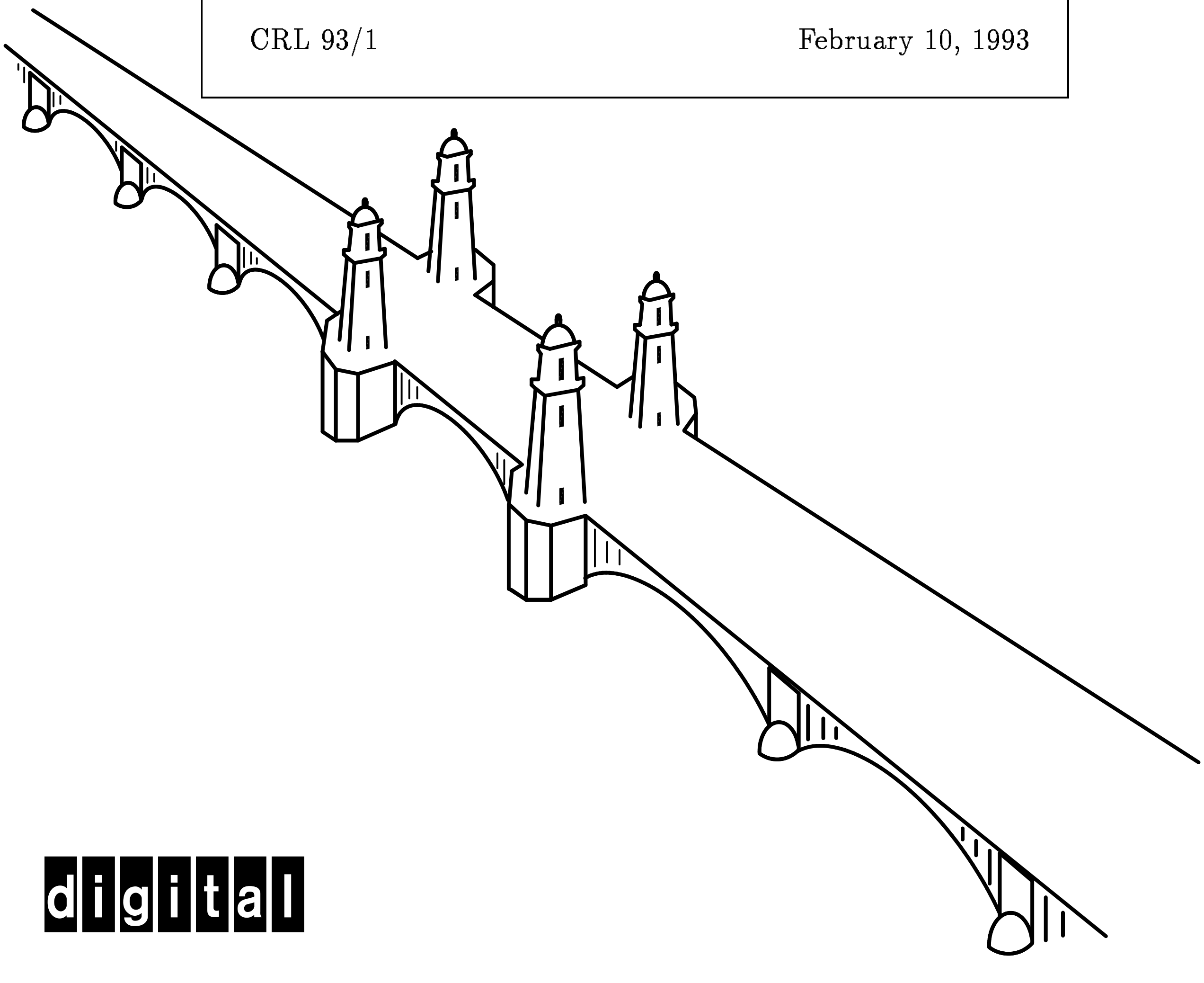
A New Presumed Commit Optimization for Two Phase Commit

Butler Lampson   David Lomet

Digital Equipment Corporation
Cambridge Research Lab

CRL 93/1   February 10, 1993



digital

CAMBRIDGE RESEARCH LABORATORY
Technical Report Series

Digital Equipment Corporation has four research facilities: the Systems Research Center and the Western Research Laboratory, both in Palo Alto, California; the Paris Research Laboratory, in Paris; and the Cambridge Research Laboratory, in Cambridge, Massachusetts.

The Cambridge laboratory became operational in 1988 and is located at One Kendall Square, near MIT. CRL engages in computing research to extend the state of the computing art in areas likely to be important to Digital and its customers in future years. CRL's main focus is applications technology; that is, the creation of knowledge and tools useful for the preparation of important classes of applications.

