Two-Phase Commit in Gigabit-Networked Distributed Databases*

Yousef J. Al-Houmaily
Dept. of Electrical Engineering
University of Pittsburgh
Pittsburgh, PA 15261

Panos K. Chrysanthis
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

In the future, different database sites will be interconnected via gigabit networks, forming a very powerful distributed database system. In such an environment, the propagation latency will be the dominant component of the overall communication cost while the migration of large amount of data will not pose a problem. Furthermore, computer systems are expected to become highly reliable. In this paper, we present a two-phase commit *variant that exploits these new domain characteristics to minimize the cost of distributed transaction commitment. Although the protocol trades off efficiency during normal processing for slower recovery, it supports* forward recovery *that potentially reduces the overall cost of recovery.*

1 Introduction

Transactions in a distributed environment access data located at different sites. Part of the correctness of a distributed transaction is to ensure its atomicity which requires that all the transaction's effects either persist at all the sites the transaction has visited or are obliterated from them. This is achieved by employing an *atomic commitment protocol* (ACP) that executes a commit or an abort operation across multiple sites as a single logical operation. The simplest and most used ACP is the *two-phase commit* protocol (2PC) [5, 7].

2PC consists of a *voting phase* during which the coordinator of a distributed transaction requests all the sites participating in the transaction's execution to *prepare to commit*, and of a *decision phase* during which the coordinator either decides to commit the transaction if *all* the participants are *prepared to commit* (voted `Yes`), or to abort if any participant has *decided to abort* (voted `No`). If a participant has voted `Yes`, it can neither commit nor abort the transaction until it receives the final decision from the coordinator. When a participant receives the final decision, it complies with the decision and sends back an acknowledgment. The coordinator completes the protocol and discards any information in its *protocol table* in main memory regarding the transaction when it receives acknowledgments from all the participants.

The resilience of 2PC to system and communication failures is achieved by recording the progress of the protocol in the logs of the coordinator and the participants. The coordinator is required to force-write a `decision` record prior to sending the final decision to the participants. Since a *force-write* ensures that all log records are written into a stable storage that sustains system failures, the final decision is not lost in the case of a coordinator failure. Similarly, each participant force-writes a `prepared` record before sending its vote and a `decision` record before acting on and acknowledging a final decision. When the coordinator completes the protocol, it writes an `end` record without forcing it into stable storage, indicating that the log records pertaining to the transaction can be garbage collected when necessary.

Future *distributed database systems* (DDBSs) are expected to execute on highly reliable computers that are connected via high speed networks with data transfer rates in the order of gigabits per second. In such *gigabit-networked* DDBSs, the propagation latency will be the dominant component of the overall communication cost while the migration of large amounts of data will not pose a problem [6, 1]. In other words, the size of messages in a database protocol is of less concern than the required number of *rounds* or *sequential phases* of message passing. Given this observation, and the need to ensure the atomicity of distributed transactions in gigabit-networked DDBS, we are prompted to ask the question: *Is it possible to improve the performance of 2PC by permitting large messages?* That is, can we reduce the number or the rounds of messages in 2PC by not placing any limitations on the size of a message?

In this paper, we present the *implicit yes-vote* (IYV) that improves on the other 2PC variants by exploiting the performance and reliability properties of gigabit computer networks. IYV effectively *eliminates* the voting phase of 2PC by combining the participants' votes with the execution of the transactions' operations, hence reducing the number of sequential coordination messages during normal processing. In case of a communication or a participant failure, IYV supports *forward* recovery by enabling a partially executed transaction to resume its execution when the failure is fixed. This is achieved by logging the *read* locks and the *redo* records that are generated during the execution of operations at both the coordinator and the participants. The underlying assumption in IYV is that all sites employ *strict two-phase locking* protocol (2PL) [4].

In Section 2, IYV is introduced and its behavior in the presence of failures is discussed in detail. We also apply to IYV the *presume abort* (PrA) optimization [9, 10] which has been adopted by the OSI-TP and X/Open standards. In Section 3, we compare IYV with four other well known ACPs and in Section 4, we evaluate them in terms of the number of sequential messages and forced writes that are required to reach a decision and to release the locks held by a transaction.

