HYDRO: A HETEROGENEOUS DISTRIBUTED DATABASE SYSTEM

William Perrizo[1]
Joseph Rajkumar and Prabhu Ram
Computer Science, Box 5075
North Dakota State University
Fargo ND 58105

ABSTRACT

In this paper we show how global serializability and atomic commit can be attained in a Heterogeneous Distributed Database Management System in which each local DBMS is assumed to be an off-the-shelf, binary-licensed commercial product providing the IBM SAA Common Programming Interface ([SAA88]). Our HYDRO system achieves global serializability using a set of objects based on the Request Order Linked List or ROLL object developed in [PER89] and [PER91]. ROLL is based on the general Serialization Graph Test methodology ([BER87]), and provides freedom from idle-wait, deadlock, livelock and restart. Atomic commitment is based on Two-Phase Commit. Two options are offered to achieve the PREPARED state locally. HYDRO-I achieves the PREPARED state by protecting writes during the uncertainty period. HYDRO-II provides more concurrency, but raises the commitment overhead in the absence of a visible PREPARED state offered by the local DBMS.

1. INTRODUCTION

Rapid advancements in communication and networking technology are dramatically changing the way in which data are processed. More and more enterprises are interested in integrating and consolidating their physically dispersed data resources. Accordingly, a level of database management software is needed for access to data from pre-existing database systems located around a computer network (from a variety of database vendors and based on a variety of models). Such systems are called Heterogeneous Distributed Database Management Systems (HDDBMSs) or Multidatabase Systems ([BRE90]).

Ideally, an HDDBMS would provide transparent, efficient global transaction atomicity, correctness, isolation and durability. Atomicity means that global transactions must be all-or-nothing units, just as local transactions are all-or-nothing units of local database work. For correctness we will use global serializability, namely transaction effects are globally equivalent to some serial order of execution. Global serializability is a natural correctness criterion, as it is almost universally used in local database management systems and provides full correctness for multipurpose data management environments. Isolation means that each user is given the illusion of being the sole user of the system. Durability means that the effects of global transactions are never lost once the transaction is committed.

In this paper we describe an HDDBMS, called Heterogeneously Distributed Request Ordering (HYDRO) system, which is being developed at North Dakota State University with the support of IBM ABS in Rochester, MN. HYDRO is intended for heterogeneous distributed database management in a networked environment containing IBM AS/400 Database Management Systems together with other systems. HYDRO uses global serializability as a correctness criterion. In our model, full local autonomy is provided, where full local autonomy is taken to mean that each local DBMS is an off-the-shelf, binary-licensed commercial product which provides local correctness, local recoverability, local atomic commit and some superset of the IBM SAA Common Programming Interface ([SAA88]). At each site, HYDRO has a local server module (LHYDRO) which is customized to take advantage of the local interface provided by the DBMS software at that site. Local database interfaces are assumed to provide transaction definition constructs which are conceptual (not necessarily syntactic) supersets of IBM's SAA Common Programming Interface ([SAA88]). The sole superset feature beyond SAA, considered at this time, is a "PREPARE" statement which, when issued and acknowledged, guarantees to the issuer that neither a COMMIT nor a ROLLBACK statement will be rejected. Some, but certainly not all commercial systems provide a visible PREPARED state in this sense (SYBASE[tm], for instance does). We have developed two local server options. The first local server option, LHYDRO-I, is used at sites where a visible PREPARED is not available. The second local server option, LHYDRO-II, can be used at any site, but at sites where the visible PREPARED state is not

[1] Partially supported by USAFOSR grant F19628-86-K-0019.

available, considerable additional commit overhead results. The two servers are designed to be interoperable.

Many advanced distributed database applications are not well accommodated by the standard recovery using full atomic commitment. Currently, research is being done on transaction management in which global atomic commitment is relaxed ([TUL90], [KOR90]). By relaxing the atomicity of commit, long duration and nested transaction can be better accommodated. However, in the presence of failure, compensating transactions must then be used to produce acceptable database states through semantic UNDO procedures ([KOR90]). These procedures require high levels of local control which may not be possible in the presence of full local autonomy. Also, many standard database applications still require full atomic commit for acceptably correct execution. Full atomic commitment in HDDBMSs is not a solved problem. As yet there have been very few proposed methods advanced for full atomic commitment in an HDDBMSs ([BRE90]). Our present work in HYDRO concentrates on providing full atomic commitment, in both HYDRO-I and HYDRO-II. In the future, alternatives will be investigated to add non-atomic commitment options.

