ML-1-2PC: An Adaptive Multi-level Atomic Commit Protocol

Yousef J. Al-Houmaily[1] and Panos K. Chrysanthis[2]

[1] Dept. of Computer and Information Programs
Institute of Public Administration
Riyadh 11141, Saudi Arabia
houmaily@ipa.edu.sa

[2] Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
panos@cs.pitt.edu

Abstract. The one-two phase commit (1-2PC) protocol is a combination of a one-phase atomic commit protocol, namely, implicit yes-vote, and a two-phase atomic commit protocol, namely, presumed commit. The 1-2PC protocol integrates these two protocols in a dynamic fashion, depending on the behavior of transactions and system requirements, in spite of their incompatibilities. This paper extends the applicability of 1-2PC to the multi-level transaction execution model, which is adopted by database standards. Besides allowing incompatible atomic commit protocols to co-exist in the same environment, 1-2PC has the advantage of enhanced performance over the currently known atomic commit protocols making it more suitable for Internet database applications.

1 Introduction

The *two-phase commit* (2PC) protocol [9, 12] is one of the most widely used and optimized *atomic commit protocols* (ACPs). It ensures atomicity and independent recovery but at a substantial cost during normal transaction execution which adversely affects the performance of the system. This is due to the costs associated with its *message complexity* (i.e., the number of messages used for coordinating the actions of the different sites) and *log complexity* (i.e., the amount of information that needs to be stored in the stable storage of the participating sites for failure recovery). For this reason, there has been a re-newed interest in developing more efficient ACPs and optimizations. This is especially important given the current advances in electronic services and electronic commerce environments that are characterized by high volume of transactions where commit processing overhead is more pronounced. Most notable results that aim at reducing the cost of commit processing are *one-phase commit* (1PC) protocols such as *implicit yes-vote* (IYV) [4, 6] and *coordinator log* (CL) [19].

Although 1PC protocols are, in general, more efficient than 2PC protocols, 1PC protocols place assumptions on transactions or the database management

systems (DBMSs). Whereas some of these assumptions are realistic (i.e., reflect how DBMSs are usually implemented), others can be considered restrictive in some applications [1, 6]. For example, 1PC protocols restrict the implementation of applications that wish to utilize *deferred consistency constraints validation*, an option that is specified in the SQL standards.

The *one-two phase commit* (1-2PC) protocol attempts to achieve the best of the two worlds. Namely, the performance of 1PC and the wide applicability of 2PC. It is essentially a combination of 1PC (in particular, *IYV*) and 2PC (in particular, *Presumed Commit – PrC* [16]). It starts as 1PC and dynamically switches to 2PC when necessary. Thus, 1-2PC achieves the performance advantages of 1PC protocols whenever possible and, at the same time, the wide applicability of 2PC protocols. In other words, 1-2PC supports deferred constraints without penalizing those transactions that do not require them. Furthermore, 1-2PC achieves this advantage on a participant (cohort) basis within the same transaction in spite of the incompatibilities between the 1PC and 2PC protocols.

This paper extends the applicability of 1-2PC to the multi-level transaction execution (MLTE) model, the one adopted by database standards and implemented in commercial systems. The MLTE model is specially important in the context of Internet transactions since they are hierarchical in nature, making 1-2PC more suitable for Internet database applications.

In Section 2, we review PrC, IYV and 1-2PC. *Multi-level 1-2PC* is introduced in Section 3. The performance of 1-2PC is analytically evaluated in Section 4.

2 Background

A distributed/Internet transaction accesses data by submitting operations to its *coordinator*. The coordinator of a transaction is assumed to be the *transaction manager* at the site where the transaction is initiated. Depending on the data distribution, the coordinator decomposes the transaction into a set of *subtransactions*, each of which executes at a single participating database site (*cohort*).

In the *multi-level transaction execution* (MLTE) model, it is possible for a cohort, to decompose its assigned subtransactions further. Thus, a transaction execution can be represented by a multi-level execution tree with its coordinator at the root, and with a number of intermediate and leaf cohorts. When the transaction finishes its execution and submits its final commit request, the coordinator initiates an atomic commit protocol.

2.1 Presumed Commit Two-Phase Commit Protocol

Presumed Commit (PrC) [16] is one of the best known variants of the two-phase commit protocol which consist of a *voting phase* and a *decision phase*. During the voting phase, the coordinator requests all cohorts to *prepare to commit* whereas, during the decision phase, the coordinator either commits the transaction if *all* cohorts are prepared-to-commit (voted "yes"), or aborts the transaction if any cohort has decided to abort (voted "no").

In general, when a cohort receives the final decision and complies with the decision, it sends an acknowledgment (ACK). ACKs enable a coordinator to discards all information pertaining to a transaction from its *protocol table* (that is kept in main memory), and forgets the transaction. Once the coordinator receives ACKs from all the cohorts, it knows that all cohorts have received the decision and none of them will inquire about the status of the transaction in the future. In PrC, cohorts ACK only abort decisions and not commit ones. A coordinator removes a transaction from its protocol table either when it makes a commit decision or when it receives ACKs from all cohorts in the case of abort decision. This means that in case of a status inquiry, a coordinator can interpret lack of information on a transaction to indicate a commit decision.

