Implementation of Commit Timestamps in Avalon

Maurice P. Herlihy Su-Yuen Ling Jeannette M. Wing

January 28, 1991
CMU-CS-91-107₂

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Atomic transactions have become a widely accepted mechanism for coping with failures and concurrency in reliable distributed systems. Much recent work has focused on concurrency control algorithms, in particular on techniques for exploiting type-specific properties of data objects to enhance concurrency. One class of concurrency control algorithms that appears particularly promising are "hybrid" schemes in which transactions are assigned timestamps as they commit. Although these algorithms have received extensive theoretical analysis, they have not been implemented because they require non-trivial systems support. In this paper, we describe the first implementation of transaction commit timestamps, as provided in Avalon/C++, a high-level language for reliable distributed computing. We focus on the run-time data structures and algorithms needed to achieve a practical implementation of transaction commit timestamps.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: Distributed computing, reliable computing, concurrent programming structures, hybrid atomicity, run-time environments, Avalon.

1. Introduction

A widely-accepted technique for preserving consistency in the presence of failures and concurrency is the organization of computations as sequential processes called *transactions*. Transactions are *atomic*, that is, serializable, transaction-consistent, and persistent. They are used in the database world to achieve data consistency in the presence of failures and concurrency, and also for reliable computations in distributed systems. Systems and languages employing transactions include Argus [LS83], Clouds [McK84], and Camelot [SBD+86].

Early work in transaction synchronization considered only untyped objects: operations were either left uninterpreted, or were treated simply as reads or writes. Experience has shown, however, that such an approach provides an inadequate level of concurrency for many non-database applications [LS83, WL85]. More recent work has focused on algorithms that enhance concurrency by exploiting properties of typed objects such as queues, directories, or counters [BGL81], [Kor83], [SS84], [Wei84a].

In Avalon/C++, the idea behind synchronization constructs is simply that each atomic object synchronizes access to itself, ensuring that transactions which access the object do so in the order in which they commit. Knowledge of the commit order comes from the commit timestamps; every Avalon transaction is assigned a timestamp generated by a logical clock [Lam78] when it commits. [1]

As an example of commit timestamp synchronization, consider this history where three transactions, X, Y, and Z operate on a queue object. Note that Ok() means that the queue operation completed.

```
Begin X
        Enqueue(1) X
Begin Y
        Enqueue(2) Y
        Ok() X
        Ok() Y
Commit(3:00) Y
Commit(3:15) X
Begin Z
        Dequeue() Z
        Ok(2) Z
        Dequeue () Z
        Ok(1) Z
Commit(3:30) Z
```

This history is not permitted by concurrency control mechanisms based on commutativity (such as read/write-locking). Since the Enqueue operation does not commute, X and Y do not commute, and so the history is not serializable in both the orders X-Y-Z and Y-X-Z. However, if we take into consideration the commit times of transactions, shown above as arguments to the Commit operation, then the above history need only be serializable in the order Y-X-Z. Thus, whereas read/write-locking does not allow concurrent Enqueues, knowledge of commit orders does.

Commit timestamp serialization has been used in a number of algorithms for highly concurrent queues, directories, etc. [DHW88], and its theoretical implications have been explored elsewhere [HW88], [Her85],

[1] These commit timestamps should not be confused with the timestamps used in multiversion protocols such as Reed's [Ree83], in which transactions are serialized in a statically predefined order.

[Wei84b].

In this paper, we describe the implementation and support for the `trans_id` class, a language level construct which permits the programmer to test transaction serialization order at run-time.

This construct has been implemented as part of Avalon/C++ [DHW88], which is a set of extensions to C++ [Str86] intended to support reliable distributed computing. Avalon/C++ is built on top of the Camelot transaction processing system [SBD+86].

Although the synchronization primitives described here are essentially language independent, our presentation assumes some familiarity with C++.

Terminology

Transactions in Avalon/C++ may be nested. A subtransaction's commit is dependent on that of its parent: aborting a parent will cause a committed child's effects to be rolled back. A transaction's effects become permanent only when it commits at the top level. We use standard tree terminology when discussing nested transactions: a transaction T has a unique parent, a (possibly empty) set of siblings, and sets of ancestors and descendants. A transaction is considered its own ancestor or descendant.

2. Programmer's View

A trans_id object is an identifier for a transaction, and its class operations can be used to test the transaction's serialization ordering at run-time. Below is the class definition of trans_id.

```
class trans_id {
     //  internal representation
public:
    trans_id(int = UNIQUE);              // constructor
    ~trans_id();                         // destructor
    trans_id&=(trans_id&)                // assignment
    bool operator==(trans_id&);          // equality
    bool operator<(trans_id&);           // serialized before?
    bool operator>(trans_id&);           // serialized after?
    bool done(trans_id&);                // committed to top level?
    friend descendant(trans_id&, trans_id&);
                  // is the first a descendant of the second?
};
```

A trans_id is usually created by calling its constructor:

```
trans_id& t = *(new trans_id());
```

The constructor creates and commits a dummy child (transaction) of the current transaction, and returns the child's trans_id to the parent. This allows a transaction to generate multiple trans_ids ordered in the serialization order of the operations that created them. The trans_id is typically used as a tag on the operation [DHW88].

On other hand, a call to

```
trans_id& t = *(new trans_id(CURRENT));
```

does not create a dummy subtransaction. Instead, it returns a trans_id for the calling transaction of the operation. Trans_id's created in this way within the same transaction are identical.

When one trans_id is assigned to another, as in

```
t1 = t2
```

the original value of t1 is lost, and t1 becomes identical to t2.

