CONCURRENCY CONTROLS FOR GLOBAL PROCEDURES IN FEDERATED DATABASE SYSTEMS

Rafael Alonso
Hector Garcia-Molina
Kenneth Salem

CS-TR-099-87

June 1987

CONCURRENCY CONTROL AND RECOVERY FOR GLOBAL PROCEDURES IN FEDERATED DATABASE SYSTEMS

Rafael Alonso
Hector Garcia-Molina
Kenneth Salem

Department of Computer Science
Princeton University
Princeton N.J. 08544

ABSTRACT

A federated database system is a collection of autonomous databases cooperating for mutual benefit. Global procedures can access several databases, but controlling concurrent database accesses by them is problematic. In particular, each database has its own (possibly different) concurrency control mechanism, and it must remain independent from the global controls. Furthermore, it may be difficult for the global controls to know exactly what granules (e.g., records or pages) are touched by each database access. In this paper we discuss the concurrency control problem for federated database systems, and suggest two mechanisms that may satisfy the requirements.

CONCURRENCY CONTROL AND RECOVERY FOR GLOBAL PROCEDURES IN FEDERATED DATABASE SYSTEMS

Rafael Alonso
Hector Garcia-Molina
Kenneth Salem

Department of Computer Science
Princeton University
Princeton N.J. 08544

1. Introduction

As the name suggests, a federated database is actually a collection of databases cooperating for mutual benefit. The particular quality that distinguishes federated databases from other distributed databases is the degree of autonomy of the members of the federation. For example, it is assumed that federation members may not be forced to perform activities for other members, that each determines which parts of its local database will be shared, and how they will be shared, that each maintains its own database schema, and that they may withdraw from the federation if they so choose [Heim85a].

The purpose of the federation is to increase the capabilities of its members, i.e., to permit transactions that deal with non-local data. The federation permits members controlled access to foreign databases. We will use a simple model of inter-database interaction: all interaction will be accomplished through the use of *global procedures.* A global procedure is initiated at some federation member (its home, or local, node), and can request other members to execute procedures (usually transactions) on its behalf. Other methods of cooperation are possible, e.g., a federation member may agree to "export" a particular portion of its local data to another member [Heim85a]. However, for this discussion we will consider only global procedures.

In general, a number of global procedures will be in progress simultaneously within the federation. The question we will address in this paper is how concurrent global procedures should be managed. Specifically, we are interested in concurrency control and recovery for global procedures. However, before we address these issues we first present simple models of the federation and its members which we can then use in our discussions.

Each member, or node, of the federation consists of a transaction and database manager. (We will refer to the local transaction/database manager of the ith member as LTM_i.) An *LTM* presents a transactional interface to the local database to users at its local node. It also presents a transactional interface (which may or may not be the same as the local interface) to a *global procedure manager* (*GPM*) residing at the local node.

The *GPM* is the interface between the local node and the rest of the federation. The *GPM*s of federation members are connected by some type of communications mechanism. A *GPM* receives requests for service from the *GPM*s of other members of the federation, translates those requests into a form that is palatable to its *LTM*, and forwards the requests to it.

In addition to handling requests from other nodes, each *GPM* provides a federation interface to applications at its local node. The local interface provided by GPM_i is a set of global procedures $P_{i1},..,P_{in_i}$, i.e., these procedures are available to local users at node i. This interface is the mechanism under which global procedures are initiated.

Each global procedure consists of a (possibly ordered) set of requests for service at other nodes. As described above, each request is translated by the target *GPM* into a transaction for its *LTM*, and the results are returned to the *GPM* making the request. The block diagram in Figure 1 describes an n-member federated system of the type we have just described.