In PrC, misinterpretation of missing information as a commit after a coordinator's failure is avoided by requiring coordinators to record in a force written initiation log record all the cohorts for each transaction before sending prepare to commit messages to the cohorts. To commit a transaction, the coordinator force writes a commit record to logically eliminate the **initiation** record of the transaction and then sends out the commit decision. When a cohort receives the decision, it writes a non-forced commit record and commits the transaction without having to ACK the decision. After a coordinator or a cohort failure, if the cohort inquires about a committed transaction, the coordinator, not remembering the transaction, will direct the cohort to commit it (by presumption).

To abort a transaction, the coordinator does not write an abort decision in its log. Instead, it sends out the abort decision and waits for ACKs. When a cohort receives the decision, it force writes an abort record and sends an ACK.

In the MLTE model, the behavior of the root coordinator and each leaf cohort remains the same as in two-level transactions. The only difference is the behavior of *cascaded coordinators* (i.e., non-root and non-leaf cohorts) which behave as leaf cohorts with respect to their direct ancestors and root coordinators with respect to their direct descendants. In *multi-level PrC*, each cascaded coordinator has to force write an initiation record before propagating the prepare to commit message to its descendant cohorts. On abort decision, it notifies its descendants, force writes an abort record and, then, acknowledge its ancestor. It forgets the transaction when it receives ACKs from all its descendants. On commit decision, a cascaded coordinator propagates the decision to its descendants, writes a non-forced commit record and, then, forgets the transaction.

2.2 Implicit Yes-Vote One-Phase Commit Protocol

Unlike PrC, the *implicit yes-vote* (IYV) [4, 6] protocol consist of only a single phase which is the decision phase. The (explicit) voting phase is eliminated by overlapping it with the ACKs of the database operations. IYV assumes that each site deploys (1) a *strict two-phase locking* and (2) *physical page–level replicated–write–ahead logging* with the undo phase *preceding* the redo phase for recovery.

In IYV, when the coordinator of a transaction receives an ACK from a cohort regarding the completion of an operation, the ACK is *implicitly* interpreted to mean that the transaction is in a prepared-to-commit state at the cohort. When

the cohort receives a new operation for execution, the transaction becomes active again at the cohort and can be aborted, for example, if it causes a deadlock or violation to any of the site's database consistency constraints. If the transaction is aborted, the cohort responds with a *negative ACK* message (NACK). Only when all the operations of the transaction are executed and acknowledged by their perspective cohorts, the coordinator commits the transaction. Otherwise, it aborts the transaction. In either case, the coordinator propagates its decision to all the cohorts and waits for their ACKs.

IYV handles cohort failures by partially replicating its log rather than force writing the log before each ACK. Each cohort includes the *redo* log records that are generated during the execution of an operation in the operation's ACK. Each cohort also includes the *read* locks acquired during the execution of an operation in the ACK in order to support the option of *forward recovery* [6]. After a crash, a cohort reconstructs the state of its database, which includes its log and lock table as it was just prior to the failure with the help of the coordinators. To limit the number of coordinators that need to be contacted after a site failure, each cohort maintains a *recovery-coordinators' list* (RCL) which is kept in the stable log. At the same time, by maintaining a local log and using WAL, each cohort is able to undo the effects of aborted transactions locally using only its own log.

In *multi-level IYV*, the behavior of a root coordinator and leaf cohorts remains the same as in IYV, whereas cascaded coordinators are responsible about the coordination of ACKs of individual operations.

As in the case of the (two-level) IYV, only a root coordinator maintains a replicated redo log for each of the cohorts. When a cascaded coordinator receives ACKs from all its descendants that participated in the execution of an operation, it sends an ACK to its direct ancestor containing the redo log records generated across all cohorts and the read locks held at them during the execution of the operation. Thus, after the successful execution of each operation, root coordinator knows all the cohorts (i.e., both leaf and cascaded coordinators) in a transaction. Similarly, each cohort knows the identity of the root coordinator which is reflected in its RCL. The identity of the root coordinator is attached to each operation send by the root and cascaded coordinators.

While the execution phase of a transaction is multi-level, the decision phase is not. Since the root coordinator knows all the cohorts at the time the transaction finishes its execution it sends its decision directly to each cohort without going through cascaded coordinators. Similarly, each cohort sends its ACK of the decision directly to the root coordinator. This is similar to the flattening of the commit tree optimization [17].

2.3 The 1-2PC Protocol

1-2PC is a composite protocol that inter-operates IYV and PrC in a practical manner in spite of their incompatibilities. In 1-2PC, a transaction starts as 1PC at each cohort and continuous this way until the cohort executes a deferred consistency constraint. When a cohort executes such a constraint, it means that the constraint needs to be synchronized at commit time. For this reason, the

cohort switches to 2PC and sends an *unsolicited deferred consistency constraint* (UDCC) message to the coordinator. The UDCC is a flag that is set as part of a *switch* message, which also serves as an ACK for the operation's successful execution. When the coordinator receives the switch message, it switches the protocol used with the cohort to 2PC.

When a transaction sends its final commit primitive, the coordinator knows which cohorts are 1PC and which cohorts are 2PC. If all cohorts are 1PC (i.e., no cohort has executed deferred constraints), the coordinator behaves as an IYV coordinator. On the other hand, if all cohorts are 2PC, the coordinator behaves as a PrC coordinator with the exception that the initiation log record (of PrC) is now called a *switch* log record.

