $T'$ is a write access and data($T'$) = $d$:

Precondition:  
$\exists'$.active = true  
$T' \not\in \exists'.write-act$  
view($T'$) = view($T'$)  
$d = \langle \{value(T'), version-no(T')\}, \{vid(T'), view-no(T'), view(T')\}\rangle$

Postcondition:  
$s'.write-act = s'.write-act \cup \{T'\}$

• REPORT-COMMIT($T'$, $V$) where $T'$ is a write access:

Postcondition:  
$s'.written-2 = s'.written-2 \cup \{o(T')\}$

• REPORT-ABORT($T'$) where $T'$ is a write access:

Postcondition:  
no change

• REQUEST-COMMIT($T$, $V'$): The last condition says that there exists a set of DMs $p$ having a read-quorum $r$ and a set of DMs $k$ having a write-quorum $w$ both belonging to the view formed by $T$.

Preconditions:  
$\exists'.active = true$  
$v = s.data.view$  
$\exists'.written-2 = view(T)$  
$p \in config(x).r$ and $k \in config(x).w$ such that $p$ and $k$  
both $\in view(T)$

Postcondition:  
$s'.active = false$

3.1.4. Read-CO

The purpose of a read-CO for $x$ is to perform a logical read access on $x$ which has any of the dm($x$) in its view containing a read-quorum for $x$. It calculates the current version-no and the associated value, vid, view-no, view of $x$ on the basis of the data returned by the read access it invokes.

A read-CO for $x$ has active, read-act, and read-2 as components.

• CREATE($T$):

Postcondition:  
$s'.active = true$

• REQUEST-CREATE($T'$) where $T'$ is a read access:

Precondition:  
$\exists'.active = true$  
$T' \not\in \exists'.read-act$  
view($T'$) = view($T$)

Postcondition:  
$s'.read-act = s'.read-act \cup \{T'\}$
• REPORT-COMMIT($T'$, $v$) where $T'$ is a read access:

Postcondition: $s'.read-2 = o(T')$

• REPORT-ABORT($T'$) where $T'$ is a read access:

Postcondition: no change

• REQUEST-COMMIT($T$, $v$):

Precondition:

\[ s'.active = true \]
\[ s'.read-2 = o(T') \]
\[ v = s'.data.value \]
\[ o(T') \in \text{view}(T) \]
\[ p \in \text{config}(x).r \text{ such that } p \in \text{view}(T) \]

Postcondition:

\[ s'.active = false \]

3.1.5. Write-CO

The purpose of a write-CO for $x$ is to write at all the DMs in its view having a write-quorum. A write-CO for $x$ has active, write-act and written-2 as components. The I/O operations and their pre- and postconditions are as follows:

• CREATE($T$):

Precondition:

\[ s'.active = false \]

Postcondition:

\[ s'.active = true \]

• REQUEST-CREATE($T'$) where $T'$ is a write access and $\text{data}(T') = d$:

Precondition:

\[ s'.active = true \]
\[ T' \notin s'.write-act \]
\[ \text{view}(T') = \text{view}(T) \]
\[ d = \{\text{value}(T), \text{version-no}(T)\} \]

Postcondition:

\[ s'.write-req = s'.write-req \cup \{T'\} \]

• REPORT-COMMIT($T'$, $V$):

Postcondition:

\[ s'.written-2 = s'.written-2 \cup \{o(T')\} \]

• REPORT-ABORT($T'$) where $T'$ is a write access:

Postcondition:

no change
- REQUEST-COMPLETE\((T, V)\):

Precondition:

\[
\begin{align*}
& s'.active = \text{true} \\
& v = \text{nil} \\
& s'.\text{written-2} = \text{view}(T) \\
& p \in \text{config}(x).w \text{ such that } p \in \text{view}(T)
\end{align*}
\]

Postcondition:

\[
\begin{align*}
& s.active = \text{false}
\end{align*}
\]

3.2. Data manager automaton

Each data manager is modeled as a serial object I/O automaton that accepts only read and write access operations. These I/O automata have CREATE\((T)\) as input and REQUEST-COMPLETE\((T)\) as output operations.

For each data manager automaton associated with the copies of the data object \(x\), the domain of values is \(D_x\).