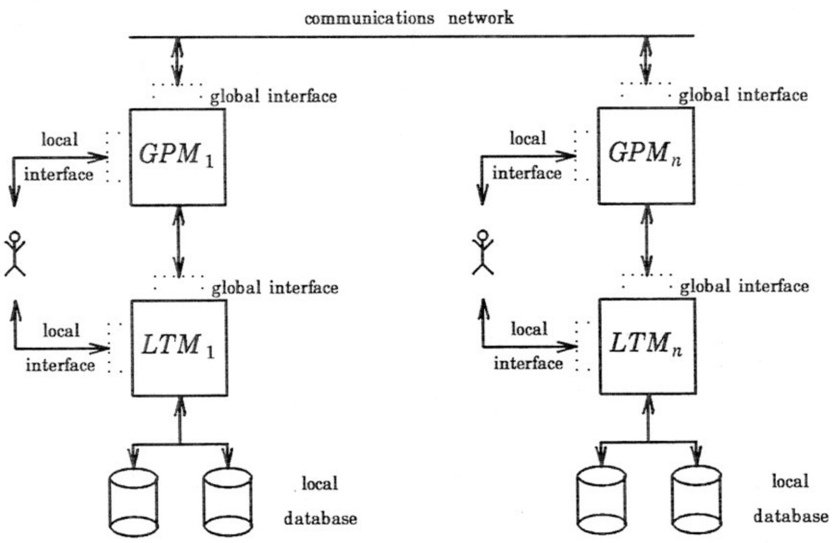

Figure 1 - Federated System Block Diagram

To reflect the autonomy of federation members, we make several assumptions in the context of the federation model:

- A *GPM* has no knowledge of the structure and organization of the local database, nor of the implementation of its *LTM*. The *GPM* knows only of the transactions available to it through the *LTM*'s global interface.
- Support for global procedures is implemented entirely within the *GPM*s at each node; no modifications to the *LTM*s are required.
- The *only* interaction between a *GPM* and its *LTM* is through that *LTM*'s global, transactional interface.

These assumptions have a number of important ramifications, some of which we discuss next:

- If a *GPM* submits a request for an update transaction to its *LTM* on behalf of a global procedure, it is not possible to lock or shield the updates once the local transaction has completed. The *GPM* may shield the updates from other *global procedures* by refusing to submit further requests to the *LTM* on behalf of those procedures until the first global procedure is finished. However, there is no way for the *GPM* to keep transactions submitted through the *LTM*'s local interface from seeing the updates. Note that if a *GPM* submits more than one transaction to its *LTM* on behalf of a single global procedure then there is no way to guarantee that that global procedure can be serialized with transactions submitted locally to the *LTM*.
- If concurrency controls are to be implemented by the *GPM*s for global procedures, then dependencies among global procedures must be maintained with a data granularity no smaller than an entire local database or federation node. In other words, if two global procedures have transactions submitted on their behalf to the same *LTM*, those transactions (and thus the global procedures they are a part of) must be assumed, from the point of view of the *GPM*s, to

conflict. Thus the lockable entities in a global concurrency control mechanism will be the nodes of the federation themselves.

- Once a transaction has been run by an *LTM* on behalf of some global procedure, updates made by that transaction cannot be rolled-back, or undone, by the *GPM*. The only recourse of the *GPM* in this case is to request the execution of a *compensating transaction* by its *LTM*. Thus a global procedure cannot be truly atomic in the transactional sense. We will discuss this issue further in Section 4.
- Local transactions submitted through the *LTM*'s local interface are not affected by global concurrency controls. Although locally and globally submitted transactions may conflict within the *LTM*, the *LTM* can deal with this conflict as it sees fit, e.g., it may abort either type of transaction at any time.

In the rest of this paper we will briefly survey two approaches to global concurrency control that operate within this framework, and then discuss recovery of global procedures. Given our space limitations, our objective will be to intuitively explain these approaches and not to provide details. The details of each of the concurrency controls, together with discussions of the recovery issues, are given in separate papers.

Altruistic locking [Sale87a] is an extension of two-phase locking. Like two-phase locking, altruistic locking results in serializable schedules. A global locking protocol, such as two-phase or altruistic locking, plus local concurrency controls that guarantee serializable schedules, can ensure the serializable execution of global procedures.[†] Altruistic locking can make this guarantee while allowing potentially greater concurrency than two-phase locking.

For many applications it is not necessary to serialize global procedures. A *saga* [Garc87a] is a procedure that can be broken up into a collection of smaller transactions which can be interleaved in any way with other transactions. The transactions in a saga are related to each other and should be executed as a (non-atomic) unit. Since global procedures in a federated database are collections of service requests (i.e., local transactions), they are natural as sagas if the application semantics do not require serial consistency for the entire procedure.

To make the discussion of sagas and altruistic locking clearer, we will consider the database facilities of a hypothetical car rental company which has a number of independently owned outlets in several cities across two states. Each outlet has its own local database which records reservations and the state of the local fleet. The outlets' local databases are joined into a single federated system, shown in Figure 2. In the figure, each node is identified by a single capital letter.

