1988

A Generic Algorithm for Transaction Processing During Network Partitioning

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Shirley Browne

Report Number:
88-787

A GENERIC ALGORITHM FOR TRANSACTION PROCESSING DURING NETWORK PARTITIONING

Bharat Bhargava
Shirley Browne

CSD TR-787
June 1988

A Generic Algorithm for Transaction Processing During Network Partitioning

Bharat Bhargava and Shirley Browne
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

June, 1988

Abstract. This research presents an algorithm that allows transaction processing to proceed during site failures and network partitioning while ensuring the consistency of replicated data. Our algorithm can be used together with various voting schemes which provide varying degrees of data availability. Different voting schemes may be used simultaneously for different groups of data. Our algorithm contains as special cases: the site quorum method in which there is a single distinguished partition, and the virtual partition method in which a (possibly different) distinguished partition is determined for each logical data item. By grouping data items in various ways, our algorithm can be tuned to yield methods which lie between these two extremes.

1 Introduction

In a distributed database system (DDBS), data may be partially or fully replicated to improve the performance of the system and to increase availability. Performance is improved by enabling transactions to read local copies. Availability is enhanced during failures since replication increases the likelihood that at least one copy of a data item will be available. When failures occur, however, problems arise in maintaining the consistency of the replicated copies. Consistency requires that the concurrent execution of a sequence of transactions on the replicated database have the same effect as a serial execution of the same transactions on a single-copy database. Each transaction should also have a consistent view of the database state. These notions of correctness are formalized by the definitions of logical-serializability in [12] and of one-serializability in [3].

The algorithm for transaction processing described in this paper is fault-tolerant in that it can handle fail-stop types of failures of sites and communication links which may lead to network partitioning. Our algorithm is correct in that it ensures the logical-serializability of user transactions. We assume that the communication subsystem handles lost or delayed messages and link failures which do not lead to network partitioning by means of retransmission and rerouting, respectively. We assume that the DDBS runs a correct concurrency control algorithm which ensures conflict-serializable execution of transactions.

We use the notion of a distinguished partition to ensure consistency of the database state during network partitioning. Network partitioning occurs when site and/or link failures cause the network to be broken up into two or more connected subgraphs which cannot communicate with each other, each of which is called a partition. If site failures occur but the remaining sites can all communicate with each other, the operational sites compose a single partition. In any case, access to a particular domain of data items is allowed to proceed in at most one partition, which is called the distinguished partition (DP) for that domain.

We assume an algorithm is given that allows a site to determine whether or not its partition is the DP for a particular domain, based on local information and on information obtainable from the sites with which it can communicate. One possible algorithm is the lexicographic voting method described in [10]. Copies which have missed updates, either due to a site failure or network partitioning, are marked as unreadable when network reconfiguration causes these copies to again become part of a DP.

Each site maintains information concerning which other sites it can communicate with. This information is updated by means of special control transactions which are serialized with user transactions. Thus, a user transaction running in a partition has a consistent view of that partition since a reconfiguration must occur logically either before or after the user transaction.

The organization of the paper is as follows: In section 2, we discuss related research.

Section 3 describes the distributed system model, as well as the necessary terminology and notation. In section 4 we describe our algorithm, and in section 5 we give a proof of its correctness in terms of logical-serializability. Conclusions and ideas for future work are given in section 6.

2 Related Work

Much previous research has been focused on the topic of transaction processing in non-partitioned networks in which site failures may occur. Algorithms for managing replicated data in this case are given in [2] and [5]. The general approach is to consider the occurrence of failures and recoveries to coincide with the execution of the corresponding control transactions. For example, the failure of a copy or a site is considered to occur at the same time that the transaction that announces that failure is executed. Use of a correct concurrency control algorithm then yields an execution history which is logical-serializable, provided a read-one/write-all-available (ROWAA) strategy is used for processing user transactions. This approach is not directly applicable when network partitioning can occur, since allowing a ROWAA strategy to be used simultaneously in different partitions can lead to the inconsistency of replicated data.