In HYDRO, local transactions are routed through the local HYDRO server module prior to being submitted to the autonomous local DBMS. In most operating system environments this can be accomplished by giving the local HYDRO server the same name, look and feel as the local DBMS (e.g., a shell on UNIX[tm] systems would allow the placement of the server code higher in the execution search path than the local DBMS).

In this paper we show how global serializability and atomic commit can be attained in HDDBMSs through HYDRO and describe current and future activities. In Section 2, we review our basic concurrency control object, the Request Order Linked List object (ROLL, see [PER91] for further details). In section 3, we define our transaction model and define two versions of HYDRO using the ROLL object. In Section 4, we describe how HYDRO-I and HYDRO-II achieve atomic commitment. In section 5, we give the conclusions and discuss future work.

2. REQUEST ORDERED LINKED LIST CONCURRENCY CONTROL

Most DBMSs use one of the two most common approaches to concurrency control namely, a waiting policy or a restart policy. Locking methods are waiting policies which use data locking to provide concurrency control. They can cause unnecessary periods of idle waiting and can also cause deadlocks in which two or more transactions are involved in a cyclic wait. To avoid deadlocks, some form of prevention or avoidance technique is required, adding complexity and delays to the system ([ESW76], [BER87]). Timestamp based protocols use transaction timestamps and restarting to eliminate access orders which will result in errors. It is possible that certain transactions in this policy may be repeatedly stopped and restarted resulting in a problem known as "livelock" ([BER87]). The phenomena of waiting and restarting are never desirable. We now describe the Request Order Linked List (ROLL) object, which can be used to minimize waiting and restarting, while providing correct serializable and recoverable executions.

The ROLL method is based on the serialization graph tester (SGT) approach ([BER87]). A serialization graph tester maintains the stored serialization graph (SSG) which contains all pertinent conflict information. The SGT approach is optimal in the sense that no serializable execution is ever rejected. SGT attains serializable executions by ensuring the SSG always remains acyclic ([BER87]).

In the ROLL method data item requests are indicated using a bit-vector in which each bit position corresponds to a different data item (1 means request and 0 means no request), n bits assigned to each data item if n lock modes are to be assigned (ie. 2 bits if read and write locks are used). The ROLL is a linked list or FIFO queue of bit vectors and is the only data structure accessed by transactions. Conceptually, each individual element of the ROLL is owned by a transaction. Each transaction can access the ROLL either through its element or through the tail pointer. Three operations, namely, POST, CHECK and RELEASE are associated with the ROLL object. Except for the two pointer setting actions in POST all actions in each of the operations can be done in parallel. Briefly, POST is an atomic "enqueue" operation in which a transaction establishes its data needs and precedence order by POSTing a Request Vector (RV). Assuming it is desirable to distinguish between read and write mode requests, a consecutive pair of bits is assigned to each data item (a read bit and a write bit). The RV has 1-bits in those read-bit-positions corresponding to data items to be read and 1-bits in those write-bit-positions corresponding to data items to be written. CHECK is an operation which allows a transaction to determine, at any point in time, exactly which needed data items are available for immediate use. To CHECK, a transaction logically ORs all RVs ahead of its own in the ROLL. Then a logical AND of the compliment of this vector with its own determines exactly those items which are currently available. The RELEASE operation allows a transaction to give access to a data item to the next requesting transaction by simply flipping the corresponding bit to zero.

For a more detailed description of these operations the reader is referred to [PER91].

