The Presumed-Either Two-Phase Commit Protocol

Gopi K. Attaluri and Kenneth Salem

Abstract—This paper describes the presumed-either two-phase commit protocol. Presumed-either exploits log piggybacking to reduce the cost of committing transactions. If timely piggybacking occurs, presumed-either combines the performance advantages of presumed-abort and presumed-commit. Otherwise, presumed-either behaves much like the widely-used presumed-abort protocol.

Index Terms—Two-phase commit protocol, distributed transaction, presumed-abort, presumed-commit, atomicity.

1 Introduction

COMMIT protocols are used to coordinate the participants in a distributed transaction so that all agree on the transaction's outcome. Presumed-abort and presumed-commit are the two best-known two-phase commit protocols [7]. Presumed-abort is widely implemented. It handles aborting transactions efficiently and requires only a single forced log write at the coordinator in the common case that the transaction commits. It also handles read-only transactions efficiently. Presumed commit requires fewer messages than presumed-abort to commit a transaction, but also requires an extra forced log write at the coordinator.

In the original paper that described presumed-abort and presumed-commit, it was noted briefly that the two protocols might be used side by side, with the protocol choice being made on a transaction-by-transaction basis [7]. This idea is also the basis of presumed-either. In presumed-either, the presumed-abort/presumed-commit choice is based on whether a list of the processes participating in the transaction can be piggybacked into nonvolatile storage before the commit protocol begins. Like presumed-abort, presumed-either commits read-write transactions using one forced write into the transaction manager's log. Read-only transactions require none. In addition, presumed-either eliminates the need for commit acknowledgement messages for those transactions for which piggybacking succeeds.

2 Two-Phase Commit

A distributed transaction will be assumed to involve a tree of processes, as described in [4], [7]. Neighboring processes can communicate by exchanging messages. Processes are of two types. Resource managers (RMs), e.g., database management systems, are at the leaves of the tree and it is at the RMs that the actual work of the transaction is performed. Processes at internal nodes or at the root are transaction managers (TMs). TMs participate in the commit protocol to ensure that all processes agree on the transaction's outcome. The protocol is initiated by the TM at the root of the process tree.

A more general peer-to-peer process model also exists and commit protocols for it have been described in [5], [6], [8]. The peer-to-peer model will not be considered further in this paper.

- *G.K. Attaluri is with IBM Canada Ltd., 1150 E. Eglinton Ave. Toronto, Ontario M3C 1H7 Canada. E-mail:gkattalu@hotmail.com.*
- *K. Salem is with the Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1 Canada. E-mail: kmsalem@uwaterloo.ca.*

Manuscript received 18 May 2000; revised 18 Apr. 2001; accepted 19 Apr. 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104341.

The process hierarchy for a particular transaction may be formed dynamically as the transaction executes. For example, the XA interface defined as part of the X/Open distributed transaction processing standard includes a "Join" method through which a process notifies a TM that it is a participant in an ongoing transaction [2]. The details of the mechanism by which processes join the tree are not particularly important to the correct execution of the commit protocol, so long as each TM has been made aware of its children prior to the protocol's initiation.

Conceptually, each process has associated with it a log, which is a read/append data structure. Each log is divided into two parts. The log tail exists in volatile storage and contains the most recently appended log data. The remainder of the log exists in nonvolatile storage. A process may fail at any time during the execution of the transaction or the commit protocol. A failure destroys contents of that process' log tail.

A *log write* operation appends data to the log tail. A *forced log write* operation appends data to the log tail and then moves the contents of the log tail into nonvolatile storage. Data that are already in the log tail at the time of a forced write are said to have been *piggybacked* into the nonvolatile log by the forced write operation. Since nonvolatile storage is often implemented using relatively slow block-oriented devices such as disks, forced log write operations are considered to be expensive, while nonforced write operations are not.

The primary goal of a two-phase commit protocol is to ensure that all processes agree on whether a transaction commits or aborts. This should be accomplished with as few messages as possible and as few log writes, particularly forced writes, as possible. The protocol should also allow a participating process to "forget" a transaction as quickly as possible once it has learned the transaction's outcome. This means that the process is free to discard information concerning that transaction without jeopardizing the ability of others to learn of the outcome.

