Transaction Management in the R* Distributed Database Management System

C. MOHAN, B. LINDSAY, and R. OBERMARCK
IBM Almaden Research Center

This paper deals with the transaction management aspects of the R* distributed database system. It concentrates primarily on the description of the R* commit protocols, Presumed Abort (PA) and Presumed Commit (PC). PA and PC are extensions of the well-known, two-phase (2P) commit protocol. PA is optimized for read-only transactions and a class of multisite update transactions, and PC is optimized for other classes of multisite update transactions. The optimizations result in reduced intersite message traffic and log writes, and, consequently, a better response time. The paper also discusses R*'s approach toward distributed deadlock detection and resolution.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed databases*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *deadlocks*; *synchronization*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; H.2.0 [**Database Management**]: General—*concurrency control*; H.2.2 [**Database Management**]: Physical Design—*recovery and restart*; H.2.4 [**Database Management**]: Systems—*distributed systems*; *transaction processing*; H.2.7 [**Database Management**]: Database Administration—*logging and recovery*

General Terms: Algorithms, Design, Reliability

Additional Key Words and Phrases: Commit protocols, deadlock victim selection

1. INTRODUCTION

R* is an experimental, distributed database management system (DDBMS) developed and operational at the IBM San Jose Research Laboratory (now renamed the IBM Almaden Research Center) [18, 20]. In a distributed database system, the actions of a transaction (an atomic unit of consistency and recovery [13]) may occur at more than one site. Our model of a transaction, unlike that of some other researchers' [25, 28], permits multiple data manipulation and definition statements to constitute a single transaction. When a transaction execution starts, its actions and operands are not constrained. Conditional execution and ad hoc SQL statements are available to the application program. The whole transaction need not be fully specified and made known to the system in advance. A distributed transaction commit protocol is required in order to ensure either that *all* the effects of the transaction persist or that *none* of the

Authors' address: IBM Almaden Research Center, K55/801, 650 Harry Road, San Jose, CA 95120.

effects persist, despite intermittent site or communication link failures. In other words, a commit protocol is needed to guarantee the uniform commitment of distributed transaction executions.

Guaranteeing uniformity requires that certain facilities exist in the distributed database system. We assume that each process of a transaction is able to *provisionally* perform the actions of the transaction in such a way that they can be undone if the transaction is or needs to be aborted. Also, each database of the distributed database system has a *log* that is used to recoverably record the state changes of the transaction during the execution of the commit protocol and the transaction's changes to the database (the UNDO/REDO log [14, 15]). The log records are carefully written sequentially in a file that is kept in *stable* (nonvolatile) *storage* [17].

When a log record is written, the write can be done synchronously or asynchronously. In the former case, called *forcing* a log record, the forced log record *and* all preceding ones are *immediately* moved from the virtual memory buffers to stable storage. The transaction writing the log record is not allowed to continue execution until this operation is completed. This means that, if the site crashes (assuming that a crash results in the loss of the contents of the virtual memory) after the force-write has completed, then the forced record and the ones preceding it will have survived the crash and will be available, from the stable storage, when the site recovers. It is important to be able to "batch" force-writes for high performance [11]. R* does rudimentary batching of force-writes.

On the other hand, in the asynchronous case, the record gets written to virtual memory buffer storage and is allowed to migrate to the stable storage later on (due to a subsequent force or when a log page buffer fills up). The transaction writing the record is allowed to continue execution before the migration takes place. This means that, if the site crashes after the log write, then the record may not be available for reading when the site recovers. An important point to note is that a synchronous write increases the response time of the transaction compared to an asynchronous write. Hereafter, we refer to the latter as simply a write and the former as a force-write.

Several commit protocols have been proposed in the literature, and some have been implemented [8, 16, 17, 19, 23, 26, 27]. These are variations of what has come to be known as the two-phase (2P) commit protocol. These protocols differ in the number of messages sent, the time for completion of the commit processing, the level of parallelism permitted during the commit processing, the number of state transitions that the protocols go through, the time required for recovery once a site becomes operational after a failure, the number of log records written, and the number of those log records that are force-written to stable storage. In general, these numbers are expressed as a function of the number of sites or processes involved in the execution of the distributed transaction.

Some of the desirable characteristics in a commit protocol are (1) guaranteed transaction atomicity always, (2) ability to "forget" outcome of commit processing after a short amount of time, (3) minimal overhead in terms of log writes and message traffic, (4) optimized performance in the no-failure case, (5) exploitation of completely or partially read-only transactions, and (6) maximizing the ability to perform unilateral aborts.

This paper concentrates on the performance aspects of commit protocols, especially the logging and communication performance during no-failure situations. We have been careful in describing when and what type of log records are written. The discussions of commit protocols in the literature are very vague, if there is any mention at all, about this crucial (for correctness and performance) aspect of the protocols. We also exploit the read-only property of the complete transaction or some of its processes. In such instances, one can benefit from the fact that for such processes of the transaction it does not matter whether the transaction commits or aborts, and hence they can be excluded from the second phase of the commit protocol. This also means that the (read) locks acquired by such processes can be released during the first phase. No a priori assumptions are made about the read-only nature of transactions. Such information is discovered only during the first phase of the commit protocol.

Here, we suggest that complicated protocols developed for dealing with rare kinds of failures during commit coordination are not worth the costs that they impose on the processing of distributed transactions during normal times (i.e., when no failures occur). Multilevel hierarchical commit protocols are also suggested to be more natural than the conventional two-level (one coordinator and a set of subordinates) protocols. This stems from the fact that the distributed query processing algorithms are efficiently implemented as a tree of cooperating processes.

With these goals in mind, we extended the conventional 2P commit protocol to support a tree of processes [18] and defined the Presumed Abort (PA) and the Presumed Commit (PC) protocols to improve the performance of distributed transaction commit.

R*, which is an evolution of the centralized DBMS System R [5], like its predecessor, supports transaction serializability and uses the two-phase locking (2PL) protocol [10] as the concurrency control mechanism. The use of 2PL introduces the possibility of deadlocks. R*, instead of preventing deadlocks, allows them (even distributed ones) to occur and then resolves them by deadlock detection and victim transaction abort.

Some of the desirable characteristics in a distributed deadlock detection protocol are (1) all deadlocks are resolved in spite of site and link failures, (2) each deadlock is detected only once, (3) overhead in terms of messages exchanged is small, and (4) once a distributed deadlock is detected the time taken to resolve it (by choosing a victim and aborting it) is small.

