ARIES/RH: Robust Support for Delegation by Rewriting History [†]

Univ. of Massachussets Computer Science Technical Report 95-51
June 8, 1995

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

Abstract

The notion of transaction delegation, as introduced in ACTA, has been shown to be useful in synthesizing Extended Transaction Models. Delegation allows a transaction to transfer responsibility over an object, and its operations, to another transaction. Delegation can be used to broaden the visibility of the delegatee, and to tailor the recovery properties of a 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 updates; for instance, a transaction may delegate some operations that will remain uncommitted but valid after the delegator transaction aborted.

In this paper, we show how to implement delegation in the context of ARIES, a robust recovery algorithm for industrial-strength transaction management systems. The resulting version of ARIES is called ARIES/RH since delegation is tantamount to Rewriting History. The modifications to ARIES are non trivial but small enough to *efficiently* support delegation. Although a naïve implementation of delegation would entail frequent and costly log accesses, our careful design has minimal cost during normal processing and involves only constant-time operations, and some stable storage access overhead, during recovery.

With our implementation of delegation, we suggest it is feasible to build efficient, robust, industrial-quality general-purpose machinery for Extended Transaction Models.

Keywords: Extended Transaction Models, Transaction Management, Recovery.

† Supported in part by Sun Microsystems and by the National Science Foundation under grant IRI - 9314376.

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 [8], each custom built for the application it addresses; alas, no one extension is of universal applicability.

Currently, each Extended Transaction Model is inflexible: if one's application deviates from the semantics foreseen by the designer of the Extended Transaction Models available, one is forced to adapt the application to the transaction system, or to build from scratch a system that supplies the needed functionality. To address this problem, we investigate how to create general-purpose, industrial-quality machinery to support the specification and implementation of diverse Extended Transaction Models. Our strategy is to work from first principles, first identifying the basic elements that give rise to different models, then proposing mechanisms for implementing these elements, and showing how to specify and implement various Extended Transaction Models.

A first step was ACTA [3], that identified, in a formal framework, the essential components of Extended Transaction Models. In more operational terms, ASSET [6] provided a set of new language primitives that enable the implementation 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 updates; 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, Open Nested (see [3, 6]).

Gehani *et al.* [6] describe how to implement some 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, that of delegate is not. For instance, close attention must be paid to logging and recovery issues in the presence of delegation. 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 [10]. Our basic additions to ARIES allow the "rewriting of history". We hence call it ARIES/RH.

In ARIES/RH we implement delegate as described in ASSET [6], i.e., a transaction delegates *all* of an *object*'s operations. (Implementing per-operation delegation is very similar.)

By providing delegation, we add substantial semantic power to a conventional Transaction Management System (TMS), allowing it to model various Extended Transaction Models. However, we achieve this expressiveness with only small modifications to the original ARIES, by carefully "piggy-backing" the delegation-related processing onto ARIES's routine processing. This is especially important during recovery, where our algorithm avoids adding costly extra sweeps to the log.

In this paper we argue that:

- Delegation is a powerful, essential primitive for realizing Extended Transaction Models. We explain its semantics and 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 present our algorithm in terms of extending a well-known, robust, efficient Transaction Management System, ARIES.
- The modifications to ARIES are non trivial but small enough to *efficiently* support delegation. The cost of delegation is minimal during normal processing. During recovery, we incorporate our rewriting of the log to the usual work ARIES does on the log, adding only constant-time operations, and some stable storage access overhead.

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 explain delegation's semantics in terms of rewriting history.

In Section 4 we develop our implementation of delegation in the context of a robust, industrial-grade transaction management system. We discuss our modification of existing data structures and some additions, and then describe how we operate on the log to effect delegations. In Section 5 we discuss why our algorithm correctly implements delegation and why it does it efficiently. In Section 6 we review related work, present our conclusions and discuss future work.

2 Delegation: Concepts, Examples, and Properties

In this section we examine the properties of delegation and present some examples of extended transaction models that can be realized using delegation.

