The Implicit–Yes Vote Commit Protocol with Delegation of Commitment*

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

The *implicit yes–vote* commit protocol (IYV), proposed for future gigabit–networked distributed databases, reduces the time required to commit a distributed transaction at the expense of independent recovery of failed participant sites. In this paper, we propose a novel coordination scheme for IYV that reduces the window of vulnerability to blocking and minimizes the time required for the sites to become operational after a failure. The new scheme combines the delegation of commitment technique with a timestamp synchronization mechanism. Although this new scheme incurs extra coordination messages and log writes, it enhances the performance of IYV during recovery in the presence of less reliable sites while still maintaining the cost of commit processing during normal processing below that of two-phase commit and its other well known variants.

Keywords: *Two–phase commit, atomic commit protocols, distributed transactions, forward recovery.*

1 Introduction

In a *distributed database system*, the sites participating in a transaction's execution need to reach an agreement about the final outcome of the transaction to ensure that the transaction is either committed or aborted at all participating sites. This "all or nothing" property of transactions is achieved by employing an *atomic commit protocol* (ACP) such as the *two–phase commit* protocol (2PC) [4, 6].

2PC consumes a substantial amount of a transaction's execution time due to the cost associated with the coordination messages and forced log writes. Hence, a number of 2PC variants and optimizations have been proposed to reduce the cost of 2PC in different environments (e.g., [4, 7, 10, 9, 1]). The *implicit yes–vote* protocol (IYV) that was proposed by us in the context of gigabit–networked distributed databases, eliminates the voting phase from 2PC [1]. However, IYV increases the size of the window of vulnerability to blocking in case of a communication or a coordinator's site failure and trade in independent recovery after a site failure for fast commit of transactions during normal processing. While the former seems to be a reasonable trade off considering, for example, the reliability characteristics of gigabit networks, it is imperative to increase the degree of independent recovery of IYV if it is to be used in environments where some sites or communication links are relatively less reliable.

Delegating the commitment of a transaction to a site other than the transaction's coordinator is a technique that has been used to enhance the performance of 2PC (e.g., the *last agent* optimization [9]) as well as its reliability (e.g., *open commit protocols* [8]). In this paper, we propose a new coordination scheme for IYV that reduces the vulnerability of the protocol to blocking and alleviates the problem due to the lack of independent recovery by combining the delegation of commitment technique with a timestamp mechanism.

In the next section, we briefly overview IYV and in Section 3, we present a new protocol called *IYV with a commit coordinator* that employ our new coordination scheme. The new protocol enhances the reliability of IYV while still maintaining its cost below that of 2PC and its most commonly known variants as it is shown in Section 4. In Section 5, we conclude this paper.

2 The Implicit Yes-Vote Protocol (IYV)

The IYV commit protocol has been proposed in the context of future *gigabit-networked* distributed databases [1] where *propagation latency* is the dominant component of the overall communication cost while the migration of large amounts of data is not a problem [5]. IYV is based on the assumptions that each site employs (1) a *strict two-phase locking* protocol (2PL) for concurrency control and (2) *physical page–level write–ahead logging* (WAL) with the undo phase *preceding* the redo phase for recovery [2]. Based on the first assumption, it is not possible for a participant in a transaction's execution to abort the transaction due to a deadlock or serializability violation once all the operations received by the participant have been executed and acknowledged. Hence, instead of initiating commit processing at the end of transactions, which is the case in 2PC, commit processing is overlapped with the execution of the transactions' operations.

Specifically, when the coordinator of a transaction (i.e., the site where the transaction has been initiated) receives an acknowledgment message (ACK) from a participant pertaining to a transaction's operation, the ACK is *implicitly* interpreted to mean that the transaction is in a prepared to commit state at the participant as shown in Figure 1. When the participant receives a new operation for execution, the transaction becomes active again and can be aborted, for example, if it causes a deadlock. If the transaction is aborted, the participant responds with a *negative acknowledgment* message (NACK). Only when all the operations

*Supported in part by a Saudi Arabian graduate student scholarship and N.S.F under grant IRI-95020091.

