CMU-CS-85-138

Time-Driven Orphan Elimination

Martin S. McKendry
Maurice Herlihy
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213
1 July 1985

Abstract

An orphan in a transaction system is an activity executing on behalf of an aborted transaction. This paper proposes a new method for managing orphans created by crashes and by aborts. The method prevents orphans from observing inconsistent states, and ensures that orphans are detected and eliminated in a timely manner. A major advantage of this method is simplicity: it is easy to understand, to implement, and to prove correct. The method is based on timeouts using clocks local to each site. The method is *fail-safe:* although it performs best when clocks are closely synchronized and message delays are predictable, unsynchronized clocks and lost messages 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.

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: indivisibility and recoverability. *Indivisibility* 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. An unsatisfactory way to ensure indivisibility is to constrain activities to execute one at a time. It is more desirable to permit activities to execute concurrently as long as they remain *serializable*: i.e., their overall effect is as if they had executed in a serial order. Atomic activities are called *transactions*.

Several research projects are studying transactions as the foundation for general distributed programs (e.g., [8], [5], [11], [17]). In these systems, 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. A transaction is a distributed computation that visits objects residing at multiple sites.

Well-known techniques such as two-phase locking [2, 13] and commit protocols [4, 16] can ensure the serializability of transactions that commit. Nevertheless, these techniques do not prevent transactions that eventually abort from observing 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. Although any such transaction will never be permitted to commit, it may observe inconsistent data while it executes. In a nested transaction system [15, 13], similar inconsistencies can arise when a transaction unilaterally aborts a nested subtransaction.

An activity executing on behalf of an aborted transaction is called an *orphan*. Although orphans eventually abort, potential inconsistencies may complicate programs that have non-atomic "benevolent site-effects." Examples of such programs include implementations of highly concurrent atomic objects that use non-atomic data at a lower level [20], and programs being debugged. Finally, orphans waste system resources.

Outside the transaction domain, the orphan elimination problem was first identified by Nelson [14], and a solution based on timeouts has been proposed by Lampson [7]. More recently, Walker [18] has proposed a transaction-based orphan elimination scheme that dynamically tracks dependencies among transactions. Walker's scheme includes optimizations based on timeouts to reduce the amount of information sent in messages. Goree [3] has given a comprehensive proof of part of Walker's algorithm. Walker has shown that a similar orphan elimination scheme proposed by Allchin [1] contains subtle errors.

This paper proposes a new method that uses timeouts to detect and eliminate orphans. The method prevents orphans created by crashes and by aborts from observing inconsistent data, and ensures that orphans are detected and eliminated in a timely manner. These features are particularly attractive for real-time systems, such as the *Alpha* kernel [12] being constructed by the Archons project. A major advantage of the method is simplicity: it is easy to understand, to implement, and to prove correct. The method is based on timeouts using clocks local to each site. The method is *fail-safe:* although it performs best when clocks are closely synchronized and message delays are predictable, unsynchronized clocks and lost messages cannot produce inconsistencies or protect orphans from eventual elimination. Although our method is simpler than Walker's method, it does place additional constraints on concurrency, and it may occasionally force additional aborts.

This paper is organized as follows. Section 2 describes the basic method. Section 2.4 extends the method to nested transactions, and Section 3 discusses implementation techniques. Section 4 presents correctness arguments, and Section 5 summarizes our results.

2. Basic Algorithm

This section describes the basic orphan elimination method. We first discuss single-level transaction systems, addressing nested transactions in Section 2.4. Our informal discussion assumes that synchronization is accomplished by some form of two-phase locking, although Section 4 shows the method is applicable to any synchronization mechanism that preserves atomicity.

2.1. Overview

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. An object issuing an invocation is a *calling* object, and an object referenced by 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 site has a clock, whose value advances monotonically. The clocks can be approximately synchronized real-time clocks with bounded drift [10], or logical clocks may be used [6]. We anticipate that practical implementations of this method will employ real-time clocks. When a message is sent from one site to another, the proof of the algorithm requires that the time at which the message is received be later than the time at which the message is sent. This property is readily achieved by including the sender's current time with each message.

