Delegation: Efficiently Rewriting History†

Cris Pedregal Martin and *Krithi Ramamritham*
Department of Computer Science
University of Massachusetts
Amherst, Mass. 01003–4610
{*cris,krithi*}*@cs.umass.edu*

Technical Report 1995-090
October 26, 1995

Abstract

The notion of transaction delegation, as introduced in ACTA, allows a transaction to transfer responsibility for the operations that it has performed on an object to another transaction. Delegation can be used to broaden the visibility of the delegatee, and to tailor the recovery properties of a transaction model. Delegation has been shown to be useful in synthesizing Extended Transaction Models.

With an efficient implementation of delegation it becomes practicable to realize various Extended Transaction Models whose requirements are specified at a high level language instead of the current expensive practice of building them from scratch. In this paper we identify the issues in efficiently supporting delegation and hence extended transaction models, and illustrate our solution in ARIES, an industrial-quality system that uses UNDO/REDO recovery. Since delegation is tantamount to rewriting history, a naïve implementation entails frequent and costly log accesses, and complicates recovery protocols. Our algorithm achieves the effect of rewriting history without rewriting the history, i.e., the log, resulting in implementations that realize the semantics of delegation at minimal additional overhead and incur no overhead when delegation is not used. Besides showing its efficient application to ARIES, we also show the correctness of the implementation of delegation.

Our work indicates that it is feasible to build efficient and robust, general-purpose machinery for Extended Transaction Models. It also leads toward making recovery a first-class concept within Extended Transaction Models.

Keywords: Extended Transaction Models, Transaction Management, Recovery.

† Supported in part by grants from Sun Microsystems and the National Science Foundation.

Contents

1 Introduction

The transaction model adopted in traditional database systems has proven inadequate for novel applications of growing importance, such as those that involve reactive (endless), open-ended (long-lived), and collaborative (interactive) activities. Various *Extended Transaction Models* (ETMs) have been proposed [10], each custom built for the application it addresses; alas, no one extension is of universal applicability. To address this problem, we have been investigating how to create general-purpose and robust support for the specification and implementation of diverse Extended Transaction Models. Our strategy has been to work from first principles, first identifying the basic elements that give rise to different models and showing how to realize various Extended Transaction Models using these elements, and then proposing mechanisms for implementing these elements.

A first step was ACTA [6], that identified, in a formal framework, the essential components of Extended Transaction Models. In more operational terms, ASSET [5] provided a set of new language primitives that enable the realization of various Extended Transaction Models in an object-oriented database setting. In addition to the standard primitives Initiate (to initialize a transaction), Begin, Abort, and Commit, ASSET provides three new primitives: *form-dependency*, to establish structure-related inter-transaction dependencies, *permit*, to allow for data sharing without forming inter-transaction dependencies, and *delegate*, which allows a transaction to transfer responsibility for an operation to another transaction.

Traditionally, the transaction invoking an operation is also responsible for committing or aborting that operation. Delegation separates these two concerns, so that the invoker of the operation and the transaction that commits (or aborts) the operation may be different. In effect, to delegate is to *rewrite history*, because a delegation makes it appear as if the delegatee transaction had been responsible for the delegated object all along, and the delegator had nothing to do with it.

Delegation is useful in synthesizing Extended Transaction Models because it broadens the visibility of the delegatee, and because it controls the recovery properties of the transaction model. The broadening of visibility is useful in allowing a delegator to selectively make tentative and partial results, as well as hints such as coordination information, accessible to other transactions. The control of the recovery makes it possible to decouple the fate of an update from that of the transaction that made the update; for instance, a transaction may delegate some operations that will remain uncommitted but alive after the delegator transaction aborted. Examples of Extended Transaction Models that can be synthesized using delegate are Joint Transactions, Nested Transactions, Split Transactions, and Open Nested Transactions [6, 11].

Biliris et al. [5] gave a high-level description of how to realize the three new ASSET primitives. Briefly, permit is done by suitably adding the permittee transaction to the object's access descriptor. Form-dependency is done by adding edges to the dependency graph, after checking for certain cycles. Whereas the realization of permit and form-dependency are rather

straight-forward, close attention must be paid to logging and recovery issues in the presence of delegation. This is because recovery usually keeps some kind of system history (e.g., log) and delegation is tantamount to *rewriting history* (a delegated object's operations appear to have been done by the delegatee).

Developing a robust, efficient, and correct implementation of delegation is the goal of this paper.

Specifically, to further the goal of providing general purpose machinery to support the specification and implementation of arbitrary Extended Transaction Models, we propose here an efficient implementation of delegation based on ARIES [14].(Also, we briefly suggest how to implement delegation on EOS [4].) Our additions allow the "efficient Rewriting of History." We hence call our implementation ARIES/RH.

By providing delegation, we add substantial semantic power to a conventional Transaction Management System, allowing it to capture various Extended Transaction Models. We efficiently achieve this expressiveness by carefully "piggy-backing" the delegation-related processing onto the routine processing. During recovery, our algorithm neither adds costly log sweeps to the recovery algorithm, nor does it demand the actual rewriting of history, i.e., the log.

In this paper we argue that:

- Delegation is a powerful, important primitive for realizing Extended Transaction Models. We describe its properties and show how it can be used to manipulate visibility and recovery properties of transactions.

- It is possible to implement delegation in an industrial-strength transaction management system. We illustrate by extending ARIES. We thus obtain the ETM semantics with little loss of efficiency, and when delegation is not used no overheads are incurred. We also demonstrate the correctness of our algorithm.

The remainder of the paper is organized as follows. In Section 2 we describe the properties of delegation and show how it can be used to synthesize some well-known extended transaction models. In section 3 we develop delegation in the context of a robust, industrial-grade transaction management system. First we explain delegation's semantics in terms of rewriting history. We then discuss the needed data structures and describe how we modify both the normal processing and the recovery phases to support delegation, and explain how to apply our algorithm to ARIES. We conclude the section sketching how to apply the same ideas to EOS, another transaction management system.

In section 4 we discuss why our algorithm correctly implements delegation and why it does it efficiently. We review related work in section 5. In section 6 we present our conclusions and discuss future work.

2 Delegation: Concepts, Properties, Examples

In this section we examine the properties of delegation and present its application to extended transaction models. First we introduce some notation, then we explain delegation in terms of visibility and recovery, and then point out some important properties. In the rest of the section we present examples of extended transaction models and show how to synthesize them using delegation.

2.1 What: Concepts and Properties

Here we describe the properties of delegation, introduce notation and state our assumptions.

2.1.1 Assumptions and Notation

- $t, t_0, t_1, t_2, ...$ denote ***transactions***; $ob, a, b, ...$ denote ***objects*** in the database.
- ***update*** is a generic operation on database objects. We write $update[ob]$ and leave other details of the update unspecified. Updates are done in-place on the updated object. Note that not all update operations conflict with each other.
- $delegate(t_1, t_2, update[ob])$ denotes ***delegation*** by t_1 to t_2 of $update[ob]$.
- ***Invoking transaction.*** We call the transaction that invoked the update on the object the *invoking* transaction. We write $update[t, ob]$ when we wish to indicate that t is the invoking transaction for that update. Updates that are never delegated, i.e., whose responsible transaction (see below) is *always* their invoking transaction, are called *boring* updates.[1]
- H denotes the ***history*** of the database, which contains ***events*** such as *delegate* and *update*, with a partial order indicated $\epsilon \rightarrow \epsilon'$ where ϵ precedes ϵ'. Operation invocations are events.
- ***ResponsibleTr.*** Let transaction t update object ob. We say that t is *responsible for its updates to ob*, when t is in charge of changes to ob. More precisely, $ResponsibleTr(update[ob]) = t$ holds from when t performs $update[ob]$, or is delegated $update[ob]$ until t either terminates or delegates $update[ob]$. Notice that without delegation, the transaction responsible for an update is always the invoking transaction.
- ***Op_List.*** The dual of $ResponsibleTr$ is the *Op_List*. It contains the operations a transaction is responsible for: $Op\_List(t) = \{update[ob] \mid ResponsibleTr(update[ob]) = t\}$.

