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

Abstract

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 Advanced Transaction Models.

With an efficient implementation of delegation it becomes practicable to realize various Advanced 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 advanced transaction models, and illustrate this with our solution in ARIES, an industrial-quality system that uses UNDO/REDO *recovery. Since delegation is tantamount to rewriting history, a naïve implementation can entail frequent, costly log accesses, and can result in complicated recovery protocols. Our algorithm achieves the effect of rewriting history without rewriting the log, resulting in an implementation that realizes the semantics of delegation at minimal additional overhead and incurs no overhead when delegation is not used.*

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

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 *Advanced Transaction Models* (ATMs) have been proposed [7, 17], 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 ATMs. 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 ATMs 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 ATMs. In more operational terms, ASSET [3] provided a set of new language primitives that enable the realization of various ATMs 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. With delegation 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 ATMs 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 ATMs that can be synthesized using delegate are Joint Transactions, Nested Transactions, Split Transactions, and Open Nested Transactions [6, 8].

Biliris et al. [3] 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).

Our goal in this paper is to develop a *robust* and *efficient* implementation of delegation that takes advantage of *existing, industrial-strength* technology. To this end, we consider log-based systems because logs are a feature of many practical systems, which rely on logs for auditing and other functions besides recovery. We do not consider alternative approaches requiring new, ad-hoc infrastructure that may be easier to achieve, but does not integrate with existing technology.

To further the goal of providing general purpose machinery to support the specification and implementation of arbitrary ATMs, we have developed efficient implementations of delegation on two log-based recovery systems. For lack of space we present here only our ARIES[13] -based solution. In a longer version of this paper [15] we demonstrate the correctness of the implementation and also discuss how to implement delegation on EOS [4]. Our additions allow the "efficient Rewriting of History," so we call our protocol ARIES/RH.

By providing delegation, we add substantial semantic power to a conventional Transaction Management System (TMS), allowing it to capture various ATMs. 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 rewriting of the log.

In this paper we argue that:

- Delegation is a powerful, important primitive for realizing ATMs. 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 such as ARIES, such that when delegation is not used no overheads are incurred. Development of a provably correct and efficient implementation is a very important contribution because it shows that a generic, flexible, and practical ATM facility can be produced.

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 ATMs. 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. In section 4 we discuss why our algorithm efficiently implements delegation. In section 5 we review related work; in section 6 we present our conclusions and discuss future work.

2 Delegation: Properties and Examples

In this section, first we introduce some notation, then we explain delegation in terms of visibility and recovery, and then point out its important properties. Finally, we present examples of ATMs and show how to synthesize one of them using delegation.

2.1 What: Concepts and Properties

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

- $t, t_0, t_1, t_2, \ldots$ denote ***transactions***; $ob, a, b, \ldots$ denote ***objects*** in the database.
- ***update*** is a generic operation on database objects. We write $updt[ob]$ 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.
- $delegate(t_1, t_2, updt[ob])$ denotes ***delegation*** by t_1 to t_2 of $updt[ob]$.
- ***Invoking transaction.*** We call the transaction that invoked the update on the object the *invoking* transaction. We write $updt[t, ob]$ when we wish to indicate that t is the invoking transaction for that update.
- 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.*** A transaction responsible for an update is in charge of committing or aborting it, unless it delegates it: $ResponsibleTr(updt[ob]) = t$ holds from when t performs $updt[ob]$ or t is delegated $updt[ob]$ until t either terminates or delegates $updt[ob]$.[1]
- ***Op_List.*** The dual of *ResponsibleTr* is the *Op_List*. It contains the operations a transaction is responsible for: $Op\_List(t) =$
  $= \{updt[ob] \mid ResponsibleTr(updt[ob]) = t\}$.

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

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

[1] Notice that without delegation, the transaction responsible for an update is always the invoking transaction.