First, some notation: $t, t_a, ..., t_1, t_2, ...$ denote transactions; $ob, ob_{del}, A, B, ...$ denote objects. Let *ob* be an object being accessed by a transaction t_1. When t_1 executes $delegate(t_1, t_2, ob)$, we say that t_1 transfers its responsibility over *ob* to transaction t_2. We say that t_1 is *responsible for ob* when t_1 is in charge of changes to *ob*. More precisely, *Responsible-trans(ob)* $= t_1$ holds from the time when t_1 either first accesses *ob*, creates *ob*, or is delegated *ob* until the time when either t_1 terminates or delegates *ob*. For instance, in some implementations responsibility may imply that t_1 holds an exclusive lock for *ob*. See [3] for a formal definition of delegation.

Basically, in the presence of delegation the fate of updates to an object is not necessarily linked to the transaction who made the updates, but instead it is linked to the fate of the transaction to which the object was last delegated. For instance, if t_1 updates *ob*, then delegates *ob* to t_2, and t_1 subsequently aborts, the changes t_1 made to *ob* will still survive if t_2 commits while it is still responsible for *ob*.

Via nested transactions, let us illustrate a simple use of delegation. Inheritance in nested transactions is an instance of delegation. Delegation from a child t_c to its parent t_p occurs when t_c commits. This is achieved through the delegation of all the operations of t_c to t_p when t_c commits. That is, all the operations that a child transaction is responsible for are delegated to its parent when it commits.

A transaction can delegate at any point during its execution, not just when it aborts or commits. For instance, in Split Transactions [12], 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 them.

Other transaction models using delegation include Reporting Transactions and Co-Transactions described in [4, 5]. The former periodically reports to other transactions by delegating its current results. In the latter, control is passed from one transaction to the another transaction at the time of the delegation.

We conclude this section with some observations. $Delegate(t_1, t_2, ob)$ is well formed when t_1 and t_2 are *initiated* and t_1 is responsible for *ob*. $Delegate(t_1, t_1, ob)$ is a null operation.

Transactions may be aborted by the system to enforce some correctness or concurrency control criterion or due to self-aborts, requested by the transaction. It is easy to see that delegate operates as per its definition: as pointed out before, if t_1 executed $delegate(t_1, t_2, ob)$ and then aborted, the changes to *ob* are *not* undone, and *ob*'s fate is that of t_2.

These properties extend to sets of objects by considering delegation of a set of objects as the concurrent execution of the corresponding single-objects delegations, or an *atomic* sequence of single-object delegations, in unspecified order. It is also straight-forward to extend these properties to per-operation delegation.

3 An Operational Semantics of Delegation

We now discuss the semantics of delegation through an operational description in the context of a Database Management System. First we present an abstract description and then we show a more detailed, but naïve, description in the context of ARIES.

3.1 Delegation as Manipulation of the Log

In a DBMS the log *is* the system's history, as it contains the records of all updates, transactional operations, etc. The idea of delegation is to *rewrite history*, selectively *altering the log*. $Delegate(t_1, t_2, ob)$ can be visualized as iterating through the log into the past, modifying all 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.

The log is a list held in stable storage, whose elements are identified by monotonically increasing values of the *Log Sequence Number* (LSN); via use of LSN we can view the log as an array. 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. We describe delegation in Figure 1 using the following operations:

- *prevLSN(startLSN, 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 LSN).
- *setTransID(LSN,t_2)*, which does $log[LSN].TransID \leftarrow t_2$, making the record appear as if it had been written by the transaction t_2.

Example. Consider the log fragment (see also Figure 1):
$\ldots update(t_1, A), update(t_2, X), update(t_1, B), update(t_1, A), update(t_2, Y)$
After the application of delegate(t_1, t_2, A), we have:
$\ldots update(t_2, A), update(t_2, X), update(t_1, B), update(t_2, A), update(t_2, Y), delegate(t_1, t_2, A)$

```
        **delegate**($t_1, t_2, ob$) **is:**

while currLSN is not the initiate record for $t_1$
        if currLSN is an update to ob
                then setTransID(currLSN,$t_2$);
        currLSN $\leftarrow$ prevLSN(currLSN,$t_1$)
```

(currLSN is initially the Log Sequence Number of delegate record)

Figure 1: semantics of delegation.

