A New Presumed Commit Optimization for Two Phase Commit

Butler Lampson and David Lomet
DEC Cambridge Research Lab
One Kendall Square, Bldg 700
Cambridge, MA 02139

Abstract

Two phase commit (2PC) is used to coordinate the commitment of transactions in distributed systems. The standard 2PC optimization is the presumed abort variant, which uses fewer messages when transactions are aborted and allows the coordinator to forget about aborted transactions. The presumed commit variant of 2PC uses even fewer messages, 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.

1 Introduction

1.1 Coordinating Distributed Commit

Distributed systems rely on the two phase commit (2PC) protocol to coordinate the commitment of transactions [1, 4]. 2PC guarantees the atomicity of distributed transactions, that is, that all cohorts of a transaction either commit or abort the transaction. The cost of 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 recorded stably to ensure that transactions remain atomic even if there is a failure during the commit protocol. This is usually done by writing information to a log. When information *must* be stable at some point in the protocol, the log must be "forced", that is, the write must be completed before proceeding to the next step. Forced writes cost more than simple writes because they require 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 the 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 for recovering the protocol database after a 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 [3, 8, 9]. We recap this discussion here, beginning with a description of the basic version of two phase commit. In this version the coordinator requires very explicit information, which is

**Proceedings of the 19th VLDB Conference,
Dublin, Ireland, 1993**

why it is often called the "presumed nothing" protocol or PrN. This is in contrast to optimized versions that do make presumptions about missing information. (Note, however, that in fact, PrN makes presumptions in some cases [10].)

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, or four messages in all. The protocol has the following steps:

1. The coordinator sends PREPARE messages to all cohorts to notify them that the transaction is to be terminated.

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 outcome messages (i.e. COMMIT or ABORT) to all cohorts.

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

2.2 Cohort Activity

A cohort must log enough information stably so that it can tolerate failures both before the commit protocol begins and during the commit protocol. If a cohort fails, it's necessary to abort every transaction that has had any activity there and is not yet prepared there. Otherwise updates might be lost, or serializability might be compromised because read locks are released prematurely as a result of the failure. Hence the cohort must vote to abort a transaction if the cohort has failed since the first time it saw any activity for the transaction. Two ways to ensure this which do not require any logging are given below.

- The client marks the first action of a transaction that it sends to each cohort. The cohort records a transaction as active when it sees an action marked as first, and votes to abort a transaction unless it's recorded as active.

- The cohort counts the number of actions it has seen for each transaction, and the client counts the number of actions it has sent to each cohort. The client passes all the counts to the coordinator with the commit request and the coordinator passes each count on to the proper cohort. The cohort votes to abort if its count is different.

Before responding with a COMMIT-VOTE, a cohort must stably record that it is prepared. This makes it possible for it to commit the transaction even if it is later interrupted by a crash. If a prepared cohort does not receive a transaction outcome message promptly, or crashes without remembering the outcome, the cohort asks the coordinator for the outcome. It keeps on asking until it gets an answer. (This is the blocking aspect of 2PC.)

Before ACKing a COMMIT or ABORT outcome message, a cohort writes the transaction outcome to its log. The ACK message tells the coordinator that the cohort will not ask again about this transaction's outcome. If the cohort crashes, its recovery will retrieve the outcome from the log without asking the coordinator. The coordinator can therefore discard the outcome for this transaction once all the cohorts have ACKed. The cohort must complete the outcome log write before sending the ACK message. There is no urgency about sending the ACK, however, because its function is only a bookkeeping one, i.e., to permit the garbage collection of the protocol database (described in the next subsection). Hence the cohort 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 inquiries from cohorts about transaction outcome. As we saw in the previous subsection, cohorts make such inquiries 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 so that it can recover the protocol database.