Comprehensive descriptions of the presumed-abort and presumed-commit protocols can be found in [7]. Assuming that a transaction's process tree consists of a single TM and one or more RMs and that the transaction commits, the presumed-abort protocols operates as follows: The TM sends `Prepare` messages to each RM, which responds by force writing `Prepare` records into its log and then sending a positive vote to the TM. The `Prepare` log record indicates that an RM has agreed to not make a unilateral commit or abort decision and has ceded control of the transaction to the coordinating TM. Once it has collected all of the votes, the TM force writes a `Commit` record into its log and sends a `Commit` message to each participant. Each RM then forces a `Commit` record into its log and acknowledges the TM's `Commit` message. The RMs can forget the transaction once they have written their `Commit` record. The TM forgets once it has received an acknowledgement from every RM.

The presumed-commit is similar, except that the TM must force-log a `Participant` record[1] before sending the `Prepare` messages and the RMs do not have to force their `Commit` records and do not acknowledge the `Commit` message. The TM can forget the transaction as soon as it has logged its `Commit` record and sent `Commit` messages to the RMs. The `Participant` record is needed in case the TM fails after sending out `Prepare` messages, but before it has collected all of the votes and logged a decision. The TM must abort the transaction in this case and it must obtain acknowledgements from all of the RMs before forgetting the transaction. The `Participant` record contains a list of the RMs involved in the transaction so that the TM can determine when all of them have acknowledged.

1. This was called a `Collection` record in [7].

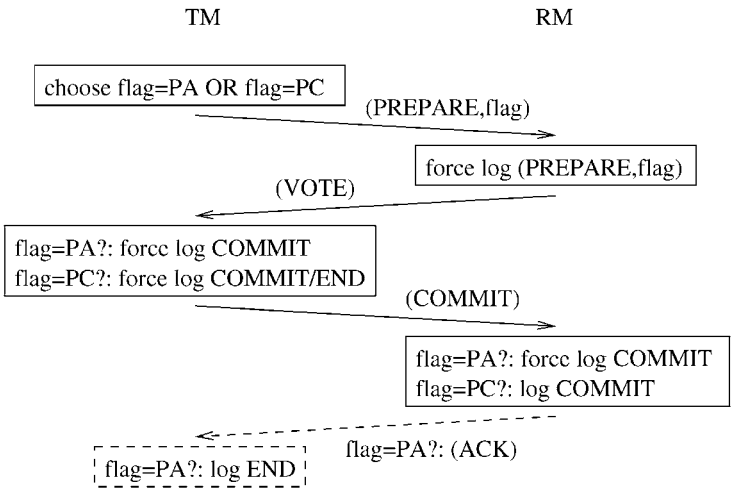

Fig. 1. The presumed-either protocol commit scenario.

3 The Presumed-Either Protocol

In this section, the presumed-either protocol is described under the assumption that the process tree consists of a single TM and one or more RMs. The more general case is considered in Section 4.

The main idea behind the presumed-either protocol is simple: Try to piggyback the list of participating RMs into the TM's log prior to the initiation of commit processing, i.e., before the `Prepare` messages are sent. If this can be accomplished, the presumed-either protocol operates like presumed-commit, but without the need to force an initial `Participant` record into the log, since all of the necessary information is already there. If piggybacking cannot be accomplished, presumed-either operates much like presumed-abort. This flexibility allows presumed-either to combine the advantages of presumed-abort and presumed-commit. Like presumed-abort, presumed-either never requires an initial forced log write. Like presumed-commit, presumed-either can avoid acknowledging `Commit` messages, provided that piggybacking is successful.

The main difficulty with this idea is that the commit protocol's operation may vary from transaction to transaction since piggybacking may be successful for some transactions but not for others. Presumed-either must maintain the distinction between these two types of transactions to ensure that all transactions are properly terminated and forgotten. The remainder of this section presents the presumed-either protocol in more detail and describes how it accomplishes this.

3.1 Logging Transaction Participants

Under presumed-either, a transaction manager logs `Participant` records during the execution of a transaction before commit processing begins. Specifically, whenever the TM learns that an RM has joined the transaction, it immediately logs a `Participant` record containing the identity of the newly-joined RM. Thus, each transaction's participants will, in general, be described by a set of `Participant` records rather than a single record as was the case in the presumed-commit protocol.

The `Participant` log writes are not forced. As a result, a failure of the TM may result in the loss of some `Participant` records. However, it is also possible that `Participant` records may be piggybacked into the nonvolatile log. For example, this may occur because a log page fills up or because the commit or abort of some other transaction causes a forced log write.

