459-89

An Abort Mechanism for Nested Distributed Transactions

Dan Duchamp
Computer Science Dept.
Columbia Univ.
New York, NY 10027

Abstract

A transaction processing facility must have a mechanism for aborting transactions on request. This paper describes a mechanism for aborting transactions that can be arbitrarily nested and/or distributed. The mechanism consists of an "abort protocol" plus an adjustment to the commit protocol. The abort protocol locates and terminates as many of the transaction's operations as it can. If after the protocol has finished it is still possible that orphaned operations exist, then a simple check during the prepare phase of the commit protocol ensures that no orphan commits.

The mechanism has many advantages: provided that the communication subsystem provides prompt failure detection, there will be no orphans; a site can abort unilaterally; there is little overhead on transaction-operation messages, and relatively few and relatively minor restrictions on the transaction facility; no information need be maintained in stable storage; and the abort protocol never blocks. The primary disadvantages of the mechanism are that the abort protocol must be synchronous, that it may over-abort in some cases, and that — if the communication subsystem *does not* provide prompt failure detection — there is no limit on the extent or lifetime of orphaned computations.

This work was supported by IBM and the Defense Advanced Research Projects Agency, ARPA Order No. 4976 (Amendment 20), under contract F33615-87-C-1499, monitored by the Air Force Avionics Laboratory, Wright Aeronautics Laboratories, Wright-Patterson Air Force Base.

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 any of the sponsoring agencies or of the United States Government.

Table of Contents

1. Introduction

Measurements of some single-site database applications indicate that about 3% of all transactions abort by request [5]. Common reasons for aborting include bad user input and system detection of conditions such as deadlock. This paper describes a mechanism for the explicit abort of transactions that can be nested and/or distributed. The mechanism consists of an "abort protocol" plus an adjustment to the commit protocol. The abort protocol locates and terminates as many of the transaction's operations as it can. If after the protocol has finished it is still possible that orphaned operations exist, then a simple check during the prepare phase of the commit protocol ensures that no orphan commits. This mechanism has been implemented within the Camelot transaction processing facility [4].

The primary advantage of the mechanism is the low overhead it imposes on the normal processing of transaction operations: there is no need ever to place extra abort-related information in stable storage, and little extra processing needs to be done on inter-site messages. The main disadvantage of the mechanism is that the abort protocol must be run synchronously. Interestingly, these properties define an abort mechanism that is nearly the dual of that used by Argus [12]; Section 6 discusses the contrast.

The body of the paper is organized as follows. Section 2 explains the assumptions that underlie the work. Section 3 develops a specification of the mechanism as a series of refinements, while Section 4 provides informal arguments that the mechanism is both safe and live. Sections 5 and 6 evaluate the mechanism in absolute terms and relative to previous work, respectively.

2. Assumptions and Restrictions

2.1. Execution Model

The model of transaction execution is quite general. A single *application* process starts a transaction, invokes the *operations* exported by other processes called *servers*, and then initiates commitment. A transaction may consist of any number of operations that call servers at any number of sites. Servers — unlike applications — manage segments of recoverable storage; they may also start and execute transactions in the course of servicing an operation. Abort can be requested by any process at any time up to the moment when the top-level transaction becomes prepared to commit.

Each site has a *transaction manager*, which is part of the transaction facility and which has two primary functions. First, it maintains bookkeeping information about which transactions are active at local servers and which have made operation calls to servers at other sites. Second, it cooperates with transaction managers at other sites to execute the commit and abort protocols that ensure multi-site atomic behavior. Whenever a site recovers from a crash, the transaction manager aborts all transactions that were active at the moment of the crash. These transactions are then immediately forgotten.

The nesting model is the following variant of the Moss model [15]:

- A transaction can spawn one or more nested transactions in parallel or in sequence. A parent is prohibited from accessing any of its locked data so long as any child is running.
- A descendant can *inherit* locks held by an ancestor.
- When a child commits, its locks — both inherited and newly acquired — are given (*anti-inherited*) to the parent.
- When a child aborts, its newly acquired locks are dropped, and its inherited locks are anti-inherited.
- The effects of a committed child are made permanent only when the top-level transaction commits.
- Aborting a transaction implies aborting all transactions nested within it.
- A "committed" nested transaction can be aborted. Aborting a committed nested transaction implies aborting all transactions up to and including its lowest active ancestor.

The entire collection of nested transactions is called a *family*. The usual tree terminology is used to refer to transactions within a family. This model is implemented by both Camelot [3, chap. 4] and Argus [11].

2.2. Assumptions and Non-assumptions

The failure model is ordinary: processes are fail-stop; sites may crash and lose their volatile memory; and the network can lose or duplicate messages and can partition, but may not manufacture or undetectably garble messages.

No assumption is made about the nature of the commit protocol for nested trans-

actions; indeed, is it not even necessary that one exist.[1] The abort mechanism is dependent upon the characteristics of the top-level commit protocol only insofar as there must be a "prepare" phase during which sites may vote to abort the top-level transaction. The topology of the protocol and the presence or absence of phases before or after the prepare phase are of no consequence to the abort mechanism.

The mechanism is likewise independent of the methods for concurrency control and recovery.[2] Keeping the abort mechanism free of any assumption about the recovery algorithm creates a subtle but severe restriction: nested transactions must be undone from the bottom up, and any particular nested transaction can be undone only once. That is, the minimum assumption one can make about the capabilities of the recovery process is that it is able to undo transactions in reverse order of creation.

2.3. Required Restrictions

Besides the recovery restriction mentioned above (which is really the consequence of a non-assumption), five additional restrictions must be imposed upon the transaction facility.

First, commitment must be synchronous: a transaction may not commit until all of its operations have been completed or aborted *and* all its child transactions have been committed or aborted. Additionally, applications and servers must abort the transaction enclosing any operation call that fails to respond. These restrictions simply ensure well-defined transactions.

Second, there must be a site (the ***commit source***) which is responsible for eventually initiating the commit protocol. It is assumed that there will be no attempt to commit the top-level transaction if the commit source crashes. The commit source is typically the creation site of the top-level transaction and the site running the application.

Third, the transaction identifier (TID) must encode the address of the creation site(s) of both family and transaction. This is easily accomplished with the classic technique of producing unique identifiers by concatenating the host-id and a monotonic integer. TIDs need not encode nesting information, implying they need not be variable-length.

Fourth, a server must eventually abort any transaction that has been active for too long. It is easy to lift this restriction, as discussed in Section 3.7, but doing so requires support from the communication subsystem that for now we do not assume ex-

---

[1]Camelot and Argus both use "lazy commitment," in which the commitment of a nested transaction consists of no more than having its local transaction manager make note of the fact in volatile memory. Locks are later anti-inherited only if they are requested by a transaction in the same family.

[2]Although this paper is written as if the recovery method is logging and the concurrency control method is locking, these conventions are adopted only in order to be precise in discussing the actions required at various times.

ists.

Fifth, the inter-site communication subsystem must support transaction management by performing two services beyond simple message transport: ***crash detection*** and ***piggybacked information accumulation***. Performing crash detection means guaranteeing the abort of an operation that is directed to a site that had earlier performed an operation for the same family but then crashed and recovered. Performing piggybacked information accumulation means intercepting outgoing operation messages, adding transaction management information to them, then having the destination intercept the (incoming) message, strip off the extra information, and merge it with information received on previous messages. Information accumulation is used as a mechanism to implement crash detection. These services are discussed next.

2.4. Communication Subsystem

2.4.1. Crash Detection

Crash detection is important in controlling orphans, which in turn is a key goal of any abort mechanism. If a server crashes, all transactions active at that server must abort. A transaction system must guard against this sequence of events:

1. Transaction T does operations at several servers, including some at a server at site S.
2. Site S crashes and recovers quickly. As part of recovery, T is aborted at that site. Other sites do not learn of the abort. No memory of T is retained at site S.
3. A second operation of T is directed to a second server at site S, reestablishing transaction T at the site.
4. Transaction T commits at all sites, including S.

The transaction should abort because of the crash, but instead is partially committed and aborted, violating atomicity. There must be some way of always detecting the server crash of step 2, so that step 3 is prevented.

If inter-site communication is done via reliable connections over which "keepalive" messages regularly travel, then crash detection is provided free by a lower layer and the technique discussed here is unnecessary. This section presupposes that inter-site communication is via datagrams, and that the transaction facility is responsible for crash detection. We assume that a transaction manager can detect the death of local server processes and will reliably initiate abort in that case. If so, then the problem of detecting server crashes becomes one of detecting site crashes.