CRL Technical Reports can be ordered by electronic mail. To receive instructions, send a message to one of the following addresses, with the word **help** in the Subject line:

On Digital's EASYnet:	CRL::TECHREPORTS
On the Internet:	techreports@crl.dec.com

digital™

Cambridge Research Laboratory
One Kendall Square
Cambridge, Massachusetts 02139

A New Presumed Commit Optimization for Two Phase Commit

Butler Lampson David Lomet

Digital Equipment Corporation
Cambridge Research Lab

CRL 93/1 February 10, 1993

Abstract

Two phase commit (2PC) is used for coordinating the commitment of transactions in distributed systems. The preferred 2PC optimization is the presumed abort variant, which reduces the number of messages when transactions are aborted, and eliminates the coordinator's need to retain information about aborted transactions. The presumed commit 2PC variant provides a larger message reduction, but its coordinator must do additional logging. We describe a new form of presumed commit that reduces the number of log writes while preserving the reduction in messages, bringing both these costs below those of presumed abort. The penalty for this is the need to retain a small amount of crash related information forever.

Keywords: commit protocol, two phase commit, protocol optimization, presumed commit

1 Introduction

1.1 Coordinating Distributed Commit

Distributed systems rely on the two phase commit (2PC) protocol to coordinate the commit of transactions [1, 3]. 2PC guarantees the atomicity of distributed transactions, i.e. that all cohorts of a transaction either commit or the transaction is aborted. The cost of the 2PC is an important factor in the performance of distributed transactions.

- It requires multiple messages in multiple phases. These messages have both substantial computational cost, which affects system throughput, and substantial delay, which affects response time.

- It requires that information about transactions be stably recorded to ensure that the system can continue to guarantee transaction atomicity even if one or more elements of the system should fail during the commit protocol itself. This is usually accomplished by writing information to a log. When information *must* be at some point in the protocol, the log must be "forced", i.e. the write must be completed before proceeding to the next step. Forced writes are are more costly than simple writes because they translate into actual I/O, whether a block of the log is filled or not.

1.2 This Paper

In this paper, we describe a new variant of 2PC whose message cost is as low as its best alternative and whose coordinator logging cost is substantially less. The paper is organized as follows. In section 2 we describe the basic form of 2PC, with particular emphasis on message cost and the coordinator's need to be able to recover its "database" of protocol related information. In section 3 we present the traditional ways of optimizing 2PC, by presuming the outcome of transactions that do not have entries in the coordinator's database. Section 4 explains what information is essential to recover the protocol database should the coordinator crash, and how it can be provided using fewer log writes. The protocol that results from exploiting this new approach to recovery of the protocol database is described in section 5. Finally, we discuss the virtues and limitations of this approach to 2PC optimization in section 6.

2 Two Phase Commit

Commit coordination and its optimizations are discussed thoroughly in [2, 5, 6]. We recap this discussion here. We begin by describing the basic version of two phase commit, before applying most optimizations. The coordinator in this version requires very explicit information, and this version has sometimes been referred to as the "presumed nothing" protocol or PrN. This is in contrast to subsequent optimizations that do make presumptions regarding missing information. (Note, however, that in fact, PrN does make some presumptions [7].)

2.1 The Protocol Messages

To commit a distributed transaction, PrN requires two messages from coordinator to cohort, and two messages from cohort to coordinator (four messages in all). The protocol involves the following steps:

1. A coordinator notifies all cohorts that the transaction is to be terminated, via the PREPARE message.

2. Each cohort then sends a vote message (either a COMMIT-VOTE or an ABORT-VOTE) on the outcome of the transaction. A cohort responding with a COMMIT-VOTE is now prepared.

3. The coordinator commits the transaction if all cohorts send COMMIT-VOTEs. If any cohort sends an ABORT-VOTE, or the coordinator times out waiting for a vote, the coordinator aborts the transaction. The coordinator sends the outcome message (i.e. COMMIT or ABORT) to all cohorts.

4. The cohort terminates the transaction according to its outcome, either committed or aborted. The cohort then ACKs the outcome message.

2.2 Cohort Activity

Cohorts must ensure that they log enough information stably so that they can tolerate failures in the midst of the commit protocol. Before the vote message, no logging is required. Transactions failing then are aborted. Before responding with a COMMIT-VOTE, however, a cohort needs to stably record that it is prepared. This makes it possible for it to commit the transaction, even if it is interrupted by a crash.

If a prepared cohort does not receive the transaction outcome message promptly or crashes without remembering the outcome, the cohort asks the coordinator for the transaction outcome. It keeps on asking the coordinator until it receives an answer. (This is the blocking aspect of 2PC.)

Before ACKing a COMMIT or ABORT outcome message, a cohort forces the transaction outcome record to its log. The ACK message tells the coordinator that the cohort will not ask again about this transaction's outcome. Recovery at the cohort ensures that the cohort knows transaction outcome after a crash without asking the coordinator. Therefore, the coordinator can discard the outcome for this transaction once all the cohorts have ACKed. This final log write must be completed prior to sending the ACK message. However, there is no urgency about the sending of the ACK as the ACK's function is only a bookkeeping one, i.e. to permit the garbage collection of the protocol database, which is described in the next subsection. Hence one can group both the log writes and the ACK messages, amortizing their costs over several transactions.

2.3 The Protocol Database

The coordinator maintains a main memory *protocol* database that contains, at a minimum, the states of all transactions currently involved in 2PC. The protocol database enables the coordinator both to execute the 2PC protocol and also to answer queries from cohorts about transaction outcome. As we saw in the previous subsection, cohorts make such queries when they recover from a crash or when messages are lost; these failures can occur at any time. Because the coordinator can also fail, it keeps a log of protocol related activity to make the protocol database recoverable.