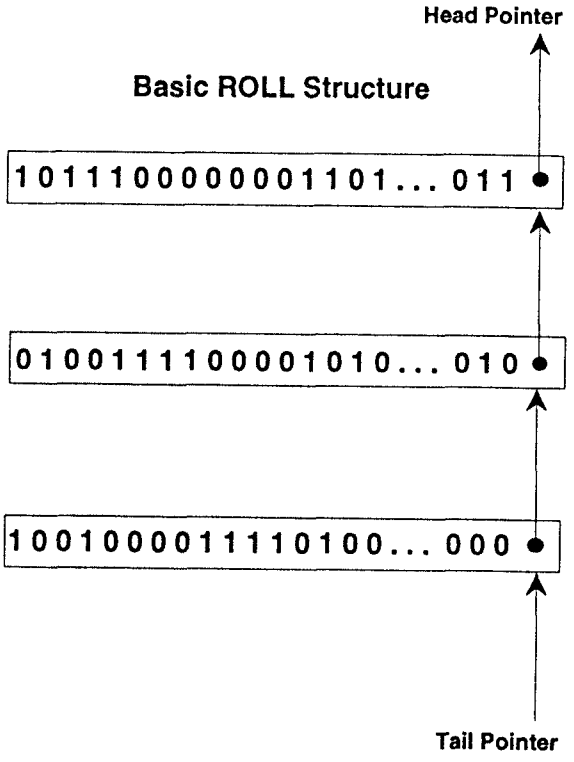

Figure 1. The basic ROLL Object Data Structure.

Two apparent problems may leap to mind at this point, the size of the Request Vectors in ROLL and its apparent static-ness. We pause at this point to address these issues. First, we note that the number of data items is small whenever the granularity is course. For instance in a HDDBMS using site level granularity, the number of bits will be equal to the number of sites. When fine granularity is needed, by partitioning the ROLL (eg. along file boundaries or even more finely, by ranges of keys within files) and providing a separate ROLL for each partition, a substantial decrease in the number of zero-bits can be realized ([PER91]). Furthermore, tuple surrogates (RIDs [BER87], RRNs [AST88],..) can be used to provide the mapping of records to bit-positions, in which case, each partition ROLL has bit-length equal only to the cardinality of the partition itself. Of course, since the size would then change over time, slightly modified basic operations must be used ([PER91]).

Secondly, once the operations are modified to handle changing RV lengths, the apparent static-ness requirement disappears. The bit vector could be made sufficiently long to accommodate expected size growth and the mapping of items to positions could be done so that positions are reused once they are freed. In the partitioned ROLL case, the same solutions apply to individual partitions and new ROLLS can be added as new portions (files) are created.

It is shown in [PER91] that ROLL concurrency control is correct. (ie. produces only serializable executions). We can summarize the proof as follows. A transaction may not access a data item until all preceding conflicting transactions in the ROLL have released it. Every operation the transaction performs which conflicts with an operation of a preceding transaction in the list must follow the operations of that transaction. Thus the serialization partial order is always equivalent to the POSTing order and therefore serializable (even though the execution may not be done in the POSTing order).

Basic ROLL is idle-wait, deadlock, livelock and restart free. In terms of the ordering of data item requests made by a transaction, Inverse Hotness Ordering ([PER88]) is automatically implemented. The simplest and most accurate measure of hotness of data items is immediate availability. With ROLL, all immediately available items are granted and unavailable (hot) items are made available as soon as possible. ROLL also automatically accommodates an incremental on-the-fly read similar to that described in [PU88] (a standard global-read-transaction [PER91]).

The main cause of delay in waiting policies, such as Two-Phase Locking (2PL), is the single system scheduler. The ROLL method does not use a scheduler; instead, each transaction POSTs its requests and monitors the availability of requested items using the CHECK operation, which is a series of simple, fast steps which are executed in parallel. While 2PL does not require predeclaration of data needs, ROLL does. Modifications to accommodate non-predeclarative transactions are described in ([PER91]). These modifications necessarily introduce the possibility of restart (at least partial restart) which is shown to be unavoidable in all non-predeclarative methods (Theorem 2, [PER91]). The details of non-predeclarative ROLL are beyond the scope of this paper, as both HYDRO-I and HYDRO-II employ the basic predeclarative ROLL. The main cause of delay in Timestamp Ordering systems is transaction restart. Since the basic ROLL method employed in HYDRO does not use restart to manage conflicts, this serious performance drawback does not exist.

