1990

Hybrid Value/Event Representation of Replicated Objects

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Shirley Browne

Report Number:
90-967
HYBRID VALUE/EVENT REPRESENTATION
OF REPLICATED OBJECTS

Bharat Bhargava
Shirley Browne

CSD-TR-967
March 1990
Hybrid Value/Event Representation of Replicated Objects *

Bharat Bhargava and Shirley Browne
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract. In this research, we define a new model for replicated objects in which an object's representation consists of a set of timestamped values plus a set of histories containing records of operation executions. A hybrid representation allows the system to trade off the higher concurrency and availability for write operations provided by an event-based representation with the higher efficiency for read operations provided by a value-based representation. We recast one-copy serializability theory in our hybrid model and develop a new correctness criterion that is directly applicable to hybrid execution schedules. We give two examples of the application of our model. For abstract data types, we show how the division of an object's state into subhistories, together with read/write concurrency control and replication control, can achieve the same degree of concurrency and availability as typespecific methods. For four categories of quorum methods, ranging from completely static to completely dynamic, we develop correctness conditions which show that the additional flexibility and adaptability provided by a hybrid representation may be combined with that provided by dynamic quorum methods.

*This research is sponsored by National Science Foundation Grant IRI-8821398.
1 Introduction

Background and motivation. Recent research in distributed systems has focused on making these systems adaptable [1, 5]. Adaptability encompasses both mutation adaptability, the ability to respond over the long-term to changes in hardware and in system specifications, as well as reconfiguration adaptability, the ability to adjust over the short-term to changes in conditions such as workload and failures [5]. The transaction model has proven to be a useful paradigm for building distributed systems [30, 22, 6]. In [5], a model for adaptable transaction processing is proposed and applied to distributed concurrency control and commit protocols, to network partitioning, and to server relocation.

Support for objects, or abstract data types, contributes to mutation adaptability by tailoring the data to the specific application, and by allowing the underlying implementation to be changed without affecting the application-level interface. The read/write transaction concept may be generalized so that a transaction consists of a set of typed operations on selected objects [29, 22]. Replication contributes to reconfiguration adaptability by improving the availability and performance of operations, but adds the problem of maintaining the mutual consistency of replicated copies. The use of quorums to maintain mutual consistency in spite of failures and network partitioning was originally proposed in [11]. The quorum method proposed in [11] is static – i.e., the set of permissible quorums for accessing an object, called a quorum assignment, is fixed. Recent research has extended the quorum model to dynamic quorum methods, in which quorum assignments may be changed, in order to allow quorum-based systems to better adapt to changing conditions [9, 15, 18]. Techniques for making dynamic quorum methods adaptable to the length and spatial extent of failures are described in [3].

The state of an object may be stored either as a value or as a history of events, or as a combination of the two representations, which we call a hybrid representation. In the value-based model, a write operation overwrites the previous value of an object. In the event-based model, a write operation appends its event to the object’s history. In a replicated database, the state of an object may be stored as a value at some sites, as a history at other sites, and by a hybrid representation at still other sites. With a quorum method in which a write quorum may consist of fewer than the total number of copies, the state of an object may be dispersed among the different sites, so that events from more than one site must be merged.
to reconstruct the object’s state.

An event-based representation allows for greater concurrency and availability, because write operations on the same object may be carried out concurrently. Event-based representation adversely affects the performance of operations that observe the state of an object, however, because such an operation incurs processing and communication overhead to collect and merge events into a value. Because the proportion of write and observer operations may change over time, an adaptable system should be able to adjust by converting between value- and event-based representations. This conversion capability is also desirable to allow switching between different data-processing algorithms. Some proposed algorithms for handling objects assume a value representation [10, 32], while others assume an event-based representation [14, 16]. Converting between different types of object representation is an example of state-conversion adaptability, as described in [5].

Splitting the history of events for an object into subhistories for different subsets of the operations can provide additional flexibility and adaptability. The subhistory for a given operation may be replicated at only some sites – for example, where updates are currently taking place. In the case of network partitioning, this subdivision allows capabilities for different operations to exist simultaneously in different partitions. Subdividing an object’s history also allows a read-write concurrency control mechanism to view each subhistory as a separate data item, thus achieving higher concurrency.

Previous theoretical work on one-copy serializability has been developed separately for value-based and event-based representation. Because a hybrid representation is often used in practice, new theory is needed for the hybrid model to allow correctness arguments to be given directly for the algorithms actually being used. When the model is more distant, we may be able to use it to prove correctness directly for the theoretical algorithm on which an implementation is based, but we must then consider many special cases to deal with differences between the theoretical algorithm and the actual implementation. Correctness arguments for adaptability algorithms that switch between different types of object representation will also need to use one-copy serializability theory for the hybrid model.

Contributions of this paper. In this paper, we define a new model for the hybrid value/event representation of objects. Operations on objects are classified as mutators and/or observers. The representation for an object consists of a set of timestamped values, together
with a set of histories in which the effects of individual mutator operations are recorded. We describe the execution of observer and mutator operations and discuss how the hybrid representation affects concurrency control.

We have recast one-copy serializability theory in terms of our hybrid model. A new definition of the \textit{READS-FROM} relation is given that reflects the one-to-many relationship needed for event-based representation and that incorporates the concepts of folding the events in a history into a value and of resetting the value of an object. We give a new definition of the one-serializability testing graph (1-STG) and prove a theorem that establishes the equivalence of one-copy serializability in our hybrid model to the existence of an acyclic 1-STG.

We extend to fit our hybrid model the definition of serialization-completeness for an object's history given in [14] for the event-based model. As this definition uses the concept of timestamped events, we first argue for the generality of commit timestamps that are generated by having a transaction manager read a local logical clock. We state a theorem (hereafter referred to as the serialization-completeness theorem) which gives a condition for one-copy serializability in terms of the serialization-completeness of histories seen by read operations.

A dependency relation that describes conflicts between pairs of operations is assumed to be given as part of an object's specification. In this paper, we show how the division of an object's state into subhistories, together with read/write concurrency and replication control, can achieve the same degree of concurrency and availability as type-specific methods. A procedure is given that translates the dependency relation specification for an object into a mapping from typed operations on the object to read and write operations on the subhistories. A method for mapping the initial and final quorum assignments for typed operations to read and write quorum assignments for the subhistories is also given. Using our mappings, we apply our hybrid one-copy serializability theory, developed in the context of read and write operations, to abstract data types. These mappings allow the adaptation of existing database software, including concurrency control, replication control, and commit protocols, for the support of objects. Using existing software would cut down on the software development time for object-oriented databases.

Lastly, we apply our serialization-completeness theorem to four categories of quorum-
based methods, ranging from completely static to completely dynamic. Conditions are given for each category, satisfaction of which in turn guarantees satisfaction of the serialization-completeness criterion in our theorem. The conditions telescope, in that an algorithm that satisfies the conditions for a given category will automatically satisfy the conditions for more dynamic categories. Hence, different categories of methods may be used concurrently in the same system. We have done other work on adapting quorum-based methods to the length and extent of failures [3] and to the use of communication-based recovery [4]. In this work also, which is cast in terms of hybrid representation of objects, we use our serialization-completeness theorem to prove correctness of our techniques.

2 Definitions

In this section, we define the components of a hybrid representation. We discuss the concept of dependencies between operations and explain how operations on a hybrid representation are executed using quorums.

An object, or abstract data type, consists of a state, or value, and a set of typed operations that are classified as mutators and observers. A mutator operation changes the state of an object. An observer operation reads the state of an object. An operation may be classified as both a mutator and an observer. An observer operation $p$ has an invocation part, denoted $inv(p)$. The execution part of a mutator operation $q$ is denoted simply by $q$.

Operations on the objects in the database are grouped into transactions, the execution of which is constrained to be one-copy serializable. That is, the concurrent execution in the replicated database must be equivalent to some serial execution on a one-copy database. A formal definition of one-copy serializability is given in section 3. The place of each transaction $T$ in the serialization order is given by its commit timestamp, $ts(T)$. The serialization time of an operation is the same as the commit timestamp of its issuing transaction. The rationale for and the generation of commit timestamps are discussed in section 4.

There is a binary dependency relation, denoted $\succ$, between invocations of operations and operation executions on the same object. We say $inv(p) \succ q$ if $inv(p)$ must see the effects of all earlier (in the serialization order) executions of $q$ in order for $p$ to return correct results. Operations $p$ and $q$ conflict if either $inv(p) \succ q$ or $inv(q) \succ p$. The dependency relation is
type-dependent and must be specified by the programmer of the abstract data type.

The following definitions are with respect to a serial one-copy execution. The state of an object \( \Theta \) as of serialization time zero, denoted \( \Theta_0 \), is its initial state, which is assumed to be given. The state of an object \( \Theta \) as of serialization time \( \sigma \), denoted \( \Theta_\sigma \), is its state as of some serialization time \( \tau < \sigma \), plus the application, in serialization order, of all mutator operations with serialization times greater than \( \tau \) and less than or equal to \( \sigma \). Let \( W = \{w_1, w_2, ..., w_m\} \) denote the set of typed operations classified as mutators, and \( R = \{r_1, r_2, ..., r_l\} \) the set of typed operations classified as observers. For a particular \( \alpha \subseteq W \), denote the substate of \( \Theta \) with respect to \( \alpha \) by \( \Theta_\alpha \). The substate \( \Theta_\alpha \) as of serialization time zero, denoted \( \Theta_{\alpha_0} \), is the initial state of \( \Theta \). The substate \( \Theta_\alpha \) as of serialization time \( \sigma \), denoted \( \Theta_{\alpha_\sigma} \), is the substate \( \Theta_{\alpha_\tau} \) for some serialization time \( \tau < \sigma \), plus the application, in serialization order, of all mutator operations in \( \alpha \) with serialization times greater than \( \tau \) and less than or equal to \( \sigma \). For an observer operation \( p \) such that \( \{q \mid inv(p) \supseteq q\} \subseteq \alpha \), it suffices for \( p \) to observe the substate \( \Theta_\alpha \) as of serialization time \( st(p) \).

For each object \( \Theta \), a family of zero or more reset operations, \( \{RESET.\alpha\} \), where each \( \alpha \) is a subset of \( W \), may be included as part of the object specification. The effect of \( RESET.\alpha \) is to write a new substate \( \Theta_\alpha \) with a serialization time equal to that of the \( RESET \) operation. If \( inv(p) \supseteq q \) for \( q \in \alpha \), then also \( inv(p) \supseteq RESET.\alpha \). A \( RESET \) operation is a mutator but not an observer, and \( RESET.\alpha \) cancels the effects of all previous mutator operations in \( \alpha \).

Let the sites in the distributed system be the set \( \{S_1, S_2, ..., S_n\} \). The hybrid representation of a replicated object \( \Theta \) consists of 1) a set of substate values \( \{\Theta^{\sigma,i}\} \), where \( \Theta^{\sigma,i} \) is a substate \( \Theta_{\alpha} \) that is stored as a timestamped value at site \( S_i \), and 2) a set of subhistories \( \{h^{\beta,i}\} \), where \( h^{\beta,i} \) denotes the record of execution of operations in \( \beta \subseteq W \) that is stored at site \( S_i \). An operation execution is recorded in a subhistory as an event consisting of a commit timestamp and the operation performed. The sets \( \{\beta\} \) for the subhistories \( \{h^{\beta,i}\} \) at a particular site \( i \) are disjoint – i.e., a mutator operation \( q \) is in at most one set \( \beta \) at a particular site. The sets \( \{\alpha\} \) for the substates stored at a particular site need not be disjoint.

A quorum for an operation is a set of sites whose participation suffices for carrying out that operation. A coquorum for a set \( Q \) of quorums is a set of sites that has nonempty intersection with every quorum in \( Q \). As in [16], each observer operation must access an
initial quorum, and each mutator operation must write its event to a final quorum. A quorum for an operation that is both an observer and a mutator must thus contain both an initial quorum and a final quorum. A quorum assignment for an object lists the permissible initial and final quorums for each operation.