*Supported in part by N.S.F. under grant IRI-9210588.

2 Implicit Yes-Vote

The essence of 2PC that ensures the atomicity of a distributed transaction is that it prevents a transaction from unilaterally committing or aborting at a site while in a *prepare to commit* state. A participant may decide to abort a transaction either for correctness reasons such as ensuring serializability, or for performance reasons such as minimizing transaction blocking. Regarding the latter, we can expect that a participant does not abort a transaction because it has not received an operation from the transaction for some time. It is the responsibility of the coordinator to decide whether or not it is necessary to abort a long-executing transaction.

Now, consider a distributed system in which all the sites employ strict 2PL (which also avoids cascading aborts). Participants *never* abort transactions to ensure atomicity and *only* abort transactions in active state, i.e., transactions having outstanding operation acknowledgements, to resolve deadlocks or conserve resources. In such a system, it is not possible for a participant to unilaterally abort a transaction after it has acknowledged the execution of an operation due to a serializability or atomicity violation, or a deadlock at that site. If the transaction was involved in a local deadlock, its operation would have been blocked or rejected rather than being acknowledged.

Based on the above assumptions, we can use the acknowledgment of the execution of an operation to *implicitly* mean that the transaction is in its prepared state at the participant. In this way, we can eliminate the need for the voting phase of 2PC. That is, a transaction enters its prepared to commit state at a site after the execution of each of its operations. When in prepared to commit state, a transaction can become active again when a new operation request is submitted and can be committed or aborted when a decision message is received by the participant (Figure 1).

A remaining question is how to ensure that a transaction can be correctly recovered after a failure without having to force-write the log records that are generated during the processing of each operation prior to acknowledging its completion as in the case of *early prepare* [11, 12]. Notice that a force-write involves a disk access that suspends the protocol until the disk access is completed. Also, unlike the *unsolicited vote optimization* [13], the participants in IYV have no knowledge about when their parts in a transaction execution have been completed. Thus, participants cannot tell when to force their logs into the stable storage. In the case of IYV, we achieve this through a low-cost *partial replication* of each participant's log at the coordinators' sites given that (1) migrating large amounts of data from a participant to the coordinator (and vice versa) in gigabit networks will not pose a problem; and (2) the cost of forcing a log is practically independent of its size, i.e., the number of records to be written, and is due to queueing delays.

We assume that each site employs page-level logging and uses a traditional *undo-redo* recovery techniques [2] in which the *undo phase* precedes the *redo phase*. As it will become apparent in Section 2.2, recovery schemes such as ARIES [8] which are in general highly efficient, may not offer the same efficiency in the context of dual logs adopted by IYV, because their redo phase precedes the undo phase.

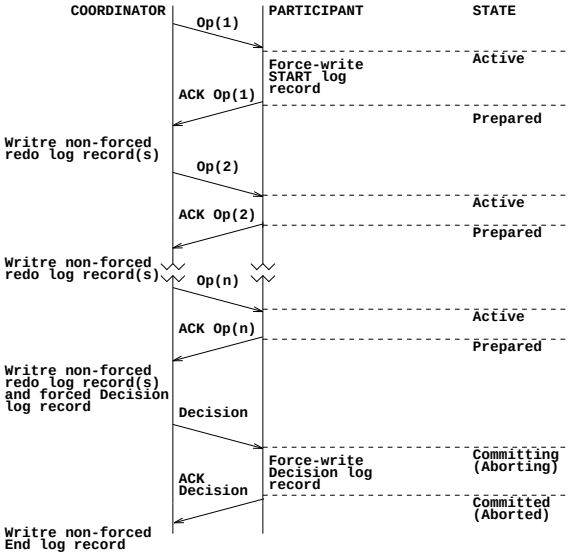

Figure 1: The IYV protocol.

