ELSEVIER

Journal of Systems Architecture 46 (2000) 809–833

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

An atomic commit protocol for gigabit-networked distributed database systems

Yousef J. Al-Houmaily [a], Panos K. Chrysanthis [b,1] *

[a] *Department of Computer Programs, Institute of Public Administration, Riyadh 11141, Saudi Arabia*
[b] *Department of Computer Science, University of Pittsburgh, 220 Alumni Hall, Pittsburgh, PA 15260, USA*

Received 10 September 1998; received in revised form 19 March 1999; accepted 10 September 1999

Abstract

In the near 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 amounts of data will not pose a problem. Furthermore, computer systems are expected to become even more reliable than today's systems with long mean time between failures and short mean time to repair. In this paper, we present *implicit yes-vote* (IYV), a one-phase atomic commit protocol, that exploits these new domain characteristics to minimize the cost of distributed transaction commitment. IYV *eliminates* the explicit voting phase of the *two-phase commit* protocol, hence reducing the number of *sequential* phases of message passing during normal processing. In the case of a participant's site failure, IYV supports the option of *forward recovery* by enabling partially executed transactions that are still active in the system to resume their execution when the failed participant is recovered.

Keywords: Atomic commit protocols; One-phase commit protocol; Distributed transaction processing; High speed networks; Distributed systems

1. Introduction and motivation

Transactions in a *distributed database system* (DDBS) 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 as if the transaction has never existed. This is achieved by employing an *atomic commit protocol* (ACP) that executes a commit or an abort operation across multiple sites as a single logical operation. The simplest and most studied ACP is the *two-phase commit protocol* (2PC) [17,22].

Commit processing consumes a substantial amount of a transaction's execution time [30] and any delay in making and propagating the final decision reduces the level of concurrency and adversely affects the performance of a DDBS. The system performance is also degraded by having to abort partially executed transactions due to communication and site failures, in particular just before a commit decision is made. In other words, an

* Corresponding author. Tel.: 41-412-6248924; fax: 41-412-6248854.

E-mail address: panos@cs.pitt.edu (P.K. Chrysanthis).

[1] Supported in part by National Science Foundation under grant IRI-9210588 and IRI-9502091.

1383-7621/00/$ - see front matter
PII: S1383-7621(99)00040-5

efficient ACP should reach a final decision with minimum communication delay and should allow partially executed transactions to resume their execution on a failed participant when the participant recovers. Owing to the relatively low data transfer rates of traditional networks, current ACPs have been designed with the focus on minimizing the *amount* of data to be transferred and the *number* of messages to be exchanged while current database management systems do not support *forward* recovery of atomic transactions.

However, future DDBSs are expected to become even more reliable than today's systems executing on computers with relatively long *mean time between failures* and short *mean time to repair* interconnected via high speed networks. These networks will support 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 [7,8,21]. That is, the size of messages in a database protocol will be less of a concern than the required number of *rounds* or *sequential phases* of message passing. Given this observation, we are prompted to ask the following two questions:

1. Is it possible to improve the performance of the 2PC by permitting large messages; that is, can we reduce the number or rounds of messages in the 2PC?
2. Is it possible to support forward recovery, i.e., recover partially executed transactions after a failure by not placing any limitations on the size of a message?

In this paper, we present *implicit yes-vote* (IYV), a one-phase ACP, that combines the properties implied in both of the above two questions. IYV is targeted for environments with fixed number of high performance coordinators or application servers. IYV improves on the current ACPs by exploiting the performance and reliability properties of future gigabit-networked DDBSs. Specifically, IYV *eliminates* the explicit *voting phase* of the 2PC by overlapping the participants' votes with the execution of transactions' operations, hence reducing the number of sequential co-ordination messages during normal commit processing. The underlying system assumption in IYV is that all sites employ *strict two-phase locking protocol* (S-2PL) for concurrency control [16,10], which is the most commonly used protocol.

In the case of a participant failure, IYV supports the option of *forward recovery* by enabling partially executed transactions to resume their execution on a failed participant when the participant recovers. By assuming *context free* transactions at the participants, [2] forward recovery is achieved through a low-cost replication of the *redo* log records that are generated during the execution of a transaction's operations at both the transaction's coordinator and the participants, and by propagating the *read* locks that are held by the transaction at a participant to the transaction's coordinator. Thus, forward recovery in IYV can be thought of as an instance of forward recovery proposed for distributed computations, e.g., [15]. As opposed to forward recovery in distributed processes, participants in IYV do not asynchronously checkpoint their local states and after a participant failure, non-failed participants do not have to partially rollback for the entire distributed transaction to reach a consistent state from which the transaction can resume its execution. Here, a transaction's view of the database state at a failed participant is reconstructed, if the local execution state of the transaction residing at its coordinator site is unaffected. In this respect, forward recovery in IYV is not the same as the notion of partial recovery in extended transactions, such as in nested transactions where a failed subtransaction may be rolled back and re-executed by design [27]. In IYV, distributed transactions are traditional, unstructured transactions which are executed in a distributed manner by the database system.

The rest of the paper is structured as follows. In Section 2, we overview the 2PC and its two common variants as well as previously proposed one-

[2] A transaction is said to be context free at a participant if each operation execution at the participant is not a function of the previously executed operations invoked by the same transaction [19]. For example, read operations using a common cursor are not independent where the cursor is part of the context of the invoking transaction.

phase ACPs. In Section 3, the IYV protocol is introduced and its behavior in the presence of failures is discussed in detail. In Section 4, we discuss the applicability of IYV and its assumptions. In Section 5, we apply the *presumed abort* 2PC (PrA) optimization [25,26] to IYV, motivated by the fact that PrA has been widely implemented [12] and adopted by the ISO OSI-TP [36] and X/Open DTP [11] standards. We also propose a *read-only* optimization that can be combined with IYV and its PrA optimization to further reduce their cost for read-only transactions and extend IYV to the *multi-level* transaction execution model [5,26]. In Section 6, we compare IYV with those protocols discussed in Section 2. Unlike the traditional way of evaluating the performance of ACPs which is based on counting the number of messages and forced log writes, in Section 7, we evaluate these different commit protocols in terms of the number of *sequential* messages and forced log writes that are required to reach a decision point and to release all the locks held by transactions as in [33,34]. Section 8 concludes this paper.

2. Background and related work

In a distributed database system, data are typically stored in disjoint partitions at different sites. This data distribution is transparent to a distributed transaction that accesses data by submitting database operations to its *coordinator* which is assumed to be, without loss of generality, the transaction manager of the site where the transaction has been initiated. In this paper, we assume that a transaction is a partial order of read (R) and write (W) operations that are confined within a begin (B) and a commit (C) or an abort (A) transaction management primitives.

When a coordinator receives an operation on a data item, it sends the operation to the appropriate site for execution. If the coordinator receives an abort request from the transaction, it sends an abort request to all the *participants*, i.e., the sites participating in the execution of the transaction. On the other hand, when the coordinator receives a commit request from the transaction, it initiates an atomic commit protocol.

As shown in Fig. 1, the basic 2PC [17,22], as the name implies, consists of two phases, namely a *voting phase* and a *decision phase*. During the voting phase, the coordinator of a distributed transaction requests all the sites participating in the transaction's execution to *prepare to commit* whereas, during the decision phase, 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, *acknowledges* the decision and releases all the resources held by the transaction (i.e., releases the locks held by the transaction, removes the transaction control block from its table, etc.). The coordinator completes the protocol when it receives acknowledgments from all the participants and forgets about the transaction by removing any entry associated with the completed transaction from its protocol table. The *protocol table* is stored in main memory and for each transaction, the coordinator records the identities of the sites that need to participate in the commitment of the transaction and the progress of the protocol once it is initiated.

The resilience of 2PC to system and communication failures is achieved by recording the

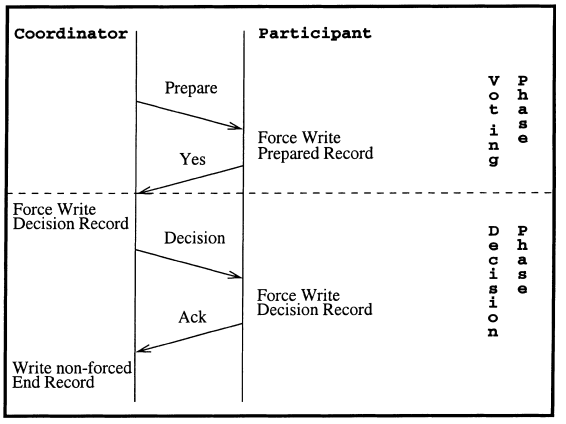


Fig. 1. The basic two-phase commit protocol.

progress of the protocol in the logs of the coordinator and the participants (see Fig. 1). 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 the 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 all participants have received the final decision and the log records pertaining to the transaction can be garbage collected when necessary.