The TM maintains a flag that controls how the commit protocol operates. There is a separate flag for each transaction. When the TM is ready to begin the commit processing for a transaction, it must choose one of two values, `PA` or `PC`, for that transaction's flag. If the TM desires to commit the transaction and all of that transaction's `Participant` records have been successfully piggybacked, the TM chooses `PC` as the flag value. Otherwise, it chooses `PA`. The TM can use log sequence numbers [2] to determine whether or not a transaction's `Participant` records have been piggybacked.

3.2 Committing and Aborting Transactions

Commit processing under the presumed-either protocol is illustrated in Fig. 1 and Fig. 2. Fig. 1 illustrates the scenario in which all RMs vote to commit the transaction. Fig. 2 illustrates the scenario in which some RM votes to abort it. The TM's interaction with a single RM is shown and that RM is assumed to have cast a positive vote in both scenarios. Similar interactions occur between the TM and the remaining RMs. In both figures, the notation `condition?:action` is used to describe a conditional action. The execution of those portions of the protocol that are drawn using dashed lines is conditional, depending on the state of the protocol flag.

As shown in Figs. 1 and 2, the flag value that has been chosen for the transaction is communicated to each participating RM by including it in the `Prepare` messages. Each RM includes the flag in the `Prepare` log record that it forces to nonvolatile memory before sending its vote to the coordinator. The TM itself does not log the flag value.[2]

When the flag is set to `PC`, the second phase of presumed-either is similar to the presumed-commit protocol's second phase. That is, the TM requires acknowledgments for the transaction if it is aborted, but does not require them if the transaction commits. If the flag is set to `PA`, presumed-either is similar to the presumed-abort protocol, and only `Abort` messages need to be acknowledged.

Both scenarios shown in Figs. 1 and 2 assume that the TM wishes to commit the transaction. In some cases, the TM may desire from the outset to abort the transaction. In that case, the first phase of the protocol can be skipped. The TM sets the flag to `PA`, regardless of whether the transaction's `Participant` records are in stable storage. It then executes the second phase of the protocol. That is, it sends `Abort` messages to all participants and then forgets the transaction. Upon receiving the `Abort` messages, the RMs write unforced `Abort` records into their logs. This behavior is

2. A similar mechanism was suggested in [7] to allow different transactions to use different protocols. In that case, however, the protocol flag was to be logged at the TM as well as the RMs.

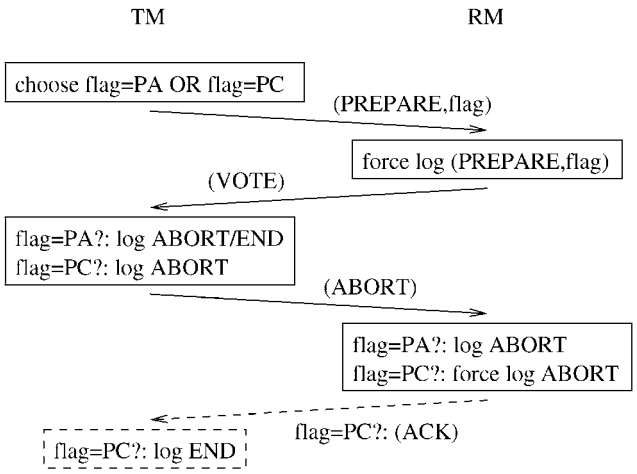

Fig. 2. The presumed-either abort scenario.

Scenario 1: TM commits the transaction.

Protocol	Messages			Log Forces		
	TM	RM	Total	TM	RM	Total
Presumed-either, flag=PC	2	1	3	1	1	2
Presumed-either, flag=PA	2	2	4	1	2	3
Presumed-abort	2	2	4	1	2	3
Presumed-commit	2	1	3	2	1	3

Scenario 2: TM aborts the transaction.

Protocol	Messages			Log Forces		
	TM	RM	Total	TM	RM	Total
Presumed-either, flag=PC	2	2	4	0	2	2
Presumed-either, flag=PA	2	1	3	0	1	1
Presumed-abort	2	1	3	0	1	1
Presumed-commit	2	2	4	1	2	3

Fig. 3. Summary of logging and message costs of presumed-either, presumed-abort, and presumed-commit.

identical to that of the presumed-abort protocol in the same situation.

The tables in Fig. 3 summarize the costs of the presumed-either protocol for the two scenarios shown in Figs. 1 and 2. Two costs for each protocol are tabulated: the number of forced log writes and the number of messages sent. Fig. 3 also shows the costs of the presumed-abort and presumed-commit protocols.

