A HIGHLY FAULT-TOLERANT DISTRIBUTED DATABASE SYSTEM WITH REPLICATED DATA

DISSERTATION

Presented to the Graduate Council of the University of North Texas in Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

By

Tsai S. Lin, B.A., M.A.
Denton, Texas
December, 1994
A HIGHLY FAULT-TOLERANT DISTRIBUTED DATABASE
SYSTEM WITH REPLICA TED DATA

DISSERTATION

Presented to the Graduate Council of the
University of North Texas in Partial
Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

By

Tsai S. Lin, B.A., M.A.
Denton, Texas
December, 1994

Because of the high cost and impracticality of a high connectivity network, most recent research in transaction processing has focused on a distributed replicated database system. In such a system, multiple copies of a data item are created and stored at several sites in the network, so that the system is able to tolerate more crash and communication failures and attain higher data availability. However, the multiple copies also introduce a global inconsistency problem, especially in a partitioned network. In this dissertation a tree quorum algorithm is proposed to solve this problem, imposing a logical tree structure along with dynamic system reconfiguration on all the copies of each data item. The proposed algorithm can be viewed as a dynamic voting technique which, with the help of an appropriate concurrency control algorithm, exhibits the major advantages of quorum-based replica control algorithms and of the available copies algorithm, so that a single copy is read for a read operation and a quorum of copies is written for a write operation. In addition, read and write quorums are computed dynamically and independently. As a result expensive read operations, like those that require several copies of a data item to be read in most quorum schemes, are eliminated. Furthermore, the message costs of read and write operations are reduced by the use of smaller quorum sizes. Quorum sizes can be reduced to a constant in a lightly loaded system, and log \( n \) in a failure-free network, as well as \( \lfloor n + 1/2 \rfloor \) in a partitioned network in a heavily loaded system. On average, our algorithm requires fewer messages than the best known tree quorum algorithm, while
still maintaining the same upper bound on quorum size. One-copy serializability is

guaranteed with higher data availability and highest degree of fault tolerance (up to

$n - 1$ site failures).
# TABLE OF CONTENTS

**LIST OF FIGURES**

**LIST OF TABLES**

1 INTRODUCTION

2 LITERATURE SURVEY

2.1 Concurrency Control for a Replicated Database

2.2 Replica Control Algorithm

2.3 Partition Processing Strategies

2.3.1 Syntactic-Optimistic Strategies

2.3.2 Syntactic-Pessimistic Strategies

2.3.3 Semantic-Optimistic Strategies

2.3.4 Semantic-Pessimistic Strategies

3 MODEL

4 GENERAL DESCRIPTION OF THE ALGORITHM

5 THE NCA QUORUM

5.1 Nca Quorums for Complete Binary Trees

5.1.1 The Preprocessing Step

5.1.2 Nca Quorums for Failure-Free Binary Trees

5.1.3 Correctness of the Algorithm

5.2 Nca Quorums for Partitioned Binary Trees

5.3 Nca Quorums for Trees of Degree $d$

5.3.1 The Preprocessing Step for Trees of Degree $d$

5.3.2 Nca Quorums for Trees of Degree $d$

5.4 Nca Quorum for Partitioned Trees of Degree $d$

5.5 Analysis

5.5.1 Comparison between Nca Quorum and Tree Quorum Algorithms

5.5.2 The General Behavior of the Nca Quorum Algorithm

5.6 The Complete Nca Algorithms

6 DETAILS OF THE ALGORITHM

6.1 Transaction Manager

6.2 Replica Controller

6.3 Concurrency Controller

6.4 Suggested Time Section Techniques
LIST OF FIGURES

5.1 A complete binary tree of 15 nodes with ID values shown in parentheses. 38
5.2 A tree of degree 3 with ID values shown in parentheses. 45
5.3 A binary tree of 7 nodes. 54
5.4 A binary tree of 15 nodes. 56
5.5 Two patterns showing how disjoint subtrees affect the sizes of nca quorums. 57

7.1 The average number of messages for 10% read operations. 87
7.2 The average number of messages for 50% read operations. 87
7.3 The average number of messages for 90% read operations. 88
7.4 The average restart rate per transaction for 10% read operations. 89
7.5 The average restart rate per transaction for 50% read operations. 89
7.6 The average restart rate per transaction for 90% read operations. 90
7.7 The throughput per 1000 milliseconds for 10% read operations. 90
7.8 The throughput per 1000 milliseconds for 50% read operations. 91
7.9 The throughput per 1000 milliseconds for 90% read operations. 91
7.10 The average response time per transaction for 10% read operations. 92
7.11 The average response time per transaction for 50% read operations. 93
7.12 The average response time per transaction for 90% read operations. 93
LIST OF TABLES

5.1 The average size of nca quorums for different percentages of requesting nodes in the underlying network. .......................... 50
5.2 The average size of nca quorums when nodes on the different levels fail in the underlying tree network. ......................... 51
5.3 The average size of nca quorums when individual nodes fail at different levels. ................................................. 52
5.4 The average size of nca quorums when nodes indicated by the first column fail. ............................................... 53

8.1 Lower/upper bounds of majority tree quorums and corresponding nca quorums for a ternary tree containing n nodes. ...................... 97
8.2 Lower/upper bounds of read root quorums and corresponding nca quorums for a ternary tree containing n nodes. ...................... 98
8.3 Lower/upper bounds of log write quorums and corresponding nca quorums for a ternary tree containing n nodes. ...................... 98
CHAPTER 1

INTRODUCTION

A transaction is a sequence of read and write operations on a set of data items. Correct concurrent transaction processing in a database system requires all transactions to be executed atomically, so that they appear to users as indivisible and isolated actions on the database. To execute transactions atomically, two conditions must be satisfied: atomic commitment and serializability. Atomic commitment requires that the effect of transaction execution be "all or nothing"—either all updates of a transaction are performed or none are performed, so that no partial effects are reflected in the database. In most systems, this requirement is established by a commit algorithm and a recovery algorithm. Serializability is another requirement for correct transaction processing which requires that correct concurrent execution of several transactions produce the same database state as some serial execution of the same transactions. A transaction, when executed alone, transforms an initially correct database state into another correct state, so a serial execution of some transactions, by induction, will transform an initially correct database state into another correct database state. Serializability is the best correctness criterion that one can hope for when only syntactic information is available.

When implemented in a conventional single copy database system, transaction processing, even atomically executed, may suffer unnecessary delay due to some nonconflicting operation requests. For example, concurrent reads are not allowed in such a system, so in order to increase data availability and improve system perfor-
mance a distributed database system must replicate and store several copies of the same data item in distinct sites with different failure modes. The replication method consequently can introduce global data inconsistency among different copies of the same data item, even in a nonpartitioned network. The purpose of a replica control algorithm is to ensure that concurrent execution of transactions on replicated data is equivalent to a serial execution on nonreplicated data. Such a property is called *one-copy serializability*. Consequently, a replicated database system acts as a single copy system. With one-copy serializability, as with serializability, no semantics of the underlying database and of executed transactions is required. For a distributed database with replicated data, when only syntactic information about the database system is available, these two properties are the sole correctness criteria that one can expect. In general, these two criteria can be enforced by a correct replica control algorithm along with an appropriate concurrency control algorithm.

The replica control algorithm is powerful enough to ensure one-copy serializability within a single partition. However, when the system is partitioned, separate updates on different copies of a data item may occur in distinct partitions. Also, a transaction ongoing at partition time may not be atomically committed. Hence, global data inconsistency among different partitions must be resolved by a partition control algorithm, which determines the trade-off between data availability and data consistency in the system.

Because of the complexity and diverseness of problems encountered in a distributed database system with replicated data, an appropriate model is expected to be difficult to construct and highly application dependent. This is one of the major criticisms of previous studies that attempted to devise strategies for isolated problems [46]. Our algorithm can not avoid this criticism either. Instead of trying to solve
all the problems at once, we introduce a replica control algorithm which, when executed with a timestamp-based concurrency control algorithm, can achieve one-copy serializability with minimum message cost.

In the next chapter, previous related work is reviewed. Then the underlying system model is defined in Chapter 3. In Chapter 4, the general behavior of the algorithm is given, followed by the description of the nca quorum algorithm in Chapter 5. The details of the underlying complete algorithm are demonstrated in Chapter 6. Chapter 7 analyzes system performance by using a simulation model program in SR. The results are collected and analyzed for networks containing seven and fifteen sites. Chapter 8 compares the performance of the tree quorum algorithm and the nca quorum algorithm by running both algorithms in various system configurations simultaneously. Finally, the major achievement of the nca quorum algorithm is reviewed.
CHAPTER 2
LITERATURE SURVEY

Since several general concurrency control algorithms and replica control algorithms have been frequently used in the previous partition processing algorithms explicitly and implicitly, we will introduce them first before reviewing most partition processing strategies.

2.1 Concurrency Control for a Replicated Database

The purpose of a concurrency control algorithm is to safeguard the global consistency of the different copies of a data item by properly granting or rejecting operation requests [39]. In general, the more information the scheduler has, the better job it can do in scheduling transactions. The information available in transaction processing includes syntactic information (such as operation types and variable names), semantic information (such as the meaning of the data and operations), and integrity constraints (such as the consistency requirement of data items). In this section we consider only syntactic information.

The first concurrency control algorithm for transaction processing is two-phase locking (2PL) introduced by Gray in 1976 [24]. In 2PL, transactions are required to be well-formed and two-phase in order to maintain consistency. Otherwise, it is possible to construct a legal but inconsistent schedule. A transaction is a sequence of n steps \( T = (T, a_i, e_i)_{i=1}^n \), where \( T \) is the transaction name, \( a_i \) is the action at step \( i \) and \( e_i \) is the entity \( e \) acted upon at step \( i \). A transaction has locked entity \( e \)
through step $i$ if for some $j \leq i, a_j = \text{lock}$ and $e_j = e$ and there is no $k$ such that $j < k < i, a_k = \text{unlock}$ and $e_k = e$. A transaction $T$ is well-formed if (1) for each step $i = 1, \ldots, n$, if $a_i = \text{lock}$, then $e_i$ is not locked by $T$ through step $i$, and (2) at step $n$, only $c_n$ is still locked by $T$ and $a_n = \text{unlock}$. A two-phase transaction has a growing phase and a shrinking phase. During the growing phase, the transaction can request any new locks. However, once a lock has been released, no new lock can be requested. In practice, the coordinator compares the requested predicate lock against the outstanding predicate locks of other transactions. If no conflict exists, execution permission can be granted. 2PL is the best among all locking polices when only syntactic information on unstructured variables is available [39].

In 2PL node failures may force some nodes to wait for the repair of failures before finishing the transaction. However, if node failures are never repaired, then the waiting nodes may be blocked indefinitely. This blocking problem can severely affect system availability. The open two-phase commit protocol (O2PC) [54] solves this problem by modeling the execution of a transaction sequence in a process tree. A transaction sequence is a collection of transactions that are performed sequentially. That is, before a new transaction can be initiated in the next transaction sequence, the previous transaction sequence must have terminated. The root of a process tree is the process which initiates all transactions in the transaction sequence. The edges in the tree represent client/server relationships between processes. Each process cooperates and communicates only with its directed neighbors in the process tree. A node which eventually recover from failures is called a trusted node. Otherwise, it is called a non-trusted node. A trusted process is called an entry process if its parent is a non-trusted process. Non-trusted processes may have trusted as well as non-trusted children in the process tree, while trusted processes are assumed to have trusted children only. Thus,
entry processes have trusted descendants and non-trusted ancestors. O2PC transfers
the coordinator function to the entry node within a process tree, or attaches a trusted
process to the root as coordinator, so that each trusted participant terminates the
transaction eventually and consistently.

Another approach to solve the 2PL blocking problem is the canonical non-
blocking protocol (three-phase locking or 3PL) [57]. A distributed 2PL protocol is
synchronous within one state transaction if one site never leads another site by more
than one state transaction during the execution of the protocol. Such a protocol is
nonblocking if and only if both following conditions are satisfied:

1. it contains no local state adjacent to both a commit and an abort state

2. it contains no noncommittable state that is adjacent to a commit state

where a local state is called committable if occupancy of that state by any site implies
that all sites have voted yes on committing the transaction. Otherwise, the state is
called noncommittable. Since in 2PL only wait states violate both constraints of the
nonblocking protocol, 3PL inserts an additional buffer state, “prepare to commit”,
between the wait and the commit states of a transaction, so that nodes in the wait
state are prevented from entering the commit and the abort states simultaneously.

The simplest concurrency control algorithm that can be used directly with a
replica control algorithm is the timestamp-based algorithm [11]. In this algorithm, a
timestamp consisting of a local time and an unique transaction identifier is assigned
when a transaction starts. For each data item the scheduler keeps track of the largest
timestamps, $R$-timestamp($x$) and $W$-timestamp($x$), of any read and write operations
on $x$. If the timestamp of a read request is greater than $W$-timestamp, the read opera-
tion can proceed. Otherwise, it will be rejected. Similarly, a write request is compared
to $R$-timestamp($x$). As a result, transactions are executed in their timestamp order.

2.2 Replica Control Algorithm

The available copies algorithm [13] is designed for a nonpartitioned network. It tolerates site failures but not communication failures. The algorithm maintains a directory on the site which runs a transaction operating on a data item $x$. This directory contains all available copies of the data item $x$ and is kept up-to-date by running a special status transaction. To read $x$, a transaction consults the directory and then reads one single copy of $x$. To write $x$, a transaction writes all accessible copies after consulting the directory. Thus, to tolerate $k$ failures, this algorithm needs $k + 1$ copies. This algorithm replicates directories and copies of a data item at the sites running the transaction, which is very expensive. It assumes, therefore, that sites must fail infrequently and data access patterns are predictable. As a result, it is a preferable system only in a reasonably stable environment.