The system's current knowledge about the transaction serialization ordering can be tested by the overloaded operators $<$ and $>$. For example, if the expression:

```
t1 < t2
```

evaluates to true, then if both transactions commit to the top level, t1 will be serialized before t2. Note that because the relative serialization ordering of active transactions is unknown, $<$ induces a *partial* order on trans_ids; with one exception, while t1 and t2 are active, both t1 $<$ t2 and t2 $<$ t1 will evaluate to false. The exception is when t1 is a descendant of t2, in which case t1 $<$ t2 is true, since the descendant always commits before the ancestor. If either transaction aborts, then the expression is vacuously true. Thus, as active transactions eventually commit, they become comparable.

The `done` operation tests whether a transaction has committed to the top level. This operation is primarily used to discard unneeded recovery information within the Avalon object. To test whether one transaction is a descendant of another in the transaction tree, the following expression may be evaluated

```
trans_id::descendant(t1, t2)
```

If the expression evaluates to true, then t1 is a descendant of t2.

Examples of the use of trans_id objects can be found in the Avalon manual [WHC+89].

3. Implementation

3.1. Overview

To support the trans_id class operations, (e.g. to answer the question `t1 < t2` ?) the Avalon run-time system must keep track of the timestamps of all active and committed transactions. Transaction families may span multiple hosts, and trans_ids for each transaction may be passed in messages to processes on different hosts. Thus this timestamp information has to be globally shared, highly available, long-lived and persistent (survive site crashes). To achieve these properties, the Avalon system maintains the timestamp information in the recoverable storage of the *TidServer* at each site. This server is implemented as a Camelot server, using Camelot recoverable storage.

So, when a trans_id class operation is called, (as in `t1 < t2`) a remote procedure call is made to the local TidServer. If `t1` or `t2` represents a remote transaction then the local TidServer may call TidServers at other sites.

Having a single network-wide server would be expensive in terms of network traffic and availability, and having one server per Avalon application would require more inter-server communication. Since we expect most transactions to touch objects on the same site, this arrangement minimizes both remote and local traffic.

Thus the class function definitions, which are compiled into each Avalon program that uses trans_id objects, are little more than stubs for calling the TidServer. The following sections describe the organization and operation of the TidServer.

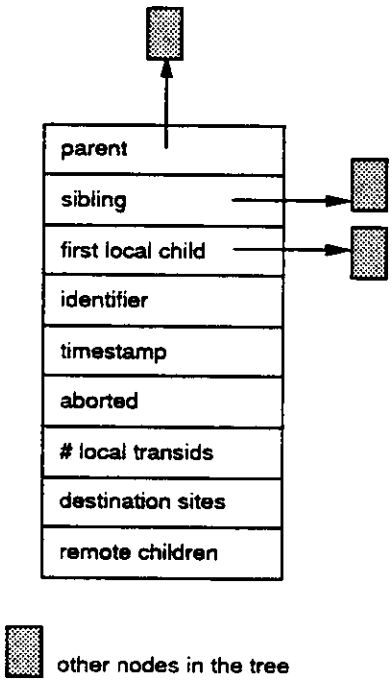
Terminology

At a particular site S, a *local transaction* is one that began on S, and a *remote transaction* is one that began elsewhere. A *local trans_id* is one whose creation and deletion is controlled by a process on site S, and a *remote trans_id* is one controlled by a remote process. For transmission, the sites sending and receiving a trans_id are respectively called the *source* and *destination*, and the site where the transaction originated is its *home*. In general, local and remote trans_ids may either point to local or remote transactions. Also, any combination of the home, source, and destination sites may in fact be the same site.

3.2. Data Structures

The TidServer at each site S keeps track of all transactions that start locally, using a tree data structure, where each node represents a transaction, and each transaction has links to its children and to its parent. This tree structure with upward links facilitates timestamp comparisons and garbage collection. For fast random lookup, the nodes are actually hashed into a table called the LTable.

Each node is a record with fields and links as shown:



The record is created when the transaction begins. If the transaction commits, the timestamp is written. If the transaction is aborted, the "aborted" field is set to true for that record and that of all its descendants. The space for the record is reclaimed later in garbage collection. The field "destination sites" is a list of sites that may want to read the timestamp of this transaction, and the field "remote children" is a list of children identifiers plus the sites on which they reside.

The TidServer also caches information about remote transactions in a hash table called the RTable. When a local process acquires a trans_id for a remote transaction, (say, by receiving it as an argument in a message) then a record for that transaction is created in the RTable, if one does not already exist.

Each record in the RTable contains (1) the home site of the transaction; (2) a list of source sites, i.e., those sites which have passed to this site a trans_id of this transaction; (3) a list of destination sites, i.e., those sites to which this trans_id has been passsed; and (4) the count of local trans_id's for this transaction.

Note that there is no need to maintain the complete ancestry of the remote transaction. However, the timestamps of its ancestors could be cached (when they become known) but this was not implemented.

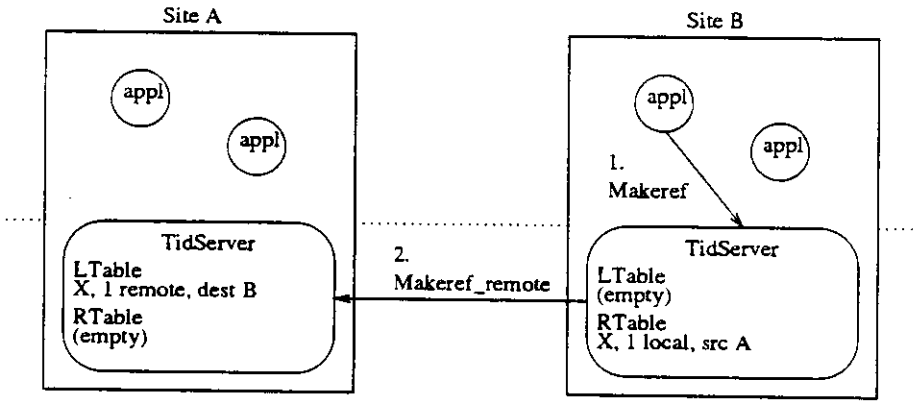
The TidServer provides four groups of operations (see list in Appendix). The first group is called at transaction "events". As transactions begin, commit or abort, a new node is created in the transaction tree, the timestamps are written or records of aborted transactions marked. The second group is called when trans_id's are created, destroyed, or used. Reference counts in the transaction records are incremented or decremented; timestamps are compared. The third group handles the transmission of trans_ids; ensuring that local and remote reference counts are properly maintained. The fourth group performs garbage collection on data within the TidServer. What happens in the first group of operations is fairly straightforward. The other operations are discussed in detail in the next few sections.