The execution of the invocation part of an observer operation $p$ with serialization time $\sigma$ ($\sigma = \infty$ if the serialization time of $p$ is as yet unknown) is carried out as follows: The transaction manager (TM) process determines the serialization time of $\Theta^{\alpha,j}$ for some $\alpha \supseteq \{q \mid \text{inv}(p) \supset q\}$ at each site $S_j$ in an initial quorum for $p$. The TM then takes the $\Theta^{\alpha,\text{max}}$ with the maximum serialization time $\tau_{\text{max}}$ among all sites $S_j$ in the initial quorum, and applies, in serialization order, all operations $q$, such that $\text{inv}(p) \supset q$, with serialization times greater than $\tau_{\text{max}}$ and less than or equal to $\sigma$, from the histories $\{h^{\beta,j}\}$, where $\{q \mid \text{inv}(p) \supset q\} \subseteq \beta$, with one such $h^{\beta,j}$ obtained from each site $S_j$ in the initial quorum.

The execution of a mutator operation $q$ is carried out as follows: The TM writes the event for $q$ to a history $h^{\beta,j}$, where $q \in \beta$, at each site $S_j$ in a final quorum for $q$.

A fold operation is a physical but not a logical operation. To write a new value for a substate $\Theta^{\alpha_1,i}$, a fold operation is executed at a site $S_i$ as part of a transaction $T$ as follows: The TM takes a substate value $\Theta^{\alpha_2,i}$, where $\alpha_1 \subseteq \alpha_2$, as of serialization time $\tau < \sigma$, and applies, in serialization order, all operations in $\alpha_1$, with serialization times greater than $\tau$ and less than or equal to $ts(T)$, from a set of subhistories $\{h^{\beta,i}\}$, where the union of the $\beta$'s contains $\alpha_1$, and writes a new value for $\Theta^{\alpha_1,i}$ with serialization time $ts(T)$.

Concurrency control for the purpose of achieving serializability may be carried out only on the subhistories, with mutually exclusive access sufficing for the substate values. Hence, transactions consisting only of read and fold operations may execute free of the overhead of concurrency control.

To limit the size of subhistories, conditions under which events may be deleted from subhistories need to be specified. An event written by a mutator operation is written to stable storage at all sites in a final quorum for that operation as part of the issuing transaction's commit protocol. It will usually be desirable, however, to propagate events to non-quorum sites holding copies of the object. If values may be propagated as easily as events, then the condition for deleting events from a history $h^{\beta,i}$ stored at site $i$ is that events with serialization times less than $st(\Theta^{\alpha,i})$, where $\beta \subseteq \alpha$ for some $\alpha$ and $\Theta_{\alpha,i}$ is a substate value stored at site
may be discarded once $\Theta^{\alpha,i}$ has been recorded in stable storage. If subsequent observer operations needing to observe events in $\beta$ are free to use any substate $\Theta^{\alpha,i}$ for which $\beta \subseteq \alpha$, then the serialization time $\sigma$ up to which events may be discarded from $h^{\beta,i}$ must be taken to be the minimum over all such $st(\Theta^{\alpha,i})$. If the events themselves must be propagated, then the additional requirement is imposed that events with serialization times less than $\sigma$ may not be deleted until all sites holding copies of the object are known to have received and written to stable storage all such events. An event propagation protocol such as that in [33] or [14] may be employed to propagate this meta-information throughout the system.

3 Hybrid One-Copy Serializability

In this section, we restrict the operations on an object to be reads and writes. We restrict the representation of an object at each site to a single value plus a history to which write operations append their events. With these simplifications, we extend one-copy serializability theory, including the concepts of the $READS-FROM$ relation and acyclicity of a one-copy serializability testing graph, to the hybrid representation model. The resulting tools may be used to directly analyze and prove correctness of hybrid execution schedules. In section 5, we extend our results to abstract data types and illustrate the use of substates and subhistories.

One-copy serializability is the generally accepted correctness criterion for transaction processing in a replicated database system [2]. Informally, one-copy serializability means that the schedule of physical operations carried out on behalf of a set of transactions has the same effect as some serial execution of the same transactions on a one-copy database. The technical definition of one-copy serializability is based on the logical $READS-FROM$ relation. Two execution schedules are equivalent if they have the same $READS-FROM$ relations. A replicated data (RD) schedule is one-copy serializable if it is equivalent to a serial one-copy schedule. The $READS-FROM$ relation is defined differently in the value-based and event-based models. In the value-based model, the $READS-FROM$ relations are unique in that a transaction can read an object from at most one other transaction. In the event-based model, however, a transaction can read events for an object from a number of other transactions. We merge these ideas to obtain the definition of $READS-FROM$ for the hybrid model. In the following definitions, a logical object is denoted by a capital letter (e.g., $X$). A physical
Definition 3.1 Hybrid +READS-FROM relation.

$T_a$ resets $X$ if $T_a$ sets $X$ to a new value without first reading $X$.

In a replicated data schedule, $T_a$ writes $x_i$ if $T_a$ writes an event to $x_i$. $T_a$ folds $x_i$ if $T_a$ writes a value to $x_i$, but not as part of a logical RESET operation. $T_a$ resets $x_i$ if $T_a$ writes a value to $x_i$ as part of a RESET operation. $T_b$ reads-xi-from $T_a$ if $T_a$ writes, folds, or resets $x_i$, and $T_b$ reads the event or value written to $x_i$ by $T_a$. $(T_a, T_b) \in$ folds-xi-from-xj if $T_b$ uses a value or an event written to $x_j$ by $T_a$ to construct the value of $X$ it writes to $x_i$. $T_b$ reads-xi-from $T_a$ if $T_b$ reads-xi-from $T_a$, and $T_b$ uses the event or value written to $x_i$ by $T_a$.

The augmented folds-from relation, denoted +folds-from, is defined as follows:

$$(T_a, T_b) \in +folds-x_i-from-x_j \text{ if either}$$

1. $$(T_a, T_b) \in folds-x_i-from-x_j,$$

2. $$(T_a, T_c) \in +folds-x_k-from-x_j \text{ and } (T_c, T_b) \in folds-x_i-from-x_k \text{ for some } x_k.$$

The augmented READS-FROM relation, denoted +READS-FROM, is defined as follows:

$$(T_a, T_b) \in +READS-X-FROM \text{ if either}$$

1. $$(T_a, T_b) \in READS-X-FROM \text{ and } T_a \text{ writes or resets (not just folds) } X, \text{ or}$$

2. $$(T_c, T_b) \in READS-X-FROM, T_b \text{ uses the value written to } x_i \text{ by } T_c, (T_a, T_b) \in +folds-x_i-from-x_j \text{ for some } x_j, \text{ and } T_a \text{ writes or resets (not just folds) } x_j.$$

In a serial one-copy schedule, with transaction order given by $<$, $T_b \in +READS-X-FROM T_a$ if $T_a$ writes or resets $X$, $T_b$ reads $X$, $T_a < T_b$, and there is no $T_c$ such that $T_c$ resets $X$ and $T_a < T_c < T_b$.

A replicated data schedule is one-copy serializable in the hybrid model if it has the same +READS-X-FROM relation for every object $X$ as some serial one-copy schedule.
It is NP-complete to determine if an RD schedule in the hybrid model is one-copy serializable. NP-completeness can be shown by reducing the problem for the value-based model to that for the hybrid model. The problem for the value-based model is shown to be NP-complete in [27]. If resets are not allowed, however, then determination of one-copy serializability in the hybrid model can be done in polynomial time, as with the pure event-based model.

In the hybrid model, some method, such as commit timestamps, must be used by a read operation to determine the relative serialization order of values and events. The value with the most recent serialization time may be used as a starting point for constructing the up-to-date value. Events with serialization times earlier than this point are discarded, while events that occurred later are merged to produce the up-to-date value.

The condition for one-copy serializability that \(+\text{READS-FROM}\) relations be the same for the replicated data schedule as for some one-copy serial schedule essentially requires that a fold operation incorporate all events that precede it back to the most recent reset in the equivalent serial schedule. To show what can go wrong when this requirement is not met, consider the following replicated data schedule \(S\), where distributed two-phase locking is being used for concurrency control on physical copies with commit timestamps generated by reading a local logical clock [21] at the lock point, and where the fold operation \(f_c[x_1]\) incorporates the event written by \(T_a\) but not the event written by \(T_b\). Logical time increases from left to right.

\[
S = \begin{array}{cccc}
T_a & T_b & T_c & T_d \\
& w_a[x_1] w_a[x_2] & r_b[y_1] w_b[x_2] & w_b[x_3] & w_c[y_1] & f_c[x_1] & r_d[y_1] r_d[x_1] r_d[x_2]
\end{array}
\]

Because the commit timestamp read by \(T_d\) for the value of \(x_1\) is greater than that read for the events at \(x_2\), these events would be discarded, and \(T_d\) would use the value of \(x_1\) as the up-to-date value for \(X\). In order to preserve the equivalence of the \(+\text{READS-Y-FROM}\) relation, \(T_b\) must precede \(T_d\) (transitively through \(T_c\)) in any equivalent serial schedule, and hence we must have \((T_b, T_d) \in +\text{READS-X-FROM}\) for any such serial schedule. However, \((T_b, T_d) \not\in +\text{READS-X-FROM}\) for the replicated data schedule \(S\). Hence, \(S\) is not one-copy serializable.
The reason we need a separate definition for the \textit{folds-from} relation is that fold is strictly a physical operation, and not a logical operation like read and write. Although fold can be implemented by reading from a read quorum and writing the resulting value to a write quorum (indeed, this is exactly what is done in a pure value-based scheme), other implementations are possible. For example, a background event-propagation protocol, such as that described in [13], might be used, with each site being able to independently fold the events for its copy of an object into a value with a particular serialization timestamp as soon as it knows its copy is serialization-complete up to that time. The fold operation might also be used together with a value-date method [23] in which the local copy is guaranteed to have received all updates that occurred before some previous time.

Both the value-based and the event-based models can be obtained as special cases of the hybrid model. To derive the value-based model, consider every transaction that does a write operation on some object to also do fold operations on the same copies, with the fold operations immediately following the write operations. The \textit{READS-FROM} relations for the RD schedule, as defined in the hybrid model, will then be identical to the \textit{READS-FROM} relations as defined in the value-based model. To derive the event-based model, we consider RD schedules that have no fold operations. For such histories, the \textit{READS-FROM} and \textit{+READS-FROM} relations as defined in the hybrid model will be identical to each other and to the \textit{READS-FROM} relations as defined in the event-based model.

In the hybrid model, there is no notion of a fold operation for a serial one-copy schedule. Accordingly, we define the one-copy serializability testing graph (1-STG) that we will use for our proofs in terms of only read, write, and reset operations and the \textit{+READS-FROM} relation. Our definition of a 1-STG is adapted from that given in [7] for the value-based model.

\textbf{Definition 3.2} A one-copy serializability testing graph (1-STG) for an RD schedule $S$ is a graph $G$ with the nodes being the transactions in $S$ and the following edges:

1. \textit{WR} edges: If $(T_a, T_b) \in +\text{READS-X-FROM}$ for some $X$, there exists an edge $T_a \rightarrow_{WR} T_b$.

2. \textit{WS} and \textit{SW} edges: If $T_a$ writes or resets $X$ and $T_b$ (re)sets $X$, then $T_a \rightarrow_{WS} T_b$ or $T_b \rightarrow_{SW} T_a$ is in $G$. 

11
3. **RW edges:**

(a) If \((T_a, T_b) \in +READS-X-FROM\) where \(T_a\) writes or resets \(X\), \(T_c\) resets \(X\), and \(T_a \rightarrow T_c\) is in \(G\), then \(T_b \rightarrow T_c\) is in \(G\).

(b) If \(T_a\) writes or resets \(X\) and \(T_b\) reads \(X\), but \((T_a, T_b) \notin +READS-X-FROM\), then either there exists an edge \(T_b \rightarrow_RW T_a\), or there exists a \(T_c\) such that \(T_c\) resets \(X\) and edges \(T_a \rightarrow_WS T_c\) and \(T_c \rightarrow_WR T_b\).

The following theorem establishes the usefulness of the 1-STG:

**Theorem 3.1** An RD schedule \(S\) of committed transactions is one-copy serializable if and only if \(S\) has an acyclic 1-STG.