3. HYDRO

As stated previously, we believe that many HDDBMSs will be developed bottom-up to accommodate currently

existing local DBMSs. Thus, we assume each local DBMS is an off-the-shelf, binary-licensed commercial product which supports local correctness through serializability, recoverability and atomic commit. Our model includes full local autonomy as defined by Elmagarmid ([ELM88]). Unlike Elmagarmid, however, we route local transactions through the local HYDRO (LHYDRO) module prior to being submitted to the local DBMS. In most operating system environments this can be accomplished by coding the LHYDRO with the same name, look and feel as the local DBMS (e.g., a LHYDRO shell on UNIX[tm] systems would allow the placement of the LHYDRO code higher in the execution search path than the local DBMS).

In our model, each global transaction G_i, is assigned a Global Transaction Manager, GTM_i. There is a Global HYDRO object, GHYDRO, available to global transactions. GHYDRO and each GTM can be viewed, for simplicity, as centralized. A GTM must interact with the GHYDRO (possibly over the network) to determine when it can begin processing at a site. When GTM_i has been cleared to process at site-j, it creates a cohort Local Transaction Manager process, $GSTM^j{}_i$, at site-j which manages the interaction at that site. Each $GSTM^j$ must interact with a local HYDRO object, LHYDRO, to determine when it can begin to access the local DBMS at that site. Local transactions are intercepted by LHYDRO and assigned local transaction managers, as well. Thus, at the local level, global subtransactions and local transactions are treated similarly.

HYDRO provides global serializability, deadlock and livelock freedom, allows multiple concurrent global transactions at any site, and reduces system-wide scheduler bottlenecks by allowing almost all concurrency control activity to execute in parallel. GHYDRO maintains a special SITE-ROLL in which each bit-position corresponds to a site. A GTM POSTs a Request Vector with a 1-bit for each site at which access is needed and 0-bits elsewhere. The LHYDRO at a local site maintains ROLLs used to govern the submission of transactions to the local DBMS by local Transaction Managers. The ROLL(s) in the LHYDRO object are structured in one of two ways. The simplest LHYDRO, called LHYDRO-I, maintains separate ROLLs for local transactions (LROLL) and global subtransactions (GROLL). LHYDRO-II maintains a single ROLL for both local and global transactions. The reason for having two LHYDRO objects is that LHYDRO-I can be used with DBMSs which do not offer a "PREPARE" statement and LHYDRO-II will be used with those DBMSs that do offer a "PREPARE". In fact, both can be used for any DBMS, but our intuition tells us that the above placement pattern is best. Further study is needed on this issue.

A global transaction begins execution by appending its Site-Request-Vector to the SITE-ROLL in GHYDRO, indicating its site access needs. It then CHECKs the SITE-ROLL to determine at which of those sites it can immediately POST a Request Vector and does so (actually the cohort GSTMs do the POSTing), and then reCHECKs periodically until local POSTing is completed. A GSTM can begin processing as soon as the local POST at its site is completed. Thus, local POSTing is done as much in parallel as possible, while still maintaining a consistent global serialization partial ordering. We assume that a Global Schema exists to provide a mapping from the required data items to specific bit-positions.

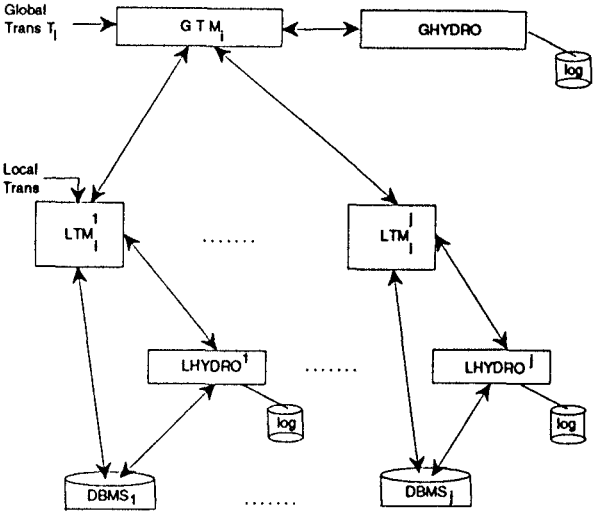

Figure 2: The HYDRO-I Model.