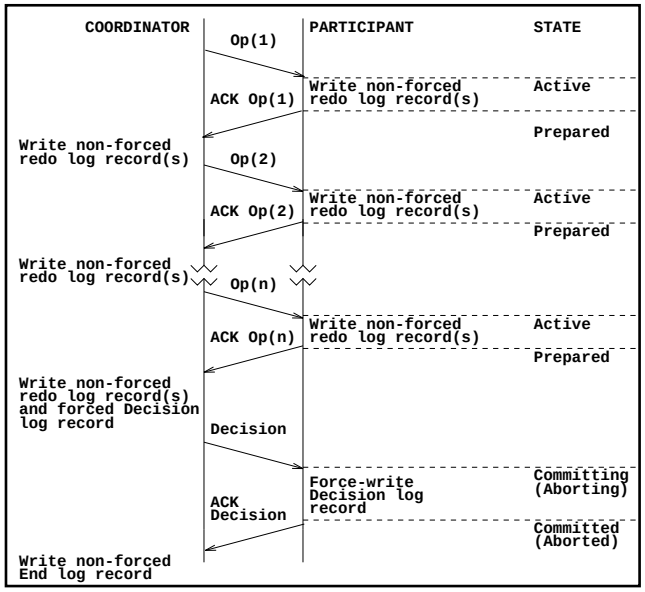

Figure 1: The IYV protocol.

pertaining to the transaction are executed and acknowledged by their perspective participants, the coordinator commits the transaction. Otherwise, the coordinator aborts the transaction. In either case, the coordinator propagates its decision to the appropriate participants and waits for their acknowledgments as it is the case in 2PC. Thus, the *voting* phase of 2PC, which polls the votes of the participants, is eliminated by overlapping it with the execution of operations in IYV while the *decision* phase remains the same as in 2PC.

To ensure correct recovery after a failure, each participant in IYV is required to include the *redo* log records generated during the execution of an operation with their corresponding *log sequence numbers* (LSNs) in the operation's ACK. Each participant also includes the *read* locks acquired during the execution of an operation in the ACK in order to support *forward recovery*. In this way, each coordinator has a partial image of the state of each of its participants in the system. After a crash, a participant can reconstruct the state of its database, which includes its log and lock table as it was just prior to the failure, by requesting copies of the partial images of its state stored at the coordinators. Hence, a participant in IYV is not only able to ensure the consistency of its database by applying the effects of committed transactions and rolling-back the effects of aborted transactions, but it is also able to allow transactions that are still active in the system to continue their execution without having to abort them. By maintaining a local log and employing WAL, each participant is able to undo the effects of aborted transactions using only its own log.

3 IYV with Delegation of Commitment

By separating the execution of operations from the commit processing of transactions, a participant in 2PC is able to save on its local stable storage all the log records pertaining to a prepared to commit transaction using a single force log write. Thus, after a system crash, a participant has all the necessary information in its log to recover independently and resume normal processing while waiting to resolve the status of in-doubt transactions (i.e., transactions with prepare log records but without associated decision records). In IYV, forcing the log every time a transaction enters the prepared to commit state would have been prohibitively expensive. Instead, the redo part of the log of a participant is replicated at the coordinators' sites. Therefore, a recovering participant in IYV needs to communicate with all the coordinators in order to determine which of the active transactions in its site have been committed and which are still in progress as well as their accessed data items. Because of this, a participant cannot independently recover and it has to block any access to its entire database until it receives replies from all the coordinators. Thus, in order to deal with unreliable coordinators, it is imperative that all participants become operational in a bounded amount of time, in a similar manner as in 2PC. Towards this end, IYV is modified to utilize two sites as coordinators and to involve delegation of commitment.

In this new variant of IYV, let us assume an unreliable coordinator being responsible for the execution of transactions initiated at its site. When a transaction finishes its execution, the coordinator at the site where the transaction has been initiated, termed the *execution coordinator* (EC), prepares itself to commit the transaction and delegates the final commit decision to a more reliable site that fails rarely and if it fails it recovers within minutes, termed the *commit coordinator* (CC). The delegation message includes the identities of the participants as well as all the redo log records generated during the execution of the transaction with their associated LSNs. In this way, a recovering participant can inquire both coordinators and will be able to finish its recovery process as soon as it receives a reply from either of the two coordinators. Since the participant will receive a response from at least the reliable CC, the participant will be able to recover in a bounded amount of time, allowing new transactions to execute at its site. Thus, reducing the overall cost of recovery in IYV.