The basic 2PC protocol is also referred to as the *presumed nothing* 2PC (PrN) [23] because it treats all transactions uniformly, whether they are to be committed or aborted, requiring information to be explicitly exchanged and logged at all times. However, in case of a coordinator's failure, there is a hidden presumption in PrN by which the coordinator considers all active transactions at the time of the failure as aborted transactions. This presumption allows a coordinator in 2PC not to record the beginning of the protocol in the stable log and hence not to force-write any log records prior to the decision phase. (Note that a force-write involves a disk access that suspends the protocol until the disk access is completed.) If a participant inquires the coordinator about an active transaction after the coordinator has failed and recovered, the coordinator, not remembering the transaction, will direct the participant to abort the transaction by presumption.

The *presumed abort* PrA is a 2PC variant that reduces the cost associated with aborted transactions by making the abort presumption of PrN explicit [25,26]. When the coordinator of a transaction decides to abort the transaction, in PrA, the coordinator discards all information about the transaction from its protocol table and submits an abort message to all the participants without logging an abort decision, as opposed to PrN. After a coordinator failure, if a participant inquires about the outcome of a transaction, the coordinator, not finding any information regarding the transaction will direct the participant to abort the transaction by presumption. Furthermore, in PrA, the coordinator of a transaction does not require abort acknowledgments from the participants because it can discard all information pertaining to the transaction from its protocol table without them. Since the participants are not required to acknowledge abort decisions, they do not have to force-write abort log records. Instead, they write (non-forced) abort records in the log buffer in main memory. Hence, in the case of abort, PrA saves a forced log write at the coordinator's site and a forced log write and an acknowledgment message from each participant. For the commit case, the cost of PrA remains the same as in the PrN.

Assuming that a transaction is most probably going to commit if it has finished its execution and issued a commit request, the PrC variant [26] was designed to reduce the cost associated with committing transactions. Instead of interpreting missing information about transactions as abort decisions which is the case in PrA, in PrC, coordinators interpret missing information about transactions as commit decisions. However, in PrC, the coordinator of a transaction has to force-write an *initiation* log record before submitting the prepare to commit message to the participants. The initiation record ensures that missing information about the transaction will not be wrongly mis-interpreted as a commit case after a coordinator's site failure. Thus, this record is necessary for the correctness of this 2PC variant. In addition, the initiation record contains the identities of the participants that are needed for recovery. In the case of PrN and PrA, this information is recorded in the decision log record.

To commit a transaction, the transaction's coordinator force-writes a commit record to logically eliminate the initiation record, then sends out the commit decision to all the participants and discards any information about the transaction. When a participant receives the commit message, it writes a non-forced commit record and commits the transaction. Since the coordinator can discard all information about a committed transaction without the acknowledgments of the participants,

a participant does not have to acknowledge a commit decision.

To abort a transaction, on the other hand, the transaction's coordinator does not have to force an abort record. Instead, the coordinator submits the abort decision to all the participants and waits for their acknowledgments. Once the coordinator receives the acknowledgments, it discards all information pertaining to the transaction from its protocol table and writes a non-forced end record. Each participant, in this case, has to force-write an abort record and then acknowledges the coordinator. As far as the total message count and forced log writes are concerned, the cost to abort a transaction remains the same as in the PrN [3] while the cost to commit a transaction is reduced by a forced log write and an acknowledgment message from each participant on the expense of an extra forced log write at the coordinator (i.e., the initiation record).

PrA and PrC are usually coupled with the *read-only* optimization [25,26,30]. In this optimization, a participant votes *read-only* if it has executed only read operations. Using this optimization, a participant can release all the locks held by the transaction once it votes. Furthermore, a read-only participant does not participate in the second phase of these protocols and hence it does not have to know about the outcome of the transaction. Also, a participant does not have to write any log records regarding a read-only transaction. If a transaction is read-only (i.e., all the operations it has submitted to all the participants are read operations), the coordinator, in both PrA and PrC, treats the transaction as an aborted one. This is because it is cheaper to abort than to commit a read-only transaction with respect to logging. Recall that a coordinator does not write any log records in PrA whereas abort records are written in a non-forced manner in PrC. (For a new read-only optimization see [4,5].)

Another optimization that eliminates a participant from the voting phase is the *unsolicited vote* (UV) optimization [35]. In this optimization, if a participant knows when it has executed the last operation pertaining to a transaction, it does not have to wait for a prepare to commit message. Instead, it sends its vote in its own initiative once it recognizes that the transaction has no more operations to process. In the case that all the participants can send unsolicited votes, this optimization completely eliminates the explicit voting phase of 2PC and becomes a one-phase ACP.

The *early prepare* protocol (EP) combines the UV with PrC [33,34]. Since PrC requires the identities of the participants to be explicitly recorded at the coordinator's log in a forced initiation record, the number of forced initiation records pertaining to a transaction is equal to the number of participants that executed the transaction in EP. This is because the initiation record has to be updated and force-written each time a new participant is involved in the execution of the transaction. Furthermore, since EP does not make any assumptions about the last operation of a transaction submitted to a participant, the participant has to prepare the transaction each time it executes an operation for the transaction and prior to acknowledging the operation. This means that the number of forced prepared records pertaining to a transaction at a participant is equal to the number of operations submitted by the transaction and executed by the participant.

Another one-phase atomic commit protocol that builds on EP is the *coordinator log* protocol (CL) which assumes that transactions are most probably going to commit and are (very) short (i.e., transactions deal with small amount of data) [33,34]. CL eliminates the need for the forced logging activities required by EP at the participants' sites by having the coordinators maintain the logs and using *distributed write-ahead logging* (DWAL) [13]. That is, the stable log of a participant is distributed (i.e., scattered) across multiple-coordinator sites. The CL protocol also eliminates the need for the initiation log records of EP at the coordinators at the expense of requiring from a coordinator to communicate with all the participants in the system during its recovery after a

[3] The forced abort record in PrN is replaced by a forced initiation record in PrC.

failure. This is in order to determine the set of active transactions prior to the coordinator's failure and to abort them instead of wrongly assuming their commitment.

As it will become apparent in the rest of the paper, the IYV protocol combines the advantages of UV, EP and CL while alleviating most of their disadvantages.

3. The implicit yes-vote protocol

In this section, we present the details of our protocol and discuss what kind of information is needed to be logged and where (see Fig. 2). Then, in Section 3.2, we discuss the recovery aspects of IYV.

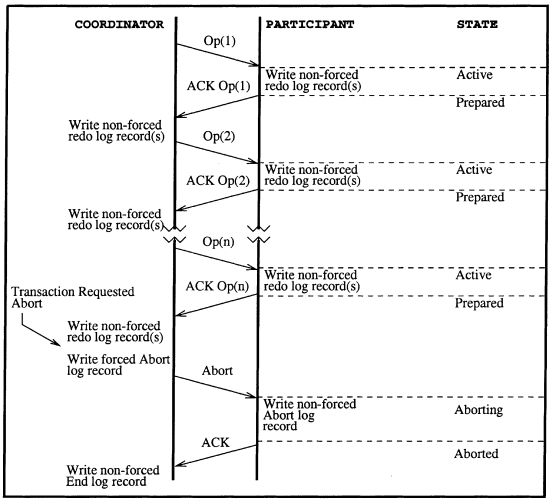
3.1. Description of IYV

As in the case of all the other protocols, a coordinator records information pertaining to the execution of a transaction in its protocol table in main memory. Specifically, a coordinator keeps for each transaction the identities of the participants and any pending request at a participant.

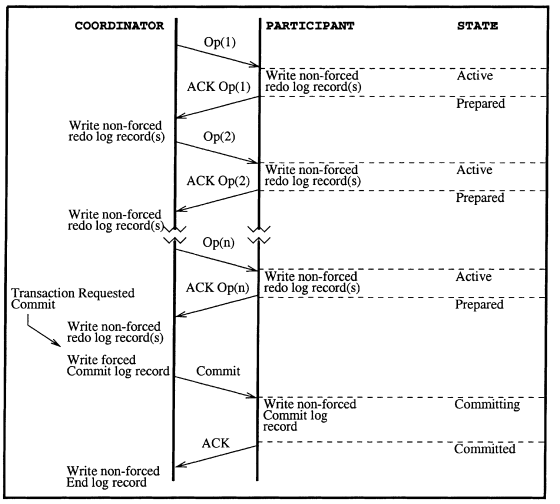
Each participant maintains a *recovery-coordinators' list* (RCL) that contains the identities of the coordinators that have active transactions at its site and must be contacted during the recovery of the participant after a failure. In order to survive failures, an RCL is kept in the stable log. Thus, when a participant receives the first operation of a transaction, if the identity of the coordinator of the transaction is not already in its RCL, the participant adds it to its RCL, force-writes the RCL in its log and then executes the operation. In order to avoid searching the entire RCL in the case that all the coordinators in the system are active at a participant, an *all-active flag* (AAF) is used. A participant sets AAF once it force-writes an RCL containing the identities of *all* the coordinators and does not consider the RCL as long as the AAF is set.