2. Sagas

Imagine a car rental customer making a business trip which takes him to cities A, B, C, and then D. At each city, he wishes to reserve a car from the local outlet for one week. P_{A1} is a global procedure (available at node A) which implements this by requesting reservations from each of the nodes. For the sake of this discussion, we will assume that P_{A1} makes requests of A, B, C and D in that order.

It is probably not necessary for P_{A1} to hold on to all of its resources until it completes. For instance, once P_{A1} successfully gets the reservation at node A, it could immediately allow other global procedures to make requests at A. However, we do not wish P_{A1} to be simply a collection of independent requests (local transactions). P_{A1} is a unit which should be successfully completed or not done at all. For example, if the reservation at C cannot be obtained then it is likely that the previously obtained reservations at A and B will have to be changed. Thus it seems natural to

[†] Global procedures can be serialized with local transactions only if they submit at most one transaction request to each node.

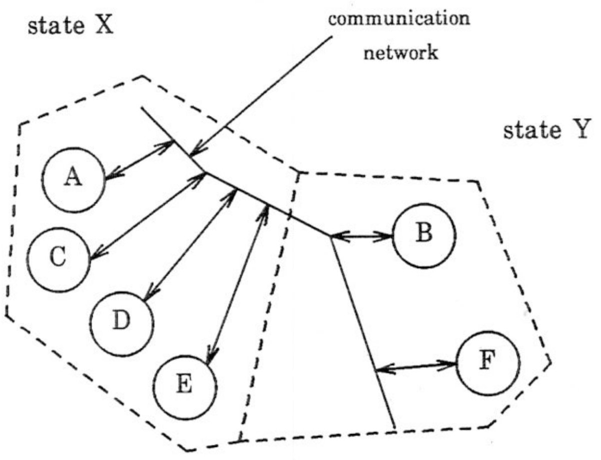

Figure 2 - Example Federated Database System

treat P_{A1} as a saga.

To amend partial executions of a saga, each transaction in the saga should be provided with a *compensating transaction.* A compensating transaction undoes, from a semantic point of view, its associated transaction. In our example, if a transaction in P_{A1} reserves a car at B, then its compensating transaction cancels the reservation at B. The system makes a *semantic atomicity* guarantee for the saga: if the saga is aborted, any transactions in the saga that have already completed will be compensated for. We discuss compensation further in Section 4.

By running P_{A1} as a saga rather than ensuring its total serializability with other global procedures, the system can obtain some potentially substantial performance benefits. In particular, it is not necessary to maintain any global locks to synchronize a (global) saga. The home node of the saga simply submits its transaction requests to the proper federation nodes.

3. Altruistic Locking

Sagas are not always the appropriate abstraction to use for global procedures. In some cases, it really is necessary to synchronize a procedure with other global procedures. For example, consider a procedure P_{A2} used to determine the total number of cars available at the outlets in state X. P_{A2} requests that nodes A, C, D, and E (i.e., all of the nodes in state X) run local transactions to determine the number of cars available at the local outlet. To guarantee an accurate count, P_{A2} should be synchronized with other global transactions that modify inventories of available cars. P_{A3}, a procedure that records the loan of cars from node A to node C, is an example of the type of global transaction that, if not serialized, could destroy the accuracy of P_{A2}'s count.

This problem can be avoided by using global two-phase locking. Before requesting service from a node, that node must be locked and the lock must be held until the end of the global procedure. A locked node cannot service requests for any other global procedures. (Recall that the locking granularity at the global level is the node. Locking a smaller granule (e.g., a relation) would require that the GPM have some knowledge of the structure of its local database.) However, the performance problems of global two-phase locking may be substantial. If P_{A2} manages to lock node A first, P_{A3} would have to wait for P_{A2} to query all four nodes in state X before the lock on A is released and it can attempt to continue. The situation gets worse as global procedures get larger and access more nodes.

Altruistic locking is a locking protocol that may ease this type of performance problem. Like two-phase locking, altruistic locking ensures that execution schedules are serializable. However, it provides a mechanism for global procedures to release locks before they finish, possibly freeing waiting global procedures to acquire the lock and continue processing.