3.2.1. Trans_id operations

Creation

When a trans_id object is created at site S, the class constructor calls the *makeref* function in the TidServer at S, which in turn increments the reference count for the transaction. The record for the transaction is in the LTable if the transaction is local and in the RTable if it is remote.

If the transaction is remote, then *makeref* also calls *makeref_remote* operation of the TidServer at the home site of the transaction. The home site records that the site S may look at the timestamp of this transaction, i.e., appending S to the list of destination sites. The diagram below illustrates the calls to and between TidServers when a remote trans_id for transaction X is created at site B. The home site of the transaction X is A.



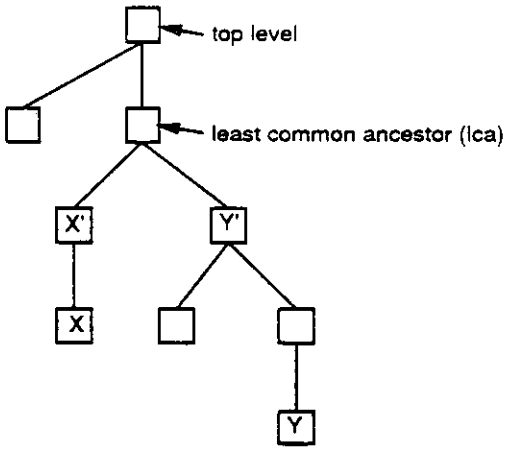
When trans_ids are destroyed, either by being popped from the stack or by an explicit call to `delete`, C++ calls a special destructor operation provided by the class. This in turn calls the TidServer's *breakref*

operation, which decrements the transaction's local count, in the LTable or RTable depending on whether the transaction is local or remote.

Comparison

When two trans_ids are compared, as in the expression t1 < t2, the overloaded < operator invokes the *compare* function of the TidServer. The TidServer searches the tree of transaction records, checks and compares timestamps, and returns the value of the expression.

Conceptually, the timestamp of a transaction is a concatenation of the timestamps of its ancestors, beginning with the top level. Suppose we are evaluating t1 < t2, where t1 is a trans_id for transaction X and t2 is a trans_id for transaction Y. The timestamps of X and Y would be the same from the root down to the least common ancestor.



If the commit time of X' is less than the commit time of Y', or if X' is committed but Y' is not, then X either has or will have an earlier commit timestamp than Y, and the comparison returns true. Otherwise it returns false. Also, if any transaction between X and X' is uncommitted, then the serialization order is not known, and the comparison also returns false.

In the implementation, locating the least common ancestor is potentially expensive, since the ancestors of X and Y could be remote. We believe that the least common ancestor tends to be near the transactions being compared, so rather than looking from the root downwards, we start with the transactions being compared, and then go up one level at the time, checking at each step to see if the next one up is the least common ancestor. If any one of the transactions on the hierarchy is remote, we make a remote call to the TidServer at the remote site to obtain its timestamp.

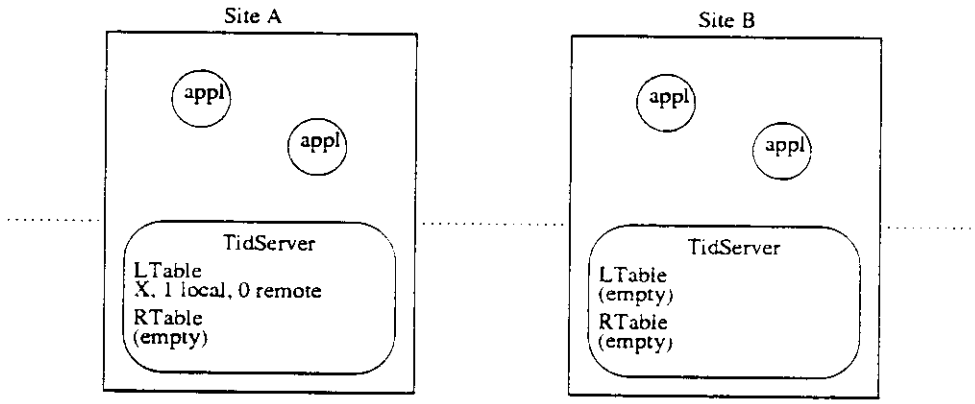
The trans_id operations `descendant` and `done` both call the TidServer operations of the same name, and read the heirachy information. If the heirarchy crosses site boundaries, a call to a remote TidServer will be required.

3.3. Transmission of Trans_ids

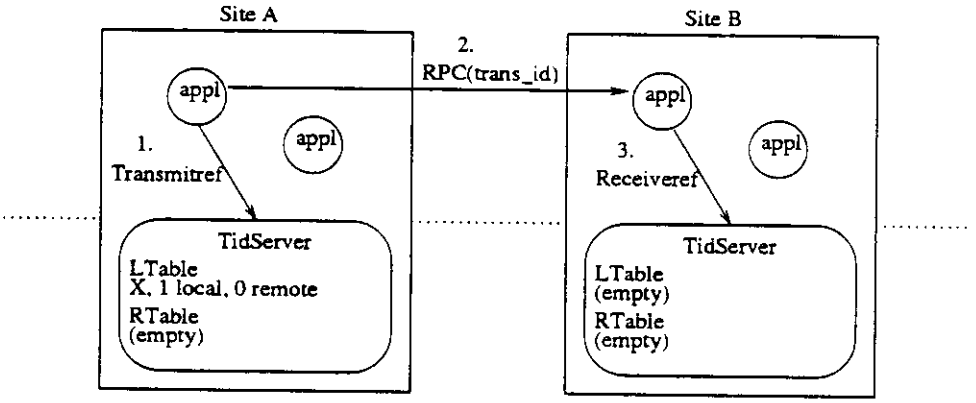
When a trans_id is sent as an argument in an RPC, the Avalon run-time system packages and unpackages the trans_id object appropriately. On both the sender and receiver sites, the TidServers are called to update their state. At the sender, the packaging routine calls *transmitref*. When the trans_id arrives at the destination, the

unpackaging routine calls *receiveref*.