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

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

Commit/Abort of Updates. In the presence of delegation, the fate of updates to an object is that of the transaction to which the operation was last delegated. For instance, if t_0 does $updt[ob]$, then delegates $updt[ob]$ to t_1, and t_0 subsequently aborts, the changes t_0 made to ob via $updt[ob]$ will still survive if t_1 commits while it is still responsible for $updt[ob]$:

- $(Commit(t) \in H) \Longleftrightarrow (\forall updt[ob] \in Op\_List(t) : (commit_t(updt[ob]) \in H))$
  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.
- $(Abort(t) \in H) \Longleftrightarrow (\forall updt[ob] \in Op\_List(t), (abort_t(updt[ob]) \in H))$
  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 $commit_t(updt[ob])$ and $abort_t(updt[ob])$ indicate the permanence or obliteration of the changes done by $updt[ob]$. In the presence of delegation, the changes may have been made by either t or other transaction(s) which eventually delegated $updt[ob]$ to t.

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.

Other Properties of Delegation. An operation can be delegated only by the transaction that is responsible for it. Since $ResponsibleTr(updt[ob])$, is at any given time, unique, only one transaction can delegate an operation at any point. 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.

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 ATMs – Examples

In this section we motivate delegation through examples of its application in the synthesis of ATMs. Other examples can be found in [6, 5].

Inheritance in Nested Transactions [14] is an instance of delegation. It is achieved through the delegation of all the changes the child transaction t_c is responsible for to its parent t_p when t_c 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 t_1 may *split* into two transactions, t_1 and t_2, at any point during its execution. 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. (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.)