3.2 Delegation in terms of ARIES

Let us now consider delegation in terms of a specific recovery approach, namely, ARIES [10]. To facilitate recovery, ARIES keeps, for each transaction, a *Backward Chain* (BWC) 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. By following the BWC, ARIES's recovery avoids repeating undos, as it can insert compensation log records to indicate how to undo an action or whether to skip an already undone action. Using Backward Chains, and in view of the semantics just presented, we could realize $delegate(t_1, t_2, ob)$, , as follows:

- locate all records in t_1's Backward Chain that correspond to operations on ob
- move each of those records into the appropriate place in t_2's Backward Chain, renaming them to be t_2's records.

Since each transaction's log records are linked in a BWC, another way to put it is:

- remove the subchain of records of operations on ob from t_1's BWC$_1$, merging it with t_2's BWC$_2$.

Example. Consider the log fragment (see also Figure 2):
$\ldots update(t_1, A), update(t_2, X), update(t_1, B), update(t_1, A), update(t_2, Y),$
$delegate(t_1, t_2, A), update(t_1, B), update(t_2, A)$
Initially the BWCs of the transactions are:
$\ldots update(t_1, A) \leftarrow update(t_1, B) \leftarrow update(t_1, A) \leftarrow delegate(t_1, t_2, A) \leftarrow update(t_1, B) \longleftarrow BWC_1$
$\ldots update(t_2, X) \leftarrow update(t_2, Y) \leftarrow update(t_2, A) \longleftarrow BWC_2$
After applying the delegation, the BWCs look like this:
$\ldots update(t_1, B) \leftarrow delegate_1(t_1, t_2, A) \leftarrow update(t_1, B) \longleftarrow BWC_1$

$$\ldots update(t_2, A) \leftharpoondown update(t_2, X) \leftharpoondown update(t_2, A) \leftharpoondown update(t_2, Y) \leftharpoondown update(t_2, A) \longleftarrow BWC_2$$

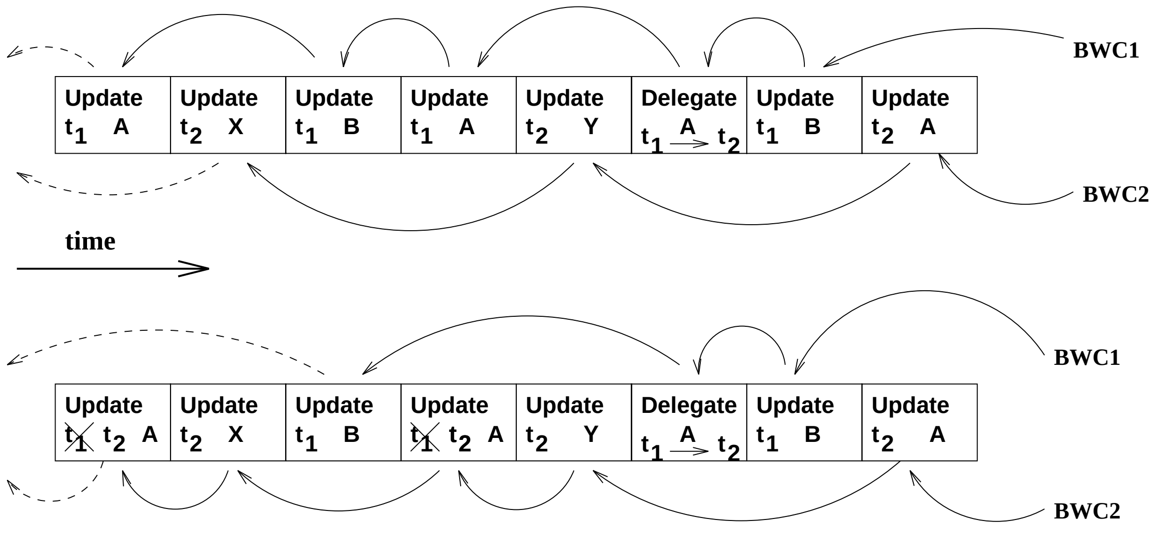

Figure 2: log before and after applying delegation

To implement delegation in ARIES according to the naïve algorithm, although correct, is inefficient, because the log is scanned every time a delegation is submitted. In the next section we present an algorithm that has a very small cost during normal processing, and is efficient during recovery.