LHYDRO-I POST and CHECK

At sites with the LHYDRO-I object, POSTing is done as follows. A Local Transaction Manager enqueues its RVs to the LROLL and copies the GROLL-tail-address as one atomic operation. A Global Subtransaction Manager enqueues its RV to the GROLL and copies the LROLL-tail-address as one atomic operation.

At sites with the LHYDRO-I object, CHECKing is done as follows. An LTM for a local transaction CHECKs the GROLL only. If there are conflicts, it waits and reCHECKs again later (the wait duration can be a

function of the number of conflicts). Once there are no conflicts, the LTM may begin interacting with the local DBMS. A GSTM for a global transaction CHECKs both the LROLL and the GROLL. If there are conflicts, it waits and reCHECKs again later.

The net effect of these CHECK operations is that no active global subtransaction will be path connected to another active transaction in the serialization graph, eliminating the possibility of indirect conflicts ([ELM88]). Direct conflicts are resolved by the local DBMS and the compatibility of local serialization partial orderings is enforced by the SITE-ROLL.

LHYDRO-II POST and CHECK

At sites with the LHYDRO-II object, POSTing is done as follows. Both LTMs and GSTMs enqueue their RVs to the LROLL.

In LHYDRO-II we get more concurrency by allowing each active global subtransaction to share its serialization-graph path-component with active local transactions only. It must be noted, however, that the increased concurrency is gained at the expense of considerable additional atomic commit overhead, unless the DBMS offers a "PREPARE" statement in its programming interface. A "PREPARE" statement, when issued and acknowledged, guarantees to the issuer that neither a COMMIT nor a ROLLBACK statement will be rejected. Since many commercial systems do not provide a full PREPARE in this sense, and since the additional atomic commit overhead required to make LHYDRO-II work at such sites may be prohibitive, we recommend LHYDRO-I be used at all such sites.

In LHYDRO-II there is just one ROLL (LROLL), but each RV has an additional bit, called the G-bit, reserved to indicate whether or not the transaction is global (=1) or local (=0). At sites with the LHYDRO-II object, CHECKing by LTMs and GSTMs is done identically as follows. A Transaction Manager is cleared to interact with the local DBMS in one of two ways. The first method provides a very fast test. The second method is exhaustive. Before an exhaustive test is made, a fast QUICKCHECK is attempted to determine if a local submission would result in just one active global transaction at that site. If this is the case, then no indirect global conflict can result, and the subtransaction can begin interacting with the local DBMS immediately. A QUICKCHECK is performed by comparing two simple counter values, the number of global transactions that were posted when the transaction was posted, and the number of transactions that have already completed. If the difference between these two values is less than 2, immediate submission is permitted. If QUICKCHECK fails then the following full CHECK algorithm is used. In this algorithm, Own is the transaction's vector; Temp and Influence are two vectors that are treated algorithmically as program variables. Temp and Influence are initialized to the value of Own. Another variable, Next, is a cursor on the LROLL queue. It is initialized to point to the successor vector of Own. The bitwise "AND" and "OR" vector operations are indicated by "&" and "|", respectively. Assignment follows the Ada and Pascal convention.

```
CHECK ALGORITHM:
Step 1.0      REPEAT
  Step 1.1      Temp:= Next & Temp
  Step 1.2      IF (Temp != AllZeros) THEN
    Step 1.2.1    IF (Temp[G_bit] = 1 ) THEN Exit
    Step 1.2.2    ELSE Influence:= Next | Influence
  Step 1.3      Temp:= Influence,  Next:= the next
                                   vector in ROLL
  Step 1.4    UNTIL (Next = HeadOfQueue)
Step 2.0      START-Transaction
```