The method used is an adaptation of a proposal [9, pp. 40-43] which was intended for an environment that did not include nested transactions; assuming that calls are synchronous and all initiated at the commit source:

- Each site maintains a timestamp generator. It may be a clock value, or — if the resolution is not fine enough — a "Lamport Clock" [8] whose value increases with every message sent or received.
- The timestamp of the arrival of the first request by a particular trans-

action at a particular site is called the ***low water mark*** (LWM) for the transaction at that site.

- Piggybacked onto every response message is the LWM of every site used in servicing the request.
- The creation site of the transaction is the repository for the LWMs of all sites involved in the transaction; this is feasible because all unaborted calls eventually return to it. Also, no harm is done if the repository crashes and the LWMs are lost: the transaction will never try to commit.
- The transaction manager processes every response by comparing a site's LWM as listed in the message with the corresponding LWM in its memory. If it does not have a LWM for that site, it records (in memory only) the LWM given by the message. If a previous LWM is recorded and it does not match the one in the message, then one of the two sites has crashed, and the transaction must be aborted.
- A site that crashes and recovers loses all recorded LWMs, and will generate a new, higher LWM if a transaction returns to it.

The basic idea is that two things are needed: something should be different about a site before and after a crash, and there must be some certain way of detecting the difference. For detecting the difference, the memory of the transaction creation site is the logical place: every non-aborted call returns there, and if it crashes, commitment will not take place.

This method is trivially extended for accommodating nested transactions. The commit source is used as the repository. Every response should contain the LWM *for the family* for every site used during the call. A detected crash results in abort of the whole family, even though strictly speaking abort need be done only up through the least uncommitted ancestor of each transaction that was active at time of crash. Abort of the particular nested transaction that established the family LWM has no effect: all that is needed is a way of denoting a difference across crashes. This approach to crash detection requires a timestamp mechanism at every site, adding timestamps to every response, storing LWMs at the commit source, and — for every response received at that the commit source — comparing the LWMs in the message with those already stored. If LWMs are also placed in *request* messages, then in certain cases LWM mismatch can take place earlier, when the request is received.

2.4.2. Information Accumulation

The list of sites visited by various transactions must be accumulated for later use by the commit and abort protocols. The crash detection algorithm likewise depends upon the accumulation of LWMs. Adding information to response messages provides backward accumulation; adding to requests provides forward accumulation.

Three kinds of information are backward-accumulated:

1. The sites used by the *transaction during the call*. The abort protocol requires knowing the sites visited by a particular transaction. So that the abort protocol can track down in-progress calls, the call destination must be regarded as a "used site" at the moment the request goes out.

2. The sites used by the *transaction and all its descendants during the call.* The list accumulated at the commit source, minus duplicates, forms the list of sites that must participate in the commitment protocol.

3. For each site used during the call, the timestamp of the first arrival of any transaction in the family at that site.

The first type of information is associated with a particular transaction, the second with a particular family, and the third with a particular site. Figure 2-1 illustrates exactly where and when extra information is added to messages. The list of sites used by a *transaction and all its descendants during the call* must be forward-accumulated; the orphan protection mechanism uses this list as described in the next section.

It may be convenient to implement forward and backward accumulation in a form slightly different from the design described above, for two reasons. First, the list of sites used by a transaction and its descendants of course subsumes the list of sites used by the transaction alone. Second, implementing the design exactly requires supporting a notion of "call" that transcends the notion of transaction, since a single call may use many transactions. Adding the notion of a "call" to an inherently connectionless message system would be inconvenient, perhaps extremely so. Further, handling requests and responses differently is inconvenient. Consequently, every request and response message may be loaded with the same extra information:

1. The identity and timestamp of every site used by the *transaction, ever.*

2. The identity and timestamp of every site used by the *family, ever.*

This information is a superset of that required by the design. The expense of this implementation would be prohibitive only if a transaction scaled to a very large number of sites.

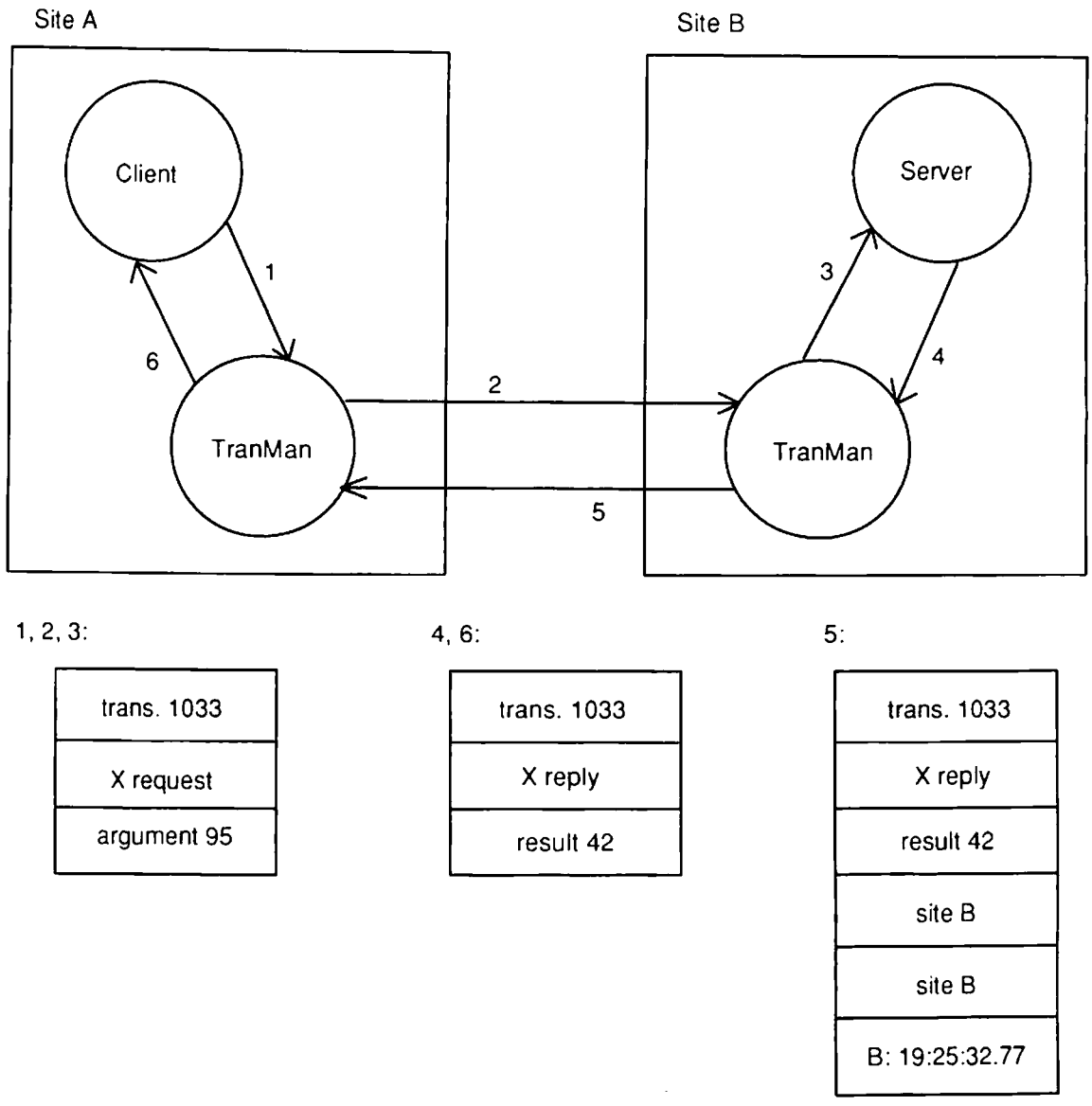

Figure 2-1: Information Accumulation

A client at Site A calls a server at site B, and the message is intercepted by the transaction managers at both sites. The format of the each message is shown below the message flow diagram. Messages 1, 2, and 3 are all requests, and all have the same format. The first field lists the transaction (1033). The second field indicates that the message is a request for operation X. The third field is the one argument that operation X requires; its value is 95. The response has fields identifying the transaction, the type of message, and the result. The message transmitted between the two transaction managers (message 5) has a fourth field listing the sites used by the family, and a fifth field listing the sites used by the transaction. Also, there is a timestamp indicating at what time the family first reached each site listed in the fourth and fifth fields.