**Proof.** (\(\Rightarrow\)): Suppose \(S\) is one-copy serializable. Let \(S_{ser}\) be an equivalent one-copy serial history — i.e., \(S_{ser}\) has the same +READS-FROM relations as \(S\). We show that the serial order < must contain a 1-STG and hence, an acyclic 1-STG.

1. **WR edges:** If \((T_a, T_b) \in +READS-X-FROM\) for \(S\) and hence for \(S_{ser}\), then by the definition of +READS-FROM for a serial one-copy history, \(T_a < T_b\).

2. **SW and WS edges:** These edges exist because < is a total order.

3. **RW edges:**

(a) Suppose \(T_a\) writes or resets \(X\), that \((T_a, T_b) \in +READS-X-FROM\) for \(S\), and that \(T_c\) resets \(X\) with \(T_a < T_c\). Then \((T_a, T_b) \in +READS-X-FROM\) for \(S_{ser}\), so by the definition of +READS-FROM for one-copy serial schedules, \(T_a < T_b\). Then \(T_b < T_c\) because otherwise, \(T_a < T_c < T_b\) and \((T_a, T_b) \notin +READS-X-FROM\) for \(S_{ser}\) and hence for \(S\), a contradiction.

(b) Suppose \(T_a\) writes or resets \(X\) and \(T_b\) reads \(X\), but \((T_a, T_b) \notin +READS-X-FROM\) for \(S\) and hence for \(S_{ser}\). If \(T_a < T_b\), suppose there is no \(T_c\) such that \(T_c\) resets \(X\) and \(T_a < T_c < T_b\). But then \((T_a, T_b) \in +READS-X-FROM\) for \(S_{ser}\) and hence for \(S\), a contradiction.
Suppose \( S \) has an acyclic 1-STG. Let \( S_{ser} \) be the serial one-copy history with serial order \(<\) corresponding to any topological sort of the 1-STG. We must show that \( S_{ser} \) has the same +READS-FROM relations as \( S \).

1. Suppose \( (T_a, T_b) \in +READS-X-FROM \) for \( S \). Then \( T_a \rightarrow T_b \) is in the 1-STG, so \( T_a < T_b \) in \( S_{ser} \). Suppose \( T_a < T_c \) where \( T_c \) resets \( X \). Then \( T_a \rightarrow_{WS} T_c \) in the 1-STG, so \( T_b \rightarrow_{RW} T_c \) in the 1-STG and hence \( T_b < T_c \) in \( S_{ser} \). Thus, there is no \( T_c \) that resets \( X \) with \( T_a < T_c < T_b \), so \((T_a, T_b) \notin +READS-X-FROM \) for \( S_{ser} \).

2. Suppose \( (T_a, T_b) \notin +READS-X-FROM \) for \( S \).

   Case 1. \( T_b \rightarrow T_a \) is in the 1-STG. Then \( T_b < T_a \) in \( S_{ser} \) and hence \((T_a, T_b) \notin +READS-X-FROM \) for \( S_{ser} \).

   Case 2. There exists a \( T_c \) such that \( T_c \) resets \( X \), and \( T_a \rightarrow_{WS} T_c \) and \( T_c \rightarrow_{WR} T_b \) are in the 1-STG. This implies that \( T_a < T_c < T_b \) in \( S_{ser} \), so by the definition of +READS-X-FROM for serial one-copy schedules, \((T_a, T_b) \notin +READS-X-FROM \) for \( S_{ser} \).

\[ \Rightarrow \]

### 4 Serialization-Completeness of Histories

The values and events making up a hybrid representation must be timestamped so that a read operation may merge them correctly. Because write events should be applied in serialization, or commit, order, the timestamps for writes must reflect this order. In addition to making a hybrid representation practical, serialization-order timestamps are useful for developing a correctness criterion that is more directly applicable to hybrid execution schedules than the acyclic 1-STG criterion. In this section, we give such a criterion which specifies the set of events that must be reflected in the histories seen by read operations in order for an execution schedule to be correct.

We assume the existence of logical clocks [21]. In addition to having events that occur within a single process (i.e., transaction) be totally ordered, we stipulate that events occurring at a single copy of an object be totally ordered. The logical clock time at which a physical operation occurs is called the operation's observation time. Each transaction \( T \)
has a commit timestamp, \( ts(T) \), which is generated by a transaction manager and gives the serialization order of the transaction. Requiring the generation of commit timestamps is not restrictive, because any conflict-based concurrency control method for physical copies should be able to generate them.

Although we assume the existence of logical clocks and of commit timestamps for the sake of analysis, an actual implementation may have neither. For example, if distributed two-phase locking is used for concurrency control and intersecting read and write quorums and version numbers for replication control, then the serialization order is implicit in the lock point ordering, even if this order is not explicitly determined by the implementation. All we need to be able to argue is that commit timestamps could have been generated by the implementation if logical clocks had also been implemented.

An operation's serialization time is given by the commit timestamp of its issuing transaction. A physical operation in an RD schedule may be represented by a 4-tuple \((lt, st, op, x_i)\), where \( lt \) is the operation's observation time, \( st \) is the operation's serialization time, \( op \) is the type of operation, and \( x_i \) is the physical copy accessed. \( op \) may be one of read, write, fold, or copy. We have already discussed the fold operation. The copy operation copies an event from one copy of an object to another. For copy and fold operations, additional information is included in the schedule to indicate the events copied or folded. A fold operation may choose for its serialization time any time greater than or equal to the greatest serialization time of any event it uses to construct the new value. A copy operation chooses its serialization time to be the same as that of the copied event.

We assume that read, copy, and fold operations use only committed events. The reason for this requirement is to ensure that the resulting execution schedules will be recoverable (i.e., no committed transaction will need to be undone because it read from an uncommitted transaction that subsequently aborted) and free from cascading aborts (i.e., no transaction will need to be aborted because a transaction it read from aborted). Any mechanism that guarantees that uncommitted updates will not be visible (e.g., strict two-phase locking, shadowing) may be used to satisfy this assumption. If uncommitted updates may be written to disk, then an undo rule for local disk-based recovery, such as that described in [12], may also be needed to satisfy the assumption.

In the terminology of [14], what is required for a read or fold operation is that the
history used to construct a value for an object be *serialization-complete* with respect to the serialization time of the operation. The following definition is given in [14] for the event-based model:

**Definition 4.1** An object's local history, \( h_i \), is *serialization-complete up to \( \sigma \) if \( h_i \) contains every event for that object with a serialization time \( \leq \sigma \).

We modify this definition for the hybrid model as follows:

**Definition 4.2** Let \( S \) be an RD schedule, \( h_i \) a local history for object \( X \), and \( \sigma \) a given serialization time. Define \( \tau \leq \sigma \) to be the serialization time of the most recent reset operation for \( X \) in \( S \). Let \( f_c(x_k) \), issued by transaction \( T_c \), be a fold operation in \( h_i \) with the greatest serialization timestamp of any fold operation in \( h_i \), and let \( \rho = \max(\tau_c, \tau) \). If no such fold operation exists, let \( \rho = \tau \). Define \( h_i^- \) as follows:

\[
h_i^- = \{ (t_{op}(x_k), ts(T_c), op, x_j) \in h_i \mid \rho \leq ts(T_c) \leq \sigma \}
\]

Define \( h_i^+ \) as follows:

- If \( \rho = \tau \), then \( h_i^+ = h_i^- \)
- Else

\[
h_i^+ = h_i^- \cup \{ (t_{op}(x_k), ts(T_c), op, x_j) \in S \mid op \in \{write, reset\}, (T_c, T_c) \in folds-x_k-from-x_j \}
\]

Then \( h_i \) is *serialization-complete with respect to \( \sigma \) if \( h_i^+ \) contains a copy of the reset operation with serialization time \( \tau \) and a copy of every logical write event for \( X \) with serialization time \( st \) where \( \tau < st \leq \sigma \).

Definition 4.2 essentially says that a serialization-complete history contains a copy of every write event that follows the most recent reset, with the write event recorded either explicitly, or implicitly as part of a folded value. Note that for the event-based model, Definition 4.2 reduces to 4.1, because there are no reset or fold operations in this model. The following lemma shows that a serialization-complete history does not contain any event, either explicitly or implicitly, that precedes the most recent reset.

**Lemma 4.1** If \( h_i \) is *serialization-complete with respect to \( \sigma \), with \( \sigma, \tau, \) and \( \rho \) as defined in Definition 4.2*, then \( h_i^+ \) contains no event with serialization time \( st < \tau \) or \( st > \sigma \).
Proof. No event with serialization time \( st < \tau \) or \( st > \sigma \) is included in \( h^\tau_i \). If \( \tau = \rho \), then \( h^\tau_i = h^{-\tau}_i \). If \( \tau < \rho \), then because \( h_i \) is serialization-complete, there is a backward chain of fold operations with decreasing serialization times from \( \rho \) down to \( \tau \). Because fold operations are assumed to be well-behaved, only events with serialization times between \( \tau \) and \( \rho \) are added to \( h_i^- \) to construct \( h^\tau_i \). \( \diamond \)

The in-memory copy of the database is likely to diverge considerably from the disk copy, especially if the memory size is large. Previous models of recovery assume local disk-based recovery following a site failure—i.e., committed updates are redone from the local log. A more general model is needed that encompasses both disk-based and communication-based recovery for replicated data. Techniques for communication-based recovery are discussed in [4]. We consider a redo operation from disk to be a copy operation from the local disk—e.g., \((lt, st, copy(lt', st, op, xj), xj)\) for redo of an operation \( op \), where \( op \) is one of write, reset, or fold. To achieve a more general model, we extend the definition of an RD schedule. The RD schedule is revised to reflect the effects of failures and copy operations. Copy operations are revised so as to attribute their effects to the transaction that originally issued the copied event.

To facilitate the definition, we replace the observation time in the 4-tuple for a write, reset, or fold operation by an open interval with left endpoint equal to the observation time of the event and right endpoint equal to infinity. The right endpoint may be changed when the corresponding revised RD schedule is constructed from the RD schedule. We make the following assumptions concerning logical time and failures:

1. The logical time of a site failure is included in an RD schedule.

2. A logical clock increments across a failure.

One way to increment a logical clock across a failure is to have the logical clock time consist of an incarnation number plus a counter value. A recovering site reads its incarnation number from stable storage, increments it, and writes it back to stable storage. The counter for the logical clock is re-initialized to zero, and the logical time of the site failure and recovery may then be considered to be equal to the new incarnation number plus a counter value of zero. Depending on the concurrency control being used, some other technique for resetting a logical clock following a failure may be more desirable. For example, for a
timestamp-based method, communication with other sites may be used to synchronize the new logical clock value with the logical clock values at other sites. The justification for the extra communication in this case is that distributed timestamp-based concurrency control methods perform better if clocks at different sites have values that are fairly close to each other.

**Definition 4.3** To obtain a revised replicated data schedule from an RD schedule, we carry out the following steps:

1. All copy operations of the form \((lt, st, copy(lt', st, op, x_i), x_j)\) are replaced by \((lt, st, op, x_j)\).

2. If the schedule contains failures of a copy at observation times \(lt_1 < lt_2 < \ldots < lt_n\), then the right endpoint of the observation time interval for any write, fold, or reset operation with observation time \(lt\) is set to \(\min(\infty \cup \{lt_{fi} | lt_{fi} > lt\})\).

3. For each write, reset, or fold operation \(((lt_1, lt_2), st, op_1, x_i)\), set \(lt_2 = \min(lt_2, lt_1^'\)), where \(lt_1^'\) is the left endpoint of the observation time interval for the reset or fold operation \(((lt_1', lt_2^'), st', op_2, x_i)\) with the smallest serialization time \(st'\) that is greater than \(st\). \(lt_1^'\) is defined to be infinity if \(op_2\) does not exist. \(\triangleright\)

The effect of step 3 is to have a reset or fold operation bound the observation time intervals of preceding write operations. Note that a single physical operation may be represented by more than one 4-tuple if the history contains failures and the operation is redone by means of a copy operation. Also note that step 3 could cause an observation time interval to become empty by making its right endpoint less than its left endpoint.

**Definition 4.4** The validity interval for a physical write, reset, or fold operation is the union of the observation time intervals for all tuples representing that operation.