In IYV, dual logging works as follows. Each participant includes the *redo* records that have been generated during the execution of an operation with their corresponding *log sequence numbers* (LSNs) in the acknowledgment (ACK) message of the operation. When the coordinator receives an ACK message, it writes a non-forced log record containing the (participant's) log records in the message. In this way, the coordinator's log contains an image of the redo part of each participant's log which can be used to reconstruct the redo part of a participant's log in case it is corrupted due to a system failure.

To facilitate *forward recovery* that allows transactions to resume their execution after a participant failure, IYV must be able to reconstruct the complete state of the recovered transactions including the lock table. For this reason, the participants are required to include in the ACK messages, along with the redo records, all the *read* locks which have been acquired during the execution of operations. In this way, a coordinator also maintains a *partial* image of the lock table of each participant. After a failure, transactions' write locks are re-acquired during the redo phase whereas read locks are extracted from the partial image of the lock table at the coordinator.

2.1 The IYV Protocol

When a participant receives the first operation of a transaction, it force-writes a `start` record which includes the identity of the transaction's coordinator in its log and then executes the operation. The `start` record is basically used by a participant to limit the number of coordinators that are needed to be contacted during its recovery after a failure. Any subsequent operations received by the participant are processed and logged (without forcing the log into stable storage) prior to acknowledging them. If a participant fails to process an operation, it aborts the transaction and replies with *negative* acknowledgement (NACK).

If the coordinator receives either an abort request from the transaction or a NACK from a participant, the coordinator decides to abort the transaction. Once the

coordinator decides to abort the transaction, it force-writes an `abort` record and then, sends abort messages to all the participants. On the other hand, when the coordinator receives a commit request from the transaction, it waits for the acknowledgment of the transaction's pending operations before deciding to commit the transaction. On a commit decision, the coordinator force-writes a `commit` record prior to sending a commit message to all the participants. In either case, the decision record includes the identities of all the participants.

When a participant receives a commit (abort) message regarding a transaction, it writes a forced `commit` (`abort`) record and commits (aborts) the transaction releasing all the transaction's resources. In the case of an abort decision, the participant undoes all the effects of the transaction (using its own log). A participant finally acknowledges a decision after the corresponding decision record is placed into the stable log.

Finally, when the coordinator receives the acknowledgments from all the participants, it writes a non-forced `end` log record and discards all information pertaining to the transaction from its protocol table knowing that no participant will inquire about the transaction's outcome in the future.

2.2 Recovery in IYV

IYV is resilient to both communication and site failures (see Figure 2). As it is the case in the 2PC and all its variants, site and communication failures are detected by timeouts.

Communication Failures

Although we assume communication failures to be rare in high speed networks, there are three places in IYV where a communication failure might occur while a site is waiting for a message. First is when a participant has no pending acknowledgments and is waiting for a new operation or a final decision. In this case, the participant is blocked until the communication with the coordinator is re-established. Then, the participant inquires the coordinator about the transaction's status. The coordinator replies with either a final decision or a *still active* message. In the former case, the participant enforces the final decision and then acknowledges it, while in the latter case, the participant waits for further operations.

The second place is when the coordinator of a transaction is waiting for an operation acknowledgment from a participant. A participant may abort the transaction, if a communication failure occurs while a participant has a pending acknowledgment. Similarly, the coordinator may abort the transaction and submits a final abort decision to the rest of the participants. Notice that the coordinator of a transaction may commit the transaction despite communication failures with some participats as long as these participants have no pending acknowledgments.

The third place is when the coordinator of a transaction is awaiting the acknowledgments of its final decision. Since the coordinator needs the acknowledgments in order to discard the information pertaining to the transaction from its protocol table and its log, it resubmits its final decision once these communication failures are fixed. When a participant receives the final decision after a failure, it either just acknowledges the decision if it has already received and enforced the decision prior to the failure[1], or enforces the decision and then sends back an acknowledgement message.