An operation submitted by a transaction can be either an *update* or a *read* operation. Following the S-2PL, before the execution of an operation, a participant places a write lock on each data item that is to be updated and a read lock on each data item that is to be read by the operation. The locks are kept in a lock table in main memory and are held until the commit time of the transaction.

Once an operation is executed successfully, the participant acknowledges (ACK) the coordinator with a message that contains the results of the operation. In IYV, in order to provide for the option of forward recovery, the participant also includes all the read locks that have been acquired during the execution of the operation. For an



(a) Abort case



(b) Commit case

Fig. 2. The IYV coordination messages and log writes.

update operation, a participant also includes in the acknowledgment message all the *redo* log records that have been generated during the execution of the operation with their corresponding local *log sequence numbers* (LSNs). [4] As shown in Fig. 2, a participant does not force its log into stable storage prior to acknowledging an operation. Once a participant acknowledges an operation, it implicitly enters a prepared to commit state with respect to the invoking transaction. While in this state, the participant cannot unilaterally commit or abort the transaction until it receives a final decision. On the other hand, a participant returns to an active state with respect to the transaction when it receives a new operation. If a participant fails to process an operation, it aborts the transaction and sends a *negative* acknowledgment (NACK) to the transaction's coordinator.

When a coordinator receives an ACK for a write operation from a participant, it writes a non-forced log record containing the received redo records along with the participant's identity. Hence, the coordinator's log contains a partial 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. The coordinator also records any read locks included in an ACK message along with the identity of the participant in a *participants' lock table* (PLT) which is part of its protocol table. As a result, the coordinator's PLT contains a partial image of the lock table of each participant. The PLT is used in the case of a participant failure in order to enable the participant to recover its state *exactly* as it was prior to the failure requiring both read and write locks, thereby allowing partially executed transactions that are still active in the system to forward recover and resume their execution after the participant has recovered without violating serializability (see Section 4).

If the coordinator receives either an abort request from a transaction or a negative acknowledgment regarding the transaction from a participant, the coordinator decides to abort the transaction. Once the coordinator decides to abort the transaction, it force-writes an abort log record and then, sends an abort message to all the participants. On the other hand, when the coordinator of a transaction receives a commit primitive from the transaction, it waits for the acknowledgments of the transaction's pending operations and then commits the transaction. On a commit decision, the coordinator force-writes a commit log record prior to sending commit messages to the participants. In either case, the decision log record includes the identities of all the participants.

When a participant receives a commit (abort) message regarding a transaction, it writes a non-forced commit (abort) log record and commits (aborts) the transaction, releasing all the transaction's resources. A participant acknowledges a decision message only after the corresponding decision log record is placed into stable storage as a result of a subsequent force-write or flush of the log onto stable storage. If the transaction was the last active transaction submitted by its coordinator, the participant resets its AAF if it is set, deletes the transaction's coordinator from the RCL, force-writes the updated list in its log and then acknowledges the message.

Finally, when the coordinator receives the acknowledgment of the decision message from all the participants, it writes a non-forced end log record and discards all information pertaining to the transaction from its protocol table including the locks in PLT, knowing that no participant will inquire about the transaction's status in the future. We summarize the IYV protocol in Fig. 3.

3.2. Recovery in IYV protocol

As shown in Fig. 4, IYV is resilient to both communication and site failures. As is the case in the basic 2PC and all its variants, site and communication failures are detected by timeouts. In the following two subsections, we discuss the correctness of IYV in the presence of failures. Our discussion also serves as an informal prove of correctness which is similar to the prove of correctness of the basic 2PC [10].

[4] The LSNs can be uniquely identified using increasing integers that are preceded by (logical) site numbers.

Coordinator's Protocol

For each transaction initiated locally

- For each database operation, submit the operation to an appropriate participant and update the participants list if necessary.
  - If a negative acknowledgment is received (Abort):
    1. Force-write an abort record.
    2. Submit an *abort* message to all participants.
    3. Write a non-forced end record when all the participants acknowledge the *abort* message.
  - Else (the transaction is implicitly prepared at the participant that executed the operation):
    1. Write any redo records received from the participant with their corresponding LSNs into the log and any read locks into the PLT.
    2. Acknowledge the operation to the transaction.
    3. Wait for the next operation to be submitted by the transaction.
- When a Commit primitive is received from the transaction:
  1. Force-write a commit record.
  2. Submit a *commit* message to all participants.
  3. Write a non-forced end record when all the participants acknowledge the message.
  4. Discard all information about the transaction from the protocol table including the locks in PLT.
- When an Abort primitive is received from the transaction:
  1. Force-write an abort record.
  2. submit an *abort* message to all participants.
  3. Write a non-forced end record when all the participants acknowledge the message
  4. Discard all information about the transaction from the protocol table including the locks in PLT.

Participant's Protocol

- For the first operation of a transaction, check the AAF. If the AAF is not set, if necessary, update the RCL and force write the log. If the updated RCL contains the identities of all coordinators in the system, set the AAF.
- For each database operation:
  1. Execute the operation and log all the records that are generated during the execution of the operation.
  2. Acknowledge the operation with a message that contains all the redo records, if any, with their corresponding local LSNs and all the read locks acquired during the execution of the operation when executed successfully. Otherwise, send a negative acknowledgment and abort the transaction, releasing its locks and deleting the identity of its coordinator from the RCL.
- When a final Decision (i.e., *commit* or *abort*) message arrives:
  1. Write a non-forced Decision log record.
  2. Release all locks.
  3. Delete the identity of the coordinator from the RCL if this is the last transaction submitted by its coordinator, force-write the list into the log and reset the AAF if it is set.
  4. Acknowledge the Decision message when the Decision record is forced into the stable log.

Fig. 3. The IYV protocol.

3.2.1. Communication failures

Although communication failures are assumed to be rare in high speed networks, there are three points during the execution of IYV where a communication failure might occur while a site is waiting for a message. The first point is when a participant has no pending acknowledgments and is waiting for a new operation or a final decision. This is shown as the first case of the communication failure in the participant algorithm in Fig. 4. 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

Coordinator's Algorithm

In case of a communication failure:

1. 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.
2. Re-submit the commit decision of each committed transaction without and end record after the failure is fixed.

In case of a site failure:

1. For each transaction that has a final 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:

1. 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.
2. Abort all active transactions (i.e., transactions with pending acknowledgments).

In case of a site failure:

1. Analysis phase: identify committed, aborted and active transactions. Also, determine the largest LSN.
2. For each coordinator in the RCL, send a *recovering* message containing the largest LSN.
3. Undo the effects of aborted and active transactions.
4. Once the *repair* 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 and retain their locks.
6. Resume normal processing.

Fig. 4. Recovery in IYV protocol.

participant enforces the final decision and then acknowledges it, while in the latter case, the participant waits for further operations.

The second point is when the coordinator of a transaction is waiting for an operation acknowledgment from a participant. This is shown as the first case of communication failures in the algorithm of the coordinator in Fig. 4. In this case, the coordinator may abort the transaction and submit a final abort decision to the rest of the participants. Similarly, the participant may abort the transaction if the communication failure has occurred while the participant has a pending acknowledgment. This is shown as the second case of communication failures in the participant algorithm in Fig. 4. Notice that the coordinator of a transaction may commit the transaction despite communication failures with some participants as long as these participants have no pending acknowledgments.

The third point is when the coordinator of a transaction is waiting for the acknowledgments of a commit 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 re-submits the decision once these communication failures are fixed (the second case of communication failures in the coordinator's algorithm). When a participant receives the commit decision after a failure, it either

just acknowledges the decision if it has already received and enforced the decision prior to the failure, [5] or enforces the decision and then sends back an acknowledgment.

3.2.2. Site failures

As mentioned above, we are assuming that each site employs physical logging and uses an Undo/Redo crash recovery protocol in which the undo phase *precedes* the redo phase. It should be pointed out that IYV can also be combined with *logical* or *physiological* write-ahead logging schemes [19,24]. However, for ease of presentation, we only discuss IYV when physical write-ahead logging (WAL) is used [19,20].

3.2.2.1. Coordinator failure. Upon a coordinator restart, after a failure, the coordinator re-builds its protocol table by scanning its stable log. The coordinator needs to consider only those transactions that have commit decision records without a corresponding end records. As shown in the coordinator recovery algorithm (the first step after a site failure in Fig. 4), 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 for each of these transactions by re-submitting its decision to all the participants and resumes normal operation.

As in the case of a communication failure, if a participant has already received and enforced a final decision prior to the 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 writes a non-forced decision record and then sends an ACK message when the decision record is in the stable log.