The protocol database for PrN contains entries for all transactions, committed, aborted, or still active, that have registered with the coordinator but have not completed the protocol. A PrN coordinator enters a transaction into its protocol database when that transaction is initiated. A transaction's

Tid	Stable	State	{Cid	Vote	Ack }
	Yes	Initiated		None	Yes
	No	Preparing		Abort	No
		Aborted		R-O	
		Committed		Commit	

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

entry includes its set of cohorts and the coordinator's knowledge of their protocol state: 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. For 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 an outcome have a stable log record documenting it.

After receiving ACKs of the outcome message from all cohorts, the PrN coordinator writes a non-forced END record to make this information durable. The END record tells the coordinator's recovery not to restore the transaction's entry in the protocol database after a crash, and hence it will not again ask 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, in order to reconstruct the protocol database precisely, including information about all aborted transactions. This requires that before sending a PREPARE message we force to the log the contents of a transaction's protocol database entry. If the coordinator crashes before the 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 undecided transactions is sacrificed to reduce logging cost. Traditionally, nothing about the transaction is logged until its outcome is logged, and hence 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, the number of cohorts that 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 choices do not affect the correctness of 2PC, but they do affect the cost of recovering from coordinator crashes.

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 (not forced). In addition, it sends two messages to each of its cohorts, PREPARE and COMMIT. In response, each cohort does 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, which is similar to the table in [9].

2PC	Coordinator		Cohort	
Variant	Updt Trans	RO Trans	Updt	RO
	m, n, p	m, n	m, n, q	m, n, q
PrN	2, 1, 2	-	2, 2, 2	-
PrA	2, 1, 2	0, 0	2, 2, 2	0, 0, 1
PrC	2, 2, 2	2, 1	2, 1, 1	0, 0, 1
NPrC	1, 1, 2	0, 0	2, 1, 1	0, 0, 1

Table 1: The message and log write costs to commit a transaction for 2PC and its optimizations: m log records, n forced log records, p messages to update cohorts, and q messages from the cohort. All protocol variants send only one message to read-only (R-O) cohorts. A read-only cohort sends no messages when the coordinator knows that the cohort is read-only.

3 Presumed Optimization

As we have seen, after a failure a PrN coordinator sometimes presumes that a transaction is aborted when it gets an inquiry about a transaction that is not in its protocol database. This works because there are only two possible outcomes of a transaction, and PrN always remembers which transactions have committed. Thus, it is safe to presume that all other transactions have aborted, whether the coordinator is aware of them or not.

We can exploit this property more extensively than PrN does by systematically purging entries from the protocol database for either the aborted or the committed transactions. We then simply presume the purged outcome for any transaction that lacks a database entry. We do not have to recover purged entries, so we do not have to log their protocol activity. Some messages as well as some log writes now become unnecessary. Below, we briefly describe two published 2PC optimizations, one presuming abort and the other presuming commit.

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 made occasionally 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 a missing entry denotes the same outcome. No information need be logged about such transactions because 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 means that as in PrN, the coordinator must make transaction commit stable before sending a COMMIT message, by forcing this outcome to its log. PrA deletes the protocol database entries for committed transactions when 2PC completes in order to limit the size of the database, just as PrN does. 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 aborts the transaction, and that is 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 the 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, PrA aborts a transaction more cheaply than PrN, and it commits one in exactly the same way. The costs of commit 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 explicitly documents committed transactions, in fact there is a fundamental difference. With PrA, we can be very lazy about making the existence of a transaction stable in the log. If there is a failure first, we presume it has aborted. But PrC needs a stable record of every transaction that has started to prepare because missing transactions are presumed to have committed, and a commit presumption is wrong for a transaction that fails early. 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 to make its database entry stable. This entry can then be recovered after a coordinator crash, so that an uncommitted transaction is aborted rather than presumed to have committed.

With PrC, a transaction's entry is removed from the protocol database when it commits, because 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 when the coordinator recovers from a crash. If this happened, we might think the transaction should be aborted. Hence, like PrN and PrA, PrC forces commit information to the log before sending the COMMIT message. Logically, this log write erases the 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 the protocol database of aborted transactions.