When the cohorts are mixed 1PC and 2PC in a transaction's execution, the coordinator resolves the incompatibilities between the two protocols as follows: (1) It " talks" IYV with 1PC cohorts, and PrC with 2PC cohorts and (2) initiates the voting phase with 2PC cohorts before making the final decision and propagating the final decision to all cohorts. This is because a "no" vote from a 2PC cohort is a *veto* that aborts a transaction. Further, in order to be able to reply to the inquiry messages of the cohorts after failures, 1-2PC synchronizes the timing at which it forgets the outcome of terminated transactions. A coordinator forgets the outcome of a committed transaction when all 1PC cohorts ACK the commit decision, and the outcome of an aborted transaction when all 2PC cohorts ACK the abort decision. In this way, when a cohort inquires about the outcome of a forgotten transaction, the coordinator replies with a decision that matches the presumption of the protocol used by the cohort which is always consistent with the actual outcome of the transaction.

1-2PC has been optimized for *read-only* transactions and for context-free transactions with a *forward recovery* option [3] but never extended for multi-level transactions which is done in the next section.

3 The Multi-Level 1-2PC Protocol

Extending the 1-2PC for multi-level transactions, there are three cases to consider: (1) all cohorts are 1PC, (2) all cohorts are 2PC and (3) cohorts are mixed 1PC and 2PC. We discuss each of these cases in the following three sections.

3.1 All Cohorts are 1PC

In the multi-level 1-2PC, the behavior of the root coordinator and each leaf cohort in the transaction execution tree remains the same as in two-level 1-2PC. The only difference is the behavior of cascaded coordinators which is similar to that of the cascaded coordinators in the multi-level IYV. Since an operation's ACK represents the successful execution of the operation at the cascaded coordinator and all its descendants that have participated in the operation's execution, the cascaded coordinator has to wait until it receives ACKs from the required descendants before sending the (collective) ACK and redo log records to its direct

coordinator in the transaction execution tree. Thus, when a transaction finishes its execution, all its redo records are replicated at the root coordinator's site. As in the two-level 1-2PC, only root coordinators are responsible for maintaining the replicated redo log records and a root coordinator knows all the cohorts (i.e., both leaf and cascaded coordinators).

The identity of the root coordinator is attached to each operation send by the root and cascaded coordinators. When a cohort receives an operation from a root coordinator for the first time, it records the coordinator's identity in its RCL and force writes its RCL into stable storage. A cohort removes the identity of a root coordinator from its RCL, when it commits or aborts the last transaction submitted by the root coordinator.

As in IYV, if a cohort fails to process an operation, it aborts the transaction and sends a NACK to its direct ancestor. If the cohort is a cascaded coordinator, it also sends an abort message to each implicitly prepared cohort Then, the cohort forgets the transaction. When the root or a cascaded coordinator receives NACK from a direct descendant, it aborts the transaction and sends abort messages to all direct descendants and forgets the transaction. The root coordinator behaves similarly when it receives an abort request from a transaction.

On the other hand, if the root coordinator receives a commit request from the transaction after the successful execution of all its operations, the coordinator commits the transaction. On a commit decision, the coordinator force writes a commit log record and then sends commit messages to each of its direct descendants. If a descendant is a leaf cohort, it commits the transaction, writes a non-forced log record and, when the log record is flushed into the stable log, it acknowledges the commit decision.

If the cohort is a cascaded coordinator, the cohort commits the transaction, forwards a commit message to each of its direct descendants and writes a non-forced commit log record. When the cascaded coordinator receives ACKs from all its direct descendants and the commit log record that it wrote had been flushed into the stable log, the cohort acknowledges the commit decision to its direct ancestor. Thus, the ACK serves as a collective ACK for the entire cascaded coordinator's branch.

3.2 All Cohorts are 2PC

At the end of the transaction execution phase, the coordinator declares the transaction as 2PC if all cohorts have switched to 2PC. When all cohorts are 2PC, 1-2PC can be extended to the MLTE model in a manner similar to the multi-level PrC which we briefly discussed in Section 2.1 and detailed in [5]. However, multi-level 1-2PC is designed in such a way that 1-2PC does not realize the commit presumption of PrC on every two adjacent levels of the transaction execution tree. In this respect, it is similar to the *rooted PrC* which reduces the cost associated with the initiation records of PrC [5].

Specifically, cascaded coordinators do not force write switch records which are equivalent to the *initiation* records of PrC and, consequently, do not presume commitment in the case that they do not remember transactions. For this reason,

in multi-level 1-2PC, the root coordinator needs to know *all* the cohorts at all levels in a transaction's execution tree. Similarly, each cohort needs to know *all* its ancestors in the transaction's execution tree. The former allows the root coordinator to determine when it can *safely* forget a transaction while the latter allows a prepared to commit cohort at any level in a transaction's execution tree to find out the final *correct* outcome of the transaction, even if intermediate cascaded coordinators have no recollection about the transaction due to a failure.

In order for the root coordinator to know the identities of all cohorts, each cohort includes its identity in the ACKs of the *first* operation that it executes. When a cascaded coordinator receives such an ACK from a cohort, it also includes its identity in the ACK. In this way, the identities of all cohorts and the chain of their ancestors are propagated to the root coordinator. When the transaction submits its commit request, assuming that that all cohorts have requested to switch to 2PC during the execution of the transaction, the coordinator force writes a *switch* record, as in two-level 1-2PC. The switch log record includes the identities of all cohorts in the transaction execution tree. Then, it sends out prepare to commit messages to its direct descendants.

When the coordinator sends the prepare to commit message, it includes its identity in the message. When a cascaded coordinator receives the prepare to commit message, it appends its own identity to the message before forwarding it to its direct descendants. When a leaf cohort receives a prepare to commit message, it copies the identities of its ancestors in the prepared log record before sending its "Yes" vote. When a cascaded coordinator receives "Yes" votes from all its direct descendants, the cascaded coordinator also records the identities of its ancestors as well as its descendants in its prepared log record before sending its collective "Yes" vote to its direct ancestor.