For those transactions without final decision records (i.e., those transactions that were active prior to the failure or their non-forced abort records did not make it to the stable log before the failure), the coordinator can safely forget them and consider them as aborted transactions (the second case of the coordinator recovery algorithm). If a participant in the execution of one of these transactions has a pending acknowledgment, when it times out due to the coordinator site failure, it will abort the transaction, as in the case of a communication failure that we discussed above. On the other hand, if the participant is left blocked (i.e., the participant has acknowledged all of a transaction's operations and is in the implicit prepared to commit state), when the coordinator recovers, the participant will inquire about the status of the transaction. The coordinator, not remembering the transaction after its recovery, will respond with an abort message by using an implicit presumption as in PrN. For those transactions that are associated with decision records as well as end records (the third case in the coordinator recovery algorithm), the coordinator can safely discard all information about these transactions, knowing that all participants are informed of its decisions and no participant will inquire about these transactions' outcome in the future.

3.2.2.2. Participant failure. Also shown in Fig. 4 are the steps of the participant recovery after a site failure. Since the entire log might not be written into a stable storage until after the log buffer overflows, the log may not contain all the redo records of the transactions committed by their perspective coordinators after a failure of a participant. Thus, at the beginning of the *analysis phase* of the restart procedure, the participant determines the largest LSN which is associated with the last record written in its log that survived the failure (the first step in the participant recovery algorithm) and sends a *recovering* message that contains the largest LSN to all coordinators in its RCL (the second step in the recovery algorithm). This LSN is used by the coordinators to determine missing redo log records at the participant which are replicated in their logs and are needed by the participant to fully recover.

While waiting for the reply messages to arrive from the coordinators, the *undo phase* can be

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

performed, even potentially completed, and the *redo phase* can be initiated. That is, the participant recovers those aborted and committed transactions that have decision records pertaining to them already stored in its stable log (the third step in the algorithm) while waiting for the reply messages to arrive from the coordinators. This ability of overlapping the undo phase with the resolution of the status of active transactions and the repairing of the redo part of the log, partially masks the effects of dual logging and communication delays. Note that because of the use of WAL, all the required undo log records that are needed to eliminate the effects of any transaction on the database are always available in the participant's stable log and never replicated at the coordinators' sites.

When a coordinator receives a recovering message from a participant, it will know that the participant has failed and is recovering from the failure. Based on this knowledge, the coordinator checks its protocol table to determine each transaction that the participant has executed some of its operations and the transaction is either still active in the system (i.e., still executing at other sites and no decision has been made about its final status, yet) or has committed but did not finish the protocol (i.e., a final decision has been made but the participant has not acknowledged the decision prior to its failure). For each transaction that is finally committed, the coordinator responds with a commit status along with a list of all the transaction's redo records that are stored in its log and have LSNs greater than the one that was included in the recovering message of the participant.

For each active transaction that is still in progress in other sites, the coordinator has the option to either abort or forward recover the transaction. If the coordinator decides to abort the transaction, it sends abort messages to all participants to rollback the transaction. If the coordinator decides to forward recover the transaction, it responds with a *still-active* status containing, as in the case of a committed transaction, a list of the redo records associated with LSNs greater than the one included in the recovering message of 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.

All these responses and redo log records are packaged with the read locks acquired by active transactions in a single *repair* message and sent back to the participant. If a coordinator has no active transactions and all committed transactions have been acknowledged as far as the failed participant is concerned, the coordinator sends an ACK repair message, indicating to the participant that there are no transactions to be recovered as far as this coordinator is concerned.

Once the participant has received reply messages from all the coordinators (the fourth step in the participant recovery algorithm in Fig. 4), the participant 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 completed (the fifth step in the participant recovery algorithm), the participant acknowledges all commit decision responses once these commit decisions are in its stable log, as in the case of normal processing. Then the participant resumes its normal processing (the last step in the participant recovery algorithm). Thus, in IYV's recovery algorithm, a long-executing transaction is not necessarily aborted as a result of a participant failure as would be the case in all other ACPs. That is, since a participant can rebuild its lock table after a failure and redo the effects of partially executed transactions, the coordinator of a transaction has the option to forward recover the transaction if it is still active in the system after the failed participant is recovered.

3.2.2.3. Simultaneous coordinator and participant failures. The case of an overlapped coordinator and participant failure is handled using the same procedure as we discussed above. That is, a failed coordinator recovers the transactions initiated at its site as we discussed above. Similarly, a failed participant recovers as we discussed above. However, it should be noticed that if one of the coordinators in the RCL of a recovering participant is down, the participant is left blocked and cannot recover until the failed coordinator has recovered

and sent to the participant the pending repair message. Although this situation might seem to be drastic, it is expected to be very rare in the context of future database systems given the reliability characteristics of modern computing systems. Hence, this represents a reasonable trade-off between fast commit processing during the normal (i.e., non-failure) case at the expense of slower recovery in the rare failure case.

4. Assumptions, correctness and applicability

4.1. Basic system assumptions

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 it is in the prepared to commit state. A participant may be required to abort a transaction either for correctness reasons, such as ensuring serializability, or for performance reasons, such as minimizing transaction blocking. Regarding the latter, given that transactions are finite, we assume 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.

As we mentioned earlier, most commercial database management systems use S-2PL for concurrency control and physical WAL for recovery. Now, consider a distributed system in which all the sites employ S-2PL. In such a distributed system, participants *never* abort transactions to ensure atomicity, i.e., there are no cascading aborts, and *only* abort transactions in active states (i.e., transactions having outstanding operations' acknowledgments) to resolve deadlocks.

Theorem 1. *If each participant employs S-2PL for concurrency control, it is not possible for a transaction to be involved in a non-serializable execution, a local deadlock at a participant, or a global deadlock when all the operations that were submitted by the transaction to the participants have been executed and acknowledged.*

Proof. The proof proceeds by contradiction. Assume that all the operations submitted by a transaction have been executed and acknowledged and the transaction is involved in (1) a non-serializable execution or (2) a deadlock.

According to the S-2PL rules, an operation submitted by a transaction is executed only after the locks required for the execution of the operation on the data items are acquired. This rule implies that an operation is not acknowledged until after the required locks for the execution of the operation are acquired.

The first part (1) contradicts the fact that S-2PL schedulers produce serializable histories [10]. If the transaction is involved in a non-serializable execution, at least one of its operations would have been blocked rather being acknowledged which contradicts the assumption that all the transaction's operations have been executed and acknowledged.

The second part (2) contradicts the fact that if a transaction is involved in a deadlock, at least one of its operations is blocked awaiting to hold some locks on some data items which again contradicts the assumption that all the operations pertaining to the transaction have been acknowledged. □

Corollary 1. *A local deadlock at a participant site that employs S-2PL involves only active transactions* (*i.e.*, *transactions with pending operations*).

Proof. As above, the proof proceeds by contradiction. For a deadlock to occur, the *hold-and-wait* condition must exist. If we assume that a transaction is involved in a local deadlock at a participant site after all the operations submitted to the participant have been executed and acknowledged, it means that the transaction is holding locks on some data items and is waiting to hold locks on other data items. However, since all the operations of the transaction have been executed and acknowledged by the participant, all the locks required for the execution of the operations submitted to the participant have been acquired. Hence, the hold-and-wait condition cannot exist after all the operations submitted by a transaction to a participant have been acknowledged and a local deadlock can only involve active transactions. □

Note that participants using an *optimistic* concurrency control protocol [10] do not exhibit the above property. That is, they might abort a transaction even though the transaction is not in an active state in order to ensure serializability [10]. Hence, IYV is not applicable in this case. On the other hand, it can be shown, as in Theorem 1, that participants using a *pessimistic* concurrency control protocol (other than S-2PL) that avoids cascading aborts *never* abort transactions to ensure atomicity and *only* abort transactions in active state to ensure serializability.

It should be pointed out that IYV is designed for client–server architecture in which interactions between sites are structured along the lines of remote procedure calls in a synchronous or asynchronous manner. IYV is not suitable for environments where requests are not always acknowledged as in peer-to-peer environments using conversations. Also, as any other one-phase ACP, IYV is not applicable in systems where a voting phase is required to perform some form of validation at commit time. Such systems include those that process transactions associated with differed consistency constraints which are validated at commit time.

4.2. Assumptions on transaction properties

Many applications trade-off increased concurrency for consistency. For this reason, ANSI SQL defines four isolation levels [6]: (1) *read uncommitted*, (2) *read committed*, (3) *repeatable read* and (4) *serializable*. The main underlying assumption of IYV is that each site employs a scheduler that produces rigorous histories such as S-2PL which corresponds to isolation level 3. Note that, in this paper, we follow the definitions of the isolation levels for locking-based systems that are presented in [9].

IYV, as any other ACP, is not necessary in the case of isolation level 0 because there is no notion of transaction at this level. A transaction running in this level can read any data item without requesting a lock and writes data items with short-term locks which are held only for the duration of the operation (non-two-phase writes).

In isolation level 1, a transaction is allowed to read uncommitted data. If isolation level 1 atomicity is defined such that a transaction is aborted if it has read data written by an aborted transaction (traditional notion of atomicity), then IYV is not applicable. This is because a participant in IYV cannot abort a transaction due to cascading aborts at the commit time of the transaction. On the other hand, if isolation level 1 atomicity is only with respect to the write operations, then IYV can be used to ensure that either all the writes are committed at all participants or none at all.