Coordinator Failure

Upon a coordinator restart, after a failure, the coordinator re-builds its protocol table by scanning its log. The coordinator needs to consider only those transactions that have decision records without a corresponding `end` record. For each of these transactions, the coordinator creates an entry in its protocol table that includes the identities of the participants as recorded in the transaction's decision record. Then, it restarts the decision phase of IYV for each of these transactions by re-submitting its decision to all the participants and resumes normal operation.

If a participant has already received and enforced the final decision prior to the failure, as in the case of a communication failure, the participant simply responds with an acknowledgment. If the participant has not received the decision, it must have been waiting for the decision and once it receives the decision, it force writes a decision record and then sends an ACK message.

For those transactions without final decision records (i.e., those transactions that were active prior to the failure), the coordinator can safely forget about them and consider them as aborted transactions. If a participant in the execution of one of these transactions has a pending acknowledgment when it timeouts, it will abort the transaction. On the other hand, if it is left blocked, when the coordinator recovers, the participant will inquire about the status of the transaction. For those transactions that are associated with decision records as well as `end` records, the coordinator can safely discard all information about these transactions knowing that no participant will inquire about their outcome in the future.

Participant Failure

After a failure and during the *analysis phase* of the restart procedure, the participant determines the set of transactions each associated with a `start` record and without a corresponding decision record, and inquires their coordinators about their status. Since the entire log might not be written into a stable storage until after a decision record is forced written, the log may not contain all the redo records of the operations committed at the site. Thus, an inquiry message for a transaction contains the largest LSN of the record in the stable log pertaining to the transaction. This log record corresponds to the transaction's last operation executed by the participant and having survived the failure. In the mean time, the participant recovers those committed and aborted transactions that have decision records pertaining to them already stored in its stable log. That is, the *undo phase* is performed and the *redo phase* is initiated while waiting for the reply messages to arrive from the coordinators.

For each active transaction finally committed, the coordinator sends back a commit message augmented with a list of all the transaction's redo records that are stored in its log and have LSNs greater than the one received

[1] A participant without any memory regarding the transaction is assumed to have already enforced the decision and discarded all information pertaining to the transaction.

Coordinator's Algorithm

In case of a communication failure:

- Abort each active transaction that has a pending acknowledgment at an inaccessible site or no participant site can be found to process one of the transaction's operations.

In case of a site failure:

1. For each transaction that has a decision record in the stable log without a corresponding **end** record, include the transaction in the protocol table and restart the decision phase.
2. Abort all active transactions (i.e., transactions without decision log records).
3. Do not consider transactions with **end** records already in the stable log.
4. Resume normal processing.

Participant's Algorithm

In case of a communication failure:

- Abort all active transactions with pending acknowledgments.
- Wait until the failure is fixed and then inquire about the status of all active transactions without pending acknowledgments.
  - Either a *decision* or a *still active* message will be received for each of these transactions.

In case of a site failure:

1. Analysis phase: identify committed, aborted and active transactions, recording for each active transaction its coordinator (from its **start** record) and largest LSN.
2. For each active transaction send an inquiry message containing its largest LSN to its coordinator.
3. Undo the effects of aborted and active transactions.
4. Once the reply messages arrive, repair the log, update the list of committed and still-active transactions and re-build the lock table.
5. Complete the redo phase.
   - Redo committed transactions and release their locks.
   - Redo still-active transactions.
6. Resume normal processing.

Figure 2: Recovery in IYV.

from the participant. On the other hand, a coordinator has to send only an abort message in response to an inquiry about an aborted transaction[2].

For each active transaction that is still in progress in all other sites (i.e., no decision was made), the coordinator replies with a *still-active* message containing, as in the case of a commit decision, a list of the redo records associated with LSNs greater than the one received from the participant. The message also contains all the read locks that were held by the transaction at the participant's site prior to its failure.

Once the participant receives all the reply messages, it repairs its log and completes the redo phase. The participant also re-builds its lock table by re-acquiring the update locks during the redo phase in conjunction with the read locks received from the coordinators. Once the redo phase is complete, the participant acknowledges all decision messages and resumes its normal processing. In this way, a long-executing transaction is not aborted as a result of a participant failure as it would have been the case with all the other 2PC variants.

