# Timestamp-Based Orphan Elimination

Maurice P. Herlihy
Martin S. McKendry

31 December 1987
CMU-CS-87-108

An orphan in a distributed transaction system is an activity executing on behalf of an aborted transaction. Orphans are undesirable because they waste system resources and because they may observe inconsistent data. This paper proposes a new method for managing orphans created by crashes and by aborts. The method ensures that orphans are detected and eliminated in a timely manner, and it prevents them from observing inconsistent states. A major advantage of this method is simplicity: it is easy to understand, to implement, and to prove correct. An "eager" version of this method uses approximately synchronized real-time clocks to ensure that orphans are eliminated within a fixed duration, and a "lazy" version uses logical clocks to ensure that orphans are eventually eliminated as information propagates through the system. The method is fail-safe: unsynchronized clocks and lost messages may affect performance, but they cannot produce inconsistencies or protect orphans from eventual elimination.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539, and in part by the USAF Rome Air Development Center under contract number F30602-84-C-0063 and the U.S. Naval Ocean Systems Center under contract number N66001-83-C-0305.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, RADC, NOSC, or the US Government.

A preliminary version of this paper appeared in the proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems, January 1986 [20].

## 1. Introduction

A *distributed system* consists of multiple computers (called sites) that communicate through a network. A *distributed program* is one whose components reside and execute at multiple sites in a distributed system. The physical components of a distributed system can fail independently: sites can crash, and communication links can be interrupted. Nonetheless, the data managed by a distributed program may be subject to consistency constraints that must be preserved in the presence of failures and concurrency. Such constraints can apply not only to individual pieces of data, but also to distributed sets of data. For example, a distributed banking system might be subject to the constraint that the books balance: money is neither created nor destroyed, only transferred from one ledger to another. A widely-accepted approach to ensuring consistency is to make the activities that manage the data *atomic*. Atomicity encompasses two properties: serializability and recoverability. *Serializability* [17] means that the execution of one activity never appears to overlap (or contain) the execution of another, while *recoverability* means that the overall effect of an activity is all-or-nothing: it either succeeds completely, or it has no effect. Atomic activities are called *transactions*.

Well-known techniques such as two-phase locking [3, 15] and commit protocols [5, 21] ensure atomicity for committed transactions. Nevertheless, these techniques make few guarantees about *orphans*, which are activities executing on behalf of aborted transactions. Orphans may be created by site crashes, or, in a nested transaction system [19, 15], when a transaction unilaterally aborts a nested subtransaction. Orphans are undesirable because they waste resources: not only do they consume processor cycles, they can introduce spurious delays and deadlocks by holding locks needed by non-orphans.

Orphans are also undesirable because they can observe inconsistent data. For example, in a system based on two-phase locking, a site crash and recovery may release a transaction's locks before that transaction has finished acquiring locks at other sites, an inadvertent violation of the two-phase locking discipline. Such inconsistencies may be of little concern in conventional database systems, where a transaction does not interact with the outside world until it commits. In a general-purpose distributed system, however, such inconsistencies may be more problematic. For example, the Argus system [10, 26] supports a methodology in which user-defined atomic data types are implemented by a mixture of atomic and non-atomic data types at a lower level. In the absence of an orphan management scheme, the implementor of such a type must take care that transient inconsistencies in the atomic components of the implementation do not produce permanent inconsistencies in the non-atomic components. Orphans may also complicate interactive programs. For example, it is acceptable for an automatic teller machine to inform a customer that a requested transfer or withdrawal has not been performed, but it may not be acceptable to display nonsensical account balances before announcing the abort. Finally, debugging may be more difficult since orphan-induced inconsistencies may be indistinguishable from logical errors.

This paper proposes a new method to detect and eliminate orphans. Our method ensures that orphans are detected and eliminated in a timely manner, and it prevents orphans from observing inconsistencies. The method employs timestamps generated at each site. Timestamps may be generated by approximately synchronized real-time clocks [13], or by a system of logical clocks [8]. The former yields an "eager" scheme in which orphans are eliminated within a fixed duration, while the latter yields a "lazy" scheme in which orphans are eventually eliminated as information propagates through the system. A major advantage of the method is simplicity: it is easy to understand, to implement, and to prove correct. The method is *fail-safe:* unsynchronized clocks and lost messages may affect performance, but they cannot protect orphans from eventual elimination, nor can they produce inconsistencies.

This paper is organized as follows. Section 2 summarizes some related work. Section 3 describes the "eager" version of our method, and Section 4 describes the "lazy" version. Section 5 discusses implementation techniques, and Section 6 presents correctness arguments. Section 7 summarizes our results.