Each state of the automaton has two components: active, which is a Boolean variable, and data \(d\), which is the most recently written value \(\in D_x\) by the write access subtransaction. Initially, active = false and data = initial data \(\in D_x\). Note that \(v = \text{nil}\) is an element of \(D_x\). The I/O operations along with their pre- and postconditions are as follows:

- CREATE\((T)\):

Precondition:

\[
\begin{align*}
& s'.active = \text{true} \\
& \text{vid}(o(T)) < \text{vid}(T) \\
& v = s'.\text{data}
\end{align*}
\]

or

\[
\begin{align*}
& s'.active = \text{true} \\
& o(T) \in \text{view}(T) \\
& \text{vid}(o(T)) = \text{vid}(T) \\
& v = s'.\text{data}
\end{align*}
\]

Postcondition:

\[
\begin{align*}
& s.active = \text{false}
\end{align*}
\]
• REQUEST-COMMIT(\(T, v\)) where \(T\) is a write access and data(\(T\)) = \(d\):

Precondition:

\(s'.active = true\)
\(o(T) \in \text{view}(T)\)
\(\text{vid}(o(T)) = \text{vid}(T)\)
\(v = \text{nil}\)

Postcondition:

\(s.data = d\)
\(s.active = false\)

4. Reorder serial correctness and replicated serial system \(B\)

We now prove that in the replicated serial system \(B\), read-TMs return proper values to the user-visible transactions. That is, each read-TM returns the value written by the previous logical write access in its view. Note that the last logical write in a view may be different than the last write in a system. Thus, a read in a view can return an old value. We first define a reordering of events in a schedule and give some of the definitions that are useful for describing logical accesses to the logical data objects in system \(B\) and for giving inductive arguments. Let \(x\) be any logical data object.

Reordering of events. Our nested transaction version of the virtual partition algorithm in a serial environment allows some top-level transactions to request the system to allow read-only subtransactions to be run "as in the past". That is, some transactions have the appearance of running in the past and therefore, reading of old values are allowed. This is in accordance with the virtual partition algorithm of [3] and [4]. Therefore, our nested transaction version of the virtual partition algorithm does not satisfy the external consistency condition of serial correctness as defined in [18] (and in Section 2.3) since the apparent serial order will have a different sequence of operations, for example, when projected on timestamp ordering. Therefore, the virtual partition algorithm cannot be proved to be "serially correct" as in the case of [11]. Thus, we have to define a weaker notion of correctness condition for the systems that permit reading of old values. To make the ordering of such transactions consistent in a serial environment, we move the events of such transactions to a point in the past consistent with the ordering of other transactions. Thus, the user receives responses now that are consistent with them having issued requests in the past and results reported to the user are valid. We formally define reorder as follows: Let \(\alpha\) be a schedule of system \(B\). Then, \(\beta = \text{reorder}(\alpha)\) is a well-formed schedule of system \(B\) where the events are reordered in the sense of the above. Later, we will observe that the schedule \(\beta\) satisfies the notion of weaker correctness condition as defined below.

Reorder serial correctness. In brief, the weaker notion of correctness, called, reorder serial correctness, can be stated as follows: A sequence \(\beta\) of actions is
reorder (not necessarily commutative operations) serially correct for transaction $T$ provided that there is some serial behavior $\gamma$ such that $\gamma|T$ is a reordering of $\beta|T$. This correctness condition allows reordering of some transactions by moving the events of those transactions in the past by assigning them out of order pseudotime intervals so that it appears that the user requested them earlier. The allowed reordering is in response to user requests and, therefore, is acceptable to the user. Thus, the user receives responses that are consistent with the ordering of other transactions such that it seems that they have been issued in the past.

**Access sequence.** The access sequence of $x$ in $\beta$ denoted by $access(x, \beta)$ is defined to be the subsequence of $\beta$ containing the CREATE and REQUEST-COMMIT operation for the members of $tm_\beta(x)$ and $tm_{\tau}(x)$ [11].

**Final-state.** The final-state of $x$ after $\beta$ denoted by $final-state(x, \beta)$ is defined to be either $value(T)$ if $REQUEST-COMMIT(T, v)$ is the last $REQUEST-COMMIT$ operation for a read-TM in $access(x, \beta)$, or $i_v$ if no $REQUEST-COMMIT$ operation for a write-TM occurs in $access(x, \beta)$.

**Final-view.** The final-view of $x$ after $\beta$ denoted by $final-view(x, \beta)$ is defined to be $view(T)$,

1. $REQUEST-COMMIT(T, v)$ is the last $REQUEST-COMMIT$ operation for a read-TM or write-TM $T$, and
2. $REQUEST-COMMIT(T', v)$ is the last $REQUEST-COMMIT$ operation for a view-CO $T'$ (child of $T$) invoked by $T$ in $access(x, \beta)$, or, $i_view$ (initial view) if no $REQUEST-COMMIT$ operation for a view-CO occurs under read- or write-TM in $access(x, \beta)$.

**View-equivalent transactions.** Two transactions are said to be view-equivalent if they maintain the same view of the DMs.

**Current-version-no.** The current-vn($x, \beta$) is defined to be the version-no($T$) if $REQUEST-COMMIT$ for $T$ is the last $REQUEST-COMMIT$ operation for a write access to $o(T) \in view(T)$ in $\beta$, otherwise current-vn($x, \beta$) = 0.