3. Protocol Description

Several factors complicate the design of an abort mechanism for the target environment. Aborting a single nested transaction may require aborting many others, and aborting these transactions should not prevent the remaining parts of the family from committing. Further, distributed transactions that are to be aborted may still be spreading to new sites and/or creating new descendants while abort is proceeding. Last, a site crash will result in the loss of the information that describes the spreading and nesting initiated at that site. There are two consequences of losing this information.

First, an abort mechanism has limited ability to retain state. For instance, it cannot wait for acknowledgement from another site that particular transactions have been undone there, because that site may have crashed and lost its memory. Second, ***orphans*** may be created. An orphan is any operation, finished or still being performed, that must be aborted but which cannot be located. Orphans are created by crashes because the record of which sites call which others is kept in memory rather than stable storage. For example, if Site A invokes an operation at Site B and then Site A crashes, then the work done at B is orphaned. It does not matter whether the operation replies before the crash. Co-existing with orphaned transactions there may be committed nested transactions in the same family that should be allowed to commit. The orphaned and non-orphaned operations must be distinguished by the time the top-level transaction commits. (A system without nested transactions can afford to be slower in eliminating orphans because they do not threaten atomicity.)

Consequently, the abort mechanism consists of two portions:

1. An "abort protocol" which is a method for locating and undoing as many operations as possible.
2. An "orphan protection mechanism" for ensuring that orphans that escape the abort protocol (due to failures) never commit up to the top level.

If the abort protocol cannot undo all operations, it is responsible for recognizing this fact and ensuring that the orphan protection mechanism has enough information to do its job.

3.1. Overview

When abort is requested, *kill* messages are sent among transaction managers from site to site in a pattern mimicking the pattern of earlier transaction-operation messages. Upon receipt of a *kill*, a transaction manager undoes the local effects of the named transactions and then sends a *kill* to all sites the dead transactions communicated with. After each of those sites responds with a *kill-ack* message, the transaction manager returns a *kill-ack* to the site that sent it a *kill*.

If some site fails to return a *kill-ack* after a reasonable period of time, then it is classified as ***dangerous***; the identity of every dangerous site is synchronously reported to the commit source with a *danger* message, which is then acknowledged by a

danger-ack. If any dangerous site is found, then when the protocol terminates, the aborting tree of transactions will be only partially undone. The list of dangerous sites is the link between the abort protocol and the orphan protection mechanism.

A dangerous site is dangerous for two reasons. First, if it is crashed, then there may exist (unknown to any other site) operations that spread from the crashed site before it went down. These operations are orphans. Second, if the site is unreachable due to communication failure, then orphaned work will be left there, possibly together with committed nested transactions that should commit up to the top level.

The fact that is the key to the design of the abort mechanism is that *every orphaned operation will either be active at a dangerous site, or will have passed through a dangerous site on its way to the site where it is active*. The orphan protection mechanism is built into the prepare phase of the commitment protocol. All dangerous sites known to the commit source are included in the prepare message.[3] Subordinates process such a "dangerous prepare" message somewhat differently from a typical prepare message. A subordinate first compares the given set of dangerous sites to its list of accumulated sites (for the entire family). If any dangerous site is in the list, the subordinate votes not to commit the top-level transaction. This policy prevents orphans from committing: every operation that passed through a dangerous site and which is now active at a site that might commit is detected and causes top-level abort. Any past communication with a dangerous site is conservatively assumed to be orphaned work. This rule overestimates the amount of orphaned work (any operation that used any dangerous site), and over-aborts (the whole family).

The key to the correctness of this method is informing the commit source when a dangerous site is detected. The restriction that a family cannot be committed until all of its children are either committed or aborted is used to ensure that orphans do not commit: the synchronous abort call does not return until the possibility of orphan problems is recorded at the commit source, thereby delaying the commitment of the enclosing transaction. If the commit source is unreachable then the top-level transaction is aborted locally. If the commit source crashes, then commitment of the top-level transaction will never be attempted, and orphans are eliminated when each server aborts the transaction for running too long. Thus, the protocol continues to operate in spite of any number of failures, but "operating" may consist of aborting the top-level transaction if the abort of a nested transaction becomes blocked. The act of aborting a top-level transaction never blocks, as explained in Section 3.6.

Thus, the approximate steps involved in aborting an arbitrary nested transaction are:

---

[3]Beyond this exception, there is no further presumption that the messages of the protocol carry any information that would simplify the abort mechanism; commit messages are assumed to carry only the identifier of the committing family and to mean no more than "commit all the operations of family X."

- Discover which transactions are ***victims*** (i.e., which transactions must be aborted).
- Proceed from site to site undoing victims.
- If a dangerous site is encountered, record it at the commit source.
- If recording danger fails, abort the whole family.
- Return to the caller once every site has either reported that its victims have been undone or has been recorded as dangerous.

The protocol described so far is oversimplified and lacks several crucial details. Refinements in Sections 3.2 through 3.6 fill in the missing features.

3.2. Defining Victims

The death of transaction X implies the death of all its descendants. Furthermore, if X is already committed, then abort must take place up to and including its lowest active ancestor (the ***abort root***). Figure 3-1 gives an example of how a request to abort one transaction makes victims of three others.

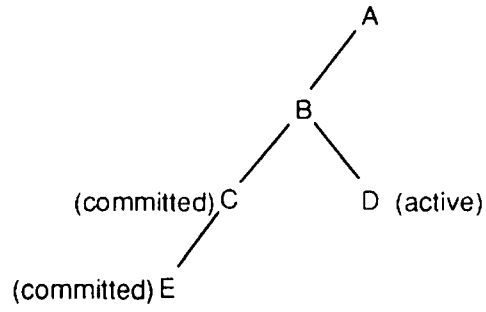

Figure 3-1: Victim Selection

Aborting transaction C results in the death of its descendant E as well. Since C is committed, B and its descendant D must also be aborted.

Locating all victims requires tracking down operations along two dimensions: following nesting relations, either up (to find the lowest active ancestor) or down (to find descendants), and following site-to-site spreading of each of these transactions. The pattern of nesting forms a ***nesting tree***. The pattern of spreading forms several (arbitrary) per-transaction graphs with edges threaded through a common set of nodes; the nodes represent sites, and the (directed) edges represent the pattern of inter-site calls. For simplicity, we refer to the whole collection loosely as ***the spreading graph***. Figure 3-2 offers an example of how an execution defines a nesting tree and a spreading graph.

Each transaction manager maintains its site-specific portion(s) of the nesting tree, a record of the sites to which its active operations spread, as well as the forward-

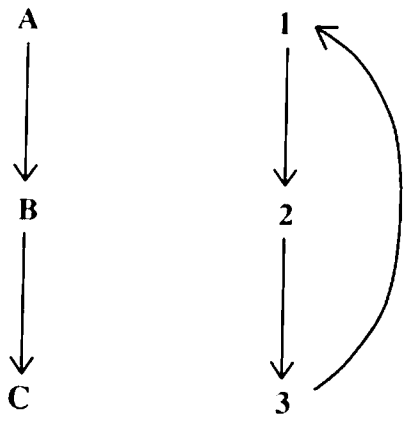

Figure 3-2: Example Nesting Tree and Spreading Graph

Suppose that Transaction A begins at Site 1, then spreads to Site 2. At Site 2, nested transaction B is created and spreads to Site 3. At Site 3, nested transaction C is created and spreads to Site 1. The nesting tree is shown on the left, and the spreading graph on the right.

accumulated list of sites used previously by active operations. To locate all victims, the nesting tree and the spreading graph must be completely traversed. For following both types of edge there is a convenient starting point. In the case of nesting, it is the abort root. In the case of the spreading graph, it is the creation site of the abort root, or ***abort source***.[4]

3.3. Locating and Undoing Victims

If the abort initiator is not also the abort source, then a series of *died messages* are sent starting at the abort initiator and ending at the abort source. The purpose is to proceed from ancestor to ancestor up the nesting tree to find the lowest active transaction, which then becomes the abort root. Accordingly, a *died* message identifies the most highly nested transaction that the sending site knows must abort and knows was created at the destination site. In general, it may happen that several transaction managers receive an "X died" message, trace the local child-parent relations up from descendant X to ancestor Y, then send a "Y died" message to another site. Figure 3-3 illustrates this: the abort initiator sends a died message to an intermediate site which then sends another one to the abort source.

Once the abort root is located, the transaction manager at the abort source begins to undo the victims. It should traverse (from the top down) the portion of the nesting tree rooted by the abort root, and for each transaction do the following:

[4]Two other significant terms will be used in the rest of the discussion:

- ***Abort target***: the transaction named in the request to abort.
- ***Abort initiator***: the site (or sites) where the request to abort is made.