## 2. Related Work

Several research projects are studying transactions as the foundation for general-purpose distributed systems (e.g., [10, 23, 14, 22, 2]). An implementation based on methods proposed here is described by Kenky [7].

Outside the transaction domain, the orphan elimination problem was first identified by Nelson [16], and solutions based on timeouts has been proposed by Lampson [9] and by Rajdoot [18]. More recently, Walker [24] has proposed a transaction-based orphan elimination scheme that dynamically tracks dependencies among transactions. Walker's scheme requires optimizations based on timeouts to keep the amount of information sent in messages to a manageable level. An orphan elimination scheme based on Walker's method has been implemented as part of the Argus system [11]. Walker has shown that a similar orphan elimination scheme proposed by Allchin [1] contains subtle errors. Although our method is simpler than the Argus method, it may occasionally force non-orphan transactions to abort.

Our formal model for nested transactions incorporates work of Lynch [12] and Weihl [25], and our correctness condition for orphan elimination is a special case of a more general condition proposed by Goree [4]. A preliminary version of the eager scheme has appeared elsewhere [20]. The method described here incorporates several improvements; most notably it does not delay committing transactions. A general formal model for orphan elimination algorithms has been proposed by the first author, Lynch, Merritt, and Weihl [6].

## 3. Eager Orphan Elimination

This section describes an orphan elimination method based on a system of approximately-synchronized real-time clocks (e.g. [13]). An advantage of this scheme is that is places a real-time bound on orphan lifetimes, hence it bounds the resources that can be consumed by orphans. We first consider single-level transaction systems, and then we extend our method to nested transactions. The informal discussion assumes that synchronization is accomplished by two-phase locking [3, 15], although Section 6 shows the method is applicable to any synchronization mechanism that preserves atomicity.

### 3.1. Overview

The basic containers for data are called *objects*. Each object has a *type*, which defines a set of possible *states* and a set of primitive *operations* that provide the (only) means to create and manipulate objects of that type. Transactions operate on objects through a sequence of *operation executions*, each consisting of a paired *invocation* and *response*. Each transaction originates at a unique *home* site. A site emitting an invocation on behalf of a transaction is known as a *calling* site; the recipient site is a *called* site. Similarly, an object issuing an invocation is a *calling* object, and the target of an invocation is a *called* object. A transaction is said to have *visited* called and calling objects and sites. When a calling object issues an invocation, execution suspends within that object and passes to the called object. Execution resumes at the calling object when the response is issued by the called object. Thus, a transaction is

*active* at only one object at a time.

Each object has a clock, which is used to generate timestamps. Clocks in a distributed system are subject to the following constraints:

1. Each object's clock generates successively increasing timestamps.
2. When a message is sent from one object to another, the time at which it is received (by the receiver's clock) is later than the time at which it was sent (by the sender's clock).

Property 2 is readily achieved by including the sender's current time with each message. In this section, we assume that the objects at a site share a single real-time clock, and that clocks at different sites are synchronized using methods such as those of [13]. We emphasize, however, that as long as clock properties 1 and 2 are satisfied, unsynchronized clocks cannot protect orphans from eventual elimination or produce inconsistencies, although performance may suffer.

When a transaction acquires a lock for an object, it is assigned a *quiesce time* and a later *release time*. The quiesce time controls how long a transaction may remain active. When the object's local clock indicates that the transaction's quiesce time has passed, that transaction may no longer execute operations at that object, although it may still commit or abort. The release time controls how soon a transaction may abort. If the transaction aborts, its locks cannot be released until its release time has passed. If the transaction is not already prepared to commit when its release time arrives, then it can be aborted unilaterally at that object, and all information about the transaction may be discarded. A transaction that commits may release its locks immediately.

Let *Quiesce(x,A)* and *Release(x,A)* denote the quiesce and release times for transaction *A* at object *x*. Let *First(Release(A))* denote the earliest release time for *A* at any object, and let *Last(Quiesce(A))* denote its latest quiesce time. A transaction's quiesce and release times are subject to the following *termination invariant*:

$$\text{Last}(\text{Quiesce}(A)) \leq \text{First}(\text{Release}(A)).$$

By the time a transaction's release time arrives at any object, all activity on its behalf has quiesced. For locking protocols, this invariant eliminates potential inconsistencies by ensuring that all transactions, even orphans, satisfy the two-phase discipline: no transaction acquires a lock once it has released a lock.

The invariant is preserved in the presence of arbitrary message delays simply by including each transaction's local quiesce and release times with each operation invocation it sends to another object. The recipient refuses any message from a transaction whose quiesce time precedes the object's local time.