2.1.2 Properties

Pre- and Postconditions. When t_1 executes $delegate(t_1, t_2, update[ob])$, we say that t_1 transfers its responsibility for $update[ob]$ to transaction t_2, i.e.,

- $pre(delegate(t_1, t_2, update[ob])) \Rightarrow (ResponsibleTr(update[ob]) = t_1)$
  t_1 must be the transaction responsible for $update[ob]$ in order to delegate the update.

[1] All updates are boring in the absence of delegation.

- $post(delegate(t_1, t_2, update[ob])) \Rightarrow (ResponsibleTr(update[ob]) = t_2)$
  After t_1 delegates $update[ob]$ to t_2, t_2 becomes the responsible transaction for the update.

Operation $delegate(t_1, t_2, update[ob])$ is well formed when t_1 and t_2 are initiated and not terminated, and t_1 is responsible for $update[ob]$.

Commit/Abort of Updates. In the presence of delegation, the fate of updates to an object is not necessarily linked to the transaction which made the updates, but instead it is linked to the fate of the transaction to which the operation was last delegated. For instance, if t_0 does $update[ob]$, then delegates $update[ob]$ to t_1, and t_0 subsequently aborts, the changes t_0 made to ob via $update[ob]$ will still survive if t_1 commits while it is still responsible for $update[ob]$, i.e.,

- $(Commit(t) \in H) \Rightarrow (\forall update[ob] \in Op\_List(t), (commit_t(update[ob]) \in H)) \wedge$
  $((\exists update[ob] \in Op\_List(t), commit_t(update[ob]) \in H) \Rightarrow (Commit(t) \in H))$
  That transaction t commits means that all of the updates in its Op_List must be committed. Notice that these are the updates for which t is responsible.
- $(Abort(t) \in H) \Leftrightarrow (\forall update[ob] \in Op\_List(t), (abort_t(update[ob]) \in H))$
  That transaction t aborts requires that all of the updates it is responsible for (i.e., those in its Op_List) will be aborted.

The events $Commit(t)$ and $Abort(t)$ denote the commit and abort of transaction t, and $commit_t(update[ob])$ and $abort_t(update[ob])$ indicate the permanence or obliteration of the changes done by $update[ob]$. In the presence of delegation, the changes may have been made by either t or other transaction(s) which eventually delegated $update[ob]$ to t.

Concurrent Delegations. An operation can be delegated only by the transaction that is responsible for it. Since $ResponsibleTr(update[ob])$, is at any given time, unique, only one transaction can delegate an operation at any point in time. Thus, while a history may contain two or more delegations of the same operation by different transactions, the delegations for the *same operation* cannot occur concurrently.

Granularity: delegating one operation vs. set of operations. In what we have discussed, a transaction delegates a single operation with each invocation of delegate. Delegation of a set of operations in a single invocation can be considered as the atomic invocation of multiple delegations, one for each of the operations in the set. Delegating an object is tantamount to delegating all the operations on that object.

In our implementation we consider the delegation of objects because in a majority of practical situations that we have come across, delegation occurs at the granularity of objects. Also, in the examples discussed in the next subsection, transactions delegate objects.

Note that it is possible for several transactions to update an object concurrently (say, when the updates commute). Delegation of one such operation by one of the concurrent transactions only delegates that transaction's operation on the object. The other transactions' operations are not affected. Similarly, when a transaction delegates an object, only that transaction's operations on the object are delegated.

Also note that a transaction can perform operations on an object even after it has delegated (an operation on) that object. Of course, since after delegation the system considers the delegated operations to have been done by the delegatee, a transaction's new operation may conflict with one of its own — one which has been delegated.

2.2 Why: Synthesizing Extended Transaction Models – Examples

In this section we motivate delegation through examples of its application in the synthesis of two extended transaction models, split/join transactions and nested transactions.

Inheritance in Nested Transactions [15] is an instance of delegation. Delegation from a child transaction t_c to its parent t_p occurs when t_c commits. This is achieved through the delegation of all the changes done by t_c to t_p when t_c commits. That is, all the changes that a child transaction is responsible for are delegated to its parent when the child commits.

A transaction can delegate at any point during its execution, not just when it aborts or commits. For instance, in Split Transactions [16], a transaction may *split* into two transactions, a splitting and a split transaction, at any point during its execution. A splitting transaction t_1 may delegate to the split transaction t_2 some of its operations at the time of the split. Thus, a split transaction can affect objects in the database by committing and aborting the delegated operations even without invoking any operation on the objects.

In the remainder of this section we show the code for split and nested transactions, synthesized using delegation and the other ASSET primitives. Other transaction models using delegation include Reporting Transactions and Co-Transactions described in [7, 8]. The former periodically reports to other transactions by delegating its current results. In the latter, control is passed from one transaction to the other transaction at the time of delegation.

2.2.1 Split Transactions

In the split transaction model [16] a transaction t_1 can *split* into two transactions, t_1 and t_2. Operations invoked by t_1 on objects in a set *ob_set* are delegated to t_2. t_1 and t_2 can now commit or abort independently. Conversely, two transactions, say t_1 and t_2 can *join* to form one transaction t_1.

Consider the following code used by t_1 to split off transaction t_2 (the code for t_2 is that of function f.)

```
t2 = initiate(f);
delegate(self(), t2, ob_set);          // self returns t1
begin (t2);
```

t_2 can join t_1 by executing:

```
wait (t2);
delegate (t2,t1);                      // t2 delegates *all* objects
```

2.2.2 Nested Transactions

Nested transactions are among the first extended transaction models; they are discussed by Moss [15]. A nested transaction consists of a *root* (or parent) transaction and *nested* component transactions, called subtransactions. The subtransactions can themselves be nested transactions. Subtransactions execute atomically with respect to their siblings, and are failure atomic with respect to their parent. That is, they can abort without causing the whole transaction to abort.

A subtransaction can potentially access any object that is currently accessed by one of its ancestor transactions without creating a conflict. Abort semantics for both root transactions and subtransactions are similar to abort semantics in atomic transactions. Commit, however, has different semantics for the root and the subtransactions. When a subtransaction commits, the objects modified by it are made accessible to its parent transaction. The effects on objects are only made permanent on the commit of the topmost root transaction.

We illustrate how nested transactions are translated into the ASSET primitives with a simple two-level example of trip arrangements.

```
tid t;
t = trans {
    trans { airline_res(); }
    trans { hotel_res();   } }
```

If the airline reservation fails, then the trip is canceled. If the hotel reservation fails, the trip is canceled too, and the effects of the airline reservation should not be made permanent. The nested transaction above is translated into:

```
tid t;
t = initiate(trip)
begin(t);
commit(t);
```

where the function `trip` is

```
void trip()
{
tid t1;
t1 = initiate(airline_res);
```

```
permit (self(),t1);
begin(t1);
if (!wait(t1))
    abort(self());
delegate(t1,self());
commit(t1);

tid t2;
t2 = initiate(hotel_res);
begin(t2);
if (!wait(t2))
   abort(self());
delegate(t2,self());
commit(t2);
}
```

We assume that `t1` and `t2` will each abort if they are unsuccessful. If they succeed, they delegate their updates to `t`. Otherwise any updates made so far are discarded. Note that after it has delegated all its changes, the fate of a reservation transaction does not matter.

3 How: Rewriting History Efficiently

In this section we discuss how to efficiently implement delegation and present our algorithm RH (rewrite history), as follows. In 3.1 we introduce the operational semantics of delegation in the context of a generic Database System (DBS). In 3.2 we examine alternative solutions and give an overview of our algorithm.

In 3.3 we set the stage with an overview of ARIES, whose UNDO/REDO protocol requires two passes, one forward and one backward, over the log.

The following subsections explain the algorithm ARIES/RH in detail: we present the data structures involved in 3.4, then we describe in 3.5 what ARIES/RH does during normal processing. In 3.6 we discuss how ARIES/RH's recovery realizes delegation efficiently using the same passes over the log as ARIES.

We end the section in 3.7 with an overview of how to apply RH to a different recovery protocol, NO-UNDO/REDO as exemplified by EOS.