In 1986, Pu [51] improved data availability in the available copies algorithm by using a "lazy" regeneration strategy. In Pu's algorithm, if some copies of $x$ become inaccessible due to node crashes, the algorithm does not restore the replication level of $x$ by automatically replacing lost copies on other machines until $x$ is being updated. This lazy regeneration algorithm is self adaptive to system configuration changes and drops the inaccessible copies. Thus, consistent replicated data can be read or written despite losing all copies but one. Compared to the available copies algorithm, this algorithm is preferable in an unstable environment that suffers from high crash frequencies, with a high number of updates and frequent reconfiguration. In spite of these improvements, the algorithm still cannot guarantee consistency in
the presence of network partition or Byzantine failure.

Since data transfer is very expensive operation, the next variant of the available copies algorithm does regeneration only when absolutely necessary. Adam and Tewavi's \textit{selectively utilize regeneration algorithm} [1] improves the read-one/write-all available copies algorithm [13] in which "virtual copies" containing state information and votes are preallocated on some nodes in the system. Whenever the number of real copies drops below a predefined generation threshold, the algorithm converts a virtual copy to a real copy. When nodes recover from failures, a real copy can be converted back to a virtual copy. Using this dynamical copy regeneration strategy, the system can efficiently respond to node failures and network partitions. To read a data item, the algorithm reads only one current real copy of the data object. To write a data object, the algorithm needs to write a minimum of one current real copy of a data item. Other copies participating in the decision making process can be virtual copies.

2.3 Partition Processing Strategies

In general, a transaction processing algorithm for partitioned networks can be classified by whether it is concerned with the trade-off between consistency and availability, or with the type of information used in defining correctness. Two extreme approaches to the first concern are the pessimistic and optimistic approaches, and two extreme approaches to the other concern are the syntactic and semantic approaches.

Pessimistic approaches prevent inconsistencies by limiting availability and allowing only one partition to make updates during partition. To regain consistency of the different copies of a data item, the database system propagates updates from
one partition to other partitions at reconnection time. In contrast, optimistic approaches maximize data availability by compromising global consistency. They allow all transactions having their needed data to proceed as usual in different partitions. Global inconsistencies are detected and resolved at reconnection time. Usually, a graph method is used to detect inconsistency among different copies of a data item, followed by a compensating transaction to nullify the external effects of a transaction and a correcting transaction to transform the database to a “correct” but not necessarily serializable state.

The major implementation cost of optimistic approaches is the cost of conflict detection and inconsistency repair. Pessimistic approaches do not have this expense (except algorithms using a transaction class conflict analysis strategy). However, the major implementation cost of the pessimistic approach is lost opportunities.

Most syntactic approaches use one-copy serializability as their sole correctness criterion, and check serializability by examining readsets and writesets of all the concurrent transactions. They are implemented by using a general-purpose concurrency control algorithm such as 2PL [24], 3PL [57], or a timestamp-based method [11], along with a standard replica control algorithm such as available copies [13], regeneration [51], or voting [28]. In contrast, semantic approaches use the semantics of the database or of the executed transactions in defining correctness.

As a result, transaction processing algorithms can be classified by their partition control policy and their definition of correctness along these two coordinates: syntactic-optimistic, syntactic-pessimistic, semantic-optimistic, and semantic-pessimistic.
2.3.1 Syntactic-Optimistic Strategies

A version vector \( v \) [50] is a sequence of \( n \) pairs of integers, where \( n \) is the number of sites at which a file \( f \) is stored. The \( i^{th} \) entry \( (S_i : v_i) \) counts the number \( v_i \) of updates to file \( f \) originating at site \( S_i \). If there are two version vectors \( v \) and \( u \) for the same file such that not all \( v_i \geq u_i \), a write-write conflict has been detected. This approach works well for a single file system, but not for multifile transactions. Also, version vectors detect write-write conflicts only. Read-write conflict detection must use an additional log technique.

The optimistic protocol [19, 20] uses a precedence graph to detect inconsistency. The nodes of the precedence graph represent transactions; the edges represent interactions between transactions. In the same partition, two types of edges represent read-write and write-read conflicts: dependency edges and precedence edges. A dependency edge \( (T_i \rightarrow T_j) \) represents one transaction \( T_j \) reading a value written by another transaction \( T_i \), while a precedence edge \( (T_i \Rightarrow T_j) \) represents one transaction \( T_i \) reading a value that is later changed by another transaction \( T_j \). Read-write and write-read conflicts between different partitions are represented by interference edges \( (T_i \rightarrow T_j) \) that represent one transaction \( T_i \) reading a value that was written by another transaction \( T_j \) in another partition. If the precedence graph is acyclic, the resulting database state is consistent and a serial order for the transactions can be obtained by topologically sorting the precedence graph. Inconsistencies are resolved by rolling back transactions until the resulting subgraph is acyclic. The optimistic protocol performs best when a small percentage of items are updated during a partitioned period and most transactions have large writesets.

Nonblocking concurrency control [40] combines locking and backup techniques.
Thus, the deadlock problem of a locking technique is solved by using backup, and the starvation problem of an optimistic approach is solved by using locking. Read operations are completely unrestricted, but writes are severely restricted. Write operations consist of two or three phases: a read phase, a validation phase, and a possible write phase. During the read phase, all writes take place on local copies of the data to be modified. Later, local copies are made global in the write phase if local write operations do not violate validation conditions in the validation phase checking.

Transactions are serialized by transaction numbers being read at the beginning of the read phase and tentatively assigned at the end of the read phase. The system checks that the write sets of all active transactions that have been assigned "tentative" transaction numbers less than that of the transaction being validated do not intersect the read set or write set of the current transaction. By repeating these steps as many times as necessary, the various number of validation stages and different degrees of parallelism can be achieved. The only restriction is that each transaction must have a final step containing both validation and write phases.

Optimistic concurrency control, proposed by Thomason and Rahm [63], embeds the validation phase and write phase of an optimistic approach into a two-phase commit protocol. In so doing, no extra messages are introduced to achieve high throughput. All objects, not just the modified ones, are requested in the read phase, before the validation phase. If the validation phase is successful, these locks are held only during the commit phase. Otherwise, the locks are retained during the reprocessing step. Thus, no transaction validation will be re-executed more than once. Also, transactions are executed according to their timestamp ordering instead of the transaction numbers of the nonblocking concurrency control algorithm.
2.3.2 Syntactic-Pessimistic Strategies

The primary site (copy) algorithm was first introduced by Alsberg and Day in 1976 [7, 61]. One copy of a data item \( x \) is designated as the primary copy for \( x \). This copy is responsible for all operations on \( x \). All read requests for \( x \) must be performed in the primary copy, and updates on \( x \) must also be propagated from the primary copy to other copies. When the system partitions, only the partition containing the primary copy of \( x \) can access \( x \). Global consistency is regained at reconnection time by forwarding all missing updates from the primary copy to the other copies. The primary site algorithm works well only if site failures are distinguishable from network failures so that, when the site where the primary copy resides fails, a new primary site can be selected.

The tokens algorithm [45] can be viewed as a dynamic version of the primary site algorithm. There is a token associated with each data item \( x \). Only the bearer of the token can access and operate on \( x \). In a partitioned network, the only partition that can access \( x \) is the one containing the token bearer. Obviously, access to \( x \) is impossible if the token of \( x \) is lost as a result of site or communication failure.

Majority Consensus Algorithm

Correctness of the two previous algorithms require the system to distinguish among site failures, communication failures and just slow response. This requirement is not necessary for all voting strategies. The first static voting approach was the majority consensus algorithm [62], where a request is performed if a majority of processes agree. Since no two requests will obtain a majority at the same time, no two operation requests will be performed simultaneously. Later, Gifford [28] generalized
the idea by assigning different numbers of votes to the copies of a data item $x$. Every transaction must collect a read quorum of $r$ votes to read $x$ and a write quorum of $w$ votes to write $x$. All quorums must satisfy two constraints: (1) $r + w$ exceeds the total number of votes $v$ assigned to $x$, and (2) $w > v/2$. The first constraint requires every read quorum to have a nonnull intersection with every write quorum. Thus, read-write conflicts existing in different partitions can not occur. Similarly, the second constraint ensures write-write conflicts can not occur. This general static voting technique allows $r$ and $w$ to be chosen according to the read/write ratio, the cost of reading and writing, and the desired reliability and performance. As $r$ decreases, the reliability and performance of reads increases. The same applies to $w$. Compared to other algorithms, the voting scheme is much more flexible and simple to implement, but it suffers a major drawback that a read operation becomes fairly expensive. To read a data item, $r$ copies must be accessed instead of one single copy. To reduce the expense of the read operation and increase data availability, Carroll [14] combined the majority voting and available copy techniques. The cost of a read operation is reduced to one and, as long as the failure-to-repair ratio remains less than or equal to one, the resulting algorithm can double the data availability of a voting algorithm.

Another variant of the majority consensus algorithm is the missing writes algorithm [22], in which transactions run in two different modes: normal mode and failure mode. When no failure occurs, the system runs in normal mode and uses a read-one/write-all technique. If some copies of a data item $x$ cannot be updated, the transaction to which $x$ belongs becomes “aware” of a missing update and runs in the failure mode using the majority consensus algorithm. Missing update information is passed along to all subsequent transactions that need the information. When failures are repaired, the information will eventually be posted at the sites that caused the
missing updates. The missing writes algorithm allows read-only transactions to “run in the past” during a failure, as long as no deadlock occurs. The major drawback of this algorithm is that too many files must be kept at each site for missing updates. Later Sandoz [55] weakened the read-access rule of the missing writes algorithm to allow no-quorum read access of a data item if this data item has been completely updated in the past. Consequently, the algorithm provides better data availability.

Another algorithm similar to the missing writes algorithm is the accessible copies algorithm [23]. This algorithm is run in a single mode only, with a data item \( x \) being read and written only in the majority partition. A read is implemented by reading the nearest copy of \( x \), and a write is implemented by writing all copies of \( x \) in the majority partition. The algorithm ensures one-copy serializability only in an ideal network in which partition failures are clean and detected instantaneously.

Coterie Voting Techniques

In addition to the high cost of a read operation, the majority voting technique suffers from high network traffic and message cost because all the nodes in the system participate in the decision making process. An alternative voting scheme that avoids this significant drawback is the coterie voting technique. A coterie \( C \) is a set of subsets of \( U \), the set of nodes in the distributed system, such that each element of \( C \) is a nonempty subset of \( U \), each pair of elements has at least one node in common, and no element is a subset of the other. Each element of the coterie is called a quorum. The coterie concept has the nonnull intersection property of the standard voting technique, and requires only a subset of nodes in the system to participate in a decision making process. Garcia-Molina has given examples showing that not all coteries have a corresponding vote assignment, and therefore coteries are more
powerful than the standard voting technique [27]. Because of these superiorities, coteries have attracted much interest as a voting scheme.

Garcia-Molina later classified coteries into dominated and nondominated coteries. A coterie \( R \) dominates another coterie \( S \) iff \( R \neq S \) and, for each \( H \) belonging to \( S \), there is a \( G \) belonging to \( R \) such that \( G \) is a subset of \( H \). A coterie \( S \) is dominated iff there is another coterie which dominates \( S \); otherwise, \( S \) is nondominated. When a partition occurs, a quorum can be found in a nondominated coterie but not necessary in a coterie it dominates. Thus, a nondominated coterie is more powerful than a dominated one. Garcia-Molina also provided an algorithm to totally enumerate nondominated coteries for 5 or fewer nodes and partially enumerate nondominated coteries for 6 or more nodes.

The process of obtaining better a vote assignment is an exhaustive work. In 1987 Barbara and Garcia-Molina [10] used heuristic methods to assign votes in such a way that the probability of having an active group is maximized. Three heuristics were suggested for finding better voting assignments: (1) For each node \( i \), multiply its reliability by the sum of the reliabilities of the links incident to \( i \), rounding the number obtained to get the number of votes assigned to \( i \). (2) The second heuristic is similar to the first but takes into account the reliability of adjacent nodes. For each link incident to node \( i \), multiply its reliability by the reliability of its other endpoint. Sum all the these products and multiply by the reliability of \( i \). After rounding, the result is the number of votes assigned to \( i \). All the vote assignments with an odd number of votes are nondominated. Thus, if the total number of votes is even, the node with most votes gets an extra one. (3) Remove any weak links from the network first, then assign votes to a node \( i \) by using heuristic 2. A weak link is a link whose reliability multiplied by the reliability of any one of its endpoints is less than 0.5.
Cheung [15] was the first to try to optimize vote and quorum assignment with respect to high system availability by using a systematic method which utilized a linear program (LP) to obtain an enumeration of vote assignable coterries with their corresponding vote assignments. LP allows a greater number of votes to be assigned to nodes with higher availability, so compared to the commonly used quorums where each node has a single vote, the algorithm provides better system availability. Also, when node availability and the read/write ratio of a network are given, the coterie with highest availability then can be computed effectively.

Recently, Kumar and Segev [38] have presented a static semi-exhaustive algorithm to generate nonequivalent vote combinations, and used a partial enumeration heuristic to produce alternative assignments for each vote combination. Their goal was to minimize communication cost, tolerate more faults, and obtain higher availability.

Dynamic Voting Techniques

In addition to using coterries to reduce the size of read and write quorums, another way of increasing system availability is to dynamically reassign votes. Since groups can not communicate with one another when partitioned, it is possible that no request will obtain a majority of votes at a given time. Under these circumstances, the system is said to become “halted”. Barbara [9] lets a group containing a majority of votes to reassign the votes dynamically in the case of a partitioned network, so that further partitions may have a better chance of having a majority of votes. Dynamic vote reassignment can improve uptime over a static assignment by a factor of 2 or 3.