Applied to global locking, the altruistic locking protocol works as follows. As with two-phase locking, a global procedure must lock a node before it can request work from that node. Once the global procedure's request has been processed, and *if the global procedure will request no further work from that node*, it can *release* its lock on the node. Releasing a lock is a special operation, peculiar to altruistic locking. Releasing a lock is like conditionally unlocking it. Other global procedures waiting to lock the released node may be able to do so, but only if they are able to abide by certain restrictions. Note that a global procedure is free to continue locking new nodes after it has released locks, i.e., locks and releases need not be two-phase.

The set of nodes that have been released by a global procedure constitute the *wake* of that procedure. If P_{A3} locks a node in P_{A2}'s wake, we say that P_{A3} is in the wake of P_{A2}. Under the simplest altruistic locking protocol each global procedure concurrent with P_{A2} must remain completely inside the wake of P_{A2}, or completely outside, until P_{A2} has finished. For example, P_{A3} must lock only nodes released by P_{A2}, or it must not lock any nodes released by P_{A2}.

While this may seem somewhat restrictive, consider our previous example in which P_{A3} was forced to wait (in the worst case) for P_{A2} to make requests of all four nodes in X. If altruistic locking were used, P_{A2} could release each node as soon as its local query at that node was successfully completed. Thus P_{A3} could lock A as soon as P_{A2} moved on to C, and could lock C as soon as P_{A2} moved on to D. One nice feature of altruistic locking is that it is certain to provide at least as much concurrency as two-phase locking, and possibly more. In other words, there is no situation in which a global procedure that would have been permitted to run under two-phase locking would be prohibited from running under the altruistic protocol. Of course, the actual performance advantages of altruistic locking would depend on the resource requirements of global procedures and the cost of global operations (such as requesting services or locks) in a particular application.

A more complicated version of the altruistic protocol permits a global procedure to "straddle" another's wake, i.e., to be partially in and partially out of the wake, in some circumstances. For example, consider a global procedure P_{A4} that records the loan of cars from A in state X to B in state Y. Imagine that P_{A2} (which counts the cars available in X) and P_{A4} run concurrently and that P_{A2} manages to lock A before P_{A4}. Under the simple protocol just described, once P_{A2} releases A, P_{A4} can access that node. However, B is outside the wake of P_{A2}. Since P_{A4} is already in the wake of P_{A2} (because it locked A), it will have to wait until P_{A2} finishes before it can lock B.

This is unfortunate because we know that P_{A4} could have been serialized after P_{A2} even if it locks B without waiting for P_{A2} to finish. We know this because P_{A2} never accesses B. The more complicated altruistic locking protocol can take advantage of information about which nodes a global procedure will visit during its lifetime. It lets global procedures make use of another special concurrency control operation called the *mark*. A global procedure marks a node to indicate that it will access that node sometime in the future.

Marking is an option, not a requirement. By marking nodes a global procedure is assisting other global procedures by informing them of its intended behavior. A global procedure that does mark must abide by several restrictions in order for the marks to be useful. A global procedure that chooses to mark may not lock a node without marking it first, and must stop marking nodes once it has issued a release operation.

As we have already hinted, a global procedure running in the wake of a marking global procedure need not always remain within the wake. It may lock a node outside of the wake provided that node has not been marked by the procedure in whose wake it is running. In our example, this means that P_{A4} would be permitted to lock B after locking A. Since P_{A2} never needs access to B, it will not have marked B.

Of course, we cannot permit P_{A4} to simply lock and request a local transaction at B even if P_{A2} hasn't marked it. Imagine a global procedure P_{A5} that records the transfer of cars from B to E. Conceivably, P_{A5} could lock B after P_{A4} was finished and then lock E before P_{A2} got that far. This results in an unserializable schedule among the three global procedures P_{A2}, P_{A4} and P_{A5}.

To avoid this kind of situation, P_{A4} should indicate that any global procedure locking B after P_{A4} must be serialized after P_{A2}. The altruistic protocol prescribes a simple mechanism for accomplishing this. P_{A4} is required to release B *on behalf of* P_{A2} before it can lock B. In other words, P_{A4} never really leaves the wake of P_{A2}. Instead, it expands the wake to include the node that it wishes to access. The protocol also permits a global procedure outside another's wake to enter the wake under similar conditions. Details can be found in [Sale87a].