The general features of the global deadlock detection algorithm used in R* are described in [24]. Here we concentrate on the specific implementation of that distributed algorithm in R* and the solution adopted for the global deadlock victim selection problem. In general, as far as global deadlock management is concerned, we suggest that if distributed detection of global deadlocks is to be performed then, in the event of a global deadlock, it makes sense to choose as the victim a transaction that is local to the site of detection of that deadlock (in preference to, say, the "youngest" transaction which may be a nonlocal transaction), assuming that such a local transaction exists.

The rest of this paper is organized as follows. First, we give a careful presentation of 2P. Next, we derive from 2P in a stepwise fashion the two new protocols, namely, PA and PC. We then present performance comparisons, optimizations, and extensions of PA and PC. Next, we present the R* approach to global deadlock detection and resolution. We then conclude by outlining the current status of R*.

2. THE TWO-PHASE COMMIT PROTOCOL

In 2P, the model of a distributed transaction execution is such that there is one process, called the *coordinator*, that is connected to the user application and a set of other processes, called the *subordinates*. During the execution of the commit protocol the subordinates communicate only with the coordinator, not among themselves. Transactions are assumed to have globally unique names. The processes are assumed to have globally unique names (which also indicate the locations of the corresponding processes; the processes do not migrate from site to site).[1] All the processes together accomplish the actions of a distributed transaction.

2.1 2P Under Normal Operation

First, we describe the protocol without considering failures. When the user decides to commit a transaction, the coordinator, which receives a commit-transaction command from the user, initiates the first phase of the commit protocol by sending *PREPARE* messages, in parallel, to the subordinates to determine whether they are willing to commit the transaction.[2] Each subordinate that is willing to let the transaction be committed *first* force-writes a *prepare* log record and then sends a *YES VOTE* to the coordinator and waits for the final decision (commit/abort) from the coordinator. The process is then said to be in the **prepared** state, and it cannot unilaterally commit or abort the transaction. Each subordinate that wants to have the transaction aborted force-writes an *abort* record and sends a *NO VOTE* to the coordinator. Since a *NO VOTE* acts like a veto, the subordinate knows that the transaction will definitely be aborted by the coordinator. Hence the subordinate does not need to wait for a coordinator response before aborting the local effects of the transaction. Therefore, the subordinate aborts the transaction, releases its locks, and "forgets" it (i.e., no information about this transaction is retained in virtual storage).

After the coordinator receives the votes from all its subordinates, it initiates the second phase of the protocol. If all the votes were *YES VOTE*s, then the coordinator moves to the **committing** state by force-writing a *commit* record and sending *COMMIT* messages to all the subordinates. The completion of the force-write takes the transaction to its **commit point**. Once this point is passed the user can be told that the transaction has been committed. If the coordinator had received even one *NO VOTE*, then it moves to the **aborting** state by force-writing an *abort* record and sends *ABORT*s to (only) all the subordinates that are in the **prepared** state or have not responded to the *PREPARE*. Each subordinate, after receiving a *COMMIT*, moves to the **committing** state,

[1] For ease of exposition, we assume that each site participating in a distributed transaction has only one process of that transaction. However, the protocols presented here have been implemented in R*, where this assumption is relaxed to permit more than one such process per site.

[2] *In cases where the user or the coordinator wants to abort the transaction, the latter sends an ABORT* message to each of the subordinates. If a transaction is resubmitted after being aborted, it is given a new name.

force-writes a *commit* record, sends an acknowledgment (*ACK*) message to the coordinator, and then commits the transaction and "forgets" it. Each subordinate, after receiving an *ABORT*, moves to the **aborting** state, force-writes an *abort* record, sends an *ACK* to the coordinator, and then aborts the transaction and "forgets" it. The coordinator, after receiving the *ACK*s from all the subordinates that were sent a message in the second phase (remember that subordinates who voted *NO* do not get any *ABORT*s in the second phase), writes an *end* record and "forgets" the transaction.

By requiring the subordinates to send *ACK*s, the coordinator ensures that all the subordinates are aware of the final outcome. By forcing their *commit/abort* records before sending the *ACK*s, the subordinates make sure that they will *never* be required (while recovering from a processor failure) to ask the coordinator about the final outcome after having acknowledged a *COMMIT/ABORT*. The general principle on which the protocols described in this paper are based is that if a subordinate acknowledges the receipt of any particular message, then it should make sure (by forcing a log record with the information in that message *before* sending the *ACK*) that it will never ask the coordinator about that piece of information. If this principle is not adhered to, transaction atomicity may not be guaranteed.

The log records at each site contain the type (*prepare*, *end*, etc.) of the record, the identity of the process that writes the record, the name of the transaction, the identity of the coordinator, the names of the exclusive locks held by the writer in the case of *prepare* records, and the identities of the subordinates in the case of the *commit/abort* records written by the coordinator.

To summarize, for a committing transaction, during the execution of the protocol, each subordinate writes two records (*prepare* and *commit*, both of which are forced) and sends two messages (*YES VOTE* and *ACK*). The coordinator sends two messages (*PREPARE* and *COMMIT*) to each subordinate and writes two records (*commit*, which is forced, and *end*, which is not).

Figure 1 shows the message flows and log writes for an example transaction following 2P.

2.2 2P and Failures

Let us now consider site and communication link failures. We assume that at each active site a *recovery process* exists and that it processes all messages from recovery processes at other sites and handles all the transactions that were executing the commit protocol at the time of the last failure of the site. We assume that, as part of recovery from a crash, the recovery process at the recovering site reads the log on stable storage and accumulates in *virtual storage* information relating to transactions that were executing the commit protocol at the time of the crash.[3] It is this information in virtual storage that is used to answer queries from other sites about transactions that had their coordinators at this site and to send unsolicited information to other sites that had subordinates for transactions that had their coordinators at this site. Having the

[3] The extent of the log that has to be read on restart can be controlled by taking *checkpoints* during normal operation [14, 15]. The log is scanned forward starting from the last checkpoint before the crash until the end of the log.