At the same time, Paris [48, 49], instead of altering read and write quorums, suggested voting with witnesses: preallocate votes to sites called witnesses. These
witness sites have votes but no actual copy of a data item. The conventional copies and witnesses can be interchanged dynamically. When a file resides on sites that are overloaded or expensive to access, the conventional copies can be transformed into witnesses. On the other hand, when the system configuration changes, witnesses can be upgraded to conventional copies without altering read and write quorums. Using witnesses, the system can dynamically change the number and location of actual copies to reflect file access patterns, system availability and storage costs. The reliability of a system consisting of $n$ copies and $m$ witnesses is the same as the reliability of a system containing $n + m$ copies.

Dovcev and Burkhard's *dynamic voting technique* [18] requires neither the reassignment of votes to nodes in the majority partition nor the preallocation of votes to witnesses. Instead, they change quorums dynamically to half of the copies in the majority partition at the last failure. Each physical copy of a replicated file contains a version number and a partition vector. The version number indicates the number of successful update operations. The partition vector $Z_i$ is used to determine "locally" whether or not site $i$ is in the majority partition and what operation is allowed. For each $j$, $Z_i[j]$ is an integer representing the version number of physical copy $i$ at the time of a failure separating physical copies $i$ and $j$. If no partition exists between these two copies, $Z_i[j] = 0$. If the number of zeros of a partition vector is greater than the number of maximum value components, an update is allowed; if the two numbers are equal, only a read is allowed; otherwise, no operation is allowed. The algorithm can tolerate up to $n - 2$ site failures, and allows distinct updates to a particular data item to be interleaved so that serial consistency is not guaranteed.

Jajodia [33] simplified dynamic voting and improved system availability by linearly ordering all sites that have copies of a file $f$. Each site $S$ maintains its
own "connection vector", which has 1 in each element corresponding to a site with which \( S \) can communicate and 0 elsewhere. Communication failures and repairs are recorded instantly in the appropriate connection vectors, so that all connection vectors in a single partition are identical. In addition, there are four values associated with each copy of \( f \): a version number, an update sites cardinality, a distinguished site, and an update sites vector. The version number of a copy \( f_i \) at a site \( S_i \) is an integer \( VN_i \) which counts the number of successful updates to \( f_i \). The update sites cardinality \( SC_i \) reflects the number of sites participating in the most recent update to \( f_i \). The distinguished site variable \( DS_i \) identifies the greatest site, in the order, that participated in the last update to \( f_i \). The update sites vector \( SV_i \) is an ordered tuple \((b_1, b_2, \ldots, b_n)\) of bits, with \( b_i \) corresponding to a site \( S_i \). If a site \( S_i \) is a participant in an update, then \( b_i \) is set to one; otherwise it is set to zero. When a communication link fails or is restored, a connection vector is updated, but the update sites vector is modified only upon failures. When these two values differ, the algorithm obtains missing updates by forming a new majority partition through a null update (increases version number only).

The linear order imposed on all the sites containing a copy of \( f_i \) can be used to "break ties" between partitions when a majority partition decomposes into sub-partitions, each containing an equal number of sites. Also, it permits updates when only a single site is up, which is not allowed in the Davcev-Burkhard algorithm. The partition vector used in the Davcev-Burkhard algorithm is an array of \( n \) integers, while the Jajodia algorithm uses two integers \( SC_i \) and \( DS_i \) and an array of \( n \) bits achieving the same effects.

Dynamic voting requires that at least a majority of the sites that were involved in the most recent quorum be available for the next quorum. As a result, a sudden
change in the state of the network may prevent a quorum from being formed. The dynamic quorum technique [32] gives more freedom to the replica controller to assign quorums and adjust them dynamically in response to node failures and recoveries. A transaction that is unable to progress using one set of quorums may switch to another, more favorable set, and transactions in different partitions may progress using different sets of quorums. When a transaction begins execution, it chooses a natural number called its level. The transactions at different levels are serialized in the order of their levels by using the resident site’s “ratchet lock”. The resident site is the site where the transaction is submitted, and the ratchet lock is a counter that records the highest level of a transaction that has read the object at that site. A site rejects write requests from any transaction whose level is less than the site’s ratchet lock, ensuring that a value read by a higher level transaction can not be overwritten by a lower level one. The transactions at the same level are serialized by their timestamp order.

Any object’s quorum assignments must satisfy the “quorum intersection invariant”: if writes to that object are enabled at level $n$, then each write quorum at level $n$ must interact with each read quorum at levels greater than or equal to $n$.

Two complementary techniques for dynamic quorum adjustment are quorum inflation and quorum deflation [32]. Quorum inflation shrinks write quorums but possibly expands read quorums. In contrast, quorum deflation shrinks read quorums but possibly expands write quorums. Inflation can enhance availability, since a transaction restarting at a higher level may be able to locate a write quorum unavailable at a lower level. Although deflation cannot make unavailable operations available again, it can enhance efficiency, since smaller read quorums require fewer messages. It can also enhance long-term availability, since smaller read quorums are more likely to
survive subsequent failures. For both techniques, quorum adjustment can be viewed as a local process because quorum inflation is undertaken by individual transactions, and quorum deflation is undertaken by individual objects. Global reconfiguration is not required and its effects propagate “lazily” through the system as circumstances require. The dynamic quorum adjustment algorithm tolerates site crashes, timing failures, and communication link failures.

All voting assignments mentioned so far are one dimensional. The first two-dimensional static quorum consensus protocol was multidimensional voting of Ahamad and Cheung [4, 17]. In multidimensional voting (MD), the number of votes assigned to a node and a quorum are k-dimensional vectors of nonnegative integers. Formally, a MD vote assignment $V_{n,k}$ is an $n$ by $k$ matrix, where $v_{i,j}$ represents the vote assignment to node $i$ in the $j^{th}$ dimension. Votes assigned in different dimensions are independent of one another. The quorum assignment $q_k = (q_1, q_2, \ldots, q_k)$ is a $k$-dimensional integer vector. A number $l$, with $1 \leq l \leq k$, is the number of dimensions of vote assignments in which the quorum requirement must be satisfied. Thus, there are two levels of requirements: vote and dimension. At the vote level, the number of votes received for a dimension must be greater than or equal to the quorum requirement in that dimension. At the dimension level, the number of dimensions for which a quorum is collected must be greater than or equal to $l$. We denote MD-voting with quorum requirement in $l$ of $k$ dimensions as $MD(l,k)$-voting, the standard voting, SD-voting [28], is $MD(1,1)$.

In the actual implementation, each node stores its vote, which consists of $k$ integers, and each operation has a quorum requirement for each dimension and the value of $l$. An operation requests permission from the nodes by sending a vote request to them. When a node receives a vote request, it votes “reject” or replies with its
vote in all dimensions. Each operation maintains \( k \) independent variables, which accumulate the votes received in each dimension. When a response containing a vote is received, the operation adds the vote in each dimension to the appropriate variable. When the sum in at least \( l \) variables are greater than or equal to the quorum requirement in the corresponding dimensions, the operation can proceed.

A replica control protocol based on MD-voting is defined as follows. Let the read quorum set be \( MD(l, k) \)-voting. To satisfy the read-write and write-write intersection properties, the write quorum must contain \( w = \{ A \cup B | A \cup B \text{ is minimal and } A \in Q_{i,k}(Vn, k, rk), B \in Q_{k-l+1,k}(Vn, k, wk) \} \), where \( Q_{i,k}(Vn, k, rk) \) is the smallest read quorum assignment represented by the \( MD(l, k) \)-voting, and \( Q_{k-l+1,k}(Vn, k, wk) \) is the smallest write quorum assignment represented by the \( MD(k - l + 1, k) \)-voting.

Any quorum set \( Q \) can be represented by an \( MD(1, k) \) vote and quorum assignment where \( k = |Q| \). MD-voting tries to represent several groups by a single dimension. Therefore, in practice, the number of dimensions needed to obtain an \( MD(1, k) \) vote assignment could be much less than \( |Q| \). Thus, the number of dimensions is smaller than the number of groups in the coterie. MD-voting allows the user to use the state information to choose the best quorum set and to reconfigure itself in anticipation of future failures. It is as powerful as the method of quorum sets and has the flexibility and ease of implementation of voting.

Logical Structure Voting Assignment

The latest development in voting schemes is to impose a logical structure on the underlying network, and to use the characteristics of the logical structure to construct quorums and to define quorum intersection constraints. The first logical structure was
Cheung's *grid protocol* [16], in which all copies of a data item are arranged in a logical grid. Read quorums and write quorums are assigned as follows. Each read quorum must contain a *C-cover* $G$, which is a set of nodes such that each column intersects with $G$, and each write quorum must contain both a *C-cover* and all nodes in some column. The grid protocol allows a data item to be shared by several read operations but it still enforces mutual exclusion on conflicting operations by requesting locks from a common node. This approach provides better load sharing and response time for a system with a high arrival rate than all previous voting schemes. Also, since the grid topology affects read and write quorum size and availability, the algorithm provides the user the flexibility of choosing the dimension of the grid for a fixed number of nodes based on read/write ratio and availability requirements.

The next logical structure used by quorum assignments is the logical tree structure. The *hierarchical quorum consensus algorithm (HQC)* was proposed by Kumar [36] whose objective was to develop a synchronization scheme that scales well even when a large number of objects are present, unlike the quorum consensus method in which quorum size increases linearly with the total number of objects. Kumar organized a set of objects into a multi-level tree with the root at level 0 and stored the physical objects in the leaves of the tree at level $n$. To construct a read quorum, start from the root with $r_1$ logical objects at level 1. Each logical object at level 1 in turn has $r_2$ logical objects at level 2, and so on. Consequently, the total number of physical objects is $r_1 \times r_2 \times \ldots \times r_n$. To perform read and write operations, appropriate quorums must be assembled by traversing the hierarchy from the root to the level of the physical objects, i.e., the bottom level. A concurrency control scheme is correct if for all levels $i$, where $i = 1, \ldots, n$, the read quorum and the write quorum have a nonnull intersection. One condition that ensures such an intersection
is \( r_i + w_i > l_i \) where \( l_i \) is the number of objects at level \( i \). Also two write quorums must have a nonnull intersection. One condition that ensures such an intersection is \( 2w_i > l_i \) where \( l_i \). \( HQC \) outperforms majority voting and dynamic voting in system availability for a low connectivity network with highly reliable links. On the other hand, majority voting and dynamic voting produce a higher system availability when either network connectivity is high or links are less reliable in a low connectivity network.

The hierarchical quorum consensus algorithm stores the physical objects in the leaves of a tree. The \textit{tree quorum} [3] algorithm, on the other hand, stores them in the internal nodes as well the leaves of the tree. Through this arrangement, tree quorum provides two degrees of freedom for choosing quorums: the number of levels and the number of children. This results in significantly lower communication costs for comparable data availability, and exhibits the property of graceful degradation, in that communication costs for executing operations are minimal in a failure-free environment but may increase as failures occur. The algorithm also provides fault-tolerance without imposing unnecessary costs on the failure-free mode of operation. A generalization of the simple tree protocol requires each write operation to write a quorum of levels only, and read operations to access a quorum of levels so that there is an intersection at some level between read and write quorums. Also, operations are required to access a quorum of children so that read and write operations have an intersection at at least one child.

A tree quorum of length \( l \) and width \( w \) is constructed by selecting the root, \( w \) children of the root and, for each child selected, \( w \) of its children, and so on to depth \( l \). If the construction is successful, it forms a tree quorum of height \( l \) and width \( w \). However, if a node is inaccessible at depth \( h' \) from the root when the tree quorum
is constructed, the node is replaced recursively by \( w \) tree quorums of height \( l - h' \) starting from the children of the inaccessible node. In a tree of height \( h \) and degree \( d \), a successful tree quorum protocol guarantees that, for any read quorum of size \( q_r = (l_r, w_r) \) and any write quorum \( q_w = (l_w, w_w) \), it will be true that that \( l_r + l_w > h \) and \( w_r + w_w > d \), so that read and write quorums have a nonempty intersection. A tree quorum of an incomplete tree is constructed the same way as that of a complete tree. Let \( n \) be the actual number of children of a node \( c \) in a tree so that \( c \) has \( d - n \) missing children. When node \( c \) is included in a tree quorum, \( d - n \) subtrees with the missing children of \( c \) as roots are implicitly included in that quorum. Instead of accessing \( w \) subtrees while constructing a tree quorum, only \( w - (d - n) \) subtrees need to be included for each node \( c \). If \( w - (d - n) \) is zero or negative, no more subtrees are needed.

Another method that is different from previous work in assigning votes to all the copies of a data item is the randomized voting algorithm [37], which is based on the concept of simulated annealing and tries to maximize overall availability. Simulated annealing is a term used in statistical mechanics to describe the process by which a substance is crystallized by first melting it at a high temperature and then cooling it gradually by lowering the temperature in successive stages, and allowing it to reach a steady state in each stage. Let a voting assignment be a tuple \((v_1, v_2, ..., v_n)\), where \( v_i \) is the number of votes assigned to a site \( i \). The algorithm starts from the initial state \((1,1,...,1)\) if \( n \) is odd; otherwise it adds one more vote to the first site \((2,1,...,1)\). For each pair of sites \( v_i \) and \( v_j \), the algorithm does four possible state changes: (1) add 1 to both \( v_i \) and \( v_j \), (2) add -1 to both \( v_i \) and \( v_j \), (3) add 1 to \( v_i \) and -1 to \( v_j \), (4) add -1 to \( v_i \) and 1 to \( v_j \). In order to gain the maximum availability a probabilistic method must be used to decide whether or not to stay on the same state when the
new state is worse. If the next state is better, the current state must be changed to the new state. The process stops when a certain number of iterations have been completed and there have been no change in state of the best solution.