A simple way to preserve the termination invariant across site crashes is to keep locks and release times in non-volatile storage, perhaps in a small "stable cache". If this technique is impractical, an alternative technique is to set a system-wide maximum value for the *quiesce interval*, the duration between a site's current clock value and the quiesce time for any transaction (see Figure 3-1). When a site recovers, it reinitializes its clock, and refuses all operation invocations until the maximum quiesce interval has elapsed at every site in the system, ensuring that all transactions aborted by the crash have quiesced. This method assumes the rate of clock drift can be bounded. Recovery can be speeded up if sites periodically checkpoint their clock values to stable storage.

### 3.2. The Refresh Protocol

A transaction that is not an orphan will be aborted unnecessarily if its quiesce time arrives at a site before its activity there completes. To avoid this difficulty, a *refresh* protocol is periodically undertaken to advance each transaction's quiesce and release times. The interval between a site's current time and the quiesce time for any transaction is the *quiesce interval*, and the interval between the quiesce and release times is the *release interval*. The interval between refresh protocols is the *refresh interval*. These terms are illustrated in Figure 3-1. Unnecessary aborts will be unlikely if clocks are closely synchronized and if the refresh interval is significantly less than half the quiesce interval.

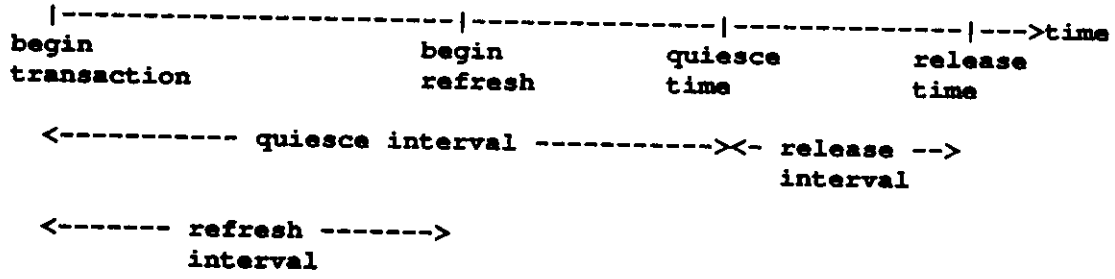


Figure 3-1: Quiesce, Release, and Refresh Intervals

The refresh protocol is a two-phase protocol similar to the two-phase commit protocol [5]. In the first phase, the home site attempts to advance the transaction's release time at all sites it has visited. If the first phase is successful, the home site attempts to advance the transaction's quiesce time at all sites visited. Two phases are necessary to ensure that the times are adjusted without violating the termination invariant. If a transaction is an orphan, it will be unable to complete the refresh protocol, thus its fixed quiesce time will bound its active lifetime. The remainder of this section describes the bookkeeping necessary to ascertain whether the first phase has succeeded.

Each site maintains two sets on behalf of each transaction. When a transaction executing at a site makes a call to an object, the called object is entered in the transaction's *outgoing* set. When a call arrives for an object at that site, the called object is entered in the transaction's *incoming* set.[1] A transaction's home site is in charge of refreshing its quiesce and release times. The home site first sends a *phase 1 refresh* message containing the new release time to sites visited by the transaction. Each site updates the transaction's local release time, and responds to the home site with a *phase 1 response* message containing the local *incoming* and *outgoing* sets. The home site builds complete *incoming* and *outgoing* sets by merging all received *incoming* and all *outgoing* sets respectively. Phase 1 is successful if the union of all sites' *incoming* sets equals the union of all sites' *outgoing* sets. This set is called the transaction's *visit list closure*.

If phase 1 completes successfully, the transaction's release time has been advanced at all sites. In phase 2, the quiesce time is advanced. The home site transmits a *phase 2 refresh* message advising visited sites of the new quiesce time. The termination invariant is preserved at each point during the protocol. Although responses to the phase 2 messages are not needed for correctness, they can reduce the likelihood of aborts caused by lost messages.

---

[1] An execution within a single site is regarded as both outgoing and incoming, but optimizations discussed below eliminate the need to maintain this data.