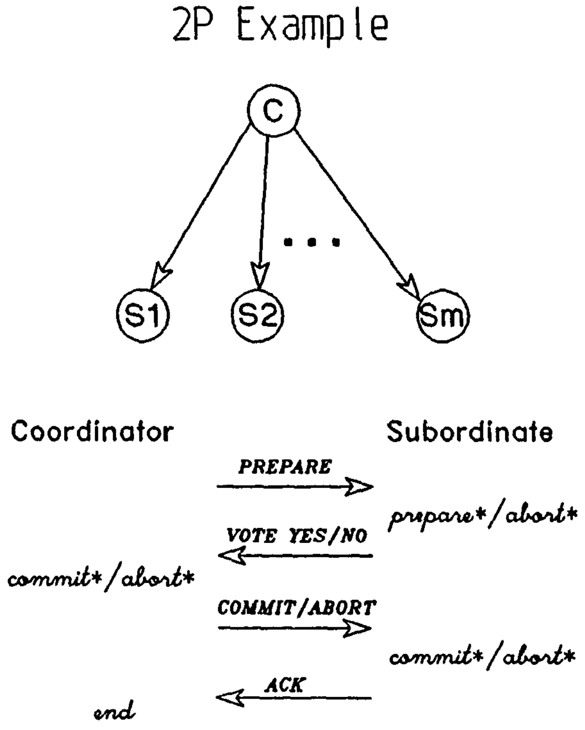


Fig. 1. Message flows and log writes in 2P. The names in italics indicate the types of log records written. An * next to the record type means that the record is forced to stable storage.

information in virtual storage allows remote site inquiries to be answered quickly. There will be no need to consult the log to answer the queries.

When the recovery process finds that it is in the **prepared** state for a particular transaction, it *periodically* tries to contact the coordinator site to find out how the transaction should be resolved. When the coordinator site resolves a transaction and lets this site know the final outcome, the recovery process takes the steps outlined before for a subordinate when it receives an *ABORT/COMMIT*. If the recovery process finds that a transaction was executing at the time of the crash and that no commit protocol log record had been written, then the recovery process neither knows nor cares whether it is dealing with a subordinate or the coordinator of the transaction. It aborts that transaction by "undoing" its actions, if any, using the UNDO log records, writing an *abort* record, and "forgetting" it.[4] If the recovery process finds a transaction in the **committing** (respectively, **aborting**) state, it periodically tries to send the *COMMIT* (*ABORT*) to all the subordinates that have not acknowledged and awaits their *ACK*s. Once all the

[4] It should be clear now why a subordinate cannot send a *YES VOTE* first and then write a *prepare* record, and why a coordinator cannot send a *COMMIT* first and then write the *commit* record. If such actions were permitted, then a failure after the message sending but before the log write may result in the wrong action being taken at restart; some sites might have committed and others may abort.

*ACK*s are received, the recovery process writes the *end* record and "forgets" the transaction.

In addition to the workload that the recovery process accumulates by reading the log during restart, it may be handed over some transactions during normal operation by local coordinator and subordinate processes that notice some link or remote site failures during the commit protocol (see [18] for information relating to how such failures are noticed). We assume that all failed sites ultimately recover.

If the coordinator process notices the failure of a subordinate while waiting for the latter to send its vote, then the former aborts the transaction by taking the previously outlined steps. If the failure occurs when the coordinator is waiting to get an *ACK*, then the coordinator hands the transaction over to the recovery process.

If a subordinate notices the failure of the coordinator before the former sent a *YES VOTE* and moved into the **prepared** state, then it aborts the transaction (this is called the *unilateral abort* feature). On the other hand, if the failure occurs after the subordinate has moved into the **prepared** state, then the subordinate hands the transaction over to the recovery process.

When a recovery process receives an inquiry message from a **prepared** subordinate site, it looks at its information in virtual storage. If it has information that says the transaction is in the **aborting** or **committing** state, then it sends the appropriate response. The natural question that arises is what action should be taken if **no information** is found in virtual storage about the transaction. Let us see when such a situation could arise. Since both *COMMIT*s and *ABORT*s are being acknowledged, the fact that the inquiry is being made means that the inquirer had not received and processed a *COMMIT/ABORT* before the inquiree "forgot" the transaction. Such a situation comes about when (1) the inquiree sends out *PREPARE*s, (2) it crashes before receiving all the votes and deciding to commit/abort, and (3) on restart, it aborts the transaction and does not inform any of the subordinates. As mentioned before, on restart, the recipient of an inquiry cannot tell whether it is a coordinator or subordinate, if no commit protocol log records exist for the transaction. Given this fact, the correct response to an inquiry in the **no information** case is an *ABORT*.

2.3 Hierarchical 2P

2P as described above is inadequate for use in systems where the transaction execution model is such that multilevel (>2) trees of processes are possible, as in R* and ENCOMPASS [8]. Each process communicates directly with only its immediate neighbors in the tree, that is, parent and children. In fact, a process would not even know about the existence of its nonneighbor processes. There is a simple extension of 2P that would work in this scenario. In the hierarchical version of 2P, the root process that is connected to the user/application acts only as a coordinator, the leaf processes act only as subordinates, and the nonleaf, nonroot processes act as both coordinators (for their child processes) and subordinates (for their parent processes). The root process and the leaf processes act as in nonhierarchical 2P. A nonroot, nonleaf process after receiving a *PREPARE* propagates it to its subordinates and only after receiving their votes does it send its combined (i.e., subtree) vote to its coordinator. The type of the subtree vote is determined by the types of the votes of the subordinates and the type of the vote of the subtree's root process. If any vote is a *NO VOTE*, then the subtree vote is a *NO VOTE* also (in this case, the subtree root process, after sending the subtree vote to its coordinator, sends *ABORT*s to all those subordinates that voted *YES*). If none of the votes is a *NO VOTE*, then the subtree vote is a *YES VOTE*. A nonroot, nonleaf process in the **prepared** state, on receiving an *ABORT* or a *COMMIT*, propagates it to its subordinates after force-writing its *commit* record and sending the *ACK* to its coordinator.

3. THE PRESUMED ABORT PROTOCOL

In Section 2.2 we noticed that, in the absence of any information about a transaction, the recovery process orders an inquiring subordinate to abort. A careful examination of this scenario reveals the fact that it is safe for a coordinator to "forget" a transaction immediately after it makes the decision to abort it (e.g., by receiving a *NO VOTE*) and to write an *abort* record.[5] This means that the *abort* record need not be forced (both by the coordinator and each of the subordinates), and no *ACK*s need to be sent (by the subordinates) for *ABORT*s. Furthermore, the coordinator need not record the names of the subordinates in the *abort* record or write an *end* record after an *abort* record. Also, if the coordinator notices the failure of a subordinate while attempting to send an *ABORT* to it, the coordinator does *not* need to hand the transaction over to the recovery process. It will let the subordinate find out about the abort when the recovery process of the subordinate's site sends an inquiry message. Note that the changes that we have made so far to the 2P protocol have not changed the performance (in terms of log writes and message sending) of the protocol with respect to committing transactions.