A survey of transaction processing methods in partitioned networks is given in [6]. This survey discusses both pessimistic and optimistic strategies and both syntactic and semantic approaches. In this paper, we restrict ourselves to pessimistic syntactic methods – that is, we use logical-serializability as the correctness criterion and we require that global inconsistencies between partitions not be allowed to occur.

Static and dynamic voting methods for partition processing are discussed in [14,7] and [9,10,4]. These methods require a transaction to access some threshold number of votes for a data item, called a quorum, before executing an operation on that item. In a partitioned network, these thresholds guarantee that conflicting operations cannot occur in different partitions. This guarantee is not without a cost, however, since accessing a quorum makes read operations expensive. The virtual partition approach in [1] requires a quorum of copies of a particular data item to be present in a given partition for access to that data item to be allowed to occur. Although a transaction does not need to access a quorum before performing an operation, it needs to know which sites are considered to be in its current partition. This knowledge is provided by identifying changes in the configuration of the network with the control transactions that announce those changes. Correctness is ensured by relying on the underlying concurrency control method to produce a logical-serializable execution history. The virtual partition method has the disadvantage that whenever a control transaction is executed within some partition, a read quorum must be accessed for each data item which has a read quorum in that partition.

Quorum-based commit and termination protocols that attempt to maximize data avail-

ability by taking into account the voting strategy used are described in [8]. Theoretical results in [13] and [11] show that, in general, nonblocking commit protocols for the case of network partitioning are impossible.

We have extended the ideas presented in [5] for managing site failures to the case of network partitioning. We propose a generic method which can incorporate various pessimistic, syntactic strategies for partition processing in an efficient manner. This flexibility allows use of the strategy which is most suitable for a particular domain.

3 Model and Terminology

As in the model of the DDBS described in [5], the users' view of an object is called a *logical data item*, or *data item*, denoted X. A data item is stored as a set of *physical copies* or *copies*. The copy of X stored at site k is denoted x_k, and the fact that X has a copy at site k is denoted $x_k \epsilon X$. We assume that the information regarding where the copies of data item X are located is available at least at the resident sites of X. In general, we use upper-case for notation pertaining to logical data and lower case for physical data.

The set of logical data items is grouped into (not necessarily disjoint) subsets called *domains*. At one extreme all data items are members of a single domain, while at the other extreme, each data item is a member of a separate domain. Data items may be grouped into domains according to different criteria. For example, data items that are replicated at the same subset of sites may compose a domain. Alternately, data items that are frequently accessed within the same transaction may be grouped into the same domain.

Users manipulate the database via *transactions*. A transaction is a program that accesses the database by issuing *logical operations* $READ$ and $WRITE$ on logical data items. Each transaction has a *coordinating* or *home* site and possibly some other *participating* or *slave* sites. We assume that a transaction's home site is the one at which the transaction is initiated.

There are two major functional modules running at each site. The *transaction manager* (TM) supervises the execution of transactions that are initiated at its site and interprets logical operations into requests for *physical operations*. The *data manager* (DM) carries out the physical operations on the copies stored at the site.

Two physical operations *conflict* if at least one of them is a *write* and they both access the same physical copy. An execution history H containing the physical operations for a set T of transactions is *conflict-serializable* if there is a serial history H_s containing the same operations such that any two conflicting operations appear in the same order in both H and H_s. (A serial history is one in which operations from different transactions are not interleaved). The *conflict graph* (CG) corresponding to H is a directed graph $(T, \rightarrow)$ in which there is an edge $T_a \rightarrow T_b$ whenever T_a and T_b contain conflicting operations op_a and

op_b, respectively, and op_a precedes op_b in H. H is conflict-serializable if and only if its CG is acyclic [3].