Notice that IYV with delegation of commitment differs from the open commit protocols in which the execution of 2PC at any given point is delegated to another reliable coordinator. In this respect, IYV with delegation of commitment is similar to the last agent optimization where *only* the decision is delegated to another site. However, the last agent decides whether to commit or abort a transaction based on its own local state, whereas in our protocol, a CC aborts a transaction only if detects that a participating site has failed. Although IYV with delegation of commitment employs two coordinators, it is different from disaster recovery protocols where a back-up site records the effects of committed transactions and takes over only when it detects the failure of the primary site (e.g., [3]).

3.1 IYV with a Commit Coordinator Protocol (IYV–WCC)

As mentioned above, in this version of IYV, each unreliable coordinator is paired with a more reliable commit coordinator that is responsible for the commit processing of the transactions initiated at the unreliable site. All such pairings are known to both EC and CC coordinators and all participants in the system.

As in IYV, when a participant receives an operation pertaining to a transaction from an EC, it includes the redo records generated and the read locks acquired dur-

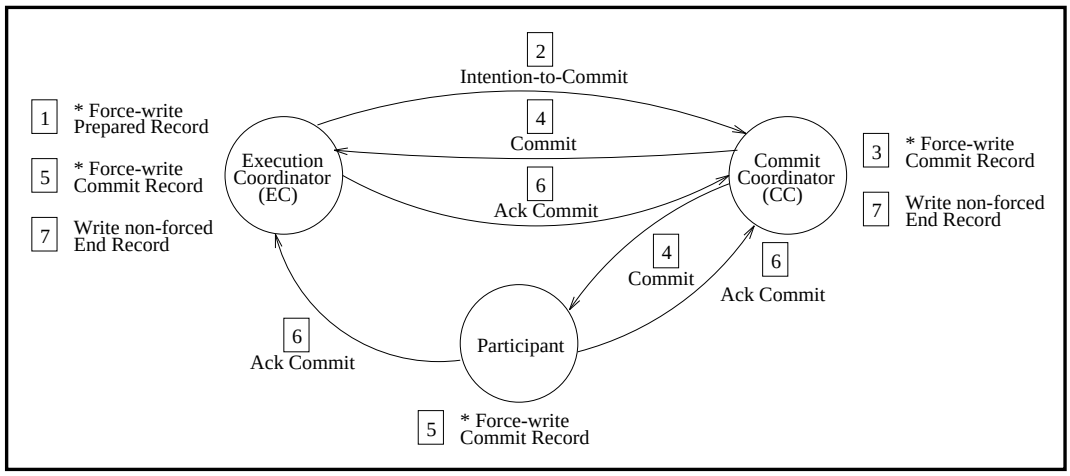

Figure 2: The coordination messages and forced log writes in IYV–WCC.

ing the execution of the operation in the ACK. The participant also generates a *timestamp* for the transaction after successfully executing the transaction's first *update* operation and includes the timestamp in the ACK as well. As it will become clear below, timestamping is the key for the correctness of this IYV variant because it ensures that a CC will always make consistent decisions about the outcome of transactions delegated to it by any EC and despite failures. If a participant fails to execute an operation, it aborts the transaction and sends a NACK. If an EC receives a NACK in response to an operation request from a participant or an `abort` primitive from the transaction, the EC aborts the transaction and sends an abort message to each participant (except the one which has sent a NACK, if any) and forgets about the transaction.

Now, let us examine the commitment of a transaction in the presence of delegation of commitment. During the discussion, we will refer to Figure 2 when numbering the actions taken in each step of the commit process. When a transaction finishes its execution at all participating sites successfully and submits its final commit primitive to its EC, the EC prepares itself to commit the transaction by force writing a `prepare` log record which includes the identities of all participants as well as the timestamps at each update participant. The `prepare` log record also causes all redo log records received from the participants to be forced into the stable log (action 1). Then, the EC delegates the commit responsibilities to its associated CC. The delegation action is achieved by sending an `intention-to-commit` message (action 2) that includes the identities of the participants, all the redo log records generated during the execution of the transaction with their corresponding LSNs, and the timestamps at the update participants. Thus, the CC is involved in a transaction's commitment process only when the transaction has finished its execution successfully at all participating sites and invoked the commit operation.