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

In summary, PrC commits a transaction with two forced log writes, the initiation record and the commit record. In addition, it sends two messages to each cohort, PREPARE and COMMIT. In response, each cohort forces a prepare log record and writes a commit log record. The commit record need not be forced because a prepare record without a commit record causes the cohort to inquire about the 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 costs are tabulated in Table 1.

3.3 Read-Only Optimizations

When a cohort is read-only, it has done no logging and does not care about the transaction outcome. It only wants to know that the transaction is completed so it can release its locks. Such a cohort does not need to receive the transaction outcome message. 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 a read-only cohort is the same. To avoid receiving an outcome message it sends a READ-ONLY-VOTE. Then it releases its locks and forgets the transaction.

Thus a read-only cohort writes no log records and sends one message. This is the read-only optimisation. 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.)

The coordinator removes read-only cohorts from the list of cohorts that should receive the transaction outcome message. If every cohort 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 an unforced abort/end record and deleting its entry from the protocol database.

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 PrA forces only one record; its 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.

4 Fewer PrC Log Writes

The PrC protocol has a decided advantage in message costs. Hence, we focus on reducing its coordinator logging costs. In particular, we want to avoid forcing the 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 (i) give up full knowledge after a coordinator crash of the transactions that were active before the crash and (ii) give up on garbage collecting the information that we do have about transactions that were active before a crash.

4.1 Potentially Initiated Transactions

Instead of full knowledge about the active transactions, after recovery we settle for minimal knowledge

Term	Definition
COM	Explicitly committed transactions (stable)
IN	Potentially initiated transactions (includes all undocumented active transactions)
REC	Recent transactions
tid_h	Last transaction that may have executed (will be higher than tid_{sta})
tid_l	A transaction lower than any undocumented active transaction
tid_{sta}	Highest *tid* with a stable log record
Δ	Max no. of active trans. with $tid > tid_{sta}$

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

about all the transactions that *may* have been active at the time of a crash. We denote this set of transactions that may have initiated but did not commit by IN. It must include all the transactions that were actually active, but it may also include transactions that were never initiated as well as transactions that aborted. Since we do not know the cohorts for transactions in IN, we cannot garbage collect their entries from the protocol database.

We can reasonably bound IN without forcing initiation records and thus eliminate the need for these forced writes. To do this, we assume that transaction identifiers (***tids***) are assigned in monotonically increasing order at a coordinator. 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. These are the "recent" *tids*; we define

$$REC = \{tid \mid tid_l < tid < tid_h\} \qquad (1)$$

(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. Then we define IN as:

$$IN = REC - COM = REC - (COM \cap REC) \quad (2)$$

$(COM \cap REC)$ 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:

$$< tid_l, tid_h, COM \cap REC > \qquad (3)$$

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 began 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 at recovery time and make sure that its *tids* are not re-used?

2. How do we represent the information contained in IN in a compact fashion, given that garbage collection may not be feasible, and hence that the value of IN after a crash may need to 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 *tids* 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, say of 100 *tids*. Then $tid_h = tid_{sta} + \Delta$. Having a fixed Δ means that no extra logging activity is needed to make it possible 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_h's 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 a number of *tids* beyond the currently used highest *tid*. This approach permits us to adapt tid_h to system load.

Regardless of how tid_h is determined, after a crash the coordinator must use *tids* that are greater than tid_h. This ensures that no *tid* of IN is re-used, and hence that the *tids* of IN have a single outcome, namely abort.

4.2.2 Determining tid_l

Recall that tid_l is 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 and have not completed the protocol have either a commit or an abort record in the log. (With the variation for recalcitrant transactions 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. We can advance tid_l whenever that transaction terminates, i.e., either commits or aborts. When this happens we write to the log the new value for tid_l along with the commit or abort record that we are writing anyway. Thus tid_l is recorded without extra log writes or forces. The log contains a series of monotonically increasing tid_l's. 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 and delete it from the protocol database. If it is aborting, then when all ACKs have been received, we delete it from the database.