We consider the topology of our network to be its virtual communication topology. That is, if two nodes can communicate with each other, then we consider the graph to have an edge between these two nodes. We assume that a failed attempt to communicate with another node will be reported to the requesting node by the communication subsystem. We do not assume that the cause of the failure is reported.

4 Algorithm

4.1 Partition Identifiers and Connection Vectors

Each site maintains a copy of the partition identifier (PID) and the connection vector (CV) for its partition. The PID consists of two parts – higher order bits which are increasing over time for any given site, and lower order bits assigned by the site which initiates the partition. Since these lower order bits are different for different sites, PID's are unique system-wide. CV and PID are considered data items of a special type, called control data (CD), and accesses to their copies are governed by the concurrency control algorithm. Three types of transactions are executed by the DDBS – user transactions, copier transactions, and control transactions. All user and copier transactions read local copies of CV and PID, but only control transactions may write control data items. We assume that PID and CV are fully replicated at all n sites. Following the convention described in section 3, the copies of PID and CV at site i are denoted by pid_i and cv_i, respectively. The values of pid_i and cv_i give the status of site i's partition as currently perceived by site i.

The only type of control transaction we will describe in detail is the RECONFIGURE transaction, which is used to determine a new partition. Other types of control transactions may be useful for efficiency reasons, but they are not necessary for the correctness of our algorithm. The use of other types of control transactions is discussed in subsection 4.3. A reconfiguration of the system is indicated whenever a change in the communication topology of the network is detected. A RECONFIGURE transaction which carries out this reconfiguration may be triggered either by a user transaction which is unable to complete or by the communication subsystem. The best method of triggering a RECONFIGURE transaction is a topic for further research.

A site i which executes a RECONFIGURE transaction requests PID's from all other sites. It determines a new PID by setting the higher order bits to be greater than the maximum of all those received from other sites and setting the lower order bits to its unique site ID. CV contains a 1 entry for each site with which site i is able to communicate and a 0 entry for all other sites. The RECONFIGURE transaction writes the PID and CV of

every member of the new partition.

4.2 Distinguished Partitions

Data items are grouped into domains as described in section 3. Each domain can be accessed in at most one partition, called the distinguished partition (DP) for that domain. In addition to setting a new PID and CV, the RECONFIGURE transaction determines for which domains the new partition is the DP. We assume some method is given for making this determination. Different methods may be used for different domains. Associated with each domain name is stored information that is needed by the method. For example, for the static quorum method, read and write quorum thresholds are stored with the domain name. A partition meets a quorum requirement for a domain if the number of sites containing a copy of some item in the domain is greater than or equal to the given quorum threshold. The new partition is the DP for the domain if it meets the requirements for both read and write quorums. As another example, for the dynamic voting method [9], an integer giving the cardinality of the set of sites having up-to-date copies of all data items in the domain is stored with the domain name. In this case, the new partition is the DP for the domain if it contains a majority of the sites that have up-to-date copies.

When a new partition is formed, there may be sites in the partition that are out-of-date with respect to a particular domain for which the new partition is the DP. If a site did not participate in the immediately previous DP for that domain, then the domain should be marked as unreadable at that site. These ideas have also been used in [4]. We explain how this marking is done in the next subsection.

4.3 Control Transactions

The RECONFIGURE transaction may be initiated whenever a change in the communication topology is detected. A RECONFIGURE transaction executes as follows: It first broadcasts read requests for CV and PID and the associated domain names and information to all sites. Let R denote the set of sites that reply. The transaction determines a new PID as described in subsection 4.1. Entries in CV are set to 1 for all sites in R and to 0 for all other sites. Using the prescribed methods, the transaction determines for which domains the new partition is the DP. For a given domain Π associated with the new partition, let PID_Π be the maximum of all PID's (returned by sites in R) which listed Π as an associated domain. For any site which returned a PID different from PID_Π, the domain Π should be considered unreadable. Finally, the RECONFIGURE transaction writes the new values for CV, PID, and the associated domain names and information to all sites in R, and marks outdated domain copies as unreadable. If any of the write requests cannot be completed due to concurrent failures, the RECONFIGURE transaction is aborted to avoid the possibility of more than one DP being formed for a given domain.