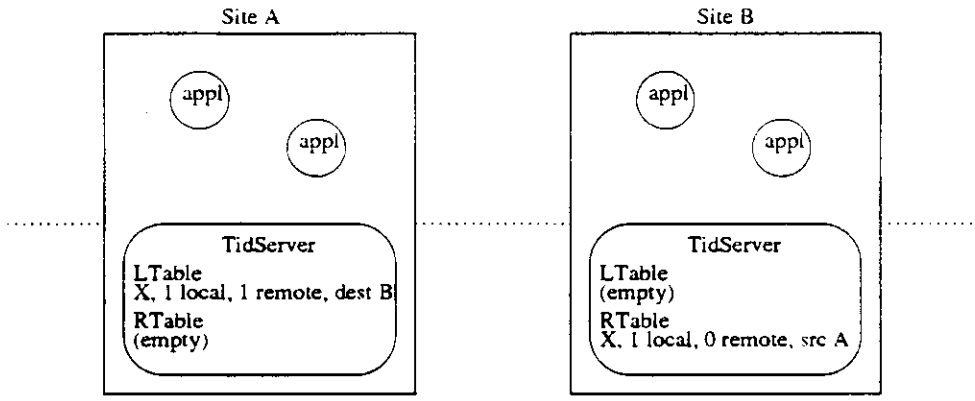
Below we illustrate a typical transmission of a trans_id from site A to B. Suppose transaction X starts on site A and a trans_id object has already been created on site A for transaction X. This is a local trans_id of a local transaction.



Suppose that an RPC is now made to a server on site B, and one of the arguments of this RPC is the trans_id for X:



Transmitref on site A records that there is a remote site, B, which has a trans_id for this transaction. *Receiveref* on B records the existence of transaction X, that there is now one trans_id on B for X, and that this trans_id came from A. If another site C sends a trans_id for X to B, B's TidServer would increment the local count (in its RTable) for X, and append C to the list of source sites.



Since either the source or the destination site may also be the home site of the transaction, the full algorithm must consider four cases as shown below:

	src = home	src = home
dest = home	case 1	case 4
dest = home	case 2	case 3

The example above is case 2, where the source is the home site and the destination is not. The other common case is case 1, where both destination and source are the home site. The complete algorithm is given in the Appendix.

This scheme works correctly in the presence of site crashes. First of all, Camelot guarantees that the TidServer's data survives crashes. Moreover, the transmitter always records the fact that some site wants the timestamp data before sending the trans_id, and the receiver always records that it received the trans_id before making the object available to the application program. In the example above, when the application program can finally request timestamp data using the trans_id, B's TidServer knows where to find it, and A's TidServer has not yet discarded the timestamp data, because it knows of a remote site that wants the data.

However, if *transmitref* has already completed, but the transmission of the trans_id object fails, then we have a dangling reference. The transmitting site thinks that the receiving site is a remote reference, when in fact it is not. This does not lead to incorrect programming results, but it will cause garbage to build up. A routine that checks on remote references to very old timestamp data can correct this problem, but has not yet been implemented.

3.4. Garbage Collection

Reference Counting

The commit timestamp for a transaction cannot be discarded as long as a trans_id object for that transaction (or any of its descendants) exists anywhere in the system. Since trans_ids are potentially long-lived objects, the space used for timestamp data is unbounded without some kind of reference counting scheme. This problem is difficult due to the distributed nature of Avalon applications.

A local and remote trans_id may point to either local or remote transactions, giving rise to four types of reference counts at each site S:

	Local trans_id (Object lives at S)	Remote trans_id (Object lives elsewhere)
Local transaction (began at S)	Local count in LTable	Remote count in LTable
Remote transaction (began elsewhere)	Local count in RTable	Remote count in RTable

These reference counts are managed by several operations in the TidServer as listed here:

Type of Reference Count	Incremented by	Decremented by
Local count in LTable	makeref receiveref	breakref
Remote count in LTable	makeref_remote transmitref	jan_breakremote
Local count in RTable	makeref receiveref	breakref
Remote count in RTable	transmitref	jan_breakremote

Of the six operations, only *jan_breakremote* requires an extra call over the network. (This operation informs a remote server that the caller has no more references to a particular transaction.) All others are calls to the local TidServer. The remote count is the number of sites where remote trans_ids live (i.e. the length of the list of destination sites) rather than the number of trans_ids themselves, so that this number (and thus the number of calls to *jan_breakremote*) is kept low. For example, if there are three (remote) trans_ids on site B pointing to a transaction on site A, then *jan_breakremote* is called only once, when all three trans_ids have been destroyed.

In the previous two sections, we saw how the creation and transmission of trans_ids results in reference counts being incremented. If the programmer is careful to destroy every trans_id object created, both types of local counts will eventually go down to zero by *breakref*. Ensuring that the remote counts go to zero is the responsibility of the garbage collection operations described below.

Triggering garbage collection

The TidServer at each site performs its own garbage collection as follows. Whenever a record for a transaction is created, a counter is checked to see if it has reached `GARBAGE-LIMIT`. If it has not, the counter is incremented; otherwise *janitor* routine is called. The counter is reset to zero if *janitor* executes successfully. Our current implementation sets `GARBAGE-LIMIT` to the size of the hash table. (Recall that

both the LTable and RTable are hashed.) We chose this number because the efficiency of searching the table drops only when the bucket length grows. If the hash function is ideal, then the bucket length will be exactly 1 when we trigger garbage collection.