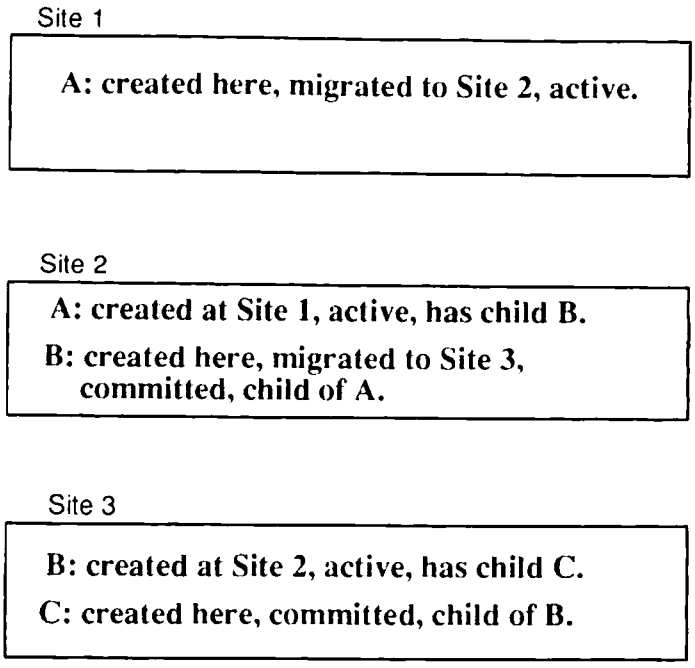

Figure 3-3: Died Messages

Each box represents the knowledge of the transaction manager at each of three sites. An operation has been performed, starting with Transaction A at Site 1. Sites 2 and 3 were called and they each created a nested transaction which committed. Transaction B is listed as committed at Site 2 and active at Site 3 because of the lazy commitment of nested transactions.

In this example, if a process at Site 3 aborts Transaction C, "B has died" would be sent from Site 3 to Site 2, and "A has died" would be sent from Site 2 to Site 1. Site 1 would become the abort source, and Transaction A the abort root.

1. Check whether the transaction (and its ancestors) is already in the process of being aborted. If so, do not traverse that portion of the tree.
2. ***Freeze*** the victim along both dimensions: prevent it from spreading, and prevent it from creating child transactions.
3. Prevent any server previously uninvolved with the victim from performing operations for it, and ***suspend*** the servers involved with the victim. A suspended server will accept no further operations for that transaction, and will await instructions from the recovery process about how to reset the portions of its data segment that have been changed by the victim.

Once the bottom of the nesting tree is reached, the traversal must return from the bottom up, performing the following actions on the way:

4. Tell the recovery process to undo the victim. Once this is finished, place an abort indicator into the log, and tell the server to drop the victim's locks.
5. Send a *kill* to every site that the victim spread to.

Other sites will initiate the same procedure when they receive a *kill*. The abort initiator can undo the abort target before receiving a *kill* for it. However, to preserve the only-once recovery restriction, its transaction manager must remember to do the check outlined in step 1, and not perform a second undo of the subtree rooted at the abort target when second when a *kill* arrives for any of those transactions.

Providing that certain conditions are met, eventually all operations of all victims are located and undone: a *kill* is sent along every path taken by any operation request. The conditions are:

- No failure occurs.
- No message is lost.
- A transaction that has already been aborted at a site is prevented from initiating new operations at that site. This condition both preserves the only-once recovery restriction and prevents a transaction from creating an infinite-size abort problem by perpetually looping through the same set of sites.
- There must be an assumption that a transaction will not continue spreading operations to an infinity of new sites, or (if it does) that the abort protocol is, on net, "faster" than the operations and will eventually catch up to and abort all of them.

Coming sections will explain how to remove the need for the first two conditions, and how to ensure the third. Ensuring the fourth condition would seem to require an *a priori* limit (e.g., time or the number of sites used) on a transaction. However, the transaction model provides for unfettered execution, so the most that can be said is that the unending spread of a transaction to new sites is exceedingly unlikely.

Besides these conditions, there are several other sloppy aspects to the protocol as it stands now:

1. It does not say when the abort call should return to the caller.
2. Lost messages cause problems. In particular, the loss of a *died* message would prevent abort from ever occurring. Also, a lost *kill* would result in needless top-level aborts.
3. Conversely, there are many reasons that a site may receive multiple *kills* for the same transaction:
    a. A transaction may spread to a particular site more than once, and from different sites. All those sites will send *kill* messages.
    b. Cycles may exist in the spreading graph: a transaction or one of its descendants may loop back to a site it has previously visited.
    c. Use of the simplified implementation of information accumulation explained in Section 2.4.2 would create yet another source of duplication of *kill* messages. For example, if while servicing an operation, a transaction spreads from Site A to Site B and from B to C before returning, then Site B will be aware of C while Site A will know of both B and C. During abort Site A will send *kill* to both B and C, not just B.
    d. The method of traversing the nesting tree leads to further message inefficiencies. For example, if N transactions created at one site had each called a second site, then N *kill* messages would be

sent from the first site to the second, even if the transactions had a common ancestor.

The first three causes of duplicates are illustrated by Figure 3-4. Remedies for each of these shortcomings are presented in the following section.

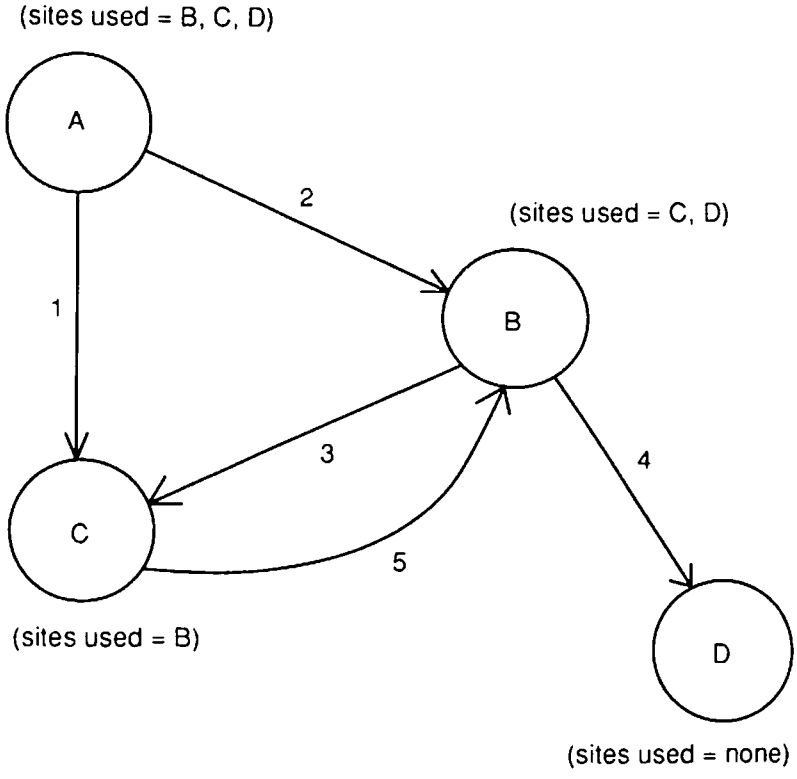

Figure 3-4: Duplicate Kill Messages

If a transaction spreads, in order, from Site A to Site C, from A to B, from B to C, from B to D, and from C to B, then during abort Site C will receive kill messages from both A and B (cause a), Site B will receive kill messages from both A and C (cause b), and Site D will receive kill messages from both A and B (cause c).

3.4. Further Refinements

3.4.1. Returning to the Caller

The process that (synchronously) calls for abort cannot be resumed until it is certain that every dangerous site has been recorded at the commit source, even if the dangerous site was in a portion of the spreading graph that will not be reached by *kill* messages sent from the initiator. Only when the abort source receives its *kill-ack* can it — and it alone — know that the entire spreading graph has been tested for dangerous sites by being sent *kill* messages. Therefore, a *kill-complete* message is added to the protocol. The abort initiator should not respond to the abort call until it has

received this message from the abort source. This message is sent by the abort source only once it has received all its *kill-acks*; its meaning is that the abort source certifies either that all operations have been undone or else that all dangerous sites have been recorded. If no *kill-complete* arrives after a time, the abort initiator should abort the entire family. Of course, if the abort initiator and the abort source happen to be the same site, then *died* and *kill-complete* messages need not be sent.