Although a site that is recovering from a failure could initiate a RECONFIGURE transaction, it may be more efficient to provide a RECOVERY transaction by means of which the site can join the current partition. A RECOVERY transaction executes at site i as follows: The transaction first issues read requests for CV and PID and the domain names associated with the partition. Since the local copies of CV and PID are considered unreadable, these read requests trigger a copier transaction which refreshes the local copies. The transaction then writes a 1 entry to $CV[i]$ at every site in the current partition, itself included. It marks local copies of data items in the domains associated with the partition as unreadable. If any of the write requests cannot be completed due to failures, the RECOVERY transaction is aborted.

Control transactions are executed concurrently with all other transactions, and they are governed by the same concurrency control algorithm and commit protocol.

4.4 User Transactions

Each user transaction reads the local copies of CV and PID before executing any other operations. The transaction is tagged with the value read for PID. The strategy used for processing read and write operations on objects within a given domain can depend on the DP method used for that domain. For example, if the quorum method is used, read and write quorum numbers may be specified for the partition which are consistent with the global read and write thresholds for determining the DP, as in [1]. If the DP is determined by means of the dynamic voting method, then a read-one/write-all strategy should be used.

Each request for reading or writing a physical copy at site k includes the PID of the transaction issuing the request. If this number does not agree with pid_k, the request is rejected. Note that this access to pid_k is considered to be a $read(pid_k)$ operation for purposes of concurrency control. If a write request is rejected or cannot be carried out because of a failure, the transaction is aborted. The transaction may or may not be aborted if a read request is rejected, since the read can be retried at another site.

4.5 Copier Transactions and Unreadable Copies

For a data copy marked as unreadable, a write operation on it removes the mark when the transaction which issued the write commits, while a request for reading it triggers a copier transaction that renovates the physical copy. The user transaction may either be blocked until the copier finishes or may read some other copy instead.

A copier transaction is responsible for refreshing a particular unreadable data copy of a data item X. It reads the local copies of CV and PID, locates a readable copy of X, uses the contents of the readable copy to renovate the local copy, and removes the unreadable mark. If version numbers are used, then it is necessary to actually copy the data only if

the version numbers are different. If the copier cannot find a readable copy in the current partition, the copier transaction is aborted.

Copier transactions may be initiated by a site as part of a recovery procedure or triggered on demand by read requests. They are executed concurrently with all other transactions and are governed by the same concurrency control algorithm and commit protocol.

5 Proof of Correctness

5.1 Correctness Concepts

In this subsection, we briefly define the fundamental concepts that are needed for the correctness proof presented in the next subsection. These concepts are based on the theory developed in [12].

An *execution history* H for a set of transactions $T = \{T_a, T_b, ...\}$ is a partially ordered set $(\sigma, <)$ containing all the physical operations interpreted from the logical operations of these transactions. Note that σ may contain read and write operations corresponding to control and copier transactions as well as to the transactions in T. An *augmented execution history* is a history with an initial transaction that writes to all data copies and a final transaction that reads from all data copies. To simplify our arguments, we consider only augmented execution histories.