**Current-view-no and Current-vid.** Same as above.

**Lemma 1.** Let $x$ be a logical data item in $l$. Let $\beta$ be a schedule of the replicated serial system $B$. If $\beta$ ends in $REQUEST-COMMIT(T, v)$ with $T \in tm_\beta(x)$ then $v = final-state(x, \beta)$. Note that $\beta$ is a reordered schedule.

**Proof.** As explained earlier, each TM performs logical reads or writes at the DMs in its associated view having a read- and write-quorum. Each DM keeps user as well as control data items. These TMs work properly only if it can be shown that vid, view-no and the associated view, version-no and the value at all the DMs will have only the corresponding current versions at any stage of $\beta$ and after $\beta$. To prove the above lemma, we need to establish the following properties involving the control and the user data items.
The first property concerns view-no. It intuitively establishes that there will always exist some DMs whose view numbers are the current-view-no. Formally,

**Property 1.** For all DMs \( O \in \text{dm}(x) \), if \( d \) is the data component of \( O \) and \( d.\text{view-no} < \text{current-view-no}(x, \beta) \) then \( \exists \) some write-quorum \( q \) in final-view \( (x, \beta) \) such that for all DMs \( O' \in \text{final-view}(x, \beta) \) if \( d' \) is the data component of \( O' \) then \( d'.\text{view-no} > d.\text{view-no} \).

The following properties state that if the view-no (a control data item) is the same for any pair of DMs then the other control data items view and vid should also be same. For all the DMs having the same final view containing read- and write-quorum, the vid, view-no and view will have only the current versions of all these components. Similar statement holds for the version-no, a user data item. The value of the data object associated with the current-version-no in the final-view will be the final-state of the data object.

**Property 2.** For all pairs of DMs \( O_1, O_2 \in \text{dm}(x) \), let \( d_1 \) and \( d_2 \) be data components of \( O_1 \) and \( O_2 \), respectively. Then \( d_1.\text{view-no} = d_2.\text{view-no} \) implies that \( d_1.\text{view} = d_2.\text{view} \) and \( d_1.\text{vid} = d_2.\text{vid} \) and view contains read- and write-quorum.

**Property 3.** There exists a write-quorum \( q \in \text{config}(x).w \) and \( q \in \text{final-view} (x, \beta) \) such that for all DMs \( O \in \text{final-view}(x, \beta) \), if \( d \) is the data component of \( O \) then \( d.\text{view-no} = \text{current-view-no}(x, \beta) \), \( d.\text{view} = \text{final-view} (x, \beta) \) and \( d.\text{vid} = \text{current-vid}(x, \beta) \).

**Property 4.** There exists a write-quorum \( q \in \text{config}(x).w \) and \( q \in \text{final-view} (x, \beta) \) such that for all DMs \( O \in \text{final-view}(x, \beta) \), if \( d \) is the data component of \( O \) then \( d.\text{version-no} = \text{current-vn}(x, \beta) \).

**Property 5.** For all DMs \( O \in \text{dm}(x) \) in final-view \( (x, \beta) \), if \( d \) is the data component of \( O \) then \( d.\text{version-no} = \text{current-vn}(x, \beta) \) implies \( d.\text{value} = \text{final-state} (x, \beta) \).

To prove the above properties and the lemma, we use induction on the length of \( \beta \).

**Base Case.** Let \( \beta \) be the empty schedule. Since \( \beta \) is empty, all the components of the user data items as well as of the control data items will hold their respective initial values after \( \beta \). Thus, all the properties stated above hold trivially. Since \( \beta \) is empty, it does not end in a REQUEST-PREPARE \((T, v)\) with \( T \in \text{tm}_1(x) \) so the lemma also holds.
**Induction step.** Let $\beta = \beta' \iota$, where $\beta'$ is a subsequence of the schedule $\beta$ with REQUEST-COMMIT as the last operation for some $T \in \text{tm}(x)$. Let $\iota$ be a portion of the schedule after $\beta'$ starting and ending with the last CREATE and REQUEST-COMMIT operations, respectively, for some $T_\iota \in \text{tm}(x)$. $\iota$ also contains operations corresponding to the subtransactions of $T_\iota$. We have $\text{access}(x, \beta) = \text{access}(x, \beta') \text{access}(x, \iota)$, where $\text{access}(x, \iota)$ begins with the last CREATE operation in $\text{access}(x, \beta')$. Assume that the lemma holds for $\beta'$. By definition, $\text{access}(x, \beta)$ contains only CREATE and REQUEST-COMMIT operations for TMs in $\text{tm}(x)$. Also, since $\beta$ is a serial and well-formed schedule, $\text{access}(x, \iota) = (\text{CREATE}(T_\iota), \text{REQUEST-COMMIT}(T_\iota, \nu_\iota))$ for some $T_\iota \in \text{tm}(x)$ and $\nu_\iota \in \mathcal{V}$.