Most syntactic-pessimistic strategies deal with individual read and write operations, whereas class conflict analysis [59] divides transactions into classes by their readsets and writesets. The readset (writeset) of a transaction class is the union of the readsets (writesets) of all its member transactions. Two classes conflict if the readset of one intersets the writeset of the other, indicating a potential read-write conflict between member transactions of the conflicting classes. When a failure occurs, each partition uses a class conflict graph to detect conflicts, then deletes a minimal set of classes to resolve potential nonserializable execution. Thus, the potential conflicts between member transactions of two conflicting classes can be avoided. The major overhead of this approach includes the cost of inconsistency repair and opportunity lost.

2.3.3 Semantic-Optimistic Strategies

Semantics of Transactions

Log transformations [12] use the semantics of transactions in defining serializability. The semantics of a transaction and the semantic properties of each pair of transactions are predefined to avoid unnecessary rollback and redo transactions. They use properties such as commutativity ($T_iT_j = T_jT_i$) and overwriting ($T_iT_j = T_j$) to reduce the size of a merge log, which is the union of rollback logs and redo logs. This log transformation can be implemented locally, so that each site can independently integrate this transformation into its transactions, regardless of what other sites are
In some situations, merely read-only transactions are needed and read operations do not change the database state, so the requirement of consistency may be relaxed. Garcia and Weiderhold [26] divided transactions into two classes based on their consistency requirement: strongly consistent transactions and weakly consistent transactions. A strongly consistent transaction is processed in the normal fashion: its execution must be serializable with respect to other updates and strongly consistent transactions. A weakly consistent transaction must see a consistent database state (the result of a serializable execution of update transactions), but its execution need not be serializable with respect to other read-only transactions. The choice of a consistency level for a read-only transaction depends on how the information returned by the transaction is used.

The other approach used by a database administrator to integrate divergent databases is a data-patch program [26]. When partition occurs, the program uses a set of rules to integrate databases. The set of rules declares what sorts of compensating or correcting actions should be taken when the values of the current data items or relations (image relations) have diverged from the expected final values (plan relations). The program defines its correctness by its final database state, not by serializability.

2.3.4 Semantic-Pessimistic Strategies

Semantics of Database

*General quorum consensus* proposed by Herlihy [30] uses type-specific properties of an abstract data type to increase data availability. Each data item is viewed as an instance of an abstract data type. An abstract data type defines the set of
operations supported by a data item of that type, and two type-specific correctness criteria: a serial specification and a behavioral specification. A serial specification describes a sequence of operations that are allowed on data items of a given type. A behavioral specification describes the conflicts between operations that limit concurrency. For example, a dequeue operation conflicts with another dequeue or enqueue operation, because these two operations affect the final value returned by a dequeue operation.

Later Herlihy [31] extended general quorum consensus to other objects such as sets and directories, and permitted quorums to be changed dynamically. Through these dynamical changes, a wider range of availability and more flexible on-the-fly reconfiguration can be achieved. The quorum intersection constraints are derived from the data type’s algebraic structure analysis. A more deliberate quorum style includes an initial quorum and a final quorum, which replace the single quorum for each operation in conventional quorum consensus. These changes plus logs and timestamp techniques reduce the constrains on quorum intersection and increase the range of realizable availability.

Semantics of Transactions

In addition to using the semantics of the database to define the correctness criteria, we can use the semantics of executed transactions. Transaction classes [26] divide transactions into a collection of disjoint classes such that the transactions that belong to the same class are compatible and can interleave arbitrarily, whereas the transactions that belong to different classes are incompatible and can not interleave at all.
Breakpoint [25] [43] further exploits the semantic information available about transactions to allow controlled nonserializable interleaving execution. No transaction classes are used, and interleaving is specified by inserting a command called “breakpoint” at the end of each step of a transaction. Associated with each breakpoint is a set of transaction types that are allowed to interrupt the transaction at that point.

Most nonserializable interleaving transaction execution uses the commutativity property of operations. The commutativity property of two operations ensures that the exchange of the execution order of two operations does not affect the final database state. In 1992, in addition to the commutativity property, Badrinath [8] used the recoverability property to decrease the delay involved in processing noncommutative operations while avoiding cascading aborts. The result reduced response time and increased throughput. An operation $o_2$ is recoverable relative to an operation $o_1$ if $o_2$ is executed immediately following $o_1$. The value returned by $o_2$, the observable semantics of $o_2$, is the same whether or not $o_1$ is executed immediately before $o_2$.

When an invoked operation is recoverable with respect to an uncommitted operation, the invoked operation can be executed by forcing a commit dependency between the invoked operation and the uncommitted operation. This commit dependency only affects the order in which the operations should commit, if both operations commit. If either operation aborts, the other can still commit, thus avoiding cascading aborts. To ensure the serializability of transactions, the recoverability relationship between transactions must be acyclic.

From the previous review, it is clear that quorum voting schemes (coterie) have dominated recent research on transaction processing. The reason is their simplicity and easy implementation. In particular, in the past five years researchers have imposed a logical structure on the underlying network. This new logical structure
concept provides both system designers and users of a network system more freedom in choosing an appropriate model based on available network information. Among those logical structures, the logical tree is the most popular, because it is concordant with the properties of the real world that most systems intend to model, such as the file and directory relations in a file system and the hierarchical structure of a computer network.

Recently, Agrawal and El Abbadi have generalized the tree quorum algorithm for managing replicated data. All copies of each data item are organized into a tree of height $h$ and degree $d$. A read quorum $q_r = (l_r, w_r)$ of length $l_r$ and width $w_r$ is constructed by selecting the root and $w$ children of the root, and for each selected child, $w$ of its children, and so on, to depth $l$. If some node is inaccessible at depth $h'$ from the root while constructing this tree quorum, the node is replaced recursively by $w$ tree quorums of height $l - h'$ starting from the children of the inaccessible node.

A write quorum $q_w = (l_w, w_w)$ is constructed in the same way. To ensure the quorum intersection property, it is required that $l_r + l_w > h$ and $w_r + w_w > d$. If version numbers are used, $2 \times l_w > h$ and $2 \times w_w > d$ are the requirements.

Unfortunately, their tree quorum algorithm encounters the same drawback suffered by most quorum protocols: the reduced cost of write operations is compromised by the increased cost of read operations. To read a data item, a quorum of copies must be read to obtain the most recent value of a data item, instead of one single copy as in the available copy or the token protocol.

Our algorithm does not have this drawback and requires only a single piggybacked value $read$ for a read request and a quorum of $writes$ for a update request. In fact, we further reduce the message cost and the size of quorums to a constant in the lightly loaded system. The upper bound of our quorum size is the same as the
tree quorum algorithm [2] in a heavily loaded system: \( \log n \) for a failure-free network and \( [(n + 1)/2] \) for a partitioned network.
CHAPTER 3

MODEL

Let a time section be a certain amount of time such that all transactions submitted in one time section are committed before the next section starts. In this way, starvation of restarting transactions can be avoided. Later we will discuss methods for guaranteeing the existence of a time section. Four components at each site are responsible for transaction processing: transaction manager, data manager, concurrency controller, and replica controller.

A user communicates with a database system indirectly by submitting a transaction to the transaction manager on that site. A transaction is a sequence of read and write operations on a set of data items which the transaction manager must ensure are executed atomically. No partial effects of a transaction will be reflected in the database state. In fact, when a transaction commits, the actual updates of data items are performed by the data manager. In a replicated database system, both site/communication failures and data replication are masked by a replica control protocol and a concurrency control protocol and with their appropriate cooperation one-copy serializability generally can be satisfied.

The only way that each pair of nodes can communicate with each other is through message exchanges. Messages are received in the order in which they were sent and no message is lost, duplicated or collapsed during delivery. A concurrency control message typically contains a timestamp, the identification of the sending node, and the most recent value of a data item. The destination node uses the timestamp.
and the sender's identification to linearly order the messages received [41]. The most recent value of a data item is also piggybacked upon the concurrency messages in such a way that a read operation can be implemented when all the member nodes of a read quorum grant their permissions, and write operations of all the members of a write quorum can be performed when the release message from the node doing the last update is received. Thus, no extra message is required to send an update value to the subsequent transactions that need the information.

A logical tree structure is assumed to be imposed on the set of all copies of each data item. This structure can be either the physical network on which the copies of a data item reside or an abstract relation imposed on the nodes containing the copies. System configuration changes are reflected on this logical tree structure directly by having all inaccessible nodes be failed nodes. Thus, all nonfailed nodes in the underlying logical tree represent all accessible nodes. The nodes and communication links in the tree can fail and restart again later. However, our algorithm requires that no failure occurs while a new coterie is being computed.

Following Harel and Tarjan, we assume that each node has the power of a random access machine (RAM) [5] in which each word contains \(O(\log n)\) bits and each operation on a word or pair of words takes \(O(1)\) time. On such a machine, it is possible to find the nearest common ancestor of any two nodes in \(O(1)\) time [29].
For each data item \( x \) the replica controller keeps the largest timestamp of any read and write operations on \( x \) in \( RW\text{-timestamp}(x) \), as well as the \textit{NODE number} and the \textit{nca quorum} of the node doing the last update on \( x \) in \( x\text{node}(x) \) and \( x\text{nca}(x) \). The \textit{NODE number} is the number of the corresponding node in the smallest complete binary tree containing the logical tree \( T \), under the conventional ordering. The \textit{nca quorum} of a node is a tree quorum containing the nearest common ancestors of the node's youngest competing descendant with all the current competing nodes. These values will be kept up-to-date by a background status transaction at the end of each time section.

As mentioned before the user interacts with the transaction manager indirectly by submitting a transaction. For each member operation \( i \) of the transaction, the transaction manager sends the local time \( t \) and the data item \( x \) to the local replica controller. The replica controller compares \( t \) with \( RW\text{-timestamp}(x) \) and, if \( t \) is smaller than \( RW\text{-timestamp}(x) \), the local clock is updated to \( RW\text{-timestamp}(x) + 1 \). After this comparison, the \textit{NODE number} is sent to \( x\text{node}(x) \). If \( x\text{node}(x) \) has failed, the first nonfailed node of \( x\text{nca}(x) \) is chosen as the backup node. After the longest period of time that any message can reach its destination in a network, \( x\text{node}(x) \) will return back a list of \textit{NODE numbers}, \textit{NODES}, of all the nodes competing for access to \( x \) at the same time. Using \textit{NODES} the replica controller computes a \textit{nca quorum} for \( i \).

To ensure the correct value of \( x \), a quorum of permissions must be collected
before it can be performed. If a quorum is collected, the local concurrency controller informs the transaction manager of the success of it and does a temporary update on x if op equals write. After all member operations of the transaction succeed, the transaction commits and the permanent updates on all the members of a write quorum can be performed when the release messages with a piggybacked update value and a success flag arrive. If one of the member operations fails, the temporary update will be discarded. Since the most recent updates are piggybacked on the concurrency control messages, the write operations of all members of a write quorum can be performed as soon as the release message coming from the node doing the last update is received. Similarly, a read request will be implemented when the permission message coming from the node doing the last update arrives. As a result, the new algorithm only requires a single read for a read request and a quorum of writes for a write request. The algorithm reduces the size of a write quorum without increasing the size of its counterpart read quorum, which is needed in conventional quorum consensus.

Since a quorum of permissions must be collected before an operation can be performed, and any pair of nca quorums must have at least one member in common, no two operations can access x at the same time. Thus, nca quorums safeguard one-copy serializability of each data item. Due to the importance of the nca quorums, we will introduce the algorithm used to compute the nca quorum for each operation in the next section, before presenting the details of the transaction processing itself.
CHAPTER 5

THE NCA QUORUM

For each request, the nca quorum algorithm computes a quorum that is a set of nodes consisting of the nearest common ancestors of the requesting node’s youngest competing descendant with all the requesting nodes. Before the nca quorum algorithm can be invoked, a preprocessing step is required. In the preprocessing step, a set of binary integer ID values, which represent an abstract relation of the nodes in a binary tree $T$, is assigned to the nodes in $T$. By using these binary integers, the nearest common ancestor of two nodes can be computed in constant time using the exclusive-or operation. This abstract relation of the nodes in a binary tree is first proposed by Harel and Tarjarn [2], and generalized by Lin and Jacob for $d$-degree trees in [42].

5.1 Nca Quorums for Complete Binary Trees

Let $U$ be the set of nodes in the distributed system. A coterie $C$ is a set of subsets of $U$ that satisfies the following conditions:

(i). If $N \in C$, then $N$ is a nonempty subset of $U$.

(ii) (Intersection property). If $G, H \in C$, then $G \cap H \neq \emptyset$.

(iii) (Minimality). There exist no $G, H \in C$ such that $G \subseteq H$.

Each element of the coterie is called a quorum. Quorum consensus algorithms are based on the three coterie properties. When a node wants to enter its critical
section, it determines a quorum, all of whose members must grant permission before
the node can enter the critical section. In some algorithms, this quorum is static; in
other algorithms, the appropriate quorum is determined at the time the node decides
to enter its critical section. The intersection property of a coterie requires that any
pair of quorums have at least one element in common. This common node acts as
arbiter, granting permission to exactly one node at a time. The minimality property
is not necessary for mutual exclusion, but provides greater efficiency [2].

In this paper we use conventional tree terminology, with the level of the root
being 0. The nca quorum algorithm computes the nearest common ancestor \( \text{nca}(x, y) \)
for any nodes \( x \) and \( y \) in a distributed system. These nearest common ancestors can be
used to form a quorum \( Q(p) \) for each node \( p \). An obvious candidate for \( Q(p) \) is the set
\( \{p\} \cup \{\text{nca}(p, x) | x \text{ competes with } p\} \). A problem arises with this definition, however,
when \( q \) is a descendent of \( p \). In this case, it is possible that \( Q(p) \subset Q(q) \), violating the
minimality property for coteries. These considerations lead to the following definition.