The precise meaning of the abort call returning can now be stated as: the abort target and its descendants have been undone at the local site, the abort source, and possibly at other sites; dangerous sites, if any, have been recorded at the commit source.

3.4.2. Increasing Message Reliability

Because unacknowledged *kills* cause the reporting of dangerous sites, and because the false reporting of a dangerous site will cause top-level abort, it is reasonable to retransmit *kill* messages a small number of times. A site should not acknowledge until all sites to which it sent *kills* have acknowledged or after it has timed out and recorded them as dangerous. There are three legal responses to site X sending a *kill* message to site Y:

1. Y responds with a *kill-ack* that indicates that the victims are all undone.
2. Y responds with a *kill-ack* that indicates that the transaction is unknown to it.
3. Y fails to respond.

In either of the last two cases, X should treat Y as dangerous, but then itself send a "victims all undone" *kill-ack*. A "transaction unknown" *kill-ack* indicates that the site has crashed and later recovered with no memory of the transaction.

There is no need to concoct an explicit acknowledgement for *died*. The fact that the abort source received a *died* is implicitly acknowledged when the abort source receives a *kill*. If no *kill* is received after a reasonable number of *died* retransmissions, the sending site should abort the entire family. This rule implies that top-level abort will be the result of a network partition that separates any of the sites along the path that *died* messages take from the abort initiator to the abort source.

3.4.3. Pruning Extra Kill Messages

If TIDs fail to encode nesting information, then comparing TIDs will yield no clue about how the different transactions are related to one another. This lack of knowledge is a source of extra *kill* messages. Consider several transactions all related as ancestor and descendant, all active at Site X, and all created at a site other than X. If each spreads from Site X to Site Y, then during abort Site X will send one *kill* for each transaction simply because X's transaction manager has no way to know that the transactions are related. The first *kill* will cause victims to be undone, and the corresponding *kill-ack* will indicate this. *Kill-acks* sent in response to later *kill* messages should likewise indicate that the victims are undone. But, because of the restriction that a transaction may be undone only once, these "late" *kills* must not cause a

second undo.

Thus a site must remember aborted transactions long enough to be able to properly handle late *kill* messages. The simplest way to remember aborted nested transactions for an adequately long period of time is to not forget them until top-level commit or abort, at which time the entire family is forgotten.

Although wasteful of network bandwidth, the duplication of *kills* increases the chance that the abort will spread to all sites. This is a property not to be discounted, since distributed abort is often triggered by a failure. Any technique for improving the efficiency of nesting-tree traversal must be designed carefully, since straightforward correctness arguments are constructed around the fact that the abort protocol traverses the path of every inter-site operation.

To alleviate the problem that — in general — a transaction manager cannot know the intra-familial relationship between any two transactions active at its site, the complete nesting history of a transaction can be piggybacked on the first call to new site. Now every transaction manager is ensured of knowing the relations among transactions active at its site. Using this information, extra *kills* can be pruned out of the traversal of the nesting tree. The protocol is changed so that if some site has already sent a *kill* to another site on behalf of one transaction, then *kill* is not resent to the same site by any more lowly nested transaction. The pruning is massive: a site sends a *kill* to any other site only once.

The TID argument of the *kill* message is reinterpreted to mean: undo all transactions at or below this level. With this change, another action (in addition to freezing) must be taken while performing the local downward traversal of the nesting subtree rooted by the *kill* argument. The traversal procedure should develop a list of <site, transaction> associations, one for each site that any of the transactions had spread to. The transaction associated with a site is the most highly nested one that had spread to it. When the traversal procedures turns around and climbs back up the tree, the *kill* sent to a site lists its associated transaction as the argument.

Giving every site a complete picture of the nesting relations of its active transactions preserves the completeness of the protocol despite the pruning. Now, the only source of duplicate *kills* is when different sites send *kills* pertaining to the same subtree. There are two cases to consider:

1. The first *kill* received is for the most highly nested transaction. If so, then all transactions will be undone as a result of the first *kill*.
2. The first *kill* received is *not* for the most highly nested transaction. In this case, the first *kill* will undo only a portion of the subtree. To preserve the only-once recovery restriction, the tree traversal caused by later *kills* should stop at the point(s) where an earlier *kill* started.

In either case, later *kills* will be handled properly, thanks to the "already aborted" check performed during local downward traversal.

With the pruning optimization, the purpose of the *died* message is limited to only determining which transactions are active; it is no longer useful in determining the intra-family connections among nested transactions.

3.5. Abort Protocol as Exception Mechanism

Experience has shown [16, pp. 123-126] [1, p. 288] that it is desirable for the abort protocol to be a distributed exception mechanism. This is easily done by having the abort initiator supply a "reason for aborting" with its request to abort. This information is then propagated to every site in *died* and *kill* messages. If the aborting process is different from that which created the transaction, then the creator should be sent a notification message giving the cause of the abort. When such a message is received, control is transferred to the end of the transaction. Since *kills* disseminate from abort root, if there are several simultaneous aborts then the reason for aborting is determined by which *died* message is the first to reach the abort source.

3.6. Aborting Top-Level Transactions

If a failure prevents a *died* (*danger*) message from being delivered to the abort (commit) source, nested abort risks ***blocking*** until the failure can be repaired. To prevent blocking, the nested abort becomes a top-level abort; therefore, the liveness of the abort mechanism depends on always being able to perform non-blocking abort of the entire family. Fortunately, because of the crash detection property of the communication subsystem, this is easily done.

To abort a top-level transaction, the initiator first undones the family locally; it then sends a *died* message directly to the family's creation site. (This site is guaranteed to be the abort source.) The initiator next sends *kill* messages to all sites that it knows the family spread to, then forgets about the family. It is permissible to forget immediately even if any combination of *died* and *kill* messages are lost. Once the abort initiator has forgotten the family, the site is indistinguishable from one that has crashed and recovered. Accordingly, the crash detection mechanism will prevent another operation from executing and committing at the site. Thus, once any site has aborted a top-level transaction, it will not commit. A simple case analysis presented in Tables 3-1 and 3-2 demonstrates that — no matter which type of message is next sent — a crashed or recovered site will always be detected, and orphans will never commit.

Aborting top-level transactions is much simpler for two reasons. First, because of the crash detection mechanism, orphaned work cannot commit: the abort initiator is sure to vote no to any request to commit. Second, the abort root is trivially identified, so there is no need to send died messages. The *kill-complete* message is eliminated as well, and the utility of a *kill-ack* is limited to increasing the reliability of *kill* delivery.

Note that the crash detection mechanism must employ a timestamp LWM rather than a simple crash count (as used in [11]) because "crash detection" is needed to

NEXT MESSAGE	RESULT
Operation request	Sender times out and aborts.
Operation response	If crashed site is commit source, all other sites must eventually time out and abort orphans; otherwise, sender times out and aborts.
Kill	Sender fails to receive kill-ack, times out and reports crashed site as dangerous.
Kill-ack	If crashed site is commit source, there will be no top-level commit; otherwise, site that sent kill times out and reports this site as dangerous.
Kill-complete	Because abort call doesn't return, enclosing transaction doesn't commit. Site that sent operation request times out and aborts.
Died	Sender fails to receive kill, times out and aborts top-level.
Danger	Site is commit source: there will be no top-level commit.
Danger-ack	Site is not commit source and has not yet sent its kill-ack: site that sent kill to it times out and reports it as dangerous.

Table 3-1: Sending to Crashed Site

The next message is either an operation request or response, one of the messages of (some) commit protocol, or one of the messages of the abort protocol. Specification of the commit protocol is beyond our scope. The table shows the result in the other cases.

detect a site that did not crash but rather performed a top-level abort and forgot.[5]

3.7. Preventing Orphans Entirely

If the communication subsystem provides the right sort of crash detection via keepalive messages[6] then the abort mechanism will prevent orphans entirely.

If a site's communication subsystem "pings" all other sites that have communicated with it (i.e., sent a message to it or received a message from it), then when some site crash or network partition occurs, all sites that have sent or received messages across the failed component will be promptly informed of the failure. Each site will then initiate the abort of all transactions that communicated across the failure. Thus a site crash *does not* result in losing the knowledge of which sites the transaction has spread to beyond the crashed one. The abort protocol "continues" on the "other side" of the failure instead of being blocked.

---

[5]Just as with a crash count, production of locally-generated timestamps requires an occasional write to stable storage to ensure monotonicity. The alternative is a real-time timestamp which could be obtained from something like an NTP [14] time server.