We note that all accesses in $\iota$ to DMs in $\text{dm}(x)$ are descendants of $T_\iota$. There are two possibilities for $T_\iota$: it is either a read- or write-TM. According to our nested transaction tree, a TM invokes a view-CO to form a new view before performing a logical access or performs a logical access in its associated view. Therefore, in order to show that the induction hypothesis holds for $\beta$, $T_\iota$ should preserve all the properties and the lemma in each of the following four possible cases:

1. $T_\iota$ is a read-TM and invokes no view-CO,
2. $T_\iota$ is a read-TM and invokes a view-CO,
3. $T_\iota$ is a write-TM and invokes no view-CO, and
4. $T_\iota$ is a write-TM and invokes a view-CO.

We know that whenever $T_\iota$ performs a logical read or write access, it invokes a read-CO first. This implies that, in all four cases, when $T_\iota$ requests to commit in $\iota$, at least one read-CO (a child of $T_\iota$) $\in \text{view}(T_\iota)$ must commit in $\iota$. Let $T'$ be the first read-CO that commits in response and let $\iota'$ be the portion of $\iota$ up to and including the commit for $T'$. Then the following holds.

**Observation 1.** If $s$ is the state of $T'$ just after a read access transaction commits to $T'$ in $\iota'$ then

(a) $s$.data.version-no and $s$.data.value contain the current-version-no and the associated final-state of the DM in $s$.read.

(b) $s$.data.view-no and $s$.data.vid contain the current-view-no, current-vid and the associated final-view in $s'$.read.

**Proof.** 1(a): A read-CO $T'$ reads the version-no and the associated value from a DM $\in \text{view}(T')$ upon commit of a read access. This version-no is the current-version-no since all the DMs $\in \text{view}(T')$ maintain only the current-version-no and the associated values. Also, since $T'$ is the first child that commits to $T_\iota$ and since $T'$ invokes only read accesses, the data components of the DM observed by $T'$ must be the same during $\iota'$ as after $\beta'$. Hence, Observation 1(a) holds. Observation 1(b) can also be proved in a similar manner. □
Since read-CO $T'$ returns the data read to $T_i$, the next observation asserts that the current-version-no and the associated final-state, current-view-no, current-vid and final-view in the state $s$ of $T_i$ after $\beta'\iota'$ are the same as they were after $\beta'$. 

**Observation 2.** The data component of the state of $T_i$ after $\beta'\iota'$ is $[[\text{current-vm}(x, \beta'), \text{final-state}(x, \beta')], [\text{current-vid}(x, \beta'), \text{current-view-no}(x, \beta'), \text{final-view}(x, \beta')]]$. 

**Proof.** Let $s'$ be the state of $T'$ when $T'$ issues its REQUEST-COMMIT operation. Then, Observation 1(b) and the induction hypothesis of Property 1 imply that $s'_{\text{read}}$ cannot contain a DM $\in \text{view}(T')$ having a read-quorum according to $\text{config}(x)\iota_r$ unless $s'_{\text{data.view-no}} = \text{current-view-no}(x, \beta')$. Also, since $T'$ is a read-CO, it commits only when $s'_{\text{read}}$ contains a DM $\in \text{view}(T')$ having a read-quorum. Therefore, by the induction hypothesis of Property 2, we know that $s'_{\text{read}}$ contains a DM $\in \text{view}(T')$ with $s'_{\text{data.view}} = \text{final-view}(x, \beta')$, $s'_{\text{data.vid}} = \text{current-vid}(x, \beta')$. By the induction hypothesis of Property 4, we know that there exists some write-quorum $w \in \text{final-view}(x, \beta')$ such that the states after $\beta'$ of all DMs in $\text{final-view}(x, \beta')$ have version-no $= \text{current-vm}(x, \beta')$. Since $\text{config}(x)$ is a legal configuration of $\text{dm}(x)$, the sets of DMs contributing to $r$ and $w$ must have a non-empty intersection. So by Observation 1(a), $s'_{\text{data.}}$ version-no $= \text{current-vm}(x, \beta')$. Therefore, by the induction hypothesis of Property 5, $s'_{\text{data.value}} = \text{logical-state}(x, \beta')$. When $T'$ commits the data component of the state of $T_i$ becomes $s'_{\text{data}}$. Also, once read-CO $T'$ commits, the data component of the state of $T_i$ never changes. Hence, Observation 2 holds. $\square$

Now, for Cases 2 and 3, we will show that $T_i$ does preserve all the properties stated and the lemma. Proofs for the other two cases are omitted.