The Janitor

The *janitor* operation runs as a top-level transaction at each site. It tries to obtain an exclusive lock on all data in the server. If it fails, it returns with an error code, and will be called again soon. Once it acquires the lock, the operation scans sequentially through the hash table, inspecting each record to see if it can be discarded.

A local transaction's record is discardable if the transaction is aborted or if the transaction has no children and no local or remote references (local count is zero, destination list is empty). A remote transaction's record is discardable if there are no local trans_ids for it, and there are no destination sites either. The space for the record is deallocated.

In addition, if a discardable record is in the RTable (i.e., for a remote transaction), a server call (*jan_breakremote*) is queued for the TidServer at the home site of the transaction. Also, if a discardable record is for a local transaction with a remote parent, then *jan_removechild* is queued for the TidServer at the parent's home site.

In each pass, only leaf transactions are garbage collected, since a transaction's timestamp may be needed to compare descendants' trans_ids. This scheme, while simple and efficient, may leave some garbage uncollected. The simplest way to reduce the likelihood of missing discardable records is to make several passes. We believe that most transaction trees are flat (transactions are rarely nested very deeply) so that only a few passes are necessary. Our current implementation makes three.

When all records have been checked, we release the exclusive lock and make the server calls which have been queued. The complete *janitor* routine can be found in the Appendix.

Cycles of Remote Counts

Note that it is possible to produce a cycle of remote reference counts. For instance, site A passes a trans_id (of a transaction, say X) to site B, which then passes it to site C, which in turn passes it to site A. Each of these sites think that another site is interested in the timestamp of transaction X and therefore do not garbage collect the storage for this transaction's record in the RTable.

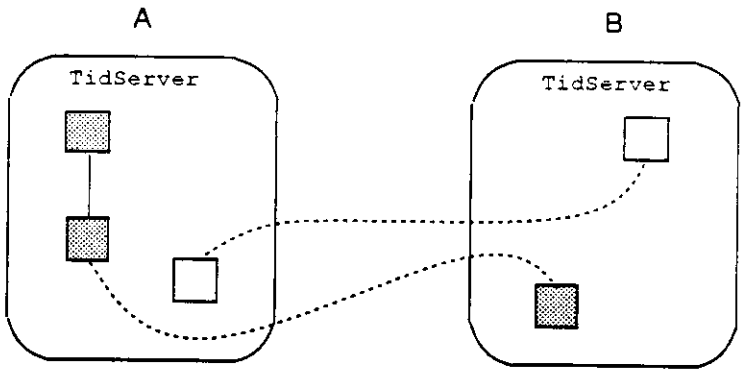
To break cycles, we should look at very old transaction records in the RTable once in a very long while and try to shorten the chain of references. In the example above, we would, at site B, decide that we want to garbage collect the space of transaction X, we would send a messages to A telling it that C is a remote reference to transaction X, and send a message to C telling it to note that A (instead of B) is the source of transaction X. The algorithm has to consider cases where for each transaction, there are multiple source sites and destination sites. Pseudo code for routines to break cycles has been written, but has not been implemented.

3.5. Locking and Deadlock Avoidance

The TidServer is multithreaded, operations can be called simultanously, and each operation runs as a lazy top level transaction [DHW88].

For integrity of the tree structure, write access has to be exclusive. Therefore, there is a single lock for all data in each TidServer which every operation must obtain before proceeding.

Some TidServer operations require calls to other TidServers, which gives rise to the possibility of deadlock. In the diagram below, for example, two sites A and B are both registering a remote child transaction, where the transaction at A has a parent at B and the transaction at B has a parent at A.



Both *register* operations need to lock the data at both A and B, and will probably attempt to obtain these locks in the opposite order, so causing a cyclical wait. Besides register, the following calls may also cause remote calls: *remove* (to propagate transaction abort downwards), *makeref*, *compare*, *done*, *descendant*, and *janitor*.

For *makeref* and *register*, we compare the node addresses of the remote site and the local site, and obtain the lock with the lower address first.

For *compare*, *done* and *descendant*, data is not written, merely read. Locking is required only to ensure that the links are not in a inconsistent state (eg. with dangling pointers). So, at each site, the lock is first acquired, data read, then the lock is dropped immediately.

In *remove*, we mark the local records, then spawn new top level transactions to call remote TidServers to mark remote descendants aborted. These new transactions run as separate threads. *Remove* then commits without waiting for these to complete. The solution for *janitor* is similar. Local records garbage collected, and remote routines i.e. *jan_removechild* and *jan_breakremote* are called as separate top level transactions, and do not have to complete for the local *janitor* operation to commit. In both cases, we avoid waiting for locks at other sites when we're holding the lock at the local site.

The semantics for *remove* and *janitor* may seem a little strange. Since no one checks to see if the remote calls abort, it may be that the local and remote site will be inconsistent. In the case of *remove* operation, if higher levels are marked aborted but this marking is not propagated to the lower level, then a comparison of trans_ids t1 < t2 may report more pessimistic serialization orders. *Done* will not report wrong answers since the marking of aborted transactions (in *remove*) begins at the top and propagates downward. However, the TidServer loses memory space gradually, since the lower levels of records might not be garbage collected. In the case of the *janitor* operation, the situation is similar. No incorrect results are ever reported, but the TidServer may lose memory space slowly, if the remote top level transactions fail. A scheme to examine very old transaction records can remedy this, but has not been implemented.

Much of this complication would go away if all records of a transaction family resided in the same site, but we believe that transaction families exhibit locality, i.e., if a transaction at site A had a remote subtransaction at site B, then furthur descendants would be also be at site B, so that to maintain all timestamp information at site A would require many more remote server calls from B to A.

