Resilient Concurrency Control in Distributed Database Systems

Bharat Bhargava, Member IEEE
Purdue University, West Lafayette

Key Words—**Concurrency control, Distributed database, Site failure, Network partitioning, Lost message, Conflict graph.**

Reader Aids—
Purpose: Widen state-of-the-art
Special math needed for explanations: Discrete math, graph theory
Special math needed to use results: Same
Results useful to: Software engineers, researchers

Abstract—**This paper presents the resiliency features of the optimistic approach to concurrency control and demonstrates how it lends itself to a design of a reliable distributed database system. The validation of concurrency control, integrity control, and atomicity control has been integrated. This integration provides a high degree of concurrency and continuity of operations in spite of failures of transactions, processors, and communication system.**

1. INTRODUCTION

To provide continuity of operations in automated systems, we need to investigate principles that can provide robustness. Many systems such as used in space program, air traffic control, nuclear plant monitors, and ballistic missile defense demand nonstop operation. Recently, research efforts have focused on the design and implementation of distributed systems in such applications [1].

A distributed system consists of a set of computers located in different sites connected by a communication network. Different programs can run on each of these computers and the programs can access local or remote resources such as databases. The programs can be viewed as transactions which consist of a sequence of atomic operations. The resources can be partially replicated or partitioned.

Though it has become feasible for computers to communicate at high speed with each other and share their resources, a failure can occur in a distributed system. Some of these causes also exist in a centralized environment but for completeness, we present them as follows.

a. *Incorrect input data.* There are two sources of incorrect data: accidents, such as mistyping of input by the user or incorrect output from sensory devices or instruments (such as terminals). If incorrect data are input into the system, they can easily contaminate the database. The *external data integrity control (XDIC)* subsystem can provide a barrier against incorrect input data.

b. *Incorrect transaction.* Since transactions are executions of programs written by users and the programs can neither be fully proven correct or tested for all possible cases, incorrect transactions are a reality. The *transaction correctness control (PCC)* subsystem can ensure the correct execution of a transaction.

c. *Incomplete transaction execution.* It might not be possible to complete the execution of a transaction due to many reasons. For example, a hardware failure, deadlock problem, or violation of database security can cause the system to abort a transaction in the middle of its execution. The *transaction atomicity control (PAC)* subsystem can ensure that either none or all updates of a transaction are executed.

d. *Incorrect concurrency control.* Several transactions can run in parallel and not maintain consistency. For example, they may read incorrect data or write in the database in an incorrect order. The *concurrency control (CC)* subsystem can provide consistency via enforcement of serializability of concurrent transactions.

e. *Site failure.* A particular processor can cease operations due to software or hardware problems. The contents of the memory of the system can be lost or even contents of secondary storage can be affected. A site failure could be local to the transaction or it could be remote. A local site failure will halt the transaction processing. A remote site can affect the completion of the transaction because a data value to be updated by the transaction resides on that site.

f. *Communication system failure.* If the communication links between two sites are broken, they no longer communicate. In addition due to malfunction of the communication facility, messages can be lost or delivered in wrong order. Maintenance of database consistency in spite of site crashes and system partitions can be the task of the *site crash/partition treatment (SCPT)* subsystem.

Included in the above list is one more subsystem, called *internal data integrity control (IDIC)* that can check integrity of database before its use by a transaction, and/or check periodically or on demand the integrity of the current database state. Complete discussion on the design of a reliability control system are in [2-3].

This paper presents a framework that can help combine the functions of each of these subsystems. This framework is based on the optimistic approach that requires transaction validation as compared to locking as its basis for concurrency control. Some components of this approach have already been discussed [4-6].

2. BASIC TERMINOLOGY AND CONCEPTS

The distributed database is modeled by a set of *logical database entities* which can have one or more *physical copies* at different sites.

A distributed database is *consistent* if it satisfies some predefined assertions on the data values. For a replicated distributed database, the physical copies of the same database entity on different sites must be identical.