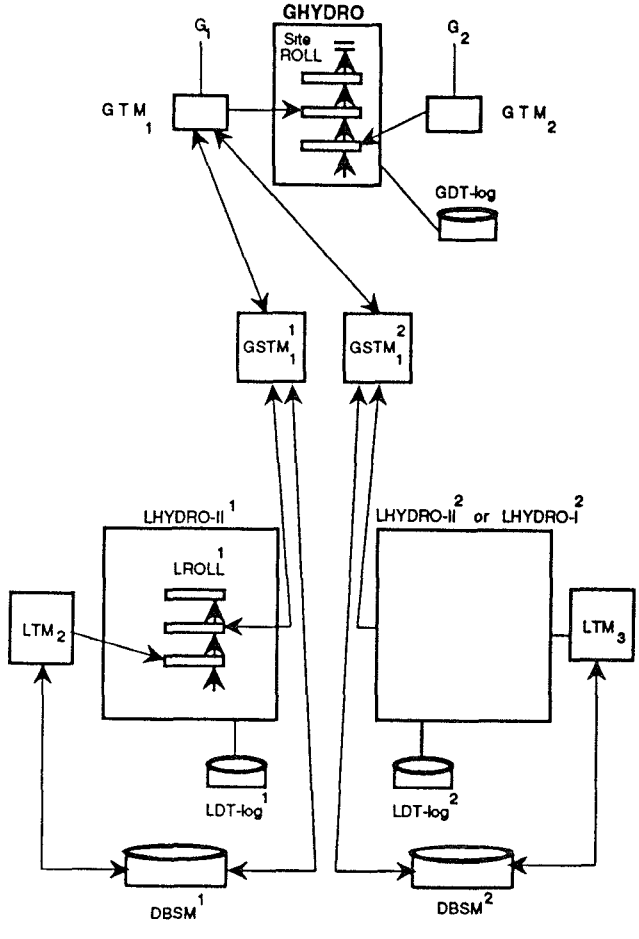

Figure 3: The HYDRO-II Model.

The Influence vector specifies all data influenced by the transaction either directly or indirectly (through conflicts with other global and local transactions). A transaction is cleared to START in Step 2.0, only if its Influence vector shows a direct or indirect conflict with no more than one global transaction. Thus, no two global transactions will ever be in direct or indirect conflict with one another. This is shown to be a necessary condition for global serializability in [ELM88]. In both LHYDRO-I and LHYDRO-II, to ensure correctness, vector removal from a ROLL must be delayed until a vector has become a source node in the serialization graph testing sense [BER87]. One way to guarantee this is to use a background process called REMOVE. The REMOVE process could be as follows.

```
REMOVE(background process):
  REPEAT
  Dequeue Head;
  Increment Counters;
UNTIL (Head = Uncommitted)
```

Enhancements to LHYDRO-II

Before going on to the atomic commitment details, we pause to discuss some additions which can further enhance the effectiveness of HYDRO-II. For example, we have investigated alternative CHECK and REMOVE operations that improve performance by keeping a local WAITING list of each transaction which has CHECKed the LROLL but failed to clear for STARTing, together with the identifier of the first blocking transaction ahead of it in the LROLL. In so doing, the REMOVE process can submit the waiting transaction as soon as the blocking transaction has completed. This eliminates the need for further CHECKing on the part of a waiting transaction and allows transactions to be submitted in batches immediately upon becoming unblocked. Note that the above discussion deals with static transactions only. To accommodate dynamic transactions (in which the scope of the read and write sets becomes known only after a certain reads have been performed), and to accommodate lock promotion ([BER87]), multiple POSTs per transaction must be allowed (via rePOSTing) and all writes locks must be delayed until the final POST. These modifications necessarily introduce the possibility of restart (at least partial) which is unavoidable in non-predeclarative approaches (Theorem 2 of [PER91]). Each succeeding element POSTed for a given transaction would be a superset (have 1's at least where its predecessor did), and with each succeeding POST, the previously POSTed element for that transaction can be zeroed. Also with each succeeding POST, if conflicting writes have intervened, the affected items must be re-read. If the re-read changes the newly POSTed element, new intervening writes could arise. This sequence of events could conceivably continue indefinitely, resulting in livelock, though it is a very remote possibility. Some type of livelock management would be required. To break livelock, the transaction could be allowed up to a certain maximum number of separate POST operations. Following these separate POSTs, it would be required to post a vector containing all ones (request everything). As soon as it is known exactly which data items it does not need, they would be released (bits reset to 0). This is a severe action to take, but it would be very unlikely. It does break the livelock, however, without introducing restarts. To reduce the potential for the above occurrence, a policy of "starving" early POSTs could be imposed. That is, only those reads which are absolutely necessary to establish the final read and write sets for the transaction would be included in early POSTs. All other locks would be delayed until the final POST. This policy would reduce still further the probability that the livelock breaking action would ever be needed.