4. Performance

We first measured the costs of the local trans_id class operations as well some of the underlying Camelot primitives. Then, since application programs that need high degrees of concurrency may benefit from highly concurrent queues, we next examined the performance of our implementation of a FIFO queue which uses trans_ids.

4.1. Cost of Primitives

We used an IBM RT/APC with 12 Mb of memory, a Sun 3/60 with 16 Mb, a μVax-3 with 16 Mb, and a Decstation 5000 (pmax) with 24 Mb, all running Camelot version 83. In measuring, we repeated each operation between 200 to 1000 times, and divided the total time taken by the number of repetitions. All numbers shown are in milliseconds. Note: t, t1 and t2 are objects of class trans_id.

Operation	IBM RT	Sun3	μVax	Pmax
Avalon primitives				
Transaction Begin and Commit with trans_id support	56	73	96	28
t = new trans_id()	10	10	13	4.1
delete t	8	8.5	11	3.3
t1 < t2	11	12	16	5.6
descendant (t1, t2)	11	11	16	5.3
Camelot primitives				
Transaction Begin and Commit	6.0	6.5	8.7	2.4
Server Call (null)	9.1	9.7	13	4.6
Server Call (write)	19	21	31	9.0

The cost of an Avalon transaction with trans_id support is that of a Camelot transaction plus two calls to the TidServer, the first to create the transaction record, the second to write the timestamp, plus computation within the server. Creation, deletion and comparison of trans_ids each costs one call to the TidServer. Note that, with the exception of the Decstation 5000, the performance on all machines is comparable.

Because we were not able to isolate and sufficiently control a network, the remote operations were not measured. However, we made two observations of our environment. First, a "ping" takes anywhere from 15 ms to 200 ms between RT's on a token ring. Second, using the same two machines at a trial run, a complete Avalon transaction which wrote on a remote array server took about 350 ms, and a transaction which wrote on a remote queue server took about 4600 ms.

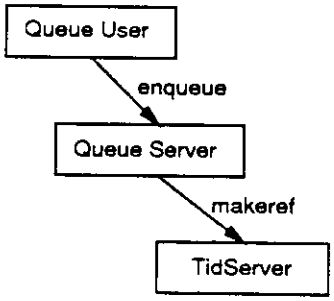
4.2. The Concurrent Queue

We next examined the operation of a concurrent queue. In this Avalon *subatomic* [DHW88] object, the class operations explicitly use the commit order of transactions to obtain concurrency. Enqueues can always proceed, and an element can be dequeued if it is comparable with (committed with respect to) all the other elements in the queue, and if the last dequeuer is committed with respect to the current dequeuer. The concurrent queue is described more fully in [HW87].

We first measured the static costs of the queue. For comparison, we also measured the operation of a simple array, an object of the class *atomic* [DHW88], which uses read and write locks to control access to each element of the array. These measurements were performed on an RT (see previous section).

Operations	time in ms
Enqueue	212
Dequeue	276
Transaction(Enq,Deq)	701
Transaction(Enq,Deq,abort)	937
Write array element	28
Read array element	10
Transaction(Write,Read)	179
Transaction(Write,Read,abort)	412

An enqueue operation involves two levels of server calls. The enqueuer first calls the queue server, which in turn calls the TidServer's *makeref* operation to obtain a trans_id object to associate with the element to be enqueued.



Within the queue server, short term locks are used (in the *when* construct) to start a subtransaction and modify the queue data, thus avoiding the conventional read/write locking. Therefore the cost for an enqueue operation is that of a server call, a write-(sub)transaction at the queue server, plus calls to the TidServer to create a trans_id object. In addition, the enqueue operation also attempts to place the elements in commit order, and so calls the TidServer's compare operation. In the worse case, as many comparisons as elements tentatively queued are needed. Each comparison costs a call to the TidServer, unless the comparison is between elements whose enqueuers are operations within the same transaction, in which case the order is already known.

Similarly, a dequeuer first calls the queue server, which in turn calls the TidServer, obtaining a dequeuer's trans_id to associate with the object. More calls are made to the TidServer to compare the commit timestamps

of the elements in the queue, since the element to be dequeued is the one with the earliest timestamp. Also the current dequeuer has to have a later timestamp than the previous dequeuer. Again, in the worst case, as many comparisons as elements in the queue are needed. Thus, the cost of a dequeue operation is that of a server call, a write-(sub)transaction, plus multiple calls to the TidServer.

Part of the overhead of the queue is the time spent in the *commit procedure* and the *abort procedure*. These procedures are called whenever a transaction commits or aborts, respectively. The commit procedure checks to see if the last dequeuer is a descendant (using the trans_id class operation) of the committing transaction. If it is, then the pool of tentatively dequeued elements can be discarded. This descendant operation costs one call to the TidServer.

In the abort procedure, elements which have been tentatively dequeued by descendants of the aborting transaction are put back into the queue. Thus the procedure costs as many calls to the TidServer as there are active dequeuers.

It is difficult to measure in isolatation the cost of a single commit or abort procedure, so the table above shows the costs of an Avalon transaction containing an enqueue followed by a dequeue operation, and then either a commit or abort. The cost of a Read and Write in the array object is basically the cost of a server call (to the array server) with a read (or write) operation, plus the cost of obtaining a lock. In the absence of contention, the cost of obtaining the lock is negligible compared to the cost of the server call.

In the next table, we show how the cost of a transaction using the queue depends on the number of other elements tentatively enqueued. The transaction performs first an enqueue operation, then a dequeue operation and then commits.

Transaction(Enqueue, Dequeue)				
Number of tentatively queued elements	2	4	8	16
Time in ms	930	1091	1491	2155

From the two tables above, we see that the overheads in our implementation of the subatomic queue are quite high, and in addition, that the overhead increases with the number of elements in the queue.

Given these overheads, we should consider using this subatomic queue only when an application requires high concurrency and/or when the duration of the transaction is long.