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
```

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 give the operational semantics of delegation. 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.

3.1 Operational Semantics

In a generic Database System 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

K ← currLSN (*LSN of* delegate *record*)
while LOG[K] is not the initiate record for t_1
  if LOG[K] is an update to ob by t_1
    then setTransID(K,t_2) *now looks as if done by* t_2
  K ← prevLSN(K,t_1) *follow* t_1 *'s BC*

Figure 1. Op. Semantics of *delegate*(t_1, t_2, ob)

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.

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 section 3.3). The `delegate` record is a new type of log record for 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 ← 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 the record pertains to), and Data. For delegate records there also exist two LSN pointers, one to the delegator and another to the delegatee (see section 3.4).

> ***Example 1.*** Consider the log fragment (see fig. 2):
>
> $\ldots updt[t_1, a], updt[t_2, x], updt[t_2, a],$
> $updt[t_1, b], updt[t_1, a], updt[t_2, y]$
>
> After the application of delegate(t_1, t_2, a), the log looks like:
>
> $\ldots updt[t_2, a], updt[t_2, x], updt[t_2, a],$
> $updt[t_1, b], updt[t_2, a], updt[t_2, y]$

before rewriting

100	101	102	103	104	105	106
update t_1 a	update t_2 x	update t_2 a	update t_1 b	update t_1 a	update t_2 y	delegate $t_1 \xrightarrow{a} t_2$

after rewriting time ⟶

100	101	102	103	104	105	106
update $\cancel{t_1}$ t_2 a	update t_2 x	update t_2 a	update t_1 b	update $\cancel{t_1}$ t_2 a	update t_2 y	delegate $t_1 \xrightarrow{a} t_2$

Figure 2. Log in Example 1.

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 (apply each delegation to the log as the delegation is issued) carries high performance costs, due to the per-delegation random-access to the log, and is also hard to prove correct because we manipulate the log outside the usual append-only mode, complicating the model with extra data.[2]

The observation that during normal processing the log is not consulted suggests an algorithm that keeps track of the effect of delegations in volatile data structures, and logs the delegations. After a a crash the delegations can be applied by modifying the log – rewriting history – during recovery. Still, one must address issues of performance and correctness in the face of failure, because the log is accessed/modified randomly. Correctness is ensured if each BC switch is done atomically.[3] Performance, however, is hostage to the way the log is accessed. In general, the log does not fit wholly into volatile storage, resulting in thrashing, as the algorithm needs to jump over possibly large sections of the log to follow backward chain pointers.

To avoid these pitfalls, we propose RH, a "lazy" algorithm for rewriting history that *does not modify the log*, as follows. 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. Thus we "rewrite the history" of the system not by modifying the log, but by *interpreting* the log during recovery according to the delegations.

3.3 Conventional Recovery: ARIES

We review ARIES to establish context and terminology. ARIES uses an UNDO/REDO protocol, which means that after a crash, updates by a *loser* (uncommitted) transaction will be undone and those by a *winner* (committed) redone. ARIES scans the log in three passes, see figure 3.[4]

The (forward) *Analysis* pass updates the information on active transactions and determines the "loser" transactions. The (forward) *Redo* pass *repeats history*, writing to the database those logged updates that had not been applied to the database before the crash; this re-establishes the DB state at failure time, including uncommitted updates. The

[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. [8] discuss the issues of correctness with extra data.

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

[4] Some variants of ARIES merge the two forward passes into one, thus we also use only one forward pass.

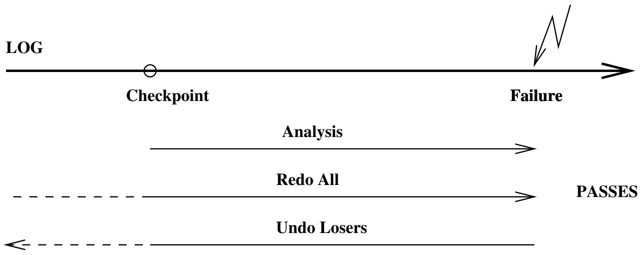

Figure 3. ARIES passes over the log.

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

ARIES keeps, for each transaction, a *Backward Chain* (BC, see figure 4). All the log records pertaining to one transaction form a linked list BC, accessible through Tr_List, which points to the most recent one. ARIES inserts *compensation log records* (CLRs) in the BC after undoing each log record's action.[5] Applying $delegate(t_1, t_2, ob)$ is tantamount to removing the subchain of records of operations on ob from BC(t_1) and merging it with BC(t_2). But this is not a viable implementation as it would demand unacceptably frequent, random accesses and updates to the log; instead, we have devised efficient algorithms that support delegation without modifying the log, which we discuss next. First we present the data structures, and we explain the normal processing. We then examine recovery processing, first the forward (analysis & redo) pass and then the backward (undo) pass.

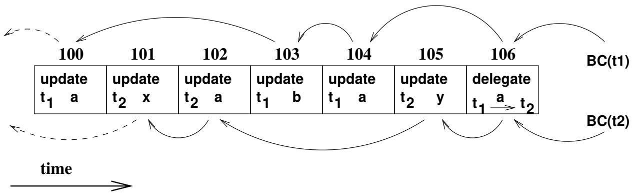

Figure 4. Backward Chains in the log.

3.4 Data Structures

We must know which operations on which objects each transaction t is responsible for, i.e., its $Op\_List(t)$ (see section 2.1). For that we use the *Transaction List* and expand each transaction's *Object List* found in conventional Database Systems; we also add a `delegate` type log record.

Tr_List. The Transaction List [2, 10, 13] 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*.[6]