**Definition:** If \( p \) is a node in a tree \( T \), then the nca quorum \( Q(p) \) is

\[
\{q\} \cup \{\text{nca}(q, x) | x \text{ competes with } q\},
\]

where \( q \) is the youngest descendent of \( p \) that is competing with \( p \).

As a result of this definition, the nca quorum for \( p \) will be the same as the nca
quorum of its youngest competing descendent \( q \).

5.1.1 The Preprocessing Step

The preprocessing step imposes an abstract relation, discovered by Harel and
Tarjan [29], on the nodes of a binary tree \( T \). This relation is represented by a set of
binary integers in such a way that the nearest common ancestor of two nodes can be
computed in constant time using the exclusive-or operation.

In the following algorithm this abstract relation is represented by the ID values of the nodes. These ID values are different from the NODE numbers, which are assumed to have been assigned before processing and which are the same as the numbers of the corresponding nodes in the smallest complete binary tree containing T under the conventional ordering. If x is a node, then we will let \( x_{id} \) denote the ID value of x, and \( x_{node} \) denote the NODE number of x.

In Figure 5.1, the ID values of the nodes with height \( h \) are, from left to right, \( i \times 2^h \), for \( i = 1, 3, 5, 7 \ldots \). From these values, Harel and Tarjan also derived the following facts.

**Lemma 5.1:** The descendants of the node x are those nodes with ID values in the range from \( x_{id} - 2^{|\log x| - |\log x_{node}|} + 1 \) to \( x_{id} + 2^{|\log x| - |\log x_{node}|} - 1 \).

**Lemma 5.2:** If x is a node and y is an ancestor of x with height \( h \), then \( y_{id} \) is \( 2^{h+1} \times \lceil \frac{x_{id}}{2^h + 1} \rceil + 2^h \).

**Lemma 5.3:** If x and y are two unrelated nodes, then the height of \( nca(x, y) \) is \( \lfloor \log (x_{id} \oplus y_{id}) \rfloor \), \( x_{id} \oplus y_{id} \) is the integer whose binary representation is the bitwise exclusive-or of the binary representations of \( x_{id} \) and \( y_{id} \).

We illustrate Lemmas 5.2 and 5.3 by considering two examples from Figure 5.1. First, we compute the nearest common ancestor \( nca(12, 13) \) of nodes 12 and 13. Since \( \text{ID}[12] = 9 \) and \( \text{ID}[13] = 11 \), Lemma 5.3 gives us that the height of \( nca(12, 13) \) is \( \lfloor \log_2(9 \oplus 11) \rfloor = \lfloor \log_2 2 \rfloor = 1 \). From Lemma 5.2, we have that \( \text{ID}(nca(12, 13)) \) is \( 2^2 \times \lfloor \frac{9}{2^2} \rfloor + 2^1 = 8 + 2 = 10 \). Since \( \text{NODE}[10] = 6 \), we have that \( nca(12, 13) = 6 \). Similarly, the nearest common ancestor \( nca(12, 14) \) of nodes 12 and 14 is 3. \( \text{ID}[12] = \)
9 and ID[14] = 13, so the height of nca(12,14) is \[\log_2(9 \oplus 13) = \log_2 4 \] = 2, the ID of nca(12,14) is \[2^3 \times \left\lfloor \frac{3}{2^2} \right\rfloor + 2^2 = 8 + 4 = 12,\] and NODE[12] = 3.

Figure 5.1: A complete binary tree of 15 nodes with ID values shown in parentheses.

These lemmas provide us the tools to compute the nearest common ancestor of any pair of nodes in the tree network.

5.1.2 Nca Quorums for Failure-Free Binary Trees

In our algorithm we assume that a list of the ID values of competing nodes is available to each participating node when the nca quorum algorithm starts.

In NCA_QUORUM, x is the current requesting node, IDS is a list of ID values of competing nodes, and n is the number of nodes in the underlying network. When x wants to enter its critical section, it executes NCA_QUORUM to get permissions from all elements of its own nca quorum. If d is the youngest competing descendant of x, then the algorithm computes the nearest common ancestors of d with all nodes in IDS. At each step in the algorithm, nca(x, y) is computed for some y in the list IDS, and the tree is pruned by deleting a subtree at nca(x, y). When x is not an
ancestor of \( y \), the subtree that does not contain \( x \) is deleted; otherwise, the subtree that does not contain \( y \) is deleted. Every competing node \( p \) that is in the pruned part may safely be deleted from IDS, since in computing \( nca(x, y) \) we have computed the appropriate nearest common ancestor for \( p \) also. Therefore, no nearest common ancestor will be computed twice. Since there are at most \( \lceil \log n \rceil - 1 \) ancestors of any node in a complete binary tree, the maximum number of elements in any nca quorum will be \( \lceil \log n \rceil \). Therefore, each node wanting to enter its critical section must send at most \( O(\log n) \) messages in the failure-free network. Only a constant number of messages are required in the best case.

Complete code for the \textit{NCA\_QUORUM} algorithm can be found in the last section of this chapter.

\textbf{Theorem 5.1:} The size of an nca quorum is \( \lceil \log n \rceil \) in the worst case, and is 1 in the best case, for a failure-free complete binary tree.

\textbf{Proof:} The nca quorum for a node \( x \) is contained in a path from the root to one of the leaves of the tree. Since the number of nodes on any path from the root to a leaf in a complete binary tree is at most \( \lceil \log n \rceil \), the size of the nca quorum is at most \( \lceil \log n \rceil \). However, if \( x \) is the only node that wants to enter its critical section, so that there is no competing node, then \( x \) is the only element in its nca quorum.

\textbf{Theorem 5.2:} If all active nodes want to enter their critical sections simultaneously, then the computation complexity of \textit{NCA\_QUORUM} for a failure-free complete binary tree is \( O(n \log n) \) in the worst case, and \( O(n) \) in the best case. When only a single node wants to enter its critical section, then the computation complexity of \textit{NCA\_QUORUM} is \( O(1) \).
Proof: Since a nearest common ancestor can be computed in $O(1)$ time and there are at most $\log n$ ancestors for any node in a complete binary tree, the time required to compute all the nearest common ancestors is $O(\log n)$ in the worst case. After computing a nearest common ancestor, the nca quorum algorithm scans IDS to prune the appropriate subtree. Since the pruned subtree is larger the nearer the computed nearest common ancestor is to the root, the size of the resulting IDS is smallest when the computed nearest common ancestor is the root. Hence, the cost of subtree pruning in the worst case is $n + (n - 2) + (n - 2 - 4) + ... + 1 \leq n \log n$, when at each stage the nearest common ancestor computed is the parent of the previous one. In contrast, the best case cost is $n + n/2 + n/4 + ... + 1 \leq 2n$, when at each stage the nearest common ancestor computed is the child of the previous one. So the computation complexity of the nca algorithm for the complete binary tree in the failure free network is at most $O(\log n + n \log n) = O(n \log n)$, and at least $O(\log n + n) = O(n)$ per request in the case when all the nodes want to enter their critical sections.

In the case where only a single node wants to enter its critical section, it is the only element of its nca quorum, so the cost of computing the nca quorum is $O(1)$.

5.1.3 Correctness of the Algorithm

The nca quorum algorithm requires each requesting node to get permissions from all the nodes in its nca quorum before entering its critical section. Since the nca quorums of every pair of competing nodes have at least one node in common, this common node can serve as an arbiter between them. The arbiter determines the request priority by using Lamport’s logical time idea[41], the node with earlier local time receiving higher priority. If both nodes have same local time, the one with smaller NODE number gets higher priority. Since NODE numbers are unique,
only one node will get permission from the arbiter, so only one node will collect the permissions from all members of its nca quorum. This node is the only one able to enter its critical section.

**Theorem 5.3:** The union of the nca quorums of all competing nodes forms a coterie that satisfies the intersection and minimality properties.

**Proof:** The theorem is an easy consequence of the nca quorum construction.

5.2 Nca Quorums for Partitioned Binary Trees

In a binary tree, each node is critical because when it fails the network will be partitioned and, consequently, the `NCA_QUORUM` algorithm will not work. The `PARTITION_NCA_QUORUM` algorithm solves this problem in the following way: when `nca(x, y)` fails, all competing nodes not in the same subtree as `x` are grouped into a list `isolated_ids`, and the inner while loop of the algorithm computes an nca quorum for each individual partition. The new nca quorum for `x` is constructed so that if `nca(x, y)` fails, it will be replaced in the nca quorum of `x` by an nca quorum for the connected component of `isolated_ids` \( \cap S \) that contains `y`, where `S` is the subtree of `nca(x, y)` that contains `y`. Hence the new nca quorum for `x` will be the union of nca quorums of isolated partitions of the tree. Care must be taken to ensure that only one quorum for each partition is included in the nca quorum for `x`.

Code for the `PARTITION_NCA_QUORUM` algorithm can be found in the last section of this chapter.

**Theorem 5.4:** Suppose `p` and `q` are unrelated nodes, such that \( x = nca(p, q) \in Q(p), \text{height}(x) \geq 1 \), and `x` fails. If `Q'(p)` is the new nca quorum for `p` after the
replacement of \( x \) by the PARTITION\_NCA\_QUORUM algorithm, then \(|Q'(p)| \geq |Q(p)|\).

**Proof:** When \( x \) fails, it is replaced in \( Q(p) \) by an nca quorum for the connected component \( T \) of isolated ids \( S \) that contains \( q \), where \( S \) is the subtree of \( nca(p, q) \) that contains \( q \). Since \( x \) is the nearest common ancestor of two unrelated nodes, \( T \) must contain at least the node \( q \). Therefore, \(|Q'(p)| \geq |Q(p)|\). When \( T \) contains more than one node, then \(|Q'(p)| > |Q(p)|\).

**Theorem 5.5:** If \( n \) is the number of nodes in a partitioned binary tree network, then the size of each nca quorum is \( \lceil (n + 1)/2 \rceil \) in the worst case, and 1 in the best case.

**Proof:** By Theorem 5.4, if \( x \) is a nearest common ancestor of two unrelated nodes such that \( \text{height}(x) \geq 1 \), then when \( x \) fails, all the quorums originally containing \( x \) may increase in size and will not decrease in size. Hence for any maximal sized quorum, there is an equivalent maximal sized quorum that contains elements of height 0 and 1 only.

Furthermore, when a node of height 1 fails, it is replaced in each quorum containing it by one of its children. Hence for any maximal sized quorum, there is an equivalent maximal sized quorum containing only leaf nodes. Therefore, the largest possible quorum is the set of all leaf nodes, which has \( \lceil (n + 1)/2 \rceil \) members.

In the best case, there is only a single node wanting to enter its critical section, so the only quorum consists of that node and no other.

**Theorem 5.6:** The algorithm PARTITION\_NCA\_QUORUM has computational complexity \( O(n^2 \log n) \) in the worst case, and \( O(1) \) in the best case.
Proof: By Theorem 5.4, if a nearest common ancestor $x$ of two unrelated nodes fails and $\text{height}(x) \geq 1$, then the new nca quorum generated by $\text{PARTITION-NCA-QUORUM}$ will be at least as large as the old one containing $x$. It follows that a quorum containing nodes of height 0 or 1 only will have maximal size. In the inner while loop of $\text{PARTITION-NCA-QUORUM}$, an nca quorum for each partition is computed as follows: after computing a nearest common ancestor, the algorithm prunes the appropriate subtree by scanning the IDS and grouping isolated nodes into the list $\text{isolated ids}$, if the computed nearest common ancestor failed. Otherwise, the appropriate subtree is deleted. Since a nearest common ancestor can be computed in $O(1)$ time and there are at most $\lceil(n + 1)/2\rceil$ partitions in a maximal size nca quorum, each one of them can have at most $\log n$ ancestors in a complete binary tree, thus in the worst cast the time required to compute all the nearest common ancestors is $O((n+1)/2 \log n)$. After computing a nearest common ancestor, the nca quorum algorithm scans IDS to prune the appropriate subtree. Since the pruned subtree is larger the nearest common ancestor is to the root, the size of the resulting IDS is smallest when the computed nearest common ancestor is the root. Hence, the cost of subtree pruning for each partition in the worst case is

$$\frac{n+1}{2} + \left(\frac{n+1}{2} - 2\right) + \left(\frac{n+1}{2} - 4\right) + \ldots + \left(\frac{n+1}{2} - 2^{\lceil \log n \rceil - 1}\right) \leq \frac{n+1}{2} \log n,$$

when at each stage the nearest common ancestor computed is the parent of the previous one.

There are at most $\lceil(n + 1)/2\rceil$ partitions in a maximal size nca quorum, thus the cost of subtree pruning in the worst case is $O((n+1)/2 \log n)$. So the computation complexity of the nca algorithm for a partitioned binary tree network is at most $O((n+1)/2 \log n) + (n+1)/2 \log n) = O((n+1)^2 \log n) = O(n^2 \log n)$.

In the case where only a single node wants to enter its critical section, it is the only element of its nca quorum, so the cost of computing the nca quorum is $O(1)$. 
5.3 Nca Quorums for Trees of Degree $d$

In this section we further generalize the nca quorum algorithm to trees of degree $d$. In this case the algorithm is simpler and more straightforward than that for the complete binary tree. The only new requirement is that each word of a random access machine hold at least $\lceil \log_d n \rceil (\lceil \log_2 d \rceil + 1)$ bits. The NODE numbers of nodes in the tree are assigned by numbering the root 1, and then going down the tree numbering from left to right on each level.