The costs for presumed-either depend on the TM's success at piggybacking `Participant` records. Like presumed-abort, presumed-either always requires a single coordinator log force to commit a transaction. Unlike presumed-abort, however, presumed-either can avoid acknowledgement messages when piggybacking is successful. Presumed-either always requires fewer log forces than presumed-commit, though it may require an acknowledgement message that presumed-commit does not. The more effective piggybacking is, the closer presumed-either will come to matching presumed-commit's message count.

3.3 The Presumed-Either Flag Revisited

In Section 3.2, it was stated that the TM sets the protocol flag to `PC` when it wants to commit a transaction whose `Participant` records have been piggybacked and to `PA` otherwise. In fact, the TM has some additional leeway in making this decision. The rule is that the TM *may* choose a flag value of `PA` for any transaction and it *must* choose a flag value of `PA` if at least one of the transaction's `Participant` records has not been piggybacked.

Fig. 3 shows that presumed-either can commit a transaction most efficiently if it chooses a flag value of `PC` and can abort a transaction most efficiently if it chooses a flag value of `PA`. If the TM wants to commit the transaction and if the most likely outcome of the commit protocol is that the transaction will, in fact, commit, then it makes sense for the TM to set the flag to `PC` whenever the flag rule allows it to do so. However, if the TM has some advance knowledge that a transaction is likely to be aborted because of a negative vote from an RM, the TM can and should choose a flag value of `PA`, even if piggybacking is completely successful.

3.4 Failure Recovery

The TM and the RMs may fail at any time while presumed-either is being used to terminate a transaction. Recovery from RM failures is straightforward. For any unresolved (prepared but neither committed nor aborted) transaction, the RM may send a `Query` message to the TM. This message carries the flag value that was logged with the transaction's `Prepare` record. If the TM has not forgotten the transaction, it can respond with a `Commit` or `Abort` message, as appropriate. Otherwise, the TM uses the `Query`'s flag value to determine its response: it sends `Commit` if the flag is `PC` and `Abort` if the flag is `PA`.

Recovery at the TM is less straightforward because the coordinator does not log its flag value. As usual, the recovering TM checks its log to determine which transactions were in progress at the time of the failure. For some transactions, the TM may find a `Commit` record but no `End`. In this case, the transaction's flag must have been `PA`. Furthermore, since `Participant` records will have been piggybacked by the forced `Commit` record (if not sooner), the TM will be able to determine the complete set of participants from the log. In this case, it simply redoes the second phase of the protocol by sending `Commit` messages to the participants and waiting for acknowledgements. For transactions for which the TM finds an `Abort` record but no `End`, the situation is similar. In that case, the flag must have been `PC`, which implies that all `Participant` records had been piggybacked prior to the start of the commit protocol. The TM sends `Abort` messages to the participants and awaits acknowledgements.

A more difficult situation arises if the TM finds transactions with `Participant` records but no `Abort` or `Commit` record. The TM must abort the transaction, however, there are several problems. The TM does not know whether the list of logged participants is partial or complete; some `Participant` records may not yet have been piggybacked to the disk at the time of the failure. The TM may or may not have chosen a flag value before the failure and sent it to one or more RMs in `Prepare` messages. Since the flag value is not logged at the TM, this value, if any, is unknown. Furthermore, there may be RMs that are in doubt about the transaction's outcome.

How should the TM recover? There are two scenarios to consider:

1. All `Participant` records may be in the log.
2. `Participant` records for some RMs may be missing from the log because they were still in the log tail at the time of the failure.

The following procedure correctly handles both cases. The TM chooses a flag value of `PC`, sends `Abort` messages containing this flag to all *known* participants, and awaits acknowledgements. When all known participants acknowledge, the TM logs `End` and forgets the transaction. This works in the first case because all participants are known. Once all participants have acknowledged, the transaction can safely be forgotten because it will have been resolved at all participants. In the second case, it works correctly for all known participants for the same reason. If the transaction remains in doubt at any unknown participants, those participants