The protocol database for PrN contains entries for all transactions, committed, aborted, or still active, that have registered with the coordinator but

Trans. ID	Stable	State	{Cohort	Vote	ACK }
	Yes No	Initiated Prepared Aborted Committed		None ABORT-VOTE READ-ONLY-VOTE COMMIT-VOTE	Yes No

Figure 1: The format of a transaction entry in the protocol database. Each transaction is identified by a "Trans. ID". "Stable" indicates whether the transaction existence is stably recorded on the log. The "States" of a transaction are (i) "Initiated" indicating that the it is known to the system; (ii)"Prepared" indicating that a PREPARE message has been sent, etc. A transaction may have several cohorts. The "Vote" indicates how the cohort voted in response to the PREPARE message, "ACK" whether the outcome message has been ACKed.

have not completed the protocol. A PrN coordinator enters a transaction into its protocol database when that transaction is initiated. A transaction's entry includes its set of cohorts and the coordinator's knowledge of their protocol state, i.e. has a cohort responded to the PREPARE message with a vote, was it a COMMIT-VOTE or an ABORT-VOTE, has it ACKed the transaction outcome message, etc. The format for a transaction entry in the protocol database is given in Figure 1.

The ACK message helps the coordinator manage the protocol database. As each cohort ACKs, the coordinator can drop the cohort from the transaction's entry. When all cohorts have so responded, the coordinator deletes the transaction entry from its database.

2.4 Coordinator Recovery

2.4.1 Logging for Recovery

We assume that a transaction manager (TM) serves as the coordinator. The TM logs protocol activity to ensure that it can recover the protocol database. It does not log for transaction durability (directly). For example, fully ACKed transactions are not present in the protocol database and do not require recovery. How much is logged affects how precisely the protocol database can be reconstructed after a coordinator crash. The PrN logging usually involves two log records.

Before sending the outcome message, the PrN coordinator forces the transaction outcome on its log. This act either commits or aborts the transaction and permits recovery of the transaction's entry from this point on. Thus, transactions that have a outcome have a stable log record documenting it.

After receiving ACKs of the transaction outcome message from all cohorts, the PrN coordinator writes a non-forced END record to make this information durable. The coordinator need not then recover the transaction's entry in the protocol database after a crash, and hence keeps it from again asking the cohorts for ACKs.

2.4.2 Less Than Full Recovery

If we take the PrN "presumed nothing" characterization literally, we need to write many additional log records, usually forced. We need to reconstruct the protocol database precisely, including information about all aborted transactions. This requires that we force to the log, before sending the PREPARE message, the contents of a transaction's protocol database entry. If the coordinator crashes before transaction outcome is decided, we then have a stable record which allows us to explicitly abort the transaction.

As PrN is usually described, however, the ability to recover information about active (hence not yet decided) transactions is sacrificed to reduce logging cost. Traditionally, the transaction's entry is not logged until its outcome is logged (but see "presumed commit" below), and the transaction entry is lost if the coordinator crashes earlier than this. Cohorts that inquire about a transaction not in the protocol database are directed to abort the transaction. That is, these transactions are "presumed" to have aborted.

There are several ways to save log writes and cope with the less than complete information that exists after recovery. For example, how many cohorts of which transactions need to be contacted to re-ACK outcome messages depends on whether each ACK is logged, only completion of all ACKing is logged, or there is no logging related to ACKs. These strategies do not affect the correctness of 2PC but do impact the cost of recovering from coordinator crashes.

Type of	Coordinator		Cohort	
2PC Protocol	Update Trans.	Read-Only Trans.	Update	Read-Only
	m,n,o,p	m,n,o	m,n,q	m,n,q
PrN	2,1,-,2	-	2,2,2	-
PrA	2,1,1,2	0,0,1	2,2,2	0,0,1
PrC	2,2,1,2	2,1,1	2,1,1	0,0,1
NPrC	1,1,1,2	0,0,1	2,1,1	0,0,1

Table 1: The message and log write costs for 2PC and its optimizations. (m) log records, (n) forced log records, (o) messages to read only cohort, (p) messages to update cohorts, (q) messages from the cohort.

2.5 Summary for the PrN Protocol

To commit a transaction, a PrN coordinator does two log writes, the commit record (forced) and the transaction end record (non-forced). In addition, it sends two messages to each of its cohorts, PREPARE and COMMIT. In response, each cohort performs two log writes, a prepare record and a commit record (both forced), and sends two messages, a COMMIT-VOTE and a final ACK. These are tabulated in Table 1. This is similar to the table in [6].

3 Presumed Outcomes and Optimization

In some failure cases, a PrN coordinator presumes that transactions are aborted when it gets inquiries about transactions that are not in its protocol database. This presumption is possible because there are only two outcomes of a transaction, and PrN always remembers which transactions have committed. Thus, it is safe to presume that all other transactions, whether the coordinator is aware of them or not, have aborted.