Ob_List. For *each* transaction t there is an $Ob\_List(t)$ (In figure 7, $Ob\_List(t_1)$ contains the objects t_1 is accessing after the delegation.) In terms of Op_List: $Ob\_List(t) = \{ob \mid \exists updt[t_0, ob] \in Op\_List(t)\}$, i.e., the objects for which there is an update for which t is responsible. The update may have been invoked by t_0 and the responsibility transferred to t via delegation.

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 5. Fields of the delegate log record

When transactions are responsible for specific updates (not a whole object), a certain object may appear in more than one Ob_List (but the associated updates will be different).[7] We identify the *updates* that a transaction is responsible for by introducing the notion of *scope*.

object	*Scopes*
a	(t_1, 100, 104)
b	(t_1, 102, 106)

Ob_List (t_1)

object	*Scopes*
a	(t_2, 102, 102)
x	(t_2, 101, 101)
y	(t_2, 105, 105)

Ob_List (t_2)

Figure 6. Ob Lists before delegation, Ex. 1.

For each object ob in $Ob\_List(t_1)$ there is a *set* of scopes $Scopes$, that covers the updates to ob *for which* t_1 *is currently responsible*. A scope is a tuple (t_0, l_1, l_2) where t_0 is the transaction that actually did the operations (the invoking transaction), l_1 is the first, and l_2 the last LSN in the range of log records that comprise the scope. This indicates that t_1 is responsible for all updates to ob (by t_0) between and including the two LSNs.[8] See figure 7.

Delegate Log Records. We add a new log record type: `delegate`. Its type-specific fields (see figure 5) store the two transactions and object involved in the delegation.

3.5 Normal Processing

We describe ARIES/RH in terms of how different events are processed, focusing on the differences with ARIES. (The current value of the log sequence number is CurrLSN.)

- $begin(t)$
  1. INITIALIZE. Add t to Tr_List; create $Ob\_List(t)$.
- $updt[t, ob]$
  1. ADJUST SCOPES. If this is the first update of t to ob since either 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.

[5] To avoid undoing an update repeatedly should crashes occur during recovery.

[6] For each transaction t, $Tr\_List(t)$ contains the head of the BC(t).

[7] For example, this can occur in the case of non-conflicting updates, such as increments of a counter.

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

object	*Scopes*
~~a~~	~~(t_1, 100, 104)~~
b	(t_1, 102, 106)

Ob_List (t_1)

object	*Scopes*
a	(t_2,102, 102) (t_1, 100, 104)
x	(t_2,101, 101)
y	(t_2,105, 105)

Ob_List (t_2)

Figure 7. Ob Lists after delegation, Ex. 1.

<u>**if**</u> $ob \notin Ob\_List(t)$ <u>**then**</u>
$Ob\_List(t) \leftarrow Ob\_List(t) \cup \{ob\}$;
<u>**if**</u> $(t, \_, \_)^9 \notin Ob\_List(t)[ob]$
<u>**then**</u> $Ob\_List(t)[ob].Scopes \leftarrow$
(t, CurrLSN,CurrLSN) *create new scope*
<u>**else**</u> $Ob\_List(t)[ob].Scopes \leftarrow$ (_,_,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: $pre(delegate(t_1, t_2, updt[ob])) \Rightarrow (ResponsibleTr(updt[ob]) = t_1)$.
  2. PREPARE LOG RECORD(S).
     (a) Record delegator, delegatee:
     $Rec.tor \leftarrow t_1$; $Rec.tee \leftarrow t_2$;
     (b) Link record into t_1's and t_2's backward chains:
     $Rec.torBC \leftarrow$ BC(t_1); $BC(t_1) \leftarrow Rec$
     $Rec.teeBC \leftarrow$ BC(t_2); $BC(t_1) \leftarrow Rec$.
  3. TRANSFER RESPONSIBILITY. Move operations on ob from $Op\_List(t_1)$ to $Op\_List(t_2)$.
     (a) Add ob to delegatee's Ob_List and record that ob was delegated by t_1:
     $Ob\_List(t_2) \leftarrow Ob\_List(t_2) \cup \{ob\}$;
     $Ob\_List(t_2)[ob].deleg \leftarrow t_1$.
     (b) Pass delegator's Scopes for ob to the delegatee:
     $Ob\_List(t_2)[ob].Scopes \leftarrow$
     $Ob\_List(t_2)[ob].Scopes \cup$
     $Ob\_List(t_1)[ob].Scopes$;
     (c) Remove ob from the delegator's Ob_List:
     $Ob\_List(t_1) \leftarrow Ob\_List(t_1) - \{ob\}$.

     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 then 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.

[9] '_' denotes a field that we do not change or are not interested in.

- $commit(t)$
  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 point up to the commit record inclusive.
- $abort(t)$
  1. ABORT OPERATIONS. Undo the updates for which t is responsible. (Recall that any object previously delegated *by* the aborting transaction is no longer 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 abort log record to log.
  3. FLUSH LOG. Flush log up to abort record.

We process other transactional events as usual [10, 13].

Note: The preceding algorithm assumes that delegation is used often, but the overhead for transactions that do not use delegation can be reduced to a minimum with the following optimization. We delay creation of an object's entry in a transaction's Ob_List until the transaction either delegates or is delegated some operations on that object. We can do that because we can assume that the first scope for an object (i.e., the scope for the first time the object is delegated) without an entry in Ob_List spans from the begin LSN to the delegate LSN of the delegating transaction, instead of starting at the first update of the transaction to that object. Furthermore, when a transaction delegates an object it removes the associated scopes but keeps an empty entry in its Ob_List, so that it knows to open a new scope when and if it accesses the object later. Without the retention of this entry it could erroneously assume that the scope starts at the begin LSN. If the algorithm is modified in accordance with this change, when delegation is not used it reduces to the traditional algorithm. (This also reduces the size of the object lists – maintained in main memory – since only entries pertaining to delegated objects need be maintained.) Thus, the only extra work for non-delegated objects during normal processing is checking whether they have an entry in Ob_List. For details see [15].