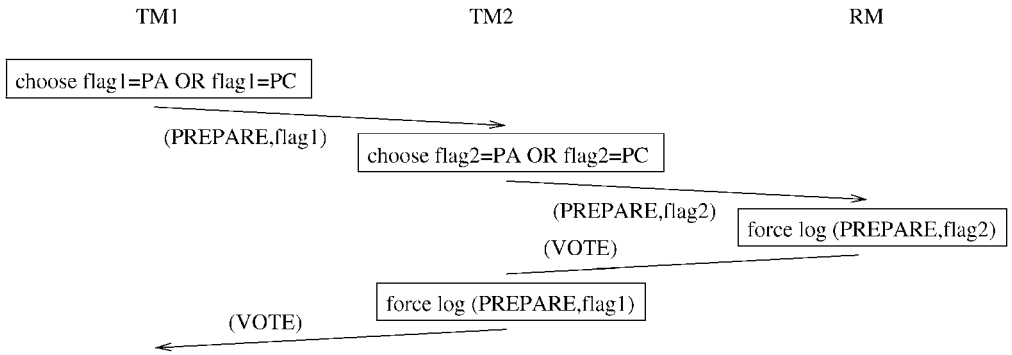

Fig. 4. Multilevel presumed-either protocol, phase one.

must have logged a flag value of PA. This is because the TM cannot have chosen a flag value of PC if some Participant records had not been piggybacked into nonvolatile storage. If such an unknown participant attempts to resolve the transaction, its Query will therefore carry a PA flag. Thus, even if the Query arrives after the TM has forgotten the transaction, the PA flag it carries will cause the TM to respond properly with an Abort message.

Normally, the flag value carried in an Abort or Commit message will match the flag value logged by an RM when it prepares a transaction. However, failures may cause them to differ. In particular, if the TM elects to use a PA flag and then fails before making a commit or abort decision, it will send Abort messages containing a PC flag when it recovers. For this reason, an RM's response to a Commit or Abort message is based on the flag value carried in that message and not on the flag value logged with the RM's Prepare record.

4 Multilevel Presumed-Either

So far, presumed-either has been presented under the assumption of a two-level process tree in which each transaction involves a single TM. Like other commit protocols, presumed-either can also operate in a multilevel process hierarchy. A TM at an internal node acts as an RM to its parent TM and as a TM to its children. Each process is normally aware only of the processes immediately above and below it in the tree.

In multilevel presumed-either, the root TM and the RMs behave exactly as shown in Figs. 1 and 2. Fig. 4 illustrates the behavior of an internal TM during the first phase of the presumed-either protocol. In the multilevel protocol, all TMs, including internal ones, write Participant records to their logs during transaction execution. The Participant records written by a TM describe only its immediate children. Each internal TM has its own per-transaction flag, which it sets to PA or PC depending on whether its own Participant records have been piggybacked at the time it receives the Prepare message. Thus, it is possible that some TMs in the hierarchy will set their flags to PA while others choose PC.

As shown in Fig. 4, when an internal TM logs its Prepare message it includes the flag value sent by its parent TM and not its own flag. In Fig. 4 (and Fig. 5), flag1 refers to the flag value chosen by the parent TM (TM1) while flag2 refers to the flag chosen by the child, TM2.

Fig. 5 illustrates the second phase of the multilevel presumed-either protocol. The four scenarios illustrate the possible behaviors that may arise depending on the coordinator's choice of flag value and its commit decision. Essentially, an internal TM acts as a TM to its children and as an RM to its parent. However, note that an internal TM need only force write its Commit/Abort log record if its parent requires an acknowledgment. Unlike the root TM, it need not force the record prior to forwarding the transaction's outcome to its children.

Fig. 6 summarizes the costs of multilevel presumed-either for the two extreme cases in which all TMs choose the same flag value. When different TMs choose different values, performance will fall between the extremes. Fig. 6 also shows the costs of multilevel presumed-abort and presumed-commit. When piggybacking is successful, presumed-either can commit a transaction in an n-process tree with one forced write per process.

4.1 Read-Only Transactions

An RM is a read-only participant in a transaction if it has not made any changes to its local data on behalf of the transaction. A TM is read-only if all of its subordinate TMs and RMs are read-only. To a read-only participant, it makes no difference whether a transaction commits or aborts.

In the presumed-abort protocol described in [7], a read-only process that receives a Prepare message votes Read-Only to its parent TM. It performs no logging and does not participate at all in the second phase of the protocol. Similarly, a TM that receives Read-Only votes from all of its children votes Read-Only to its parent (if it has one) and avoids the second phase of the protocol. In the event that all participants in a transaction are read-only, the entire second phase of the protocol is avoided. The net effect is that no logging at all is required to terminate a completely read-only transaction using presumed-abort [7], [8]. Since read-only transactions are common in many systems, this is a very desirable property.