Before stating and proving our condition for one-copy serializability, we give an assumption and two useful lemmas.

**Concurrency Control Assumption.** Let \(S\) be a revised RD schedule. If the operations \((lt, st, read, x_i)\) for \(T_a\) and \(((lt_1', lt_2'), st', op, x_i)\) for some \(T_a\), where \(op\) is one of \(write, reset,\)
or fold, are in $S$ for copy $x_i$ with $s't < s$ and $lt \in (lt'_1, lt'_2)$, then $(T_a, T_b) \in reads-x_i-from$. Otherwise, $(T_a, T_b) \notin reads-x_i-from$.

The concurrency control assumption is very weak and does not ensure conflict serializability of the physical RD schedule. Indeed, we allow the observation time order of physically conflicting operations to be different from their serialization order. Such an execution may still be one-copy serializable, however, because the definition of one-copy serializability depends on the logical $+READS-FROM$ relation and not on the physical $reads-from$ and $folds-from$ relations. The concurrency control assumption merely states that if the observation time of a read operation on a copy falls within the validity interval of a write, fold, or reset operation $op$ that has an earlier serialization time than the read, then the read is guaranteed to observe the effects of $op$. Note that an event that occurs earlier than a read operation in logical time but with a later serialization time will not be observed, thus allowing read operations to serialize in the past.

The next lemma establishes an equivalence between the augmentation of a serialization-complete history and the $+READS-X-FROM$ relations.

**Lemma 4.2** If the history $h_i$ seen by a read operation issued by $T_b$ is serialization-complete with respect to $ts(T_b)$, then $(lt_{op[x_j]}, ts(T_a), op, x_j)$, where $op$ is write or reset, is in $h_i^+$ for some $x_j$ if and only if $(T_a, T_b) \in +READS-X-FROM$.

**Proof.** ($\Rightarrow$): Suppose $(lt_{op[x_j]}, ts(T_a), op, x_j)$ is in $h_i^+$. If $(lt_{op[x_j]}, ts(T_a), op, x_j)$ is in $h_i^-$, then $(T_a, T_b) \in READS-X-FROM$ and hence $(T_a, T_b) \in +READS-X-FROM$. If $(lt_{op[x_j]}, ts(T_a), op, x_j)$ is not in $h_i^-$, then by the definition of $h_i^+$ (in Definition 4.2), there exists a $T_c$ that folds a copy $x_k$ such that $(T_c, T_b) \in READS-X-FROM$ and $(T_a, T_c) \in +folds-x_k-from-x_j$. So by the definition of $+READS-X-FROM$ (in Definition 3.1), $(T_a, T_b) \in +READS-X-FROM$.

($\Leftarrow$): Suppose $(T_a, T_b) \in +READS-X-FROM$.

**Case 1.** $(T_a, T_b) \in READS-X-FROM$. Then $(lt_{op[x_j]}, ts(T_a), op, x_j)$ is in $h_i$ for some $x_j$ and, because $h_i^+$ is serialization-complete, $ts(T_a) \geq p$. Hence, $(lt_{op[x_j]}, ts(T_a), op, x_j)$ is in $h_i^+$. 

18
Case 2. \((T_a, T_b) \notin \text{READS-X-FROM}\). Then, by the definition of \(+\text{READS-X-FROM}\) (in Definition 3.1), there exists a \(T_c\) that folds a copy \(x_k\) such that \((T_c, T_b) \in \text{READS-X-FROM}\) and \((T_a, T_c) \in \text{+folds-x_k-from-x_j}\) for some \(x_j\). Because \(h_i\) is serialization-complete, \(ts(T_c) = \rho > \tau\) for \(\rho\) and \(\tau\) as defined in Definition 4.2. Hence, by the definition of \(h_i^+\), \((t_{op_a[x_j]}, ts(T_a), op_a, x_j)\) is in \(h_i^+\).

We can now give a condition for one-copy serializability in terms of serialization-complete histories.

**Theorem 4.1** Let \(S\) be a revised RD schedule and let \(S_{com}\) be its committed projection. If the history seen by every read operation in \(S_{com}\) is serialization-complete with respect to the serialization time of its issuing transaction, then \(S_{com}\) is one-copy serializable in commit timestamp order.

**Proof.** To prove that \(S_{com}\) is one-copy serializable, we need to show that \(G = \{T_a \rightarrow T_b \mid ts(T_a) < ts(T_b)\}\) contains a 1-STG for \(S_{com}\). Suppose not.

**Case 1.** Let \(T_a \rightarrow T_b\) be a missing WR edge. By the definition of 1-STG (Definition 3.2), \((T_a, T_b) \in \text{+READS-X-FROM}\) for some \(X\). If \((T_a, T_b) \in \text{READS-X-FROM}\), then by the Concurrency Control Assumption, \(ts(T_a) < ts(T_b)\) and so \(T_a \rightarrow T_b\) is in \(G\). Otherwise, by the definition of \(+\text{READS-X-FROM}\) for an RD schedule (Definition 3), \((T_c, T_b) \in \text{READS-X-FROM}\) for some \(T_c\) with \(ts(T_c) < ts(T_b)\) and there is some sequence of fold operations with increasing serialization times from \(T_a\) to \(T_c\). Hence, \(ts(T_a) < ts(T_b)\).

**Case 2.** WS and SW edges. These edges exist because the commit timestamp ordering is a total order.

**Case 3.** RW edges.

(a) Suppose \((T_a, T_b) \in \text{+READS-X-FROM}\) for \(S\), that \(T_c\) resets \(X\), and that \(ts(T_a) < ts(T_c)\), but that \(ts(T_c) < ts(T_b)\) - i.e., \(T_b \rightarrow T_c\) is a missing RW edge. Without loss of generality, assume \(T_c\) has the greatest commit timestamp possible. Because the history \(h_i\) seen by \(T_b\) is serialization-complete with respect to \(ts(T_b)\), by Lemma 4.1, no copy of
the write event for $T_a$ is in $h_i^+$. Then by Lemma 4.2, $(T_a, T_b) \notin +READS-X-FROM$, a contradiction.

(b) Suppose that $T_a$ writes or resets $X$, $T_b$ reads $X$, and that $(T_a, T_b) \notin +READS-X-FROM$ for $S_{com}$, but that $ts(T_a) < ts(T_b)$. Suppose there is no $T_c$ that resets $X$ such that $ts(T_a) < ts(T_c) < ts(T_b)$. Because $h_i$ is serialization-complete with respect to $ts(T_b)$, there is a copy of the write event issued by $T_a$ in $h_i^+$. Then by Lemma 4.2, $(T_a, T_b) \in +READS-X-FROM$, a contradiction. \( \Rightarrow \)

5 Serializability for Abstract Data Types

As an example of the application of hybrid representation, we show how to implement the higher concurrency and availability of ADTs using only read/write concurrency control and replication control on substates and subhistories. We give a procedure that maps the dependency relation specification for an object into a mapping from typed operations on the object to read and write operations on substates and/or subhistories. We also derive a mapping from initial and final quorum assignments for the typed operations to read and write quorum assignments for the subhistories. We prove that our mappings preserve correctness in the context of one-copy serializability of transactions consisting of typed operations, and that the mappings preserve the concurrency and availability properties of the abstract data type. Hence, concurrency control and replication control algorithms developed, proven correct, and implemented for the less complicated read/write model can be applied to ADTs by using our mappings.

5.1 Mapping of dependency relations

For ease of explanation, we represent dependency relations by means of a square table. There is an X in the box at the intersection of the row for operation $p$ and the column for operation $q$ if $inv(p) \triangleright q$. As an example, consider the following dependency relation for a 3-dimensional geometric object (Here *Lower left corner* returns the $x_{min}, y_{min}$ coordinates for the object):
A mutator operation $q$ may change the state of an object, but not have any other operations dependent upon it from a concurrency control point of view. Likewise, an observer operation $p$ may see the results of other operations but not depend on them. Hence, we distinguish between read and write operations that are subject to concurrency control, denoted by $ccread$ and $ccwrite$, and those that are subject only to mutual exclusion during the duration of the operation itself, denoted $read$ and $write$. A $ccread$ conflicts with a $ccwrite$. Two $ccwrite$ operations do not conflict. Ordinary $read$ and $write$ operations do not conflict with any other operations.

The procedure for converting an arbitrary dependency relation into a mapping from typed operations on the object to read and write operations on subhistories arranged in the form of a directed acyclic graph, or DAG, is as follows:

1. Classify every typed operation that has an X in its row as a $ccread$. Classify every typed operation that has an X in its column as a database $ccwrite$.

2. Let $X_0 = \{w_1, w_2, \ldots, w_m\}$ be the set of typed operations that are classified as $ccwrites$. Let $R = \{r_1, r_2, \ldots, r_l\}$ be the set of typed operations that are classified as $ccreads$. For each $r_i$, let $X_i = \{q | inv(r_i) \supset q\}$. (It is possible that $X_i = X_j$, $i \neq j$). For each $i$, let $\beta_i = X_i - \cup \{X_j | X_j \subset X_i\}$. $\beta_0$ represents the root subhistory in the DAG (i.e., the unique node with in-degree of zero). The parent-child relation in the DAG is defined
by using proper set inclusion: $\beta_i$ is a child of $\beta_j$ if $X_i \subset X_j$ and if there is no $X_k$ such that $X_i \subset X_k \subset X_j$.

3. We use $f$ to denote the function that maps typed operations to sets of database operations. $f$ is defined as follows:

$$f(p(x)) = \begin{cases} ccwrite(h^{\beta_p}), & \text{if } p \in \beta_k; \\ write(h^{\beta_p}), & \text{if } p \text{ writes an event but } p \notin \beta_k \text{ for any } k. \end{cases}$$

$$f(inv(p(x))) = \begin{cases} ccread(h^{\alpha_p}) | X_j \subseteq X_p \} \cup \\ \{read(\Theta^\alpha) \text{ for some } \alpha \text{ such that } X_p \subseteq \alpha \} \cup \\ \{ccwrite(h^{\beta}) \mid p \text{ needs to observe some event written by } q \text{ and } q(x) \text{ is mapped to } ccwrite(h^{\alpha_r}) \text{ or to } write(h^{\beta_r}) \} \end{cases}$$

For an abstract data type, either all the events written by an operation at a given copy are read by a subsequent operation invocation, or none are. To achieve the same consistency, we support grouping of the read and write operations that are mapped from a single $p$-event. All the read operations in $f(inv(p(x)))$ are performed atomically at a given copy by setting a mutual exclusion lock. The lock may be released immediately after performing the reads. Likewise, at commit time, a mutual exclusion lock is set at a given copy before the timestamp field is filled in for the operations in $f(p(x))$. Again, the lock may be released immediately after performing the writes. This mutual exclusion locking is local and is independent of any locking that is performed for purposes of concurrency control.

To illustrate the procedure, we give the derived composite object and the mapping for the example given in the previous section.

Example 5.1 $X_0 = \{\text{Rotate, Translate, Magnify, Change height}\}$

$R = \{\text{Area, Lower left corner, Volume, Height}\}$

$\beta_1 = \{\text{Magnify}\}$

$\beta_2 = \{\text{Rotate, Translate}\}$

$\beta_3 = \emptyset$

$\beta_4 = \{\text{Change height}\}$
We illustrate the need for ordinary read and write operations with two examples. The following minimal dependency relation is given in [17] for a semiquue (a queue from which items need not be removed in FIFO order):

### Example 5.2

<table>
<thead>
<tr>
<th>Ins</th>
<th>Rem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Our conversion procedure produces the following mapping:

<table>
<thead>
<tr>
<th>p(x)</th>
<th>f(inv(p(x)))</th>
<th>f(p(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ins(x)</td>
<td>write(h^0)</td>
<td></td>
</tr>
<tr>
<td>Rem(x)</td>
<td>ccread(h^0)</td>
<td>ccwrite(h^0)</td>
</tr>
</tbody>
</table>

In [17], the following two minimal dependency relations are given for a FIFO queue:
Example 5.3  

<table>
<thead>
<tr>
<th></th>
<th>Enq</th>
<th>Deq</th>
<th></th>
<th>Enq</th>
<th>Deq</th>
</tr>
</thead>
<tbody>
<tr>
<td>(DR1)</td>
<td></td>
<td></td>
<td>(DR2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deq</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

(DR2) is rather interesting in that the Deq operations does not depend on the Enq operation. This independence allows a Deq operation to execute concurrently with an Enq operation, provided the queue contains some committed items.