4 Implementation: Rewriting History Efficiently

Our algorithm is lazy: it takes note of the individual delegations as they happen (when they are submitted by transactions), but only applies them later, when the log is scanned for recovery.

It is enough to apply the delegations to the log during recovery. During normal processing, we can rely on volatile data structures to keep track of delegations. In this section we present our algorithm ARIES/RH and its data structures.

4.1 Data Structures

In order to defer the application of delegation to recovery time, we must keep track of which objects have been delegated to which transactions, and how each transaction's records are linked together in their Backward Chain (see 3.2).

The abstract operations *prevLSN* and *setTransID* (defined in 3.1) suggest what information we need to manipulate. For each delegated object we must keep track of its delegator transaction, when it was delegated (see scope, below), and to whom (the delegatee). For each transaction, we must know which objects it is responsible for (see 2). This information must be updated according to the delegations. In the following, we present the data structures we use.

ARIES maintains a *Transaction List* TL that contains, for each Trans-ID, the Log Sequence Number (i.e., the address of a log record) for the *last* record for that transaction. Since each record has a *PrevLSN* pointer, TL(t_1) is the head of the BWC$_1$, TL(t_2) of BWC$_2$, and so on.

Associated with *each* transaction there is an *Object List* OL that contains a list of the objects that the transaction is currently responsible for (in some implementations OL may have pointers to locks or other resource management devices). We augment the Object List with a *deleg* field for each object that indicates whether this object was acquired through a delegation and from whom. Specifically, $OL_2(ob).deleg = t_1$ if ob was delegated **by** t_1 **to** t_2, or $OL_2(ob).deleg = t_2$ if it was created or accessed originally by t_2. See Figure 3.

We create and maintain a *Delegated Objects Table* (DOT). For an object ob, DOT(ob)= t_1 indicates that ob has been last delegated **to** t_1. See Figure 3. Since the effects of updates on ob are committed (respectively: discarded) when *Responsible-trans(ob)* commits (aborts), DOT supplements the conventional mechanisms that track the fate of updates. DOT also keeps track of the scope of an object's delegation. When an object is delegated, all previous *uncommitted* updates to the object are relabeled as if they had been done by the delegatee. However, if a transaction which is responsible for ob commits, its updates to ob cannot be delegated. Hence we introduce the notion of *scope* of a delegation: it is the extent on the log to which the delegation applies, and it is indicated by the Log Sequence Number of the last commit record that caused the object's updates to be committed. I.e., the *scope* of a delegation is the minimum LSN to which the delegation applies.

The Delegated Objects Table is kept in volatile storage and saved to stable storage when the system writes a checkpoint.

object	*deleg by trans ID*	*locks, etc.*
A	~~t_1~~	
B	t_1	

Object List 1

object	*deleg by trans ID*	*locks, etc.*
A	t_1	
X	t_2	
Y	t_2	

Object List 2

object	*deleg to trans ID*	*scope LSN*
A	t_2	1034

Delegated Objects Table

Figure 3: data structures after applying example delegation

Note that although DOT(ob)= t_1 implies that transaction t_1 is *responsible for ob*, the reverse does not hold, e.g. a transaction may have become responsible for an object by accessing it, so DOT $\subset_{strict}$ *Responsible-trans.*

We also introduce a new log record type: *delegate*. Its fields record the two transactions and object(s) involved in the delegation. The fields are: LSN (log-sequence number), Type (delegation), Trans-ID (which transaction created the record, in this case the delegator), PrevLSN (link for the Backward Chain), Page-ID, UndoNxtLSN (for recovery backward chain), and Data (in this type, Data contains the delegated object and delegatee Transaction ID). For other record types, we just note here that all update and transactional event log records have *Trans-ID* field. *Trans-ID* indicates the name of the transaction whose action the record logs. This follows the format of log records in ARIES, see [10] for details on other record types.