What if there are invocations in progress during the refresh protocol? There are two cases to consider. First, if an invocation occurs immediately before the transmission of a phase one refresh, the called object might appear at the calling site's *outgoing* set, but not (yet) in the called site's *incoming* set. In this situation, the home site can simply retry phase 1. Retransmission intervals should be chosen to minimize the risk of starvation in this case. Second, a site issuing a invocation after phase 1 but before phase 2 will use the old quiesce time but the new release time. The called site may retain the old quiesce time, which, although it does not violate the termination invariant, may cause the transaction to abort unnecessarily. This difficulty can be addressed by choosing a refresh interval substantially less than half of the quiesce interval, ensuring that any such site will be refreshed again before its quiesce time. In practice, the refresh and quiesce intervals may have to be tuned to incorporate such factors as lost refresh messages and the retransmission rate.

### 3.3. The Termination Protocol

When a transaction is aborted, its locks cannot be released until its release time has passed. If the quiesce interval is acceptably small, the aborted transaction's locks will eventually be released as its release times elapse. To hasten lock release, a *termination protocol* can be used to adjust the release time without violating the termination invariant. The termination protocol is similar to the refresh protocol. The first phase attempts to move the the quiesce time back to the present. If the visit list closure is successfully formed, indicating that all visited sites have moved the quiesce time, the second phase can move the release time back to the present.

### 3.4. Nested Transactions

Instead of treating transactions as monolithic entities, it is often useful to provide hierarchically structured nested transactions or *subtransactions* [15, 19]. A hierarchical transaction structure provides several benefits. Concurrency is enhanced by the ability to create parallel subtransactions. Fault-tolerance is facilitated and recovery is simplified because a subtransaction can abort without aborting its parent, an important consideration in distributed systems subject to faults. A subtransaction's commit is dependent on that of its parent; aborting the parent will undo the child's effects. A transaction's effects become permanent only when it commits at the top level. A transaction can commit only when all of its subtransactions have either committed or aborted.

We use standard tree terminology (parent, child, ancestor, descendant) when discussing nested transactions. (A transaction is considered its own ancestor or descendant.) Each nested subtransaction is given a quiesce and release time at each object it has visited. The quiesce time controls how long the subtransaction can execute operations at the object, and the release time controls when the subtransaction abort becomes visible to its parent. Quiesce and release times are subject to the following generalized termination invariant. If $A$ is an ancestor of $B$:

$$\text{Last}(\text{Quiesce}(B)) \leq \text{First}(\text{Release}(A))$$

By the time a transaction's release time arrives at any object, all activity on behalf of its descendants has quiesced.

The generalized termination invariant can be maintained by controlling descendants' refreshes from the parent's home site. Each transaction carries a *descendant count* as part of its state on all invocations and responses. The descendant count, in combination with the transaction's identity, is used to generate

names for sub-transactions. Since a transaction is active at only one site at a time, such names are unique. Initially, a nested transaction is given the same quiesce and release times as its parent, thus observing the termination invariant. During subsequent refresh protocols, the parent includes notification of the descendant's existence, along with the parent's *incoming* and *outgoing* sets. In the absence of aborts, and until it commits, the descendant is included in refreshes of its parent's quiesce and release times.

A transaction cannot abort a subtransaction until the latter's release time has elapsed at some object. Rather than waiting, the parent may undertake a termination protocol to move the subtransaction's quiesce and release times to the present. Note that the termination invariant permits a parent's quiesce and release times to be refreshed even if its descendants are inaccessible. When a site recovers from a crash, the techniques described above must be used to retain locks until the release times elapse for the top-level aborted transactions.

Eager orphan elimination imposes a negligible cost for short, successful transactions. Long transactions incur the cost of refresh protocols, and aborted transactions incur the cost of delays. The choice of the refresh interval trades one cost against the other: a long duration reduces the cost of refreshing long transactions, while a short duration provides faster orphan elimination. The choice should take into account the expected distribution of transaction lengths, the frequency of aborts, and the cost of delay. Eager orphan elimination is works best for systems in which transaction lengths are predictable and aborts are infrequent.

## 4. Lazy Orphan Elimination

This section introduces a modified version of the previous section's scheme. Instead of using the clock to drive lock acquisition and release, we use lock acquisition and release to drive the clock. Real-time clocks are replaced by logical clocks [8]. Logical clocks are counters associated with each object (or each site). Whenever a transaction visiting an object requests a timestamp, the counter is incremented, and the new value is returned. Whenever one object sends a message to another, the sender includes its current logical time, and the recipient advances its own logical clock beyond the observed value. A system of logical clocks clearly satisfies properties 1 and 2 stated above, but logical timestamps may be otherwise unrelated to physical time. Logical timestamps provide a simple and efficient technique for extending the natural partial order of events in a distributed system to an arbitrary total order.

