Reprinted with permission from the *Proceedings of the Second ACM Symposium on Principles of Distributed Computing.*

Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions

C. Mohan, B. Lindsay

IBM San Jose Research Laboratory
San Jose, CA 95193

ABSTRACT: This paper describes two efficient distributed transaction commit protocols, the **Presumed Abort (PA)** and **Presumed Commit (PC)** protocols, which have been implemented in the distributed data base system R* [DSHLM82, LHMWY83]. PA and PC are extensions of the well-known two-phase (2P) commit protocol [Gray78, Lamp80, LSGGL80]. PA is optimized for read-only transactions and a class of multi-site update transactions, and PC is optimized for other classes of multi-site update transactions. The optimizations result in reduced inter-site message traffic and log writes, and, consequently, a better response time for such transactions. We derive the new protocols in a step-wise fashion by modifying the 2P protocol.

INTRODUCTION

In a distributed data base system, the actions of a transaction (an atomic unit of consistency and recovery [Gray81]) may occur at more than one site. Our model of a transaction, unlike that of some other researchers' [RBFGH80, Ston79], permits multiple data manipulation and definition statements to constitute a single transaction. When a transaction execution starts, the whole transaction need not be already specified and made known to the system. A distributed transaction commit protocol is required in order to insure either that all the effects of the transaction persist or that none of the effects persist, despite site or communication link failures and loss of messages. 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 data base 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 data base of the distributed data base system has a **log** which

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 data base (the **UNDO/REDO** log [GMBLL81, HaRe82]). The log records are written sequentially in a file which is kept in stable (non-volatile) storage [Lamp80].

When a log record is written, the write can be done synchronously or asynchronously. In the former case, which is also called forcing a log record, that log record and all preceding ones are immediately moved from the virtual memory buffers to stable storage. The log writer 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 would have survived the crash and would be available for reading from the stable storage when the site recovers.

On the other hand, in the asynchronous case, the record gets written to the 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 writer 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 it recovers. An important point to note is that a synchronous write increases the response time of the writer 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 [Borr81, Gray78, HaSh80, Lamp80, LSGGL80, MoSF83, Skee81, Skee82]. 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 written synchronously 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 for a commit protocol are: (1) Guaranteed transaction atomicity always, (2) ability to "forget" outcome of commit processing after a while, (3) requirement of minimal overhead in terms of log writes and message sending, (4) optimized performance in the no-failure case, and (5) exploitation of read-only transactions.

While other researchers have concentrated primarily on improving the reliability characteristics of the commit protocols without being too much concerned about the performance aspects, here we concentrate on the latter, especially the logging and communication performance during no-failure situations. While describing the two-phase protocol and the other protocols, we will carefully point out 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.

The rest of this paper is organized as follows. First, we give a careful presentation of the two-phase (2P) protocol. Next, we derive two new protocols, namely Presumed Abort (PA)

and Presumed Commit (PC), from 2P in a step-wise fashion. We conclude by comparing the performance and discussing the trade-offs involved in choosing between PA and PC. We also point out how our protocols have been extended to the contexts in which the transaction execution tree could have more than two levels, as in R* [DSHLM82, HSBDL82, LHMWY83] and ENCOMPASS [Borr81].

THE TWO-PHASE (2P) COMMIT PROTOCOL

In 2P, the model of a distributed transaction execution is such that there is one process, called the **coordinator**, which 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).[2] All the processes together accomplish the function of a distributed transaction.

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, in parallel, ***PREPARE*** messages to the subordinates to determine whether they are willing to commit the transaction.[3] 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. 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 get any more information from the coordinator. 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, force-writes a *commit* record, and sends ***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, force-writes an *abort* record and sends ***ABORT*** messages to (only) all the subordinates that are in the **prepared** state. Each subordinate, after receiving a ***COMMIT*** message moves to the **committing**

---

2 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 successfully implemented in R* [DSHLM82, HSBDL82, LHMWY83], where this assumption is relaxed to permit more than one such process per site.

3 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 gets resubmitted after being aborted it is given a new name.

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

By requiring the subordinates to send *ACK*s, the coordinator makes sure 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* message. 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 and the names of the 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 2 records (*prepare* and *commit*, both of which are forced) and sends 2 messages (*YES VOTE* and *ACK*). The coordinator sends 2 messages (*PREPARE* and *COMMIT*) to each subordinate and writes 2 records (*commit*, which is forced, and *end*, which is not).