Consider using an (initially empty) array in producer-consumer fashion. If we use read and write locks, writers cannot be concurrent. With a subatomic queue however, all writers (enqueuers) can proceed, although readers (dequeuers) have to wait until the enqueuers commit with respect to each other. So we expect this concurrent queue to have an advantage when there are many enqueuers and few dequeuers. We also expect an advantage over read/write locking when the write transactions are long, since it implies that the writer will hold an exclusive lock for a long time, thus stalling other transactions.

In our experiment (see code fragment below), we used one reader, and varied the number of writers, n (2, 4 and 8). A writer (or enqueuer) is a transaction that first writes (or enqueues), performs the independent operations, and then commits. The reader (or dequeuer) is a transaction that reads (or dequeues) and then commits. We varied the length of the writer (enqueuing) transaction by varying the time taken in "independent operations". These are operations that touch objects without any contention with other coarms (transactions in the costart set). All transactions (the coarms) of a the costart statement are executed concurrently [WHC*89].

```
costart {
    transaction(write/enqueue;...independent operations...); // 1
    :
    :
    transaction(write/enqueue;...independent operations...); // n
    transaction(read/dequeue);
}
```

In the table below, we show the execution time for the array and queue implementations of the costart, with potential concurrency increasing as we go across, and length of each write/enqueuing transaction increasing as we go downwards. We used a simple counting loop to simulate the independent operations. Execution of the loop took 170 ms, therefore the values in the leftmost column of the table below are multiples of 170 ms. So the entry marked with an asterix (*), for example, is the time taken for two enqueuing transactions and one dequeuing transaction, where each enqueuing transaction also performs 1020 ms worth of other operations, and where all transactions try to proceed concurrently. Note that these numbers were obtained by simulating the concurrent transactions on a uniprocessor (an RT).

Other operations	2 writers		4 writers		8 writers	
(in multiples of 170 ms)	Array	Queue	Array	Queue	Array	Queue
680	1625	-	3139	-	-	-
850	1925	2318	3749	4305	7608	9625
1020	2362	2513*	4491	4335	9050	9143
1190	2542	2881	-	-	10044	9727
1360	2895	2928	-	-	-	-
1530	-	3088	-	-	-	-

With two writers, a write transaction should be about 1190 ms or more to justify the cost of the queue. We see that as the number of writers increases from 2 to 4, the length of the transaction need not be so long to compensate for the overheads. As we go from 4 to 8 writers, even longer transactions are required since queue operations are more expensive when there are more active enqueuers.

Our implementation of Avalon constructs and objects freely used what were assumed to be cheap primitives: transactions, recoverable memory and server calls. An effort to tune or optimize for performance would have to consider these costs more seriously. However, our the implementation is neither tuned nor optimized. Therefore, the reader should not take our measurements as indicative of worse-case or average-case performance. Rather they represent data points useful for comparison purposes in future work.

5. Final Remarks

This paper describes our implementation of `trans_ids`, a new programming language construct for implementing atomic data types. We believe this construct raises interesting issues in several areas. As a programming language construct, trans_ids present a simple but powerful interface to the programmer, and we have worked out a number of example data types that rely on trans_ids to achieve a high level of concurrency [DHW88]. As a concurrency control mechanism, it can be shown formally that commit-timestamp serialization permits strictly more concurrency than certain conventional techniques based on strict two-phase locking [HW88].

This paper is concerned with a third area of interest: implementation. This is the first implementation of a general mechanism for commit-timestamp serialization. The goals of the implementation were to minimize message traffic and keep the volume of long-lived data to a manageable level. We faced several interesting problems such as the placement (across processes and sites) of distributed timestamp information, the management of remote references, which types of cross boundary links to maintain, and the garbage collection of these data structures with minimal disruption. We found that maintaining timestamp in recoverable storage means using a Camelot server, which results in an overhead of a server call (and thus a context switch) for any update or read. Currently our implementation of distributed garbage collection is not perfect; memory may leak when the network is unreliable, and cycles may occur.

Therefore, to really assess our decisions in placement we need to profile the varied use of trans_ids, especially in distributed applications. We also need to monitor the reliability of the network. In an extremely unreliable network, the imperfect garbage collection may cause the TidServer to lose memory in large quantities, so it might be necessary to stop the whole system to garbage collect once in a while.

In conclusion, we hope to see an increasing use of trans_ids and timestamp ordering for concurrency control, particularly in applications that can benefit from high levels of concurrency.

Acknowledgments

We thank Alfred Spector and the rest of the Camelot group for providing the basis for the Avalon work. We thank David Detlefs, Stewart Clamen, Richard Lerner, and Karen Kietzke for their contributions to Avalon/C++ and letting us run timing experiments on their machines.

References

[BGL81] P.A. Bernstein, N. Goodman, and M.Y. Lai. Two-part proof schema for database concurrency control. In *Proc. Fifth Berkeley Workshop on Distributed Data Management and Computer networks*, February 1981.

[CD88] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-154, CMU, 1988.

[DHW88] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, December 1988.

[Her85] M.P. Herlihy. Comparing how atomicity mechanisms support replication. In *Fourth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, August 1985. Reprinted in Operating Systems Review 20(3), and CMU-CS-85-123.

[HW87] Maurice P. Herliy and Jeannette M. Wing. Reasoning about atomic objects. Technical Report CMU-CS-87-176, CMU, 1987.

[HW88] M.P. Herlihy and W.E. Weihl. Hybrid concurrency control for abstract data types. In *Seventh ACM-SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, pages 201–210, March 1988.

[Kor83] H.F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1), January 1983.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LS83] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Language and Systems*, 5(3):382–404, July 1983.

[McK84] M.S. McKendry. Clouds: A fault-tolerant distributed operating system. *IEEE Tech. Com. Distributed Processing Newsletter*, 2(6), June 1984.