If any direct descendant has voted "No", the cascaded coordinator force writes an abort log record, sends a "No" vote to its direct ancestor and an abort message to each direct descendant that has voted "Yes" and waits for their ACKs. Once all the abort ACKs arrive, the cascaded coordinator writes a non-forced end record and forgets the transaction.

As in multi-level PrC, when the root coordinator receives "Yes" votes from all its direct descendants, it force writes a commit record, sends its decision to its direct descendants and forgets the transaction. When a cascaded coordinator receives a commit message, it commits the transaction, propagates the message to its direct descendants, writes a non-forced commit record and forgets the transaction. When a leaf cohort receives the message, it commits the transaction and writes a non-forced commit record.

If the root coordinator receives a "No" vote, it sends an abort decision to all direct descendants that have voted "Yes" and waits for their ACKs, knowing that all the descendants of a direct descendant that has voted "No" have already aborted the transaction. When the coordinator receives all the ACKs, it writes a non-forced end record and forgets the transaction. When a cascaded coordinator receives the abort message, it behaves as in multi-level PrC. That is, it propagates the message to its direct descendants and writes a forced abort record.

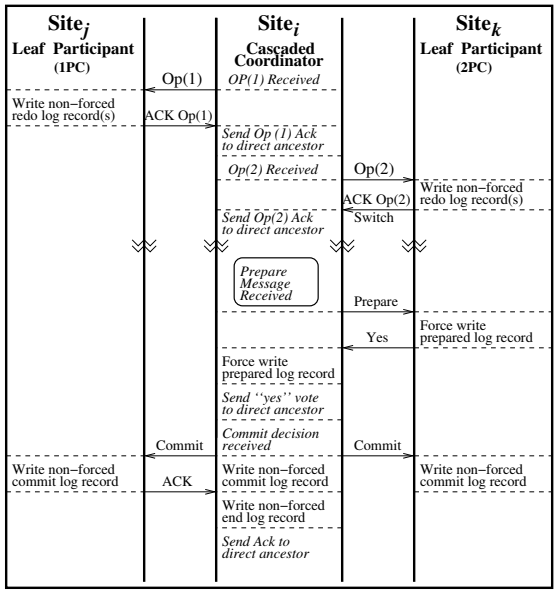

Fig. 1. Mixed cohorts in a 2PC cascaded coordinator's branch (commit case).

Then, it acknowledges its direct ancestor. Once the cascaded coordinator has received ACKs from all its direct descendants, it writes a non-forced end record and forgets the transaction. When a leaf cohort receives the abort message, it first force writes an abort record and, then, acknowledges its direct ancestor.

3.3 Cohorts are Mixed 1PC and 2PC

Based on the information received from the different cohorts during the execution of a transaction, at commit time the coordinator of the transaction knows the protocol of each of the cohorts. It also knows the execution tree of the transaction. That is, it knows all the ancestors of each cohort and whether a cohort is a cascaded coordinator or a leaf cohort. Based on this knowledge, the coordinator considers a direct descendant to be 1PC if the descendant and *all* the cohorts in its branch are 1PC, and 2PC if the direct descendant or *any* of the cohorts in its branch is 2PC. For a 1PC branch, the coordinator uses the 1PC part of multi-level 1-2PC with the branch, as we discussed above (Section 3.1). For a 2PC branch, the coordinator uses 2PC regardless of whether the direct descendant is 1PC or 2PC. That is, the coordinator uses the 2PC part of multi-level 1-2PC discussed in the previous section (Section 3.2). Thus, with the exception in the way a coordinator's decide on which protocol to use with each of its direct descendants, the coordinator's protocol proceeds as in the two-level 1-2PC.

For leaf cohorts, each cohort behaves exactly in the same way as in two-level 1-2PC regardless of whether the leaf cohort descends from a 1PC or 2PC branch. That is, a cohort behaves as 1PC cohort if it has not requested to switch protocol or as 2PC if has made such a request during the execution of the transaction.

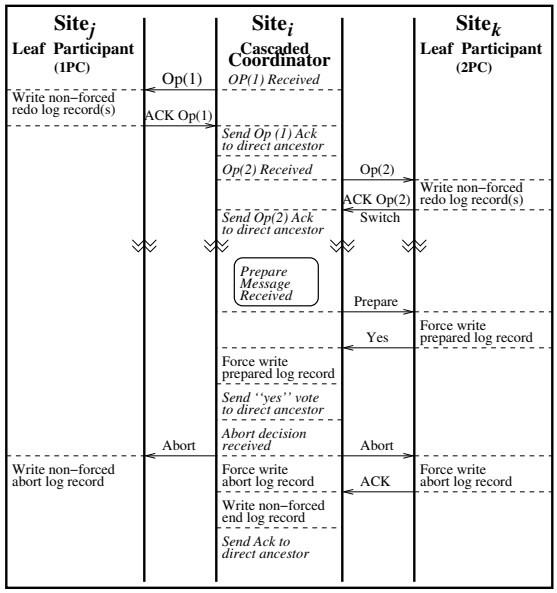

Fig. 2. Mixed cohorts in a 2PC cascaded coordinator's branch (abort case).