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.[4] It is this information in virtual storage that is used to answer queries from other sites about transactions which had their coordinators at this site and to send unsolicited information to other sites which had subordinates for transactions that had their coordinators at this site. Having the 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.

---

4 The extent of the log that has to be read on restart can be controlled by taking "checkpoints" during normal operation [GMBLL81, HaRe82]. The log is scanned forward starting from the last checkpoint before the crash until the end of the log.

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*** message). 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.[5] 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 ***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 some transactions during normal operation by local coordinator and subordinate processes which notice some link or remote site failures during the commit protocol (see [LHMWY83] for information relating to how and when 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 had voted yes and gotten 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 gone 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 which says that 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*** messages, (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 inquiree cannot tell whether it is a coordinator or subordinate, since no commit protocol log records exist for the transaction.

---

5 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 ***COMMIT*** messages 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 (some sites might have committed and others may abort) being taken at restart.

Given this fact, the correct response to an inquiry in the no information case is an ***ABORT*** message.

THE PRESUMED ABORT (PA) PROTOCOL

In the last section 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 writes an *abort* record.[6] 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 data base, while the others do. A transaction is (completely) read-only if no process performs any updates. We do not need to know before the transaction starts whether it is read-only or not.[7] If a subordinate receives a ***PREPARE*** message 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*** message by the coordinator.

There will not be a second phase of the protocol if the coordinator is read-only and gets only ***READ VOTE***s. In this case the coordinator, just like the subordinates, writes no log records for the transaction. On the other hand, if the coordinator or one of the subordinates votes ***YES*** and none of the others vote ***NO***, then the coordinator behaves as in 2P. But note that it is sufficient for the coordinator 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 be sent ***COMMIT*** messages). If the coordinator or one of the subordinates votes ***NO*** then the coordinator behaves as described earlier in this section.

---

6 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.

7 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 data base state.

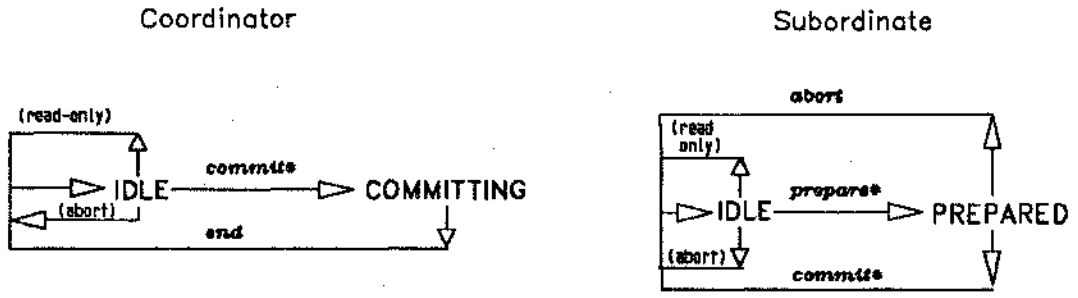
To summarize, for a (completely) read-only transaction, neither the coordinator nor any of the subordinates writes any log records, but each one of the subordinates sends 1 message (*READ VOTE*) and the coordinator sends 1 message (*PREPARE*) to each subordinate.

For a committing partially read-only transaction, the coordinator sends 2 messages (*PREPARE* and *COMMIT*) to update subordinates and 1 message (*PREPARE*) to the others, and it writes 2 records (*commit*, which is forced, and *end*, which is not) if there is at least one update subordinate and only 1 record (*commit*, which is forced), otherwise. A read-only subordinate behaves just like the one in a completely read-only transaction, and an update subordinate behaves like a subordinate of a committing transaction in 2P.

By making the above changes to 2P, we have generated the **Presumed Abort** (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* message. Figure 1 shows the state transitions and log writes performed by the coordinator and subordinate processes following PA.

THE PRESUMED COMMIT (PC) PROTOCOL

Since most transactions are expected to commit, it is only natural to wonder if, by requiring *ACK*s to *ABORT* messages, commits could be made cheaper by eliminating the *ACK*s to *COMMIT* messages. A simplistic idea that comes to mind is to require that *ABORT*s be



State Changes and Log Writes
for Presumed Abort

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 parenthesis indicate under what circumstances such transitions take place. IDLE is the initial and final state for each process.

Figure 1

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*** message when a subordinate inquires. However, there is a problem with this approach.