[6]This requires that the interval between successive keepalives must be shorter than the time in which a site can recover. Currently, typical sites take minutes to recover.

NEXT MESSAGE	ACTION	RESULT
Operation request	If crash detection check is made on requests as well as responses, operation will be rejected. Otherwise, operation wil be performed and crash will be detected (and abort initiated) when response is received.	If caller is still up, its crash detection detects inconsistency and initiates abort. If caller is down, operation becomes (possibly isolated) orphan; will abort via local server timeout or via AP. Same if caller crashed and recovered.
Operation response	Response rejected by crash detection	Abort initiated because of crash detection.
Kill	Send "transaction unknown" kill-ack.	Sender will report this site as dangerous.
Kill-ack	Ignore.	Site that sent kill to this one times out and reports this one as dangerous.
Kill-complete	Ignore.	Because abort call doesn't return, enclosing transaction doesn't commit. Site that sent operation request times out and aborts.
Died	Ignore.	Sender will time out and abort top-level.
Danger	Ignore.	Site is commit source: there will be no top-level commit.
Danger-ack	Ignore.	Site is not commit source and has not yet sent its kill-ack: site that sent kill to it times out and reports it as dangerous.

Table 3-2: Sending to Recovered Site

One advantage of coupling low-level failure detection with an "eager" abort protocol is that a much stronger guarantee can be made about orphan elimination: there will be no orphans — the abort protocol will locate and undo all operations. Another major advantage is that there is no longer a need to require servers to abort long-running transactions: the transaction is guaranteed to abort if abort is requested or if there is a failure, and these are exactly the circumstances under which it should abort. So there is no need to bound transaction lifetime. Of course, servers may continue to do so as part of their resource control policy, but it no longer need be required by the abort mechanism. The disadvantage of aborting when a lower layer reports a failure is that a partition — even a transient one — causes abort.

4. Informal Correctness Arguments

This section offers informal arguments that the abort mechanism is both *safe* and *live*. In the context of an abort mechanism, safety means that no orphaned operation ever commits, and liveness means that all aborted operations do eventually abort.

4.1. Safety

The skeleton of the safeness argument is:

1. The crash detection algorithm guarantees that the top-level transaction cannot commit if any site crashed or aborted the top-level transaction.
2. Because of the crash detection guarantee, top-level abort is safe. It is guaranteed that if the top-level transaction aborts anywhere at any time, then it will commit nowhere.
3. Use of a dangerous site is a necessary-but-not-sufficient condition for an operation to be an orphan.
4. Therefore, if the top-level transaction aborts whenever danger is detected, no orphan will commit.

These statements are elaborated below.

Crash detection. Simple arguments show that once a transaction's LWM is forgotten at a site, then if the transaction returns there, that fact will always be revealed by LWM mismatch. The reason is that when a forgotten transaction returns to a site, its LWM will initially be missing there and then will be set to a value higher than any previously recorded LWM for that transaction at that site. If LWMs are recorded only in response messages and LWM matching occurs only at the commit source, then a "crash" will always be detected at the commit source when the LWM arriving on a message is greater than that recorded earlier. If LWMs are recorded in both requests and responses, then mismatch will be discovered at the site that crashed if the sender had previously communicated with that site. Table 4-1 enumerates what happens in each case of this scenario.

MESSAGE	PREV COMM?	WHERE DETECT	HOW DETECT
Request	Yes	destination	dest=missing, msg=Prev
Request	No	commit source	cs=Prev, msg=higher
Response	Yes	destination	dest=missing, msg=Prev
Response	No	commit source	cs=Prev, msg=higher

Table 4-1: Crash Detection

There are four cases: whether the message is a request or response, and whether the sender and destination have previously communicated. For each of these cases, the table indicates where the LWM mismatch will occur and what two values the LWMs will have. "Prev" denotes whatever LWM value was recorded at another site because of the first operation that executed at the destination before it forgot its LWM.

Top-level abort is safe. As part of "forgetting" about a family, its LWM will be expunged. So, from the point of view of the next message pertaining to that family that arrives at that site, the site is the same as one that had crashed and recovered or one where the family had never been active. As shown above, any attempt to reestablish any transaction within the family at the aborted site will be detected and lead to abort.

Aborting in case of danger overestimates orphans. By definition of the abort protocol, every orphaned operation will either be active at a dangerous site, or will have passed through a dangerous site on its way to the site where it is active. An orphaned operation is one that it is unreachable by the abort protocol. Since the protocol duplicates the pattern of operation calls, an unreachable operation is one that followed an inter-site path that the protocol cannot follow.

4.2. Liveness

The basic abort mechanism is live only in a degenerate sense. That is, if failures produce orphans, the only guarantee about their elimination is that eventually all orphans will be aborted by server timeout. So the abort protocol is simply a performance optimization to orphan elimination by timeout.

If the abort mechanism can depend upon failure reports from the communication subsystem as discussed in Section 3.7, then the abort mechanism is live because the abort protocol does — in the absence of failures — succeed in locating all operations of aborting transactions. Liveness in the absence of failures translates to overall liveness because, when a site crashes, the communication subsystem will report a failure to every site that had communicated with the failed site. This is what would happen were the site to remain up and send *kill* messages.

It is easy to see that, provided there are no failures, the abort protocol without the optimization of Section 3.4.3 will locate all operations. A *kill* message is sent along the path taken by any operation request. If the *kill* is lost, then retransmission will succeed in delivering it. If retransmission fails, then a failure has occurred.

It is also true that, provided there are no failures, the optimized abort protocol will locate all operations. Although the optimization reduces the number of *kills* sent, it remains the case that a site sends one *kill* to every other site to which any of its transactions spread. The *kill* argument is the transaction most highly nested at the sending site that also spread to the destination site. Therefore, at every destination site it is the case that some site will send it a *kill* which has *its* most highly nested transaction as the argument.

5. Evaluation

Beyond the obvious goal of correctness, there are many properties that an abort mechanism should have:

1. *Low overhead*: do not add to the overhead of normal processing only to facilitate aborting. Specifically,
    a. Do not add log writes.
    b. Do not add messages to the commit protocol.
    c. Add as little extra information as possible to messages or log records.
2. *Efficiency*: perform abort as efficiently as possible.
    a. Traverse the nesting tree and spreading graph as efficiently as possible.
    b. Do not unnecessarily abort enclosing transactions.
3. *Speed*: since the abort protocol is a performance optimization, it should be fast.
    a. Drop locks as fast as possible.
    b. Resume the process that invoked abort as soon as possible.
    c. Do not delay the commitment of an enclosing transaction beyond the return of the abort call of the nested transaction.
4. *Special cases of liveness*: there are several things that should be done faster than "eventually."
    a. Allow ***unilateral abort***, which means that unless a transaction is prepared, a site may abort the transaction without having first to communicate with any other site.
    b. Exterminate orphans as quickly as possible.[7]
    c. Continue operating in spite of failures. (Abort is often triggered because of failure.)

For top-level abort, some of these goals (such as 2b and 3c) are vacuous. It seems highly unlikely that any abort mechanism could satisfy all these goals completely and simultaneously, since meeting the overhead and efficiency goals denies the mechanism the information it would need to accomplish other goals such as 2b and 4b, which require precise and timely identification of orphans.

The abort mechanism described in this paper completely satisfies all the goals except these, which are mostly, but not completely, satisfied:

1. *Overhead:*
    c. Extra information is added to messages, but its size is proportional to the number of sites involved in the transaction, not to the (possibly much larger) number of transactions in the family.
2. *Efficiency:*
    a. In general, more than the optimal number of messages (*kills* and *kill-acks*) are sent while traversing the nesting tree and spreading graph.

---

[7]The Argus orphan elimination algorithm provides a stronger and more quantified guarantee: an orphan will be eliminated before it can read items whose values were read earlier by another transaction and passed to it as arguments [12, 6].

b. The policy of aborting the top-level transaction if any active operation used a dangerous site aborts more than just orphaned operations.

3. *Speed*:

b. The abort call is synchronous; that is, the caller does not regain control until the protocol has finished.

4. *Liveness guarantees*:

b. The only guarantee made about orphans is that they will never commit.

The following sections discuss — and to some extent justify — the reasons underlying the failure to meet these goals.