4. ATOMIC COMMIT PROTOCOLS

To achieve atomic commitment for global transactions HYDRO-I and HYDRO-II use a variation of the standard Two-Phase Commitment (2PC) [BER87]. Two-Phase Commitment protocols involve a *voting* phase in which all GSTMs vote yes or no in response to a prepare-message from their GTM. Before voting yes, the GSTM must guarantee that it can go either way (Commit or Abort) at that site. In the presence of full local autonomy this state is difficult to achieve. The second phase of a 2PC protocol is the *decision* phase in which the GTM makes a decision and broadcasts it to the GSTMs (all GSTMs must have been PREPARED in order for the GTM to come to a Commit decision). We discuss how to achieve the PREPARED state using LHYDRO-I first.

Atomic Commit using LHYDRO-I

In LHYDRO-I, upon getting acknowledgment from the local DBMS that the subtransaction has been locally committed, and before resetting any 1-bits in its RV to 0, the GSTM is in the PREPARED state. This is the case, since the GSTM can certainly commit if the decision is to commit. Since the local DBMS has acknowledged commitment locally, durability of the local DBMS guarantees that the effects of the local subtransaction are permanent. In fact, if the decision is to commit, the GSTM need take no action other than to release its RV 1-bits in the GROLL. If the decision sent down from the GTM is to abort, the GSTM would submit a *compensating* transaction to undo its effects locally. In general, compensating transactions are difficult to compose ([KOR90]). However in this case, since no write-bits in the GSTM's RV have been turned off and therefore no over-writes or read-froms have occurred, compensation is a simple matter of restoring the before-values of all writes by that GSTM. In order to insure that this is

possible, before-values would have to have been force written to a Local-Distributed-Transaction log (LDT-log) prior to sending the yes vote.

Ordinarily, following local commitment, the effects of the local subtransaction are exposed and vulnerable to access (read or overwrite). Such subsequent access cannot be allowed until global commit, since the subsequent accessing transactions might need to be aborted also (cascading abort). These subsequent transactions might well have been globally committed themselves during the uncertainty period between the time the GSTM sends its yes vote and the time it receives the decision message. Durability at the local level dictates that these other committed transactions cannot be undone. This difficulty is circumvented in LHYDRO-I, since the Request Vector blocks other transactions from accessing data in a mode which conflicts with GSTM's access until the 1-bits in the GSTM's RV are set to 0's. As stated this RELEASE operation is delayed until after global commit or abort is achieved. We note finally that Two-Phase-Commit works in a fairly straight forward manner with LHYDRO-I due to the fact that an active global subtransaction is maintained as serialization-graph-path-disconnected from other transactions at that site.

The condition that global subtransactions be path-isolated in the serialization graph, appears to be a necessary condition for the achievement of a legitimate "PREPARED" state for fully autonomous DBMSs without the PREPARE interface statement. Many DBMSs, designed using the client-server architecture (e.g. SYBASE[tm]) provide a visible PREPARED state. For sites containing such systems, LHYDRO-II is recommended.

Atomic Commit using LHYDRO-II

First, if the DBMS provides a PREPARE command and a visible PREPARED state, atomic commitment is identical to that of LHYDRO-I. Thus, we assume in the rest of this section that no visible PREPARED state is offered.

In LHYDRO-II (assuming no visible PREPARED state) the path-isolation condition of HYDRO-I must also be achieved, but the mechanisms for achieving it are only used when necessary. This allows more concurrent access to local data than with LHYDRO-I. On the other hand, when it *is* necessary, it reduces concurrency severely for a short period of time. During the uncertainty period, when a global transaction can decide either to commit or to abort, LHYDRO-II must trap all commit-operations issued from LTMs which are path connected to the GSTM in the serialization graph (LROLL provides the information necessary to determine which LTMs to include). This amounts to a temporary switch to LHYDRO-I in which no read-from or over-write accesses are allowed to become permanent. Note that only commit-operations of serialization-graph-path-connected local transactions are trapped, so that other transaction activities are not held up to the same extent that they are in LHYDRO-I. If the decision is made to commit, the trapped commit-operations can be released. If the decision is to abort, each trapped commit-operation must be sent on as an abort and each GSTM must be informed of the abort. This is a severe action which will waste considerable processing progress in the event that a GSTM votes yes but the eventual decision is to abort. However, this is unavoidable as the global transaction must be an atomic global unit of work on the database.