As before, each transaction has a quiesce and release time at each object, satisfying the same termination invariant, but now these times are logical clock values, not real-time values. Lock acquisition and release are subject to the following rules. An object will refuse lock requests from any transaction whose quiesce time is less than the object's current clock value. When a transaction encounters such an object, however, it may attempt a refresh protocol to advance its quiesce time beyond the object's current clock value. When an aborted transaction releases its locks at an object, that object's clock is advanced beyond the transaction's release time.

The termination invariant is maintained across crashes by techniques analogous to those used for the eager scheme. For example, each object may periodically record its logical clock value on stable storage, maintaining a maximum difference, say $n$, between the current logical time and the latest release time. Upon recovery, the object adds $n$ to its recorded timestamp, and immediately resumes operation.

The lazy scheme has a number of attractive features. Since refresh protocols are "demand-driven" rather than "time-driven", they are executed only when conflicts arise, instead of at regular intervals. It is never necessary to wait for a transaction's release time to elapse, either for crash recovery or to abort a subtransaction, because an object's logical clock can be advanced instantaneously. Instead, a different kind of cost is incurred: additional refresh protocols may be triggered as clock advances propagate through the system. Whether the eager scheme's combination of periodic refresh protocols with delays is more cost-effective than the lazy scheme's demand-driven refresh protocols without delays depends on the expected frequency of aborts and the relative costs of delay and of message traffic. Perhaps the principal disadvantage of the lazy scheme is that it provides no real-time guarantees about orphan elimination. An orphan will continue to execute until it attempts to acquire a lock at an object whose logical clock exceeds the orphan's quiesce time.

## 5. Implementation Techniques

In this section, we discuss some simple implementation techniques and optimizations. One immediate optimization is to employ a robust lower-level protocol to ensure that lost messages do not cause unnecessary aborts. More significant optimizations can be obtained in the refresh protocol, reducing the number and sizes of messages.

In the first optimization, each site maintains its own list mapping transactions to objects visited at that site. *Incoming* and *outgoing* sets are extended to include both object and site names. When an invocation is issued, the calling site enters the object referenced in its *outgoing* set. When the invocation is received, the called site enters the name of the called object in its *incoming* set, then includes its site name in the response to the invocation or in an earlier lower-level protocol message. Once this is received, the calling site replaces the name of the object referenced in its *outgoing* set with the name of the site visited. If a lower-level protocol uses positive acknowledgement to the final response, receipt of this message can permit the called site to remove the object called from the *incoming* set. Thus, the size of incoming and outgoing sets is less, particularly if the number of sites visited by a transaction is small.

A second optimization also reduces the size of messages. In response to refresh messages, the sites transmit only changes to *incoming* and *outgoing* since the last refresh. During refreshes, the home site accumulates a list of visited sites. This list is then used, together with the *incoming* and *outgoing* sets received and information retained at visited sites, to ensure that the visit list closure is successfully formed.

The last optimization exploits broadcasting. A home site batches all refreshes for all transactions for which it is home. The refresh messages implicitly refer to all such transactions, except those explicitly excluded. Refresh messages are broadcast to all sites, which must determine whether they have been visited by any applicable transactions. The visited sites respond to a phase 1 refresh with the *incoming* and *outgoing* sets for all applicable transactions. The home site also broadcasts the phase 2 refresh message, specifying only those transactions for which the visit list closure was not formed successfully. This optimization is particularly effective if the number of aborts is low, and transactions visit few sites.

## 6. Correctness Arguments

So far, our discussion has assumed a transaction system based on two-phase locking. The restrictions imposed by our method can be generalized to apply to arbitrary concurrency control mechanisms (e.g., timestamp-based systems) as follows: no transaction may execute an operation at an object after its quiesce time there has elapsed, and no transaction may abort at an object before its release time there has elapsed.

This section presents formal correctness arguments for the orphan elimination method. The correctness arguments are valid for arbitrary data types (not just files), for arbitrary concurrency control methods (not just two-phase locking). One proof suffices for both the lazy and the eager schemes, since clocks properties 1 and 2 of Section 3 are the only assumptions needed about clock synchronization.

### 6.1. Objects and Transactions

Let OBJECT be a universal set of objects. Each object has a set of primitive *operations* that provide the (only) means to create and manipulate objects of that type. For example, a File might provide Read and Write operations, and a FIFO Queue might provide enqueue and dequeue operations. An *operation execution* is a paired invocation and response.