3.1 Operational Semantics

In a DBS the log *is* the system's history, as it contains the records of all updates and transactional operations. The idea of delegation is to *rewrite history*, selectively *altering the log*. Suppose that $delegate(t_1, t_2, ob)$ is the first delegation of ob by t_1. Applying this delegation can be visualized as iterating through the log into the past, modifying the records pertaining to ob, so that each record of an access to ob by t_1 will now show that the access was done by t_2.

```
K ← currLSN   (LSN of delegate record )
while LOG[K] is not the initiate record for t1
      if LOG[K] is an update to ob by t1
           then setTransID(K,t2)        alter it to look done by t2
      K ← prevLSN(K,t1)                 follow t1's BC
```

Figure 1: Operational semantics of $delegate(t_1, t_2, ob)$

The log is a list held in stable storage, whose elements are identified by monotonically increasing values of the *Log Sequence Number* (LSN). During normal execution, the only valid operation is appending a log record to the end of the log (with the corresponding increment of the current Log Sequence Number). During recovery, the log can be rolled back and replayed, by going to the LSN of the last checkpoint and extracting, sequentially, the records from there on.

Figure 1 gives the operational description of delegation in terms of the log, for a scenario where K indicates the LSN being operated on in the current iteration. Records have a PrevLSN field, that contains the LSN of the previous record for the same transaction. The chain formed by the previous LSN pointers of log records of a transaction is called *Backward Chain* (see 3.3). The `delegate` record is a new type that records a delegation, with pointers to the previous records of both the delegator and delegatee (see section 3.4).
In figure 1 we use the following operations on the log:

prevLSN(K, t_1) which returns the Log Sequence Number of the previous (most recent) log record written by t_1 (i.e., before, or to the left of K).

setTransID(K,t_2), which does LOG[K].TransID $\leftarrow t_2$, making the record appear as if it had been written by the transaction t_2.

The fields in a log record are: LSN (log-sequence number), Type (update, delegation, commit, etc.), Trans-ID (the ID of the transaction that created the record), and Data. For delegate records there also exist two LSN pointers to the delegator and delegatee (see 3.4).

Example 1. Consider the log fragment (see figure 2):

$\ldots update[t_1, a], update[t_2, x], update[t_2, a], update[t_1, b], update[t_1, a], update[t_2, y]$

After the application of delegate(t_1, t_2, a), the log looks like:

$\ldots update[t_2, a], update[t_2, x], update[t_2, a], update[t_1, b], update[t_2, a], update[t_2, y]$

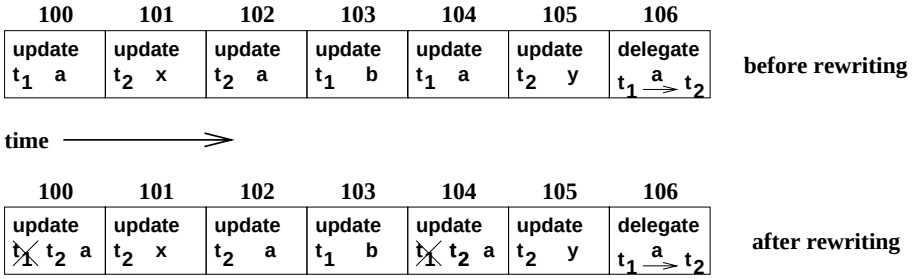

Figure 2: Delegation Log Example

3.2 Implementing Delegation Efficiently

The idea of rewriting history by modifying the log is simple, but its implementation is not. The naïve implementation of the algorithm in figure 1 would be to apply each delegation to the log as it is issued. That is, every time a delegation is issued, the system traverses the log backwards modifying the records pertaining to the object being delegated. This "eager" approach carries high performance costs, and is also hard to prove correct. The performance penalty is due to the random nature of the accesses (as opposed to the usual append-only to logs), and the fact that a single delegation will generate many accesses, in principle sweeping the whole log [17]. Ensuring recovery correctness is hard because we manipulate the log outside the usual append-only mode, complicating the model with extra data,[2] whose integrity in the face of crashes is not guaranteed by the standard recovery algorithm and must be ensured ad-hoc.

A better approach may be to use a "lazy" algorithm that defers the alteration of the log to recovery. This is based on the observation that during normal processing it is not necessary to have the delegations applied to the log. The algorithm can keep track of the effect of delegations in volatile data structures, and log the delegations to have the necessary information after a crash. It modifies the log – rewrites history – during recovery, which manipulates the log anyway, based on the information on the log about updates and delegations. For example, in UNDO/REDO (see section 3.3), the algorithm uses the logged information on updates and delegations to reconstruct the information about delegations during the analysis/redo (forward) pass. Then in the undo (backward) pass, it modifies the log as suggested in section 3.1 and figure 4, moving records from the delegator's backward chain to the delegatee's, and rewriting the record to make it appear as if created by the delegatee.

Although this is workable, it still suffers from drawbacks. Because it modifies the log in other than append mode, issues of correctness in the face of failure and performance must be addressed. It is possible to solve the correctness problem by ensuring that each BC switch

[2]Extra data: information accessed by transactions that is not part of the database schema; for example, the log, the system clock, wait-for graph. Gehani et al. [11] discuss the issues of correctness with extra data.

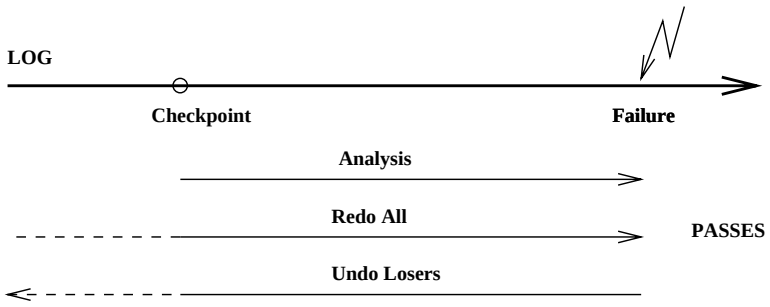

Figure 3: ARIES passes over the log

is done atomically.[3] The performance, however, is inherently hostage to the way the log is accessed. Recall that in general the log does not fit into volatile storage. The buffering can result in thrashing, as the algorithm needs to jump over possibly large sections of the log to follow backward chain pointers.[4]

To avoid these pitfalls, we propose RH, a "lazy" algorithm for rewriting history that does not modify the log. We give a brief overview here. As pointed out before, delegation can be supported easily during normal processing. During normal processing, we use a volatile table to keep track of which objects are updated by which transactions. When a delegation happens, we change the corresponding object binding, and log delegations to be able to reproduce the change after the crash. During recovery, on encountering delegations during the log sweeps, we reconstruct the bindings between operations on objects and transactions, but do not actually rewrite the log records. We "rewrite the history" of the system not by modifying the log, but by *interpreting* the log during recovery according to the delegations. That is, we obtain the desired semantics – rewrite of the history according to the delegations – without having to actually rewrite the log.

3.3 Conventional Recovery: ARIES

Before going into the details of the algorithm to implement delegation, we present the conventional version of ARIES, to establish context and terminology.

ARIES uses an UNDO/REDO protocol, which means that after a crash, some updates will be undone and some redone, according to whether the responsible transaction is a *winner* or *loser*. ARIES scans the log in one or two *forward* passes: analysis and redo; and then a *backward* pass: undo. See figure 3.

The *Analysis* pass starts at the last checkpoint, updates the information on active transactions and dirty pages up to the end of the log, and also determines the "loser" transactions, to be rolled back in the undo pass. The *Redo* pass *repeats history*, writing to the database those updates that had been posted to the log but not applied before the crash. This re-establishes the state of the database at failure time, including uncommitted updates. Because some ARIES

[3] It is easier to tolerate unusual log manipulations during recovery than during normal processing.

[4] The problems of this approach are discussed in detail in [17].