The user operations on a distributed database consist of a sequence of atomic operations.

Definition 2-1. An *atomic operation* is represented by $\sigma_i = A_i^j[x]$, where i is a unique identification for a transaction, j is a unique identification for a site, A is either R or W representing read or write operation, and x is one or more logical database entities.

Definition 2-2. Atomic operations are grouped into logical units called *transactions* that preserve the database consistency if executed alone.

Definition 2-3. Two atomic operations σ_i, σ_j *conflict* if: 1) they belong to different transactions; 2) both access the same database entity at the same site; 3) at least one of them is a write operation.

Definition 2-4. The concurrent activities of a distributed database system can be modeled as a sequence of all atomic operations called the *history* of the system. The history is represented by a quadruple $h = <D, T, \Sigma, \pi>$, where D is a distributed database, T is the transaction set, Σ is the atomic operation set, and π is a *permutation function* which gives the permutation indices for atomic operations σ in h ($\sigma \in \Sigma$).

For example, if a history h is the following sequence:

$$\alpha\beta\gamma \ldots \omega$$

then $\pi(\alpha) = 1, \pi(\beta) = 2, \ldots, \pi(\omega) = |\Sigma|$.

An atomic operation in a history can be freely rearranged in the history as long as the order of conflicting accesses is preserved.

A site projection represents the operations that are performed on a site.

Definition 2-5. A site projection g^j is obtainable from the site projection h^j through *conflict preserving exchange* (cp-exchange) if and only if there exist σ_1 and $\sigma_2 \in \Sigma^j$ such that σ_1 and σ_2 do not conflict, and the site projections are of the form:

$$h^j\colon \theta_1\sigma_1\sigma_2\theta_2; \quad g^j\colon \theta_1\sigma_2\sigma_1\theta_2$$

where θ_1 and θ_2 are strings of atomic operations. The only difference between h^j and g^j is that the order of σ_1 and σ_2 is reversed.

Since only adjacent non-conflicting operations are cp-exchanged, cp-exchange necessarily generates equivalent histories. Let $h^j \sim g^j$ mean that g^j is derived from h^j through a single cp-exchange, and let $\sim^*$ be the transitive closure of cp-exchange operator.

Definition 2-6. Two histories are *equivalent* (indistinguishable) if they transform a given initial state to the same final database state. The notation $\equiv$ denotes the equivalence relation between histories.

Proposition 2-1. If $h^i \sim^* g^j$ for all $j \in N$, then $h \equiv g$.

Proof: The proof is in [5].

Definition 2-7. A *serial history* is one in which each transaction runs to completion before the next one starts.

Definition 2-8. A history h is *serializable* if and only if there exists a serial history g such that $h^j \equiv g^j$ for every site j.

A history can be serialized according to the order of several events in the progress of a transaction. The instance when the transaction reads the first entity in the database is called α and the instance when the transaction writes on the first entity is called the event ω. The order of transactions in the serial history can also be established based on some other event in between α and ω, such as the instance when a transaction arrives for validation.

Definition 2-9. A concurrency control algorithm is correct if all its allowed histories are serializable.

The concurrency controller represents the software module of the system which manages the concurrent access of the database by transactions issued by users. The purpose of the concurrency control is to guarantee that the concurrent activity does not result in an inconsistent database state.

2.1. Design of Concurrency Control

There are two generic approaches that can be used to design concurrency control algorithms. The synchronization can be accomplished using three types of actions:

1. *Wait.* If two transactions conflict, the conflicting operations of the new transaction must wait until the operations of the other transaction are completed. The wait can be enforced by using locks on database entities or checking time-stamps [7] that can be assigned to transactions. One simple protocol that ensures serializability using locks is: *2-phase locking* (2PL) [8]. The protocol requires that in any transaction, all locks must precede all unlocks. The idea of waiting based on time-stamps has been used in [9]. The order of conflicting transactions in the serial history is established by the order of α event. This approach is *pessimistic* because some transactions could be blocked before knowing their write set and the conflict type.