We can exploit the two outcome characteristic of transaction termination more extensively than PrN does. We systematically purge either the aborted or the committed transactions from our protocol database. We then simply presume the alternative outcome for these transactions from the outcome of the retained entries. Because our protocol database no longer contains these purged entries, we do not have to recover them. Hence we need not log their protocol activity. Some messages as well as some log writes now become unnecessary. Below, we briefly describe two published 2PC optimizations. Each optimization presumes a different outcome for transactions missing from the protocol database.

3.1 Presumed Abort

In the absence of information about a transaction in its protocol database, a presumed abort (PrA) coordinator presumes the transaction has aborted. This abort presumption was already occasionally made by PrN. PrA makes it systematically to further reduce the costs of messages and logging. Once a transaction has aborted, its entry is deleted since its absence denotes the same outcome. No information need be logged about such transactions, as their protocol database entries need not be recovered.

We must guarantee that the protocol database always contains entries for committed transactions which have not yet completed all phases of 2PC. These entries must be recoverable across coordinator crashes. This requires the coordinator to make transaction commit stable before sending the COMMIT message by forcing this outcome to its log. The protocol database entries for committed transactions should be deleted when 2PC completes to limit database size, just like with PrN. And the same garbage collection strategies are also possible.

A coordinator need not make a transaction's entry stable before its commit because an earlier crash results in transaction abort, the presumed outcome in the absence of information. Only a COMMIT outcome needs to be logged (with a forced write). Since there is no entry in our protocol database for an aborted transaction, there is no entry in need of deletion, and hence no need for an ACK of the ABORT outcome message.

In summary, as with PrN, to commit a transaction, a PrA coordinator does two log writes, the commit record (forced) and the transaction end record (which isn't forced). In addition, it sends two messages to each cohort, PREPARE and COMMIT. In response, each cohort performs two log writes, a prepare record and a commit record (both forced), and sends two messages, a COMMIT-VOTE and a final ACK. These are tabulated in Table 1.

3.2 Presumed Commit

For presumed commit (PrC), the coordinator explicitly documents which transactions have aborted. While this has some apparent symmetry with PrA, which needs to know explicitly about committed transactions, in fact there is a fundamental difference. With PrA, we can be very lazy about making stable in the log the existence of a transaction. If it fails before our acquiring knowledge of it, we presume it has aborted. But PrC needs to know about and have a stable record of initiated transactions because missing transactions are presumed to have committed. A commit presumption is not correct for early failing transactions. Traditionally this has meant that at the time 2PC is initiated and a transaction is entered into the protocol database, the coordinator forces a transaction initiation record to the log, making its entry stable. This entry can then be recovered after a coordinator crash. Uncommitted transactions are then aborted rather than presumed to have committed.

With PrC, a transaction's entry is removed from the protocol database when it commits. Missing entries are presumed to have committed. If cohorts subsequently inquire, they are told the transaction committed (by presumption). Thus, PrC avoids ACK messages for committed transactions, which is the common case and hence a significant saving (much more important than avoiding ACKs for aborted transactions).

We must ensure that a committed transaction's entry is not re-inserted into the protocol database during coordinator crash recovery. If this happened, we might think this transaction should be aborted. Hence, we force commit information to the log before sending the COMMIT message. Logically, this log write erases the transaction initiation log record, since lack of information implies commit. However, given the nature of logs, it is easier to simply document the commit by forcing a commit record to the log tail. The commit log record tells us not to include the transaction in our protocol database of aborted transactions.

With PrC, both protocol database entry and transaction initiation log record list all cohorts from which ACKs are expected if the transaction aborts. When all ACKs have arrived, the entry can be garbage collected from the protocol database. We write a non-forced end record to the log to keep this transaction from being re-entered into the protocol database, just as is done with PrN. No separate abort record is needed.

In summary, to commit a transaction, a PrC coordinator does two forced log writes, the transaction initiation record and the commit record. In addition, it sends two messages to each cohort, PREPARE and COMMIT. In response, each cohort forces two log records, prepare and commit. The commit record need not be forced because a prepare record without a commit record causes the cohort to inquire about transaction outcome. The coordinator, not finding the transaction in its protocol database, will respond with a COMMIT message. The cohort sends one message, its COMMIT-VOTE. No final ACK is required. These are tabulated in Table 1.

3.3 Read-Only Cohorts and Transactions

When a cohort is read-only, it has done no logging and does not care about the transaction outcome, merely that the transaction is completed and that locks can be released. Such a cohort does not need to receive the transaction outcome message. To avoid receiving this second message, it sends a READ-ONLY-VOTE, then releases its locks and forgets the transaction. Thus, a read-only cohort writes no log records and sends one message.

The coordinator removes read-only cohorts from the list of cohorts to receive the transaction outcome message. If all cohorts sends a READ-ONLY-VOTE, then the coordinator sends no outcome message. In addition, it no longer matters whether the transaction is considered committed or aborted. Hence the coordinator can choose whichever outcome permits the least logging.

PrA: Abort the transaction by deleting its entry from the protocol database.

PrC: Abort the transaction by writing (non-forced) an abort/end record and deleting its entry from the protocol database.