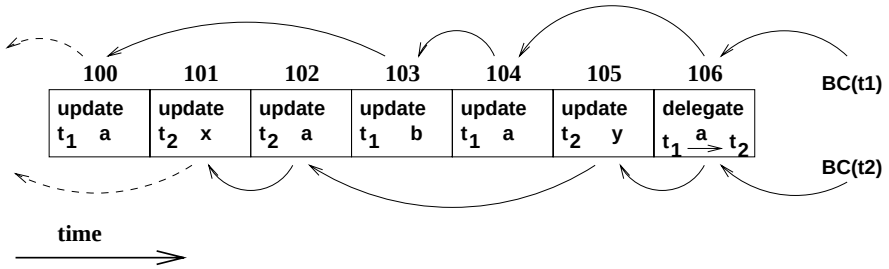

Figure 4: Backward Chains in the log

variants merge the analysis and redo passes in a single forward pass, ARIES/RH relies on a single forward pass to add delegation.

The *Undo* pass rolls back all the updates by loser transactions in reverse chronological order starting with the last record of the log.

To facilitate its UNDO/REDO recovery, ARIES keeps, for each transaction, a *Backward Chain* (BC) linking the transaction's records in the log. That is, all the log records pertaining to one transaction form a linked list, beginning with the most recent one, which is accessible through the Tr_List. By following the BC, ARIES's recovery avoids repeating undos, as it can insert *compensation log records* (CLRs) to indicate how to undo an action or whether to skip an already undone action.

In terms of Backward Chains, applying $delegate(t_1, t_2, ob)$ is tantamount to removing the subchain of records of operations on ob from $\mathrm{BC}(t_1)$, merging it with $\mathrm{BC}(t_2)$.

In the remainder of this section we explain ARIES/RH, our extension to ARIES for delegation, in detail. We present the data structures, and we indicate how the normal processing keeps the tables up to date. We then examine recovery processing, first the forward (analysis & redo) pass and then the backward (undo) pass. Finally, we examine how to apply RH to a different recovery protocol, NO-UNDO/REDO as exemplified by EOS.

3.4 Data Structures

For each transaction, we must know which operations on which objects it is responsible for, i.e., its Op_List. To that end, we augment *Transaction List* and each transaction's *Object List* found in conventional DBSs. We also add a `delegate` type log record.

Tr_List. We use the standard Transaction List Tr_List [2, 12, 14] that contains, for each Trans-ID, the LSN for the *most recent* record written on behalf of that transaction, and, during recovery, whether a transaction is a *winner* or a *loser* (see 3.3). Notice that for each transaction t, $Tr\_List(t)$ contains the head of the backward chain $\mathrm{BC}(t)$. Figure 4 shows the backward chains in the delegation example of section 3.1.

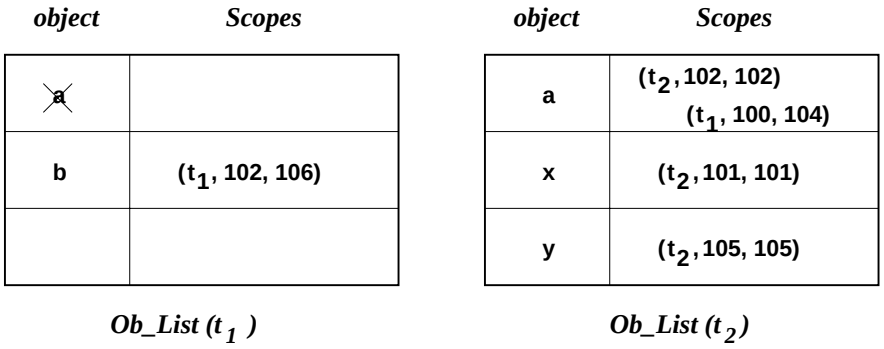

Figure 5: Object Lists after applying delegation of Example 1

Ob_List. Conventionally, associated with *each* transaction t there is a set $Ob\_List(t)$.[5] In figure 5, $Ob\_List(t_1)$ contains the objects t_1 is currently accessing. In terms of Op_List (see 2.1.1): $Ob\_List(t) = \{ob \mid \exists update[t_0, ob] \in Op\_List(t)\}$, that is, the objects for which there is an update that t is responsible for. Note that the update may have been originally done by t_0 and the responsibility transferred via delegation.

Because transactions are responsible for specific updates, and not a whole object, a certain object may appear in more than one Ob_List (but the associated updates will be different). For example, this can occur in the case of non-conflicting updates, e.g., increments of a counter. We identify the *updates* that a transaction is responsible for by introducing the notion of *scope*. (The scope supports the notion of Op_List.)

> ***Example 2.*** Consider a transaction t that updates ob, then delegates ob[6] to t_1, then again updates ob and finally delegates it to t_2:
>
> $\ldots update[t, ob], delegate(t, t_1, ob), update[t, ob], delegate(t, t_2, ob), abort(t_2), commit(t_1) \ldots$
>
> Regardless of t's fate, if t_1 commits and t_2 aborts, the first update (delegated to t_1) must persist, whereas the second (to t_2) must be undone.

For each object ob in $Ob\_List(t_1)$ there is a *set* of scopes, stored in the field $Scopes$, covering the updates to that object that t_1 is *currently* responsible for. A scope is of the form (t_0, l_1, l_2) (see figure 5). t_0 is the transaction that actually did the operations (the invoking transaction). The other two are LSN values: l_1 is the first, and l_2 the last LSN in the range of log records that comprise the scope. See figure 5. This indicates that t_1 is responsible for all updates to ob (by t_0) between the two LSNs.[7]

Delegate Log Records. We also introduce a new log record type: `delegate`. Its type-specific

[5] In some implementations Ob_List may have pointers to locks on the objects.

[6] Remember that $delegate(t, t_1, ob)$ really delegates the updates to ob that t is responsible for.

[7] This allows us to compute $ResponsibleTr$ (and Op_List) without having to store/update it with each update.

field name	*function*
LSN	position within the LOG
tor	transaction id of delegator
torBC	delegator's backward chain
tee	transaction id of delegatee
teeBC	delegatee's backward chain

Figure 6: Fields of the delegate log record

fields record the two transactions and object involved in the delegation (see 3.1). The fields are shown in figure 6. The other record types are not modified, and are as discussed in 3.1.

3.5 Normal Processing

Our algorithm augments the normal processing of ARIES; we focus on the changes entailed by delegation. We describe ARIES/RH in terms of how different events are processed. The current value of the log sequence number is CurrLSN.

- *begin*(t)
  1. INITIALIZE. Add t to Tr_List; create $Ob\_List(t)$.

- *update*[t, ob]
  1. ADJUST SCOPES. If this is the first update t does on ob since t started or last delegated ob we must open a new scope. Otherwise, there is an active scope of t on ob that we must extend.
     if $ob \notin Ob\_List(t)$ **then** $Ob\_List(t) \leftarrow Ob\_List(t) \cup \{ob\}$;
     if $(t, \_, \_)$[8] $\notin Ob\_List(t)[ob]$
     then $Ob\_List(t)[ob].Scopes \leftarrow$ (t, CurrLSN,CurrLSN) *(create new scope)*
     else $Ob\_List(t)[ob].Scopes \leftarrow$ (_,_,CurrLSN) *(extend existing scope)*

- *delegate*(t_1, t_2, ob)
  1. WELL-FORMED? Verify that $ob \in Ob\_List(t_1)$, which tests, for this case, the precondition in 2.1.2: $pre(delegate(t_1, t_2, update[ob])) \Rightarrow (ResponsibleTr(update[ob]) = t_1)$.
  2. PREPARE LOG RECORD(S).
     Record delegator, delegatee.
     $Rec.tor \leftarrow t_1$; $Rec.tee \leftarrow t_2$;
     Link this log record into t_1's and t_2's backward chains.
     $Rec.torBC \leftarrow$ BC(t_1).$PrevLSN$; $Rec.teeBC \leftarrow$ BC(t_2).$PrevLSN$.
  3. TRANSFER RESPONSIBILITY. Move operations on ob from $Op\_List(t_1)$ to $Op\_List(t_2)$.
     Add ob to delegatee's Ob_List and record that ob was delegated by t_1.

[8] We use '_' in a field to denote we do not change it or are not interested in its content.

$Ob\_List(t_2) \leftarrow Ob\_List(t_2) \cup \{ob\}$; $Ob\_List(t_2)[ob].deleg \leftarrow t_1$.

Pass delegator's Scopes for *ob* to the delegatee and remove *ob* from the delegator's *Ob_List*.