A CC keeps track of the time of the most recent crash of each participant by maintaining a *failure timestamp list* (FTSL). Thus, given a transaction, a CC can determine whether the transaction was active at the time of the crash or not by comparing the crash's timestamp with the timestamp of the transaction. So, when a CC receives an `intention-to-commit` message pertaining to a transaction, the CC checks its FTSL and makes a commit decision only if the timestamp of the transaction at each update participant is *greater than* the failure timestamp of the participant. Otherwise, the CC makes an abort decision.

On a commit decision, the CC force writes a `commit` log record (action 3) and sends a commit message to each participant (including the EC) (action 4). When a participant receives such a message, it force writes a `commit` log record (action 5) and sends a *commit acknowledgment* to both of the CC and EC (action 6). When the CC and EC receive the required acknowledgments, they write non-forced `end` log records (action 7) and forget about the transaction. The commit acknowledgment sent to the EC has a dual role. First, it tells the EC that the transaction has committed which is necessary in the case that the CC has failed after making the decision but before sending it to the EC. Second, it allows the EC to take over after the CC has failed and direct any participant that inquires about the outcome of the transaction to commit the transaction.

On an abort decision, on the other hand, the CC sends an abort message to the EC and to each participant where the transaction is still active (i.e., has its failure timestamp less than the transaction's timestamp) and forgets about the transaction without writing any log records. There is no need to send abort messages to participants that have failure timestamps greater than the transaction's timestamp because these participants have already aborted and undone the effects of the transaction during their recovery. When a participant receives an abort message from a CC pertaining to a transaction, it aborts the transaction and releases all the resources held by the transaction, without writing any log records or acknowledging the decision.

3.2 Recovery in IYV–WCC

We now discuss the recovery aspects of IYV–WCC. IYV–WCC is resilient to both communication and site failures that are detected by timeouts, as it is the case in all other ACPs.

3.2.1 Communication Failures

In IYV–WCC, there are five situations where a site is waiting for a message. The first situation is when the EC has forced a prepared log record for a transaction and has sent an `intention-to-commit` message to the CC. In this situation, the EC is blocked from being able to ACK the commitment of the transaction until it receives a final decision from the CC or a commit acknowledgment from one of the participants. While in the first situation, the EC inquires the CC once it re-establishes communication with the CC if it has not heard from any of the participants. The CC replies with an abort message if it has no recollection about the outcome of the transaction. Otherwise, the CC responds with a commit message that has to be acknowledged by the EC.

The second situation is when the EC is waiting for the acknowledgments from the participants pertaining to a committed transaction. While in this situation, the EC re-submits a commit message to each participant that has not acknowledged the commitment of the transaction. When a participant receives such a message, it either force writes a commit log record and then acknowledges the message if it did not receive the final decision from the CC, or it simply responds with an acknowledgment message if it has already received the final decision. Once the required acknowledgments arrive, the EC completes the protocol by writing a non-forced `end` log record and then forgetting about the transaction.

The third situation is similar to the second one but with respect to the CC, that is when the CC has already made its final commit decision and some participants have not acknowledged the decision.

The fourth situation is when a participant has timed out and it does not have any pending acknowledgments (i.e., all operations have been acknowledged). In this case, the participant inquires both the EC and the CC about the status of the transaction. While in this situation, a participant can receive one out of twelve different combinations of responses (see Table 1). A *No–response* in the table indicates that a participant did not receive a response from a coordinator during a specified amount of time while a *No–info* indicates that the responding coordinator has no recollection about the transaction at the time it has received the inquiry message from the participant. A *Still–active* response from an EC indicates that the transaction is still active at other sites and no final decision has been made regarding its final outcome. The rest of responses in the table are self explanatory.

In Table 1, response combinations numbered 1, 3, 4, 6, 8 and 9 indicate to the participant that it cannot do anything regarding the transaction except to wait until it receives further instructions form either the EC or CC. Response combinations 2 and 7 indicate to the participant that the transaction has been aborted because it is not possible for an EC to forget about a transaction without the acknowledgment of the inquiring participant. The rest of the response combinations indicate to the participant the transaction has been committed. For example, response combination 10 indicates that the transaction has been committed even though no response has been received from its EC, thereby, a participant is not blocked as it would have been the case in the basic IYV protocol.