[Ree83] D.P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[SBD+86] A.Z. Spector, J.J. Bloch, D.S. Daniels, R.P. Draves, D. Duchamp, J.L. Eppinger, S.G. Menees, and D.S. Thompson. The Camelot project. *Database Engineering*, 9(4), December 1986. Also available as Technical Report CMU-CS-86-166, Carnegie Mellon University, November 1986.

[SS84] P. Schwarz and A. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[Wei84a] W.E. Weihl. *Specification and implementation of atomic data types*. PhD thesis, MIT, 1984. Available as Technical Report MIT/LCS/TR-314.

[Wei84b] W.E. Weihl. *Specification and implementation of atomic data types*. PhD thesis, MIT, 1984. Available as Technical Report MIT/LCS/TR-314.

[WHC+89] J. M. Wing, M. Herlihy, S. Clamen, D. Detlefs, K. Kietzke, R. Lerner, and S. Ling. The Avalon/C++ programming language. Technical Report CMU-CS-88-209R-2, CMU, 1989.

[WL85] W. E. Weihl and B. Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Language and Systems*, 7(2):244–269, April 1985.

Appendix

TidServer Interface

1. Transaction management

   register (X : transaction-name, Xparent : transaction-name)
   Creates a record for transaction X. Called when X begins.

   child (X : transaction-name, Xchild : transaction-name, Xlevel : integer)
   Updates a local transaction's record to show that a remote child exists. Called when a local transaction becomes the parent of a remote subtransaction.

   stamp (X : transaction-name, Xts : timestamp)
   Records Xts as the commit time of transaction X. Called when X commits.

   remove (X : transaction-name)
   Recursively sets the 'aborted' flag on the record of X and on the records of X's descendants. This makes the records available for garbage collection. Called when X is aborted.

2. Trans_id class operations

   makeref (X : transaction-name)
   Called from the trans_id constructor. Increments the reference count of local trans_id's for transaction X.

   makeref_remote (X : transaction-name, S : site-name)
   Called from the trans_id constructor. Increments the reference count of remote trans_id's transaction X.

   breakref (X : transaction-name)
   Called by the trans_id destructor. Decrements the count of trans_id's that refer to transaction X.

   copyref (X : transaction-name, Y : transaction-name)
   Called when a trans_id is assigned to another. Decrements the reference count of transaction X, and increments the reference count of transaction Y.

   compare (X : transaction-name, Y : transaction-name, result : boolean)
   Compares the timestamps of X and Y. Returns true if X serializes before Y, otherwise returns false. Called by the trans_id operators < and >.

   yield (X : transaction-name, Xparent : transaction-name, Xdepth : integer, Xts : timestamp, Xvalid : boolean)
   Returns the timestamp, the depth and name of the parent of transaction X. requested by one TidServer of another. Also resets 'valid' if the transaction has aborted. Called by one TidServer to another.

   done (X : transaction-name), **descendant**(X : transaction-name, Y : transaction-name)
   These are called by the trans_id member functions of the same name.

3. Trans_id transmission. Transmitref and receiveref are called by the routines that package and unpackage arguments in a remote procedure call.

   transmitref (X : transaction-name, Dest : site-name)
   Writes in the record of transaction X that a trans_id is to be transmitted to site Dest.

   receiveref (X : transaction-name, Src : site-name)
   Writes in the record of transaction X that a trans_id has been received from site Src.

Algorithms for Transmission of `trans_id`

```
Case 1: Source = Home, Destination = Home
    Transmitref (X : transaction-name, dest : destination-site-name)
        Do nothing

    Receiveref (X : transaction-name, src : source-site-name)
        Get record of X from LTable
        Increment local count

Case 2: Source = Home, Destination = Home
    Transmitref (X : transaction-name, dest : destination-site-name)
        Get record of X from LTable
        If dest is not in dest-list
            Append dest on dest-list
            Increment remote count

    Receiveref (X : transaction-name, src : source-site-name)
        If there is a record for X in RTable
            If src is not on src-list
                Increment local count
                Put src on src-list
            else Do nothing
        else
            Mmake a record for X in RTable
            Put src on src-list
            Set local = 1

Case 3: Source = Home, Destination ≠ Home
    Transmitref (X : transaction-name, dest : destination-site-name)
        Get record of X from RTable
            If dest isn't in dest-list,
                Put dest on dest-list
                Increase remote count

    Receiveref (X : transaction-name, src : source-site-name)
        If there is a record for X in RTable
            If src is not on src-list
                Put src on src-list
                Increment local count
            else Do nothing
        else
            Make an record for X in RTable
            Put src on src-list
            Set local = 1

Case 4: Source ≠ Home, Destination = Home
    Transmitref (X : transaction-name, dest : destination-site-name)
        Do nothing

    Receiveref (X : transaction-name, src : source-site-name)
        Get record of X from LTable
        Increase local count
```

Algorithms for Garbage Collection

```
Janitor() :
    Get exclusive lock on both tables
    For each local transaction X (a record in the LTable)
        If X is active, do nothing.

        If X is aborted
            remove the record, free storage
            for visit all child recursively
        If X is committed
            If X has no children and there are no local
                or remote trans_id's for X
            then
                Remove X as child of parent(X)
                If the parent(X) is local,
                then remove self from sibling list of parents
                else
                    queue a server call to home site of parent(X)
                        (jan_removechild(parent(X), X))
                Remove record of X

    For each remote transaction X (a record in the RTable):
        If both local count and remote count are zero
        then
            For each node on the list of source sites
                queue a server call to the node to strike out this-site as a destination
                    (jan_breakremote(X, this-site))
                Remove record of X
    Release exclusive lock
    Make the server calls that have been queued


Jan_removechild (P : transaction-name, C : transaction-name) :
    Check that P is remote
    Get record of P from LTable
    Remove C from P's list of children

Jan_breakremote (X : transaction-name, N : site-name) :
    Check that N is different from the this-site
    Get record of X from LTable
    Decrement the count of remote sites
    Remove the node_id from the destination list
```