4.2 Normal Processing

For normal processing ARIES/RH augments ARIES in two ways. It logs delegations as transactional operations, updating the *Delegated Object Table.* Also, it cleans up the Delegated Objects Table and other data structures on transaction termination. Specifically, he processing is as follows:

delegate When a transaction t_1 executes $delegate(t_1, t_2, ob)$,

1. DELEGATION WELL-FORMED? We search the *Delegated Objects Table*:
   - If there is no entry for ob in Delegated Objects Table (the object had never been delegated before), the delegation is OK (well-formed).
   - If $ob \in$ DOT, and DOT(ob)= t_1 (the transaction listed in Delegated Objects Table matches the delegating transaction), the delegation is OK.
   - If $ob \in$ DOT but $t_1 \neq$ DOT(ob) (the delegating transaction does not match the transaction listed in Delegated Objects Table), the delegation is malformed (NOK) so we ignore it,[1] ending the processing of the delegation.
2. APPLY DELEGATION. If the previous step is OK,
   (a) Write to the log a *delegate* record containing the delegating transaction id t_1, the delegatee's id t_2, and the object name ob. (This log record is linked into the delegating transaction's BWC$_1$.)
   (b) DOT(ob)$\leftarrow t_2$, i.e., update DOT to reflect the delegation (creating a new entry if ob was not there already).

[1]A good implementation will issue a warning.

(c) Modify other data structures to reflect the transfer of responsibilities. I.e., transfer *ob* from t_1's to t_2's object list OL_2, $\mathrm{OL}(ob)$.deleg $\leftarrow t_1$ (marking *ob* as obtained through delegation from t_1).

commit When a transaction commits, write a commit record to the log. Find, from the transaction's Object List, which objects were acquired through delegation. For each delegated object ob_{del}, delete its entry or adjust its scope as necessary (e.g., $\mathrm{DOT}(ob_{del}) \leftarrow null$).

abort Write an abort record to the log. Discard the changes to objects owned by the aborting transaction. Find, from the transaction's Object List, which objects were acquired through delegation. For each delegated object ob_{del}, delete its entry $\mathrm{DOT}(ob_{del})$. Notice that any object that had been delegated *by* the aborting transaction will no longer be in the transaction's OL.

The other transactional events are processed as in ARIES [10].

4.3 Recovery Processing

After a crash, the transaction system must do some recovery processing to return to a state consistent with the correctness criteria. 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.

Since our recovery follows ARIES, we summarize it (from [10]) briefly here. ARIES scans the log in one or two *forward* passes: analysis and redo; and then a *backward* pass: undo. See Figure 4. 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 reestablishes the state of the database at failure time, including uncommitted updates. Finally, the *undo* pass rolls back all the updates by loser transactions in reverse chronological order starting with the last record of the log.

In some ARIES variants the analysis and redo passes can be merged in a single forward pass. ARIES/RH uses one forward pass and one backward pass only, so it does not add any passes to ARIES. We describe next the additions to ARIES that allow ARIES/RH to support delegation and recovery.

4.3.1 Forward Pass

During the forward pass, ARIES/RH examines the delegate records in the log, updating Delegated Objects Table accordingly. At the end of the forward pass, we know, for each delegated

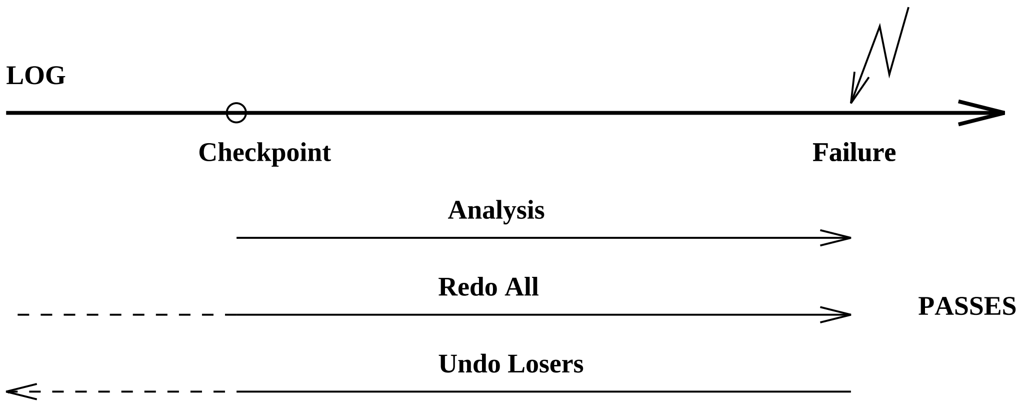

Figure 4: ARIES passes over the log

object, the relevant transactions that manipulated it; i.e., the last delegating and the last delegatee transactions.