These typed operations and dependency relations map to the following DAGs and read/write operations:

\[
\begin{align*}
(DR1) & \quad p(x) & f(\text{inv}(p(x))) & f(p(x)) \\
& \quad \text{Enq}(x) & \text{ccwrite}(h^{\beta_0}) & f(p(x)) \\
& \quad \text{Deq}(x) & \text{ccread}(h^{\beta_0}) & \text{ccwrite}(h^{\beta_0}) \\
(DR2) & \quad p(x) & f(\text{inv}(p(x))) & f(p(x)) \\
& \quad \text{Enq}(x) & \text{ccread}(h^{\beta_{\text{Enq}}}) & \text{ccwrite}(h^{\beta_{\text{Enq}}}) \\
& \quad \text{Deq}(x) & \{\text{ccread}(h^{\beta_{\text{Deq}}}), \text{read}(h^{\beta_{\text{Deq}}})\} & \text{ccwrite}(h^{\beta_{\text{Deq}}})
\end{align*}
\]

5.2 Mapping of quorum assignments

For a replicated database, we need to assign read and write quorum sets to the nodes in the DAG of subhistories. We assume that we are given an assignment of initial and final quorum sets for the typed operations on the object and that this assignment satisfies the general quorum intersection invariant and achieves maximal availability. The general quorum intersection invariant is stated as follows [18]:

If \( \text{inv}(p) \succ q \), then each final quorum for \( q \) must intersect each initial quorum for \( p \).

By maximal availability, we mean that no operation can have its availability increased without decreasing the availability of some other operation. The assumption of maximal availability imposes certain constraints upon the possible quorum assignments. These constraints are characterized in the following two lemmas.

Lemma 5.1 If \( X_p = X_q \), then \( p \) and \( q \) have the same initial quorum assignments.
Proof. Suppose $X_p = X_q$ but the initial quorum assignments for $p(x)$ and $q(x)$ are different. Without loss of generality, there is a failure scenario in which a final quorum $S_p$ is available for $p$ but not for $q$. Let $t$ by any operation such that $\text{inv}(p) > t$. Since $X_p = X_q$, $\text{inv}(p) > t$. By the general quorum intersection invariant, any final quorum for $t$ must intersect $S_p$. But then the availability of $q$ can be increased by adding $S_p$ as an initial quorum for $q$ without decreasing the availability of any other operation, contradicting our assumption of maximal availability. $\blacksquare$

Lemma 5.2 If $\text{ccwrite}(h^{\bar{r}}) = f(p(x))$ and $\text{ccwrite}(h^{\bar{r}}) = f(q(x))$, then $p$ and $q$ have the same final quorum assignments.

Proof. Suppose the final quorum assignments for $p$ and $q$ are different. Without loss of generality, there is a failure scenario in which a final quorum $S_p$ is available for $p$ but not for $q$. By the general quorum intersection invariant, any initial quorum for $r'$ such that $X_r \subseteq X_{r'}$ must intersect $S_p$. Since $q \in X_r \subseteq X_{r'}$, this means that any initial quorum for any $r'$ such that $\text{inv}(r') > q$ must intersect $S_p$. But then the availability of $q$ can be increased by adding $S_p$ to the final quorum set for $q$ without decreasing the availability of any other operation, contradicting our assumption of maximal availability. $\blacksquare$

We are now able to give the mapping from initial and final quorums for typed operations on $x$ to read and write quorums for nodes in the composite object DAG for $X$. We assign read and write quorums for a node $h^{\bar{r}}$ as follows:

1. If there exists a $q$ such that $\text{ccwrite}(h^{\bar{r}}) = f(q(x))$, then use final quorums for $q$ as $\text{ccwrite}$ quorums for $h^{\bar{r}}$. By Lemma 5.2 this assignment is well-defined (i.e., it does not depend on the choice of $q$).

2. If there exists a $q$ such that $\text{ccwrite}(h^{\bar{r}}) = f(q(x))$, let $F_r$ be the union of the final quorums for $q$. Again by Lemma 5.2, the value assigned to $F_r$ is independent of the choice of $q$. If there is no such $q$, let $F_r$ be the empty set. Ccread quorums for $h^{\bar{r}}$ are defined to be the intersections of the initial quorums for $r(x)$ with $F_r$. By Lemma 5.1, this assignment is well-defined (i.e., independent of the choice of $r$) in the event that $X_r = X_p$ for some $p$. 25
3. For ordinary read and write operations, any one repository makes up a quorum.

For a replicated database, we impose the additional requirement that if an operation \( p \) is mapped to one or more \( ccreads \) and to either a \( ccwrite \) or a \( write \), then all the events read by a \( ccread \) are copied (using \( writes \)) to the write quorum being used for the write operation. This requirement is necessary to ensure that the histories constructed by read operations are serialization-complete. We call this requirement the **Event Copying Rule**. The events are copied to the appropriate nodes in the DAG. Note, however, that the set of sites to which the events read from a particular node are copied is a write quorum for \( f(p(x)) \), which is not necessarily a write quorum for that node.

### 5.3 Correctness and preservation of concurrency and availability

This section states three theorems which establish that our mappings from operations on abstract objects to read and write operations on subhistories preserve the correctness, concurrency, and availability properties of the abstract objects.

We first define serialization-completeness in the context of abstract data types.

**Definition 5.1** Let \( S \) be an RD schedule, \( h_i \) a local history for object \( X \), \( W \) the set of mutator operations for \( X \), \( p \) an observer operation, \( \alpha_p = \{ q \in W \mid inv(p) > q \} \), and \( st(p) \) the serialization time of \( p \). Define \( \tau \leq st(p) \) to be the serialization time of the most recent reset operation for \( X \) in \( S \). Let \( f_c[x_k] \), issued by transaction \( T_c \), be a fold operation in \( h_i \) with the greatest serialization timestamp of any fold operation in \( h_i \), and let \( \rho = max(ts(T_c), \tau) \).

If no such fold operation exists, let \( \rho = \tau \). Define \( h_i^- \) as follows:

\[
h_i^- = \{ lto, \langle op(x), op, x_j \rangle \in h_i \mid \rho \leq ts(T_o) \leq st(p) \}
\]

Define \( h_i^+ \) as follows:

If \( \rho = \tau \), then \( h_i^+ = h_i^- \)

else

\[
h_i^+ = h_i^- \cup \{ lto, \langle op(x_j), ts(T_o), op, x_j \rangle \in S \mid op \in W, (T_a, T_c) \in +folds-x_k-from-x_j \}
\]

Then \( h_i \) is serialization-complete for \( inv(p) \) if

1. \( h_i^+ \) contains a copy of the reset operation with serialization time \( \tau \),

2. \( h_i^+ \) contains a copy of every logical \( \alpha_p \)-event with serialization time greater than \( \tau \) and less than or equal to \( st(p) \),

26
3. for any r-event in $h_i^r$, where $r$ is an observer operation, $h_i^r$ contains every logical $\alpha_r$-event with serialization time greater than $r$ and less than or equal to $st(p)$, where $\alpha_r = \{ q \in W \mid inv(r) \succ q \}$.

We define one-copy serializability for ADTs in terms of serialization-completeness of the histories seen by observer operations. Our definition is roughly equivalent to those in [16] and [15].

**Definition 5.2** Let $S$ be a revised RD schedule and let $S_{com}$ be its committed projection. If the history seen by every observer operation $p$ in $S_{com}$ is serialization-complete for $inv(p)$, then $S_{com}$ is one-copy serializable in commit timestamp order.

We prove correctness by showing that any execution schedule that results from ordering conflicting read and write operations in timestamp order is one-copy serializable. It suffices to show that the set of events observed by $f(inv(p))$ is serialization-complete for $inv(p)$ for every observer operation $p$ executed in the abstract schedule.

**Lemma 5.3** The quorum intersection invariant holds for the derived ccread-ccwrite quorum assignments (i.e., any ccread quorum for $h^r$ intersects with any ccwrite quorum for $h^r$).

**Proof.** Let $Q_R$ be a ccread quorum for $h^r$, $Q_W$ a ccwrite quorum for $h^r$. Then $Q_W$ is a final quorum for some $q$, where $ccwrite(h^r) = f(q(x))$. $Q_R$ is the intersection of some initial quorum $IQ(r(x))$ with the union of the final quorums for $q$. Because $q \in X_r$, $inv(r) \succ q$, and hence initial quorums for $r$ intersect final quorums for $q$. Thus, $Q_R \cap Q_W = IQ(r(x)) \cap Q_W \neq \phi$. \( \Box \)

**Theorem 5.1** If $inv(p(x))$ is mapped to $f(p(x)))$, then the history observed by $f(p(x)))$ is serialization-complete for $inv(p)$.

**Proof.** Let $g$ be the history observed by $f(p(x)))$. Suppose $g$ contains a $q$-event $ccwrite(h^r)$ and let $t$ be such that $inv(q) \succ t$ and $g$ contains an earlier $t$-event. By induction, the set of events observed by $f(inv(q(x)))$ is serialization-complete and hence contains every earlier $t$-event. Because $Tr(q)$ copies every event observed by $\{ccread(h^r) \mid X_s \subseteq X_q\}$
and \( f(t(x)) = \text{ccwrite}(h^{\beta r}) \) for some \( X_s \subseteq X_q \), each of these \( t \)-events will be observed by \( f(\text{inv}(p(x))) \).

Next we show that if \( q(x) \) maps to \( \text{ccwrite}(h^{\beta r}) \) and \( \text{inv}(p) \succ q \), then \( g \) contains the event written by \( \text{ccwrite}(h^{\beta r}) \) if and only if \( \text{ts}(\text{Tr}(q)) < \text{ts}(\text{Tr}(p)) \). Because \( \text{inv}(p) \succ q, q \in X_p \) and \( X_r \subseteq X_p \). Hence \( \text{ccread}(h^{\beta r}) \in f(\text{inv}(p(x))) \). By Lemma 5.3, \( \text{ccread} \) and \( \text{ccwrite} \) quorums for \( h^{\beta r} \) intersect. Because physically conflicting operations are executed in timestamp order, \( \text{ccread}(h^{\beta r}) \) observes the event written by \( \text{ccwrite}(h^{\beta r}) \) if and only if \( \text{ts}(\text{Tr}(q)) < \text{ts}(\text{Tr}(p)) \).

The next theorem shows that our mapping from typed operations to read and write operations preserves the degree of concurrency achieved. The theorem says that if database operations for two transactions conflict, then there is a corresponding conflict between typed operations.

**Theorem 5.2** If \( \text{ccread}(h^{\beta i}) \in f(\text{inv}(p(x))) \) and \( \text{ccwrite}(h^{\beta i}) = f(q(x)) \), then \( \text{inv}(p) \succ q \) (i.e., \( p \) and \( q \) conflict).

**Proof.** Suppose \( \text{inv}(p) \not\succ q \). Then \( q \not\in X_p \). But \( q \in X_i \) because \( \text{ccwrite}(h^{\beta i}) = f(q(x)) \) and \( X_i \in \{X_s \mid X_s \subseteq X_p\} \) because \( \text{ccread}(h^{\beta i}) \in f(\text{inv}(p(x))) \). This implies that \( q \in X_p \), a contradiction. Hence \( \text{inv}(p) \succ q \). □

The next theorem shows that our mapping from initial and final quorums for typed operations to read and write quorums for database operations preserves maximal availability.

**Theorem 5.3** If initial and final quorums are available for \( p(x) \), then \( \text{ccread} \) and \( \text{ccwrite} \) quorums are available for all operations in \( f(\text{inv}(p(x))) \) and \( f(p(x)) \).

**Proof.** Suppose \( p \) is classified as a \( \text{ccwrite} \). Then \( p(x) \) is mapped to \( \text{ccwrite}(h^{\beta i}) \) for some \( i \) and the \( \text{ccwrite} \) quorums for \( h^{\beta i} \) are the same as the final quorums for \( p(x) \). Thus if a final quorum is available for \( p(x) \), a \( \text{ccwrite} \) quorum is available for \( \text{ccwrite}(h^{\beta i}) \).