**Case 2.** $T_i$ is a read-TM under which a view-CO commits before a read-CO.

Let $T'$ be the first view-CO that commits to $T_i$ and let $i'$ be a portion of $i$ up to and including that commit. Then the following holds.

**Observation 3.** If $s$ is the state of $T'$ just after a read access commits to $T'$ in $i'$, then $s_{\text{data.view-no}}$ contains the highest view-no and the associated view among DMs in $s_{\text{read}}$. Similar statements hold for the version-no and the associated value.

**Proof.** Similar to that of Observation 1 and hence, omitted. $\square$
After a new view is formed, a view-CO $T'$ invokes write accesses $T$ in $t$ with data$(T)$ to update all the DMs in view $(T')$. The following observation implies that the data component after $\beta$ of $T$ has current-version-no, and the associated final-state, current-view-no, current-vid and final-view and also the current-version-no and current-view-no after $\beta$ are greater than what they were after $\beta'$.

**Observation 4.** If $T$ is a write access invoked by the view-CO for $x$ in $t$ then data $(T) = \{[\{\text{current-vn}(x, \beta), \text{final-state}(x, \beta)\}, \{\text{current-vid}, \text{current-view-no}(x, \beta), \text{final-view}(x, \beta)\}] \text{ and current-view-no}(x, \beta) > \text{current-view-no}(x, \beta'), \text{current-vid}(x, \beta) > \text{current-vid}(x, \beta') \text{ and final-view}(x, \beta) \text{ will be the new final-view after } \beta'.

**Proof.** Analogous to that of Observation 2 and hence, omitted. □

We now show with the help of Observations 1–4 that all the properties and Lemma 1 hold in Case 2. Note that read-TM $T_t$ cannot request to commit until at least one of its read-COs commits after a committed view-CO by the definition of $T_t$. Let $T_t$ be the first read-CO that commits to $T_t$ and let $t''$ be the portion of $t$ up to and including the commit of $T_t$. We claim that all the properties of the induction hypothesis hold after $\beta' t''$. If $t''$ contains one or more commits of view-COs, then each view-CO $T_t$ cannot commit until it has received commit operation for all write accesses initiated by $T_t$ in view$(T_t)$. We now show that Property 1 holds after $\beta' t''$. Consider all the DMs that have view-no = current-view-no($x, \beta'$) after $\beta' t''$. They must have view = final-view($x, \beta'$) (Observation 4 and Property 1 of the induction hypothesis). Observation 4 implies that view-no($T_t) = \text{current-view-no}(x, \beta') + 1$ and view $(T_t) = \text{new final-view after } \beta'$. Therefore, Property 1 holds after $\beta' t''$.

We also know by Observation 4 that all write accesses for $x$ in $t''$ with view-no $\neq$ (note that $\neq$ is a special place holder symbol) have view-no greater than any view-no for $x$ in $\beta'$. Furthermore, since all such write accesses have the same view-no, the view and vid are also the same. Therefore, Property 2 continues to hold. Since $T_t$ is a read-CO, it cannot request to commit until it has received commit operation for read access in a view$(T_t)$ having a read-quorum. Since $T_t$ is a read-CO and by Observation 2, Properties 3–5 continue to hold after $\beta' t''$. Thus, the claim is true and hence, all the properties hold after $\beta' t''$.

By Observation 4, any view-CO that may commit in $t$ after $t''$ propagates new vid, view-no and view. So $T_t$ preserves all the properties of the induction hypothesis. Any read-CO that executes in $t$ after $t''$ cannot change the data components of the DMs. Since $T_t$ is a read-TM, final-state($x, \beta) = \text{final-state}(x, \beta')$ by Observation 2. Thus, Lemma 1 holds in Case 2.

**Case 3.** $T_t$ is a write-TM and invokes no view-CO.
Since $T_i$ is a write-TM, it invokes write-COs for $x$ to write at all the DMs in the view of $T_i$; then the following holds.

**Observation 5.** All write-COs $T_w$ for $x$ invoked in $i$ have $\text{version-no}(T_w) = \text{current-vn}(x, \beta') + 1$, $\text{value}(T_w) = \text{value}(T_i)$, $\text{view}(T) = \text{final-view}(x, \beta')$.

**Proof.** Let $s$ be the state of $T_i$ when it issues REQUEST-CREATE($T_w$) for some $T_w \in \text{view}(T_i)$. Then by the definition of a write-TM, $\text{version-no}(T_w) = s.\text{data.} \text{version-no} + 1$, $\text{value}(T_w) = \text{value}(T_i)$, and $\text{view}(T_w) = \text{final-view}(x, \beta')$. Since $T_i$ is a write-TM, it cannot invoke a write-CO $\in \text{view}(T_i)$ until at least one of its read-CO $\in \text{view}(T_i)$ commits. So, all the REQUEST-CREATEs for write-COs $\in \text{view}(T_i)$ in $i$ occur after $\beta'$. Therefore, by Observation 2, $s.\text{data.} \text{version-no} = \text{current-vn}(x, \beta')$ and $\text{view}(T_i) = \text{final-view}(x, \beta')$. Thus, Observation 5 holds. □

A write-CO in turn invokes write access transactions $T$. The data component (i.e., current-version-no and the associated final-state) associated with $T$ after $\beta$ is as given below. Also, the current-version-no after $\beta$ is greater than the current-version-no after $\beta'$.