5. CONCLUSIONS AND FUTURE WORK

We have shown how global serializability and atomic commit can be attained in a Heterogeneous Distributed Database Management System in which full local autonomy is provided to the local DBMSs. HYDRO is being developed for heterogeneous distributed database management in a network environment containing IBM AS/400 Database Management Systems and other systems. HYDRO uses global serializability as a correctness criterion. Full local autonomy is taken to mean that each local DBMS is an off-the-shelf, binary-licensed commercial product which provides local correctness, local recoverability, local atomic commit and some superset of the IBM SAA Common Programming Interface ([SAA88]).

Serializability is achieved using the Request Order Linked List (ROLL) method within a global HYDRO object, GHYDRO, and local HYDRO objects (LHYDROs) at each site. ROLL provides freedom from idle-wait, deadlock, livelock and restart. It is based on the general Serialization Graph Testing methodology ([BER87]), which allows all serializable executions.

At each site, HYDRO has a local server module (LHYDRO) which is customized to take advantage of the local interface provided by the DBMS software at that site. Atomic commitment is achieved using Two-Phase commitment. LHYDRO-I achieves the prepared state by protecting writes during the uncertainty period between a yes vote and the commit or abort decision. LHYDRO-II provides more concurrency, but raises the commitment overhead in the absence of a visible PREPARED state offered by the local DBMS.

Implementations of HYDRO-I and HYDRO-II are in progress in a HDDBMS setting consisting of a Solbourne machine, four Sun workstations, two NeXT machines and two AS/400 machines. A group is also working on simulating HYDRO-I and HYDRO-II using simulation languages. This could be useful when there is a need to compare HYDRO with other approaches.

Work is also being done to provide non-atomic commit and global recovery features to HYDRO-I and HYDRO-II. Also, non-serializable concurrency control is being investigated.

REFERENCES

[AST88] IBM Application System/400 Technology, IBM Rochester, MN, SA21-9540-0.

[BER87] P.A Bernstein, V. Hadzilacos and N. Goodman, Concurrency Control and Recovery in DBMS, Addison-Wesley, 1987.

[BRE90] Breitbart, Y., Silberschatz, A., and Thompson, G., Reliable Transaction Management in Multidatabase Systems SIGMOD, 1990.

[ELM88] Elmagarmid, A. and Helal, A.A. , Supporting Updates in Heterogeneous DDBMSs, IEEE Data Engineering Conf., 1988.

[ESW76] Eswaran, K., et al, The Notions of Consistency and Predicate Locks in a Database System, CACM, Vol. 19 No. 11, Nov. 1976, pp. 624-633

[GAR87] H. Garcia-Molina and K. Salem, SAGAS, ACM SIGMOD 1987, pp. 249-259.

[KOR90] Korth, H.F., Levy, E. and Silberschatz, A., A Formal Approach to Recovery by Compensating Transactions, Proc. of VLDB-90 pp. 95-106, 1990

[PER88] Perrizo, W., Luo, M. and Varvel, D., Ordering Accesses to Improve Transaction Processing Performance, Int'l Conf., on Data Engineering, Los Angeles, Feb, 1988, pp. 58-63.

[PER89] Perrizo, W. and Richter, R., Concurrency Control Using an Extended Query Language, 4th Int'l Conference on Supercomputing, Santa Clara, CA, 1989.

[PER91] Perrizo, W., Request Order Linked List (ROLL): A Concurrency Control Object, to appear in IEEE Int'l Conf. on Data Engineering, April, 1991, Kobe, Japan.

[PU88] Pu, C., Hong J. & Wha, Performance Evaluation of Global Reading of Entire Databases, Symp on Databases in Parallel & Distr. Sys. Austin, Dec 1988.

[SAA88] System Application Architecture, Common Programming Interface Database Reference, IBM-SC26-4348-1, 1988.

[TUL90] NSF Workshop on Multidatabases and Semantic Interoperability, Tulsa, OK, November 2-4, 1990.