2. *Validate after computation.* Transactions can proceed freely in the system and be validated for correct concurrency control just before their commitment. If two transactions conflict, some operations of a transaction are undone or rolled back or else one of the transactions is restarted. This approach is called *optimistic* because it is hoped that only a few transactions will be rolled back. Since recovery and rollback facilities (including logs representing histories) are already provided in most systems, a concurrency control algorithm can take advantage of them. Locking and pessimistic time-stamp ordering provide a low degree of concurrency [10].

2.2. Design of Atomicity Control

An atomicity control system requires that a transaction is either committed on all sites of a distributed system or is aborted on all sites. Several transaction atomicity control protocols have been discussed [11]. They extend the 2-phase commit protocols [12] for distributed systems where multiple failures of sites can occur, e.g., a 3-phase commit protocol. Multiple failures of a site might require several additional rounds of messages before deciding to commit or abort.

2.3. Design of Integrity Control

An integrity control system can ensure that no erroneous values (generated due to an incorrect transaction processing) are written in the database. Ref. [3] presents a short survey of integrity control algorithms and [13] discusses some performance issues. The integrity assertions are expressed as predicates on database values and they must be true for database to be in an integral state.

3. BASIC STEPS IN OPTIMISTIC CONCURRENCY CONTROL

The six steps in the execution and completion of a transaction are:

1. *Read.* The transaction reads the values for the required database entities into a private work space. This work space could be defined by a set of variables that correspond one to one with the entities. The set of variables representing the entities that have been read is called the read set, S(Ri).

2. *Compute.* The transaction executes by computing on the values of the variables in the private workspace and obtains a new set of values for its variables.

3. *Write in workspace.* The new values are written on local variables in the workspace. The variables correspond to the entities that will be updated by this transaction. The set of such entities is called the write set, S(Wi).

4. *Validate on the initiating site.* The site to whom the transaction is submitted is called its initiating site and all other sites in the distributed system are referred to as remote sites. The transaction validates against the set of concurrent transactions that have already validated on the initiating site. If the transaction is validated on the initiating site, it is put into the semi-commit state.

5. *Validate globally.* The transaction is sent to all other sites for validation. The validation process on each site is identical.

6. *Commit and write in database.* If the transaction is validated, it is put into the commit state. The local values of the variables in the workspace become the values of the corresponding entities in the database. If the transaction wants to read some entities and then wants to compute before reading additional entities, the read, compute, write local steps of the transaction can be interleaved. The validation step on each site is done in the critical section.

3.1. General Comments on Validation

Since the transactions are isolated from each other before they reach the validation stage, while a transaction *Ti* is still computing, another transaction *Tj* could commit or semi-commit and update the read set of *Ti*. Hence *Ti* must validate against all such *Tj*.

If only the write sets of the semi-committed or committed transactions are kept in the database [14], the concurrency provided by the algorithm will be the same as the locking algorithm. On the other hand if the read set and write sets of the committed transaction are available in the validation, the degree of concurrency provided by the algorithm will be higher [4-5]. A theorem [5] shows that in a 2-step transaction model (where all reads precede writes), a rollback in the optimistic approach corresponds to a deadlock when locking is used.

4. VALIDATION ON THE INITIATING SITE

First the integrity assertions are verified against the write set. If successful, the transaction validates against all committed and semi-committed transactions on the site for correct concurrency control. The validation step considers both the read set and the write set of all transactions. A conflict graph (CG) as defined below is constructed and is checked for cycles. If a cycle exists, the validating transaction is rejected else it is included in the set of semi-committed transactions and sent for global validation.

Definition 4-1. A *Conflict Graph* (CG) for a history $h = <D, T, \Sigma, \pi>$ is a digraph $<V, E>$ where V is the set of vertices representing T, the set of transactions; E is the set of edges, where $<i, j>$ is an edge if and only if there exist conflicting atomic operations σ_i, σ_j for which $\pi(\sigma_i) < \pi(\sigma_j)$.