Let us now consider completely or partially *read-only* transactions and see how we can take advantage of them. A transaction is partially read-only if some processes of the transaction do not perform any updates to the database while the others do. A transaction is (completely) read-only if no process of the transaction performs any updates. We do not need to know before the transaction starts whether it is read-only or not.[6] If a leaf process receives a *PREPARE* and it finds that it has not done any updates (i.e., no UNDO/REDO log records have been written), then it sends a *READ VOTE*, releases its locks, and "forgets" the transaction. The subordinate writes **no** log records. As far as it is concerned, it does not matter whether the transaction ultimately gets aborted or committed. So the subordinate, who is now known to the coordinator to be read-only, does not need to be sent a *COMMIT/ABORT* by the coordinator. A nonroot, nonleaf sends a *READ VOTE* only if its own vote and those of its subordinates' are also *READ VOTE*s. Otherwise, as long as none of the latter is a *NO VOTE*, it sends a *YES VOTE*.

[5] Remember that in 2P the coordinator (during normal execution) "forgets" an abort only after it is sure that all the subordinates are aware of the abort decision.

[6] If the program contains conditional statements, the same program during different executions may be either read-only or update depending on the input parameters and the database state.

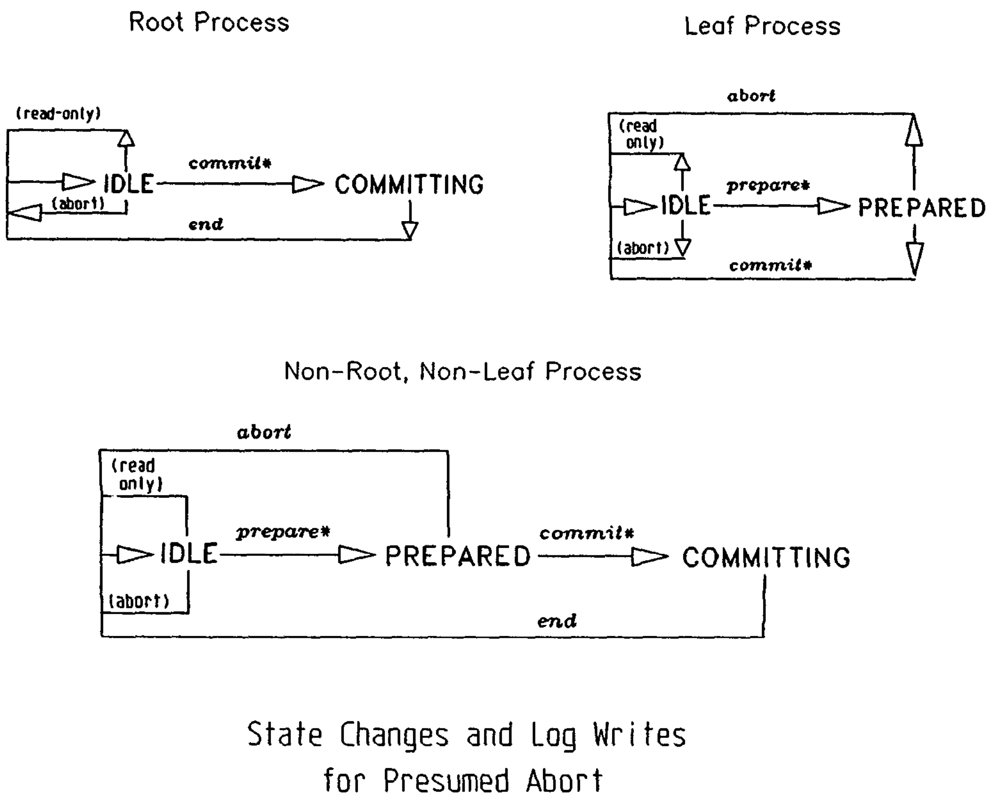


Fig. 2. The names in italics on the arcs of the state-transition diagrams indicate the types of log records written. An * next to the record type means that the record is forced to stable storage. No log records are written during some transitions. In such cases, information in parentheses indicates under what circumstances such transitions take place. IDLE is the initial and final state for each process.

There will not be a second phase of the protocol if the root process is read-only and it gets only *READ VOTE*s. In this case the root process, just like the other processes, writes **no** log records for the transaction. On the other hand, if the root process or one of its subordinates votes *YES* and none of the others vote *NO*, then the root process behaves as in 2P. But note that it is sufficient for a nonleaf process to include in the *commit* record only the identities of those subordinates (if any) that voted *YES* (only those processes will be in the **prepared** state, and hence only they will need to be sent *COMMIT*s). If a nonleaf process or one of its subordinates votes *NO*, then the former behaves as described earlier in this section.

To summarize, for a (completely) read-only transaction, none of the processes write any log records, but each one of the nonleaf processes sends one message (*PREPARE*) to each subordinate and each one of the nonroot processes sends one message (*READ VOTE*).

For a committing partially read-only transaction, the root process sends two messages (*PREPARE* and *COMMIT*) to each update subordinate and one message (*PREPARE*) to each of the read-only subordinates. Each one of the nonleaf,

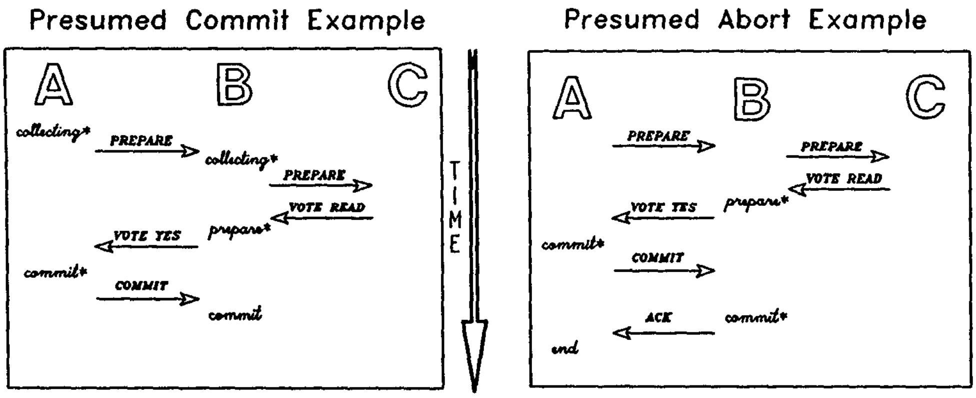


Fig. 3. Message flows and log writes in PA and PC. *A* (update/read-only) is the root of the process tree with *B* (update) as its child. *C* (read-only) is the leaf of the tree and the child of *B*.

nonroot processes that is the root of an update subtree sends two messages (*PREPARE* and *COMMIT*) to each update subordinate, one message (*PREPARE*) to each of the other subordinates, and two messages (*YES VOTE* and *ACK*) to its coordinator. Each one of the nonleaf, nonroot processes that is the root of a read-only subtree behaves just like the corresponding processes in a completely read-only transaction following PA. Each one of the nonleaf processes writes three records (*prepare* and *commit*, which are forced, and *end*, which is not) if there is at least one update subordinate, and only two records (*prepare* and *commit*, which are forced) if the nonleaf process itself is an update one and it does not have any update subordinates. A read-only leaf process behaves just like the one in a completely read-only transaction following PA, and an update leaf process behaves like a subordinate of a committing transaction in 2P.