delegate Scanning the log, on finding a record of $delegate(t_1, t_2, ob)$, we do:

1. LOOKUP. We search for ob in *Delegated Objects Table.* If not found (the object had never been delegated before), we create an entry for ob in Delegated Objects Table. Otherwise we have DOT(ob)= t_1.
2. RECONSTRUCT Delegated Objects Table. We set DOT(ob) $\leftarrow t_2$.

commit On encountering a $commit(t)$ record, we scan t's object list Object List and we *adjust scope* for each delegated object in Delegated Objects Table. Notice that any subsequent delegation of ob should *not* affect the log records preceding this commit. This is because a transaction can only delegate uncommitted objects, and the same object may be used after it has been committed by t. To avoid applying delegation to updates that have been committed, we maintain and check the scope of the delegation, which is the Log Sequence Number of the last transaction to commit this object. So adjusting scope consists of recording, on a per-object basis, the LSN of the transaction that committed it, and it is done as part of the commit event.

On other types of records, the forward pass proceeds as in ARIES. At the end of this pass Delegated Objects Table reflects the delegations that had taken place up to the failure.

Notice that we do *not* alter the log in this pass.

4.3.2 Backward Pass

In this section we discuss the delegation processing that is integrated with the *undo* pass of ARIES. The only records that require extra processing are the updates; all others are processed by unmodified ARIES.

During the backward pass we actually apply the delegations, as follows. For each delegated object *ob*, we *move* each of its update records, from the Backward Chain of the transaction that was responsible for *ob* at the last checkpoint to the BWC of the transaction that last received *ob* in delegation. Note that this entails moving some *Log Sequence Number pointers* in the log records concerning *ob*, not the log records themselves. We also *change* each update log record so that its *Trans-ID* field now contains the transaction id of the last delegatee transaction. For each changed update record we write a *Compensation Log Record* (CLR) to prepare for crashes during recovery.

To process update records in the backward pass we need Delegated Objects Table to list objects with pending delegations at the point of the failure. We use DOT to check if an update is affected by some delegation and thus needs to be modified. As pointed out above, DOT is reconstructed in the forward pass of ARIES/RH recovery.

We also need TL to find the last record written by every transaction and initialize the BWCs. TL is maintained by ARIES and reconstructed during the forward pass. As we process a record for e.g. t_1, we update BWC_1 to point to the predecessor of the record processed.

update Scanning backwards, on finding, at LSN, a record for $update(t_1, ob)$ we do:

1. MODIFY RECORD? We search for *ob* in *Delegated Objects Table*. If $ob \in$ DOT, *and* DOT(*ob*).scope $<$ LSN (the record is within the scope of the delegation) this record must be changed and we go to the next step. Else ($ob \notin$ DOT or LSN $\leq$ DOT(*ob*).scope) we are done with this record, and we exit.

2. REWRITE RECORD. We change the record's *Trans-ID* to the delegatee transaction DOT(*ob*) (say, t_2). I.e., $update(t_1, ob).TransID \leftarrow$ DOT(*ob*), which turns the current record into $update(t_2, ob)$.

3. MOVE RECORD TO NEW BWC. We move $update(t_2, ob)$ from BWC_1 to BWC_2 (recall DOT(*ob*)$= t_2$). We write the CLR for this update to the record. Now we are done with this record.