IYV is applicable in isolation level 2 (which includes the cursor stability and repeatable read refinements). In this level, transactions read only committed data while write locks are of long duration (i.e., writes locks are held until a transaction is either committed or aborted). Thus, there is no possibility of cascading aborts and there is a need to synchronize the writes. But, it should be noted here that the forward recovery option is not applicable except in the case of repeatable reads since read locks are of short term and reads cannot be repeated.

4.3. Assumptions on recovery

As discussed above, recovery in IYV is based on the traditional Undo/Redo schemes in which the undo phase precedes the redo phase. This allows the analysis phase at a participant that involves communication with the coordinators to proceed concurrently with the undo phase and potentially part of the redo phase. This is not possible in the case of recovery schemes such as ARIES [24], in which the redo phase precedes the undo phase. In the case of a recovery scheme where the redo phase precedes the undo phase, a participant is blocked and cannot initiate recovery until it receives responses from the required coordinators (i.e., all coordinators in the RCL of the participant). That is, IYV can incorporate other recovery schemes such as ARIES, which is a physiological recovery scheme, but it will not offer the same efficiency during recovery in this case.

Now, let us also make the need for replicating the read locks that are held by the transactions at the coordinators' sites clear, a need which allows

the option of forward recovery without violating consistency. Assume that we have two transactions T_1 and T_2, submitted at two different coordinators. T_1 reads data item x, writes data item y and then commits, whereas T_2 writes both data items x and y, and then commits. Here, r_i [x] (w_i [x]) denotes a read (write) operation performed by transaction T_i

$$T_1 : \ r_1[x]\, w_1[y] c_1,$$
$$T_2 : \ w_2[x]\, w_2[y] c_2.$$

Furthermore, assume that the first operation of T_1, r_1 [x], has been executed successfully and acknowledged. After that, the participant where the data items are stored fails. The resulting *history* of execution is as follows:

$$H_1 : r_1[x]\, \text{Crash}.$$

At this point, assume that the participant has received acknowledgment messages in response to its recovering messages from all coordinators including the coordinator of T_1 which has indicated that T_1 is still active in its acknowledgment message. Based on these replies, the participant finishes its recovery procedure using its own log and knowing that T_1 is still in progress but there is no redo actions that are needed to be used with this transaction since the transaction has performed only a read operation. Now, if we allow T_1 to forward recover after the participant has recovered without being able to reconstruct the exact lock table of the participant as it was before the failure, we might end up with the following non-serializable history

$$H_2 : \ r_1[x]\, w_2[x]\, w_2[y]\, c_2\, w_1[y] c_1.$$

H_2 is not serializable because it is cyclic (i.e., $T_1 \rightarrow T_2 \rightarrow T_1$). H_2 is possible because the participant after losing its lock table, could not re-acquire the read lock on x on behave of T_1 as it was the case prior to the failure, allowing T_2 to acquire a write lock and change x after the participant has recovered. Once T_2 has committed, the participant receives a new operation pertaining to T_1 that also modifies the value of x. Since T_2 has already committed and all its locks have been released, T_1 can acquire a write lock on x and modifies it, resulting in a non-serializable execution. However, duplicating the read locks at the coordinators in IYV allows a participant to re-build its lock table after a failure and to prevent execution histories similar to the one described above from occurring.

Here, it should be pointed out again that forward recovery is an option in IYV which is currently applicable only for context-free transactions with respect to participants. A transaction program always executes at the site of its coordinator and cannot be forward recovered in the case of the coordinator's site failure without the help of checkpoints which save the transaction program's local state [15]. In IYV, by assuming context-free transactions at the participants, the read locks and write log records are sufficient to reconstruct the context of transactions and forward recover them on a failed participant. For more complicated type of contexts, a coordinator needs to store a high level description of a transaction's context to enable the transaction to forward recover at a participant site. Finally, because forward recovery is an option in IYV, IYV can be used to commit transactions with complex context without this option.

5. IYV optimizations and extensions

5.1. IYV presumed abort (IYV-PRA) optimization

Since the decision phase of IYV is exactly the same as in the PrN, we augmented IYV with the PrA optimization which makes the abort presumption more explicit than IYV. In IYV-PrA (see Fig. 5), the coordinator of a transaction needs only to force-write a commit log record and 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 log record at all and by discarding all the information about an aborted transaction from the protocol table.

IYV-PrA also reduces the number of coordination messages for aborted transactions. The participants do not have to acknowledge abort messages as they do not have to force-write abort decisions. If a participant fails before an abort decision is in stable storage, upon its recovery, it

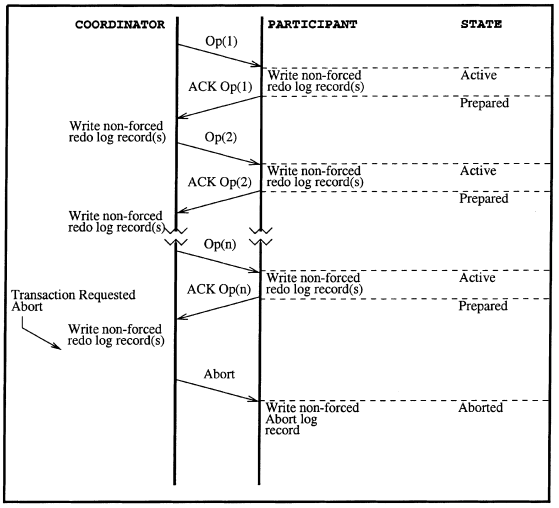


Fig. 5. The IYV-PrA protocol.

will undo the effects of such an (active) transaction and inquire the transaction's coordinator with a recovering message. Since the transaction has been aborted, its coordinator would not have any information pertaining to the transaction in its protocol table and therefore, it either replies with a repair message that does not contain any information about the transaction or an *inactive* message. An inactive message is sent as opposed to a repair message if the coordinator does not find any transaction in its protocol table that needs to be recovered at the participant. In IYV-PrA, this is possible since the coordinator may discard the information in its protocol regarding an aborted transaction which happens to be the last transaction executing at the participant before the participant has the chance to update its RCL. In either case, when the participant receives the reply message, it does not have to take any further actions since it has already obliterated the effects of the aborted transaction during the undo phase. The cost of a commit decision, however, remains the same as in the (basic) IYV.

5.2. IYV read-only optimization

The read-only optimization (see Section 2) can also be coupled with IYV in order to release locks at the participants earlier and to eliminate some forced log writes. A participant is termed as *read-only* if it has executed only read operations on behalf of a transaction. Otherwise, it is termed as an *update* participant. Since read operations do not affect the state of the database, a read-only participant does not have to be concerned with failures but only to ensure the serializability of the transaction. That is, as opposed to an update participant, when a read-only participant receives a commit or an abort decision, it does not have to perform any logging activities before releasing the transaction locks. In the absence of logging activities, read-only participants do not have to acknowledge that the decision has been forced.

In IYV read-only optimization (IYV-RO), when a commit primitive is received from a transaction, the coordinator determines which participants are read-only based on the acknowledgments it has received during the execution of the transaction. Specifically, once the last operation submitted to a participant is acknowledged and the participant did not send a redo record as part of any of its acknowledgment messages, the coordinator knows that the participant is read-only. If a participant is read-only, the coordinator immediately sends a *read-only* message to the participant without having to wait until a final decision is reached and is in the stable log. Once a participant receives a read-only message regarding a transaction, the participant releases the locks associated with the transaction and updates its RCL if necessary.

Since read-only participants in a transaction execution do not have to wait for a final decision to be reached and forced written into the stable log of the transaction's coordinator, read-only participants release the locks held by the transaction earlier than their update counterparts. Furthermore, as mentioned above, regardless of the final outcome of a transaction, a read-only participant does not have to acknowledge a read-only message. In the case that all the participants in the execution of a transaction are read-only and the coordinator has sent them read-only messages, the coordinator does not have to write decision and end log records. IYV's read-only optimization can be generalized and made applicable to the two-phase commit protocols [1,4,5].

5.3. IYV in multi-level transactions

The *multi-level transaction execution* (MLTE) model, the one specified by the standards and adopted by commercial database systems, is similar to the tree of processes model [5,26]. In this model, a participant is a process that is able to decompose a subtransaction further. Thus, a participant can initiate other participant processes at its site or different sites. Hence, the processes pertaining to a transaction can be represented by a multi-level execution tree where the coordinator process resides at the root of the tree. In this model, the interactions between the coordinator of the transaction and any process have to go through all the intermediate processes that have caused the creation of a process.

In the MLTE model, the behavior of the root coordinator and each leaf participant in the transaction execution tree, in both IYV and IYV-PrA, remains the same as in two-level transactions. The only difference is the behavior of *cascaded coordinators* (i.e., non-root and non-leaf participants) since an acknowledgment message of an operation represents the successful execution of the operation at the cascaded coordinator as well as all the descendants of the cascaded coordinator.