5.3.1 The Preprocessing Step for Trees of Degree $d$

Except for the root of the tree, whose ID value is 1, the ID values of nodes of height $h$ are assigned by the following rule: If $p$ is a node of height $h + 1$, $\text{base} = 2^{\lceil \log_d d \rceil + 1}$, and $\text{dif} = \text{base}^{H - \text{depth} + 1}$, where $H$ is the height of the tree and $\text{depth}$ is the depth of the children of $p$, then the ID values of the children of $p$ are $p_{id} + \text{dif}$, $p_{id} + 2 \times \text{dif}$, $p_{id} + 3 \times \text{dif}$, $\ldots$, $p_{id} + d \times \text{dif}$ from left to right. For trees of degree $d$, we have the following lemmas analogous to Lemmas 5.1-5.3.

**Lemma 5.4:** If $x$ and $y$ are nodes in the system, $h = \lceil \log_{\text{base}}(x_{id} \oplus y_{id}) \rceil$, and $\text{dif} = \text{base}^{h+1}$, then $\text{nca}(x, y)$ is

$$\text{NODE}[\text{dif} \times \lceil \frac{x_{id}}{\text{dif}} \rceil + 1].$$

**Lemma 5.5:** If $x$ and $y$ are nodes in the system and $h = \lceil \log_{\text{base}}(x_{id} \oplus y_{id}) \rceil$, then $x$ is in the subtree rooted on the $i^{th}$ child of $\text{nca}(x, y)$, where

$$i = \left\lfloor \frac{(x_{id} - \text{nca}(x, y)_{id})}{\text{base}^h} \right\rfloor$$
In Figure 5.2, as an example of Lemmas 5.4, the nearest common ancestor of nodes 8 and 9 is 3. We compute this answer as follows: ID[8] = 37, ID[9] = 41, h = \lfloor \log_4(37 \oplus 41) \rfloor = \lfloor \log_4 12 \rfloor = 1, \text{diff} = 4^{1+1} = 16, \text{NODE}[16 \cdot \left\lfloor \frac{37}{16} \right\rfloor + 1] = \text{NODE}[33] = 3.

**Lemma 5.6:** If x and y are nodes, then x and y are not in the same subtree of nca(x, y) if

\[ \text{nca}(x, y)_{id} < y_{id} < \text{nca}(x, y)_{id} + i \cdot \text{base}^h \]

or

\[ \text{nca}(x, y)_{id} + (i + 1) \cdot \text{base}^h \leq y_{id} \leq \text{nca}(x, y)_{id} + \text{base}^{h+1} - 1 \]

where i and h are as in Lemma 4.

5.3.2 Nca Quorums for Trees of Degree d

Merging these facts with the same procedure used in the complete binary tree case, the processing of NCA \_ QUORUM.D is similar to that of the complete binary tree. The major differences are the method of computing the nearest common ancestors and the way that subtrees are pruned. When a node x wants to enter its
critical section, it calls \texttt{NCA\_QUORUM\_D} to get its nca quorum by computing all nearest common ancestors of its youngest descendant with other current competing nodes. After each nearest common ancestor is computed, we can safely delete nodes in the subtrees that do not contain \( x \). Because no nearest common ancestor will be computed twice, the maximal number of ancestors of a node in the tree is \( \log_d n \), and the maximum number of messages sent per request is \( O(\log_d n) \) for the failure-free network.

Complete code for the \texttt{NCA\_QUORUM\_D} algorithm can be found in the last section of this chapter.

**Theorem 5.7:** The worst size of an nca quorum is \( \log_d n \), and 1 in the best cast for a failure-free \( d \)-degree network.

**Proof:** The construction of an nca quorum of a node \( x \) contains only the nodes on some path, containing \( x \), from the root to a leaf. Since there are at most \( \log_d n \) nodes on this path, the size of the nca quorum for \( x \) is \( \log_d n \) at worst. When only a single node wants to enter its critical section, the nca quorum contains one element only.

**Theorem 5.8:** The algorithm \texttt{NCA\_QUORUM\_D} has computational complexity \( O(n \log_d n) \) in the worst case, and \( O(n) \) in the best case, if all the active nodes want to enter their critical sections. If fewer nodes want to enter their critical sections, then the best case will be \( O(1) \).

**Proof:** Since a nearest common ancestor can be computed in \( O(1) \) time and there are at most \( \log_d n \) ancestors for any node in a \( d \)-degree tree, the time required to compute all the nearest common ancestors is \( O(\log_d n) \) in the worst case. After computing a nearest common ancestor, the nca quorum algorithm scans IDS to prune
the appropriate subtree. Since the pruned subtree is larger the nearer the computed nearest common ancestor is to the root, the size of the resulting IDS is smallest when the computed nearest common ancestor is the root. Hence, the cost of subtree pruning in the worst case is $n + (n - d) + (n - d - d^2) + \ldots + 1 \leq n \log dn$, where at each stage the nearest common ancestor computed is the parent of the previous one.

In contrast, the best case cost is $n + \frac{d^{H-1}}{d-1} + \frac{d^{H-2}}{d-1} + \ldots + 0 \leq 2n$ where $H$ is the height of the tree, when at each stage the nearest common ancestor computed is the child of the previous one. So the computation complexity of the nca algorithm for the $d$-degree tree in the failure free network is at most $O(\log dn + n \log d n) = O(n \log d n)$, and at least $O(\log dn + n) = O(n)$ per request in the case when all the nodes want to enter their critical sections.

In the case where only a single node wants to enter its critical section, it is the only element of its nca quorum, so the cost of computing the nca quorum is $O(1)$.

5.4 NCA Quorum for Partitioned Trees of Degree $d$

For a partitioned $d$-degree tree network, the $NCA\_QUORUM\_D$ can be modified in the same way as $PARTITION\_NCA\_QUORUM$ by adding an inner loop that groups isolated nodes into a separate list $isolate\_ids$, and computes nca quorum for each partition recursively until the list $isolate\_ids$ becomes empty. In this way the new nca quorum of node $x$ is constructed by replacing each failed $nca(x, y)$ with the union of nca quorums, each of which is a nca quorum of one of the competing nodes in a separated subtree. For example, in Figure 5.2 when nodes 2, 3, and 8 want to enter their critical sections and node 1 has failed, the nca quorum for node 2 will be $\{2, 3, 8\}$, where node 1, the failed $nca(2, 3)$, is replaced by the nca quorum $\{3, 8\}$. 
In the case where all active nodes want to enter their critical sections, unlike the tree quorum algorithm, the nca quorum algorithm does not restrict the height of a tree by the number of failures that can be tolerated. However, the extra message cost introduced by a failed node $x$ is the same as that of the tree quorum algorithm. If $k$ is the height of the tree, and a node $x$ fails at depth $i$, then the size of any nca quorum containing $x$ increases from $k+1$ to $i + d(k - i)$. As a consequence, if $t$ nodes have failed, then maximal sized nca quorums will result when the failed nodes are the $t$ nodes with NODE numbers 1, 2, \ldots, $t$. In that case, when all the active nodes are competing, then the sizes of their nca quorums will be at least $d^i + 1 \times (k - i)$, where $i$ is the depth of the subtree of failed nodes. $t \leq (d^i + 1 - 1)/(d - 1)$, has maximal size.

**Theorem 5.9:** Suppose $p$ and $q$ are unrelated nodes such that $x = nca(p, q) \in Q(p)$, $\text{height}(x) \geq 1$, and $x$ fails. If $Q'(p)$ is the new nca quorum for $p$ after the replacement of $x$ by the partitioned $NCA\_QUORUM\_D$ algorithm, then $|Q'(p)| \geq |Q(p)|$.

**Proof:** The proof is the same as that of Theorem 5.4, so we omit it.

**Theorem 5.10:** If $n$ is the number of nodes in a partitioned $d$-degree tree $T$, then the size of each nca quorum is $d^k$ in the worst case, and 1 in the best case, where $k$ is the depth of $T$.

**Proof:** As in Theorem 5.5, the worst case occurs when the quorum consists of all the leaf nodes. Hence, in the worst case, a quorum will contain $d^k$ nodes.

In the best case, there is only a single node wanting to enter its critical section, so the nca quorum for that node consists only of itself.
**Theorem 5.11:** The nca quorum algorithm for a partitioned $d$-degree tree $T$ has computational complexity $O(d^{2k} \log n)$ in the worst case, and $O(1)$ in the best case, where $k$ is the depth of $T$.

**Proof:** By Theorem 5.10, the nca quorum $Q$ containing all leaves has maximal size. Each element of the quorum is a separate partition, and all computed nearest common ancestors are failed nodes. In the inner while loop of partitioned $\text{NCA-QUORUM-D}$, after computing a nearest common ancestor, subtrees are pruned by scanning IDS and grouping the appropriate subtrees into a separate list $\text{isolated ids}$, until IDS becomes empty. Since a nearest common ancestor can be computed in $O(1)$ time and there are at most $\log_d n$ ancestors for each element of $Q$ in a $d$-degree tree, the time required to compute all the nearest common ancestors is $O(d^k \log_d n)$ in the worst case. Since the pruned subtrees is larger the nearer the computed nearest common ancestor is to the root, the size of the resulting IDS is smallest when the computed nearest common ancestor is to the root. Hence, the cost of subtree pruning for each element of $Q$ in the worst case is $d^k + (d^k - d) + (d^k - d - d^2) + (d^k - d - d^2 - d^3) + \ldots + 1 <= d^k \log_d n$, when at each stage the nearest common ancestor computed is the parent of the previous one. The cost of subtree pruning for $Q$ is at most $d^{2k} \log_d n$. Therefore, the computation complexity of partitioned $\text{NCA-QUORUM-D}$ is at most $O(d^k \log_d n + d^{2k} \log_d n) = O(d^{2k} \log_d n)$ in the worst case.

In the case where only a single node wants to enter its critical section, it is the only element of its nca quorum, so the cost of computing the nca quorum is $O(1)$.

For a arbitrary tree, the reader can refer to Harel-Tarjan[29] which transforms a arbitrary tree into a complete binary tree. We can then use nca quorum algorithm on this complete binary tree.
5.5 Analysis

In this section, we compare the performance of the nca quorum algorithm with that of the tree quorum algorithm (mutual exclusion) on the complete binary trees with 3, 7, 15, 31, 63 and 127 nodes.

<table>
<thead>
<tr>
<th>percentage</th>
<th>7</th>
<th>15</th>
<th>31</th>
<th>63</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>1</td>
<td>2.0</td>
<td>2.558</td>
<td>3.73</td>
<td>4.780</td>
</tr>
<tr>
<td>20%</td>
<td>1</td>
<td>2.533</td>
<td>3.544</td>
<td>4.716</td>
<td>5.579</td>
</tr>
<tr>
<td>30%</td>
<td>2</td>
<td>2.875</td>
<td>3.995</td>
<td>5.131</td>
<td>6.013</td>
</tr>
<tr>
<td>50%</td>
<td>2.725</td>
<td>3.694</td>
<td>4.622</td>
<td>5.622</td>
<td>6.639</td>
</tr>
<tr>
<td>60%</td>
<td>2.717</td>
<td>3.725</td>
<td>4.776</td>
<td>5.793</td>
<td>6.771</td>
</tr>
<tr>
<td>70%</td>
<td>2.917</td>
<td>3.838</td>
<td>4.864</td>
<td>5.880</td>
<td>6.850</td>
</tr>
<tr>
<td>80%</td>
<td>3</td>
<td>3.962</td>
<td>4.951</td>
<td>5.953</td>
<td>6.961</td>
</tr>
<tr>
<td>90%</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5.998</td>
<td>6.992</td>
</tr>
<tr>
<td>100%</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 5.1: The average size of nca quorums for different percentages of requesting nodes in the underlying network.

Table 5.1 shows the average sizes of nca quorums computed by NCA-QUORUM when varying percentages of nodes in the underlying failure-free network want to enter their critical sections. For example, when an average of 30% of the nodes in the network containing 15 nodes want to enter their critical sections, the average size of their nca quorums will be 2.875.

Since each quorum computed by the tree quorum algorithm consists of all the members of a path starting at the root and ending at a leaf, the sizes of tree quorums will be \([\log n]\) no matter how many nodes are competing.

From Table 5.1, we can see that when not all nodes are competing, the average nca quorum is usually smaller than the average tree quorum. As a consequence, the
competing nodes will generate less message traffic when nca quorums are used. For example, when fewer than 30% of the nodes want to enter their critical sections simultaneously, the size of the average nca quorum is approximately one less than that of the average tree quorum. This means that the competing nodes with nca quorums will generate approximately \( (\log n - 1)/\log n \) of the messages generated in the tree quorum case.

<table>
<thead>
<tr>
<th>level</th>
<th>failed nodes</th>
<th>the number of nodes in the underlying network</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2-3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4-7</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>8-15</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16-31</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>32-63</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: The average size of nca quorums when nodes on the different levels fail in the underlying tree network.

Under the assumption that all nodes are competing, Table 5.2 shows the average sizes of the nca quorums computed by PARTITION.NCA.QUORUM when all nodes at the same level in the tree fail. The second column shows the NODE numbers of the failed nodes in each case. At level 3, the failed nodes will be those numbered 4, 5, 6 and 7. When all nodes are competing, the nca quorums are the same as the tree quorums. The major advantage of the nca quorum algorithm is that it tolerates failures at any level, including the leaves, while the tree quorum algorithm can not tolerate leaf failures.
Table 5.3: The average size of nca quorums when individual nodes fail at different levels.