4. Recovery

Whether global procedures are scheduled serializably or are treated as sagas, some recovery mechanism will be needed. For this reason, the *GPM* at each node must have access to some stable storage mechanism on which it can maintain a record of its activity. Such a log would record the progress of global procedures initiated at the local node, plus the status of requests being executed by the local *GPM* on behalf of other nodes. Of course, requests being executed for a global procedure can fail for reasons other than power losses, e.g., the local transaction spawned by the request might deadlock in the *LTM*.

As we have already observed, true atomicity is not possible for global procedures. However, some useful guarantees can be made in case global procedures run into problems when executing. In general, a global procedure can be recovered *forwards* or *backwards* from the failure of a request. Forward recovery involves retrying the failed request. This is useful for transient failures like deadlocks of local transactions. The decision to retry might be made by the *GPM* at the site of the failure, or by the *GPM* at the node from which the request initiated.

Backward recovery is the abortion of the global procedure, undoing its effects. Backward recovery in a federated system poses special problems because of the autonomy of the federation members. As has already been mentioned, it is not possible, in general, to completely erase the effects of a global procedure. The means of aborting a global procedure is the execution of *compensating transactions* at the nodes visited by the global procedure. A compensating transaction undoes, from the view of the semantics of the local database, the effect of a previous transaction.

Global procedures cannot be said to have a commit point in the same sense as transactions do. Updates made on behalf of global procedures are committed, i.e., made available to others, as soon as the local transaction that implements the update on behalf of the global procedure has committed. However, at some point the initiating *GPM* must decide whether to externalize the results of the global procedure. Thus it is perhaps better to say that there are "externalization dependencies" among global procedures, rather than commit dependencies. Such dependencies might arise if a global procedure releases (or unlocks) a node that has been updated on its behalf.

Depending on the application, it may or may not be desirable to delay the externalization of a global procedure until those procedures on which it depends have been externalized. When a global procedure is aborted, other procedures dependent on it can be aborted as well (using compensating transactions). Note, however, that there is no way to keep locally-submitted local transactions from seeing updates made on behalf of a global procedures and externalizing them. In some applications, maintenance of externalization dependencies may be deemed unnecessary. For example, if a global procedure increments a flight's reservation count by one in an airline reservation system, it may be acceptable to allow other procedures to see the modified count without making them dependent on the reservation procedure.

Whether or not dependencies are maintained, the use of compensating transactions allows the *GPM*'s to make a *semantic atomicity* guarantee [Garc83a] for global procedures: either the procedure will be completely executed or compensation will be requested for the completed parts of a partially executed procedure. Global procedures which use altruistic (or two-phase) locking can have a stronger guarantee if they do not release nodes at which local update transactions have been executed on their behalf, i.e., if they do not allow other global procedures to become dependent on them. (Note that global procedures which use altruistic locking but do not release their updates can still achieve more concurrency than global procedures using two-phase locking.) If P is such a procedure, it can be ensured that other global procedures will not see P's updates unless it has finished or compensation has been requested for the updates.

5. Conclusions

We have looked at two general ways of synchronizing global procedures in a federated database system, using a number of examples. Altruistic locking at the global level guarantees the serializability of global procedures, while allowing more concurrency than global two-phase locking.

If serializable execution of global procedures is not important to the application and the procedures can be broken up into a collection of transactions, then sagas can be used. Sagas provide semantic atomicity but do not serialize the execution of global procedures. Sagas require no global locking and permit more concurrency than mechanisms that treat global procedures as a single unit.

Because global procedures can affect local resources only through a transactional interface, it is not possible to make them completely atomic. However, compensating transactions can be used to provide a semantic atomicity guarantee for global procedures.

References

Garc83a.

Garcia-Molina, Hector, "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems*, vol. 8, no. 2, pp. 186-213, June 1983.

Garc87a.

Garcia-Molina, Hector and Kenneth Salem, "Sagas," *Proc. ACM SIGMOD Annual Conference*, pp. 249-259, San Francisco, CA, May, 1987.

Heim85a.

Heimbigner, Dennis and Dennis McLeod, "A Federated Architecture for Information Management," *ACM Transactions on Office Information Systems*, vol. 3, no. 3, pp. 253-278, July, 1985.

Sale87a.

Salem, Kenneth, Hector Garcia-Molina, and Rafael Alonso, "Altruistic Locking: A Stratagy for Coping with Long-Lived Transactions," CS-TR-087-87, Dept. of Computer Science, Princeton University, April, 1987.