$Ob\_List(t_2)[ob].Scopes \leftarrow Ob\_List(t_2)[ob].Scopes \cup Ob\_List(t_1)[ob].Scopes$;
$Ob\_List(t_1) \leftarrow Ob\_List(t_1) - \{ob\}$.

Remark: We use a union because t_2 may already be responsible for some operations on *ob* before receiving the delegation. Therefore, the *Scopes* field may actually contain several scopes: contiguous ranges of LSNs on the log, each tagged with the transaction that initially was responsible for that scope, which is the invoking transaction for those updates. (Notice that the scopes may overlap on the log segment they cover but cannot share the same invoking transaction.)

4. WRITE DELEGATION LOG RECORD(S).

   Write log record and mark it as the current head of the backward chains of delegator and delegatee.

   LOG[CurrLSN] $\leftarrow$ *Rec* ; BC(t_1) $\leftarrow$ CurrLSN ; BC(t_2) $\leftarrow$ CurrLSN.

- *commit*(t)
  1. COMMIT OPERATIONS. Write to the log the operations for which t is responsible.
  2. WRITE COMMIT RECORD. Write a commit record to the log after the operations.
  3. FLUSH LOG. Write to stable storage all records in the log, from the previous flush up to the commit record inclusive.

- *abort*(t)
  1. ABORT OPERATIONS. Undo the updates for which t is responsible. Recall that any object that had been delegated *by* the aborting transaction will no longer be in the transaction's *Ob_List*, unless it updated it *after* the delegation.

     Obtain minLSN = min $\{begin \mid Ob\_List(t)[ob].Scopes = (\_, begin, \_)\}$ on objects in $Ob\_List(t)$. For each object in $Ob\_List(t)$, undo all updates contained in its Scopes, writing to the log the compensation log records, going backwards in the log until minLSN is reached.
  2. WRITE ABORT RECORD. Write log abort record to the log.
  3. FLUSH LOG. Flush log up to abort record.

The other transactional events are processed as usual [12, 14].

3.6 Recovery

After a crash, the transaction system must do some recovery processing to return to a consistent state. This entails restoring the state from a checkpoint (retrieved from stable storage), and using the log (also from stable storage) to reproduce the events after the checkpoint was taken. For simplicity of presentation, we ignore checkpoints from now on, but it is easy to see how data structures can be rebuilt using checkpoints instead of going back to the beginning.

Crash is the event that represents a failure; *RecoveryComplete* is the event that appears in the history to indicate that the recovery is complete.

Winners is the set of transactions that had committed before the crash. The recovery algorithm ensures that their updates will be reinstalled. *Losers* is all the other transactions that were active but had not committed, or had aborted, before the crash. Their updates must be undone.

In the rest of this section, we present the forward pass of ARIES/RH, we establish the state after that pass and, we describe the backward pass.

3.6.1 Forward Pass

ARIES starts with a forward pass that finds out which transactions were active, and which committed before the crash (analysis). Committed transactions are *Winners*; active but uncommitted, or aborted transactions are *Losers*. ARIES also uses the forward pass to *redo* logged updates. At the end of this pass the *Object Lists* are up to date, including the s of the updates.

Before the first pass of recovery starts, $Winners = Losers = \phi$. For brevity we omit some details already explained for the normal processing. We describe the treatment of the log records matter relevant to delegation in ARIES. Other records are processed as usual.

- *begin*(t)
  1. INITIALIZE. Add t to Tr_List; create $Ob\_List(t)$.
  2. LOSER BY DEFAULT. Consider t a loser by default.
     $Losers \leftarrow Losers \cup \{t\}$.
- *update*$[t, ob]$
  1. ADJUST SCOPES. This is done just as *update* (1) in normal processing.
  2. REDO. Redo *update*$[t, ob]$.
- *delegate*(t_1, t_2, ob)
  1. TRANSFER RESPONSIBILITY. This is done just as *delegate* (3) in normal processing.
- *commit*(t) t committed and thus is a winner.
  1. COMMIT. Declare t committed. Notice that t's updates were redone during this forward pass.
  2. WINNER. Declare t as a winner.
     $Winners \leftarrow Winners \cup \{t\}$; $Losers \leftarrow Losers - \{t\}$.

After the Forward Pass we have updated the following.

- *Ob_Lists* are restored to their state before the crash, for all transactions.
- *Winners* has all the transactions whose updates must survive (i.e., which had committed before the crash). *Losers* has those whose updates must be obliterated.

- *LsrObs* includes all objects in the *Ob_List*s of loser transactions. We compute it after the forward pass ends, as $LsrObs = \bigcup_{t \in Losers} Ob\_List(t)$.

3.6.2 Backward Pass

ARIES undoes exactly all the updates *invoked* by the loser transactions. It follows the backward chains (of records for each transaction) for each of the loser transactions, undoing *all* their updates. ARIES continually takes the maximum Log Sequence Number (LSN) for an outstanding undo, ensuring monotonically decreasing (by LSN) accesses to the log, with the attendant efficiencies.

This approach is not possible in the presence of delegation. What we need to achieve is the undo not of all the updates invoked by a loser transaction, but instead of *all the updates that were ultimately delegated to a loser transaction.* Thus, the analogy to ARIES would be to have backward chains linking those updates, but constructing and maintaining them is complicated and expensive. One could scan *all* log records backwards, identifying the *loser* updates (to be undone), which are the updates whose responsible transaction is a loser. This is undesirable as it entails unnecessarily inspecting many winner updates.

Fortunately, the necessary information to undo loser updates efficiently is in the object lists of the loser transactions: the update scopes used for delegation. In the rest of the section we discuss how undo and delegation are integrated in the backward pass of ARIES/RH. The only records that require special processing are update and delegation; all others are processed as in ARIES.

Notice that by undoing the *loser* updates instead of the updates invoked by loser transactions, we are in fact applying the delegations, as we undo according to the fate of the final delegatee of each update.[9] As in ARIES, for each undone update we write a *Compensation Log Record* (CLR), to avoid undoing an update repeatedly should crashes occur during recovery.

Recall that scopes keep track of updates whose fate is the same (i.e., that were delegated together). It is enough to inspect records within the *loser* scopes to find all loser updates. To do this efficiently, we introduce the notion of *cluster* of scopes. Scopes may overlap; a cluster of scopes is a maximal set of overlapping scopes. (We only care about "loser" clusters of scopes, so we omit "loser" from now on.) Within each cluster we must examine every log record, but between clusters we examine none. For instance, in figure 7 there are three clusters; the middle one contains four overlapping loser scopes. (In figure 7 loser scopes are depicted in dark shades). The last cluster has already been processed, and we are processing the middle cluster (K indicates the current log record). The current cluster begins at begCluster; the first (i.e., to be processed last) cluster in the log begins at begLsrScopes.

[9] In ARIES, all loser updates are those invoked by loser transactions, so ARIES/RH reduces to ARIES when there is no delegation.

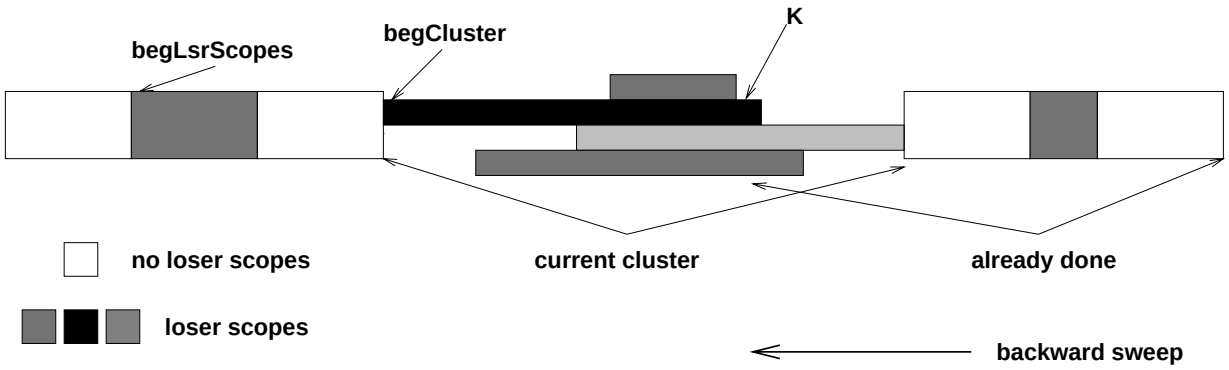

Figure 7: Loser scope clusters in a the log

We can now outline the algorithm for the backward pass of ARIES/RH (see also figure 8).