In general, not all the log fits into volatile memory, so the transaction system brings portions of the log into a buffer as needed. The (contiguous) portion of the log currently in the buffer is the *buffer window*, and it is delimited by the Log Sequence Numbers of its two ends. To move a record from BWC_1 to BWC_2 we must change various pointers in both chains. These pointers are the PrevLSN field part of various log records written by t_1 and t_2. In the algorithm presented above, we assume that all the necessary records are contained in buffer window. Because of locality, this is a reasonable assumption, but one can envision situations, such as multiple successive delegations, or long-lived transactions, where some of the relevant records are in a portion of the log *outside* the current buffer window. Simply fetching the appropriate individual records from disk is undesirable because it leads to short, scattered accesses with high overhead. Long, contiguous accesses (like the ones that shift the window along the log) are better because they spread out fixed access costs (such as latency) over many records.

In some cases, it is possible to bring in all the records needed by increasing the size of the buffer, and reading in the next portion of the log, which was to be examined next anyway. In general, the solution is to defer those pointer changes whose records are currently outside the buffer, and apply them when their portion of the log is reached by the backward pass. We accumulate the pointer changes in a priority queue (e.g. a heap) sorted by Log Sequence Number. Each time the buffer window is moved, we apply all the pointer changes for the new range of LSN s. Unfortunately, this strategy is vulnerable to crashes, which may leave inconsistent BWCs if some pointers are changed and others are left undone in the priority queue. We see two possible ways of addressing this. One is to log the pending changes, so that the priority queue can be reconstructed after a crash, but we must investigate how to avoid nested recovery problems as using CLRs becomes more complex. The other solution is to require that the chain move be atomic. We achieve this by keeping more than one contiguous portion of the log in main memory, in effect using extra buffer windows, and only logging a CLR after all pointer switches have been done. This second solution imposes extra disk overhead due to non-contiguous accesses. However, since we expect that locality will generally keep related records close together in time and in the log, either strategy should not be needed frequently enough to have a major effect on the performance of the system.

5 Discussion

In this section we discuss two issues on the implementation. First we analyze aspects of the algorithm presented in (Section 4) to argue that it implements delegation correctly. In the second part, we look at the implementation from the point of view of efficiency.

5.1 ARIES/RH Implements Delegation

Here we make some observations on why our algorithm correctly implements delegation. The algorithm operates correctly during normal processing, i.e., without considering failures. It is easy to see that the supporting volatile data structures (Delegated Objects Table, per-transaction Object List) are manipulated correctly by delegate, commit, and abort.

For the semantics in the face of crashes, we consider the forward and backward passes. The forward pass reconstructs the supporting data structures from a checkpoint and the log, and then it is as in normal processing. Notice that in the case of successive delegations of an object, it is correct to only retain the last delegation and its scope, as we identify which update needs to be modified (in the backward pass) based on the *object* not the transaction involved.

The backward pass applies the changes indicated by a delegation. We retain ARIES's use of Compensation Log Records, for the case of crashes *during* recovery. We extend its use to the case of modification of the log –when we move a record from one BWC to another–, and we ensure that such modifications are done atomically. This ensures that delegation is implemented correctly in the face of crashes.

5.2 ARIES/RH is Efficient

We claim that ARIES/RH is efficient in the following senses. (i) When delegation is used its overhead is proportional to how much it is used (i.e., number of delegations). No overhead is incurred when delegation is not used, as the algorithm reduces to conventional ARIES. (ii) Delegation's overhead during normal processing is low, comparable to the overhead of e.g. posting an update. (iii) During recovery, delegation's overhead is generally low, as it amortizes its accesses to the log by piggy-backing them whenever possible to the conventional ARIES log sweep. We point out pathological recovery cases, which we expect will rarely occur; further research is need to evaluate this. (iv) Even in the case of costs e.g. proportional to the number of delegations, they are not significant if they entail e.g. a search on an in-memory table (such as Delegated Objects Table) as compared to accesses to disk.

Let's examine (i) and (ii). It is easy to see that if delegation is not used, there is no information to upkeep. Since ARIES keeps the Backward Chain anyway, we do not need to do extra work to keep track of where the updates done by a specific transaction are. (ii) Every time a delegation is invoked, there are two operations to consider for cost. A lookup/update to the transaction's Object List (proportional to the number of objects the transaction is responsible for), And a lookup/update on the Delegated Object Table, which can be optimized but is at worst proportional to the number of delegations. Both OL and DOT are in main memory, and we expect Object Lists will be short, and the Delegated Object Table will be implemented for speed. At transaction termination we also have the added cost of removing a transaction's objects from Delegated Objects Table (or mark their scope); the Object List can be simply discarded.