Consider the situation when a coordinator has sent the ***PREPARE*** messages, one subordinate has gone into the **prepared** state, and before the coordinator is able to collect all the votes and make a decision, the coordinator crashes. Note that so far the coordinator would not have written any commit protocol log records. When the crashed coordinator'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 coordinator's site, its recovery process would respond with a ***COMMIT***[8] message, causing an unacceptable inconsistency.

The way out of this problem is for the coordinator to record the names of the subordinates safely **before** any of them 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*** messages, the restart process will know whom to inform (and get ***ACK***s) about the abort. These modifications give us the **Presumed Commit** (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*** message.

In PC, the coordinator behaves as in PA except: (1) at the start of the first phase (i.e., before sending the ***PREPARE*** messages) it force-writes a *collecting* record, which contains the names of all the subordinates, and moves into the **collecting** state; (2) it force-writes both *commit* and *abort* records; (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 may 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*** message.

To summarize, for a (completely) read-only transaction, the coordinator writes 2 records (*collecting*, which is forced, and *commit*, which is not) and sends 1 message (***PREPARE***) to

---

8 Note that as far as the recovery process is concerned, this situation is the same as when a coordinator, 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*** message.

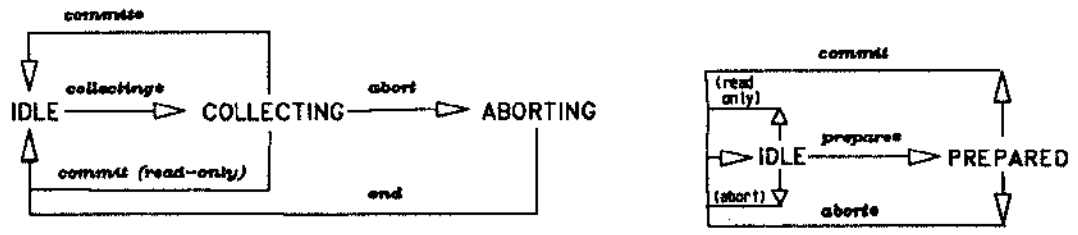
each subordinate. The subordinates write no log records, but each one of the subordinates sends 1 message (***READ VOTE***).

For a committing partially read-only transaction, the coordinator sends 2 messages (***PREPARE*** and ***COMMIT***) to update subordinates and 1 message (***PREPARE***) to the others, and it writes 2 records (*collecting* and *commit*, both of which are forced). A read-only subordinate behaves just like the one in a completely read-only transaction and an update subordinate sends 1 message (***YES VOTE***) and writes 2 records (*prepare*, which is forced, and *commit*, which is not).

Figure 2 shows the state transitions and log writes performed by the coordinator and subordinate processes following PC.

DISCUSSION

In the table of Figure 3 we summarize the performance of 2P, PA, and PC with respect to committing update and read-only transactions. 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 2 log writes, including a force) and in the case of partially read only transactions in which only the coordinator does any updates (saving the



State Changes and Log Writes
for Presumed Commit

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 parenthesis indicate under what circumstances such transitions take place. IDLE is the initial and final state for each process.

Figure 2

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

Figure 3: Comparison of Log I/O & Messages for Committing Transactions With 2P, PA and PC

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 data base, 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 system-wide basis) at the time of the start of the first phase by the coordinator.[9] 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.

EXTENSIONS

In the distributed data base systems R* [DSHLM82, HSBDL82, LHMWY83] and ENCOMPASS [Borr81], a distributed transaction execution is carried out by a multi-level (not just a two-level) tree of processes. Our protocols are easily extended to this model. Non-root, non-leaf nodes act as coordinators as well as subordinates, while leaf nodes act only as subordinates, and the root node acts only as coordinator. Each process communicates directly with only its immediate neighbors in the tree, i.e. father and sons. In fact, a process would not even know about the existence of its non-neighbor processes.

The root coordinator and leaf subordinates act the same way as before. On receiving a ***PREPARE***, each non-root coordinator instead of sending its vote immediately has to solicit the votes of its subordinates and decide on the vote for the subtree for which it is the root, and then send this vote to its coordinator. This decision is similar to what the root coordinator does when it has all the votes. If the vote is a ***NO VOTE***, then it has to abort its prepared subordinates. Each non-root coordinator has to also propagate a ***COMMIT/ABORT*** received from its coordinator to its subordinates, possibly after sending an ***ACK*** to its coordinator. The details of these extensions are easily derived. These extended protocols are the ones that have been implemented and are operational in R*.