Regardless of whether a transaction commits or aborts, whether it is an update transaction or a read-only transaction, and what variant of 2PC is used, the activity of read-only cohorts is the same. These cohorts respond to a PREPARE message with a READ-ONLY-VOTE message, do no log writes, and forget the transaction. These cohorts expect no transaction outcome message from the coordinator. This is the read-only optimization. It only guarantees serializability if it is known before the commencement of the 2PC protocol that cohorts have completed all their normal activity. (Section 6 discusses the impact if normal transaction activity can continue after 2PC begins.)

3.4 Advantage of Presumed Abort

It is the coordinator logging that makes PrA preferable to PrC. To commit a transaction, a PrC coordinator forces two log records, while with PrA only one force is required while the other log write is not forced. The extra forced write is for PrC's transaction initiation record and it is needed for every transaction. Hence, it shows up in both update and read-only transactions. Furthermore, for committed transactions it is necessary to force the commit record documenting that a previously initiated transaction has *not* aborted.

4 Reducing PrC Coordinator Log Writes

Considering message cost only, the PrC protocol has a decided advantage. Hence, we focus on reducing its coordinator logging costs. In particular, we want to avoid forcing the transaction initiation record. This forced log write documents that the transaction has initiated the commit protocol. It permits us to explicitly notify cohorts when a transaction aborts because the coordinator crashes, and to garbage collect its entry in the protocol database once all the cohorts have ACKed the abort. To avoid this log write, we need to know how a coordinator identifies transactions that were in the active protocol phase at the time of a crash, and how it manages the protocol database when it cannot garbage collect transactions that are aborted by a crash. Our fundamental idea is to give up on (i) having full knowledge after coordinator recovery as to the specific transactions active when the coordinator failed and (ii) garbage collecting the information that we do have about transactions active when the coordinator failed.

4.1 Potentially "Initiated" Transactions

Instead of full knowledge about active transactions, we reconstruct information about all the transactions that may have been active at the time of a crash. We denote this set of potentially initiated transactions as IN. IN must include all transactions actually active, but it may also include never

Term	Definition
AB	Set of explicitly aborted transactions (stable)
COM	Set of explicitly committed transactions (stable)
IN	Set of potentially initiated transactions that includes all undocumented active transactions
REC	Set of recent transactions near a crash
tid_h	last transaction that may have executed will be higher than tid_{sta}
tid_l	a transaction lower than all active transactions that are undocumented
tid_{sta}	highest tid with stable log record
Δ	max number of active transactions with $tid > tid_{sta}$

Table 2: The terms used in describing the NPrC 2PC optimization.

initiated transactions as well. Since we do not know cohorts for transactions in this set, we cannot garbage collect entries for these transactions from the protocol database.

We can reasonably bound IN without forcing transaction initiation log records, hence eliminating the need for these forced writes. We assume that transaction identifiers ($tids$) are assigned in monotonically increasing order. Then, we find a high tid_h and a low tid_l such that the $tids$ of all such undocumented transactions must lie between them. (Table 2 defines the notation that we use in this paper.)

Let us denote the set of tids of committed and stably documented transactions as COM. Let REC denote the set of "recent" $tids$, defined as

$$REC = \{tid \mid tid_l < tid < tid_h\} \tag{1}$$

Then we define IN as:

$$IN = REC - COM = REC - (COM \cap REC) \tag{2}$$

$(COM \cap REC)$ is simply the set of $tids$ in REC which have committed.

No undocumented transaction that has begun 2PC has a tid less than tid_l. No transaction with a tid higher than tid_h has begun 2PC. Neither tid_h

nor tid_l need be a *tid* of an actual transaction. They are simply bounds on transaction identifiers associated with this set.

To sum up the preceding discussion, we represent the set of initiated transactions IN for each system crash with the following data structure:

$$< tid_l, tid_h, COM \cap REC > \tag{3}$$

All *tids* in IN have abort outcomes by presumption, whether they initiated the 2PC protocol or not. IN contains the set active at the time of a crash and hence aborted. Thus, responding to inquiries about these transactions with an abort is appropriate. The set IN may include non-existent transactions and those that never begun the 2PC protocol. It does not matter whether these are deemed to have committed or aborted because no cohorts will ever inquire as to their status. We must, however, ensure that these *tids* are not re-used for this to remain true.

Two problems persist:

1. How do we determine IN for a crash at recovery time and make sure that its *tid*s are not re-used?

2. How do we represent the information contained in IN in a compact fashion, given that garbage collection is not feasible, and hence that IN must be retained permanently?

4.2 Recovering IN After a Crash

4.2.1 Determining tid_h

We describe two straightforward approaches to determining tid_h. Both prevent transactions with *tid*s greater than tid_h from beginning.

Δ Method: We refer to the transaction with the highest *tid* present on the log as tid_{sta}. After a crash, we determine tid_{sta} by reading the log. We choose a fixed Δ, e.g. of 100 *tids*. Then $tid_h = tid_{sta} + \Delta$. Having a fixed Δ means that no extra logging activity is needed to recover tid_h.