The idea is to examine each loser cluster, skipping all other records. Within a cluster we examine all records, undoing loser updates. We also adjust the current cluster by adding or deleting scopes, closing the cluster when we reach its first record, at begCluster. The algorithm ends when we reach the beginning of the first cluster, at begLsrScopes.

We begin by computing the set LsrScopes, which contains all the scopes for which loser transactions are responsible. We compute begLsrScopes which marks the beginning of the leftmost (oldest) loser scope in the log. We start sweeping the log backwards (i.e., right-to-left in the figure 7) at the end of the rightmost loser scope, and end at begLsrScopes (see **while** loop[10]). This sweep consists of two steps: we identify a cluster and undo all the loser updates in its component scopes (α); and we move to the next cluster once the current one is exhausted (β).

We process a cluster (α) in four steps. First, we check whether the current record is the right end of a scope, in which case we add the scope to the current cluster ($\alpha 1$). Then we check if the record is a loser update, and if so we undo it ($\alpha 2$). Specifically, a record is a loser update if it is within the ends of a loser scope whose invoking transaction is the same as the update's invoking transaction. Then we check if a scope ended in the record just processed, in which case we remove it from the current cluster, because the scope has already been treated ($\alpha 3$). Finally we move K (left) to the next record ($\alpha 4$).

Since when we add a scope to the current cluster Cluster ($\alpha 1$) we remove it from LsrScopes, (β) finds the next cluster by looking at the remaining scopes in LsrScopes and finding the rightmost end.

The repeat loop ends because we decrement K by at least one on each iteration, and although (α)'s limit begCluster may decrease, it may never go below begLsrScopes (because begLsrScopes is the minimum of scope begins), so eventually K reaches it. The while loop ends because we

[10]This and the following references in parentheses are to figure 8.

```
LsrScopes ← { scope | ∃ob, ∃t ∈ Losers : scope ∈ Ob_List(t)[ob].Scopes}   all loser scopes
if LsrScopes ≠ φ then
   begLsrScopes ← min { left | (_, left, _) ∈ LsrScopes }   start of earliest scope
   K ← max { right | (_, _, right) ∈ LsrScopes }            end of latest scope
   Cluster ← φ ;  begCluster ← K

   while begLsrScopes ≤ K    K sweeps log backwards to left end of earliest loser scope

   (α) repeat   identify and process cluster of overlapping slopes

          add to Cluster the loser scope that ends in K
       (1) if ∃ (_, left, K) ∈ LsrScopes then
              Cluster ← Cluster ∪ {(_, left, K)}          put in Cluster
              LsrScopes ← LsrScopes − {(_, left, K)}      and remove from LsrScopes
              begCluster ← min (left, begCluster)         updating where cluster starts

          undo if it is loser update
       (2) if LOG[K] = update[t, ob] and ∃ (t,_,_) ∈ Cluster then
              undo(update[t, ob])
              CLR.PrevLSN ← LOG[K].PrevLSN                and other undo information ...

          discard scope that begins at new record LOG[K]
       (3) if ∃ (_, left, _) ∈ Cluster and K = left then   already processed,
              Cluster ← Cluster − {(_, left,_)}           so discard

          processed record LOG[K], next (left) ...
       (4) K ← K −1

   (end α) until K < begCluster   finished this cluster (sweep backwards)

          find next cluster of scopes
   (β) K ← max { right | (_, _, right) ∈ LsrScopes}

RecoveryComplete
```

Figure 8: Backward pass of ARIES/RH

decrement K by at least one each time (α) (and more when we skip between clusters (β)).

Notice that we visit each log record at most once and in a monotonically decreasing way (K gets decremented by at least one in each iteration of the while loop). This is important as the log will be brought in from memory, and mimics ARIES's strategy of continually taking the largest LSN of the updates to be undone.

The set of scopes LsrScopes is constructed once and depleted in the reverse order of scopes, so an efficient data structure is a priority queue (on a heap) sorted by right end of scopes, with the largest value first. The set of scopes Cluster is searched by invoking transaction, gains scopes to the left and loses scopes to the right. A binary tree keyed on transaction ids is a reasonable implementation.

3.7 Implementing delegation in EOS

Rewriting History can be applied to other recovery algorithms, for instance, EOS [4], which uses a NO-UNDO/REDO protocol. Next, we give an overview of EOS and then discuss briefly how to apply RH to implement delegation.

To avoid having to undo changes in the database, EOS avoids applying those changes until the transaction that made them is ready to commit. This is achieved by keeping a *global* log, in which only transaction commits are recorded, and per-transaction *private* logs. If a transaction commits, its private log is flushed to stable storage; if it aborts, the private log is discarded. The recovery of EOS is simpler than that of ARIES, because no undo is necessary; only committed changes are logged, so they are reapplied during a single forward sweep of the global log (that in turn brings in the private logs of committed transactions).

We can support delegation within EOS by applying an algorithm very similar to ARIES/RH for the normal processing and the forward pass of the recovery. The differences are due to the private logs kept by EOS. "Rewriting History" must now be implemented across different private logs. When two transactions do concurrent (but non-conflicting) updates on an object, and one delegates its operations to the other, the net effect should be as if the delegatee had executed all the operations in the original order. In EOS the transactions keep separate logs, so reconstructing that order is not straightforward. This situation does not arise when we restrict the operations to reads and writes, because in this case, all updates are translated to write operations and so even compatible update operations execute in isolation. In the read/write case, then, it is enough for the delegator to supply the delegatee with an image of the current state of the object at the time of the delegation. This image is stored as part of the delegate record.

Supporting delegation in EOS entails logging the delegation both at the delegator and the delegatee. The delegator *filters out* updates it has delegated when it comes time to commit, to avoid committing updates it no longer is responsible for. If it aborts, its private log is discarded,

but the delegated updates are preserved in the log of the delegatee. When the delegatee commits, it has the updates it has received through delegation preserved in the delegatee record, so it does not need to rely on the delegator (which may not be around any more). If the delegatee aborts, the updates it received in delegation will not be applied (delegatee's log is discarded, delegator's is either discarded or filtered).

Recovery is simple, because we only need to redo the winner updates. Loser transactions have their private logs discarded, and winner transactions have their private logs redone. This takes care of the undelegated updates. For delegated updates, first we see that they get redone if they were winner updates (i.e., their Responsible Transaction at crash time was a winner, that is, had committed). Regardless of the fate of the invoking transaction (and any intervening delegating transactions) the winner update was delegated to the winner transaction last. Thus when redoing the winner transaction's log, we restore the state of the object from the delegation record in the winner's private log. Notice that there is no possibility for this ever being undone because no updates are ever undone.

If an update was in a loser transaction, it will not be redone because the loser transaction's private log is discarded. It also does not get redone by other transactions for the following reason. If the invoking transaction is a winner, it must have delegated the update (for it to be a loser now). When a transaction delegates an update it filters it out when saving its private log, so that update will not be there when the private log is redone. This applies to any transaction that was at some point responsible for the update and delegated it. Thus the loser updates do not get redone.

4 Discussion

In this section we discuss two issues with regard to the implementation of the recovery algorithm. First we analyze aspects of the algorithm presented in section 3 to show that it implements delegation correctly. In the second part, we examine the implementation and argue that it is efficient.

4.1 Correctness

To ensure correctness in a recovery protocol, we must guarantee that, after recovery, all operations by *loser* transactions have been rolled back completely (their effects obliterated) and all by *winner* transactions have been committed (their effects guaranteed to be permanent). Conventional ARIES complies with this by using the UNDO/REDO protocol (EOS does NO-UNDO/REDO). We show in this section that ARIES/RH, that is, ARIES with our modifications complies with this requirement when it is rephrased to include delegation. That is, operations delegated to loser transactions will be aborted, and operations delegated to winner transactions

will be committed. Naturally, boring operations (i.e., never delegated), are treated as in the conventional case, because in the absence of delegation ARIES/RH reduces to ARIES.