To see (iii), first note that during recovery, the simpler algorithm (buffering issues ignored) presented in the first part of 4.3 is very efficient, as it visits log records when ARIES brings them to effect the recovery. In other words, the log manipulations necessary for delegation are applied alongside with the other tasks of ARIES: analysis, redo, and undo. More specifically, the forward pass of recovery is only different to ARIES in its processing of commit and delegate. For a commit record we have the additional work of recording the scope for each of the delegated objects for which the committing transaction is responsible. This entails a lookup in Delegated Object Table for each object in the transaction's Object List. For a delegation we just make an entry in the DOT to record the delegation. Most of the time both the Delegated Object Table and the Object List are small; in general they are much smaller than the log being analyzed.

This is important because the dominant cost during recovery is bringing the log from stable storage; this supports (iv).

For the backward pass of recovery, we need only examine the cost for updates. For each update, we do a lookup in Delegated Object Table, and check of scope, changes to the record itself, changes to pointers (to move the record from one backward chain to the other), and writing a Compensation Log Record. There are two issues to consider here. One is disk access: in contrast with ARIES, here we must schedule not just an append, but also write (the modified record) in the middle of the log. We expect to cluster those modifications to optimize access.

The other factor that can degrade performance severely is fragmented buffer problem, discussed at the end of 4.3.2. We expect that locality of updates by the transactions will make this an infrequent occurrence, but studies are needed to assess the effect of fragmented-buffer, as well as mid-log writes, on performance.

In summary: we expect that ARIES/RH will perform almost as well as ARIES during normal processing and somewhat worse (depending on locality and certain disk access parameters) during recovery. Although a slower recovery is undesirable, we believe it is a reasonable trade-off in exchange for good performance during normal processing.

6 Conclusions

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 [3, 4, 5], and the language primitives introduced in ASSET [2], which provides linguistic primitives to realize Extended Transaction Models.

The main contribution of this paper is ARIES/RH, an algorithm to implement delegation in an efficient, robust, industrial-grade transaction management system. This is a crucial step towards the flexible synthesis of Extended Transaction Models. Current work has produced many ETMs, but each with its own tailor-made implementation; with delegation (and the other two ASSET primitives, namely, permit and form-dependency, which are easy to implement [2]) we believe we can now offer performance comparable with ad-hoc implementations coupled with great flexibility.

We introduced the novel concept of *rewriting history*, which provides the framework for an efficient implementation, and especially, a clear *recovery model* for transaction delegation. 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. Basing our algorithm on ARIES allowed us to secure the benefit of ARIES's robustness and efficiency, by modifying ARIES with close attention to the performance and recovery issues.

Rewriting history is a natural extension of the repeating history paradigm of ARIES [10] and is a generalization of ARIES/NT [14]. ARIES/NT is an extension to ARIES for nested transactions [11].

Delegation allows a transaction to manipulate a TMS data structure –the log– whose access is traditionally restricted or denied altogether to the application level. A good discussion of the issues of allowing transactions access to *extra data* is in [6].

Related work includes Structured Transaction Definition Language [1], 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.

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 [7]. 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 certain Extended Transaction Model. The Transaction Management Mechanism is programmable, but using 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 or *pre-built* components, and so it approaches the problem at a coarser grain. This may allow for initial gains in performance, but we believe that language primitives is a more flexible approach.

Also related is work in the ConTract model [15]. 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 we introduce a small set of transaction management primitives that can be embedded in a host language.

In further work we will model and evaluate the performance of ARIES/RH, and study the broader issues for providing robust, efficient, and flexible transaction processing.

References

[1] 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.

[2] 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.

[3] P. K. Chrysantis, and Krithi Ramamritham. Synthesis of Extended Transaction Models using ACTA. ACM Trans. on Database Systems (to appear).

[4] 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.

[5] 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.

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

[7] 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.

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

[9] Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufman, 1993.

[10] 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.

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

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

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

[14] 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.

[15] H. Wächter and A. Reuter. The ConTract Model. In [8].

[16] 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.