As in the case of the (two-level) IYV, in multi-level IYV, the root coordinators are responsible for maintaining the replicated redo logs. In multi-level IYV, all the redo log records that are generated during the execution of an operation are propagated to the root coordinator. Thus, when a cascaded coordinator receives acknowledgment messages from all its descendants that participated in the execution of an operation, it sends an acknowledgment to its direct ancestor in the transaction tree containing the redo log records generated across all participating sites as well as the read locks held at them during the execution of the operation. That is, an acknowledgment message from a cascaded coordinator represents the implicit vote of its subtree. When the root coordinator receives an acknowledgment message, it writes any redo records contained in the message in its log and any read locks in its PLT as it is the case of IYV in the two-level transaction execution model.

With respect to RCL, a cascaded coordinator updates its RCL to include the identity of the root coordinator only if it participates in the execution of the operation. If a cascaded coordinator does not participate in the execution of an operation, it simply sends the operation to the appropriate participant(s) without including the identity of the root in its RCL. Thus, when a transaction finishes its execution, all the replicated redo records and read locks are replicated at the root coordinator and, therefore, the coordinator knows all the participants (i.e., both leaf and cascaded coordinators). Similarly, each participant knows the identity of the root coordinator which is reflected in its RCL.

While the execution phase of a transaction is multi-level, the decision phase is not because the root coordinator of the transaction knows all the participants at the time the transaction finishes its execution. Therefore, the coordinator sends its decision directly to each participant without going through cascaded coordinators as in the two-level IYV. Similarly, each participant sends its acknowledgment of the decision directly to the root coordinator without going through the cascaded coordinators. [6]

In summary, in multi-level IYV, the behavior of the root coordinators remains the same as in IYV, cascaded coordinator is responsible about the coordination of acknowledgment messages of individual operations, while the behavior of participants remains the same as in the basic IYV. After a failure, the recovery of the participants (both cascaded coordinators and leaf participants) is as in IYV discussed in Section 3.2. A recovering participant waits for the responses of the coordinators included in its RCL, performs the undo and redo phases, and then resumes normal processing. Similarly, a recovering root coordinator gathers the acknowledgments of all participants in a

[6] This is similar to the flattening of the commit tree optimization [31]. Notice that the execution phase cannot be flattened as the decision phase because the acknowledgment message reflects the execution of a complete operation that is executed at several participants and the collective implicit vote.

transaction execution before forgetting the transaction.

IYV-PrA can be extended in a similar manner as IYV. The only difference is that a coordinator does not wait for the acknowledgment of any participant for the abort case.

6. Comparison with other atomic commit protocols

IYV combines the advantages of UV, EP and CL protocols, as we mentioned earlier. Here, we compare IYV with these protocols and in particular with CL which shares the same basic idea with IYV in order to reduce the number of log force-writes while eliminating the explicit voting phase of 2PC.

To avoid force writing the log records that are generated during the execution of each and every operation prior to acknowledging them, UV assumes that each site knows when it has executed the last operation on behalf of a transaction [35]. This means that the coordinator either submits to a site all the operations at the same time (which is a form of predeclaration) or indicates to the participant the last operation at the time that the operation is submitted. The former is possible in very special cases. The latter is only possible if each transaction has knowledge of the data distribution in the system and indicates to the coordinator the last operation to be executed at a participant.

In contrast to UV, IYV does not make any assumption about the structure of transactions and does not assume any knowledge of the data distribution by the transactions. Thus, IYV is more general compared to UV. In the special cases in which UV is applicable, IYV and UV would exhibit similar behavior during normal processing.

In EP, which is derived from PrC, the number of forced log writes pertaining to a transaction at its coordinator is equal to the number of participants that executed the transaction since EP requires the identities of the participants to be explicitly recorded at the coordinator's log in a forced initiation log record. This is because an initiation log record has to be force-written each time a new participant is about to execute an operation of the transaction. Furthermore, a participant force-writes a prepared log record before acknowledging a transaction's operation. In contrast, in IYV, a coordinator does not force-write any initiation record and a participant does not force-write any prepared log records.

To alleviate the drawback of the initiation records as well as the prepared log records of EP, CL uses *distributed write-ahead logging* DWAL. Using DWAL, only the coordinators maintain stable logs. In the case of CL, since a participant might inquire a coordinator about the latest forced log write (i.e., to ensure the DWAL), CL might become very costly and less efficient when compared with any of the 2PC variants. Consider the case when a number of long-living transactions execute at a participant without excessive main memory. In this case, the participant might request a transaction's coordinator (explicitly) to force-write its log more than once resulting in a great number of sequential messages. 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, which is a significant overhead.

Another significant difference between IYV and CL is the case of a coordinator's recovery after a failure. A coordinator in IYV can recover independently without communicating with any participant. In contrast, a recovering coordinator, in CL, has to communicate with all possible participants in the system in order to determine the set of "unknown transactions" so that it can abort them instead of presuming their commitment since CL is a descendant from PrC protocol [33,34]. An unknown transaction is a transaction that is executing at a participant but the coordinator has no recollection about the transaction since it does not find any log records pertaining to the transaction in its log during its recovery. Furthermore, a recovering participant in CL 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 up to the point prior to its failure and resume its normal processing without having to wait until all active transactions have been decided upon, allowing long-living transactions to forward recover and resume their execution. Aborted transactions in IYV are handled locally by a participant without any communication with the coordinators (i.e., the undo log records do not have to be requested from the coordinators).

Even though IYV requires that the redo log records generated during the execution of a transaction's operation be logged at both its coordinator as well as at the participant that executed the operation, 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 does 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.

It should be pointed out that not forcing commit records at the participants in IYV differs from the *group commit optimization* [14,18] in two ways, although both trade-off performance during normal processing for reduced independent recovery for participants. First, in IYV, there is no notion of a group or a timer that determines when a force should take place. Second, by not forcing individual commit records, group commit increases blocking after a participant failure whereas in IYV the blocking of a participant is the same irrespective of whether commit records are forced or not. This is because, in IYV, a participant cannot determine all transactions that were active at its site prior to a failure or their final status without contacting all coordinators in its RCL.

7. Analytical evaluation

In this section, we evaluate the performance of IYV and compare its performance with the performance of 2PC, PrA, PrC, EP and CL. The evaluation is based on the log, message and time complexities. In our evaluation, we use best (ideal) and worst case scenarios, as in [33,34], to highlight the performance differences among the various ACPs. Also, 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 cost of the protocols in both scenarios are evaluated during normal processing and in absence of failures.

Fig. 6 graphically illustrates the sequence of coordination messages and forced log writes involved in 2PC and IYV to reach a decision point and to release the resources held at the participants for the commit as well as the abort case. The figure shows how we will evaluate the performance of the ACPs that we listed above, considering the sequential effects of coordination messages and forced log writes. In our evaluation, we will assume that AAF is always set by the participant sites since we do not evaluate the cost of recovery after a site failure.

Tables 1 and 2 compare the different protocols under the worst case scenario. It should be pointed out that this "worst" case scenario is very close to the expected average behavior of transactions and the distributed environment. This is because the assumptions that we make in this case are more realistic than the assumptions that we make in 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 the following:

- A transaction has more than one write operation at each participant it accesses (i.e., $d > n$).
- Transactions execute sequentially (e.g., an operation is submitted by a transaction only when the previous operation has been executed and acknowledged).
- The participants are not known at the beginning of transactions.

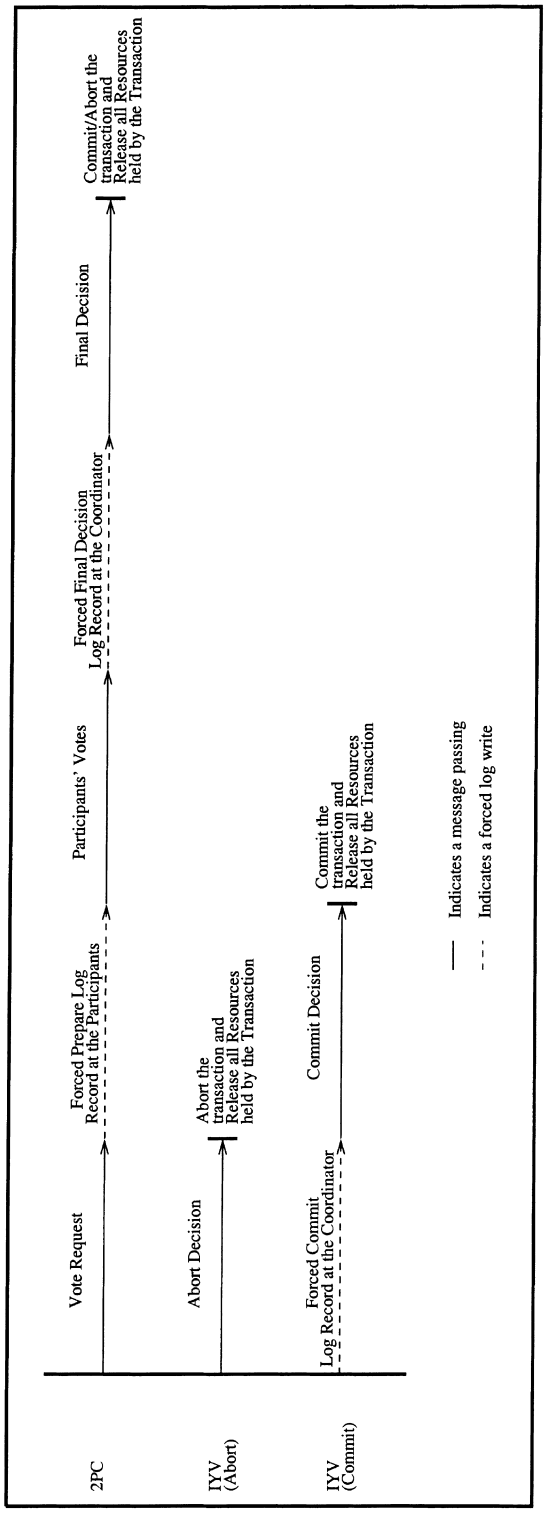


Fig. 6. The sequence of coordination messages and forced log writes required during normal processing.