On the other hand, the behavior of cascaded coordinators is different and depends on the types of its descendant cohorts in the branch. A cascaded coordinator uses multi-level 1PC when all the cohorts in its branch, including itself, are 1PC. Similarly, a cascaded coordinator uses multi-level 2PC when all the cohorts in the branch, including itself, are 2PC. Thus, in the above two situations, a cascaded coordinator uses multi-level 1-2PC as we discussed it in the previous two sections, respectively.

When the protocol used by a cascaded coordinator is different than the protocol used by at least one of its descendants (not necessarily a direct descendant), there are two scenarios to consider. Since, for each scenario, cascaded coordinators behave the same way at any level of the transaction execution tree, below we discuss the case of the last cascaded coordinator in a branch.

2PC cascaded coordinator with 1PC cohort(s) When a 2PC cascaded coordinator receives a prepare message from its ancestor after the transaction has finished its execution, the cascaded coordinator forwards the message to each 2PC cohort and waits for their votes. If any cohort has decided to abort, the cascaded coordinator force writes an abort log record, then, sends a "no" vote to its direct ancestor and an abort message to each prepared cohort (including 1PC cohorts). Then, it waits for the ACKs from the prepared 2PC cohorts. Once it receives the required ACKs, it writes a non-forced end log record and forgets the transaction. On the other hand, if all the 2PC cohort have voted "yes" and the cascaded coordinator's own vote is a "yes" vote too, the cascaded coordinator force writes a prepared log record and then sends a (collective) "yes" vote of the

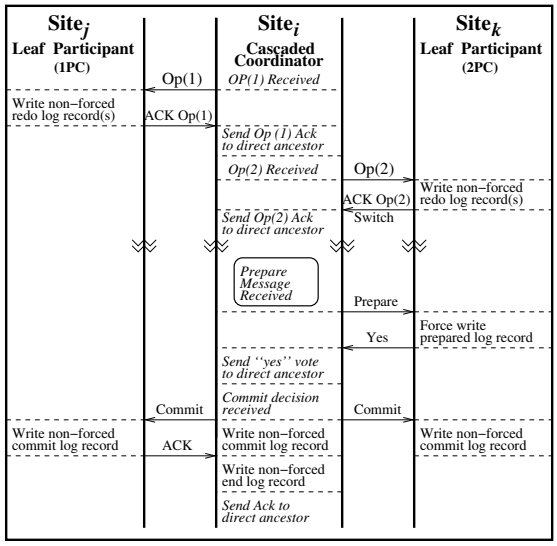

Fig. 3. Mixed cohorts in a 1PC cascaded coordinator's branch (commit case).

branch to the its direct ancestor, as shown in Figure 1. Then, it waits for the final decision.

If the final decision is a commit (Figure 1), the cascaded coordinator forwards the decision to each of its direct descendants (both 1PC and 2PC), and writes a commit log record. The commit log record of the cascaded coordinator is written in a non-forced manner, following PrC protocol. Unlike PrC, however, a cascaded coordinator expects each 1PC cohort to acknowledge the commit message but not 2PC cohorts since they follow PrC. When a cascaded coordinator receives ACKs from 1PC cohorts, it writes a non-forced end log record. When the end record is written into the stable log due to a subsequent forced write of a log record or log buffer overflow, the cascaded coordinator sends a collective ACK to its direct ancestor and forgets the transaction.

On the other hand, if the final decision is an abort (Figure 2), the cascaded coordinator sends an abort message to each of its descendants and writes a forced abort log record (following PrC protocol). When 2PC cohorts acknowledge the abort decision, the cascaded coordinator writes a non-forced end log record. Once the end record is written onto stable storage due to a subsequent flush of the log buffer, the cascaded coordinator sends an ACK to its direct ancestor and forgets the transaction.

Notice that, unlike two-level 1-2PC, a 2PC cohort that is cascaded coordinator has to acknowledge both commit and abort decisions. A commit ACK reflects the ACKs of all 1PC cohorts while an abort ACK reflects the ACKs of all 2PC cohorts (including the cascaded coordinator's ACK).

1PC cascaded coordinator with 2PC cohort(s) As mentioned above, a 1PC cascaded coordinator with 2PC cohorts is dealt with as 2PC with respect

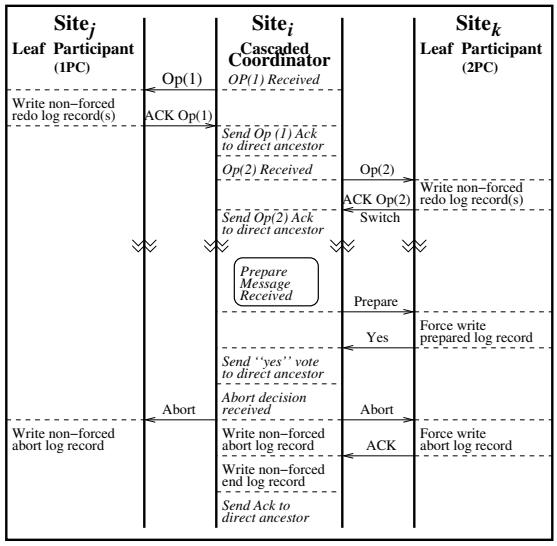

Fig. 4. Mixed cohorts in a 1PC cascaded coordinator's branch (abort case).

to messages. Specifically, when a 1PC cascaded coordinator receives a prepare message from its ancestor, it forwards the message to each 2PC cohort and waits for their votes. If any cohort has decided to abort, the cascaded coordinator force writes an abort log record, then, sends a "no" vote to its direct ancestor and an abort message to each prepared cohort (including 1PC cohorts). Then, it waits for the abort ACKs from the prepared 2PC cohorts. Once the cascaded coordinator receives the required ACKs, it writes a non-forced end log record and forgets the transaction. On the other hand, if all the 2PC cohort have voted "yes", the cascaded coordinator sends a (collective) "yes" vote of the branch to the its direct ancestor, as shown in Figure 3, and waits for the final decision.