It should be pointed out that our commit protocols are blocking [Skee81] 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. In [MoSF83] we have proposed an approach to dealing with this problem in the context of the Highly Available Systems project in our Laboratory.

ACKNOWLEDGEMENT

We would like to convey our thanks to Guy Lohman for his detailed editorial comments.

REFERENCES

Borr81 Borr, A. "Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing", *Proc. International Conference on Very Large Data Bases*, September 1981.

Coop82 Cooper, E. "Analysis of Distributed Commit Protocols", *Proc. SIGMOD Int. Conf. on Management of Data*, June 1982.

---

9 If this approach is taken (as we have done in R*), then the coordinator 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.

DSHLM82 Daniels, D., Selinger, P., Haas, L., Lindsay, B., Mohan, C., Walker, A., Wilms, P. "An Introduction to Distributed Query Compilation in R*", *Proc. Second International Symposium on Distributed Data Bases*, Berlin, September 1982. Also IBM Research Report RJ3497.

GMBLL81 Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., Traiger, I. "The Recovery Manager of the System R Database Manager", *ACM Computing Surveys*, Vol. 13, No. 2, June 1981.

Gray78 Gray, J. "Notes on Data Base Operating Systems", In *Operating Systems - An Advanced Course*, Lecture Notes in Computer Science, Volume 60, Springer-Verlag, 1978.

Gray81 Gray, J. "The Transaction Concept: Virtues and Limitations", *Proc. Seventh Int. Conf. on Very Large Data Bases*, October 1981.

HaRe82 Harder, T., Reuter, A. "Principles of Transaction Oriented Database Recovery - A Taxonomy", Technical Report 50/82, University of Kaiserslautern, W. Germany, April 1982.

HaSh80 Hammer, M., Shipman, D. "Reliability Mechanisms for SDD-1", *ACM Transactions on Data Base Systems*, December 1980.

HSBDL82 Haas, L.M., Selinger, P.G., Bertino, E., Daniels, D., Lindsay, B., Lohman, G., Masunaga, Y., Mohan, C., Ng, P., Wilms, P., Yost, R. "R*: A Research Project on Distributed Relational DBMS", *Database Engineering*, Volume 5, Number 2, December 1982. Also IBM Research Report RJ3653, October 1982.

Lamp80 Lampson, B. "Atomic Transactions", Chapter 11 in *Distributed Systems - Architecture and Implementation*, B. Lampson (Ed.), Lecture Notes in Computer Science Vol. 100, Springer Veralg, 1980.

LHMWY83 Lindsay, B., Haas, L., Mohan, C., Wilms, P., Yost, R. "Computation and Communication in R*: A Distributed Database Manager", To Appear in *Proc. 9th ACM Symposium on Operating Systems Principles*, Bretton Woods, October 1983. Also IBM Research Report RJ3740, January 1983.

LSGGL80 Lindsay, B., Selinger, P., Galtieri, C., Gray, J., Lorie, R., Putzolu, F., Traiger, I., Wade, B. "Single and Multi-Site Recovery Facilities", In *Distributed Data Bases*, Edited by I.W. Draffan and F. Poole, Cambridge University Press, 1980. Also Available as "Notes on Distributed Databases", IBM Research Report RJ2571, San Jose, July 1979.

MoSF83 Mohan, C., Strong, R., Finkelstein, S. "Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors", *Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed. Computing*, Montreal, Canada, August 1983. Also IBM Research Report RJ3882, June 1983.

RBFGH80 Rothnie, J.B., Bernstein, P.A., Fox, S., Goodman, N., Hammer, M., Landers, T., Reeve, C., Shipman, D., Wong, E. "Introduction to a System for Distributed Databases (SDD-1)", *ACM Transactions on Database Systems*, Vol. 5, No. 1, March 1980.

Skee81 Skeen, D. "Nonblocking Commit Protocols", *Proc. ACM/SIGMOD International Conference on Management of Data*, Ann Arbor, Michigan, 1981, pp. 133-142.

Skee82 Skeen, D. "A Quorum-Based Commit Protocol", *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, May 1982, pp. 69-90.

Ston79 Stonebraker, M. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", *IEEE Transactions on Software Engineering*, Vol. 5, No. 3, May 1979.