Table 1
The cost of the protocols to *commit* a transaction assuming the worst case scenario

	2PC	PrC	PrA	EP	CL	IYV	IYV-PrA
Log force delays	2	3	2	$d+n+1$	1	1	1
Total log force writes	$2n+1$	$n+2$	$2n+1$	$d+n+1$	1	1	1
DWAL message delays	0	0	0	0	$2d$	0	0
Message delays (commit)	2	2	2	0	$2d$	0	0
Message delays (locks)	3	3	3	1	$2d+1$	1	1
Total messages	$4n$	$3n$	$4n$	n	$2d+n$	$2n$	$2n$
Total messages with piggybacking	$3n$	$3n$	$3n$	n	$2d+n$	n	n

Table 2
The cost of the protocols to *abort* a transaction assuming the worst case scenario

	2PC	PrC	PrA	EP	CL	IYV	IYV-PrA
Log force delays	2	2	1	$d+n$	0	1	0
Total log force writes	$2n+1$	$2n+1$	n	$d+2n$	0	1	0
DWAL message delays	0	0	0	0	$4d$	0	0
Message delays (abort)	2	2	2	0	$2d$	0	0
Message delays (locks)	3	3	3	1	$4d+1$	1	1
Total messages	$4n$	$4n$	$3n$	$2n$	$4d+n$	$2n$	n
Total messages with piggybacking	$3n$	$3n$	$3n$	n	$4d+n$	n	n

- The participants in a transaction execution do not have excessive main memories and each operation generates a single log record. This means that each and every 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 we do not include the number of forced log writes that are due to the operations and which are the same in all the protocols except for EP where the log records have to be force-written all the time. In CL, on the other hand, operations add extra overhead because each force-write is explicitly associated with two messages due to the DWAL.

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. 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 the 2PC protocol is two 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, as shown in Fig. 6. Also, "Message delays (commit)" and "Message delays (locks)" are 2 and 3 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. In the row labeled "Total message with piggybacking", we apply *piggybacking* of the acknowledgments of the decision messages, which is a special case of the lazy commit optimization [19], to eliminate the final round of messages.

It is clear from Tables 1 and 2, that IYV, IYV-PrA and CL outperform all other 2PC variants with respect to 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 log force-write whereas for the abort case neither IYV-PrA nor CL force-write

any log records. In this respect, EP is the most expensive of all protocols.

CL becomes more expensive than IYV when message delays and total number of messages are considered. Due to DWAL, CL requires two explicit sequential 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 the recovery scheme of CL, ARIES, is another operation that has to be executed and logged. Since CL uses DWAL, undoing an operation requires two more explicit messages to be exchanged between the coordinator and the participant in the worst case scenario. Note that because of its dependency on the number of data operations, CL can potentially involve more messages to commit or abort a transaction than any of the 2PC variants in the case of long transactions. On the other hand, with respect to messages, IYV and EP perform better than the other protocols. For the commit case, EP incurs the least number of total messages. This situation changes when piggybacking is considered where both EP and IYV incurs the same message complexities.

Piggybacking the acknowledgments of the decision messages can only 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 acknowledgment is sent immediately because it contains the undo log records of the aborted transaction.

Tables 3 and 4 compare the number of messages and forced log writes that are needed for the commit and abort cases, respectively, for the different protocols based on the ideal case scenario. In this ideal scenario, we make the following assumptions:

- A transaction performs at most one write operation per each site it accesses (i.e., $n = d$).
- The operations of a transaction execute in parallel.
- The participants are known at the beginning of transactions.
- The participants have excessive main memory.

Table 3
The cost of the protocols to *commit* a transaction assuming the ideal case scenario

	2PC	PrC	PrA	EP	CL	IYV	IYV-PrA
Log force delays	2	3	2	3	1	1	1
Total log force writes	$2n+1$	$n+2$	$2n+1$	$n+2$	1	1	1
Message delays (commit)	2	2	2	0	0	0	0
Message delays (locks)	3	3	3	1	1	1	1
Total messages	$4n$	$3n$	$4n$	n	n	$2n$	$2n$
Total messages with piggybacking	$3n$	$3n$	$3n$	n	n	n	n

Table 4
The cost of the protocols to *abort* a transaction assuming the ideal case scenario

	2PC	PrC	PrA	EP	CL	IYV	IYV-PrA
Log force delays	2	2	1	2	0	1	0
Total log force writes	$2n+1$	$2n+1$	n	$2n+1$	0	1	0
Message delays (abort)	2	2	2	0	0	0	0
Message delays (locks)	3	3	3	1	1	1	1
Total messages	$4n$	$4n$	$3n$	$2n$	$2n$	$2n$	n
Total messages with piggybacking	$3n$	$3n$	$3n$	n	$2n$	n	n

Table 5
The cost of the different protocols for *read-only* transactions assuming ideal case scenario

	PrC	PrA	EP	CL	IYV	IYV-PrA
Log force delays	1	0	1	1	0	0
Total log force writes	1	0	1	1	0	0
Message delays (decision)	2	2	0	0	0	0
Message delays (locks)	1	1	1	1	1	1
Total messages	$2n$	$2n$	n	n	n	n

Table 6
The cost of the different protocols for *read-only* transactions assuming worst case scenario

	PrC	PrA	EP	CL	IYV	IYV-PrA
Log force delays	1	0	n	1	0	0
Total log force writes	1	0	n	1	0	0
Message delays (decision)	2	2	0	0	0	0
Message delays (locks)	1	1	1	1	1	1
Total messages	$2n$	$2n$	n	n	n	n

In the ideal case, CL and IYV have the same cost as far as the number of sequential force log writes and messages are concerned for the commit case, dominating the other ACPs with CL dominating IYV and IYV-PrA with respect to the total number of messages. When piggybacking is considered, CL, IYV and IYV-PrA have exactly the same cost. For the abort case, CL dominates IYV with respect to the number of sequential force-writes and the total number of forced log writes while they have the same cost considering the number of sequential and total number of messages. On the other hand, IYV-PrA is better than CL by n in the total message count.

Comparing the two scenarios, we note that the cost associated with EP is highly dependent on the number of operations submitted by the transactions while CL is also dependent on the behavior of the system (e.g., the propagation latency of the communication network, the log buffer size, and the main memory size of the participants, etc.). This makes EP and CL completely inefficient in distributed database systems with long-living transactions where a transaction typically executes a large number of operations at a small number of sites (i.e., $d \gg n$), a situation common in advanced distributed database applications. Ruling out EP and CL, it is clear from our comparison's tables that IYV and its PrA optimization promise the minimum cost among the other ACPs.

Tables 5 and 6 show the cost of the different ACPs for read-only transactions assuming a commit final outcome. Table 5 considers the best case scenario while Table 6 considers the worst case scenario. The standard read-only optimization is used with PrC and PrA while the IYV-RO optimization is used with IYV and IYV-PrA. All the protocols in the two tables have the same cost as far as the number of sequential messages to release all the locks held by read-only transactions is concerned (i.e., only a single message). However, IYV and IYV-PrA dominate CL, EP and PrC as far as the number of sequential forced log delays to reach the commit point as well as the total number of forced log writes in both scenarios. As far as the total number of messages and message delays to reach a commit decision is concerned, EP, CL, IYV, and IYV-PrA have the same cost which dominates the cost associated with PrC and PrA.

8. Conclusions

In this paper, we presented IYV, a new one-phase atomic commit protocol for future gigabit-

networked distributed databases. IYV is targeted for environments with fixed number of high performance coordinators or application servers. IYV exploits the semantics of (1) database management mechanisms and (2) gigabit-networks to enhance performance over the other well-known atomic commit protocols [1,2]. Specifically, it exploits the S-2PL and the huge capacity of the network. IYV reduces the cost of commit processing at the cost of independent recovery. However, IYV compensates for the lack of independent recovery by providing the option of forward recovery where partially executed transactions at a failed participant that are still active in the system can resume their execution after the participant has recovered.