The case that both the coordinator and a participant site fail at the same time is handled in the same way as above. However, it is necessary that the coordinator recovers before the participant.

2.3 Implicit Yes-Vote Presume Abort

In 2PC, there is a hidden presumption that allows the coordinator not to force write any log records prior to the decision phase. During the recovery of a coordinator, not finding a decision record pertaining to a transaction is interpreted as an abort decision. This presumption is made more explicit in the *presume abort protocol* (PrA) [9, 10]. *Implicit yes-vote presume abort* (IYV-PrA), and in a manner similar to PrA, adopts the abort presumption. In IYV-PrA, the coordinator of a transaction needs only to force write a `commit` record. Any missing information about a transaction is presumed to mean that the transaction has been aborted. The abort presumption is made explicit by not writing an `abort` record at all and by discarding all the information about an aborted transaction from the protocol table.

In IYV-PrA, participants also do not have to force-write an abort decision, nor do they have to acknowledge an abort message. Thus, in addition to log writes, IYV-PrA reduces the number of coordination messages for aborted transactions. If a participant fails before the abort decision record is in stable storage, upon its recovery, it will inquire the transaction's coordinator about the transaction's status. Since the coordinator would not have any information about the transaction, it will direct the participant to abort the transaction by presumption. The cost of a commit decision, however, remains the same as in the IYV.

3 Comparison of IYV with other ACPs

As mentioned above, in contrast to *unsolicited vote* (UV) optimization, IYV does not assume that each site knows when it has executed the last operation on behalf of a transaction in order to avoid writing the log records associated with an operation prior to acknowledging it.

[2]Note that the effects of such a transaction have already been undone by the participant during the undo phase.

	2PC	PrC	PrA	EP	CL	IYV	IYV-PrA	IYV	IYV-PrA
						With Start log records		Without Start log records	
Log force delays	2	3	2	3	1	2	2	1	1
Total log force writes	2n+1	n+2	2n+1	n+2	1	2n+1	2n+1	n+1	n+1
Message delays (Commit)	2	2	2	0	0	0	0	0	0
Message delays (Locks)	3	3	3	1	1	1	1	1	1
Total messages	4n	3n	4n	n	n	2n	2n	2n	2n
Total messages with piggybacking	3n	3n	3n	n	n	n	n	n	n

Table 1: The cost of the protocols to *commit* a transaction assuming the best case scenario.

	2PC	PrC	PrA	EP	CL	IYV	IYV-PrA	IYV	IYV-PrA
						With Start log records		Without Start log records	
Log force delays	2	2	1	2	0	2	1	1	0
Total log force writes	2n+1	2n+1	n	2n+1	0	2n+1	n	n+1	0
Message delays (Abort)	2	2	2	0	0	0	0	0	0
Message delays (Locks)	3	3	3	1	1	1	1	1	1
Total messages	4n	4n	3n	2n	2n	2n	n	2n	n
Total messages with piggybacking	3n	3n	3n	n	2n	n	n	n	n

Table 2: The cost of the protocols to *abort* a transaction assuming the best case scenario.

Thus, IYV is more general compared to UV. In the special cases in which UV is applicable, IYV and UV would exhibit the similar behavior during normal processing.

The *early prepare* protocol (EP) [11, 12] combines UV with *presume commit* (PrC) [9, 10]. PrC requires the identities of the participants to be explicitly recorded by the coordinator in a forced `initiation` log record, to ensure that an aborted transaction is not presumed as committed after a failure. Thus, in EP, the number of forced log writes pertaining to a transaction is equal to the number of the participants that executed the transaction (without a form of predeclaration). This is because the list of participants has to be updated and a new `initiation` record has to be forced written each time a new participant executes an operation of the transaction. In contrast, in IYV a coordinator does not have to force write any `initiation` log records for correctness purposes whereas the forced `start` record by the participants aims to reduce the cost of recovery. In the next section we evaluate the IYV variant which eliminates the `start` record at the expense of slower recovery.