Let TRANS be a universal set of atomic transactions. Transactions have an *a priori* tree structure, with a distinguished transaction $U$ as the root. For a transaction $A$ distinct from $U$, let *parent(A)* denote $A$'s unique parent, *anc(A)* and *desc(A)* denote $A$'s ancestors and descendants (which include $A$), *proper-anc(A)* and *proper-desc(A)* denote $A$'s proper ancestors and descendants (which do not include $A$), and *lca(A,B)* denote the least common ancestor of $A$ and $B$. Let *siblings* denote the set $\{(A,B) \in TRANS^2 \mid parent(A) = parent(B)\}$. Let $seq \subseteq siblings$ be the partial order representing sequential dependency; if $(A,B) \in seq$, then $A$ is constrained to run before $B$.

### 6.2. Serial and Concurrent Specifications

A *system* is a set of objects. A *serial history* is a sequence of pairs of the form *[x e]*, where $x$ is an object and $e$ is an operation execution. A *serial specification* for a system is a set of *legal* serial histories. A system's serial specification characterizes its behavior in the absence of failures and concurrency. For example, the serial specification for a system including a FIFO queue would include all and only histories in which items are enqueued and dequeued at the queue in FIFO order.

A *concurrent history* is a sequence of triples of the form: *[x e A]*, where $x$ is an object, $e$ is either an operation execution, *commit*, or *abort*, and $A$ is a transaction. When a transaction commits at an object, its changes there become visible (e.g. through lock release). When a transaction aborts at an object, its effects there are discarded (e.g. through roll-back and lock release). Abort events encompass both explicit aborts, and aborts that occur as a side-effect of site crash and recovery. For brevity, a transaction commits (aborts) if it executes a commit (abort) at any object.

A *concurrent specification* for a system is a set of *legal* concurrent histories. A system's concurrent specification characterizes its behavior in the presence of failures and concurrency. A concurrent history is *well-formed* if it satisfies the following properties:

- Operation executions are associated only with leaf transactions.
- No transaction both commits and aborts.

- If *A* precedes *B* in *seq*, then *A* commits before *B* executes any operations.
- Each transaction commits at most once at each object, and it does not execute any events there after it has committed.
- No transaction commits until all of its children have either committed or aborted.

Henceforth, all concurrent histories are assumed to be well-formed. Well-formedness places no constraints on the behavior of orphans; once a transaction has aborted, it may do anything except commit.

Let *h* be a concurrent history, and let *Commit(h)* be the set of transactions that have committed in *h*. A transaction *B* has *committed to A* in *h* if $anc(B) \cap proper\text{-}desc(lca(A,B)) \subseteq Commit(h)$. Let *View(h,A)* denote the subhistory of *h* containing all events of transactions committed with respect to *A*. Let *Perm(h)* be *View(h,U)*, the subhistory of transactions committed to the top level.

We are now ready to define the basic correctness property for our orphan elimination method. A partial order $\rangle\rangle \subseteq siblings$ is *linearizing* if it is compatible with *seq* and it totally orders all siblings in TRANS. A linearizing partial order thus induces a total order (also denoted by $\rangle\rangle$) on the operation executions of the leaf transactions. A concurrent history is *serializable* if there exists a $\rangle\rangle$ such that reordering leaf transactions' object-operation pairs in the order $\rangle\rangle$ yields a legal serial history. A concurrent history *h* is *atomic* if *perm(h)* is serializable. Informally, a concurrent history *h* is *internally serializable* if each transaction has a serializable view for each operation execution. More precisely,

1. The empty history $\Lambda$ is internally serializable.
2. $h \cdot [x\ e\ A]$ is internally serializable if *h* is internally serializable and $View(h,A) \cdot [x\ e\ A]$ is serializable.

Internal serializability does not require that each transaction's view remain serializable after its last event has completed.

A concurrent specification is *atomic* if each history in the specification is atomic. To model schedulers that have no advance knowledge of transactions, we assume that an active transaction can choose to commit whenever the result is well-formed. A concurrent specification *S* is *on-line atomic* if it is atomic, and whenever *h* is in *S* and $h' = h \cdot [x\ commit\ A]$ is well-formed, then *h'* is also in *S*.

### 6.3. Proof of Correctness

A distributed system is modeled as an automaton *A* that accepts an on-line atomic concurrent specification *S*. Our orphan management scheme is modelled as a technique for embedding any such *A* in a derived automaton *A'* that accepts only the internally serializable histories in *S*.