By making the above changes to hierarchical 2P, we have generated the PA protocol. The name arises from the fact that in the **no information** case the transaction is presumed to have aborted, and hence the recovery process's response to an inquiry is an *ABORT*. Figure 2 shows the state transitions and log writes performed by the different processes following PA. Figure 3 shows the message flows and log writes for an example transaction following PA.

4. THE PRESUMED COMMIT PROTOCOL

Since most transactions are expected to commit, it is only natural to wonder if, by requiring *ACK*s for *ABORT*s, commits could be made cheaper by eliminating the *ACK*s for *COMMIT*s. A simplistic idea that comes to mind is to require that *ABORT*s be acknowledged, while *COMMIT*s need not be, and also that *abort* records be forced while *commit* records need not be by the subordinates. The consequences are that, in the **no information** case, the recovery process responds with a *COMMIT* when a subordinate inquiries. There is, however, a problem with this approach.

Consider the situation when a root process has sent the *PREPARE*s, one subordinate has gone into the **prepared** state, and before the root process is able to collect all the votes and make a decision, the root process crashes. Note that so far the root process would not have written *any* commit protocol log records. When the crashed root process's site recovers, its recovery process will abort this transaction and "forget" it without informing anyone, since **no** information is available about the subordinates. When the recovery process of the **prepared** subordinate's site then inquires the root process's site, the latter's recovery process would respond with a *COMMIT*,[7] causing an unacceptable inconsistency.

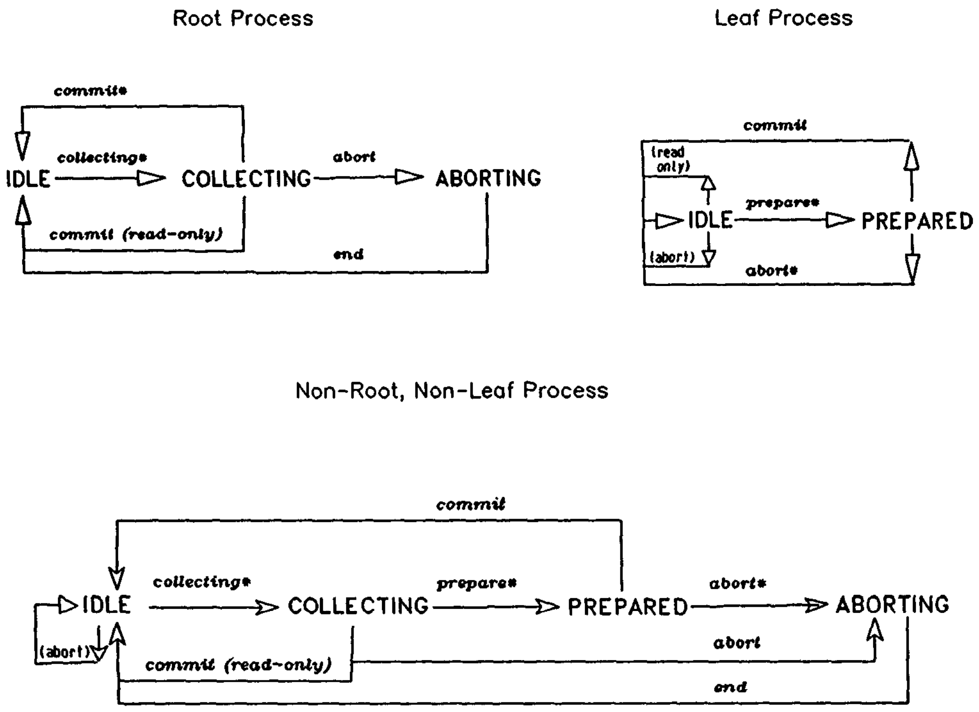
The way out of this problem is for each coordinator (i.e., nonleaf process) to record the names of its subordinates safely *before* any of the latter could get into the **prepared** state. Then, when the coordinator site aborts on recovery from a crash that occurred after the sending of the *PREPARE*s (but before the coordinator moved into the **prepared** state, in the case of the nonroot coordinators), the restart process will know who to inform (and get *ACK*s) about the abort. These modifications give us the PC protocol. The name arises from the fact that in the **no information** case the transaction is presumed to have committed and hence the response to an inquiry is a *COMMIT*.

In PC, a nonleaf process behaves as in PA except (1) at the start of the first phase (i.e., before sending the *PREPARE*s) it force-writes a *collecting* record, which contains the names of all the subordinates, and moves into the **collecting** state; (2) it force-writes only *abort* records (except in the case of the root process, which force-writes *commit* records also); (3) it requires *ACK*s only for *ABORT*s and not for *COMMIT*s; (4) it writes an *end* record only after an *abort* record (if the abort is done after a *collecting* record is written) and not after a *commit* record; (5) only when in the **aborting** state will it, on noticing a subordinate's failure, hand over the transaction to the restart process; and (6) in the case of a (completely) read-only transaction, it would not write any records at the end of the first phase in PA, but in PC it would write a *commit* record and then "forget" the transaction.

The subordinates behave as in PA except that now they force-write only *abort* records and not *commit* records, and they *ACK* only *ABORT*s and not *COMMIT*s. On restart, if the recovery process finds, for a particular transaction, a *collecting* record and no other records following it, then it force-writes an *abort* record, informs all the subordinates, gets *ACK*s from them, writes the *end* record, and "forgets" the transaction. In the **no information** case, the recovery process responds to an inquiry with a *COMMIT*.

To summarize, for a (completely) read-only transaction, each one of the nonleaf processes writes two records (*collecting*, which is forced, and *commit*, which is not) and sends one message (*PREPARE*) to each subordinate. Furthermore, each one of the nonleaf, nonroot processes sends one more message (*READ VOTE*). The leaf processes write no log records, but each one of them sends one message (*READ VOTE*) to its coordinator.

[7] Note that, as far as the recovery process is concerned, this situation is the same as when a root process, after force-writing a *commit* record (which now will not contain the names of the subordinates), tries to inform a *prepared* subordinate, finds it has crashed, and therefore "forgets" the transaction (i.e., does not hand it to the recovery process). Later on, when the subordinate inquires, the recovery process would find no information and hence would respond with a *COMMIT*.