The same technique can be applied to the PE protocol. As is the case for read-only presumed-abort, read-only presumed-either requires no logging at all during the execution of the commit protocol itself. This is true whether the TMs set the protocol flag to PA or to PC. However, under presumed-either, TMs attempt to log Participant records while the transaction is still active, in the hope that they will be piggybacked before the commit protocol begins. Assuming that a transaction is not known to be read-only until the end of the first phase of the commit protocol, TMs will still log Participant records for read-only transactions under presumed-either. Thus, it is more accurate to say that, for read-only transactions, read-only presumed-either requires no forced log writes at the TMs, while read-only presumed-abort requires no logging at all.

5 Further Optimizations

A number of further optimizations of presumed-abort and presumed-commit have been described. This section presents several of the major ones and and their relationships to presumed-either.

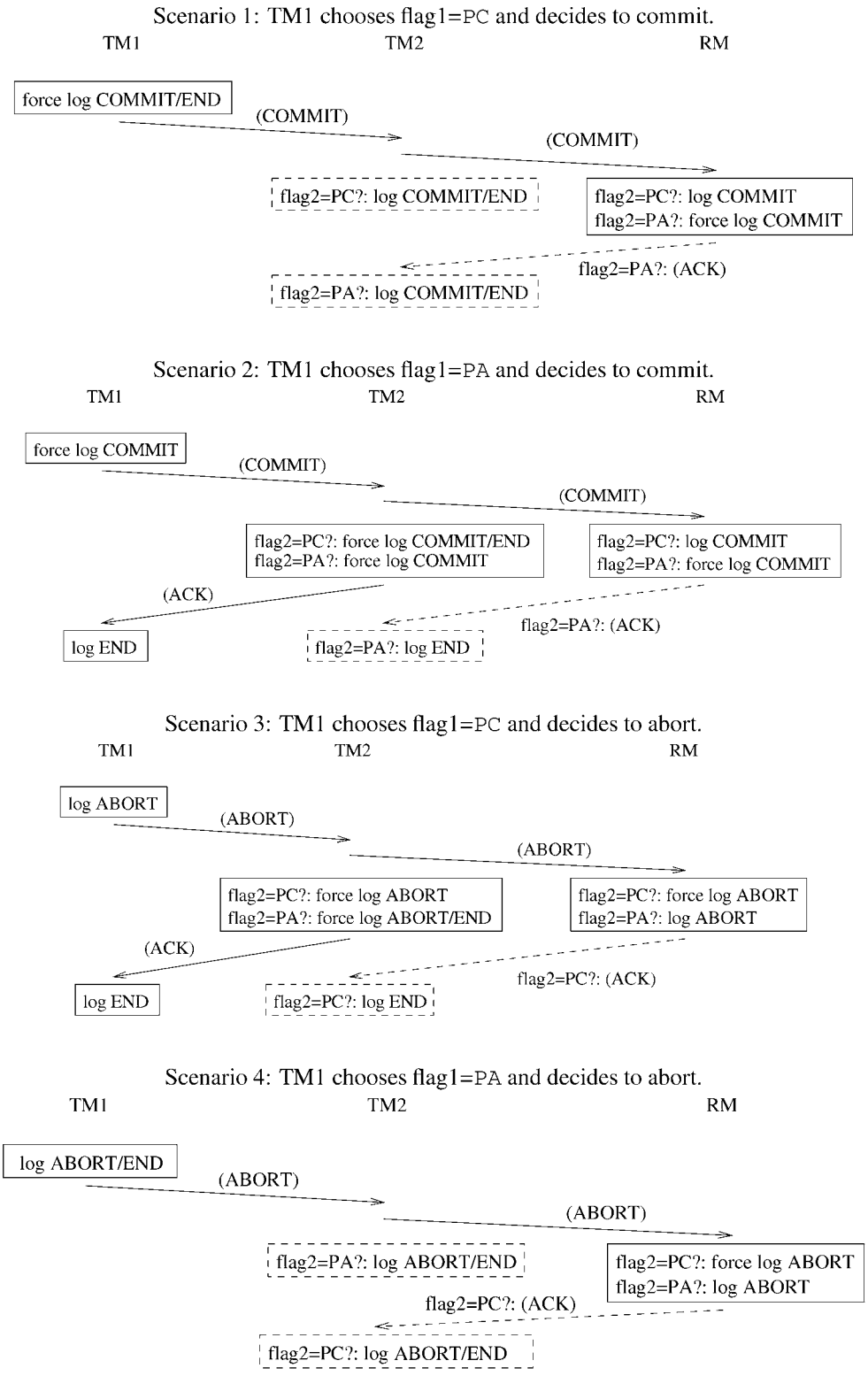

Fig. 5. Multilevel presumed-either protocol, phase two.