Another non-2PC atomic protocol behaving similar to EP is the *coordinator log* protocol (CL) which assumes that transactions are short and most probably going to commit [11, 12]. CL eliminates the need for any (force-write) logging at the participants' sites by having the coordinators maintain the logs and using *distributed write-ahead logging* (DWAL) [3]. That is, the log of a participant is distributed among multiple coordinator sites. CL also eliminates the `initiation` record of EP.

Since a participant in CL might inquire a coordinator about the latest forced log write (i.e., to ensure the WAL protocol), this might become completely costly when compared with any of the 2PC variants. Also, rolling back aborted transactions has to be performed completely over the network. This means that when a participant aborts a transaction, it cannot release the resources held by the transaction until it communicates with the transaction's coordinator and receives the undo log records pertaining to the transaction.

Another problem with CL, as presented in [11, 12], is that the log records of transactions cannot be garbage collected by the coordinators and have to be remembered forever. In CL, garbage collection is given up for committed as well as aborted transactions even though abort decisions are acknowledged by the participants. (In this case, there is no actual benefit from the acknowledgment messages except that they contain the undo log records of aborted transactions.)

In case of failures, coordinators in IYV can make their own decisions regarding active transactions without communicating with any participant, whereas a recovering coordinator in CL has to communicate with all possible participants in the system. This is the cost that has to be paid in CL for eliminating the `initiation` record. A recovering participant in CL, on the other hand, has to wait until it receives all the log records from the coordinators and until all active transactions have been decided upon. In IYV, however, using the "still active" message, a participant can recover its state prior to its failure without having to wait until all active transactions have been decided upon. Further, aborted transactions in IYV are handled locally by a participant without any communication with the coordinators (the undo records do not have to be propagated to the coordinators).

4 Evaluation of IYV

In this section, we evaluate IYV along with the protocols considered above, namely, 2PC, PrC, PrA, EP and CL. Due to space limitations, we assume that the reader is familiar with these protocols and hence we do not elaborate on them. In our evaluation, we further consider IYV-PrA as well as IYV variants which do not require `start` records to be forced by the participants at the expense of having to communicate with all the coordinators in order to recover a failed participant.

In our evaluation, we use best and worst case scenarios as in [11, 12] to highlight the performance differences among the various ACPs protocols and we consider the number of coordination messages and forced log writes that are due to the protocols only (e.g., we do not con-

	2PC	PrC	PrA	EP	CL	IYV	IYV-PrA	IYV	IYV-PrA
						With Start log records		Without Start log records	
Log force delays	2	3	2	d+n+1	d+1	n+1	n+1	1	1
Total log force writes	2n+1	n+2	2n+1	d+n+1	d+1	2n+1	2n+1	n+1	n+1
DWAL Message delays	0	0	0	0	2d	0	0	0	0
Message delays (Commit)	2	2	2	0	2d	0	0	0	0
Message delays (Locks)	3	3	3	1	2d+1	1	1	1	1
Total messages	4n	3n	4n	n	2d+n	2n	2n	2n	2n
Total messages with piggybacking	3n	3n	3n	n	2d+n	n	n	n	n

Table 3: The cost of the protocols to *commit* a transaction assuming the worst case scenario.

	2PC	PrC	PrA	EP	CL	IYV	IYV-PrA	IYV	IYV-PrA
						With Start log records		Without Start log records	
Log force delays	2	2	1	d+n	d	n+1	n	1	0
Total log force writes	2n+1	2n+1	n	d+2n	d	2n+1	n	n+1	0
DWAL Message delays	0	0	0	0	4d	0	0	0	0
Message delays (Abort)	2	2	2	0	2d	0	0	0	0
Message delays (Locks)	3	3	3	1	4d+1	1	1	1	1
Total messages	4n	4n	3n	2n	4d+n	2n	n	2n	n
Total messages with piggybacking	3n	3n	3n	n	4d+n	n	n	n	n

Table 4: The cost of the protocols to *abort* a transaction assuming the worst case scenario.

sider the number of messages that are due to the operations and their acknowledgments).