State Changes and Log Writes
for Presumed Commit

Figure 4

For a committing partially read-only transaction, the root process writes two records (*collecting* and *commit*, both of which are forced) and sends two messages (*PREPARE* and *COMMIT*) to each subordinate that sent a *YES VOTE* and one message (*PREPARE*) to each one of the other subordinates. Each one of the nonleaf, nonroot processes that is the root of an update subtree sends two messages (*PREPARE* and *COMMIT*) to each subordinate that sent a *YES VOTE*, one message (*PREPARE*) to each one of the other subordinates, and one message (*YES VOTE*) to its coordinator, and it writes three records (*collecting* and *prepared*, which are forced, and *commit*, which is not). Read-only leaf processes, and processes that are roots of read-only subtrees, behave just like the corresponding processes in a completely read-only transaction. An update leaf process sends one message (*YES VOTE*) and writes two records (*prepare*, which is forced, and *commit*, which is not).

Figure 4 shows the state transitions and log writes performed by the different processes following PC. Figure 3 shows the message flows and log writes for an example transaction following PC.

Process Type / Protocol Type	Coordinator			Subordinate	
	U Yes US	U No US	R	US	RS
Standard 2P	2,1,-,2	-	-	2,2,2	-
Presumed Abort	2,1,1,2	1,1,1	0,0,1	2,2,2	0,0,1
Presumed Commit	2,2,1,2	2,2,1	2,1,1	2,1,1	0,0,1

U -	Update Transaction
R -	Read-Only Transaction
RS -	Read-Only Subordinate
US -	Update Subordinate
m,n,o,p -	m Records Written, n of Them Forced
	o For a Coordinator: # of Messages Sent to Each RS
	For a Subordinate: # of Messages Sent to Coordinator
	p # of Messages Sent to Each US

Fig. 5. Comparison of log I/O and messages for committing two-level process tree transactions with 2P, PA, and PC.

5. DISCUSSION

In the table of Figure 5 we summarize the performance of 2P, PA, and PC with respect to committing update and read-only transactions that have two-level process trees. Note that as far as 2P is concerned all transactions appear to be completely update transactions and that under all circumstances PA is better than 2P. It is obvious that PA performs better than PC in the case of (completely) read-only transactions (saving the coordinator two log writes, including a force) and in the case of partially read-only transactions in which only the coordinator does any updates (saving the coordinator a force-write). In both cases, PA and PC require the same number of messages to be sent. In the case of a transaction with only one update subordinate, PA and PC are equal in terms of log writes, but PA requires an extra message (*ACK* sent by the update subordinate). For a transaction with $n > 1$ update subordinates, both PA and PC require the same number of records to be written, but PA will force $n - 1$ times when PC will not. These correspond to the forcing of the *commit* records by the subordinates. In addition, PA will send n extra messages (*ACK*s).

Depending on the transaction mix that is expected to be run against a particular distributed database, the choice between PA and PC can be made. It should also be noted that the choice could be made on a transaction-by-transaction basis (instead of on a systemwide basis) at the time of the start of the first phase by the root process.[8] At the time of starting a transaction, the user could give a *hint*(*not* a guarantee) that it is likely to be read-only, in which case PA could be chosen; otherwise PC could be chosen.

It should be pointed out that our commit protocols are blocking [26] in that they require a **prepared** process that has noticed the failure of its coordinator to wait until it can reestablish communication with its coordinator's site to determine the final outcome (commit or abort) of the commit processing for that transaction. We have extended, but not implemented, PA and PC to reduce the probability of blocking by allowing a **prepared** process that encounters a coordinator failure to ask its peers about the transaction outcome. The extensions require an additional phase in the protocols and result in more messages and/or synchronous log writes even during normal times. In [23] we have proposed an approach to dealing with the blocking problem in the context of the Highly Available Systems project in our laboratory. This approach makes use of Byzantine Agreement protocols. To some extent the results of [9] support our conclusion that blocking commit protocols are not undesirable.

To handle the rare situation in which a blocked process holds up too many other transactions from gaining access to its locked data, we have provided an interface that allows the operator to find out the identities of the **prepared** processes and to forcibly commit or abort them. Of course, the misuse of this facility could lead to inconsistencies caused by parts of a transaction being committed while the rest of the transaction is aborted. In cases where a link failure is the cause of blocking, the operator at the blocked site could use the telephone to find out the coordinator site's decision and force the same decision at his or her site.

Given that we have our efficient commit protocols PA and PC, and the fact that remote updates are expected or postulated to be infrequent, the time spent executing the commit protocol is going to be small compared to the total time spent executing the whole transaction. Furthermore, site and link failures cannot be frequent or long-duration events in a well-designed and well-managed distributed system. So the probability of the failure of a coordinator happening after it sent *PREPARE*s, thereby blocking the subordinates that vote *YES* in the **prepared** state until its recovery, is going to be very low.

In R*, each site has one transaction manager (TM) and one or more database managers (DBMs). Each DBM is very much like System R [5] and performs similar functions. TM is a new (to R*) component and its function is to manage the commit protocol, perform local and global deadlock detection, and assign transaction IDs to new transactions originating at that site. So far we have pretended that there is only one log file at each site. In fact, the TM and the

[8] If this approach is taken (as we have done in R*), then the nonleaf processes should include the name of the protocol chosen in the *PREPARE* message, and all processes should include this name in the first commit protocol log record that each one writes. The name should also be included in the inquiry messages sent by restart processes, and this information is used by a recovery process in responding to an inquiry in the *no information* case.

DBMs each have their own log files. A transaction process executes both the TM code and one DBM's code (for each DBM accessed by a transaction, one process is created). The DBM incarnation of the process should be thought of as the child of the (local) TM incarnation of the same process. When the process executes the TM code, it behaves like a nonleaf node in the process tree, and it writes only commit-protocol-related records in the TM log. When the process executes the DBM code, it behaves like a leaf node in the process tree, and it writes both UNDO/REDO records and commit-protocol-related records. When different processes communicate with each other during the execution of the commit protocol, it is actually the TM incarnations of those processes, not the DBM incarnations, that communicate. The leaf nodes of the process tree in this scenario are always DBM incarnations of the processes, and the nonleaf nodes are always TM incarnations of the processes.

In cases where the TM and the DBMs at a given site make use of the same file for inserting log information of all the transactions at that site (i.e., a common log), we wanted to benefit from the fact that the log records inserted during the execution of the commit protocol by the TM and the DBMs would be in a certain order, thereby avoiding some synchronous log writes (currently, in R*, the commit protocols have been designed and implemented to take advantage of the situation when the DBMs and the TM use the same log). For example, a DBM need not force-write its *prepare* record since the subsequent force-write of the TM's *prepare* record into the same log will force the former to disk. Another example is in the case of PC, when a process and all its subordinates are at the same site. In this case, the former does not have to force-write its *collecting* record since the force of the *collecting/prepared* record by a subordinate will force it out.