Suppose \( p \) is classified as a \( \text{ccread} \). Then \( \text{inv}(p(x)) \) is mapped to \( \{\text{ccread}(h^{\beta r}) \mid X_q \subseteq X_p\} \). We claim that if an initial quorum is available for \( p(x) \) and \( X_q \subseteq X_p \), then an initial quorum is also available for \( q(x) \). Suppose an initial quorum \( S_p \) is available for \( p \), but not for \( q \). Consider any \( t \) such that \( \text{inv}(q) \succ t \). Then \( t \in X_q \subseteq X_p \) so \( \text{inv}(p) \succ t \). Hence, by the general
quorum intersection invariant, any final quorum for \( t \) must intersect \( S_p \). But then \( S_p \) can be added as an initial quorum for \( q \) without decreasing the availability of any other operation, contradicting our assumption of maximal availability. Thus, if an initial quorum is available for \( p(x) \), initial quorums are available for all \( q(x) \) such that \( X_q \subseteq X_p \), and hence all \( ccread \) quorums are available for all \( ccread(h^p) \) where \( X_q \subseteq X_p \).

6 Application to Quorum Methods

We apply our serialization-completeness theorem to four categories of quorum-based methods, ranging from completely static to completely dynamic. With a static method, objects have only active quorum assignments whereas with a dynamic method, objects have both active and backup quorum assignments. The four categories and their characteristics are as follows:

1. static – active quorum assignments fixed [11, 16, 25],
2. semi-static – active quorum assignments may be changed [19, 28],
3. semi-dynamic – backup quorum assignments fixed, but active quorum assignments may be changed under less stringent conditions on the availability of active quorums than for semi-static [9, 31].
4. dynamic – both active and backup quorum assignments may be changed [15, 18].

The references given for each category are examples of methods that fit into that category. For a selected sample of these examples, an explanation of why each method fits into its category, how it satisfies our correctness conditions for that category, and how it may be extended to use a hybrid representation is given in [8].

6.1 Static quorum methods

We first state a static quorum intersection requirement (abbreviated s.q.i.):

There is a fixed quorum assignment for each object such that every read quorum is a write coquorum. In any RD schedule \( S_{\text{com}} \) of committed transactions,

\[ 29 \]
every logical read, write, and reset operation maps to a set of physical operations at a quorum (write quorums are used for both write and reset). Furthermore, if \( st'' \) is the serialization time of a read operation \((lt'', \text{read}, x_i)\) and \( \tau = \max(0 \cup \{st | st < st'' \text{ where } st \text{ is the serialization time of a reset or fold operation at } x_i \}) \) whose validity interval covers \( lt'' \}), then \( lt'' \) must fall within the validity interval for any write or reset operation with serialization time \( st' \), where \( \tau \leq st' \leq st'' \), which used \( x_i \) as a member of its write quorum.

Note that this requirement involves both concurrency control and recovery. It can be satisfied, for example, by the combination of 1) intersecting read and write quorums, 2) physical conflict serializability at quorum sites, and 3) local disk-based recovery with redo of committed operations. In the absence of failures, the requirement is equivalent to intersecting read and write quorums with \( st' < st'' \) implying that \( lt' < lt'' \) at \( x_i \), where \( lt' \) is the observation time of the write. This is not quite as strong as conflict serializability at quorum sites, because reading in the past is allowed.

For recovery, the quorum intersection requirement makes each copy responsible for the operations in which it participated as a quorum member. To carry out recovery from a failure, a site must either explicitly redo committed operations on its copy of an object, or it may write a folded value to its copy with a serialization time greater than or equal to the lost operations. To ensure that the resulting schedule is one-copy serializable, however, fold operations must use serialization-complete histories, as stated in the next theorem.

**Theorem 6.1** If a revised RD schedule \( S_{\text{com}} \) of committed transactions satisfies the static quorum intersection requirement and if all fold operations use histories that are serialization-complete in \( S_{\text{com}} \) with respect to the serialization time of the fold operation, then \( S_{\text{com}} \) is one-copy serializable in commit timestamp order.

**Proof.** We need to show that the history seen by every read operation is serialization-complete. Let \( h \) be the history seen by a read operation \( rop(X) \), \( \sigma \) the serialization time of the read operation, and \( h^+ \) as in Definition 4.2. Let \( \tau_1 \) be the \( \tau \) in Definition 4.2. Consider the reset operation with serialization time \( \tau_1 \) or any write operation with serialization time \( st \), where \( \tau_1 \leq st \leq \sigma \). Designate this reset or write op as \( wop(X) \). Then the read quorum used by \( rop(X) \) and the write quorum used by \( wop(X) \) intersect in at least one copy \( x_i \).
Let \( \tau_2 \) be the \( \tau \) in the static quorum intersection requirement. Choose \( x_i \) so that \( \tau_2 \) has the maximum possible value. Because the quorum used by the reset intersects the quorum used by the read, \( \tau_2 \geq \tau_1 \).

Case 1. \( \tau_2 = \tau_1 \). Then by s.q.i., the observation time of \( rop(X) \) at \( x_i \) falls with the validity interval of \( wop(X) \) at \( x_i \), and hence \( wop(X) \) is in \( h \) and also in \( h^+ \).

Case 2. \( \tau_1 < \tau_2 \). Then \( \tau_2 \) is the serialization time of a fold operation at \( x_i \). If \( st > \tau_2 \), then by s.q.i., \( wop(X) \) is in \( h \) and also in \( h^+ \). If \( st \leq \tau_2 \), then because folds are serialization-complete, \( wop(X) \) is in \( h^+ \).

In either case, the required event is in \( h^+ \), and \( h \) is serialization-complete with respect to \( \sigma \). Then by Theorem 4.1, \( H_e \) is one-copy serializable in commit timestamp order. \( \triangleright \)

Fold operations may be carried out independently at the different sites in conjunction with an event propagation protocol [14]. It is then the responsibility of this protocol to ensure that any fold operation uses a serialization-complete history. Alternately, fold operations may be integrated with read and write operations. In this case, we restate the theorem in a slightly different form.

**Lemma 6.1** If the static quorum intersection requirement is satisfied and every fold operation uses a history read from a read quorum, then all fold operations use serialization-complete histories.

**Proof.** Let \( h \) be the history used by a fold operation with serialization time \( \sigma \) to construct a new value. Let \( rop(X) \) be the read operation done on behalf of the fold, and let \( \tau_2 \) be the serialization time of the starting point value used by the fold. The proof is by induction on \( \tau_2 \).

**Basis.** \( \tau_2 = 0 \). Let \( st \) be the serialization time of any write or reset operation \( wop(X) \), where \( 0 < st \leq \sigma \). Then \( \tau_2 \) is the \( \tau \) in Definition 4.2 and also the \( \tau \) in the s.q.i. for any \( x_i \) at which the read quorum for \( rop(X) \) and the write quorum for \( wop(X) \) intersect, because otherwise the fold operation would not have used \( \tau_2 \) as a starting point. By s.q.i., the observation time of \( rop(X) \) at \( x_i \) falls in the validity interval for \( wop(X) \) at \( x_i \), and hence \( wop(X) \) is in \( h \) and in \( h^+ \).
Inductive step. \( \tau_2 > 0 \). Let \( \tau_1 \) be the \( \tau \) in Definition 4.2. Then \( \tau_1 \leq \tau_2 \) by s.q.i., because otherwise the fold would not have used \( \tau_2 \) as a starting point. Let \( st \) be the serialization time of \( \text{wop}(X) \), where \( \tau_1 \leq st \leq \sigma \), and \( \text{wop}(X) \) is a reset or write operation.

Case 1. \( \tau_1 = \tau_2 \). Let \( x_i \) be a copy at which the read quorum for \( \text{rop}(X) \) and the write quorum for \( \text{wop}(X) \) intersect. Then by s.q.i., the copy of \( \text{wop}(X) \) at \( x_i \) is in \( h \) and in \( h^+ \).

Case 2. \( \tau_1 < \tau_2 \). If \( st > \tau_2 \), then by s.q.i., \( \text{wop}(X) \) is in \( h \) and in \( h^+ \). If \( st \leq \tau_2 \), then because \( \tau_2 \) is the serialization time of a fold operation which by induction is serialization-complete, \( \text{wop}(X) \) is in \( h^+ \).

Theorem 6.2 Let \( S_{\text{com}} \) be a revised RD schedule of committed transactions. If the static quorum intersection requirement is satisfied and every fold operation uses a history read from a read quorum, then \( S_{\text{com}} \) is one-copy serializable in commit timestamp order.

Proof. By Lemma 6.1, every fold operation uses a serialization-complete history. By Theorem 6.1, \( S_{\text{com}} \) is one-copy serializable in commit timestamp order.

6.2 Semi-static quorum methods

With a semi-static method, the active quorum assignment for an object may be changed by means of a control transaction that writes a new quorum assignment. The ability to change quorum assignments allows the system to enhance availability in response (e.g., by implementing dynamic voting [20]). The set of active quorum assignments used for an object in a particular RD schedule is totally ordered by the commit timestamp ordering of the corresponding control transactions. Let \( ts(QA_n) \) denote the commit timestamp of the control transaction that installs \( QA_n \). The next theorem gives conditions ensuring that any RD schedule satisfying them will be one-copy serializable.

Theorem 6.3 Let \( S_{\text{com}} \) be a revised RD schedule of committed transactions. If the following conditions are met, then \( S_{\text{com}} \) is one-copy serializable in commit timestamp order:

1. The s.q.i. requirement is satisfied for quorum assignment metadata as well as for user data.
2. A control transaction that changes a quorum assignment writes the new quorum assignment to at least an old read coquorum and an old write coquorum.

3. The control transaction that writes a new quorum assignment $QA_n$ also writes the history obtained from an old read quorum to the members of a new read coquorum. This action consists of either fold or copy operations at the new coquorum sites. An exception to this rule occurs if the control transaction resets the value of an object, in which case only the new value need be written to the new read coquorum.

4. Every fold operation uses a history obtained from a read quorum.

Before proving Theorem 6.3, we state a useful lemma.

**Lemma 6.2** Let $S_{com}$ be a revised schedule of committed transactions that satisfies the conditions of Theorem 6.9. Then

1. for any read operation $rop(X)$ with serialization time $st_r$, $ts(QA_n) < st_r \leq ts(QA_{n+1})$ if and only if $rop(X)$ uses $QA_n$.

2. for any write operation $wop(X)$ with serialization time $st_w$, $ts(QA_n) \leq st_w < ts(QA_{n+1})$ if and only if $wop(X)$ uses $QA_n$.

**Proof of Lemma 6.2.** ($\Rightarrow$): Suppose the serialization time $st$ of the operation $op(X)$ is between $QA_n$ and $QA_{n+1}$ (with equality possible at one of the endpoints, depending on whether the operation is read or write), but that the operation does not use $QA_n$.

**Case 1.** $op(X)$ uses $QA_m$ for some $m < n$. The set of sites accessed by the control transaction that installs $QA_{m+1}$ intersects the quorum used by $op(X)$.

**Case 1.1.** $op(X)$ is a read operation. If $op(X)$ is a read operation performed by the control transaction that installs $QA_{m+1}$, then $st = ts(QA_{m+1})$. If $op(X)$ is a user read operation, then $ts(QA_{m+1}) > st$, because otherwise, by s.q.i., $op(X)$ would have seen $QA_{m+1}$ and hence would not have used $QA_m$. In either case, $st \leq ts(QA_{m+1}) \leq ts(QA_n)$, a contradiction.
Case 1.2. $\text{op}(X)$ is a write operation. If $\text{op}(X)$ is a write operation performed by the control transaction that installs $QA_m$, then $st = ts(QA_m)$. If $\text{op}(X)$ is a user write operation, then $ts(QA_{m+1}) > st$, because otherwise, by s.q.i, $\text{op}(X)$ would have seen $QA_{m+1}$ and would not have used $QA_m$. In either case, $st < ts(QA_{m+1}) \leq ts(QA_n)$, a contradiction.