When a transaction acquires a lock at a site, it is assigned a *quiesce time* and a later *release time*. When the site's local clock indicates that the transaction's quiesce time has passed, that transaction may no longer execute operations at that site, although it may still commit or abort. The transaction's locks at that site cannot be released, however, until its release time has passed. If the transaction's status is unknown when its release time arrives, then it can be aborted unilaterally at that site, and all information about the transaction may be discarded.

Let *Quiesce(x,A)* and *Release(x,A)* denote the quiesce and release times for transaction *A* at object *x*. A transaction's quiesce and release times are subject to the following *termination invariant*. For all objects *x* and *y* visited by *A*:

$$\text{Quiesce}(x,A) \leq \text{Release}(y,A)$$

The transaction's quiesce time at any object is less than its release time at any object. This invariant eliminates potential inconsistencies by ensuring that all transactions, even orphans, satisfy the two-phase locking discipline: no transaction will acquire a lock after 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 site. The recipient ignores any message from a transaction whose quiesce time precedes the site's local time.

One way to preserve the termination invariant in the presence of crashes is to have each site record each transaction's locks and release time on stable storage. Upon recovery, active transactions are aborted by resetting their quiesce times to the present. An alternative approach 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 2-1). 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 any transactions active at the time of 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.

2.2. 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 2-1. Unnecessary aborts will be unlikely if clocks are closely synchronized and if the refresh interval is significantly less than half the quiesce interval.

```
|------------------------|---------------|--------------|--->time
begin                    begin          quiesce        release
transaction              refresh         time           time

<----------- quiesce interval ----------><- release -->
                                           interval

<------- refresh ------->
         interval
```

Figure 2-1: Quiesce, Release, and Refresh Intervals

The refresh protocol is a two-phase protocol similar to the two-phase commit protocol [4]. 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. The two phases are necessary to ensure that the times are adjusted without violating the termination invariant. 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, that object is entered in the action's *outgoing* set. When a transaction makes a call to an object residing at that site, that object is entered in the action'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.

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.

2.3. The Termination Protocol

When a transaction commits or aborts, its locks cannot be released until its release time has passed. To avoid waiting for a transaction's release time to arrive, 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

[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.

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. When a transaction commits, the termination protocol can be integrated with the commit protocol. When a transaction aborts, an explicit abort protocol can be used to release its locks, or if the quiesce interval is acceptably small, its locks will gradually be released as its release times elapse.

2.4. Nested Transactions

This section extends the method to nested transactions [13, 15]. We use standard tree terminology (parent, child, ancestor, descendant) when discussing nested transactions. A transaction is considered its own ancestor or descendant. A nested transaction can abort without aborting its parent, and the commit of a nested transaction is conditional on the commit of its parent.

The termination invariant is extended to nested transactions as follows. If *A* is an ancestor of *B*, and *x* and *y* are objects:

$$\text{Quiesce}(x,B) \leq \text{Release}(y,A)$$

Each transaction's release times exceed its descendents' quiesce times. The 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. This, 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.

The termination invariant requires that a transaction cannot release its locks (i.e., its release time cannot be moved to the present) until all descendants are known to have terminated. If some descendant cannot be contacted to perform an explicit commit or abort (i.e., move the descendant's quiesce and release time to the present) the parent must wait to commit or abort until the descendant's release time has passed.

3. 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.

4. Correctness Arguments

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). We make no assumptions about clock synchronization.

Our model for nested transactions is based on work of Lynch [9]. Unlike our model, Lynch's model associates with each primitive operation execution a *label* indicating the object's state as observed by the transaction. A limitation of the label formalism is that it assumes that each object can be assigned a well-defined value at each step, and therefore it cannot model concurrency control mechanisms in which responses to invocations may be chosen on the basis of partial knowledge about state information. (For example, the response to a queue's *Deq* operation may be well-defined even if the response to the *Size* operation must await the outcome of concurrent transactions.) Our model remedies this limitation using a formalism based on work of Weihl [19], in which computations are modeled as sequences of events that may be reordered by serialization. Our correctness condition for orphan elimination is a special case of a more general condition proposed by Goree [3].

4.1. Serial Computations

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 Queue object might provide *Enq* and *Deq* operations. An *operation execution* is a paired invocation and response, e.g. *Enq(x);Ok()*, *Deq();Ok(x)*, and *Deq();Empty()*.