A transaction T_b *reads-x_i-from* T_a, denoted $T_a \Rightarrow_{x_i} T_b$, in H if $w_a[x_i] \in \sigma$, $r_b[x_i] \in \sigma$, $w_a[x_i] < r_b[x_i]$, and there is no transaction T_c such that $w_c[x_i] \in \sigma$ and $w_a[x_i] < w_c[x_i] < r_b[x_i]$. A (non-copier) transaction T_b *READS-X-FROM* a (non-copier) transaction T_a, denoted $T_a \Rightarrow_X T_b$, in H if either there is a copy $x_i \in X$ such that $T_a \Rightarrow_{x_i} T_b$ (T_b *READS-X-FROM* T_a directly), or there are copies x_i, x_j and a copier transaction T_c such that $T_a \Rightarrow_{x_j} T_c$ and $T_c \Rightarrow_{x_i} T_b$ (T_b *READS-X-FROM* T_a indirectly via the copier transaction T_c, similarly defined for a sequence of copier transactions). A *logical serial* history for T is a totally ordered set containing the logical operations from T such that operations from different transactions are not interleaved. We consider only augmented logical serial histories that have an initial transaction that writes to all logical data items and a final transaction that reads from all data items. A logical serial history defines its $READ-FROM$ relations from the total order.

Two augmented execution histories for the same set of transactions are *logically equivalent* if they define the same $READ-FROM$ relations. A history H is *logical-serializable* if there exists a logical serial history H_s that is logically equivalent to H.

A *logical-serializability testing graph* (LSTG) of a history H is a directed graph $(T, \rightarrow)$ with the following properties:

(i) If $T_a \Rightarrow_X T_b$, then there exists an edge $T_a \rightarrow T_b$ (called a WR edge or a $READ-FROM$ edge);

(ii) There is an edge between any two non-copier transactions that write to copies of the same logical object X (called a WW edge, denoted $\rightarrow_X$);

(iii) If $T_a \Rightarrow_X T_b$ and $T_a \rightarrow_X T_c$, then there is an edge $T_b \rightarrow T_c$ (called an RW edge).

The main result of logical-serializability theory can be stated as follows:

Theorem 1. [12] A history H is logical-serializable if and only if H has an acyclic LSTG.

If we modify the definition of LSTG by replacing the word "edge" by "path", the notation $\rightarrow$ by $\rightarrow^*$, and $\rightarrow_X$ by $\rightarrow_X^*$, Theorem 1 remains correct.

For purposes of our proof of correctness, we consider two categories of data: user data (UD) and control data (CD). CD items include CV and PID, as well as the domain names and attributes. Note that with respect to CD, user transactions are read-only transactions. To simplify our arguments, the only type of control transaction we consider is the RECONFIGURE transaction. Our arguments can be extended to prove that the algorithm is still correct if other types of control transactions are also used.

5.2 Correctness Proof

We assume that the DDBS runs a correct concurrency control algorithm that ensures conflict-serializability with respect to $UD \cup CD$, but what we need to prove is logical-serializability with respect to UD. The conflict graph with respect to $UD \cup CD$ is insufficient to prove logical-serializability since it may not contain all the necessary write-order and read-before paths. Hence, we add edges to the conflict graph so as to obtain these paths while still preserving acyclicity of the resulting graph.

We assume that the algorithm provided for determining the DP for a particular domain guarantees that there is at most one DP at any given time and that the current DP has a nonempty intersection with the previous DP (i.e., at least one site in common). We say that a RECONFIGURE transaction that associates a domain Π with a new partition *activates* Π. The activations of a particular domain can be numbered consecutively, starting with 1.

We make use of the following two lemmas which follow from the fact that the execution history is conflict-serializable.

Lemma 1. If $T_a \Rightarrow_Y T_b$, $Y \epsilon UD \cup CD$, then there is a path in CG from T_a to T_b.

Proof. If T_b *READS-Y-FROM* T_a directly, then there is a copy y_i such that T_b *reads-y_i-from* T_a and hence an edge $T_a \rightarrow T_b$ in CG. If T_b *READS-Y-FROM* T_a indirectly via copier transactions, then there is a path $T_a \rightarrow^* T_b$ in CG.

Lemma 2. There is a path in CG between any two control transactions which activate a common domain.