The fifth situation is when a participant times out and it has a pending acknowledgment. That is, the participant realizes that a communication failure has occurred with the EC before acknowledging an operation pertaining to a transaction. In this case, the participant aborts the transaction. Similarly, an EC aborts a transaction if it times out without receiving an operation acknowledgment from a participant.

No.	EC Response	CC Response	Participant Conclusion
1	No–response	No–response	Wait for further instructions
2	No–info	No–response	Abort
3	Prepared	No–response	Wait for further instructions
4	Still–active	No–response	Wait for further instructions
5	Commit	No–response	Commit
6	No–response	No–info	Wait for further instructions
7	No–info	No–info	Abort
8	Prepared	No–info	Wait for further instructions
9	Still–active	No–info	Wait for further instructions
10	No–response	Commit	Commit
11	Prepared	Commit	Commit
12	Commit	Commit	Commit

Table 1: Responses to a communication failure.

3.2.2 Site Failures

Participant Failure

During its recovery process, a participant sends *recovering* messages to all (EC and CC) coordinators in the system and creates a *non-responding coordinators list* (NRCL). A recovering message contains the largest LSN associated with the latest record written into the participant's stable log as well as the value of the participant's time clock. While waiting for the reply messages to arrive, the participant starts the undo phase by undoing the effects of each aborted as well as each partially executed transaction, i.e., each transaction without a commit record in its own log. Once the undo phase is completed, the participant starts the redo phase by redoing all transactions that have been committed prior to the failure according to its log. Then, the participant blocks awaiting the reply messages to arrive from the coordinators.

When an EC receives the recovering message, it responds with a message that contains all the redo log records with LSNs greater than the one received in the recovering message for each prepared to commit, committed and still active transaction. For each still active transaction, the EC also includes all the read locks held by the transaction at the participant prior to the failure. The EC also indicates, in its reply message, which transactions are in their prepared to commit states (even though a prepared to commit transaction might not have redo log records with LSNs greater than the one received from the participant), which transactions have been committed and which transactions are still in their active state. Then, the EC waits for an acknowledgment message from the participant before it can submit any further operations for execution. If the recovering participant has not participated in any of the prepared to commit, committed or active transactions at the EC, the EC responds with a *nothing for you* message. This message indicates to the participant that it has recorded

No.	EC Response	CC Response	Participant Conclusion
1	No–response	No–response	Suspend the recovery process
2	Prepared	No–response	Redo and wait for further instructions
3	Commit	No–response	Redo and commit
4	Still–active	No–response	Redo and wait for further instructions
5	Nothing for you	No–response	Do nothing
6	Prepared	Nothing for you	Abort
7	Still active	Nothing for you	Redo and wait for further instructions
8	No–response	Nothing for you	Do nothing
9	Nothing for you	Nothing for you	Do nothing
10	No–response	Commit	Redo and commit
11	Prepared	Commit	Redo and commit
12	Commit	Commit	Redo and commit

Table 2: Responses to a site failure.

in its log and acknowledged the commitment of all transactions initiated at the EC prior to the failure.

Similarly, when a CC receives a recovering message from a recovering participant, it responds with a message that contains all the redo log records of committed transactions that have LSNs greater than the one received from the participant. The CC also indicates which transaction have been committed and that the participant did not acknowledge its commitment prior to the failure. This is also the case even if the transaction did not have any redo log records associated with LSNs greater than the one contained in the recovering message. If each committed transaction that the participant has participated in its execution has been acknowledged prior to the failure, the CC sends a *nothing for you* message. The CC also updates its FTSL to reflect the time at which the participant has started recovering and force writes the list into the stable storage prior to sending its reply message.