With a common log, in addition to explicitly avoiding some of the synchronous writes, one can also benefit from the batching effect of more log records being written into a single file. Whenever a log page in the virtual memory buffers fills up, we write it out immediately to stable storage.

If we assume that processes of a transaction communicate with each other using virtual circuits (as in R* [20]), and that new subordinate processes may be created even at the time of receipt of a *PREPARE* message by a process (e.g., to install updates at the sites of replicated copies), then it seems reasonable to use the tree structure to send the commit-protocol-related messages also (i.e., not flatten the multilevel tree into a two-level tree just for the purposes of the commit protocol). This approach avoids the need to set up any new communication channels just for use by the commit protocol. Furthermore, there is no need to make one process in each site become responsible for dealing with commit-related messages for different transactions (as in ENCOMPASS [8]).

Just as the R* DBMs take checkpoints periodically to bound DBM restart recovery time [14], the R* TM also takes its own checkpoints. The TM's checkpoint records contain the list of active processes that are currently executing the commit protocol and those processes that are in recovery (i.e., processes in the **prepared/collecting** state and processes waiting to receive *ACKs* from subordinates). Note that we do not have to include those transactions that have not yet started executing the commit protocol. TM checkpoints are taken without completely stopping all TM activity (this is in contrast with what happens in the R* DBMs). During site restart recovery, the last TM checkpoint record is read by a recovery process, and a transaction table is initialized with its contents. Then the TM log is scanned forward and, as necessary, new entries are added to the transaction table or existing entries are modified/deleted. Unlike in the case of the DBM log (see [14]), there is no need to examine the portion of the TM log before the last checkpoint. The time of the next TM checkpoint depends on the number of transactions initiated since the last checkpoint, the amount of log consumed since the last checkpoint, and the amount of space still available in the circular log file on disk.

6. DEADLOCK MANAGEMENT IN R*

The distributed 2PL concurrency control protocol is used in R*. Data are locked where they are stored. There is no separate lock manager process. All locking-related information is maintained in shared storage where it is accessible to the processes of transactions. The processes themselves execute the locking-related code and synchronize one another. Since many processes of a transaction might be concurrently active in one or more sites, more than one lock request might be made concurrently by a transaction. It is still the case that each process of a transaction will be requesting only one lock at a time. A process might wait for one of two reasons: (1) to obtain a lock and (2) to receive a message from a cohort process of the same transaction.[9] In this scenario, deadlocks, including distributed/global ones, are a real possibility. Once we chose to do deadlock detection instead of deadlock avoidance/prevention, it was only natural, for reliability reasons, to use a distributed algorithm for global deadlock detection.[10]

In R*, there is one deadlock detector (DD) at each site. The DDs at different sites operate asynchronously. The frequencies at which local and global deadlock detection searches are initiated can vary from site to site. Each DD wakes up periodically and looks for deadlocks after gathering the wait-for information from the local DBMs and the communication manager. If the DD is looking for multisite deadlocks during a detection phase, then any information about Potential Global (i.e., multisite) Deadlock Cycles (PGDCs) received earlier from other sites is combined with the local information. No information gathered/generated during a deadlock detection phase is retained for use during a subsequent detection phase of the same DD. Information received from a remote DD is consumed by the recipient, at the most, during one deadlock detection phase. This is necessary in order to make sure that false information sent by a remote DD, which during many subsequent deadlock detection phases may not have anything to send, is not consumed repeatedly by a DD, resulting in the repeated detection of, possibly, false deadlocks. If, due to the different deadlock detection frequencies of the different DDs, information is received from multiple phases of a particular remote DD before it is consumed by the recipient, then only that remote DD's last phase's information is retained for consumption by the recipient. This is because the latest information is the best information.

The result of analyzing the wait-for information could be the discovery of some local/global deadlocks and some PGDCs. Each PGDC is a list of transactions

[9] All other types of waits are not dealt with by the deadlock detector.

[10] We refer the reader to other papers for discussions concerning deadlock detection versus other approaches [3, 4, 24].

(*not* processes) in which each transaction, except the last one, is on a lock wait on the next transaction in the list. In addition, the first transaction's one local process is known to be expected to send response data to its cohort at another site, and the last transaction's one local process is known to be waiting to receive response data from its cohort at another site. This PGDC is sent to the site on which the last transaction's local process is waiting if the first transaction's name is lexicographically less than the last transaction's name; otherwise, the PGDC is discarded. Thus wait-for information travels only in the direction of the real/potential deadlock cycle, and on the average, only half the sites involved in a global deadlock send information around the cycle. In general, in this algorithm only one site will detect a given global deadlock.

Once a global deadlock is detected, the interesting question is how to choose a victim. While one could use detailed cost measures for transactions and choose as the victim the transaction with the least cost (see [4] for some performance comparisons), the problem is that such a transaction might not be in execution at the site where the global deadlock is detected. Then, the problem would be in identifying the site that has to be informed about the victim so that the latter could be aborted. Even if information about the locations of execution of every transaction in the wait-for graph were to be sent around with the latter, or if we pass along the cycle the identity of the victim, there would still be a delay and cost involved in informing remote sites about the nonlocal victim choice. This delay would cause an increase in the response times of the other transactions that are part of the deadlock cycle. Hence, in order to expedite the breaking of the cycle, one can choose as the victim a transaction that is executing locally, assuming that the wait-for information transmission protocol guarantees the existence of such a local transaction. The latter is the characteristic of the deadlock detection protocol of R* [6, 24], and hence we choose a local victim. If more than one local transaction could be chosen as the victim, then an appropriate cost measure (e.g., elapsed time since transaction began execution) is used to make the choice. If one or more transactions are involved in more than one deadlock, no effort is made to choose as the victim a transaction that resolves the maximum possible number of deadlocks.

Depending on whether or not (1) the wait-for information transmission among different sites is synchronized and (2) the nodes of the wait-for graph are transactions or individual processes of a transaction, false deadlocks might be detected. In R* transmissions are not synchronized and the nodes of the graph are transactions. Since we do not expect false deadlocks to occur frequently, we treat every detected deadlock as a true deadlock.