A *serial history* models a computation in the absence of failures and concurrency. A serial history is a sequence of object-operation execution pairs. For example:

```
q1 Enq(x);Ok()
q1 Enq(y);Ok()
q2 Enq(z);Ok()
q1 Deq();Ok(x)
q2 Deq();Ok(z)
```

is a serial history for two queue objects, *q1* and *q2*.

Each object has a *serial specification*, which is the set of serial histories for that object that characterizes the object's behavior in the absence of failures and concurrency. For example, the serial specification for a queue includes all and only histories in which items are enqueued and dequeued in FIFO order. A serial history for a single object is *legal* if it is included in that object's serial specification. A serial history for multiple objects is legal if the subhistory associated with each individual object is legal. For example, the serial history shown above is legal because the subhistories associated with *q1* and *q2* are each legal.

4.2. Action Trees

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 transactions *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* itself), and *lca(A,B)* denote the least common ancestor of *A* and *B*. Let *siblings* denote the set $\{(A,B) \in \text{TRANS}^2 \mid \text{parent}(A) = \text{parent}(B)\}$. Let $seq \subseteq siblings$ be the partial order representing sequential dependency; If $(A,B) \in seq$, then *A* is constrained to run before *B*.

A *behavioral history* is a sequence of object-event-action triples, where an event is an operation execution, *begin*, *commit*, or *abort*. The ordering of operation executions in a behavioral history reflects the order in which responses are returned, not necessarily the order in which invocations are issued. A behavioral history is *well-formed* if it satisfies the following constraints.

- A transaction may begin only once.
- A transaction must begin before it can execute any other events.
- A transaction may begin only after its parent has begun.
- If $(A,B) \in seq$ then A must commit at some object before B may begin.
- A transaction cannot begin after its parent has committed at any object.
- A transaction may commit only once at an object.
- A transaction may not execute any operations after it has committed at any object.
- A transaction may not commit until all of its children have committed or aborted at some object.
- A transaction may not commit at any object after it has aborted at any object.

Note that well-formedness places no constraints on operation executions of aborted transactions.

A partial order $\gg \subseteq siblings$ is *linearizing* if it is compatible with *seq* and it totally orders all siblings in TRANS. A linearizing partial order induces a total order (also denoted by $\gg$) on the elements of a behavioral history. A behavioral history is *serializable* if a legal serial history results from reordering elements in the order $\gg$, discarding all begin, commit, and abort events, as well as all transaction identifiers.

Let *commit(h)* denote the set of transactions that have committed in h, and let *proper-desc(A)* denote $desc(A)\text{-}\{A\}$. A transaction B has *committed to* A in h if $anc(A) \cap proper\text{-}desc(lca(A,B)) \subseteq Committed(h)$. Let $A_0 = U$, $A_k = A$, and $A_{i-1} = Parent(A_i)$, for i in 0,...,k. Let *View(h,A)* denote the subhistory of h containing the following events:

- All events of A_i that precede $Begin\ A_{i+1}$
- 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.

A *behavioral specification* S for a set of objects is the set of well-formed behavioral histories that characterizes computations in the presence of failures and concurrency. S is *on-line atomic* if every *h* in *S* satisfies the following properties: (1)

Perm(h) is serializable.

If *h* • *[x commit A]* is well-formed, it is in *S*.

The first condition is the standard definition of atomicity, and the second condition permits an active transaction to commit at any time.

Lemma 1: If $h \in S$ and *h* contains no abort events for an ancestor of *A*, then *View(h,A)* is serializable.
Proof: Construct *h'* by committing *A*'s ancestors in leaf-to-root order. *View(h',A)* = *Perm(h')* is serializable, hence so is *View(h,A)*.

Informally, a behavioral history *h* is *internally serializable* if each transaction has a serializable view immediately after executing an event. More precisely,

1. Λ is internally serializable.

2. *h* • *[x e A]* is internally serializable if *h* is internally serializable and *View(h,A)* • *[x e A]* is serializable.

In the next section we model our orphan elimination scheme as an automaton that accepts only internally serializable histories of *S*.