If the final decision is a commit (Figure 3), the cascaded coordinator forwards the decision to each of its direct descendants (both 1PC and 2PC), and writes a commit log record. The commit log record of the cascaded coordinator is written in a non-forced manner, following IYV protocol. Unlike IYV, however, a cascaded coordinator expects each 1PC cohort to acknowledge the commit message but not 2PC cohorts since they follow PrC. When a cascaded coordinator receives ACKs from 1PC cohorts, it writes a non-forced end log record. When the end record is written into the stable log due to a subsequent forced write of a log record or log buffer overflow, the cascaded coordinator sends a collective ACK to its direct ancestor and forgets the transaction.

On the other hand, if the final decision is an abort (Figure 4), the cascaded coordinator sends an abort message to each of its descendants and writes a non-forced abort log record (following IYV protocol). When 2PC cohorts acknowledge the abort decision, the cascaded coordinator writes a non-forced end log record. Once the end record is written onto stable storage due to a subsequent flush to

the log buffer, the cascaded coordinator sends an ACK to its direct ancestor and forgets the transaction.

As in the case of a 2PC cascaded coordinator with mixed cohorts, a 1PC cohort that is cascaded coordinator has to acknowledge both commit as well as abort decisions. A commit ACK reflects the ACKs of all 1PC cohorts (including the cascaded coordinator's ACK) while an abort ACK reflects the ACKs of all 2PC cohorts.

3.4 Recovering from Failures

As in all other atomic commit protocols, site and communication failures are detected by *timeouts*. If the root coordinator times out while awaiting the vote of one of its direct descendants, it makes an abort final decision, sends abort messages to all its direct descendants and wait for their ACKs to complete the protocol. Similarly, if a cascaded coordinator times out while awaiting the vote of one of its direct descendants, it makes an abort decision. It also force writes an abort log record, sends a "no" vote to its direct ancestor and abort messages to all its direct descendants and waits for their abort ACKs.

After a site failure, during its recovery process, a 2PC leaf cohort inquires its direct ancestor about the outcome of each prepared to commit transaction. In its inquiry message, the cohort includes the identities of its ancestors recorded in the prepared log record. In this way, if the direct ancestor of the prepared cohort does not remember the transaction, it uses the list of ancestors included in the inquiry message to inquire its own direct ancestor about the transaction's outcome rather than replying with a commit message by presumption. (Recall that a 2PC cascaded coordinator does not write initiation records for transactions, therefore, it cannot presume commitment in the absence of information about a transaction.) Eventually, either one of the cascaded coordinators in the path of ancestors will remember the transaction and provide a reply, or the inquiry message will finally reach the root coordinator. The root coordinator will respond with the appropriate decision if it remembers the outcome of the transaction or will respond with a commit decision by presumption. Once the cohort receives the reply message, it enforces the decision and sends an ACK only if the decision is abort.

On the other hand, if the leaf cohort is a 1PC cohort, the cohort uses its list of RCL to resolve the status of those transactions that were active prior to the failure, as in IYV. Specifically, the cohort inquires each of the coordinators recorded in its RCL with a recovering message. Once the repair messages arrive from the listed coordinators, the cohort repairs its log by applying the missing redo records and finish its recovery procedure. If the failure is a communication failure and the cohort is left blocked in an implicit prepared state, the cohort keeps inquiring its direct ancestor until it receives a final decision. Once the final decision arrives, the cohort continues its protocol as during normal processing.

In the event that the root coordinator fails, during its recovery process, the root coordinator identifies and records in its protocol table each transaction with

a switch log record without a corresponding commit or end record. These transactions have not finished their commit processing by the time of the failure and need to be aborted. For each of these transactions, the coordinator sends an abort message to its direct descendants, as recorded in the switch record, along with their lists of descendants in the transaction execution tree. The recipient of the abort message can be either a cascaded coordinator or a leaf cohort. In the case of a cascaded coordinator, if it is in a prepared-to-commit state, the cascaded coordinator behaves as in the case of normal processing discussed above. Otherwise, it responds with a *blind* ACK, indicating that it has already aborted the transaction. Similarly, if the abort message is received by a leaf cohort, the cohort behaves as in the case of normal processing if it is in a prepared-to-commit state or replies with a blind ACK.

Similarly, for each transaction with each that has a commit log record but without corresponding switch and end record, the coordinator knows that all cohorts in this transaction execution are 1PC and the transaction has not finished the protocol before the failure. For each of these transactions, the coordinator adds the transaction in its protocol table and sends a commit message to each of its direct ancestors. Then, the coordinator waits for the ACKs of the direct descendants. Once the required ACKs arrive, the coordinator writes an end log record and forgets the transaction.