**Observation 6.** If $T$ is a write access invoked in $i$ then $\text{data}(T) = [[\text{current-vn}(x, b), \text{final-state}(x, b)] , [\neq, \wedge, \wedge]]$ and $\text{current-vn}(x, \beta) > \text{current-vn}(x, \beta')$.

**Proof.** Since $T$ is invoked by some write-CO in its associated view for $x$, by Observation 5 and the definition of a write-CO, $\text{data}(T) = \{[\text{current-vn}(x, \beta') + 1, \text{value}(T_i)], \{\neq, \wedge, \wedge\} \}$. Since $\text{current-vn}(x, \beta') + 1$ is the only version-no for $x$ written in $i$ in view($T_i$), it follows from the definition of current-version-no that $\text{current-vn}(x, \beta) = \text{current-vn}(x, \beta')$. Since $T_i$ is a write-TM, logical-state($x, \beta$) = value($T_i$). Thus, Observation 6 holds. □

By Observation 6, the vid, view-no and view in the states of DMs for $x$ are not changed during $i$. Therefore, Properties 1–3 hold after $\beta$. Since $T_i$ is a write-TM, it cannot request to commit until at least one of its write-COs $\in \text{view}(T_i)$ commits. Let $T_w$ be the first write-CO that commits to $T_i$ and let $i''$ be the portion of $i$ up to and including the commit for a write access in final-view($x, \beta'$). By Observation 5, $\text{view}(T_w) = \text{final-view}(x, \beta')$ which equals final-view($x, \beta$). Therefore, by Observation 6, Properties 4 and 5 hold after $\beta''$.

We now show that all properties hold after $\beta''$. By Observation 6, any write-CO that may execute in $i$ after $i''$ merely propagates the new value and the version-no and any read-CO that may execute in $i$ after $i''$ cannot change the values at the DMs since they do not invoke write accesses. Therefore, all the properties hold after $\beta''$.

Since $T_i$ is not a read-TM, Lemma 1 holds vacuously.
5. Correctness

The correctness proof of the algorithm has a two-tier structure. First, we will observe that the replicated serial system $B$ under the virtual partition algorithm in a nested transaction environment is a structural extension of the non-replicated serial system $A$, and system $B$ is serially correct with respect to the system $A$. That is, the user-visible transactions cannot distinguish between the systems $A$ and $B$. In effect, the user-visible transactions in system $B$ will have the same executions as the corresponding user-visible transactions in system $A$ and the values observed by them are the same in both $A$ and $B$.

5.1. Correctness of replicated serial system $B$

Notice that in both systems $A$ and $B$, the transaction tree structures up to the level of user-visible transactions are the same. In the non-replicated serial system $A$, all non-access transactions are user-visible transactions. Also, if $T$ is a user-visible transaction in $B$ then there is a corresponding user-visible transaction in $A$. Each non-access transaction of $A$ is represented by the same automaton in both $A$ and $B$. If $T$ is a TM in system $B$ then there is a corresponding access transaction in system $A$. There are no transactions in $A$ which correspond to the coordinators and access transactions of system $B$. The objects that correspond to $tm(x)$ in $B$ will be the single copy read–write objects $o(x)$ in system $A$. The user-visible objects in both systems, modeled by the same automata, are the same. By comparing these two systems, it follows that systems $A$ and $B$ are such that the sets $\tau_b$ and $\tau_a$ contain $\tau_a$ and $\tau_a$, respectively, $V_b = V_a$, and parent$_b$, restricted to $\tau_a$, = parent$_a$. That is, system $B$ is a structural extension of the system $A$. The following two properties are satisfied by systems $A$ and $B$:

1. The corresponding user-visible transactions in both systems $A$ and $B$ are modeled by the same corresponding automata and, therefore, they will have the same schedule in the schedules of systems $A$ and $B$.
2. The user-visible data objects are modeled by the same automata in both systems $A$ and $B$. Therefore, the states of all user-visible data objects will be same in some executions of both $A$ and $B$. In other words, the sequence of REQUEST-COMMIT of write accesses on the user-visible data objects will be the same in some schedules of both systems $A$ and $B$. That is, the user-visible data objects will have the same schedules in both $A$ and $B$.