3.6 Crash Recovery

After a crash, the transaction system must recover to a consistent state, 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.[10]

In the rest of this section, we present the recovery phase of ARIES/RH, which includes a forward pass and a backward pass. Here *Crash* is the event that represents a failure; *RecoveryComplete* indicates that the recovery is complete. $Winners$ and $Losers$ are as described in 3.4.

3.6.1 Forward Pass

Before the first pass of recovery starts, $Winners = Losers = \phi$. At the end of the forward pass *Winners*, *Losers*, and *Object List*s are up to date, including the scopes of the updates.[11]

- $begin(t)$
  1. INITIALIZE. Add t to Tr_List; create $Ob\_List(t)$.
  2. LOSER BY DEFAULT. Consider t a loser by default.
     $Losers \leftarrow Losers \cup \{t\}$.
- $updt[t, ob]$
  1. ADJUST SCOPES. This is done just as in step (1) of $update$ in normal processing.
  2. REDO. Redo $updt[t, ob]$.
- $delegate(t_1, t_2, ob)$
  1. TRANSFER RESPONSIBILITY. This is done just as in step (3) of $delegate$ in normal processing.
- $commit(t)$
  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 the state is as follows.

- 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_Lists 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

To undo loser transactions, ARIES continually undoes the update with maximum Log Sequence Number (LSN), ensuring monotonically decreasing (by LSN) accesses to the log, with the attendant efficiencies.

ARIES undoes all the updates *invoked* by a loser transaction. In the presence of delegation, what we need instead is to undo *all the updates that were ultimately delegated to a loser transaction*. 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.[12]

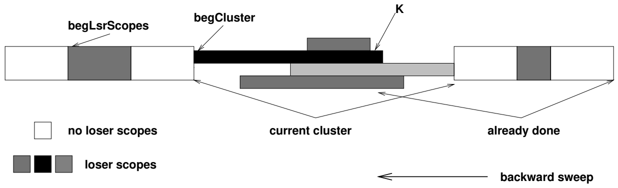

Figure 8. Loser scope clusters in the log.

An extension analogous to ARIES involves constructing and maintaining backward chains of loser updates, an expensive solution. Or 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 also undesirable as it unnecessarily inspects many winner updates.

Fortunately, the information of update scopes suffices to efficiently undo loser updates. In the rest of the section we discuss how undo and delegation are integrated in the backward pass. Update and delegation are the only records that require special processing.

Recall that scopes keep track of updates that were delegated together (see section 3.4). It is enough to inspect records within the *loser* transaction scopes to find all loser updates. To do it efficiently, we introduce the notion of a *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 8 there are three clusters; the middle one contains four overlapping loser scopes. 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.

We can now outline the algorithm for the backward pass of ARIES/RH.[13]

Within a loser cluster we examine all records, undoing loser updates. We also adjust the current cluster by adding or deleting scopes, closing it 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 LsrScopes, the set of all the scopes covering loser updates. LsrScopes contains 4-tuples:[14] the object, the invoking transaction, and the two LSN values for the range of update records. We compute

[10] For instance, the scope information is saved at checkpoints.

[11] We only describe the treatment of the log records relevant to delegation; other records are processed as usual.

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

[13] References in parentheses correspond to the algorithm in figure 9.

[14] The scopes of section 3.4 plus the object id.

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 figure 8) from the end of the rightmost loser scope, and end at begLsrScopes (see **while** loop). 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 ($\alpha 2$). 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 ($\alpha 4$).

The next cluster to process begins at the end of the now rightmost scope in LsrScopes (β), because when we add a scope to Cluster we remove it from LsrScopes ($\alpha 1$).

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 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. This reduces the cost of bringing the log from disk. We construct the set of scopes LsrScopes once and deplete it in the reverse order of scopes, so it can be kept in a priority queue (on a heap) sorted by right end of scopes, with the largest value first. We follow invoking transactions to process the set of scopes Cluster; we add scopes on the left and delete scopes on the right. A binary tree keyed on transaction ids is a reasonable implementation.

Transaction Rollback

In the preceding we discussed how to support the more complex recovery from a *Crash*, which forces the abortion of many in-progress transactions. Rolling back a single transaction (aborted because of a logical error, deadlock, etc.) is similar. As before, we must undo only the (loser) updates for which the transaction t is responsible at abort time. We only sketch the algorithm here in the interest of space. For each object in $Ob\_List(t)$, we must ensure the updates are undone. We construct the set of loser object scopes (see figure 9) and we process it undoing the updates in reverse chronological order. The main issue is the manipulation of the log in the buffer, which must be done carefully to preserve consistency in the face of failures. We adopt a strategy similar to that of ARIES/NT [18], writing CLRs as we undo the updates.

See sections 3.4 and 3.6.2 and figure 8 for explanations of the variables used in this algorithm.