After the event *Crash*, we initiate recovery, which ends with the event *RecoveryComplete*. Between *Crash* and *RecoveryComplete* all events are generated by the recovery system. For simplicity, we ignore checkpoints and assume that the system restarts from the beginning.

A brief recapitulation of the algorithm is in order. The forward pass reads but does not write anything to the log. It redoes the updates present in the log, and constructs the sets *Winners* of transactions whose updates will survive after the recovery. It also records the *Losers*, i.e., active transactions that did not commit before the crash. ARIES/RH also computes *LsrObs* after the forward pass. The backward pass reads the log, interpreting it according to the delegations, and undoes updates on loser objects.

In the remainder of this section, we characterize loser and winner transactions and their associated updates, we explain the idea of delegation chain, and formalize the correctness properties. Then we show the correctness of the algorithm, to wit, that all loser updates get undone and all winner updates get redone.

Winners, Losers, LsrObs.

- $t \in Winners \iff (Commit(t) \rightarrow Crash)$
  t is a winner if it committed before the crash.

- $t \in Losers \iff (Begin(t) \rightarrow Crash \wedge \nexists Commit(t) \in H)$
  t is a loser if it was active but did not commit before the crash.

  Losers: an active transaction is by default a loser. If there is a commit record before the crash, its transaction is moved to *Winners*. Note that these sets are disjoint, and that *Losers* includes transactions that had aborted before the crash.

- $LsrObs = \bigcup_{t \in Losers} Ob\_List(t)$ i.e., $ob \in LsrObs \Rightarrow \exists t \in Losers : ob \in Ob\_List(t)$
  LsrObs is the set of all objects for which there is a loser transaction that is responsible for an update to that object. This means that a loser object has at least one update that will be undone.

Delegation Chain.

We assert that if t_n is responsible for an update, either t_n invoked the update itself ($n = 0$) or t_n received it from t_0 through a sequence of delegations. Formally:

$update[t_0, ob] \in Op\_List(t_n) \Longrightarrow (\exists (n \geq 0), t_0, t_1, , ..., t_{n-1}, t_n$ **such that**

that is, if t_n is responsible for the update, then there is a sequence of transactions, starting with t_0, the invoking transaction, and ending with the responsible transaction t_n, such that

$[update[t_0, ob] \rightarrow delegate(t_0, t_1, ob) \rightarrow ... \rightarrow delegate(t_{n-1}, t_n, ob)] \ \wedge$

$[\nexists y$ **such that** $delegate(t_{n-1}, t_n, ob) \rightarrow delegate(t_n, t_y, ob) \rightarrow Crash] \ \wedge$

$[\nexists i \ (0 < i < n), \nexists t_x$ **such that** $delegate(t_{i-1}, t_i, ob) \rightarrow delegate(t_i, t_x, ob) \rightarrow delegate(t_i, t_{i+1}, ob)]$

t_0 delegated the update to t_1, and so on, until finally t_n received it in the last delegation; and for each t_i, t_{i+1} is the first transaction to which t_i delegates ob, i.e., there is no other intervening delegate (to, say, transaction t_x) of that update.

We prove that if t_n is responsible for an update, there is a sequence of delegations that links the original update log record to t_n by induction on n. The base case, for $n = 1$, is immediate from the algorithm that applies delegation during normal processing and the forward pass of recovery.[11] Specifically, when a transaction invokes an update, it creates or enlarges its current scope to include it (see delegate in 3.5). Each time a delegation is invoked, the scope of the delegated object, which includes the delegator's updates, is passed to the delegatee (see delegate in 3.5). The scope defines uniquely the updates being delegated (see *Ob_List* in 3.4 and the remark in 3.5).

For the inductive case, note that a delegated scope is never modified by the *delegatee*. For a given object, a delegatee either keeps the scope(s) it received in the delegation, or it augments them with its own scope for its updates on the object. Thus in a $delegate(t_k, t_{k+1}, ob)$, the scopes that t_{k+1} keeps for the object ob are $Ob\_List(t_k)[ob].Scopes \cup$ the scope t_{k+1} has on ob. Thus an update contained in t_k's scope for ob will be in t_{k+1}'s.

Correctness Properties

Here we state the properties of undo and redo that describe correct recovery.

undo $(\forall t \in Losers\ \forall update[t_0, ob] \in Op\_List(t))(Undo(update[t_0, ob]) \rightarrow RecoveryComplete)$
All updates ultimately delegated to a loser transaction are undone before the recovery ends.

redo $(\forall t \in Winners,\ \forall update[t_0, ob] \in Op\_List(t))(Redo(update[t_0, ob]) \rightarrow RecoveryComplete)$
All updates ultimately delegated to a winner transaction are redone before the recovery is finished.

In other words, updates whose responsible transaction did not commit before the crash are undone (obliterated), and updates whose responsible transaction committed before the crash are redone (made permanent). In the following paragraphs we discuss how the implementation satisfies the requirements.

Correctness of ARIES/RH

For the correctness of the normal processing, notice that the scope information on the *Ob_List* associated with each transaction is sufficient to decide whether to commit or abort a specific update in the absence of crashes. Then notice that the updates covered by scopes in the *Ob_List* of a loser transaction are aborted, and those in the *Ob_List* of a winner are committed. This is easy to see by inspection of the algorithm, specifically, the update, commit, and abort cases in 3.5.

For the recovery, we first show that **undo** holds, that is, that all *loser* updates are undone. An update is a loser if $ResponsibleTr(update[t_0, ob]) = t$ and $t \in Losers$. (We already know

[11] For $n = 0$, reduces to no delegation and holds trivially: it is the boring update case.

that there must be a delegation chain from t_0 to t.) This means that there is a scope $(t_0, l, r) \in Ob\_List(t)[ob].Scopes$, and a log sequence number q, $l \leq \text{q} \leq r$ such that LOG[q] = $update[t_0, ob]$, by the definitions of scope, loser update, and responsible transaction. Then by the construction of LsrScopes in the algorithm (figure 8), $(t_0, l, r) \in$ LsrScopes. Let us show that the record for the update will be eventually checked and undone. Initially $(t_0, l, r) \in LsrObs$, $l \leq$ initial value of K, and begLsrScopes $\leq r$, by construction. Because K starts at the maximum value and goes down by one, or jumps to the next right end of a scope, it will eventually reach r and put (t_0, l, r) in Cluster. When K reaches q, it is within the scope and the update is undone.

We prove the **redo** property by contradiction. Recall that all updates are redone in the forward pass. We show that no *winner* update gets undone. We proceed by contradiction; we suppose that a winner update is erroneously undone in the backward pass. If the update was undone, it means that it appeared in some *loser* scope (see figure 8). But, because the delegation chain applies here too (it does not depend on the fate of the final transaction), this means that there is a chain of delegations that starts with the invoking transaction of the update and ends with a loser transaction. But that means that the responsible transaction of the update was a loser, contradicting that it was a winner update. *Q.E.D.*

4.2 Efficiency

We claim that ARIES/RH is efficient in the following senses:

- **No delegation, no overhead**. In the absence of delegation ARIES/RH reduces to the original algorithm, so no penalty is incurred due to the extra functionality when it is not used.
- **Normal processing: low overhead.** Posting one delegation during normal processing has the cost of adding a log entry and updating the object bindings. The cost of delegations is linear in the number of operations delegated. For instance, the updating of Object Lists for a delegation is linear in the length of the *Ob_List*s.
- **Recovery: low overhead.** The costs of the recovery passes are similar to those of conventional ARIES; ARIES/RH does not add any extra passes. For all operations, supporting delegation only entails costs at most linear in the number of delegated operations (see previous item). Also, recovery costs are dominated by disk log accesses, which ARIES/RH does as efficiently as ARIES. For instance, on the backward pass, log records are visited at most once and in strict decreasing order, as in ARIES, allowing for the usual optimizations.