Case 2. $\text{op}(X)$ uses $QA_m$ for some $m \geq n + 1$.

Case 2.1. $\text{op}(X)$ is a read operation. Then $st > ts(QA_m) \geq ts(QA_{n+1})$, a contradiction.

Case 2.2. $\text{op}(X)$ is a write operation. If $\text{op}(X)$ is a write operation performed by the control transaction that installs $QA_m$, then $st = ts(QA_m)$. Otherwise, $st > ts(QA_m)$. Hence, $st \geq ts(QA_m) \geq ts(QA_{n+1})$, a contradiction.

($\Leftarrow$): Suppose $\text{op}(X)$ with serialization time $st$ uses $QA_n$. Clearly $st > ts(QA_n)$ for a read operation and $st \geq ts(QA_n)$ for a write operation.

Case 1. Suppose $\text{op}(X)$ is a write operation and $st = ts(QA_{n+1})$. Then $\text{op}(X)$ is a write operation performed by the control transaction that writes $QA_{n+1}$, and hence $\text{op}(X)$ uses $QA_{n+1}$, a contradiction.

Case 2. Suppose $st > ts(QA_{n+1})$. By condition 3, the quorum used by $\text{op}(X)$ intersects the set of copies to which $QA_{n+1}$ is written. Let $x_i$ be a copy in the intersection. Then $QA_{n+1}$ must have been written at $x_i$ after $\text{op}(X)$ was performed at $x_i$, because otherwise $\text{op}(X)$ would not have used $QA_n$. But then the transaction manager for the control transaction installing $QA_{n+1}$ would have chosen $ts(QA_{n+1}) > st$, a contradiction.

Proof of Theorem 6.3. Let $rop(X)$ be a read operation issued by transaction $T$ that uses $QA_n$ for $X$. We need to show that the history $h$ seen by $rop(X)$ is serialization-complete with respect to $ts(T)$. Let $\tau_1$ be the $\tau$ in Definition 4.2. Let $wop(X)$ be a reset or write operation with serialization time $st_w$, where $\tau_1 \leq st_w \leq ts(T)$. The proof is by induction on $ts(QA_n)$.

Basis. $ts(QA_n) = 0$. By Lemma 6.2, $wop(X)$ must also use $QA_n$. Hence, the write quorum used by $wop(X)$ intersects the read quorum used by $rop(X)$ in at least one copy $x_i$. 34
Let \( \tau_2 \) be the \( \tau \) in s.q.i. Choose \( x_i \) so that \( \tau_2 \) has the maximum value possible. If \( st_w > \tau_2 \), then \( \text{wop}(X) \) is in \( h \) by s.q.i. and in \( h^+ \). If \( st_w \leq \tau_2 \), then by induction on the number of fold operations at \( x_i \) since \( st_w \), \( \text{wop}(X) \) is in \( h^+ \).

**Inductive step.** \( ts(QA_n) > 0 \).

**Case 1.** \( st_w < ts(QA_n) \). The control transaction that installs \( QA_n \) reads a history from a read quorum as specified by \( QA_{n-1} \). By induction, this history, which is written to a new write quorum as specified by \( QA_n \), is serialization-complete with respect to \( ts(QA_n) \). Let \( \tau_2 \) be the \( \tau \) in s.q.i. Choose \( x_i \) so that \( \tau_2 \) has the maximum possible value. If \( ts(QA_n) > \tau_2 \), then \( \text{wop}(X) \) is in \( h \) by s.q.i. and in \( h^+ \). If \( ts(QA_n) \leq \tau_2 \), then by induction on the number of fold operations since \( ts(QA_n) \), \( \text{wop}(X) \) is in \( h^+ \).

**Case 2.** \( st_w \geq ts(QA_n) \). By Lemma 6.2, \( \text{wop}(X) \) must use \( QA_n \). Hence, the write quorum used by \( \text{wop}(X) \) intersects the read quorum used by \( rop(X) \) in at least one copy \( x_i \). Let \( \tau_2 \) be the \( \tau \) in s.q.i. Choose \( x_i \) so that \( \tau_2 \) has the maximum possible value. If \( st_w > \tau_2 \), then \( \text{wop}(X) \) is in \( h \) by s.q.i. and in \( h^+ \). If \( st_w \leq \tau_2 \), then by induction on the number of fold operations at \( x_i \) since \( st_w \), \( \text{wop}(X) \) is in \( h^+ \).

Hence, \( h \) is serialization-complete with respect to \( ts(T) \) for any such read operation, and by Theorem 4.1, \( S_{com} \) is one-copy serializable in commit timestamp order.

The control transaction will typically also write the new quorum assignment to all possible new quorum members, but this is not required for correctness. Alternative implementations of the fold operation are possible, in which case condition 4 may be replaced by the requirement that fold operations use serialization-complete histories.

### 6.3 Semi-dynamic quorum methods

With a semi-dynamic method, every object has two types of quorum assignments - (1) an active quorum assignment that is read by user transactions to determine what copies need to be accessed to carry out an operation, and (2) a backup quorum assignment that is referenced during failures to determine whether or not the object is available and, if
available, to determine the new active quorum assignment. The backup quorum assignments are assumed to be fixed, or static. A semi-dynamic method provides the availability of a static quorum method while allowing active quorum assignments to be tuned for better performance (e.g., to read-one write-all).

The improvement a semi-dynamic method achieves over a semi-static method is that active quorum assignments may be changed even when active read and write quorums are not available (i.e., even when the quorum intersection requirement we gave in section 6.2 for quorum assignment metadata, condition 2 in Theorem 6.3, cannot be satisfied). Such changes to active quorum assignments for different objects are coordinated by means of views. Failure to coordinate the changes can result in non-serializable executions. Informally, a view is a set of sites that can communicate with each other, together with the copies of objects residing at those sites. Each view has a unique view id which is monotonically increasing over time for views in which a given site participates. An object must have a backup quorum in a view in order for the object to be accessible in that view. Accessibility can be on a per operation basis (e.g., an object can be read accessible in a view but not write accessible). Accessibility can be determined and the new active quorum assignment can be set either at view formation time or on demand when a transaction attempts to access an object. A transaction must execute entirely within a single view. The view id is prepended to the logical clock time to obtain a commit timestamp. Hence, serialization in commit timestamp order implies serialization in order of view ids.

Before giving the theorem for semi-dynamic methods, we state a dynamic quorum intersection requirement (abbreviated d.q.i.):

There is a fixed backup quorum assignment for each object such that every backup read quorum is a backup write coquorum. Furthermore, any active write quorum that is assigned for an object must be a backup read coquorum (i.e., must include a backup write quorum).

**Theorem 6.4** Let $S_{com}$ be a revised RD schedule of committed transactions, each of which executes entirely within a single view. If the following conditions are met, then $S_{com}$ is one-copy serializable in commit timestamp order:

1. The dynamic quorum intersection requirement holds.
2. The control transaction that writes the first active quorum assignment for an object in a new view writes it to at least a backup read quorum and a backup write quorum.

3. The conditions for a semi-static method are satisfied within a given view, except that for the control transaction that writes the first active quorum assignment in a new view:

   (a) Condition 2 of Theorem 6.3 is replaced by condition 2 of this theorem.

   (b) For condition 3 of Theorem 6.3, the history is obtained from an old write quorum, as an old read quorum may not be accessible.

Before proving Theorem 6.4, we restate Lemma 6.2 for a semi-dynamic method as Lemma 6.3. The proof of Lemma 6.3 is almost the same as for Lemma 6.2, except that we must now consider the possibility that two different quorum assignments may belong to different views.

**Lemma 6.3** Let $S_c$ be a revised schedule of committed transactions that satisfies the conditions of Theorem 6.4. Then

1. for any read operation $rop(X)$ with serialization time $st_r$, $ts(QA_n) < st_r \leq ts(QA_{n+1})$ if and only if $rop(X)$ uses $QA_n$.

2. for any write operation $wop(X)$ with serialization time $st_w$, $ts(QA_n) \leq st_w < ts(QA_{n+1})$ if and only if $wop(X)$ uses $QA_n$.

**Proof of Lemma 6.3.** ($\Rightarrow$):

Case 1. The proof of Lemma 6.2 still holds if $QA_m$ and $QA_{m+1}$ are in the same view, or if $op(X)$ is performed by a control transaction. Suppose $QA_m$ and $QA_{m+1}$ are in different views and $op(X)$ is a user read or write operation with serialization time $st$. Let $ts(QA_1) \leq ts(QA_m)$ be the commit timestamp of the control transaction $C_1$ that installs the first active quorum assignment for the view $v_1$ to which $QA_m$ belongs. Let $C_2$ be the control transaction that installs $QA_{m+1}$ in view $v_2$. Because both $C_1$ and $C_2$ access a backup read quorum and a backup write quorum, the sets of copies accessed intersect. Hence, the viewid $v_2$ for $C_2$ is greater than $v_1$ for $C_1$, and $ts(QA_{m+1}) > st$, 37
because \( ts(QA_{m+1}) \) has higher order part equal to \( v_2 \) and \( st \) has higher order part equal to \( v_1 \).

Case 2. Same as proof of Lemma 6.2.

\((\Leftarrow\Rightarrow)\): The proof of Lemma 6.2 still holds if \( QA_n \) and \( QA_{n+1} \) are of the same view, or if \( \text{op}(X) \) is performed by a control transaction. Suppose \( QA_n \) and \( QA_{n+1} \) are of different views and that \( st > ts(QA_{n+1}) \). Let \( v_1 \) be the viewid for the view to which \( QA_n \) belongs and \( v_2 \) the viewid for \( QA_{n+1} \)'s view. Then the timestamp \( st \) has higher order part at least as great as \( v_2 \). By the same reasoning as above for \((\Rightarrow)\), \( v_2 > v_1 \). Hence, \( \text{op}(X) \) would not have used \( QA_n \) because the fact that \( ts(QA_n) \) has a different viewid from \( st \) would have been detected by the transaction manager, which would not have allowed the transaction to commit. \( \Rightarrow\)

Proof of Theorem 6.4. Using Lemma 6.3 instead of Lemma 6.2, the proof of Theorem 6.3 still holds. \( \Rightarrow\)

### 6.4 Dynamic quorum methods

With a completely dynamic method, the backup quorum assignment for an object may be changed by a control transaction. This additional flexibility allows both enhanced performance, provided by turning active quorum assignments, and enhanced availability, provided by changing backup quorum assignments in response to failures.

Theorem 6.5 Let \( S_{com} \) be a revised RD schedule of committed transactions each of which executes within a single view. If the conditions for a semi-dynamic method are met, along with the following additions, then \( S_{com} \) is one-copy serializable in commit timestamp order:

1. A control transaction that changes a backup quorum assignment writes the new backup quorum assignment to at least an old backup read quorum and an old backup write quorum.

2. If changing the backup quorum assignment causes the dynamic quorum intersection requirement to no longer be satisfied (because some active write quorum is no longer a backup read coquorum), then the active quorum assignment is also changed to satisfy the requirement.
Proof. We assume that a copy’s most recent backup quorum assignment is stored on stable storage and recovered in the event of a failure. Because of condition 1, any access to a backup quorum is guaranteed to use the most recent backup quorum assignment. Because condition 2 ensures that d.q.i. is still satisfied, the proof of Theorem 6.4 holds for Theorem 6.5.

6.5 Modifications for read-only or write-only access within a view

With a semi-dynamic or dynamic method, it is possible for an object to have a backup read quorum in a view, but not a backup write quorum, and consequently, be read-accessible but not write-accessible in the view. Or, the other way around, the object may be write-accessible but not read-accessible. In either case, condition 2 of Theorem 6.4 can no longer be satisfied. Instead, the new active quorum assignment will be written to a backup read quorum in the case of read-accessibility, and to a backup write quorum in the case of write-accessibility. For write-only accessibility, the new active quorum assignment need only satisfy d.q.i.