<table>
<thead>
<tr>
<th>level of failed node</th>
<th>the average size of nca quorums</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>5.933</td>
</tr>
<tr>
<td>2</td>
<td>5.2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

As Table 5.3 shows, the average size of new nca quorums generated by PARTITION-NCA-QUORUM when a node fails depends only on the level of the failed node. Table 5.4 shows the average size of nca quorums when various sets of nodes fail. From the table, we can see that no matter how many nodes at height 1 fail, the average size of the nca quorums remains \([\log n]\), where \(n\) is the number of nodes in the underlying network. Furthermore, when the leaves fail from left to right, the average size of the nca quorums decreases approximately 0.1 for every 2 additional node failures. Finally, in the case where the root fails, the average size of the nca quorums is \(2 \times [\log n]\), the same as that for tree quorums.

5.5.1 Comparison between Nca Quorum and Tree Quorum Algorithms

To demonstrate the difference between our algorithm and the logarithmic tree algorithm of Agrawal and El Abbadi, we will use the example of a complete binary tree on seven nodes used by them in their paper\[2\].

In general, regardless of node and communication failures, when all active nodes want to enter their critical sections, the nca quorum algorithm (NCAQ) and the tree quorum algorithm (TQ) generate the same quorums. However, if fewer nodes
want to enter their critical sections, then the nca quorums are usually smaller than
the tree quorums.

<table>
<thead>
<tr>
<th>failed nodes</th>
<th>the number of nodes in the underlying network</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2-3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4-5</td>
<td>2.667</td>
</tr>
<tr>
<td>4-6</td>
<td>2.5</td>
</tr>
<tr>
<td>4-7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>8-9</td>
<td>3.85</td>
</tr>
<tr>
<td>8-10</td>
<td>3.83</td>
</tr>
<tr>
<td>8-11</td>
<td>3.67</td>
</tr>
<tr>
<td>8-12</td>
<td>3.6</td>
</tr>
<tr>
<td>8-13</td>
<td>3.4</td>
</tr>
<tr>
<td>8-14</td>
<td>3.25</td>
</tr>
<tr>
<td>8-15</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>16-17</td>
<td>4.933</td>
</tr>
<tr>
<td>16-18</td>
<td>4.923</td>
</tr>
<tr>
<td>16-19</td>
<td>4.857</td>
</tr>
<tr>
<td>16-20</td>
<td>4.846</td>
</tr>
<tr>
<td>16-21</td>
<td>4.77</td>
</tr>
<tr>
<td>16-22</td>
<td>4.75</td>
</tr>
<tr>
<td>16-23</td>
<td>4.66</td>
</tr>
<tr>
<td>16-24</td>
<td>4.636</td>
</tr>
<tr>
<td>16-25</td>
<td>4.545</td>
</tr>
<tr>
<td>16-26</td>
<td>4.5</td>
</tr>
<tr>
<td>16-27</td>
<td>4.4</td>
</tr>
<tr>
<td>16-28</td>
<td>4.33</td>
</tr>
<tr>
<td>16-29</td>
<td>4.22</td>
</tr>
<tr>
<td>16-30</td>
<td>4.12</td>
</tr>
<tr>
<td>16-31</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.4: The average size of nca quorums when nodes indicated by the first column fail.

For example, in Figure 5.3, when all nodes participate the competition, both
NCAQ and TQ will generate the same coterie \{(1,2,4), (1,2,5), (1,3,6), (1,3,7)\}. Similarly, if the root fails, then both algorithms will generate \{(2,3,4,6), (2,3,5,6), (2,3,4,7), (2,3,5,7)\}. In cases like these, where NCAQ and TQ generate the same quorums, we will see the maximal sized nca quorums. When fewer nodes compete, NCAQ generates smaller quorums than TQ. For example, if only nodes 2 and 3 want to enter their critical sections, NCAQ will generate the coterie \{(1,2), (1,3)\}, while TQ generates a coterie consisting of two members of the set \{(1,2,4), (1,2,5), (1,3,6), (1,3,7)\}. As a result, using NCAQ instead of TQ will save at least two messages in this case. In the extreme case, when a leaf node fails, NCAQ will generate quorums while TQ will not. For example, when nodes 1, 2, and 4 fail, both (3,5,6) and (3,5,7) are nca quorums, but not tree quorums. Since no tree quorum can be generated when a leaf fails, NCAQ is more robust to failure.

Figure 5.3: A binary tree of 7 nodes

In addition to resiliency to failures, NCAQ also exhibits the desirable property of graceful degradation. For a partitioned network, when all active nodes participate in the competition, the behavior of NCAQ is similar to that of TQ. When a failure occurs, the increase in size of the affected nca quorums depends on the level of the failed node. If the failed node is at depth \(i\) in a tree of height \(k\), then the quorum
size increases from $k + 1$ to $i + 2(k - i)$, where the failed node is substituted by two paths going from the children of the failed node to the leaves. If fewer nodes are competing, then the nca quorums are usually smaller than the corresponding tree quorums, since the nca quorum algorithm computes only the nearest common ancestors of the youngest competing node with the current requesting nodes, while the tree quorum algorithm follows a fixed length path from the root to a leaf.

A major disadvantage of the nca quorum algorithm is the preprocessing time required to assign the ID values. However, this computation is done only once and requires $O(n)$ time. Another disadvantage is that, unlike tree quorums, the nca quorums must be computed each time. In practice, however, this computation cost is negligible unless failures occur at a rapid rate.

5.5.2 The General Behavior of the Nca Quorum Algorithm

To better understand the way that the nca quorum algorithm acts in different environments, we consider how quorum size is affected by various properties of the set $S$ of competing nodes, such as the number of nodes in $S$, the diameter of $S$, and the number of disjoint trees in $S$. From these considerations, a general pattern relating the connected components to the size of nca quorums is derived.

How does the number of competing nodes affect the size of nca quorums? Generally, since such a quorum contains the nearest common ancestors of some node with all other competing nodes, the greater the number of competing nodes, the larger the quorum. However, as the following example shows, this is not always the case. In Figure 5.4, when $S$ contains only nodes 10 and 11, the quorum for each node has size 2; and when $S$ contains nodes 5, 10 and 11, the quorum for each node again has size 2. Another example shows that equal sized compete sets can generate different sized...
quorums. When $S$ contains the nodes 3, 6 and 7, then the quorum for each node has size 2; but when $S$ contains the nodes 5, 6 and 7, $Q(5)$ contains only 2 nodes while $Q(6)$ and $Q(7)$ contain 3. Thus, while quorum size can be affected by the size of the competing set, it is not necessarily so.

Does the diameter of $S$ affect nca quorum size? In Figure 5.4, if $S$ contains only the nodes 7 and 15, so that it has diameter 1, then the size of the quorums is 2. When $S$ instead contains the nodes 8 and 15, then the diameter is maximum but quorum size is still only 2. Hence there is no relationship between the diameter of $S$ and quorum size.

![Figure 5.4: A binary tree of 15 nodes](image)

Does the number of disjoint trees in $S$ affect nca quorum size. At first glance, this would seem likely since a common ancestor would have to be included for each pair of disjoint trees. However, as the following example from Figure 5.4 shows, quorum size can increase while the number of disjoint trees decreases. When $S = \{1, 3, 5, 7, 10, 11, 15\}$, we have the coterie $\{(1,5,10), (1,5,11), (1,3,7,15)\}$. If we replace the subtree containing nodes 5,10, and 11 by the subtree containing nodes 6,12, and
13, then we get the coterie \{ (1,3,6,12), (1,3,6,13), (1,3,7,15) \}.

Although these examples show that there is no relationship between nca quorum size and various properties of \( S \), the last example contains an interesting pattern showing how disjoint subtrees affect the sizes of the nca quorums.

In Figure 5.5, suppose that the subtrees \( A \) are identical in both cases, as are the \( B \) subtrees. Since in Case 1 the nearest common ancestor of the nodes of tree \( B \) with the nodes of tree \( A \) is a node of \( A \), the nca quorum for each node in \( B \) must include all the nodes from \( p \) to \( q \). However, in Case 2, the nearest common ancestor of nodes in \( B \) with nodes in \( A \) is the root, which is the only node outside \( B \) that must be included in the quorum for any node in \( B \). Hence the nca quorums generated by \text{PARTITION\_NCA\_QUORUM} for each node in \( B \) are larger in Case 1 than in Case 2. We can see, then, that nca quorum size is governed by the relative placement in the network of the disjoint subtrees. Quorums will be larger when one subtree lies below another, and smaller when the subtrees are entirely unrelated.

Case 1: 

Case 2: 

Figure 5.5: Two patterns showing how disjoint subtrees affect the sizes of nca quorums
5.6 The Complete Nca Algorithms

FUNCTION ASSIGN_ID (n)

; assign ID values to the nodes in a complete binary
; tree, where n is the number of nodes in the tree

let tree_height = \lfloor \log n \rfloor;

node_no = 0;

for i = 0 to tree_height do

j = 1;

node_height = tree_height - i;

no = 2^i;

dif = 2^{node_height};

for k = 1 to no do

node_no = node_no + 1;

NODE[j * dif] = node_no;

[ID[node_no]] = j * dif;

j = j+2;

endfor;

endfor;

FUNCTION DELETE_SUBTREE (x_id nce_node nce_id n IDS)

;prune appropriate subtree from IDS.

begin

ids = ids - nce_id;

end function;
if \( x_{id} > nca_{id} \) 
then \( r_{node} = nca_{node} \times 2; \)
\[ r_{id} = ID[r_{node}]; \]
else if \( x_{id} < nca_{id} \) 
then \( r_{node} = 2 \times nca_{node} + 1; \)
\[ r_{id} = ID[r_{node}]; \]
else \( x = nca \)
\[ r_{node} = nca_{node}; \]
\[ r_{id} = nca_{id}; \]
let \( \text{dif} = 2 \lfloor \log_2 n \rfloor - \lfloor \log_2 r_{node} \rfloor; \)
remove all elements \( y \) from \( \text{ids} \), for
\[ r_{id} - \text{dif} + 1 \leq y \leq r_{id} + \text{dif} - 1; \]
end;

FUNCTION NCA.2 \((x_{id} y_{id})\);
; compute ID value of the nearest common ancestor of \( x \) and \( y \).
begin
\[ h = \lfloor \log_2 (x_{id} \cap y_{id}) \rfloor; \]
\[ nca.2 = 2^h + \left\lfloor \frac{x_{id}}{2^{h+1}} \right\rfloor \times 2^{h+1}; \]
end;

PROGRAM NCA.QUORUM \((x_{id} n \text{ IDS})\);
; the nca quorum algorithm for a complete binary tree.
begin
$x_{node} = NODE[x_{id}]$;

GRANT PERMISSION($x_{node}$);

IDS = IDS - $x_{id}$;

while IDS <> empty do

    let $y_{id}$ be the first element of IDS;

    $y_{node}$ be NODE[$y_{id}$];

    if $x$ is a descendant of $y$

        then GET PERMISSION ($y_{node}$);

        ; delete subtree rooted on $y$ which does not
        ; contain $x$

        DELETE-SUBTREE($x_{id} y_{node} y_{id} n$ (IDS - $y_{id}$));

    else if $x$ is an ancestor of $y$

        then GET PERMISSION ($y_{node}$);

        ; delete subtree rooted on $x$ which does not
        ; contain $y$ and replace $x$ by $y$

        DELETE-SUBTREE($y_{id} x_{node} x_{id} n$ (IDS - $y_{id}$));

        $x_{id} = y_{id}$;

        $x_{node} = y_{node}$;

    else $x$ and $y$ are unrelated

        then $nca_{id} = NCA.2 (x_{id} y_{id})$;

        $nca_{node} = NODE[nca_{id}]$;

        GET PERMISSION($nca_{node}$);

        ; delete subtree rooted on $nca_{node}$ which does
        ; not contain $x$ from IDS
PROGRAM PARTITION_NCA_QUORUM(xid n IDS);

; the nca quorum for a partitioned complete binary tree.

begin

x_node = NODE[xid];

GRANT_PERMISSION(x_node);

while IDS <> empty do

while IDS <> empty do

let yid be the first element of IDS;

y_node be NODE[yid];

if x is a descendant of y

then GET_PERMISSION(y_node);

; delete subtree rooted on y which does not

; contain x

DELETE_SUBTREE(xid y_node yid n (IDS - yid));

else if x is an ancestor of y

then GET_PERMISSION(y_node);

; delete subtree rooted on x which does not

; contain y and replace x by y

DELETE_SUBTREE(yid x_node xid n (IDS - yid));

x_node := y_node;

endwhile;

end;
\( x_{id} = y_{id}; \) 

else \( x \) and \( y \) are unrelated

then let \( nca_{id} = \text{NCA}_2(x_{id} y_{id}); \)

\( nca_{node} = \text{NODE}[nca_{id}]; \)

\( b_{ids} = \text{IDS} \)

; delete subtree rooted on \( nca_{node} \) which

; does not contain \( x \) from \( \text{IDS}. \)

\( \text{DELETE_SUBTREE}(x_{id} nca_{node} nca_{id} \text{ n IDS}); \)

if \( nca_{node} \) is not an element of \( \text{failJist} \)

then \( \text{GET_PERMISSION}(nca_{node}); \)

else

; group nodes of isolated subtree

\( \text{isolateJds} = b_{ids} - \text{IDS}; \)

while end;

if \( |\text{isolateJds}| = 1 \)

then \( \text{GET_PERMISSION}(\text{NODE}[\text{isolateJds}]); \)

\( \text{IDS} = \text{empty}; \)

else

\( x_{id} = \text{the first element of } \text{isolateJds}; \)

\( x_{node} = \text{NODE}[x_{id}]; \)

\( \text{IDS} = \text{isolateJds} - x_{id}; \)

\( \text{isolateJds} = \text{empty}; \)

\( \text{GET_PERMISSION}(x_{node}); \)

while end;
FUNCTION ASSIGN_ID (d n);

; assign ID values to nodes in a d-degree tree, where
; d = the degree of a node
; n = the number of the nodes in the
; tree

begin

let height = \left\lfloor \log_d n \right\rfloor;

base = 2^{\log_2 d} + 1;

node_no = 1;

ENQUEUE(Q,1);

NODE[1] = 1;

ID[1] = 1;

for h = 0 to height - 1 do

let j = d^h;

dif = base^{height-h};

for i = 0 to j - 1 do

\text{p}_{id} = \text{DEQUEUE}(Q);

for k = 1 to d do

\text{node_no} = \text{node_no} + 1;

ID[\text{node_no}] = \text{p}_{id} + k \times \text{dif};

\text{NODE}[\text{p}_{id} + k \times \text{dif}] = \text{node_no};

ENQUEUE(Q, \text{p}_{id} + k \times \text{dif});
endfor;
endfor;
endfor;
end.