Now consider a schedule $\beta$ of system $B$ consisting of operations corresponding to user-visible transactions and user-invisible transactions, namely, TMs, COs and access subtransactions. We construct a sequence of operations $\alpha$ of system $A$ by

(i) removing from $\beta$ all the REQUEST-CREATE($T$), CREATE($T$), REQUEST-COMMIT($T, v$), REPORT-COMMIT($T, v$) and REPORT-
ABORT($T'$) operations for all transactions $T$ in acc($x$), and CO($x$) for all $x \in I$.

(ii) by interpreting $T$, a user-visible transaction in $B$, to stand for the corresponding user-visible transaction in $A$, and

(iii) replacing read-/write-TM operations by equivalent read/write access operations.

Note that the equivalent user-visible transactions as well as user-invisible data objects in the constructed sequence of operations $\alpha$ of $A$ and in the schedule $\beta$ of $B$ have the same order. Therefore, to establish the correctness of the serial system $B$ with respect to system $A$, we have to only show that $\alpha$ is indeed a schedule of the non-replicated serial system $A$, and that the read accesses in $\alpha$ return the same value as those returned by the corresponding read-TMs in $\beta$. Note that $\beta$ is a reordered schedule.

The proof of this is by induction on the length of $\beta$.

**Induction Hypothesis.** If $\beta$ is a schedule for system $B$, then the sequence $\alpha$ of operations obtained by the above construction is a schedule of $A$ and the read accesses in $\alpha$ return the same values as those returned by their counterparts: read-TMs in $\beta$.

**Base Case.** Consider $\beta$ to be the empty schedule. Therefore, $\alpha$ is also empty and, hence, $\alpha$ is a schedule of $A$.

**Induction step.** Assume that the hypothesis is true for all schedules of length $k$ or less of $\beta$. Let $\beta'$ be a schedule of length $k$, and let $\alpha'$ be the equivalent schedule of $A$. Let $\beta = \beta' \Pi_\beta$. There are five cases depending upon the type of operation $\Pi_\beta$.

**Case 1.** $\Pi_\beta$ is an I/O operation REQUEST-CREATE($T$), CREATE($T$), REQUEST-COMMIT($T$, $v$), REPORT-COMMIT($T$, $v$) or REPORT-ABORT ($T$) for replica accesses or coordinators.

Proof of Case 1. In this case, by construction, there is no operation in $\alpha$ which corresponds to $\Pi_\beta$ and therefore, $\alpha = \alpha'$ is a schedule of $A$. □

**Case 2.** $\Pi_\beta$ is a REQUEST-COMMIT for non-replica accesses.

Proof of Case 2. If $\Pi_\beta$ is a REQUEST-COMMIT operation of access transactions $T$ for non-replicated data objects (user-visible) then by construction the corresponding operation in $A$ is the same as $\Pi_\beta$. Also, by the induction hypothesis, $\alpha'|o = \beta'|o$. Since all user-visible objects and transactions are modeled by the same automaton in both systems $A$ and $B$, the fact that preconditions for
\( \Pi_{\beta} \) are satisfied after \( \beta' \) implies that they must also be satisfied after \( \alpha' \). Therefore, \( \alpha = \alpha' \Pi_{\beta}[T] \) is a schedule of \( T' \) and, by the composition lemma [18], \( \alpha \) is a schedule of \( A \). \( \square \)

**Case 3.** \( \Pi_{\beta} \) is an output operation of a TM.

**Proof of Case 3.** If \( \Pi_{\beta} \) is a REQUEST-COMMIT\((T, v)\) operation for some \( T' \in \text{tm}(x), x \in I \), then by construction the corresponding operation in system \( A \) is the same as \( \Pi_{\beta} \). Therefore, by the definition of \( A \), there is an access (transaction) to a single copy read–write object, which corresponds to transaction \( T \). Since \( \beta \) is a well formed, CREATE\((T)\) occurs in \( \alpha' \). Therefore, the precondition for REQUEST-COMMIT\((T, v')\) is satisfied in \( A \) for some \( v' \). We need to show that \( v = v' \) if \( T \) is a read-TM. By Lemma 1, we know that \( v = \text{final-state}(x, \beta') \). Therefore, by the induction hypothesis, \( v' \) is the value in the state of \( o(x) \) after \( \alpha' \). By construction, we know that \( \alpha'|o(x) \) (a subsequence of \( \alpha' \) consisting of operations of \( o(x) \) only) = access\((x, \beta') \) is the sequence of operations in \( \beta' \) on the logical data object \( x \). Therefore, the last write access in \( \alpha' \) on \( o(x) \) has the same value as the last write-TM on logical \( x \) in \( \beta' \). Hence, the value associated with the state of \( o(x) \) after \( \alpha' \) is final-state\((x, \beta') \). Therefore, we have \( v = v' \).