Proof. (by induction)

Let T_c and T_d be RECONFIGURE transactions which both activate a common domain Π with corresponding activation numbers A_c and A_d, respectively, for Π. Assume without loss of generality that $A_c < A_d$. If $A_d = A_c + 1$, let site k be common between the two

activations. Then since T_c and T_d both write cv_k and pid_k, there is an edge $T_c \rightarrow T_d$ in CG. If $A_d > A_c+1$, let T_e be the RECONFIGURE transaction that forms the DP corresponding to $A_e = A_d - 1$. Again, there is an edge $T_e \rightarrow T_d$ in CG. By induction there is a path $T_c \rightarrow^* T_e$ in CG. Thus, there is a path $T_c \rightarrow^* T_d$ in CG.

Lemma 2 implies a total ordering on the control transactions that activate a particular domain.

Theorem 2. Based on the algorithm stated in section 4, the conflict graph (CG) with respect to $UD \cup CD$ can be augmented to form an acyclic graph G which contains an LSTG with respect to UD.

Proof. Form graph G by adding edges to CG as follows: For each domain Π and each control transaction T_c that activates Π, for any non-control transaction T_a such that $T_c \Rightarrow_{CV} T_a$, add an edge from T_a to the control transaction T_d (if any) that immediately follows T_c in the total order, if this edge does not already exist (the edge already exists if T_d writes $home(T_a)$). The resulting graph G is clearly still acyclic (since if there is a path in CG from T_d to T_a, then T_a does not *READ-CV-FROM* T_c).

For a given domain Π, we consider the subgraph G_Π of G consisting of user and copier transactions that access objects in Π and of control transactions that activate Π. The edges in G_Π define a partial order on these transactions in general and a total order on the control transactions (by Lemma 2). The non-control transactions which fall between two consecutive control transactions execute during a particular activation of Π. We claim that logically conflicting operations on a data item X in Π are totally ordered by edges in G. Let $op_1[X]$ and $op_2[X]$ be two such logically conflicting operations. If op_1 and op_2 occur during the same activation, then we assume op_1 and op_2 physically conflict and thus are ordered in CG according to the concurrency control algorithm. If op_1 and op_2 occur during different activations, then there is a path in G between the corresponding transactions of which they are a part, and we take the direction of this path to be the ordering. We assume that the method given for determining the DP with respect to Π, together with the strategy given for processing read and write operations on objects in Π, guarantees that if a transaction T_b *READS-X*, then T_b *READS-X-FROM* T_a, where T_a is the most recent transaction in the total order for conflicting operations on X which writes X.

We now prove that G contains an LSTG with respect to UD.

(i) Suppose $T_a \Rightarrow_X T_b$. By Lemma 1, there a path $T_a \rightarrow^* T_b$ in G.

(ii) Suppose T_a and T_b both write to copies of X, $X \epsilon \Pi$. Then since conflicting operations on X are totally ordered (by the above argument) by edges in G_Π, either $T_a \rightarrow^* T_b$ or $T_b \rightarrow^* T_a$ in G_Π and hence in G.

(iii) Suppose $T_a \Rightarrow_X T_b$ and $T_a \rightarrow_X T_c$. Since logically conflicting operations on X are totally ordered by edges in G_Π, either $T_b \rightarrow^* T_c$ or $T_c \rightarrow^* T_b$ in G_Π. Suppose $T_c \rightarrow^* T_b$. Then T_b does not *READ-X-FROM* T_a, a contradiction. Thus, $T_b \rightarrow^* T_c$ in G.

6 Conclusions and Future Work

One of the main ideas in this paper is the grouping of data into domains so that a suitable partition processing strategy can be applied separately to each domain. By adjusting the granularity of the data domains, we can attempt to provide maximum data availability. If all user data are members of a single domain, then transaction processing can occur only in the DP for this domain. Even if some other partition contains quorums for all the data items which a transaction initiated in that partition needs to access, the transaction will be blocked. On the other hand, if a DP is established separately for each data item, then the DP's for two different data items may be different so that a transaction that needs to access both data items cannot be executed in either partition. For example, in a banking application, DP's for checking account information and savings account information may be different, and transactions that need to access both cannot execute. By grouping logically related data items together, greater availability may be achieved. Domains can be dynamically adjusted by means of control transactions, provided that a control transaction making such an adjustment is executed in the DP for all the domains concerned. To minimize the reduction in data availability that can occur when transactions are blocked during the commit process because of failures, a generalization of the quorum-based commit and termination protocols presented in [8] should be used.