Table 1 and Table 2, compare the number of messages and forced log writes that are needed to commit and abort a transaction, respectively, for the different protocols based on the best case scenario. We denote by n the number of participants that executed a transaction and by d the number of data items that have been accessed by the transaction. In this scenario, we assume that: The participants are known at the beginning of a transaction, each participant executes at most an operation on a single data item for each transaction (i.e., d=n) and the operations of a transaction execute in parallel on the participants. Also, participants have sufficient memory space that prevents them from having to force the log during the execution of a transaction.

The rows labeled "Log force delays" contain the sequence of forced log writes that are required up to the point of commit/abort decision is made. The rows labeled "Message delays (commit/abort)" contain the number of sequential messages up to the commit/abort point, and the rows labeled "Message delays (Locks)" contain the number of sequential messages that are involved in order to release all the locks held by a committing/aborting transaction. For example, in Table 1, the "Log force delays" for 2PC is 2 because there are two force log writes between the beginning of the protocol and the time a commit decision is made by a transaction's coordinator. Also, 2PC involves two sequential messages in order for a coordinator to make its final decision regarding a transaction (i.e., the first phase), and three sequential messages to release all the resources (i.e., locks) held by the transaction at the participants.

In the best case scenario, CL dominates all other 2PC variants as far as the logging activities and forced log writes are concerned for the commit case. It requires a single log force write and a single message to be sent to each participant. In the abort case, although IYV-PrA (with the `start` log records) and CL need the same number of sequential messages to abort and release locks, IYV-PrA trades off a forced write for a message in CL. Once the `start` log record is eliminated from IYV-PrA, it dominates the CL by n messages in the total message count.

Piggybacking the acknowledgments is an optimization that can be used to eliminate the final round of messages for the commit case in 2PC, PrA, IYV, and IYV-PrA, but not in the case of PrC, EP and CL because a commit final decision is never acknowledged in these protocols. Similarly, this optimization can be used in the abort case with the 2PC, PrC, EP, and IYV but not with PrA, CL and IYV-PrA. In PrA and IYV-PrA, an abort decision is never acknowledged while in CL, the acknowledgement is sent immediately because it contains the undo log records of the aborted transaction.

Tables 3 and 4 compare the different protocols under the worst case scenario. In this scenario, we assume that: (1) participants are not known at the beginning of a transaction, (2) each participant executes more than one operation on behalf of a transaction (i.e., d > n), (3) transactions execute serially (e.g., an operation is submitted by a transaction only when the previous operation has been executed and acknowledged), (4) each operation generates a single log record, and (5) participants have limited memory space. That is, each log record that is generated due to the execution of an operation has to be forced written into the stable log as a worst case scenario. (Note that in our evaluation, we do not include the number of forced log writes which are due to the operations and which are the same in all the protocols except for EP where the log records have to be forced written all the time.)

In the worst case scenario, CL requires two explicit messages to be exchanged between a participant and the coordinator of a transaction for each operation executed by the participant for the commit case (thus, the 2d in "DWAL Message delays"). For the abort case, four messages are needed to be exchanged between the participant and the coordinator of an aborted transaction. This is because undoing an operation using ARIES [8],

the recovery scheme of CL is another write operation that has to be logged. Since CL uses a DWAL logging protocol, undoing an operation requires two more explicit messages to be exchanged between the coordinator and the participant in the worst case scenario.

The two scenarios show that the cost associated with EP is highly dependent on the number of operations of the transactions while CL is also dependent on the size of main memory and the percentage of committed and aborted transactions. Thus, EP and CL are rather inefficient in DDBSs with long-living transactions where a transaction might execute a large number of operations at each site it accesses, a situation that is typically found in advanced distributed database applications. Together with EP and CL, IYV and IYV-PrA with `start` record involve the least number of sequential messages. Although their performance depends on the number of sequential forced `start` records, these are at most equal to the number of participants which is generally very small compared to the number of write operations. It is clear from the tables that IYV and its PrA optimization without `start` records promise the minimum cost among the other 2PC variants.