Instead of the order of individual atomic operations, the edges can also be obtained by considering the order of α or ω events for each transaction.

Lemma 4-1. The CG of a serial history is acyclic.

Proof: The proof is in [5].

Based on the CG and the conflict (dependency) relation, a serializability class of histories for distributed database systems has been defined.

Definition 4.2. The class DCP contains all histories that are *distributed conflict-preserving.*

Definition 4-3. A history h is distributed conflict-preserving if and only if there exists a serial history g such that $h^j \sim * \; g^j$ for all $j \in N$.

For a history h in DCP, the site projection h^j is serializable since it can derive an equivalent serial site projection through a sequence of cp-exchanges, and all site projections are mutually consistent because the equivalent serial counterparts follow the same serialization order of transactions. The serializability of the histories in the class DCP can be easily tested by constructing and examining a graph that can be constructed for each DCP history.

Theorem 4-1. A history h is in DCP if and only if the CG of h is acyclic.

Proof: The proof is in [5]. The class DCP contains the locking class [10].

Corollary. For a history with an acyclic CG, the topological sort order of the vertices is the order of transactions in the equivalent serial history.

5. GLOBAL VALIDATION

When a transaction is validated on the initiating site, it is put into the semi-commit state on that site. Next it is sent to all the other sites for global validation. This step is combined with the first phase of the commit protocol of the atomicity control. A validation similar to that performed at the initiating site is performed at each remote site. If the transaction fails either the integrity tests or creates a cycle in the conflict graph, a reject message is sent to the initiating site. Otherwise, the transaction enters into the semi-commit state and an accept message is sent to the initiating site.

When the initiating site receives an acceptance message from all sites, it commits the transaction and sends this information to all sites. This step coincides with the second phase of the commit protocol of the atomicity control subsystem.

When a site receives a message that a transaction has been committed, it changes the state of the transaction from semi-commit to commit. For a rejected transaction, a global cycle will occur at least at one of the sites because all transactions are sent to all sites for validations. Exception cases are discussed in [4]. The transaction is either committed on all sites or rejected at all sites [4].

Figure 1 shows the states of the transaction and transitions.

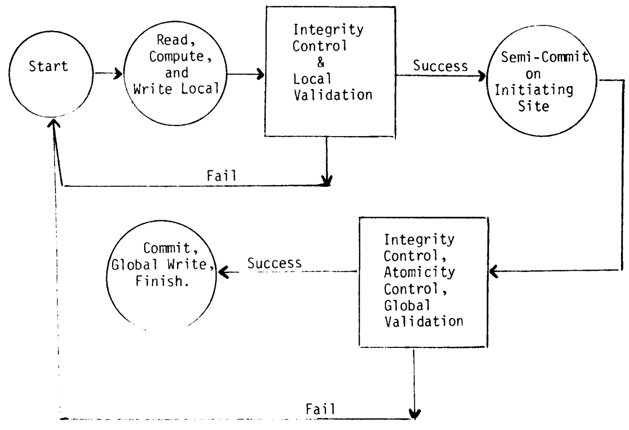
Start
Read, Compute, and Write Local
Integrity Control & Local Validation
Success
Semi-Commit on Initiating Site
Fail
Commit, Global Write, Finish.
Success
Integrity Control, Atomicity Control, Global Validation
Fail

Fig. 1. States of a Transaction.

Due to variable transmission delays, transactions from two different sites can reach other sites in different order. Since it is not desirable to block transactions from progressing, the following rule should be followed.

Rule: A transaction cannot be rejected at a site due to the validation of some newly arrived transaction after a site has agreed to semi-commit the transaction and sent this message to other sites. A semi-committed transaction can only be committed or rejected by the initiating site.

There is a difficulty with this rule. Assume that T_i semi-commits on site A and T_j semi-commits of site B. If T_i and T_j create a cycle in the conflict graph, then it is possible that that in global validation T_i is rejected on site B and T_j is rejected on site A. Thus both transactions are rejected.