When the participant receives a reply message from a coordinator, the participant removes the coordinator from its NRCL. If the coordinator is an EC, the participant sends back an acknowledgment message indicating that the EC has responded in a timely manner before the redo phase has finished. Therefore, the EC updates the timestamp of each still active transaction so that the transaction will not subsequently get aborted by the CC because its timestamp is less than the participant failure timestamp. That is, since each CC will update its FTSL to reflect the failure of the participant, an active transaction will be aborted by the CC if its timestamp is not updated by its EC. Therefore, an EC updates the timestamp of each active transaction at its site once it receives an acknowledgment from the recovering participant. The recovering participant finishes its recovery process only when it receives a reply from either the EC or the CC for each pair of EC–CC coordinators. This is the minimum number of replies that allow a recovering participant to finish its recovery process. Otherwise, the participant suspends its recovery process. As in the case of communication failures, there are twelve possible combinations of responses to a recovering participant. Table 2 summarizes these different response combinations, on a per transaction basis.

Once the minimum reply messages arrive, the participant continues its redo phase and re-builds its lock table. Once the redo phase is finished, the participant resumes its normal processing. During normal processing, if the participant receives an operation message from an EC or a commit message from a CC that are still in its NRCL, the participant declines to process the operation and sends back a *decline* message that contains the most recent participant failure timestamp. The decline message is interpreted by the coordinator to mean that it has not responded in a timely fashion to the most recent participant site failure. If the message is received by an EC, the EC ignores the timestamp contained in the message and aborts all active transactions that have performed update operations at the participant's site. This is because the participant has already recovered and aborted each such transaction based on a *nothing for you* reply message from the associated CC. For each transaction that has performed only read operations at the participant's site, the EC aborts the transaction only if it attempts to access (i.e., read or write) any data object at the participant. That is, a transaction that has performed only read operations is not aborted if it does not send any operation to be performed at the participant's site after the participant has recovered. This is because it does not matter whether the read–only transaction is committed or aborted at the participant's site as long as it preserves serializability. Since the transaction is serializable prior to the participant's failure, it can be committed as long as it does not submit a new operation for execution that might consequently violate serializability. Once the EC has acted upon the *decline* message, it includes an acknowledgment flag in the next operation it sends to the participant. When the participant receives an operation with an acknowledgment flag, it removes the EC from its NRCL and executes the operation knowing that the EC has complied with its *decline* message.

If on the other hand, the decline message is received by a CC, it updates its FTSL and force writes the list into stable storage. Then, the CC re–sends any commit operation that has been declined by the participant including, as in the case of an EC, an acknowledgment flag that indicates to the participant that the coordinator has updated its FTSL and the coordinator can be removed from the participant's NRCL.

Coordinator Failure

During its recovery after a site failure, an EC aborts each transaction without a `prepared` log record and forgets it. On the other hand, for each transaction with a `prepared` log record, the EC inquires its associated CC. If the CC has decided to commit the transaction, the CC responds with a commit message and waits for the acknowledgment of the EC. If the CC does not remember the transaction, it presumes that the transaction has been aborted and tells the EC by sending back an abort message.

In the event of a CC's site failure, the CC re-builds its FTSL and protocol table by adding each transaction with a `commit` record but without an associated `end` log record into the table during its recovery by using its own log. Then, the CC sends a commit message to each participant in the execution of a transaction including the transaction's EC. Once the required acknowledgments arrives, the CC writes a non-forced `end` record and forgets the transaction.

	2PC	PrC	PrA	IYV	IYV-PrA	IYV–WCC
Log force delays	2	3	2	1	1	2
Total log force writes	2n+1	n+2	2n+1	n+1	n+1	n+2
Message delays (Commit)	2	2	2	0	0	1
Message delays (Locks)	3	3	3	1	1	2
Total messages	4n	3n	4n	2n	2n	3(n+1)
Total messages with piggybacking	3n	3n	3n	n	n	n+3

Table 3: The cost of the protocols to *commit* a transaction.

	2PC	PrC	PrA	IYV	IYV-PrA	IYV–WCC	
						Abort by EC	Abort by CC
Log force delays	2	2	1	1	0	0	1
Total log force writes	2n+1	2n+1	n	n+1	0	0	1
Message delays (Abort)	2	2	2	0	0	0	1
Message delays (Locks)	3	3	3	1	1	1	2
Total messages	4n	4n	3n	2n	n	n	n+2
Total messages with piggybacking	3n	3n	3n	n	n	n	n+2

Table 4: The cost of the protocols to *abort* a transaction.