Logging tid_h: We determine tid_h during recovery by reading its value explicitly from the log. This requires us to periodically write candidate tid_hs to the log. The last candidate tid_h logged before a crash becomes the tid_h for the crash. To avoid having to force a log record when a transaction begins the 2PC protocol, we set tid_h to be several tids beyond the currently used highest tid. This approach permits us to adapt tid_h to system load.

On recovery from a crash, regardless of how tid_h is determined, we require the coordinator to use tids that are greater than tid_h. This ensures that no tid of IN is re-used. This is important as it enforces that tids of IN have a single outcome, i.e. abort.

4.2.2 Determining tid_l

We denote by tid_l the lower bound for the tids of active and undocumented transactions. All transactions with tids less than tid_l that have begun since the last crash have either a commit or an abort record in the log. (With the variation described in section 6, they might also have an explicit transaction initiation log record.) Having a tight bound for tid_l permits us to minimize the number of transactions in IN. This is important because IN must be stored permanently.

We ensure that tid_l is known after a crash by writing it to the log. Whenever tid_l can be advanced, we write to the log the new value for tid_l along with the commit or abort record for the transaction whose termination permits us to advance tid_l. Thus tid_l is recorded without extra log writes or forces. The log is marked with a series of monotonically increasing tid_ls. The last tid_l written before a crash is the tid_l used in representing IN.

While the system is executing normally, we know which transaction is this oldest active undocumented one. (Here, an active transaction means any transaction known to the coordinator to have begun, whether or not it has initiated the commit protocol.) The termination of this transaction permits tid_l to be advanced. Thus, we log transaction termination as follows:

Not oldest active transaction: If it is committing, we force a commit record for it. If it is aborting, then when all ACKs have been received, we simply delete it from the protocol database.

Oldest active undocumented transaction: If it is committing, we write to the log, along with its commit record, the new tid_l. tid_l might not be the tid of the completing transaction. Rather, it will sometimes be a higher *tid*, indicating that subsequent transactions has also completed. If it is aborting, then when all ACKs are received we write (non-forced) a new tid_l to the log.

If the coordinator fails before tid_l is advanced past the *tid* of a committed transaction, the log contains the transaction's commit record which keeps it out of IN. If the coordinator fails after tid_l advances past the committed transaction's *tid*, then the transaction is committed by presumption.

If the coordinator fails before tid_l is advanced past the *tid* of an aborted transaction, then the transaction becomes part of IN and hence is remembered as an aborted transaction. If the coordinator fails after tid_l is advanced past the aborted transaction's *tid*, ACKs from all cohorts must have been received. Hence there will be no inquiries about this transaction.

4.2.3 Determining and Representing $COM \cap REC$

Because IN needs to be permanently recorded, it is important that the representation for IN be small. The quantities tid_h and tid_l consume a trivial amount of storage. The only question is how compactly one can represent $COM \cap REC$. All transactions that commit have commit records stored on the log. So determining which transactions have committed requires simply searching the log for commit records.

There are two standard ways to represent sets which can be effective in representing $COM \cap REC$, depending on set size and sparseness.

Consecutive *tids* When *tids* are allocated consecutively, a compact representation for a set is a bit vector. Our tid_l becomes the origin for the bit vector (BV). BV need only have a size of $tid_{sta} - tid_l$ where $COM \cap REC = \{tid \mid BV[tid - tid_l] = 1\}$ This is because there are no committed transactions with *tid*s greater than tid_{sta}.

Non-Consecutive *tids* When *tids* are sparsely allocated, a bit vector is not a compact representation. Sparse allocation might arise if timestamps are used within *tids*. Here we represent $COM \cap REC$ as an explicit list of *tids*, i.e. of transactions with *tids* between tid_l and tid_h that have committed. If each *tid* is 16 bytes, and the cardinality of $COM \cap REC$ is around 50, and assuming that 2:1 compression is possible on this set of *tids*, then the amount of information stored for each crash is not more than 500 bytes.

4.3 Persistent IN and Its Use

No transactions in IN have committed. But we do not know whether they were aborted or whether they never ran. And if aborted, we do not know whether they began the 2PC protocol or not. Hence, we do not know whether we will receive inquiries about this set or not. Nor do we know how many inquiries we might receive or by which cohort. It is thus impossible to garbage collect the information concerning transactions in IN. The set IN thus has to be recorded permanently. Fortunately, the cardinality of $COM \cap REC$ will typically be small. Also, the stably recorded information will be linear in the number of system crashes.

Given the representations for IN described above, persistently storing IN is quite manageable. Even assuming that the system crashes once a day (which is high for a well managed system), and the system is in operation seven days a week, it would take 2000 days or six years to accumulate one megabyte of crash related IN information. The current purchase price of a megabyte of disk space is two dollars.

So that the transaction manager can respond quickly to requests for transaction outcomes, information from IN should be maintained in main memory. While IN may be too large to be stored entirely in main memory, we can easily cache information about the last several crashes. Almost all inquiries will be for transactions involved in these crashes, and maintaining this information in main memory has a trivial cost. This should easily suffice for efficient system operation.