To reduce the rejection of all transactions involved in a cycle, the system must set a criterion for rejection which is independent of the delays in the message transmission and is independent of when a transaction reached a site.

For example, a weight can be assigned to each transaction. This weight can be a function of the age (based of the value of the α) of a transaction, the cost of rolling back or restarting a transaction, the number of times a transaction has been rejected, etc. If both transactions have the same weight, both can be rejected or else the older one can be accepted. The implementation and performance issues are in [4].

6. THE OPTIMISTIC CONCURRENCY CONTROL ALGORITHM

A complete optimistic concurrency control algorithm is in the revised version of [4].

7. RESILIENCY FACILITIES DUE TO OPTIMISTIC CONCURRENCY CONTROL

The design of optimistic concurrency control as discussed in the previous sections provides several resiliency features. This section discusses the treatment of failures discussed in the introduction.

7.1. *Resiliency to Transaction Failure*

If partial updates of an uncommitted transaction are posted in the database and the transaction fails, the recovery procedures must be invoked. One recovery procedure using UNDO and REDO logs has been recommended [15]. The execution of UNDO actions and rollback of individual transactions can be expensive. It this system, the executing transactions are transparent to other transactions. A failure prompts the system to initialize the values of read set and write set variables and to inform the user. Thus transaction failures are handled fast and without any cascade effects.

7.2. *Resiliency to Site Failure*

The system can detect that either a local (initiating) site or a remote site for a transaction has failed. The failure can occur due to either a hardware or a software failure [16]. A failure can change the values of the contents in the memory or any other volatile storage to null. The site can also refuse to accept any new transactions or might not respond to messages from other sites. In our approach, the system changes the state of the transactions on the failed site as shown in table 1.

TABLE 1
Local Site Failure

Local Site Failure	System's Decision at Local Site
After Committing/Aborting. a local transaction	Do nothing (Assume: Message has been sent to remote sites)
After Semi-Committing a local transaction	Abort Transaction when local site recovers Send abort message to other sites
During Computing/Validating a local transaction	Abort transaction when local site recovers. Send abort message to other sites.

If a remote site fails, the local site can assume that the remote site has rejected the transaction. But an optimistic solution will be to hope that the transaction will validate on the remote site. The local site can either wait for the remote site to recover (causing blocking the transactions) or assume that the remote site will be validated successfully. Table 2 shows these cases.

The message sent to the failed remote site can be handled by the communication system or else spoolers can be used at each site to store messages sent by other sites. The failed site is required to update itself with all committed transactions from other sites before it may start to accept new transaction for validation. If the recovered site finds that its spooler also crashed during its failure, it requests the other sites to send information about transactions that were committed during its failure. Many interesting protocols for terminating a transaction have been proposed [11]. An approach to resiliency for site failures based on the time-stamp approach has been presented [17].

7.3. Resiliency to Communication System Failure

When one or more of the following types of problems occur, the communication system has failed.

a. *Network Partition.* A network partition occurs when the sites are separated into two or more partitions and the partitions are unable to communicate with each other. The sites in one partition can communicate with each other. A partition might consist of only one site.

b. *Lost Messages.* This type of failure occurs when the messages sent by one site do not reach the designated site.

c. *Message Order Messed Up.* This occurs when the messages are not delivered in the same order as they were sent. This can happen due to variable transmission delays.

7.3.1. Network Partition

When a network partitions, the system could decide to proceed as follows:

a. Do not process any transactions that want to update the database. Allow only retrievals from the database. This alternative is not very attractive except for special applications.

b. Process transactions in only one partition. This partition can be chosen based on some criterion such as containment of important nodes (that must not halt their operations) or containment of a majority of sites etc.. The updates of committed transactions will be sent to the other partitions when they join the active partition.

These two approaches tend to be pessimistic. This approach is based on the optimism that few transactions in different partitions will be involved in a cyclic conflict. There are several alternatives to allow transaction processing in all partitions.