Let $QA_{n_1} QA_{n_1 + 1} \ldots QA_{n_2 - 1} QA_{n_2}$, with $n_2 > n_1 + 1$, be a sequence of quorum assignments for an object such that

i) the object is read-accessible (and possibly also write-accessible) in the view corresponding to $QA_{n_2}$, and

ii) the object is read-accessible (and possibly also write-accessible) in the view corresponding to $QA_{n_1}$, where the views corresponding to $QA_{n_1}$ and $QA_{n_2}$ intersect in at least a write coquorum for $QA_{n_1}$, with $n_1$ as great as possible. (If the object was only read-accessible in the view for $QA_{n_1}$ this write coquorum may be only for the write performed by the control transaction that installed $QA_{n_1}$.

Consequently, in the views corresponding to $QA_{n_1 + 1}$ through $QA_{n_2 - 1}$, the object is either write-accessible only, or read-accessible only but with no intersection in at least a write coquorum for itself with with $QA_{n - 2}$’s view. The object will not be both read- and write-accessible in any of these views, because then ii) above would be satisfied, and that view would be the view for $QA_{n_1}$. Then condition 3 of Theorem 6.3 is modified so that the history
is obtained from a set of copies that contains write coquorums for $QA_{n_1}$ through $QA_{n_2-1}$. These coquorums can be determined because the backup write quorums to which the write quorum assignments $QA_{n_1}$ through $QA_{n_2-1}$ were written intersect the backup read quorum accessed by the control transaction installing $QA_{n_2}$. A backup read quorum contains the necessary coquorums and will suffice for obtaining the history. In addition, s.q.i. must be satisfied within the entire sequence $QA_{n_1} Q_{A_{n_1+1}} \ldots Q_{A_{n_2}}$. Note that $n_1$ in condition ii) above will always be defined because $n_1 = 0$ will satisfy the condition if no later quorum assignment does.

We now show how the proof of Theorem 6.4 can be modified to guarantee correctness when we allow read-only or write-only access within a view.

**Lemma 6.4** Let $S_{com}$ be a revised schedule of committed transactions that satisfies the conditions of Theorem 6.4, but with the modifications for read-only and write-only access.

1. Let $QA_{n_1} Q_{A_{n_1+1}} \ldots Q_{A_{n_2-1}} Q_{A_{n_2}}$ be a sequence of quorum assignments such that the object is write-accessible in the views of $QA_{n_1}$ and $QA_{n_2}$. Then for any read operation $rop(X)$ with serialization time $st_r$, $ts(Q_{A_{n_1}}) < st_r \leq ts(Q_{A_{n_2}})$ if and only if $rop(X)$ uses a quorum assignment with timestamp greater than or equal to $ts(Q_{A_{n_1}})$ and less than $ts(Q_{A_{n_2}})$.

2. Let $QA_{n_1} Q_{A_{n_1+1}} \ldots Q_{A_{n_2-1}} Q_{A_{n_2}}$ be a sequence of quorum assignments such that the object is read-accessible in the views of $QA_{n_1}$ and $QA_{n_2}$. Then for any write operation $wop(X)$ with serialization time $st_w$, $ts(Q_{A_{n_1}}) \leq st_w < ts(Q_{A_{n_2}})$ if and only if $wop(X)$ uses a quorum assignment with timestamp greater than or equal to $ts(Q_{A_{n_1}})$ and less than $ts(Q_{A_{n_2}})$.

**Proof for read operation $rop(X)$.** ($\Rightarrow$): Suppose $ts(Q_{A_{n_1}}) < st_r \leq ts(Q_{A_{n_2}})$.

Case 1. Suppose $rop(X)$ uses $QA_m$ for some $m < n_1$. If $rop(X)$ is performed by the control transaction that installs $QA_{m+1}$, then $st_r = ts(Q_{A_{m+1}})$. If $rop(X)$ is a user read operation, consider the following cases:

Case 1.1 $QA_m$ and $QA_{n_1}$ are of different views with view ids $v_m$ and $v_{n_1}$, respectively. Then $v_m < v_{n_1}$ and because the view id component of $st_r$ is $v_m$, $st_r < ts(Q_{A_{n_1}})$. 

40
Case 1.2 \(QA_m\) and \(QA_{n_1}\) are of the same view. Then \(QA_m\) and \(QA_{m+1}\) are of the same view. Hence, \(st_r < ts(QA_{m+1}) \leq ts(QA_{n_1})\), because otherwise, by s.q.i., \(rop(X)\) would have seen \(QA_{m+1}\) and would not have used \(QA_m\).

In any case, \(st_r \leq ts(QA_{n_1})\), a contradiction.

Case 2. Suppose \(rop(X)\) uses \(QA_m\) for some \(m \geq n_2 + 1\). The proof is the same as for Lemma 6.2, \((\Rightarrow)\), Case 2.1, with \(n + 1\) replaced by \(n_2\).

\((\Leftarrow)\): Suppose \(rop(X)\) with serialization time \(st_r\) uses a quorum assignment with timestamp greater than or equal to \(ts(QA_{n_1})\) and less than \(ts(QA_{n_2})\). Clearly \(st_r > ts(QA_{n_1})\).

Suppose \(st_r > ts(QA_{n_2})\).

Case 1. Suppose \(st_r\) and \(ts(QA_{n_2})\) have different view ids \(v_r\) and \(v_{n_2}\), respectively. Then \(v_r > v_{n_2}\). But \(v_{n_2}\) is greater than the viewid for any quorum assignment that could have been used by \(rop(X)\), including \(v_r\), a contradiction.

Case 2. Suppose \(st_r\) and \(ts(QA_{n_2})\) are of the same view. Then the proof is the same as for Lemma 6.2, \((\Leftarrow)\), Case 2, with \(n + 1\) replaced by \(n_2\). \(\Leftarrow\)

Proof for write operation \(wop(X)\). \((\Rightarrow)\): Suppose \(ts(QA_{n_1}) \leq st_w < ts(QA_{n_2})\).

Case 1. Suppose \(wop(X)\) uses \(QA_m\) for some \(m < n\). If \(wop(X)\) is performed by the control transaction that installs \(QA_m\), then \(st_w = ts(QA_m)\). If \(wop(X)\) is a user write operation, consider the following cases:

Case 1.1 \(QA_m\) and \(QA_{n_1}\) are of different views with view ids \(v_m\) and \(v_{n_1}\), respectively. Then \(v_m < v_{n_1}\) and because the view id component of \(st_w\) is \(v_m\), \(st_w < ts(QA_{n_1})\).

Case 1.2 \(QA_m\) and \(QA_{n_1}\) are of the same view. Then \(QA_m\) and \(QA_{m+1}\) are of the same view. Hence, \(st_w < ts(QA_{m+1}) \leq ts(QA_{n_2})\), because otherwise, by s.q.i., \(wop(X)\) would have seen \(QA_{m+1}\) and would not have used \(QA_m\).

In any case, \(st_w < ts(QA_{n_1})\), a contradiction.

Case 2. Suppose \(wop(X)\) uses \(QA_m\) for some \(m \geq n_2 + 1\). The proof is the same as for Lemma 6.2, \((\Rightarrow)\), Case 2.2, with \(n + 1\) replaced by \(n_2\).
Suppose \( w_{op}(X) \) with serialization time \( st_w \) uses a quorum assignment with timestamp greater than or equal to \( ts(QA_{n_1}) \) and less than \( ts(QA_{n_2}) \). Clearly \( st_w \geq ts(QA_{n_1}) \).

Suppose \( st_w \geq ts(QA_{n_2}) \).

Case 1. Suppose \( st_w \) and \( ts(QA_{n_2}) \) have different view ids \( v_w \) and \( v_{n_2} \), respectively. Then \( v_w > v_{n_2} \). But \( v_{n_2} \) is greater than the viewid for any quorum assignment that could have been used by \( w_{op}(X) \), including \( v_w \), a contradiction.

Case 2. Suppose \( st_w \) and \( ts(QA_{n_2}) \) are of the same view. Then the proof is the same as for Lemma 6.2, (\( \Leftarrow \)), Case 2, with \( n + 1 \) replaced by \( n_2 \).

Theorem 6.6 Let \( S_{com} \) be a revised schedule of committed transactions that satisfies the conditions of Theorem 6.4, but with the above modifications for read-only and write-only access. Then \( S_{com} \) is one-copy serializable in commit timestamp order.

Proof. Let \( rop(X) \) be a read operation issued by \( T \) that uses \( QA_{n_2} \) for \( X \). Let \( QA_{n_1} \), \( QA_{n_1+1} \ldots QA_{n_2} \) be a sequence as defined above, unless \( ts(QA_{n_2}) = 0 \), in which case \( QA_{n_1} \) is the same as \( QA_{n_2} \). Let \( \tau_1 \) be the \( \tau \) in Definition 4.2. Let \( w_{op}(X) \) be a reset or write operation with serialization time \( st_w \), where \( \tau_1 \leq st_w \leq ts(T) \). The proof is by induction on \( QA_{n_1} \).

Basis. \( ts(QA_{n_1}) = 0 \). The proof makes the obvious modifications to the Basis step of the proof of Theorem 6.3, using Lemma 6.4 instead of Lemma 6.2.

Inductive step. \( ts(QA_{n_1}) > 0 \).

Case 1. \( st_w < ts(QA_{n_1}) \). Let \( QA_{n_0} \), \( QA_{n_0+1} \ldots QA_{n_1-1} \), \( QA_{n_1} \) be a sequence as defined above for right endpoint \( QA_{n_1} \). The control transaction that installs \( QA_{n_1} \) reads a history from write coquorums for \( QA_{n_0} \) through \( QA_{n_1-1} \). This history is serialization-complete by induction and is written to a write quorum for \( QA_{n_1} \).

Hence, Case 1 turns into the following Case 2.

Case 2. \( ts(QA_{n_1}) \leq st_w < ts(QA_{n_2}) \). By Lemma 6.4, \( w_{op}(X) \) must use a quorum assignment with timestamp greater than or equal to \( ts(QA_{n_1}) \) and less than \( ts(QA_{n_2}) \). Hence, the write quorum used by \( w_{op}(X) \) intersects the read quorum used by the control transaction that installs \( QA_{n_2} \) and writes the history read to
a new read coquorum. The rest of the proof for Case 2 is that same as for Case 1 of the Inductive step for Theorem 6.3, with $QA_n$ replaced by $QA_{n_2}$.

Case 3. $st_w > ts(QA_{n_2})$. $wop(X)$ must use $QA_{n_2}$, because otherwise $st_w \geq ts(QA_{n_2+1}) > ts(QA_{n_2}) \geq st_r$, a contradiction. The rest of the proof for Case 3 is the same as for Case 2 of the Inductive step for Theorem 6.3, with $QA_n$ replaced by $QA_{n_2}$. \[\square\]

7 Conclusions

We have proposed a model for hybrid value/event representation of objects that should be close to what is used in practice for many distributed applications. Consider, for example, a distributed banking application, where an account object might be stored as a balance plus histories of debits and credits, with the histories dispersed among different sites in the system. We have extended the definition of one-copy serializability to our hybrid model. Our definition is general in that the definitions for both the value-based and event-based models may be derived from it. We have explained how commit timestamps may be generated by a system using distributed conflict-based concurrency control, and we have given a sufficient condition for transactions to be serializable in commit timestamp order.

We have shown how a hybrid representation, with the history of events divided into different subhistories for different well-defined disjoint subsets of operations, may be used to implement abstract concurrency control and replication control policies in terms of read/write mechanisms. This mapping from the abstract object realm to the read/write model may be carried out without any loss in concurrency or availability for the abstract objects.

We have applied our correctness condition for the hybrid model to four categories of quorum methods, ranging from completely static to completely dynamic. Our results show that the additional flexibility and adaptability provided by a hybrid representation may be combined with that provided by dynamic quorum assignment changes.

For future work, a number of issues remain to be investigated. One issue involves strategies for partial replication of substates and subhistories. In this paper, we have assumed that the granularity of replication is the entire object, but a finer granularity would result in more flexibility. Another issue is how to incorporate the use of additional semantic information, beyond the consideration of conflicts between pairs of operations, to schedule operations.
A particularly interesting approach, described in [24], uses a history abstraction to express the ordering relationships among concurrent operations in an application-independent manner. An open question, raised in [26], is how to combine such semantic scheduling with read/write concurrency control. Isolating the use of semantic scheduling to particular subhistories should make this problem easier.
References


45