Even though the general impression might be that our database systems release all locks of a transaction only at the end of the transaction, in fact, some locks (e.g., short duration page-level locks when data are being locked at the tuple-level and locks on nonleaf nodes of the indices) are released before all the locks are acquired. This means that when a transaction is aborting it will have to reacquire those locks to perform its undo actions. Since a transaction could get into a deadlock any time it is requesting locks, if we are not careful we could have a situation in which we have a deadlock involving only aborting transactions. It would be quite messy to resolve such a deadlock. To avoid this situation, we permit, at any time, only one aborting transaction to be actively reacquiring locks in a given DBM. While the above-mentioned potential problem had to be dealt with even in System R, it is somewhat complicated in R*. We have to ensure that in a global deadlock cycle there is at least one local transaction that is not already aborting and that could be chosen as the victim.

This reliable, distributed algorithm for detecting global deadlocks is operational now in R*.

7. CURRENT STATUS

The R* implementation has reached a mature state, providing support for snapshots [1, 2], distributed views [7], migration of tables, global deadlock detection, distributed query compilation and processing [20], and crash recovery. Currently there is no support for replicated or fragmented data. The prototype is undergoing experimental evaluations [21].

REFERENCES

1. ADIBA, M. Derived relations: A unified mechanism for views, snapshots and distributed data. Res. Rep. RJ2881, IBM, San Jose, Calif., July 1980.
2. ADIBA, M., AND LINDSAY, B. Database snapshots. In *Proceedings of the 6th International Conference on Very Large Data Bases* (Montreal, Oct. 1980). IEEE Press, New York, 1980, 86–91.
3. AGRAWAL, R., AND CAREY, M. The performance of concurrency control and recovery algorithms for transaction-oriented database systems. *Database Eng. 8*, 2 (June 1985), 58–67.
4. AGRAWAL, R., CAREY, M., AND MCVOY, L. The performance of alternative strategies for dealing with deadlocks in database management systems. Tech. Rep. 590, Dept. of Computer Sciences, Univ. of Wisconsin, Madison, Mar. 1985.
5. ASTRAHAN, M., BLASGEN, M., CHAMBERLIN, D., GRAY, J., KING, F., LINDSAY, B., LORIE, R., MEHL, J., PRICE, T., PUTZOLU, F., SCHKOLNICK, M., SELINGER, P., SLUTZ, D., STRONG, R., TIBERIO, P., TRAIGER, I., WADE, B., AND YOST, R. System R: A relational data base management system. *Computer 12*, 5 (May 1979), 43–48.
6. BEERI, C., AND OBERMARCK, R. A resource class-independent deadlock detection algorithm. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, Sept. 1981). IEEE Press, New York, 1981, 166–178.
7. BERTINO, E., HAAS, L., AND LINDSAY, B. View management in distributed data base systems. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, Oct. 1983) VLDB Endowment, 1983, 376–378. Also available as Res. Rep. RJ3851, IBM, San Jose, Calif., Apr. 1983.
8. BORR, A. Transaction monitoring in ENCOMPASS: Reliable distributed transaction processing. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, Sept. 1981). IEEE Press, New York, 1981, 155–165.
9. COOPER, E. Analysis of distributed commit protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Orlando, Fla., June 1982). ACM, New York, 1982, 175–183.
10. ESWARAN, K. P., GRAY, J. N., LORIE, R., A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
11. GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS fast path. *Database Eng. 8*, 2 (June 1985), 3–10.
12. GRAY, J. Notes on data base operating systems. In *Operating Systems—An Advanced Course.* Lecture Notes in Computer Science, vol. 60. Springer-Verlag, New York, 1978.
13. GRAY, J. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, Oct. 1981). IEEE Press, New York, 1981, 144–154.

14. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the system R database manager. *ACM Comput. Surv. 13*, 2 (June 1981), 223–242.
15. HAERDER, T., AND REUTER, A. Principles of transaction oriented database recovery—A taxonomy. *ACM Comput. Surv. 15*, 4 (Dec. 1983), 287–317.
16. HAMMER, M., AND SHIPMAN, D. Reliability mechanisms for SDD-1: A system for distributed databases. *ACM Trans. Database Syst. 5*, 4 (Dec. 1980), 431–466.
17. LAMPSON, B. Atomic transactions. In *Distributed Systems—Architecture and Implementation.* Lecture Notes in Computer Science, vol. 100, B. Lampson, Ed. Springer-Verlag, New York, 1980, 246–265.
18. LINDSAY, B. G., HAAS, L. M., MOHAN, C., WILMS, P. F., AND YOST, R. A. Computation and communication in R*: A distributed database manager. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 24–38. Also Res. Rep. RJ3740, IBM, San Jose, Calif., Jan. 1983.
19. LINDSAY, B., SELINGER, P., GALTIERI, C., GRAY, J., LORIE, R., PUTZOLU, F., TRAIGER, I., AND WADE, B. Single and multi-site recovery facilities. In *Distributed Data Bases*, I. W. Draffan and F. Poole, Eds. Cambridge University Press, New York, 1980. Also available as Notes on distributed databases. Res. Rep. RJ2571, IBM, San Jose, Calif., July 1979.
20. LOHMAN, G., MOHAN, C., HAAS, L., DANIELS, D., LINDSAY, B., SELINGER, P., AND WILMS, P. Query processing in R*. In *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. Batory, Eds. Springer-Verlag, New York, 1984. Also Res. Rep. RJ4272, IBM, Apr. 1984.
21. MACKERT, L., AND LOHMAN, G. Index scans using a finite LRU buffer: A validated I/O model. Res. Rep. RJ4836, IBM, San Jose, Calif., Sept. 1985.
22. MOHAN, C. *Tutorial: Recent Advances in Distributed Data Base Management.* IEEE catalog number EH0218-8, IEEE Press, New York, 1984.
23. MOHAN, C., STRONG, R., AND FINKELSTEIN, S. Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. In *Proceedings of the 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). ACM, New York, 1983, 89–103. Reprinted in ACM/SIGOPS Operating Systems Review, July 1985. Also Res. Rep. RJ3882, IBM, San Jose, Calif., June 1983.
24. OBERMARCK, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7*, 2 (June 1982), 187–208.
25. ROTHNIE, J. B., JR., BERNSTEIN, P. A., FOX, S., GOODMAN, N., HAMMER, M., LANDERS, T. A., REEVE, C., SHIPMAN, D. W., AND WONG, E. Introduction to a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (Mar. 1980), 1–17.
26. SKEEN, D. Nonblocking commit protocols. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data* (Ann Arbor, Mich., May 1981). ACM, New York, 1981, 133–142.
27. SKEEN, D. A quorum-based commit protocol. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks* (May 1982). Lawrence Berkeley Laboratories, 1982, 69–90.
28. STONEBRAKER, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Softw. Eng. 5*, 3 (May 1979), 235–258.

Received September 1985; revised July 1986; accepted July 1986