Oldest active undocumented transaction: If it is committing, we write the new tid_l to the log along with the commit record. This might not be the *tid* of the completing transaction; it may be a higher *tid* if later transactions have also terminated. If the transaction is aborting, then when all ACKs are received we do an unforced write of the 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 *tid* of an aborted transaction, ACKs from all cohorts must have been received. Hence there will be no inquiries about this transaction, so it doesn't matter that an inquiry would be told that the transaction committed.

An illuminating way to think about the protocol is that after a crash the coordinator presumes abort for *tids* greater than tid_l and presumes commit for *tids* less than tid_l. Thus after a crash tid_l is the boundary between a presumed abort regime and a presumed commit regime. During normal operation the regime is presumed commit for all transactions. This means that recovery must make an abort entry in the protocol database for each *tid* greater than tid_l that doesn't have a commit record in the log.

4.2.3 Determining the Set $COM \cap REC$

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

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 *tids* 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*. A common way of doing this is to define a *tid* as $< timestamp, nodeid >$. Such *tids* are both monotonic at a coordinator and unique across the system. 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 which cohorts might make them. It can thus be very difficult to garbage collect the information concerning transactions in IN. One way to deal with this is to permanently retain IN.

Permanently retaining transaction outcome was originally proposed in [2]. There all transaction outcomes were retained permanently in one of the commit protocols described. Our technique immediately dispenses with the greater part of this information by the presumed commit strategy. For aborted transactions, we normally garbage collect transaction outcomes by requiring explicit ACK messages. Only transactions that abort because of a system crash cannot be garbage collected. 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, storing storing it forever 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 PrC 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 coordinator writes to the log, and hence in the information that the coordinator recovers after a crash. We assume that a transaction manager coordinates commit and has its own log [3]. We write the description for a flat transaction cohort structure; an extension to the tree model is discussed in section 6.

During normal operation, NPrC's extra complexity is minimal. It needs only to delimit persistently the set of potentially initiated transactions. This it does by occasionally doing an unforced write of a small amount of extra information to the log so that tid_l and tid_h can be recovered after a crash. This is much less costly than the forced log writes required by PrC. At recovery time, an NPrC coordinator needs to do more work than a PrC coordinator because it knows less. But crashes are rare, and the extra work at recovery is not large in any event.

5.1 Coordinator Begins Protocol

The 2PC protocol begins when the coordinator receives a commit directive from some cohort of the transaction or from 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. This is just like PrN.

5.2 Aborting Transactions

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 there are 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 need not be forced. Aborting cohorts send only the one ABORT-VOTE message.

5.3 Committing Update Transactions

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. This record need not contain the names of cohorts, and no END record is needed later 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 cost of this coordinator activity is one log record forced (the commit record with or without tid_l) and three messages per update cohort, PREPARE, COMMIT-VOTE, and COMMIT. The ACK message is avoided.

5.4 Committing R-O Transactions

NPrC writes no log record 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 logs, and have "forgotten" this transaction. There is no need for the coordinator to write any log record or to send any additional messages.

If the system crashes, the value of IN will imply 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 be committed. However, no cohort will make an inquiry so the apparent outcome is irrelevant.

The protocol cost in this case is no log records written at the coordinator, one message (PREPARE) to each cohort, and one message (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 for NPrC to commit a transaction are tabulated in Table 1. Its costs are never worse, and are usually better, than the costs of either the standard PrN 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. Furthermore, to abort a transaction usually entails no log write. Occasionally a tid_l record might need to be written, but it need not be forced.

The NPrC protocol does less logging than 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. In addition, NPrC sacrifices the ability to recover information used to garbage collect protocol database entries. This means that some information about transaction outcome may need to be retained forever. However, the amount of 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 PrC or NPrC, 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

Here we discuss some additional issues related to our NPrC commit protocol.

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 [9], 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 *tids*. 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 an explicit initiation record for it to the log. Later this record will be logically deleted by an (unforced) end record for an aborted transaction or a (forced) commit record for a committed transaction. We permit tid_l to be greater than the *tids* 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. This is the original PrC protocol, but we use it only for recalcitrant transactions.