An *automaton* is a tuple $<Q, q_0, E, \delta>$, where *Q* is a set of *states*, $q_0$ is the *initial state*, *E* is a set of object-event-transaction triples, and $\delta \subseteq Q \times E \times Q$ is a *transition relation*. It is convenient to extend the transition relation to sets of states:

$$\delta(\varnothing, [x\ e\ A]) = \varnothing$$

$$\delta(X, [x\ e\ A]) = \cup_{q \in X} \delta(q, [x\ e\ A])$$

and to sequences of events:

$$\delta(X, \Lambda) = X$$

$$\delta(X, h \cdot [x\ e\ A]) = \delta(\delta(X, h), [x\ e\ A])$$

A history $h$ is *accepted* by an automaton if $\delta(q_0, h) \neq \emptyset$.

Let TIMESTAMP be a totally ordered domain of timestamps. Given an automaton $A = <Q, q_0, E, \delta>$ that accepts an on-line atomic concurrent specification $S$, we construct the automaton $A' = <Q', q_0', E, \delta'>$ as follows. An element of $Q'$ is a tuple $<q, Clock, Quiesce, Release>$, where $q \in Q$, *Clock* is simply a timestamp representing the current time, either real or logical, and *Quiesce* and *Release* model each object's quiesce and release times for each transaction:

Quiesce: OBJECT × TRANS → TIMESTAMP

Release: OBJECT × TRANS → TIMESTAMP

*Quiesce* and *Release* are subject to the termination invariant:

If A ∈ anc(B) and x,y ∈ OBJECT then Quiesce(x,B) ≤ Release(y,A) (1)

The first component of the new initial state $q_0'$ is $q_0$, *Clock* has an arbitrary initial value, and *Quiesce* and *Release* have arbitrary initial values satisfying Property 1.

The new transition relation $\delta'$ is defined as follows. $\delta'(<q,Clock,Quiesce,Release>,[x\ e\ A])$ is undefined if either:

1. The event e is an operation execution and Quiesce(x,A) < Clock, or
2. The event e is *abort* and Release(x,A) > Clock.

These conditions capture the constraints that a transaction cannot execute an operation at an object if its quiesce time there has passed, and it cannot abort until its release time there has passed.

Otherwise, the transition relation's value is the set $\{<q',Clock',Quiesce',Release'>\}$ such that:

1. $q' \in \delta(q,[x\ e\ A])$,
2. Clock' > Clock, and
3. Quiesce' and Release' satisfy the termination invariant, and their values are unchanged for aborted transactions.

The first condition captures the notion that accepted events have their usual effect on objects' states, the second that the clock's value is increasing, and the third models refresh and termination protocols for active transactions.

Let $S'$ denote the histories accepted by the automaton $A'$. $S'$ is clearly a subset of $S$. It remains to show that:

**Theorem 1:** All concurrent histories in $S'$ are internally serializable.

**Proof:** The proof is by induction on the length of the accepted history. Clearly, the property holds for the empty history $\Lambda$. Assume $A'$ has accepted the internally serializable history $h$, and then accepts a new event $[x\ e\ A]$. Let $h' = h \cdot [x\ e\ A]$. If $e$ is *commit* or *abort*, then $h'$ is internally serializable. If $e$ is an operation execution, no *abort* events for ancestors of $A$ appear in $h$, because $Clock < Quiesce(x,A) \leq Release(y,B)$ for any object $y$ and any ancestor $B$ of $A$. Construct $h''$ by committing $A$'s ancestors in leaf-to-root order up to $U$, aborting all other active transactions. Because $S$ is on-line atomic, $h''$ is also in $S$, and therefore $Perm(h'')$ is serializable, and so is $View(h',A)$.

## 7. Conclusions

This paper has proposed a new method for managing orphans in a distributed transaction system. This method ensures that orphans cannot observe inconsistencies, and that orphans are eventually eliminated. The "eager" version of this method uses synchronized real-time clocks to ensure that orphans are eliminated within a fixed duration, and the "lazy" version uses logical clocks to ensure that orphans are eventually eliminated as information propagates through the system. Transactions are assigned timeouts at different sites. These timeouts are related by a global invariant, and they may be adjusted by simple two-phase protocols. The principal advantage of this method is simplicity: it is easy to understand, to implement, and it can be proved correct. Although the method is informally described in terms of two-phase locking, the formal argument shows it is applicable to any concurrency control method that preserves atomicity.

## References


[1] J. Allchin.
*An Architecture for Reliable Decentralized Systems.*
Technical Report GIT-ICS-83/23, Georgia Institute of Technology, 1983.

[2] K.P. Birman.
Replication and Fault-Tolerance in the ISIS System.
In *Proc. 10th Symposium on Operating Systems Principles.* December, 1985.
Also TR 85-668, Cornell University Computer Science Dept.

[3] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.
The Notion of Consistency and Predicate Locks in a Database System.
*Communications ACM* 19(11):624-633, November, 1976.