1. *Alternative 1.* The system will allow transaction processing in all the partitions but transactions will remain in the semi-commit state and will be subject to validation at unavailable sites when the partitions merge. A conflict graph CGi is maintained for each partition i. The results of transactions in the semi-commit state can be made available to the user with proper warnings and can also be used by other transactions in the same partition. The merger of semi-committed transactions from several partitions can proceed as follows:

a. Combine CG1, CG2, ..., CGn on one of the sites. If no cycle exists, then commit all transactions. Minimize rollback if a cycle exists. Unfortunately this minimization is NP-complete because the problem reduces to "Feedback Vertex Problem" [18-19].

b. Assign a weight to each transaction in all partitions. Determine the CGi with maximum weight. Select transac-

TABLE 2
Remote Site Failure

Remote Site Failure	System's Decision at Local Site	System's Decision at Failed Site
Before receiving the remote transaction	a. Pessimistic: Wait for failed site to recover & validate	a. Pessimistic: Read messages on spooler. If spooler failed reject the transaction.
	b. Optimistic: Commit the transaction based on information from other sites & send message to failed site's spooler. (Assume failed site will semi-commit the transaction)	b. Optimistic: Semi-commit the transaction before processing new transactions. If spooler failed, request other sites to send transactions that were committed or aborted during its failure.
After semi-committing the remote transaction	Same as b. above	Same as b. above
After aborting the remote transaction	Abort the transaction	Do nothing

tions that do not create cycle from other CGj's ($j \neq i$) one at a time. There are two ways to deal with transactions that create a cycle:

b.1 Abort the transaction that creates the cycle.

b.2 Consider each transaction that creates a cycle one at a time and break the cycle by aborting the transaction that minimize transaction abort. This minimization can be done in computational complexity $\mathbf{O}(n^3)$ but will not be necessarily optimal if a global CG was formed. The minimization of abort can be based on several criterion. For example, reduce the number of transactions that are aborted, or reduce the loss of transaction processing cost. The processing cost can be a function of the parameters such as I/O cost or CPU cost. Some algorithms to reduce the number of nodes that must be removed from a graph to break a cycle are in [6]. The performance of a protocol based on these ideas is in [20].

2. *Alternative 2.* Until now, the granularity of database partition has been considered at the site level. A network partition can be defined based on the partitioning of the entities in the database. One copy of each entity in the database is designated as the primary copy (or the true copy). Each primary copy has a token.

The system allows a transaction to be processed and commit in a partition if the partition has a primary copy token for all the database entities needed. Thus transaction processing can continue in more than one partition. How the true copy tokens are maintained determines which transactions can be processed in which partition. A concurrency control algorithm that maintains tokens is in [21]. Since no transactions in different partitions can use the same entity, the database remains consistent.

7.3.1.1. Semantic Considerations

Semantics of the operations of the transactions can be used to decrease the amount of transaction abort and to increase processing in spite of failures.

7.3.1.1.1. Commutative Operations:

Two operations σ_i and σ_j are commutative if $\sigma_i \sigma_j \equiv \sigma_j \sigma_i$. For commutative operations value-based updates are not allowed. A simple example of a commutable operation is "Give Rupees 500 bonus to every employee". If the bonus was 5% of the salary, the operation is not commutable. Two adjacent read operations are also commutable.

The commutable operations of transactions can continue to execute in different partitions and after the merge of the partitions the values of the database entities can be set to the consistent state. This can be accomplished by remembering the value of the entity at the time of partition and applying the cumulative changes in the database values from each partition.

For example, if the value of a certain entity was x at the time of partition and at the time of partition merge, the values due to commutative actions of transactions in two different partitions are y and z. The correct value of the database entity at merge of the partitions can be set to $y + z - x$.

If the commutative property of the transaction can not be determined in advance, the system might be able to deteŗimine it at run time. For example, the system could recognize absolute increments and decrements in the database values. Every time the transaction issues an update, the system can check whether the transaction is value based.