It is important to note that transactions become recalcitrant only when they prevent us from advancing tid_l as we would like. That is, we do not need to identify these transactions at transaction initiation. When they cause us trouble with tid_l, we write their initiation log records and then log the advance of tid_l.

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 [8]). That is, it forces an initiation record to its log before continuing with this transaction, and in particular before forwarding this *tid* to other cohorts.

The important point is that the vast majority of transactions will not need initiation records and 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 one cannot externally distinguish the coordinator behavior used for logging any given transaction.

6.2 Transaction Timestamping

In [5], timestamped voting was used both to optimize 2PC and to provide each committed transaction with a timestamp that agrees with transaction serialization. This guarantees serializability even when transaction termination is not guaranteed, while permitting the read-only and other optimizations. 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

To support transaction-time databases in which versions of data are timestamped with the commit time of the transaction [6, 7], 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. Hence presumed abort (PrA), which remembers the committed transactions, is better in this case because it can simply keep the timestamps with its committed transaction entries. No form of presumed commit can be used.

6.2.2 Timestamps Only for Commit Protocol

So long as databases are using transaction timestamps *not* to timestamp data but solely as part of the commit protocol [5], 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: commit or abort. Presumed commit protocols can be used in these instances, and our NPrC protocol is not only applicable but desirable.

6.3 Garbage Collecting *IN*

If we knew all cohorts of the transactions active at the time of a crash, we would not need to retain *IN* forever. We could simply broadcast the news of the crash, including *IN*, to all such cohorts, wait for them to ACK this CRASH message, and then discard *IN*. If another crash occurs during this process, we simply repeat the message.

Knowing precisely which cohorts are involved in active transactions at the time of a crash is, of course, one of the reasons that PrC needs the extra forced log write. However, just as we do not need precise information about which transactions are active at the time of a crash, we also do not need precise information about the cohorts of these transactions. In both cases, a superset that bounds the size of the actual set is sufficient. The wasted CRASH messages sent to cohorts in the superset that are not actually involved in active transactions are a modest cost because crashes don't happen often. This can be an asynchronous background activity that is performed lazily.

Our problem thus reduces to knowing the superset of potential cohorts of transactions active at the time of a crash. This can be done by maintaining a cohort database. Each time a cohort that we have not seen before becomes involved in a transaction, we update this database. This requires a log record to make the update stable, and the affected part of the cohort database should eventually be written to disk as well. Such a cohort database should be small enough to stay in main memory, and its update activity should be very low. It might be initialized a priori with the set of expected or permitted cohorts. If the size of this database becomes a problem, we could occasionally delete entries that have not recently participated in transactions.

Acknowledgments

Johannes Klein, Jim Gray, and Alan Fekete read the technical report on which this paper is based and made helpful comments, especially about garbage collecting crash related information and about cohort failures.

References

[1] Gray, J. Notes on Database Systems. IBM Research Report RJ2188 (Feb.1978) San Jose, CA

[2] Gray, J. Minimizing the Number of Messages in Commit Protocols. *Workshop on Fundamental Issues in Distributed Computing*, Pala Mesa, CA (Feb. 1981), 90-92.

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

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

[5] Lomet, D. Using Timestamps to Optimize Two Phase Commit. *Proceedings of the PDIS Conference*, San Diego, CA (Jan 1993), 48-55.

[6] Lomet, D. and Salzberg, B. Access Methods for Multiversion Data. *Proc. ACM SIGMOD Conference*, Portland, OR (June 1989), 315-324.

[7] Lomet, D. and Salzberg, B. Transaction-time Databases. In *Temporal Databases: Theory, Design, and Implementation* (A. Tansel et al., editors), A. Benjamin Cummings, Redwood City, CA (Jan 1993).

[8] Mohan, C. and Lindsay, B. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. *Proc. 2nd Symposium on Principles of Distributed Computing*, Montreal, CA (Aug. 1983).

[9] 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.

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