In the case of a 2PC cascaded coordinator failure, during the recovery process, the cascaded coordinator adds to its protocol table each *undecided* transaction (i.e., a transaction that has a prepared record without a corresponding final decision record) and each decided (i.e., committed or aborted) transaction that has not been fully acknowledged by its direct descendants prior to the failure. For each undecided transaction, the cascaded coordinator inquires its direct ancestor about the outcome of the transaction. As in the case of a leaf cohort failure, the inquiry message contains the identities of all ancestors as recorded in the prepared record. Once the cascaded coordinator receives the final decision, it completes the protocol as in the normal processing case discussed above. For each decided but not fully acknowledged transaction, the cascaded coordinator re-sends decision messages to its direct descendants (according to the protocol specification) and waits for all their ACKs before completing the protocol as during normal processing, e.g., by writing a non-forced end log record.

4 Analytical Evaluation

In this section, we evaluate the performance of 1-2PC and compare it with the performance of PrC and IYV. In our evaluation, we also include the *presumed abort* (PrA) protocol [16] which is the other best known 2PC variant. As opposed to PrC, PrA coordinators assume that lack of information on a transaction indicates an aborted transaction. This eliminates the need for an abort log record at the coordinator and the need of an ACK and force write abort decision log records at the cohorts.

	PrC	PrA	IYV	1-2PC (1PC)	1-2PC (2PC)	1-2PC (MIX)
Log force delays	$2d$-1	d	1	1	d+1	3 ~ (d+1)
Total forced log writes	$2c+l+2$	$2c+2l+1$	1	1	$c+l+2$	n-p+2
Message delays (Commit)	2(d-1)	2(d-1)	0	0	2(d-1)	2 ~ (2d-1)
Message delays (Locks)	3(d-1)	3(d-1)	d-1	d-1	3(d-1)	(2+d) ~ (3(d-1)
Total messages	$3n$	$4n$	$2n$	$2n$	$3n$	$4c_{Mix}+2p_{1PC}+3p_{2PC}$
Total messages with piggybacking	$3n$	$3n$	n	n	$3n$	$3c_{Mix}+2p_{1PC}+3p_{2PC}$

Table 1. The cost of the protocols to *commit* a transaction.

	PrC	PrA	IYV	1-2PC (1PC)	1-2PC (2PC)	1-2PC (MIX)
Log force delays	$2d$-2	d-1	0	0	d	2 ~ d
Total forced log writes	$3c+2l+1$	$c+l$	0	0	$2c+2l+1$	2(n-p)+1
Message delays (Abort)	2(d-1)	2(d-1)	0	0	2(d-1)	2 ~ (2d-1)
Message delays (Locks)	3(d-1)	3(d-1)	d-1	d-1	3(d-1)	(2+d) ~ 3(d-1)
Total messages	$4n$	$3n$	n	n	$4n$	$4c_{Mix}+p_{1PC}+4p_{2PC}$
Total messages with piggybacking	$3n$	$3n$	n	n	$3n$	$3c_{Mix}+p_{1PC}+3p_{2PC}$

Table 2. The cost of the protocols to *abort* a transaction.

Our evaluation method is based on evaluating the *log*, *message* and *time* (message delay) complexities. In the evaluation, we consider the number of coordination messages and forced log writes that are due to the protocols only (e.g., we do not consider the number of messages that are due to the operations and their acknowledgments). The costs of the protocols in both commit and abort cases are evaluated during normal processing.

Tables 1 and 2 compare the costs of the different protocols for the commit case and abort case on a per transaction basis, respectively. The column titled "1-2PC (1PC)" denotes the 1-2PC protocol when *all* cohorts are 1PC, whereas the column titled "1-2PC (2PC)" denotes the 1-2PC protocol when *all* cohorts are 2PC. The column titled "1-2PC (MIX)" denotes the 1-2PC protocol in the presence of a mixture of both 1PC and 2PC cohorts. In the table, n denotes the total number of sites participating in a transaction's execution (excluding the coordinator's site), p denotes the number of 1PC cohorts (in the case of 1-2PC protocol), c denotes a cascaded coordinator, l denotes a leaf cohort and d denotes the depth of the transaction execution tree assuming that the root coordinator resides at level "1".

The row labeled "Log force delays" contains the sequence of forced log writes that are required by the different protocols up to the point that the commit/abort decision is made. The row labeled "Message delays (Decision)" contains the number of sequential messages up to the commit/abort point, and the row labeled "Message delays (Locks)" contains the number of sequential messages that are involved in order to release all the locks held by a committing/aborting transaction at the cohorts' sites. In the row labeled "Total messages with piggybacking", we apply *piggybacking* of the ACKs, which is a special case of the lazy commit optimization to eliminate the final round of messages.

It is clear from Tables 1 and 2 that 1-2PC performs as IYV when all cohorts are 1PC cohorts, outperforming the 2PC variants in all performance measures including the number of log force delays to reach a decision as well as the total

number of log force writes. For the commit case, the two protocols require only one forced log write whereas for the abort case neither 1-2PC nor IYV force write any log records. When all cohorts are 2PC, 1-2PC performs by about d less in the number of sequential forced log writes and c less in the total forced log writes for both the commit as well as the abort case. This makes the performance enhancement of 1-2PC much more significant in the presence of deep execution trees. This performance enhancement is reflected on the 1-2PC when there is a cohorts' mix where the costs associated with log force delays, message delays to reach a decision and message delays to commit depends on the number of sequential 2PC cohorts as well as their positions in the execution tree.