[4] J. Goree.
*Internal Consistency of a distributed transaction system with orphan detection.*
Technical Report TR-286, Massachusetts Institute of Technology Laboratory for Computer Science, January, 1983.

[5] J.N. Gray.
Notes on Database Operating Systems.
*Lecture Notes in Computer Science 60.*
Springer-Verlag, Berlin, 1978, pages 393-481.

[6] M.P. Herlihy, N.A. Lynch, M. Merritt, and W.E. Weihl.
On the correctness of orphan elimination algorithms.
In *17th Symposium on Fault-Tolerant Computer Systems (FTCS)*, pages 8-13. July, 1987.
Abbreviated version of MIT/LCS/TM-329.

[7] G.G. Kenky.
An action management system for a distributed operating system.
Master's thesis, Georgia Inst. of Tech., December, 1985.

[8] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
*Communications of the ACM* 21(7):558-565, July, 1978.

[9] B. Lampson.
Remote Procedure Calls.
*Lecture Notes in Computer Science 105.*
Springer-Verlag, Berlin, 1981, pages 365-370.

[10] B. Liskov, and R. Scheifler.
Guardians and actions: linguistic support for robust, distributed programs.
*ACM Transactions on Programming Languages and Systems* 5(3):381-404, July, 1983.

[11] B.H. Liskov, R. Scheifler, E. Walker, and W.E. Weihl.
Orphan Detection.
In *17th Symposium on Fault-Tolerant Computer Systems (FTCS)*, pages 2-7. July, 1987.

[12] N.A. Lynch.
Concurrency control for resilient nested transactions.
In *Proc. 2nd ACM Symposium on Principles of Database Systems.* March, 1983.
Revised version to appear in Advances in Computing Research.

[13] K. Marzullo and S. Owicki.
Maintaining time in a distributed system.
In *Proceedings of the second ACM Symposium on Principles of Distributed Computing*, pages 295-305. August, 1983.

[14] M.S. McKendry.
Clouds: A Fault-Tolerant Distributed Operating System.
*IEEE Tech. Com. Distributed Processing Newsletter* 2(6), June, 1984.

[15] J.E.B. Moss.
*Nested Transactions: An Approach to Reliable Distributed Computing.*
Technical Report MIT/LCS/TR-260, Massachusetts Institute of Technology Laboratory for Computer Science, April, 1981.

[16] B. Nelson.
*Remote Procedure Call.*
Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1981.

[17] C.H. Papadimitriou.
The serializability of concurrent database updates.
*Journal of the ACM* 26(4):631-653, October, 1979.

[18] Rajdoot.
*A remote procedure call mechanism supporting orphan detection and killing.*
Technical Report TR 200, University of Newcastle upon Tyne, April, 1985.

[19] D.P. Reed.
Implementing atomic actions on decentralized data.
*ACM Transactions on Computer Systems* 1(1):3-23, February, 1983.

[20] M.S. McKendry and M.P. Herlihy.
Time-driven orphan elimination.
In *Fifth Symposium on Reliability in Distributed Software and Database Systems.* January, 1986.
Also available as CMU-CS-85-138.

[21] M.D. Skeen.
*Crash Recovery in a Distributed Database System.*
PhD thesis, University of California, Berkeley, May, 1982.

[22] A.Z. Spector, D.S. Daniels, D.J. Duchamp, J.L. Eppinger, R. Pausch.
Distributed Transactions for Reliable Systems.
In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 127-146. ACM, December, 1985.
Also available in *Concurrency Control and Reliability in Distributed Systems*, Van Nostrand Reinhold Company, New York, and as Technical Report CMU-CS-85-117, Carnegie-Mellon University, September 1985.

[23] A.Z. Spector, J.J. Bloch, D.S. Daniels, R.P. Draves, D. Duchamp, J.L. Eppinger, S.G. Menees, D.S. Thompson.
The Camelot Project.
*Database Engineering* 9(4), December, 1986.
Also available as Technical Report CMU-CS-86-166, Carnegie Mellon University, November 1986.

[24] E.F. Walker.
*Orphan Detection in the Argus System.*
Technical Report TR-326, Massachusetts Institute of Technology Laboratory for Computer Science, June, 1984.

[25] W.E. Weihl.
*Specification and implementation of atomic data types.*
Technical Report TR-314, Massachusetts Institute of Technology Laboratory for Computer Science, March, 1984.

[26] W.E. Weihl, and B.H. Liskov.
Implementation of resilient, atomic data types.
*ACM Transactions on Programming Languages and Systems* 7(2):244-270, April, 1985.