4 Evaluation of IYV–WCC

In this section we evaluate the performance of IYV–WCC along with 2PC, *presumed abort* (PrA) 2PC [7], *presumed commit* (PrC) 2PC [7] and IYV *presumed abort* (IYV-PrA) [1]. Due to space limitations, we assume that the reader is familiar with these protocols.

In our evaluation, we consider the number of coordination messages and forced log writes that are only due to the protocols (e.g., we do not consider the number of messages that are due to the operations and their acknowledgments). The cost of the protocols are evaluated during normal processing and in absence of failures. It should be pointed out that in our evaluation we concentrate on the number of sequential messages passed and forced log writes rather than their total numbers. This is because the latter is less of an issue on the performance of a protocol in gigabit–networked database environments. In Table 3, we denote by n the number of participants that participated in a transaction execution not including the site coordinating commit processing whereas, in Table 4, n does not include either the EC or the CC when the EC aborts the transaction.

The rows labeled "Log force delays" contain the sequence of forced log writes that are required by the different protocols up to the point that the commit/abort decision is made and then acknowledging the invoking transaction. 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 3, the "Log force delays" for the 2PC protocol is two because there are two force log writes (i.e., a `prepare` and a `commit` log record) between the beginning of the protocol and the time a commit decision is made by a transaction's coordinator. Also, "Message delays (Commit)" and "Message delays (Locks)" are two and three respectively, because the 2PC involves two sequential messages in order for a coordinator to make its final decision regarding a transaction (i.e., the first phase of the protocol), and three sequential messages to release all the resources (i.e., locks) held by the transaction at the participants.

The last row shows the cost of the protocols considering *piggybacking* the acknowledgments of the decision messages. This optimization can be used to eliminate the final round of messages for the commit case in 2PC, PrA, IYV, IYV-PrA and IYV–WCC; but not in the case of PrC 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, and IYV; but not with PrA, IYV–PrA and IYV–WCC. This is because an abort decision is never acknowledged in these protocols.

The tables show that for the commit case (Table 3), IYV–WCC has increased both of the number of *sequential* forced log writes and coordination messages by one, to reach a commit decision and to release the locks held by a committing transaction, when compared with IYV and its presumed abort variant. The cost of sequential forced log writes remains the same as in 2PC and PrA, but one less than PrC. The cost of sequential coordination messages to reach a commit point and release the locks held by a committing transaction is less by one when compared with 2PC, PrA and PrC. In the case of IYV–WCC, there are $3(n+1)$ total messages. This is because there are n commit messages, one for each participant, and $2n$ acknowledgment messages, two from each participant. The other 3 messages are the `intention-to-commit`, the commit final decision sent to the EC and the EC's acknowledgment.

For the abort case (Table 4), the cost to abort a transaction in IYV–WCC remains the same as in IYV-PrA when the abort decision is made by the EC. On the other hand, the cost to abort a transaction by the CC incurs an extra sequential coordination message to reach an abort decision as well as to release the locks held by the aborting transaction. Notice that if a transaction is to be aborted, it will be aborted by its EC rather than the CC in the absence of failures. Hence, the cost of aborting transactions, in the absence of failures, remains the same as in IYV-PrA, the best alternative.

Figure 3 graphically summarizes Tables 3 and 4 illustrating the sequence of coordination messages and forced log writes that involved in 2PC, IYV and IYV–WCC to reach a decision point and to release the resources held at the participants for the commit as well as the abort case.

The above evaluation shows that IYV–WCC increases the cost of commit processing when compared with the basic IYV while still maintaining its cost below 2PC and its two most common variants.