The first two points follow from the fact that ARIES/RH only adds some fields to data structures that are already updated by the conventional algorithm. When there is no delegation, these fields are just left undefined. Delegating adds the constant time of logging the delegation operation and updating the *Ob_List*s of the delegator and delegatee transactions by moving as many scopes as objects are delegated (hence the linearity). This entails lookup/updates to the transactions' *Ob_List*, which resides in main memory and can be organized for efficient lookup/update. At transaction termination the *Ob_List* can be simply discarded.

As for recovery, ARIES/RH's forward pass incurs the same overhead as ARIES does to reconstruct transactional data structures and redo updates. Again, the only additional information that is collected is piggy-backed in those data structures. No special sweep of the log is required: ARIES/RH obtains its information during the same accesses as the conventional algorithm. Specifically, the forward pass of recovery is only different from that of ARIES in its processing of update (there is an extra check for $ob \in Ob\_List(t)$) and delegate (same check and the move from one *Ob_List* to the other). Thus, ARIES/RH adds neither extra log sweeps, nor costs proportional to the length of the log, as it uses the same sweeps of the log as ARIES to reconstruct the delegation information.

We expect the *Ob_List* to be much smaller than the log being analyzed, and to wholly reside in main memory. Thus the cost of accessing *Ob_List* is small compared to bringing the log from stable storage, the dominant cost during recovery.

The backward pass of recovery reads the log in much the same way as ARIES, by continually taking the maximum Log Sequence Number that must be undone (in ARIES) or the scope clusters within which updates must be undone (in ARIES/RH). In ARIES/RH we examine log records in clusters formed by loser scopes, but, as in ARIES, we do it in a monotonically decreasing way. To compare with ARIES, we need only examine the costs for processing update records (the rest are just as in ARIES). For each update, we do a lookup in Cluster for a check of delegation scope (to decide whether to undo it), and possibly write a Compensation Log Record. Otherwise, we just add or remove scopes from the Cluster and the LsrScopes sets.

In summary, the ARIES/RH algorithm adds only minimal overhead to support delegation.

5 Related Work

We have benefited from insights gained in an effort with goals closer to ours: the work at GTE Labs on Transaction Specification and Management Environment (TSME, [9]). The architecture of TSME consists of a Transaction Specification Facility that understands TSME's transaction specification language, and drives the Transaction Management Mechanism which configures the run-time system to support a specific Extended Transaction Model. The Transaction Management Mechanism is programmable, but uses templates to describe existing extended transaction models, and also to drive the incorporation of only the components necessary for a given Extended Transaction Model. It is a toolkit approach, in which certain expressions in the specification language are mapped to certain configurations of *pre-built* components, so it approaches the problem at a coarser grain. This may allow for initial gains in performance, but we believe that the use of language primitives is a richer and more flexible approach.

The recent work of Barga and Pu [1], also inspired in part by ACTA, explores another modular approach, based on the ideas of metaobject protocols [13], and incorporates some elements of the TSME approach and some of our language-based approach.

Also related is the work on the ConTract model [20]. In ConTract, a set of steps define individual transactions; a script is provided to control the execution of these transactions. But ConTract scripts introduce their own control flow syntax, while ASSET introduces a small set of transaction management primitives that can be embedded in a host language.

Other related work also includes Structured Transaction Definition Language [3], a persistent programming language geared to portability and the integration of legacy applications. Its emphasis, however, is in Application Programming Interfaces conforming to existing conventional transactional technology.

Finally, the idea of rewriting history is a natural extension of the repeating history paradigm of ARIES [14] and is a generalization of ARIES/NT [19], an extension to ARIES for nested transactions [15].

6 Conclusions

Recent work has produced many Extended Transaction Models (ETMs), but each has its own tailor-made implementation. With delegation (and the other two ASSET primitives, permit and form-dependency [5]) we believe we can offer the flexibility to synthesize a wide range of ETMs at a performance comparable to that of tailor-made implementations. Delegation, by allowing changes in the visibility and recovery properties of transactions, is a very useful primitive for synthesizing Extended Transaction Models. Our work builds on the formal foundation provided by ACTA [6, 7, 8], and the primitives introduced in ASSET [5].

The main contribution of this paper is the concept of *rewriting history* (RH), designed to achieve the semantics of delegation in an efficient and robust manner. We believe that this work forms a crucial step towards the flexible synthesis of ETMs:

- By casting delegation in terms of rewriting history, we were able to express the issues of delegation in terms amenable to the specification of a recovery algorithm.
- We showed how to achieve RH in the context of a practical system (ARIES), and sketched how to apply it to another (EOS), suggesting the practical implementability of delegation. As indicated in section 4, the cost of delegation in ARIES/RH is very low, and its support incurs no cost at all when delegation is not being used.
- We have also demonstrated the correctness of our implementation, showing that it satisfies the desired transaction properties in the presence of delegation.

We are currently implementing RH within EOS. We will continue investigating the broader issues of providing robust, efficient, and flexible transaction processing. In particular, we are interested in making recovery a first-class concept within transaction management and in pro-

viding a variety of recovery primitives to a transaction programmer so that different recovery requirements and recovery semantics can be achieved flexibly.

Acknowledgments. We thank Lory Molesky, Jagan Peri, and Jayavel Shanmugasundaram for valuable discussions. We also thank Mohan Kamath and Amer Diwan, whose comments helped make this paper more readable.

References

[1] Roger S. Barga and Calton Pu. A Practical and Modular Implementation of Extended Transaction Models. In *Proceedings of the 21st International Conference on Very Large Data Bases*, September 1995.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison Wesley, Reading, Mass. 1987.

[3] Philip A. Bernstein, Per O. Gyllstrom, and Tom Wimberg. STDL – A Portable Language for Transaction Processing. In *Proceedings of the 19th International Conference on Very Large Databases*, pages 218–229, Dublin, 1993.

[4] A. Biliris, E. Panagos. EOS User's Guide. AT&T Bell Labs Report, May 1993.

[5] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* , Minneapolis, Minn., June 1994.

[6] P. K. Chrysantis and Krithi Ramamritham. Synthesis of Extended Transaction Models using ACTA. *ACM Trans. on Database Systems*, September 1994.

[7] P. K. Chrysanthis. *ACTA, A Framework for Modeling and Reasoning about Extended Transactions.* Computer Science TR 91-90. PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, Mass., September 1991.

[8] P. K. Chrysantis, and Krithi Ramamritham. Delegation in ACTA as a Means to Control Sharing in Extended Transactions. IEEE Data Engineering, 16(2): 16-19, June 1993.

[9] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and Management of Extended Transactions in a Programmable Transaction Environment. In *Proceedings of 10th International Conference on Data Engineering*, Houston, Tex., February 1994.

[10] A. K. Elmagarmid, editor, Database Transaction Models for Advanced Applications. Morgan Kaufman, 1991.

[11] Narain Gehani, Krithi Ramamritham, Oded Shmueli. Accessing Extra Database Information: Concurrency Control and Correctness. Computer Science TR 93-081, University of Massachusetts, Amherst, 1993.

[12] Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufman, San José, Calif. 1993.

[13] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow. The Art of the Metaobject Protocol. MIT Press, Cambridge, Mass., 1991.

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwartz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In *ACM TODS*, 17(1):94–162, 1992.

[15] J. Eliot B. Moss. *Nested Transactions: An approach to reliable distributed computing.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Mass., April 1981.

[16] C. Pu, G. Kaiser, G., and N. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 26–37, Los Angeles, CA, Sept. 1988.

[17] Cris Pedregal Martin and Krithi Ramamritham. ARIES/RH: Robust Support for Delegation by Rewriting History. TR 95-51 Computer Science Dept., University of Massachusetts, Amherst, June 1995.

[18] Cris Pedregal Martin and Krithi Ramamritham. Delegation: Efficiently Rewriting History. TR 95-90 Computer Science Dept., University of Massachusetts, Amherst, October 1995.

[19] Rothermel, K., and C. Mohan. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In *Proceedings of the 15th International Conference on Very Large Databases*, pages 337–346, Amsterdam, 1989.

[20] H. Wächter and A. Reuter. The ConTract Model. In [10].

[21] Gerhard Weikum, Christof Hasse, Peter Broessler, Peter Muth. Multi-Level Recovery. In *ACM International Symposium on Principles of Database Systems*, pages 109–123, Nashville, 1990.