5 A New Presumed Commit Protocol

Building on the preceding ideas, we now describe a new presumed commit protocol (NPrC) that does not require a log force at protocol start. NPrC has a message protocol that is identical to the PrC protocol, and it manages its volatile protocol database in much the same way. NPrC differs from PrC in what its coordinators write to the log, and hence in the information that the coordinators recover after a crash. We assume that a transaction manager coordinates commit and has its own log [2]. We write the description for a flat transaction cohort structure. An extension to the tree model is discussed in section 6.

5.1 Coordinator Begins NPrC Protocol

The 2PC protocol begins when the coordinator receives a commit directive from some a cohort of the transaction or the application. The coordinator sends out PREPARE messages to cohorts asking them whether to commit the transaction. No log record is forced, or even written. The coordinator then waits to receive responses from all cohorts.

We distinguish the cases where a transaction is aborted, where the transaction has done updating, and where the transaction is read-only. In particular, a transaction cohort sends an ABORT-VOTE message if it wishes to abort the transaction, a COMMIT-VOTE message if the cohort has updated, and a READ-ONLY-VOTE message if the cohort has only read data.

5.2 Aborting a Transaction

If any of the cohorts sends an ABORT-VOTE, or if the responses do not arrive in a timely fashion, then the coordinator sends an ABORT outcome message to cohorts that have not sent an ABORT-VOTE. When all such cohorts have ACKed the ABORT message, the coordinator deletes the transaction from its protocol database. Now tid_l can be advanced past its *tid*.

Should the system fail before all ACKs for an aborted transaction are received or after ACKs are received but before tid_l is advanced past its *tid*, the transaction will be part of IN, and on a cohort inquiry the coordinator will respond that the transaction has aborted. If the system fails after tid_l is advanced past its *tid*, then the transaction is presumed to have committed. However, that cannot happen until after all ACKs are received, and hence no inquiries will ever be made.

Thus for transaction abort we require four messages per update cohort that sent COMMIT-VOTEs, two from coordinator to cohort (PREPARE and ABORT), and two from cohort to coordinator (COMMIT-VOTE and ACK) and a log write only if the aborting transaction was the oldest active transaction. This records the new value of tid_l. It needn't be forced. Aborting cohorts send only the one ABORT-VOTE message.

5.3 Committing an Update Transaction

If all cohorts have voted, no cohort has sent an ABORT-VOTE, and at least one cohort has sent a COMMIT-VOTE, then this is an update transaction. The coordinator forces a commit log record. The commit record need not contain the names of cohorts, and no END record is needed since there are no ACK messages expected. The transaction's entry is deleted from the protocol database and the transaction is presumed to have committed. When the committing transaction is the oldest active transaction, a new tid_l record is forced to the log along with the commit record.

Should the system fail before the commit record is forced, the transaction is in IN and will be aborted. If it fails after the commit record is forced, but before tid_l advances past its tid, its tid is part of REC, but it is in COM and hence not in IN. If the system fails after tid_l is advanced past its tid, the transaction is correctly presumed to have committed.

Thus, for transaction commit, the message/log cost of this coordinator activity is one log record written and forced, the commit record with or without the tid_l record, and three messages per update cohort, PREPARE, COMMIT-VOTE, and COMMIT. The ACK message is avoided.

5.4 Committing a Read-Only Transaction

No log record is written in the protocols above until after the votes for all cohorts have been received. If all cohorts send READ-ONLY-VOTEs, the transaction is a read-only transaction. All cohorts have terminated without writing to their log, and have "forgotten" this transaction. There is no need for the coordinator to write any log record, nor to send any additional messages.

If the system crashes, the information that is derived to document the NPrC requirements will suggest different outcomes, depending on how close to the crash the read-only transaction finished. If the tid for this transaction is greater than tid_l, then it will be in IN, and the transaction will appear to be aborted. If less than tid_l, then it will appear to have been committed. However, no cohort will make an inquiry and the apparent result of the transaction is irrelevant.

The protocol cost in this case is no log records written at the coordinator, and one message(PREPARE) to and one(READ-ONLY-VOTE) from each (read-only) cohort. A cohort need not write a log record for the usual 2PC protocol.

5.5 Summary and Comparison

The message and log write costs of NPrC are included in Table 1. Its costs are never worse, and are usually better than the costs of either the standard PrN 2PC protocol or the two common optimized forms of 2PC, presumed abort (PrA) and presumed commit (PrC). Note in particular that to commit an update transaction, an NPrC coordinator needs fewer log writes than either PrA or PrC, and an NPrC cohort sends fewer messages than PrA. Further, to abort a transaction usually entails no log write. Occasionally, a tid_l record might to be written, but it needn't be forced.

The NPrC protocol achieves logging cost lower than that of PrA by focusing on the main memory protocol database. In particular, it is only necessary to correctly identify commit or abort outcomes for those transactions that are engaged in the protocol and whose cohorts may ask for the outcomes. Presuming an incorrect outcome for other transactions in no way compromises correctness of the protocol . Further, we sacrifice the recovery of information used to garbage collect protocol database entries. This requires that some information about transaction outcome be retained forever. However, the information preserved for each crash is small. So long as the coordinator does not crash often, retaining this information is only a minor burden. The reduction in coordinator logging is substantial.