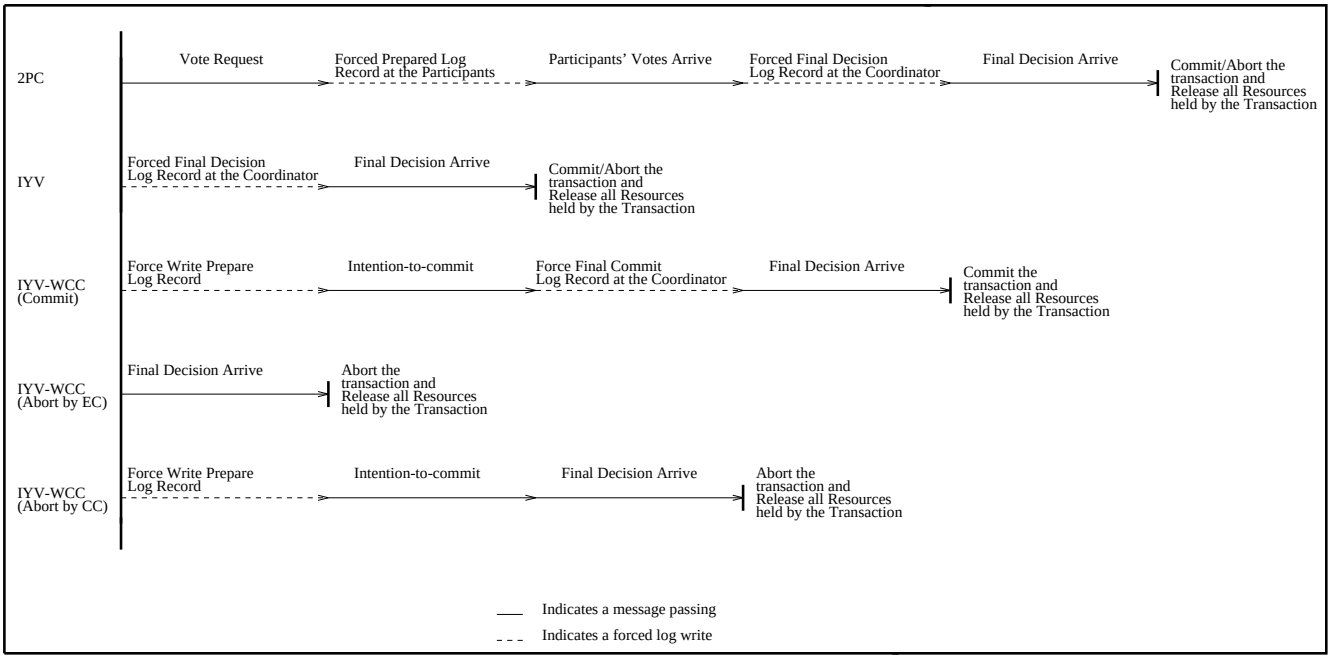

Figure 3: The number of sequential coordination messages and forced log writes required during normal processing.

5 Conclusion

The major contribution of this paper is a new coordination scheme that combines the delegation of commitment technique with a timestamp mechanism and its application to the *implicit-yes vote* commit protocol. The resulting protocol increases the degree of independent recovery in implicit yes-vote protocol, after a participant site failure, and at the same time reduces its vulnerability to blocking in the case of a communication or coordinator's site failure. Although the new protocol incurs extra coordination messages and forced log writes, we showed that its cost is still below the cost of 2PC and its most commonly known variants. The new scheme is applicable to other commit protocols such as the *coordinator log transaction protocol* [10], which shares the same basic idea behind implicit yes-vote, and *open commit protocols* [8].

Given the performance potential of IYV and IYV–WCC, a detailed simulation study of the performance of IYV, IYV–WCC and the three 2PC variants is currently underway.

Acknowledgment

We would like to thank Steven P. Levitan for his positive feedback on this work.

References

[1] Y. J. Al-Houmaily and P. K. Chrysanthis, "Two-Phase Commit in Gigabit-Networked Distributed Databases," *Proc. of the 8th Int'l Conf. on Parallel and Distributed Computing Systems*, pp. 554–560, Sept. 1995.

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

[3] H. Garcia-Molina, N. Halim, R. P. King and C. A. Polyzois, "Management of a Remote Backup Copy for Disaster Recovery," *ACM TODS*, Vol. 16, No. 2, pp. 338–368, 1991.

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

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

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

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

[8] K. Rothermel and S. Pappe, "Open Commit Protocols Tolerating Commission Failures," *ACM TODS*, Vol. 18, No. 2, pp. 289–332, June 1993.

[9] G. Samaras, K. Britton, A. Citron and C. Mohan, "Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment," *Proc. of the 9th Int'l Conf. on Data Eng.*, pp. 520–529, 1993.

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