Piggybacking can be used to eliminate the final round of messages for the commit case in PrA, IYV and 1-2PC (1PC). That is not the case for PrC, and 1-2PC (2PC) because a commit decision is never acknowledged in these protocols. Similarly, this optimization can be used in the abort case with PrC and 1-2PC (2PC) but not with PrA, IYV or 1-2PC (1PC) since a cohort in the latter set of protocols never acknowledges an abort decision. 1-2PC (MIX) benefits from this optimization in both commit and abort cases. This is because, in a commit case, a 1PC leaf cohort and each cascaded coordinator with mixed cohorts acknowledge the commit decision, whereas in an abort case, a 2PC leaf cohort and each cascaded coordinator with mixed cohorts acknowledge the abort decision, which can be both piggybacked.

Finally, the performance of multi-level 1-2PC can be further enhanced by applying three optimizations: read-only, forward recovery and flatting of the commit tree. These were not discussed in this paper due to space limitations.

5 Conclusions

Recently, there has been a re-newed interest in developing new atomic commit protocols for different database environments. These environments include gigabit-networked, mobile and real-time database systems. The aim of these efforts is to develop new and optimized atomic commit protocols that meet the special characteristics and limitations of each of these environments.

The 1-2PC protocol was proposed to achieve the performance of one-phase commit protocols when they are applicable, and the (general) applicability of two-phase commit protocols, otherwise. The 1-2PC protocol clearly alleviates the applicability shortcomings of 1PC protocols in the presence of (1) deferred consistency constraints, or (2) limited network bandwidth. At the same time, it keeps the overall protocol overhead below that of 2PC and its well known variants, namely presumed commit and presumed abort. For this reason, we extended 1-2PC to the multi-level transaction execution model, the one specified by the database standards and adopted in commercial database systems. We also evaluated its performance and compared it to other well known commit protocols. Our extension to 1-2PC and the results of our evaluation demonstrates the practicality and efficiency of 1-2PC, making it a specially important choice for Internet transactions that are hierarchical in nature.

References

1. Abdallah, M., R. Guerraoui and P. Pucheral. One-Phase Commit: Does it make sense? *Proc. of the Int'l Conf. on Parallel and Distributed Systems*, 1998.
2. Abdallah, M., R. Guerraoui and P. Pucheral. Dictatorial Transaction Processing: Atomic Commitment without Veto Right. *Distributed and Parallel Databases*, 11(3):239-268, 2002.
3. Al-Houmaily, Y. J. and P. K. Chrysanthis. 1-2PC: The One-Two Phase Atomic Commit Protocol. *Proc. of the ACM Annual Symp. on Applied Computing*, pp 684-691, 2004.
4. Al-Houmaily, Y. J. and P. K. Chrysanthis. Two Phase Commit in Gigabit-Networked Distributed Databases. *Proc. of the Int'l Conf. on Parallel and Distributed Computing Systems*, pp. 554–560, 1995.
5. Al-Houmaily, Y., P. Chrysanthis and S. Levitan. An Argument in Favor of the Presumed Commit Protocol. *Proc. of the Int'l Conf. on Data Engineering*, pp. 255–265, 1997.
6. Al-Houmaily, Y. J. and P. K. Chrysanthis. An Atomic Commit Protocol for Gigabit-Networked Distributed Database Systems. *Journal of Systems Architecture*, 46(9):809–833, 2000.
7. Attalui and Salem. The Presumed-Either Two-Phase Commit Protocol, *IEEE TKDE*, 14(5):1190–1196, 2002.
8. Chrysanthis, P. K. , G. Samaras and Y. Al-Houmaily. Recovery and Performance of Atomic Commit Processing in Distributed Database Systems. *Recovery Mechanisms in Database Systems*, V. Kumar and M. Hsu, Eds., Prentice Hall, 1998.
9. Gray, J. Notes on Data Base Operating Systems. In Bayer R., R.M. Graham, and G. Seegmuller (Eds.), *Operating Systems: An Advanced Course*, LNCS-60:393–481, 1978.
10. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
11. Gupta, R., J. Haritsa and K. Ramamritham. Revisiting Commit Processing in Distributed Database Systems. *Proc. of ACM SIGMOD*, pp. 486–497, 1997.
12. Lampson, B. Atomic Transactions. *Distributed Systems: Architecture and Implementation - An Advanced Course*, B. Lampson (Ed.), LNCS-105:246–265, 1981.
13. Lampson, B. and D. Lomet. A New Presumed Commit Optimization for Two Phase Commit. *Proc. of VLDB*, pp. 630-640, 1993.
14. Liu, M., D. Agrawal and A. El Abbadi. The Performance of Two Phase Commit Protocols in the Presence of Site Failures. *Distributed and Parallel Databases*, 6(2):157–182, 1998.
15. Mohan, C. and B. Lindsay. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. *Proc. of the ACM Symp. on Principles of Distributed Computing*, 1983.
16. Mohan, C., B. Lindsay and R. Obermarck. Transaction Management in the R^* Distributed Data Base Management System. *ACM TODS*, 11(4):378–396, 1986.
17. Samaras, G., K. Britton, A. Citron and C. Mohan. Two-Phase Commit Optimizations in a Commercial Distributed Environment. *Distributed and Parallel Databases*, 3(4):325–360, 1995.
18. Samaras, G., G. Kyrou, P. K. Chrysanthis. Two-Phase Commit Processing with Restructured Commit Tree. *Proc. of the Panhellenic Conf. on Informatics*, LNCS-2563:82–99, 2003.
19. Stamos, J. and F. Cristian. Coordinator Log Transaction Execution Protocol. *Distributed and Parallel Databases*, 1(4):383–408, 1993.
20. Tal, A. and R. Alonso. Integration of Commit Protocols in Heterogeneous Databases. *Distributed and Parallel Databases*, 2(2):209–234, 1994.