To highlight the performance enhancement of our commit protocol, we compared its performance to the performance of other known protocols based on the traditional analytical method. This method is based on evaluating the performance of the different protocols under worst and ideal case scenarios. Our evaluation reveals that the performance of our proposed protocol is very promising, a fact that has also been shown by our initial simulation results [1].

Even though IYV seems to be a very promising protocol with respect to performance, it is not applicable in resource-constrained systems (e.g., low bandwidth networks, small main memory or high cost disk access systems) or in client–server systems where the interactions between sites are not structured along the lines of remote procedure calls. Also, as any other one-phase protocol, IYV is not applicable in systems that require atomic commit protocols with explicit voting phase to perform some form of validation at commit time. This includes systems that use optimistic concurrency control protocols and systems that support validation of deferred consistency constraints.

Finally, IYV was designed for relatively reliable systems in mind. However, in the case of less reliable sites, unlike in [32], its blocking aspects can be reduced by combining IYV with the delegation of commitment technique, found in *open commit protocols* [28,29], and a novel timestamp synchronization mechanism while retaining its performance advantages over the other atomic commit protocols [3].

A clear conclusion of our analysis is that there is no single atomic commit protocol that is best for any system and for all transaction types. Furthermore, this points to the direction of integrated atomic commit protocols where for example IYV can be integrated with presumed abort protocol. This in fact constitutes the line of our current investigation along with the performance evaluation of the different atomic commit protocols using simulation.

References

[1] Y.J. Al-Houmaily, Commit processing in distributed database systems and in heterogeneous multidatabase systems, Ph.D. thesis, Department of Electrical Engineering, University of Pittsburgh, Pittsburgh, Pennsylvania, April 1997.

[2] Y.J. Al-Houmaily, P.K. Chrysanthis, Two-phase commit in gigabit-networked distributed databases, in: Proceedings of the Eighth ISCA International Conference on Parallel and Distributed Computing Systems, 1995, pp. 554–560.

[3] Y.J. Al-Houmaily, P.K. Chrysanthis, The implicit yes-vote commit protocol with delegation of commitment, in: Proceedings of the Ninth International Conference on Parallel and Distributed Computing Systems, 1996, pp. 804–810.

[4] Y.J. Al-Houmaily, P.K. Chrysanthis, S.P. Levitan, Enhancing the performance of presumed commit protocol, in: Proceedings of the 12th ACM Annual Symposium on Applied Computing, 1997, pp. 131–133.

[5] Y.J. Al-Houmaily, P.K. Chrysanthis, S.P. Levitan, An argument in favor of the presumed commit protocol, in: Proceedings of the 13th International Conference on Data Engineering, 1997, pp. 255–265.

[6] ANSI X3.135-1992, American National Standard for Information Systems – Database Language – SQL, November 1992.

[7] S. Banerjee, P.K. Chrysanthis, Data sharing and recovery in gigabit-networked databases, in: Proceedings of the Fourth International Conference on Computer Communications and Networks, 1995.

[8] S. Banerjee, V. Li, C. Wang, Distributed database systems in high-speed wide-area networks, IEEE Journal on Selected Areas in Communications 11 (4) (1993) 617–630.

[9] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil, A critique of ANSI SQL isolation levels, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1995, pp. 1–10.

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

[11] E. Braginski, The X/Open DTP effort, in: Proceedings of the Fourth International Workshop on High Performance Transaction Systems, CA, 1991.
[12] A. Citron, LU 6.2 Directions, in: Proceedings of the Fourth International Workshop on High Performance Transaction Systems, CA, 1991.
[13] D. DeWitt et al., The gamma database machine project, IEEE Transactions on Knowledge and Data Engineering 2 (1) (1990) 44–69.
[14] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, D. Wood, Implementation techniques for main memory database systems, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1984, pp. 1–8.
[15] E.N. Elnozahy, W. Zwaenepoel, Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit, IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing 41 (5) (1992) 526–531.
[16] K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger, The notion of consistency and predicate locks in a database system, Communications of the ACM 19 (11) (1976) 624–633.
[17] J. Gray, Notes on data base operating systems, in: R. Bayer, R.M. Graham, G. Seegmuller (Eds.), Operating Systems: An Advanced Course, Lecture Notes in Computer Science, vol. 60, Springer, Berlin, 1978, pp. 393–481.
[18] D. Gawlick, D. Kinkade, Varieties of concurrency control in IMS/VS fast path, IEEE Database Engineering 8 (2) (1985).
[19] J.N. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, Los Altos, CA, 1993.
[20] T. Haerder, A. Reuter, Principles of transaction-oriented database recovery, ACM Computing Surveys 15 (4) (1983) 287–317.
[21] L. Kleinrock, The latency/bandwidth tradeoff in gigabit networks, IEEE Communications Magazine 30 (4) (1992) 36–40.
[22] B.W. Lampson, Atomic transactions, in: B.W. Lampson (Ed.), Distributed Systems: Architecture and Implementation – An Advanced Course, Lecture Notes in Computer Science, vol. 105, Springer, 1981, pp. 246–265.
[23] B. Lampson, D. Lomet, A new presumed commit optimization for two phase commit, in: Proceedings of the 19th VLDB Conference, 1993, pp. 630–640.
[24] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz, ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging, ACM Transactions on Database Systems 17 (1) (1992) 94–162.
[25] C. Mohan, B. Lindsay, Efficient commit protocols for the tree of processes model of distributed transactions, in: Proceedings of the Second ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing, 1983.
[26] C. Mohan, B. Lindsay, R. Obermarck, Transaction management in the R^* distributed data base management system, ACM Transactions on Database Systems 11 (4) (1986).
[27] J.E.B Moss, Nested Transactions: An Approach to Reliable Computing, MIT Press, Boston, 1985.
[28] K. Rothermel, S. Pappe, Open commit protocols for the tree of processes model, in: Proceedings of the 10th International Conference on Distributed Computing Systems, 1990, pp. 236–244.
[29] K. Rothermel, S. Pappe, Open commit protocols tolerating commission failures, ACM Transactions on Database Systems 18 (2) (1993) 289–332.
[30] G. Samaras, K. Britton, A. Citron, C. Mohan, Two-phase commit optimizations and tradeoffs in the commercial environment, in: Proceedings of the Ninth International Conference on Data Engineering, 1993, pp. 520–529.
[31] G. Samaras, K. Britton, A. Citron, C. Mohan, Two-phase commit optimizations in a commercial distributed environment, Distributed and Parallel Databases 3 (4) (1995) 325–360.
[32] D. Skeen, Non-blocking commit protocols, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1982, pp. 133–147.
[33] J. Stamos, F. Cristian, A low-cost atomic commit protocol, in: Proceedings of the Ninth Symposium on Reliable Distributed Systems, 1990, pp. 66–75.
[34] J. Stamos, F. Cristian, Coordinator log transaction execution protocol, Distributed and Parallel Databases 1 (1993) 383–408.
[35] M. Stonebraker, Concurrency control and consistency of multiple copies of data in distributed INGRES, IEEE Transactions on Software Engineering 5 (3) (1979) 188–194.
[36] F. Upton IV, OSI distributed transaction processing, an overview, in: Proceedings of the Fourth International Workshop on High Performance Transaction Systems, CA, 1991.

Yousef J. Al-Houmaily received a B.S. in computer engineering from King Saud University, Riyadh, Saudi Arabia in 1986, a M.S. in computer science from George Washington University, Washington, D.C. in 1990, and a Ph.D. in computer engineering from the University of Pittsburgh, Pennsylvania in 1997. Currently, Dr. Al-Houmaily is an Assistant Professor and the director of Computer Programs Department at the Institute of Public Administration, Riyadh, Saudi Arabia. He served on a number of program committees including MobiDE'99 and IRI'99. His current research interests are in the areas of transaction processing and management in distributed and mobile database environments. Dr. Al-Houmaily is a member of ACM, ISCA and the Saudi Computer Society (SCS).

Panos K. Chrysanthis is currently a tenured Associate Professor of Computer Science at the University of Pittsburgh. He received the B.S. degree in Physics with concentration in Computer Science from the University of Athens, Greece, in 1982. He earned the M.S. and Ph.D. degrees in Computer and Information Sciences from the University of Massachusetts at Amherst, in 1986 and 1991 respectively, and joined the University of Pittsburgh in 1991 as an Assistant Professor. His research interests lie within the areas of database systems, distributed and mobile computing, operating system and real-time systems. In 1995, he was a recipient of the National Science Foundation CAREER Award for his investigation on the management of data for mobile and wireless computing. Dr. Chrysanthis has chaired and served on Program Committees of several Database and Distributed Computing Conferences. He also served as guest editor and reviewer for several journals. He is a member of the ACM (Sigmod, Sigops), the IEEE Computer Society, Sigma Xi the Scientific Research Society, and the USENIX Association.