The system can maintain a log of commutative and noncommutative operations. When the partitions merge, it can be checked if operations in different partitions were conflicting and at the same time noncommutative. If so the transactions will have to partially rollback up to the point where they started the noncommutative operations and will restart from such a point.

7.3.1.1.2. Compensating Operations:

An operation σ_c is compensating for the history $\sigma_2.\sigma_3....\sigma_n$ if and only if $\sigma_1.\sigma_2.\sigma_3 \ldots \sigma_n \sigma_c \equiv \sigma_1$. With the facility of an UNDO log and compensating operations, the system might be able to allow the transactions to commit in the different partitions.

When partitions merge, if there is a cycle in the global graph, the system can check if the cycle can be broken by removing some semi-committed transactions. If the cycle in the global conflict graph contains only committed transactions the system will be required to undo or compensate a committed transaction. If any committed transaction remains invisible to the environment, i.e., its updates were never seen by any transaction or user, then it could be selected to break the cycle. The other possibility is that the results of the transaction have been read by other transactions which might themselves have stayed invisible to the environment (transitive invisibility). In such related cases all such transactions can be aborted. If the results of a committed transaction have been made available to the environment, depending on the application, the system can issue a compensating transaction. If none of the transactions involved in the cycle can be compensated, the system should be programmed to pay a penalty (such as free airfare for overbooked passengers or not sufficient charge to a customer by a bank).

The solutions range from keeping all transactions in the semi-commit state to committing transactions in one partition or more. The commitment of all the transactions is possible with the necessity of a rollback or compensating transactions.

7.3.2. Lost Message

A transaction remains in a semi-commit state unless it has been validated at all other sites. Absence of the decision of a remote site will only delay the decision on the transaction. A time-out mechanism can be used at each site and if a transaction has been in the semi-commit state for a long time due to nonreceipt of the decision of a remote site, a request for the decision can be repeated. If the decision of a site is also sent to other sites, a site might be able to get it indirectly from other sites. A lost message can thus block only the processing of a transaction but can not affect system consistency.

7.3.3. Message Order Reversal

Some times the requests for global validation for two transactions arrive in a different order than their sending. A site will only send requests for validation for two transactions in the semi-commit state if they do not cause a cycle in the conflict graph at that site. If they do not create a cycle in the conflict graph on the remote site, the order of their arrival does not matter. If both of them create a cycle they both will be rejected and so once again their order does not matter.

ACKNOWLEDGMENT

This work has been partially supported by the U S Department of Transportation contract DTRS5680-C-0026. Thanks are due to Dr. Cecil Hua for the "DCP class" and Prof. Errol Lloyd for "complexity" in 7.3.1.

REFERENCES

[1] B. Bhargava (editor), *Concurrency and Reliability in Distributed Systems,* Van Nostrand and Reinhold 1985. In Press.

[2] B. Bhargava, "Reliability issues in distributed systems," *IEEE Trans. Software Eng.* vol SE-8, Guest Editorial, 1982 May.

[3] B. Bhargava, L. Lilien, "Reliability in distributed database systems," Research Report, University of Pittsburgh, 1982 Jul. Under revision with ACM Computing Surveys.

[4] B. Bhargava, "Performance evaluation of the optimistic concurrency control approach to distributed database systems and its comparison with locking," *IEEE Int. Conf. Distributed Computing Systems,* Miami, Florida, 1982 Oct 18-22.

[5] B. Bhargava, "Resiliency features of the optimistic concurrency control approach for distributed database systems," *Proc. IEEE Second Symp. Reliability in Distributed Software and Database Systems,* Pittsburgh, 1982 Jul 20-21.

[6] B. Bhargava, "Concurrency control and reliability in distributed database system," *Software Engineering Handbook,* Van Nostrand Reinhold 1984.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Comm. ACM,* vol 21, 1978 Jul, pp 558-564.

[8] Eswaran, Gray, Lorie, Traiger, "The notions of consistency and predicate locks in a database system," *Comm. ACM,* vol 19, 1976 Nov, pp 624-633.