5.1 Presumed Commit with Bulk Transaction Resolution

Lampson and Lomet have developed a presumed-commit optimization that retains that protocol's main advantage (no `Commit` acknowledgments) while eliminating its main disadvantage (two log forces required at the coordinator) [3]. They point out that, for a transaction that is aborted due to a failure at a TM, the first presumed-commit log force at the TM serves two purposes at crash recovery: It identifies the transaction as an aborted transaction for which acknowledgments are required and it identifies the participants that must acknowledge the abort decision. The optimized protocol addresses these two functions differently. It assumes that the transaction identifiers assigned to new transactions increase monotonically. At any time, the set of available transaction identifiers is confined to a fixed range and all identifiers within that range are implicitly logged. In the event of

Scenario 1: TM commits the transaction.

Protocol	Total Messages	Total Log Forces
Presumed-either (All TMs choose flag=PA)	$2n + 2c - 2$	$2n - 1$
Presumed-either (All TMs choose flag=PC)	$n + 2c - 1$	n
Presumed-abort	$2n + 2c - 2$	$2n - 1$
Presumed-commit	$n + 2c - 1$	$n + c$

Scenario 2: TM aborts the transaction.

Protocol	Total Messages	Total Log Forces
Presumed-either (All TMs choose flag=PA)	$n + 2c - 1$	$n - 1$
Presumed-either (All TMs choose flag=PC)	$2n + 2c - 2$	$2n - 2$
Presumed-abort	$n + 2c - 1$	$n - 1$
Presumed-commit	$2n + 2c - 2$	$2n + c - 2$

Fig. 6. Summary of logging and message costs of multilevel presumed-either, presumed-abort, and presumed-commit. There are n processes in the tree, including c TMs and $n - c$ RMs.

a failure, the TM resolves any transaction within that range as aborted and in need of acknowledgement, unless it has a `Commit` or `End` entry in the log. Because the participants in these aborted transactions cannot be determined (from the log or otherwise), the coordinator cannot demand acknowledgments from them. The coordinator cannot forget such transactions. Therefore, their identifiers are recorded in nonvolatile storage and are retained indefinitely.

What Lampson and Lomet achieve using monotonic and persistent transaction identifiers, the presumed-either protocol attempts to achieve by piggybacking `Participant` records into the log. The performance of the optimized presumed-commit protocol is the same as that of presumed-either when piggybacking is successful. Both require a single log force at the TM and neither requires commit acknowledgements. Ideally, the overall performance of presumed-either will approach that of optimized presumed-commit. However, when piggybacking fails, presumed-either requires `Commit` acknowledgements that the optimized presumed-commit does not. On the other hand, presumed-either does not require that transaction identifiers increase monotonically, nor does it require that certain transaction identifiers be retained indefinitely in nonvolatile storage.

5.2 Rooted Multilevel Presumed Commit

The rooted multilevel presumed commit protocol is an optimized multilevel presumed-commit protocol that eliminates some forced log writes for TMs at internal nodes in the process tree [1]. Like presumed-either and the optimized presumed-commit described in the previous section, rooted-presumed-commit seeks to eliminate the forced logging of the `Participant` record that occurs in presumed-commit. The approach taken by the rooted-presumed-commit protocol is to use the root TM to log the identities of all processes in the tree so that internal TMs need not log `Participant` records.

In rooted-presumed-commit, the root TM's forced `Participant` record includes the identities of all other processes in the tree and not just its immediate descendants. In addition, every other process in the tree must include a list of all of its ancestors in its `Prepare` log record. Rooted-presumed-commit also differs from multilevel presumed-commit during failure recovery. When an internal TM receives a `Query` from a subordinate about a transaction that it has forgotten, it does not simply respond with a `Commit` message, as would be the case in multilevel presumed-commit. Instead, it must pass the `Query` up to its own parent, which may or may not remember the transaction.