Even though IYV requires the redo records generated during the execution of a transaction's operation be logged at both its coordinator as well as the participants, such a duplicate logging should incur negligible overhead because the log records are written in a non-forced manner and without involving any extra coordination messages. The only overhead is that IYV requires more buffer space for the log of the coordinator so that logging do not cause frequent flushing to the log buffer. As mentioned earlier, we believe that, in general, the overhead associated with the duplication of logs and the extra information contained in the commit and still-active messages is well offset by the reduction in the number of sequential coordination messages and the gain of being able to support forward recovery of interrupted, possibly long-lived, transactions due to participant and communication failures.

5 Conclusion

In this paper, we proposed *implicit yes-vote* (IYV), a 2PC variant that is targeted toward those future highly reliable distributed database sites interconnected via gigabit networks. In such environments, the propagation latency is more of an issue than the size of messages. IYV exploits these new domain characteristics to minimize the cost of transaction commitment at the cost of slower recovery.

However, after a participant or a communication failure, IYV allows partially executed transactions that are still active on other participants to resume their execution, a situation that is not possible in any other 2PC variant. IYV supports forward recovery through a low-cost partial replication of the log and lock table of each participant at the coordinator sites. Forward recovery potentially reduces the overall cost of recovery that has been traded off for efficiency during normal processing. Similar to all other 2PC variants, IYV is a blocking protocol in the face of communication and site failures.

Currently, we are investigating methods of reducing the cost of recovery in the case of IYV without `start` record and we are extending IYV to be used in the context of multilevel distributed transactions.

Acknowledgment

We would like to thank George Samaras, Sujata Banerjee and Ed Cymbalak for their helpful feedback on this work.

References

[1] S. Banerjee and P. K. Chrysanthis. "Data Sharing and Recovery in Gigabit-Networked Databases," Proc. of the 4th International Conference on Computer Communications and Networks, 1995.

[2] P. A. Bernstein, V. Hadzilacos and N. Goodman, Concurrency Control and Recovery in Database Systems, Adison-Wesley, Reading, MA, 1987.

[3] D. DeWitt, et al., "The Gamma Database Machine Project," IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, pp. 44–69, 1990.

[4] K. P. Eswaran, et al., "The Notion of Consistency and Predicate Locks in a Database System," Communications of the ACM, Vol. 19, No.11, pp.624–633, 1976.

[5] J. Gray, "Notes on Data Base Operating Systems," In Operating Systems: An Advanced Course, LNCS, Vol. 60, pp. 393–481, Springer-Verlag, 1978.

[6] L. Kleinrock, "The Latency/Bandwidth Tradeoff in Gigabit Networks," IEEE Communications Magazine, Vol. 30, No. 4, pp. 36–40, 1992.

[7] B. W. Lampson, "Atomic Transactions," In Distributed Systems: Architecture and Implementation - An Advanced Course, LNCS, Vol. 105, pp. 246-265, Springer-Verlag, 1981.

[8] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," ACM Transactions on Database Systems, Vol. 17, No. 1, pp. 94–162, 1992.

[9] C. Mohan, B. Lindsay and R. Obermarck, "Transaction Management in the R^* Distributed Database Management System," ACM Transactions on Database Systems, Vol. 11, No. 4, pp. 378–596, 1986.

[10] G. Samaras, K. Britton, A. Citron and C. Mohan, "Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment," Proc. of the 9th International Conference on Data Engineering, pp. 520–529, 1993.

[11] J. Stamos and F. Cristian, "A Low-Cost Atomic Commit Protocol," Proc. of the 9th Symposium on Reliable Distributed Systems, pp. 66–75, 1990.

[12] J. Stamos, and F. Cristian, "Coordinator Log Transaction Execution Protocol," Distributed and Parallel Databases, Vol. 1, pp. 383–408, 1993.

[13] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," IEEE Transactions on Software Engineering, Vol. 5, No. 3, pp. 188–194, 1979.