Our algorithm guarantees correctness by imposing the following conditions on the sub-methods used:

(1) Any two logically conflicting operations on some domain that occur during a single activation of that domain should physically conflict.

(2) In the total order imposed on logically conflicting operations for a particular data item X, a read operation always reads the value written by the most recent write operation in the total order.

For example, a quorum-based method used within a partition for processing read and write operations together with a correct concurrency control algorithm guarantees (1). A static or dynamic voting scheme for determining the DP for a domain together with (1) should ensure (2).

Thus, the burden of proving the algorithm correct is reduced to the problem of proving that the above conditions are satisfied for each method used.

One of our goals was to provide efficient recovery from failures. When a site recovery or network merge occurs, we would like for normal transaction processing to proceed as soon as possible, without having to wait for data copies to be brought up-to-date. We provide for efficient recovery by marking out-of-date copies as unreadable and allowing copier transactions which refresh such unreadable data copies to run concurrently with user transactions. By providing a special-purpose RECOVERY transaction, our algorithm achieves the same efficiency in the case of site failures as other algorithms which deal only with site failures.

For future work, we wish to incorporate optimistic and semantic approaches into our general method, allowing different approaches to be used with different domains. We plan to establish conventions that must be followed and conditions that must be satisfied by the given methods to maintain consistency of the replicated data. We also plan to explore various means of initiating control and copier transactions, perhaps having them be initiated by supporting subsystems rather than triggered by user transactions. We intend to implement our partition processing methods in the prototype RAID distributed database system so as to be able to simulate failure scenarios under various conditions and evaluate the performance of our algorithms.

References

[1] A. E. Abbadi and S. Toueg. Availability in partitioned replicated databases. In *Proc. Fifth ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 240–251, March 1986.

[2] P. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst.*, 9(4):596–615, Dec. 1984.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] B. Bhargava and P. Ng. A dynamic majority determination algorithm for reconfiguration of network partitions. *to appear in International Journal of Information Science*, Sep. 1988.

[5] B. Bhargava and Z. Ruan. Site recovery in replicated distributed database systems. In *Proceedings of the 6th Intl. Conf. on Distributed Computing Syst.*, May 1986.

[6] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *Computing Surveys*, 17(3):341–370, Sep. 1985.

[7] D. K. Gifford. Weighted voting for replicated data. In *Proc. Seventh Symposium on Operating Systems Principles*, pages 150–162, ACM, Dec. 1979.

[8] C. Huang and V. Li. A quorum-based commit and termination protocol for distributed database systems. In *Proc. Fourth International Conference on Data Engineering*, pages 136–143, IEEE Computer Society Press, Feb. 1988.

[9] S. Jajodia and D. Mutchler. Dynamic voting. In *Proc. ACM SIGMOD 1987 Annual Conference*, pages 227–238, May 1987.

[10] S. Jajodia and D. Mutchler. Integrating static and dynamic voting protocols to enhance file availability. In *Proc. Fourth International Conference on Data Engineering*, pages 144–153, IEEE Computer Society Press, Feb. 1988.

[11] K. Ramarao. Transaction atomicity in the presence of network partitions. In *Proc. Fourth International Conference on Data Engineering*, pages 512–519, IEEE Computer Society Press, Feb. 1988.

[12] Z. Ruan. *File replication in distributed systems.* PhD thesis, Purdue University, Aug. 1986.

[13] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transaction on Software Engineering*, SE-9(3):219–227, May 1983.

[14] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.