```
LsrScopes ← { ob, scope | ∃ob, ∃t ∈ Losers :   all loser
                scope ∈ Ob_List(t)[ob].Scopes}  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 backwards to left
                          end of earliest loser scope
  (α) repeat  identify & process overlap'g scopes cluster
      (1) add to Cluster the loser scope that ends in K
          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
      (2) undo if it is loser update
          if LOG[K] = updt[t, ob] and
          ∃ (ob, t,_,_) ∈ Cluster then
              undo(updt[t, ob])
              CLR.PrevLSN ← LOG[K].PrevLSN
      (3) discard scope that begins at new record LOG[K]
          if ∃ (_,_, left,_) ∈ Cluster and K = left then
              already processed, so discard
              Cluster ← Cluster − {(_,_, left,_)}
      (4) processed record LOG[K], next (left) ...
          K ← K − 1
  (end α) until K < begCluster
      finished this cluster (sweep backwards)

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

Figure 9. Backward pass of ARIES/RH

4 ARIES/RH is efficient

ARIES/RH, presented in sections 3.5 and 3.6, is correct and efficient. To ensure correctness in a recovery protocol, we must guarantee that, after recovery, all updates by *loser* transactions have been rolled back completely (their effects obliterated) and all updates by *winner* transactions have been committed (their effects made permanent). Conventional ARIES complies by using the UNDO/REDO protocol. With delegation, we must also ensure that operations *ultimately delegated* to loser transactions will be aborted, and operations *ultimately delegated* to winner transactions will be committed. For lack of space we omit the proof here, which can be found in our technical report [15].

The rest of this section argues that ARIES/RH is efficient.

- **No delegation: negligible overhead**. With the optimization outlined in the note in 3.5, in the absence of delegation ARIES/RH reduces to the original algorithm plus a test: at updates, checking whether there is an entry for the object in the Ob_List. Thus the penalty when the delegation functionality is not used is negligible.
- **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_Lists.
- **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 similar to those used by the conventional algorithm. When there is no delegation, these extra fields are not used. Delegating adds the constant time of logging the delegation operation and updating the Ob_Lists of the delegator and delegatee transactions by moving as many scopes as objects are delegated (hence the linearity), affecting Ob_Lists, which reside 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. No special sweep of the log is required because ARIES/RH does the same accesses as ARIES. 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, especially because it only has objects that are delegated. 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 other cost 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, ARIES/RH adds only minimal overhead to ARIES to support delegation.

5 Related Work

Previous work has produced many Advanced Transaction Models (ATMs), but each has its own tailor-made implementation. For instance, both Argus [12] and Camelot [19] each supports its ATM (variants of nested transactions). Split-Transactions [16] allows the commitment of partial results but in a way that is less general than delegation (see section 2.2). Thus, although efficient, these systems are limited in the models they can synthesize.

Delegation, by allowing changes in the visibility and recovery properties of transactions, is a powerful primitive for synthesizing ATMs. Our work builds on the formal foundation provided by ACTA [6, 5], and the primitives introduced in ASSET [3]. With delegation (and the other two ASSET primitives, *permit* and *form-dependency* [3]) we believe we can offer the flexibility to synthesize a wide range of ATMs. Given the efficient implementation of delegation presented in this paper, we believe we can achieve this flexibility at a performance comparable to that of tailor-made implementations.