5.1. Overhead

An abort mechanism must be able to distinguish between orphaned and non-orphaned operations by the time the top-level transaction attempts to commit. To do so in a model that places no limits on transaction lifetime, nesting, or distribution, either extra messages must be sent or else information must be added to at least some existing messages in order that sites that are aware of the existence of orphans can inform the other sites. Appendix I contains a detailed statement of what information must be maintained by a transaction manager.

A presumption of this work is that a transactional computation is more likely to consist of a very large number of transactions than a very large number of sites. Hence, the abort mechanism was designed so that the amount of information piggybacked on messages was not proportional to the number of transactions within the family. The simplified implementation of information accumulation described in Section 2.4.2 violates this principle, and so should be used with care.

5.2. Efficiency

Efficiency of tree/graph traversal. Despite the pruning optimization of Section 3.4.3, the abort protocol still sends more than the optimal number of *kill* messages. These extra messages arise because of the speed goal, which dictates a *parallel* traversal of the tree/graph. An optimal message count could be obtained by traversing the nesting tree in a more orderly fashion; e.g., by breadth-first search.

Using a message-optimal protocol would likely not be a wise tradeoff of speed versus efficiency. At the present time, the wasted network bandwidth is not likely to be a severe problem, since distributed transactions typically do not involve many sites, and since aborts are rare. It is conceivable that experience will show that distributed operation increases the percentage of aborted transactions; for example, nested transactions may enclose RPCs to read replicated data from several sites, with the slower calls being aborted once the required number of calls return. If the percentage of distributed transactions that abort is not negligible, then the resource consumption of

the abort protocol should be reexamined.

Extent of unnecessary aborts. If the sites used by different nested transactions are disjoint, then the top-level transaction can commit even if some nested transactions failed to abort completely. Ensuring this property was an explicit design objective, because of the pleasant consequence when nested transactions are used to enclose single-site RPCs: the failed abort of an RPC will not prevent the enclosing transaction from committing. An example is:

1. Site A (which is the commit source, for simplicity) makes calls within nested transactions to replicated data at sites B, C, and D.
2. Sites B and C respond, and since 2 out of 3 sites satisfies the read quorum condition of the data replication method, the nested transaction enclosing the call to Site D is aborted.
3. Coincident with the abort, Site D crashes, so kill messages sent from A to D are not acknowledged.

In this case, Site D will be recorded as a dangerous site. When commitment happens, a "dangerous prepare" is sent from A to B and C. Since neither has any record of communicating with D, both vote yes, and the transaction commits. This sort of situation may be common if the use of replicated data is popular, and if remote procedure calls are wrapped within nested transactions to facilitate failure isolation. Argus does this, as does Avalon [2].

Retention of data structures. Although aborted families are forgotten immediately, aborted nested transactions are not. The overhead of retaining the descriptors of aborted nested transactions is small, since there are likely to be many more committed nested transactions (which must be remembered anyway) than aborted ones.

5.3. Speed

The performance of an abort protocol is variable since it depends upon the depth of nesting, the extent of spreading, and the amount of work to undo at each site. Roughly speaking, the latency of an abort in which no failures occur is proportional to the diameter of the spreading graph. *Kill* messages spread outward until the site farthest from the abort source has received one. The latency of returning to the synchronous abort call is the cost of this kill phase plus the variable cost of sending *died* messages to locate the abort source plus sending a *kill-complete* back to the initiator.

Because abort is assumed to be rare, to an extent its performance is not one of the important parameters of a transaction facility. It is far more important that the abort mechanism intrude as little as possible on the typical behavior of the system (i.e., failure-free commitment). In this sense, the abort mechanism succeeds: all processing related to aborting a particular transaction happens only after the abort call has been invoked.

5.4. Special Liveness Guarantees

Unilateral abort. Unilateral abort is a somewhat abstruse but traditional goal [10]. Its motivation is that a command to abort a transaction may represent an "emergency" at the aborting site, and that any unnecessary delays in undoing the transaction at that site are intolerable.

Orphan elimination. The abort mechanism offers orphan *protection* more so than orphan *elimination.* Orphans *at sites which are involved in the commit protocol* do not survive beyond the commit protocol. Orphans at isolated sites live until their servers decide to abort them for having lived too long. Therefore, there is no elegant guarantee of when orphans disappear except that, because of one of the assumptions, they eventually do.

If the communication subsystem provides crash detection via keepalive messages then a much stronger statement can be made: there will be no orphans; the abort protocol will locate and undo all operations. The reason is that the sites on the "other side" of the failure will detect the failure and themselves initiate (additional) aborts. For this reason, it is a major advantage for an abort mechanism to include an abort protocol.

5.5. Effect of Restrictions

The only-once recovery restriction is an artificial one, imposed to make the work more general. Each of the other four restrictions of Section 2.3 is essential.

1. As explained, the correctness of the mechanism depends upon the synchronicity restriction: all dangerous sites must be recorded before the enclosing commit can be allowed to proceed. Thus, the abort call must be synchronous.

2. Dangerous sites must be stored (in memory) somewhere such that, if the information were lost, there would be no attempt to commit the top-level transaction. This is the definition of the commit source and reason that such a site is required to exist.

3. The need for any site to be able to locate either the commit source (in order to report danger) or the abort source (in case of top-level abort) forms the requirement that the address of both of these sites appear in the TID.

4. The requirement that servers abort long-running transactions appears to be fundamental to the transaction model. The Argus abort mechanism discussed below seems to suggest that the alternative is to retain and systematically communicate an unbounded amount of information so that any orphan, no matter how old, will eventually be detected. The optimized versions of the Argus mechanism bound the amount of information that must be maintained and added to messages, but they depend on limiting orphan lifetime in order to do so.

Of these restrictions, the abort-on-timeout is the most bothersome. Fortunately, it can be removed provided that the communication subsystem reliably detects failures, a feature that is common. The remaining restrictions are not very constraining, and

so the abort mechanism should be applicable to a wide variety of domains.

6. Related Work

Every transaction facility must have some way of aborting transactions, so one would except a well-developed body of literature on the problem. However, implementations of powerful nesting models are rare. Because of this and because the only aspect of aborting non-nested, non-distributed transactions is the recovery algorithm, builders of transaction systems have rarely written about abort. There is only one set of documents that consider in detail how to perform abort within a Moss-based nesting model: namely, those written by the Argus group [11, 17, 12, 6] and derivative works that explore the same idea [13, 7]. Since Argus is a programming language, the goal of its abort mechanism is to provide a bound on orphan lifetime that would be useful as a basis for defining language semantics.

The Argus abort mechanism differs from the one discussed here in that there is no abort protocol; there is only an orphan protection scheme. Argus' abort call returns immediately after undoing the victim only within the local "Guardian."[8] Therefore, the abort call systematically creates orphans, both up the nesting tree (an *up-orphan* is an ancestor of a committed nested transaction that aborts) as well as down (a *down-orphan* is simply the descendant of an aborted transaction). Site crashes — even if detected by keepalives — also create orphans, since there is no abort protocol to spread notice of the detected crash. This is a distinct disadvantage compared to our abort mechanism, which can take advantage of a detected site crash: all sites that had been in communication with the lost site will initiate the abort protocol, which ensures that no orphans exist.

The major advantage of the Argus mechanism is that it guarantees that an orphan will be terminated "by the time it should be:" specifically, before it can read values from the database that were read earlier by another (intra-family) transaction and passed to the orphaned operation in its arguments. This situation can arise if data items are replicated and the replicas must satisfy an invariant. The danger is that a snapshot taken at one replica may be in conflict with a snapshot taken earlier at another replica if, during the interim, a third transaction is able to obtain locks at both replicas and make a change. If the orphan were to see the invariant violated, then languages primitives may fail to work correctly.

This possibility is avoided because the aborting Guardian disseminates notice of the dead transaction in all future messages (including those of the commit protocol). All other Guardians repeat this notice in the information they add to their outgoing messages. Any flow of messages that could cause the locks of a first transaction to be dropped at one replica, followed by commitment of an intervening transaction at all replicas, followed by a read at some replica by an orphan, will necessarily also spread to all replicas the identity of the orphan. Hence the orphan's attempted read can be

---

[8] A Guardian is a combined server and transaction manager.

stopped, and the orphan terminated. Similar but simpler thinking applies to eliminating crash-created orphans.

To detect both kinds of orphans, every Guardian maintains the following information and transmits it with *every* message (including those of the commit protocol):

- A list consisting of the roots of all subtrees ever aborted. Argus TIDs encode the list of all ancestors of the transaction, so whether two transactions are related as ancestor and descendant can easily be deduced from comparing the two identifiers. The TID of the abort root identifies all down-orphans.
- The list of all guardians used by a transaction and its committed descendants, their age (as measured by a crash count recorded in stable storage) at the time they were last used, and the age of every guardian that ever existed.