A cohort need not know whether the coordinator is executing the PrC or the NPrC protocols because the message protocol is the same. It is only within the coordinator that behavior is different. We have traded the ongoing logging necessary to permit us to *always* garbage collect our protocol database entries after a coordinator crash for the cost of storing forever a small amount of information about each crash. This appears to be a good trade.

6 Discussion

There are times when the model for distributed commit that we have assumed is too simple. Below, we explore some more complex scenarios.

6.1 Recalcitrant Transactions

There are a number of situations in which tid_l may be prevented from advancing or in which we may want to violate its requirements.

- A transaction has been aborted because a cohort has failed; it will be a long time before the failed cohort ACKs the abort. Given our prior approach, tid_l cannot be advanced past this transaction's *tid*.

- A transaction is very long-lived. While it is active, it prevents tid_l from being advanced past its *tid*.

- In the tree of processes model of transactions [6], a coordinator at one level of the transaction tree can be a cohort at the next higher level. Such a coordinator as cohort does not control the issuing of *tid*s. Hence, this coordinator may receive a *tid* that is earlier than its current tid_l.

There is a common solution for each of these recalcitrant transactions. Write to the log an explicit initiation record for it. We logically delete an initiation log record via an end record (unforced) for aborted transactions, and a commit record (forced) for committed transactions. For these cases, we have resorted to the original PrC protocol. We permit tid_l to be greater than the *tid*s of these explicitly initiated transactions. At recovery time, we restore to our protocol database all transactions with initiation records on the log that have not been terminated explicitly.

We can frequently piggyback the transaction initiation record for these transactions on a commit or abort already in progress. Advancing tid_l can also be done at this time. So long as the log record advancing tid_l is written after the transaction initiation record, there are no additional log forces.

When a coordinator in the tree of processes transaction model receives a *tid* that is below its tid_l it acts like a PrC coordinator (see [5]). The coordinator forces a transaction initiation record to its log before proceeding with transaction activity, and particularly before forwarding this *tid* to further cohorts,

The important thing here is that the vast majority of transactions will not require such initiation records, hence will save the log writes. All our optimizations occur within the coordinator. Externally, the message and cohort protocols are those usually associated with PrC in any event. Hence, externally, one cannot distinguish the coordinator behavior used for logging any given transaction.

6.2 Transaction Timestamping

In [4], timestamped voting was used both to optimize 2PC and to provide each committed transaction with a timestamp that agrees with transaction serialization. It guaranteed transaction serializability in the case that transaction termination is not guaranteed, while permitting the read-only and other optimization. Given the performance of the read-only optimization, and the fact that commercial commit protocols usually do not require transaction termination, this is important. There are two cases that we need to consider.

6.2.1 Timestamps for Versioned Data

Supporting timestamped versioned databases violates the assumptions upon which presumed commit protocols are based. It is no longer sufficient to know only that a transaction has committed. We must know its commit timestamp as well. This means that we cannot presume commit since we cannot presume the timestamps. Obviously, we want the coordinator to garbage collect these entries once they are no longer needed. The consequence of this is that presumed abort (PrA), which remembers the committed transactions, is more desirable in this case as it can simply keep the timestamps with its committed transaction entries. Forms of presumed commit cannot be used.

6.2.2 Timestamps Only for the Commit Protocol

So long as databases are *not* using transaction timestamps to timestamp data, but are using them solely as part of the commit protocol [4], it is not necessary to remember the timestamp of a committed transaction. The coordinator will have sent its COMMIT message with a timestamp that is within the bounds set by the timestamp ranges of all cohorts. If asked, the coordinator responds that the transaction was committed, and the cohort then knows that the commit time was within the timestamp range of its COMMIT-VOTE message.

The cohort uses the knowledge of whether the transaction committed or aborted to permit it to install the appropriate state, before state in the case of abort, after state in the case of commit. It can safely release all locks, both read and write locks, at the time denoted by the upper bound in its COMMIT-VOTE timestamp range.

Because the coordinator need not remember a committed transaction's timestamp, the information about transactions that have completed the commit protocol is again binary, i.e. commit or abort. Presumed commit protocols can be used in these instances, and our NPrC protocol is not only applicable but desirable.

References

[1] Gray, J. Notes on Database Systems IBM Research Report, 1978.

[2] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques* Morgan-Kaufman, Redwood, CA. 1992

[3] Lampson, B. and Sturgis, H. Crash Recovery in a Distributed System Xerox PARC Research Report, 1976.

[4] Lomet, D. Using Timestamps to Optimize Two Phase Commit. Proceedings of the PDIS Conference, San Diego, CA (Jan 1993) (to appear)

[5] Mohan, C. and Lindsay, B. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions Proceedings of the 2nd ACM SIGACT/SIGOPS Symposium on PODC, Montreal, CA (Aug. 1983).

[6] Mohan, C., Lindsay, B. and Obermark, R. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Systems*11,4 (Dec. 86) 378-396.

[7] Samaras, G., Britton, K., Citron, A., and Mohan, C. Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. *Proc. Data Engineering Conf.*, Vienna, Austria (Feb. 1993).

Contents