Our research also builds on the substantial work in ARIES [13], in particular ARIES/NT [18], an extension of ARIES that supports nested transactions [14]. Rewriting history is a natural extension of the repeating history paradigm of ARIES, and it has served as a powerful abstraction to guide our efforts.

More specifically, in order to support the rewriting of history, we extended the idea of chaining back log records introduced in ARIES to a smaller granularity: the sub-chain of the log pertaining to a particular set of object updates. ARIES/NT extends the backward chain to a tree (reflecting the nested transaction structure). Our method is more general: by keeping information about which transaction is responsible for individual updates, we can support not just Nested Transactions, but many ATMs, such as Split Transactions, Co-Transactions, and Reporting Transactions. Thus our extension of ARIES, while including it, goes beyond the functionality provided in ARIES/NT, and allows the efficient synthesis of arbitrary ATMs, not just nested transactions.

Work related to ours includes the Transaction Specification and Management Environment (TSME, [9]), in which specifications are mapped to certain configurations of *prebuilt* 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. Barga and Pu [1] explore another modular approach, based on metaobject protocols [11], and incorporate some elements of the TSME approach and some of our language-based approach. Also related is the ConTract model [20], where 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.

6 Conclusions

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 ATMs:

- 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 we have also discussed elsewhere [15] 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 used.

We have also demonstrated (elsewhere, [15]) the correctness of our implementation, showing that it satisfies the desired transaction properties in the presence of delegation.

We are exploring the issues of delegating components of complex objects. 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 providing a variety of recovery primitives to a transaction programmer so that different recovery requirements and recovery semantics can be achieved flexibly.

References

[1] R. S. Barga and C. Pu. A practical and modular implementation of extended transaction models. In *Proc. of the 21st Int'l Conf. on Very Large Data Bases*, Zürich, Sept. 1995.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass., 1987.

[3] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, and K. Ramamritham. ASSET: A system for supporting extended transactions. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Minneapolis, Minn., June 1994.

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

[5] P. K. Chrysanthis and K. Ramamritham. Delegation in ACTA as a means to control sharing in extended transactions. *IEEE Data Eng. Bull.*, 16(2):16–19, June 1993.

[6] P. K. Chrysantis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Trans. on Database Systems*, Sept. 1994.

[7] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, Calif., 1991.

[8] N. Gehani, K. Ramamritham, and O. Shmueli. Accessing extra database information: Concurrency control and correctness. Technical Report 93-081, University of Massachusetts, Amherst, Mass., 1993.

[9] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and management of extended transactions in a programmable transaction environment. In *Proc. IEEE Int'l Conf. on Data Eng.*, page 462, Houston, Tex., Feb. 1994.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, Calif., 1993.

[11] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Mass., 1991.

[12] B. Liskov and R. W. Scheifler. The Argus Language and System. In *Lecture Notes on Computer Science* Springer Verlag, volume 190, Berlin, 1982.

[13] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. on Database Systems*, 17(1):94–162, Mar. 1992.

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

[15] C. Pedregal Martin and K. Ramamritham. Delegation: Efficiently rewriting history. Technical Report 95-090, University of Massachusetts, Amherst, Mass., Dec. 1995.

[16] C. Pu, G. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. In *Proc. of the 14th Int'l Conf. on Very Large Data Bases*, pages 26–37, Los Angeles, Calif., Sept. 1988.

[17] Krithi Ramamritham and Panos K. Chrysanthis Advances in Concurrency Control and Transaction Processing. IEEE Computer Society Press, Los Alamitos, Calif., 1996.

[18] K. Rothermel and C. Mohan. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In *Proc. of the 15th Int'l Conf. on Very Large Data Bases*, Amsterdam, Aug. 1989.

[19] A. Z. Spector, R. Pausch, and G. Bruell. Camelot: A flexible, distributed transaction processing system. In *Proc. of Compcon*, Feb. 1988.

[20] H. Wächter and A. Reuter. *The ConTract Model*. In Elmagarmid [7], 1991.