This information is a complete history of all aborts and crashes; it is maintained across crashes by recording it in every "prepare" log record. To detect down-orphans, whenever a new call is received all old calls are compared against the received list of aborted subtrees, and received calls are compared against the old list. Any match identifies an operation that must be terminated. To detect up-orphans, all old calls are compared against the received list of guardian ages, and received calls are compared against the old list. Any mismatch identifies an orphan.

The piggybacked information required by this design is enormous, so two optimized designs exist. In both cases, in order to limit the amount of information piggybacked onto messages, limits must be imposed on transaction lifetime. The first of these optimizations is "time-driven orphan elimination," an idea which is described also in [13, 7]. To have time-driven orphan elimination, the execution model must limit transactions to finish before a *quiesce* time lest they be aborted by a later *release* time. Orphans are guaranteed to be eliminated by the release time and so messages must carry only enough extra information to detect orphans that may exist between the present moment and their release time. For those cases when the quiesce and release times are defined to be too soon, a two-phase "refresh protocol" exists to push the deadlines back. The second optimized version of the Argus abort orphan elimination algorithm also places limits on transaction lifetime, but it drastically reduces the information added to messages. The cost is that the information needed to detect orphans must be kept a central repository; Guardians exchange information with the repository in the background. The repository must be replicated for high availability. The existence of a replicated repository does not recreate the problem it was intended to solve because the consistency constraint is weak, and so the repository does not require transactional update.

We argue that the abort mechanism developed in this paper is superior to that of Argus. Argus provides very quick partial abort and then depends upon its "strong" but higher-overhead orphan elimination mechanism. Our method presumes that both aborts and orphans are rare, and so makes a tradeoff that more properly assigns the

cost of aborting and orphan protection to the aborting transactions and places less burden on the normal events: inter-site communication and commitment. Table 6-1 summarizes and compares the overhead of our mechanism with the first optimization of the Argus algorithm. Furthermore, our abort mechanism can take advantage of a communication subsystem's failure detection feature in order to ensure the complete absence of orphans. This is a consequence of having an abort protocol that performs "eager" orphan elimination as opposed to the "lazy" method of elimination in which an orphan is not detected until it calls a Guardian that knows it is an orphan.

	ABORT MECHANISM	ARGUS
Added to req/resp	site id, timestamp for every transaction used in call	all aborted transactions with less than certain lifetime; site id, timestamp for every transaction used in call
Added to other msg	prepare msg: dangerous sites	same as above
Placed in stable storage	none	same as above
Req/resp processing	accumulate sites & compare timestamps with those stored	accumulate aborted transactions, sites, and timestamps
Other msg processing	prepare msg: compare dangerous sites with sites visited	accumulate aborted transactions, sites, and timestamps

Table 6-1: Overhead Comparison

7. Summary

We have presented a method of aborting transactions that my be arbitrarily nested and distributed. The mechanism possesses many desirable features:

- The mechanism is operates within a general transaction model and a realistic failure model.
- Provided that the communication subsystem provides prompt failure detection, there will be no orphans.
- A site can abort unilaterally.
- The mechanism imposes relatively few and relatively minor restrictions on the transaction facility, and so promises to be portable.
- Little overhead is imposed on transaction-operation messages.
- No information need be maintained in stable storage.
- The abort protocol never blocks.

The primary disadvantages of the mechanism are that its abort protocol must be synchronous, that it may over-abort in some cases, and that — if the communication subsystem *does not* provide prompt failure detection — there is no limit on the extent or lifetime of orphaned computations.

8. Acknowledgements

Lily Mummert and Dean Thompson discovered many ambiguities in the first description of the mechanism, and collapsed the protocol from two phases to one, pointing out that having a second "forget" phase is a poor tradeoff. Lily helped implement the protocol in Camelot.

I. Required Information

The transaction manager maintains a descriptor for every transaction; this decriptor must contain:

- Transaction state (with values such as active, committed, and aborted). The initial state is active. The state should be changed when abort begins, so that any attempt to perform another abort can be detected and stopped.
- The identity of the parent transaction, if any.
- The identity of the transaction's children, if any.
- The site, if any, *from* which the transaction spread to this one.
- A list of sites spread *to*, if any. This information is both forward-accumulated on requests and backward-accumulated on responses.
- A list of the local servers involved in the transaction.
- If the transaction is top-level: a list of dangerous sites, if any.
- The id of the process that began the transaction.
- The id of the process that aborts the transaction.
- The reason for aborting.
- A count of the number of kill-ack messages awaited.
- A count of the number of danger-ack messages awaited.
- Its complete nesting history. This information is needed only for the optimization of Section 3.4.3.

Additionally, the transaction manager must also maintain information associated with the family (for commitment) and a mapping of site-to-LWM (for crash detection).

There are six messages sent during the abort protocol. They should contain the following information:

- Died:
    1. The transaction that must die.
    2. The reason for dying.
    3. The original target transaction at the abort initiator.
    4. The identity of the site that originally initiated abort.

  The last two items are needed to that the abort root can return a *kill-complete*.

- Kill:
    1. The root of the subtree to be aborted.
    2. The reason for dying.

- Kill-ack:
    1. The root of the subtree that was to have been aborted.
    2. The return code, which indicates either that the abort root and everything below it were undone, or that the root is unknown.

- Kill-complete:
    1. The original abort target (sent to the abort root from the abort source by a series of died messages).

- Danger:
    1. The family id.
    2. A list of dangerous sites.

- Danger-ack:
    1. The family id.

References

[1] M. R. Brown, K. N. Kolling, and E. A. Taft.
The Alpine File System.
ACM Trans. on Computer Systems 3(4):261-293, November, 1985.

[2] D. L. Detlefs, M. P. Herlihy, and J. M. Wing.
Inheritance of Synchronization and Recovery Properties in Avalon/C++.
IEEE Computer 21(12):57-69, December, 1988.

[3] D. Duchamp.
Transaction Management.
PhD thesis, Carnegie Mellon Univ., December, 1988.
Available as Technical Report CMU-CS-88-192.

[4] D. Duchamp, et. al.
Design Rationale of the Camelot Distributed Transaction Facility.
Technical Report CUCS-008-90, Columbia Univ. Computer Science Dept., March, 1990.

[5] J. N. Gray, et. al.
The Recovery Manager of the System R Database Manager.
ACM Computing Surveys 13(2):223-242, June, 1981.

[6] M. Herlihy, N. Lynch, M. Merritt, and W. Weihl.
On the Correctness of Orphan Elimination Algorithms.
In *Proc. 17th Intl. Symp. on Fault-Tolerant Computing*, pages 8-13. IEEE, July, 1987.

[7] M. Herlihy and M. S. McKendry.
Timestamp-Based Orphan Elimination.
Technical Report CMU-CS-87-108, Carnegie-Mellon University, December, 1987.

[8] L. Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
Comm. ACM 21(7):558-565, July, 1978.

[9] B. Lindsay et. al.
Notes on Distributed Databases.
Technical Report RJ2571, IBM Almaden, July, 1979.

[10] B. Lindsay et. al.
Computation and Communication in R*: A Distributed Database Manager.
ACM Trans. on Computer Systems 2(1):24-38, February, 1984.

[11] B. Liskov.
Progress Report of the Programming Methodology Group.
In *MIT LCS Progress Report*, pages 142-176. MIT Press, 1984.

[12] B. Liskov, R. Scheifler, E. Walker and W. Weihl.
Orphan Detection.
In *Proc. 17th Intl. Symp. on Fault-Tolerant Computing*, pages 2-7. IEEE, July, 1987.

[13] M. S. McKendry and M. Herlihy.
Time-Driven Orphan Elimination.
In *Proc. Fifth IEEE Symp. on Reliability in Distributed Software and Database Systems*, pages 42-48. 1986.

[14] D. Mills.
Network Time Protocol (Version 1) Specification and Implementation.
Technical Report RFC 1059, Network Working Group, July, 1988.

[15] J. E. B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
MIT Press, 1985.

[16] R. Pausch.
Adding Input and Output to the Transaction Model.
PhD thesis, Carnegie Mellon Univ., August, 1988.
Available as Technical Report CMU-CS-88-171.

[17] E. Walker.
Orphan Detection in the Argus System.
Master's thesis, MIT, June, 1984.
Available as MIT LCS Technical Report 326.