By eliminating forced log writes at internal TMs, the rooted-presumed-commit protocol can commit a transaction in an n-process tree using $n + 1$ log forces. This may be substantially less than the $n + c$ log forces required by regular multilevel presumed-commit and it is only one more than the number required by presumed-either in the best case, when all TMs choose a flag value of `PC`. The extra forced log write occurs at the root TM, where rooted-presumed-commit requires two forced log writes and presumed-either requires one.

A rooted-presumed-either protocol would also be possible. In such a protocol, the root TM would behave much as it does in presumed-either, while the remaining processes would behave as they do in rooted-presumed-commit. All processes in the tree would propagate their identities to the root TM as in the rooted-presumed-commit protocol. If the root TM is able to piggyback the identities of all processes into its log before the commit protocol begins, it may choose its flag value to be `PC`, otherwise it must choose `PA`. Internal TMs would not choose flag values. Instead, the root TM's flag value would be propagated down the tree to the RM. Such a protocol would be able to commit a transaction using n forced log writes if the root TM chooses a presumed-commit flag and $n + 1$ forced log writes if it chooses PA. Thus, the worst-case number of forced log writes would be the same as the number required by rooted-presumed-commit. However, the rooted-PE protocol would also inherit some of the disadvantages of rooted-presumed-commit. In particular, the root TM must be aware of all other processes in the tree.

5.3 Other Optimizations

The "last agent" optimization allows the root TM to transfer responsibility for the commit/abort decision to a subordinate [8]. It may improve performance by eliminating a message when only a single subordinate is involved or when communication with one subordinate is more costly than communication with others. Tree flattening optimizations may be applied to multilevel protocols. Flattening allows the root TM to communicate directly with the leaf RMs, bypassing intermediate TMs. In the flattening optimization described in [8], the identities of the leaf RMs are propagated to the root TM during the first phase of the commit protocol. Flattening is then applied during phase two. The restructured presumed-commit protocol described in [1] assumes that the RM identities are propagated to the root TM during transaction execution, so communication is flattened during both phases of the commit protocol. The application of any of these optimizations to presumed-either is straightforward.

6 Conclusion

Most two-phase commit protocols, such as presumed-abort and presumed-commit, handle all committing transactions uniformly. In contrast, the presumed-either protocol uses a dynamic strategy. Transactions which can be resolved efficiently using presumed-commit are resolved that way. For other transactions, presumed-abort is used. Presumed-either exploits the fact that transactions

are processed concurrently and that log records for one transaction may be piggybacked into the log by log writes generated by others.

Presumed-either offers a potential performance gain with little risk. In the worst case, committing a transaction using presumed-either is no more costly than committing under the widely-implemented presumed-abort protocol. Furthermore, presumed-either has the attractive property that it is likely to perform best in high-performance systems that flush log pages frequently to stable storage.

ACKNOWLEDGMENTS

The authors would like to thank C. Mohan for a very useful discussion about two-phase commit protocols.

REFERENCES

[1] Y. Al-Houmaily, P. Chrysanthis, and S. Levitan, "An Argument in Favor of the Presumed Commit Protocol," *Proc. Int'l Conf. Data Eng.*, pp. 255-265, 1997.

[2] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1992.

[3] B. Lampson and D. Lomet, "A New Presumed Commit Optimization for Two Phase Commit," *Proc. Int'l Conf. Very Large Data Bases*, pp. 630-640, 1993.

[4] B.G. Lindsay, L.M. Haas, C. Mohan, P.F. Wilms, and R.A. Yost, "Computation and Communication in R*: A Distributed Database Manager," *ACM Trans. Computer Systems*, vol. 2, no. 1, pp. 24-38, 1984.

[5] C. Mohan, K. Britton, A. Citron, and G. Samaras, "Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU 6.2 Commit Protocols," Technical Report RJ 8684, IBM Research Division, Mar. 1992.

[6] C. Mohan and D. Dievendorff, "Recent Work on Distributed Commit Protocols, and Recoverable Messaging and Queueing," *Bull. IEEE Technical Committee on Data Eng.*, vol. 17, no. 1, pp. 22-28, Mar. 1994.

[7] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System," *ACM Trans. Database Systems*, vol. 11, no. 4, pp. 378-396, 1986.

[8] G. Samaras, K. Britton, A. Citron, and C. Mohan, "Two-Phase Commit Optimizations in a Commercial Distributed Environment," *Distributed and Parallel Databases*, vol. 3, pp. 325-360, 1995.