4.3. An Automaton

An *automaton* is a tuple $\langle Q, q_0, S, \delta \rangle$, where Q is a set of *states*, q_0 is the *initial state*, S is a set of *input symbols*, and $\delta \subseteq Q \times S \times Q$ is a *transition relation*. The transition relation can be extended to sets of states:

$$\delta(\emptyset, s_0) = \emptyset$$

$$\delta(X, s_0) = \cup_{q \in X} \delta(q, s_0)$$

and to sequence of input symbols:

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

$$\delta(X, s \bullet s_0) = \delta(\delta(X, s), s_0)$$

Here Λ denotes the empty string. A string s is *accepted* by an automaton if $\delta(q_0, s) \neq \emptyset$.

Let TIMESTAMP be a totally ordered domain of timestamps. Consider an automaton having the following components:

- log: EVENT*

- clock: TIMESTAMP

- Quiesce: OBJECT $\times$ TRANS $\rightarrow$ TIMESTAMP

- Release: OBJECT $\times$ TRANS $\rightarrow$ TIMESTAMP

The *log* component records the sequence of events already accepted by the automaton, *clock* models a system of clocks, and *Quiesce* and *Release* model each object's quiesce and release times for each transaction.

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

If $A \in anc(B)$ and $x,y \in$ OBJECT then $\text{Quiesce}(x,B) \leq \text{Release}(y,A)$ (2)

Each transaction's release times exceed its descendents' quiesce times.

In the automaton's initial state, the log component is empty, the clock has an arbitrary initial value, and *Quiesce* and *Release* have initial values satisfying Property 2.

We define a function *Filter* that suppresses *Commit* and *Abort* events for transactions whose release times have not arrived. If *e* is an event,

Filter(e) = if (e = [x Commit A] or e = [x Abort A]) and Release(x,A) < clock then Λ
else e

Filter is extended to histories in the obvious way (deleting occurrences of Λ). The automaton has the following preconditions for accepting an event *[x e A]*.

1. *log* $\bullet$ *[x e A]* $\in$ ***S***

2. *Filter(log)* $\bullet$ *[x e A]* $\in$ ***S***

3. If *e* is an operation execution then *Quiesce(x,A)* $>$ *Clock*.

The first condition ensures that behavioral histories are well-formed. Under two-phase locking, the second condition implies that a transaction's locks cannot be released until its release time has passed, and that a parent must remain blocked until its children's release times have passed. The third condition implies that a transaction cannot access data once its quiesce time has passed.

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

Theorem 2: All behavioral histories in ***S'*** are internally serializable.

Proof: Clearly, the property holds for Λ. Suppose $h \in$ ***S'*** is internally serializable, and $h' = h \bullet [x\ e\ A] \in$ ***S'***. The result is immediate if *e* is *begin*, *commit*, or *abort*. If *e* is an operation execution, no *Abort* events for ancestors of *A* appear in *Filter(h',A)* because *clock* $<$ *Quiesce(x,A)* $\leq$ *Release(y,B)* for any object *y* and any ancestor *B* of *A*. By Lemma 1, *View(Filter(h'),A)* is serializable, but any serialization of *View(Filter(h'),A)* is a serialization of *View(h',A)* = *View(h,A)* $\bullet$ *[x e A]*.

5. Conclusions

This paper has proposed a new method for managing orphans in a distributed transaction system. This method ensures that orphans cannot observe inconsistent data, and that orphans are eventually eliminated. The method is based on timeouts of clocks local to each site. Timeouts at different sites 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.

Acknowledgement

Lui Sha participated in early discussions of this approach.

References

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

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

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

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

[5] Jensen, E. D., and Pleszkoch, N.
Arch OS: a physically dispersed operating system.
IEEE Tech. Com. Distributed Processing Newsletter 2(6), June, 1984.

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

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

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

[9] Lynch, N. A.
Concurrency control for resilient nested transactions.
In *Proc. 2nd ACM Symposium on Principles of Database Systems.* March, 1983.

[10] Marzullo, K.
Loosely-Coupled Distributed Services: A Distributed Time Service.
PhD thesis, Stanford University, 1983.

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

[12] McKendry, M. S., and Northcutt, J. D.
The Alpha Kernel.
Technical Report CSL-85-???, CMU, 1985.
In Preparation.

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

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

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

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

[17] Spector, A. Z., Butcher, J., Daniels, D. S., Duchamp, D. J., Eppinger, J. L., Fineman, C. E., Heddaya, A., Schwarz, P. M.
Support for Distributed Transactions in the TABS prototype.
IEEE Transactions on Software Engineering 11(6):520-530, June, 1985.

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

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

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