[9] Rothnie, Bernstein, Fox, Goodman, Hammer, Landers, Reeve, Shipman, Wong, "Introduction to a system for distributed databases (SDD-1)," *ACM Trans. Database Systems,* vol 5, 1980 Mar, pp 1-17.

[10] C. Hua, B. Bhargava, "Classes of serializable histories and synchronization algorithms in distributed database systems," *Proc. IEEE Int. Conf. Distributed Computing Systems,* Miami, 1982 Oct 18-22.

[11] Dale Skeen, "A decentralized termination protocol," *Proc. Symp. Reliability in Distributed Software and Database Systems,* Pittsburgh, 1981 Jul, pp 21-22.

[12] J. N. Gray, "Notes on database operating systems," pp 393-481, *Operating Systems: An Advanced Course,* ed. Goos & Hartmanis, Springer-Verlag 1978. Lecture Notes in Computer Science 60, also as IBM Research Report RJ2188.

[13] L. Lilien, B. Bhargava, "A scheme for verification of database integrity," *IEEE Trans. Software Eng.,* 1984 Nov.

[14] H. T. Kung, J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Systems,* vol 6, 1981 Jun, pp 213-226.

[15] Jim Gray, "The transaction concept: Virtues and limitations," *Proc. VLDB Conf.,* Cannes, France, 1981 Sep 9-11.

[16] Randell, Lee, Treleaven, "Reliability issues in computing systems design," *ACM Computing Surveys,* vol 10, 1978 Jun.

[17] Bajaj, Hua, Bhargava, "Time-stamp resiliency to node failures in distributed systems," *Proc. Int. Computer Symp.,* Taiwan, 1982 Dec 18-20.

[18] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman and Co., San Francisco 1979.

[19] S. B. Davidson, "Evaluation of an optimistic protocol for partitioned distributed database system," Technical Report #299, Princeton Univ. 1982.

[20] B. Bhargava, "Performance evaluation of reliability control algorithms for distributed database systems," *J. Systems and Software,* vol 4, 1984 Jul.

[21] T. Minoura, and G. Wiederhold, "Resilient extended true-copy token scheme for a distributed database systems," *IEEE Trans. Software Eng.,* vol SE-8, 1982 May.

AUTHOR

Bharat Bhargava; Computer Science Department; Purdue University, West Lafayette, IN 47907, USA.

Prof. Bharat Bhargava teaches at Purdue University. He is pursuing research to identify principles that are necessary to increase reliability in concurrent transaction processing in distributed database systems. He has studied the performance of some of these principles as they apply to different applications, specifically, the enroute air traffic control system. Bhargava is one of the founders of the IEEE Computer Society's Symposium on Reliability in Distributed Software and Database Systems. He has served on program committees of the 1982 Distributed Computing System and 1984 Data Engineering conferences. He was a member of the CODASYL Systems committee from 1978-1982. Bhargava received his Ph D from Purdue University in 1974.

Manuscript TR82-74 received 1982 July 6; revised 1983 November 7.

★★★

Manuscripts Received.... For Information, write to the author at the address listed; do NOT write to the Editor

"Maximum likelihood estimation in the right-truncated Weibull distribution", Dallas R. Wingo ☐ 28 Riverside Ave, 8E ☐ Red Bank, NJ 07701 USA. (TR83-131)

"A new technique in minimal path and cutset evaluation", Ghauth Jasmon ☐ Dept. of Electrical Engineering ☐ University of Malaya ☐ Pantai Valley ☐ Kuala Lumpur, MALAYSIA. (TR83-132)

"Opitmal structure for some systems of independent components", James Brucker ☐ Dept. of Information Science ☐ Tokyo Institute of Technology ☐ O-okayama, Meguroku ☐ Tokyo 152 JAPAN. (TR83-133)

"A truncation methodology for evaluation of large fault trees", Dr. M. Modarres ☐ Dept. of Chemical & Nuclear Engineering ☐ University of Maryland ☐ College Park, MD 20742 USA. (TR83-134)