If there is no write-TM in access\((x, \beta') \), there are no write accesses to \( o(x) \) in \( \alpha' \) and, therefore, the value \( v \) of \( o(x) \) after \( \alpha' \) is \( l_x \), which is the final-state\((x, \beta') \).

Hence, the read-TM returns the same value in \( B \) as those returned by corresponding read access in \( A \). \( \square \)

**Case 4.** \( \Pi_{\beta} \) is an output operation of a user-visible transaction.

**Proof of Case 4.** If \( \Pi_{\beta} \) is an output operation of some user-visible transaction \( T \), then by construction, the corresponding operation in \( A \) is the same as \( \Pi_{\beta} \). By construction, the user-visible transaction \( T \) in \( B \) and the corresponding transaction \( T' \) in \( A \) are modeled by the same automaton. By the induction hypothesis, \( \alpha'|T' = \beta'|T \). Since the preconditions for \( \Pi_{\beta} \) are satisfied after \( \beta' \), they must also be satisfied after \( \alpha' \). Therefore, \( \alpha'[T] \) is a schedule of \( T' \) and by the composition lemma [18], \( \alpha \) is a schedule of \( A \). \( \square \)

**Case 5.** \( \Pi_{\beta} \) is an output operation of the scheduler.

**Proof of Case 5.** If \( \Pi_{\beta} \) is a CREATE\((T)\), COMMIT\((T)\), or an ABORT\((T)\), where \( T \) is a user-visible transaction, or \( T \) is an access to a user-visible data object, or \( T \in \text{tm}(x) \) for some \( x \in I \), then by construction the corresponding operation in \( A \) is the same as \( \Pi_{\beta} \). Using the fact that the schedule \( \beta \) is well-formed and by construction, the preconditions for \( \Pi_{\alpha} \) are satisfied for all three output operations (details are omitted). Therefore, \( \alpha \) is a schedule of \( A \). \( \square \)
Hence, in all cases, \( z \) is a schedule of the non-replicated system \( A \) and the read-TMs of \( B \) return the same values as those returned by equivalent read accesses of \( A \).

Thus, we have shown that the replicated serial system \( B \) is correct by showing that it is the same as the serial non-replicated system \( A \) in terms of user-visible transactions.

6. Open problems

In our discussion so far, we have dealt with serial systems only. However, to complete the correctness, we need to prove that the concurrent nested transaction replicated system under the virtual partition algorithm is also correct. In other words, we need to find a concurrency control algorithm which satisfies a weaker notion of correctness as defined in this paper so that our replication algorithm can be placed on the top of it.

To prove the correctness of the concurrent part of the algorithm in [11], it has been stated that any “correct” concurrency control algorithms (that satisfy the serializability theorem stated in [18]) can be combined with their replication algorithm to yield a correct system. In general, any concurrency control algorithm that provides atomicity (as defined in [18]) at the levels of replicas may work with their replication algorithm and will produce a correct system. However, we cannot state this with respect to our replication algorithm as the concurrency control algorithms in [7] satisfy the “serial correctness” as defined there. However, as stated, in our replication algorithm, a read may miss a preceding write in another view, and thus can produce an execution which is not atomic. Thus, our replication algorithm cannot be proven serially correct using the definition of [18]. Thus, we have defined a reorder serial correctness. Therefore, to prove the correctness of a concurrent system, which can be combined with our replication algorithm, we have to find a concurrency control algorithm that allows some read-only transactions to run “as if in the past” and then we should be able to prove the correctness of that algorithm using reorder serial correctness. We also need to state and prove a reorder serializability theorem based on reorder serial correctness, which should conclude that a sequence of operations is reorder serializable for the root transaction \( \tau_0 \). Thus, these are still open problems. Once having proved that, we can also simply state as in [11] that our replication algorithm can be combined with any concurrency control algorithm that satisfies the reorder serial correctness.

7. Conclusions

In this paper, we have modeled the virtual partition algorithm in a nested transaction environment using the I/O automaton model. We have given
precise specifications of each component of the algorithm that explain their behavior and interactions. These specifications are given using some pre- and postconditions. We have proved that the replicated serial system under the virtual partition algorithm is same as the non-replicated serial system as far as the user is concerned. Our proof treats issues of replication separately from the concurrency control algorithm. More importantly, we have identified that not all classes of replication algorithms can be proven "serially correct" in the sense of [11, 18]. We have defined a new correctness condition called reorder serial correctness. For future work, we would like to modify our algorithm for a reconfigurable quorum where the quorum changes dynamically. Another direction of research could be to design a proof of a virtual partition algorithm by not separating the replication issue from concurrency and recovery. We are working on the proof of correctness of the reorder serializability theorem.

References