FUNCTION NCA_D(xid yid base)
;compute ID value of the nearest common ancestor of x and y.
begi
let h = \lfloor \log_{base}(xid \oplus yid) \rfloor;
    a = base^{h+1};
NCA_D = a \times \lfloor \frac{xid}{a} \rfloor + 1;
end.

PROCEDURE DELETE_SUBTREE(xid nca_id h base IDS);
;prune appropriate subtree from IDS for d-degree tree.
begi
let \text{i}^{th}\_child = \lfloor \frac{xid - nca_id}{base^h} \rfloor;
    up1 = nca_id + \text{i}^{th}\_child \times base^h;
    low2 = nca_id + (\text{i}^{th}\_child + 1) \times base^h;
    up2 = nca_id + base^{h+1} - 1;
delete y from IDS (include deletion of nca_id)
    nca_id \leq y < up1;
    or low2 \leq y \leq up2;
end.
PROGRAM NCA QUORUM D (x_id IDS d n);

;the nca quorum for a d-degree tree.

begin

let base = \(2^{\log_2 d} + 1\);
height = \(\log_d n\);
x_node = NODE[x_id];
IDS = IDS - x_id;
GRANT PERMISSION(x_node);
while IDS <> empty do

let y_id = the first element of IDS;

y_node = NODE[y_id];
nca_h = \(\log_{\text{base}}(x_id \oplus y_id)\);
nca_id = NCA_D(x_id y_id base);
if nca_id = x_id then

GET PERMISSION(y_node);

DELETE_SUBTREE(y_id nca_id nca_h base (IDS - y_id));
x_node = y_node;
x_id = y_id;
else GET PERMISSION(NODE[nca_id]);

DELETE_SUBTREE(x_id nca_id nca_h base IDS);
endwhile;

end.
6.1 Transaction Manager

The job of a transaction manager (TM) is to ensure atomic commitment of a transaction. No partial effects of any member operations of a transaction will be visible to the user, so that the result of a transaction execution is “all or nothing”. The local updates performed by the concurrency controller must be confirmed by the transaction manager before permanent updates can take place. Thus, when a user submits a transaction to the transaction manager for that site, the transaction manager stores all member operations in an array transaction-array. The transaction-array is used to check the success of all member operations in the following way. If a member operation fails, the transaction aborts and restarts immediately. Since the lock of a data item would not be released until the transaction to which it belongs succeeds or aborts, the operations of a transaction on the same data item will be collapsed into one single operation. Based on this transformation, a transaction contains reads and writes on the same data item, only a write operation is included in the request list. Thus, the write operations of a transaction implicitly include read operations. For example, if a transaction contains a sequence of operations on \( x \): (\( \text{read}(x), \text{write}(x), \text{read}(x) \)), its request list will include \( \text{write}(x) \) only. However, when the transaction succeeds, the transaction manager will execute the original operation sequence (\( \text{read}(x), \text{write}(x), \text{read}(x) \)) instead of \( \text{write}(x) \). This is a reasonable transformation considering the conflict properties of read and write operations: A write
request conflicts with another read or write request but a read request conflicts only with another write request. Thus, when a transaction successfully gain a write lock on a data item $x$, it can safely perform a read on $x$ too.

In addition to committing all member operations atomically, the TM also tries to increase its success rate by updating its local clock from the local information available in the replica controller. When an operation request is received for a data item $x$, TM compares its local time with $RW$-$timestamp(x)$. If the local time is less than $RW$-$timestamp(x)$, the local clock is reset to $RW$-$timestamp(x) + 1$. In such a way we can ensure that, when a data item that has not been accessed for a long time, the new request on it will not affect the success rate of a transaction.

For each data item $x$ being operated on, an active site also keeps track of its lastest version number and value by received messages and granted permissions. The version number of $x$ is initialized to zero when the system starts, and increased by one whenever a update of $x$ occurs. The concurrency controller of each member of a write quorum permanently updates its version number and value of $x$ when a $release(x, t, vt, v, true)$ message with the current local time $t$ is received. Basically, the concurrency controller of each active node does local updating to keep track of the largest version number and its corresponding value when receiving $permission(x, t, vt, v, true)$ messages with the current local time from all the members of its quorum. Thus, the version number $vt$ and value $v$ of $x$ stored at each active site $i$ will contain the latest update of $x$ known by $i$, and when the transaction succeeds, the concurrency controller of each member of a read quorum will obtain the most recent value of $x$ from $v$. By this way, when a transaction containing reads and writes on $x$ succeeds, even when only $write(x)$ is included in its request list, the transaction can read the most recent value of $x$ and then update $x$. The details of
the transaction manager are listed as follows:

Step 1: Each transaction consists of a sequence of read-write operations on a set of data items. When a new transaction is received then for each data item \(x\), TM sends \(tm-rc(x,t)\) message, containing the local time \(t\) and \(x\) to local RC.

Step 2: TM waits for a return \(rc-tm(x,t,f)\) message, containing a timestamp \(t\) and a boolean-valued checking result \(f\), for each data item \(x\) from local RC. If \(f\) is false and the current local clock is less than or equal to \(t\), TM resets the local clock to \(t+1\).

Step 3: TM waits for a \(cc-tm(x,t,vt,v,f)\) message, indicating whether or not a member operation has succeeded, from the local concurrency controller. \(vt\) and \(v\) refer to the latest version number and value of \(x\) known by the sender of the \(cc-tm\) message. If \(f\) is true and \(t\) equals the local clock, the operation has succeeded so, TM records it in \(transaction-array\) and increments the counter \(agree\) by one. When the value of \(agree\) equals the number of member operations of the transaction, the transaction succeeds; TM continues from step 5. Otherwise, TM continues from step 4.

Note that a \(cc-tm\) message of a successful operation with earlier timestamp, indicating the operation belongs to a aborted transaction, will be ignored by TM.

Step 4: When a \(cc-tm(x,t,vt,v,false)\) arrives, containing the timestamp \(t\) of the request which caused the operation to fail, TM updates the local clock to this timestamp plus one and restarts the transaction by resetting \(agree\)
to zero, deleting successful operations from transaction-array, releasing all
granted permissions (uses release(y, vt, v, false) messages for a data item y),
and resending request(y, op, t) messages to all members of the nca quorum
of each member operation on y. Then TM continues from step 3.

Step 5: When the transaction has succeeded, TM increases x's version number
vt and does a permanent update v for each write request on the data item
x. Then TM sends release(x, vt, v, true) messages to the concurrency
controller of all members of the nca quorum for each member operation
on x.

6.2 Replica Controller

Three values are kept and updated by the replica controller for each data item
x: RW-timestamp(x), xnode(x), xnca(x). RW-timestamp(x) is the largest timestamp
of any read and write operations on x. The transaction manager uses this information
to update its local clock when a request on x is issued.

Xnode(x) is the node which performed the last update on x in the previous
time section, and xnca(x) is the write quorum of xnode(x) whose members contain
the most recent value of x. If xnode(x) fails, then the first nonfail node in xnca(x)
will be its backup node. At the beginning of each time section, after RC receives a
rm-rc(x, t) message from the local TM, it returns a rc-tm(x, t, f) message to the local
TM and sends a rc-reg(x, NODE) message to the RC of xnode(x). When the RC of
xnode(x) has received rc-reg(x, NODE) messages from all the competing nodes for a
data item x, it groups their NODE numbers into a list NODES, and adds its NODE
number to NODES if xnode(x) is not a competing node for x. The RC of xnode(x)
then sends NODES to the RC of each competing node, computes a nca quorum with respect to its NODE number and NODES, and sends a release(x,vt,v,true) message to each member of this quorum. Since each pair of nca quorums has a node in common, the most recent value of x can be passed from transaction to subsequent transactions by release messages and granted permissions.

Site and communication failures and replicated copies of the same data item are hidden by the replica controller at each site. With the cooperation of the concurrency controller, the replica controller ensures that a multiple copy system acts as a single copy system. A read will access the most recent update value, and a write will occur as a single copy write for each write request, so that no inconsistent updates will appear among multiple copies of the same data item. To achieve this goal, some decisions must be made by all the copies of a data item. The nca quorum algorithm lets this decision be made by a quorum of the copies rather than by all the copies of a data item. Using the characteristics of the logical tree structure, each request node computes all the nearest common ancestors of the node's youngest competing descendant with all requesting nodes. The nearest common ancestor between any pair of competing nodes ensures that there is a nonnull intersection of any pair of nca quorums. Thus, conflicting operations can not occur on different copies of the same data item simultaneously. For efficiency, we also enforce the minimization property of a coterie by including all the common ancestors of the requesting node's youngest competing descendant with all the competing nodes.

By using this nca quorum algorithm, the number of messages needed to make a decision is reduced and the quorum size of a write operation is smaller in the lightly loaded system, while having the same upper bound as that of the tree quorum algorithm [2] in the heavily loaded system. The nca quorum algorithm is a dynamic
version of the tree quorum algorithm. The major achievement of the nca quorum
algorithm is that its write quorum size is not influenced by the size of the read
quorum. A decrease in a write quorum size does not increase the corresponding read
quorum size. The two are independent. This is totally different from all the previous
quorum consensus protocols for a replicated database system. Depending on the type
of the arriving message, the replica controller responds as follows:

1: When a \texttt{tm-rc}(x,t) message coming from the local TM for a data item \(x\)
arries, RC checks \(t\) against \(RW\)-timestamp(\(x\)).

If \(t\) is less than \(RW\)-timestamp(\(x\)), RC sends an \texttt{rc-lm}(x, \(RW\)-timestamp(\(x\)),
\textit{false}) message to the local TM.

Else RC returns an \texttt{rc-lm}(x,t,\textit{true}) message to the local TM.

Next, RC sends an \texttt{rc-reg}(x,NODE) message, containing a data item
\(x\) and NODE number of the sender, to the RC of xnode(\(x\)) that has not
failed; otherwise RC sends the message to the RC of the first node in
\texttt{xnca}(\(x\)) that has not failed.

2: When a \texttt{rc-reg} message for a data item \(x\) arrives a node \(i\) which is \texttt{xnca}(\(x\)),
RC accumulates the node numbers of all competing nodes for \(x\) from
arriving \texttt{rc-reg} messages. After the longest period of time that any message
can reach its destination in a network, RC sends the NODES, the NODE
numbers of all competing nodes including xnode(\(x\)), to each competing
node.

3: When the NODES for a request \(r\) arrives at a node \(i\), the local RC com-
putes the nca quorum of \(r\) by using \(i\)'s NODE and NODES as input.
Then, RC sends the nca quorum to the local concurrency controller.

6.3 Concurrency Controller

The concurrency controller (CC) is the safeguard of global consistency of different copies of the same data item. Our algorithm uses Lamport's logical time concept to prioritize requests. If one request has a smaller local time than other requests, then this request will be granted the permission. If two requests have the same local time, then the one with a smaller NODE number gets the permission.

During the transaction processing, a local queue for each data item being operated on is created at each member of an operation's quorum and a quorum-array for each operation request is created at the site where the transaction containing the operation is submitted. At that time the replica controller passes an nca quorum of a request r for a data item x to the CC. The CC stores this nca quorum in the quorum-array and uses x as its index. If all members of r's quorum grant permission, then CC notifies the transaction manager of the success of r by sending a $cc-tm(x,t,vt,v,true)$ message. Otherwise, a $cc-tm(x,t,vt,v,false)$ message will be sent to TM, which will restart the transaction immediately.

In order to implement concurrent reads, the CC keeps track of two variables $nr$ and $nw$ for each data item $x$. If the first $k$ requests on $x$ are reads and the next one is a write, then CC grants permissions for the reads and $nr$ is set to $k$. Consequently, $k$ release messages must be received before CC can grant permission to the next request on $x$. If the first request on $x$ is write, then CC grants permission only for the write request, and sets $nw$ to one.

Since any pair of nca quorums has a node in common, we piggyback the most
recent value of a data item $x$ upon the concurrency control messages. In this way, a read operation can get the most recent value from the received permissions, and a write operation can propagate the update value to the members of its quorum by sending back release messages. Thus, when the value of $nw$ is not zero and CC receives a release message with a success flag, local updates become permanent. Otherwise, CC discards local updates if the transaction aborts. Depending on the type of the arriving message, the concurrency controller responds as follows:

1: When a nca quorum of a operation $op$ on a data item $x$ from local RC arrives, CC stores it in $quorum-array$, and sends a $request(x, op, t)$ message, containing operation type $op$ and the local time $t$, to each member of the nca quorum.

2: When a $request(x, op, t)$ message arrives, CC compares $t$ with the timestamp $cc-time(x)$ of the last blocked request on $x$.

   If $t$ is less than $cc-time(x)$, then the new request will be rejected immediately by sending a $cc-tm(x, cc-time(x), vt, v, false)$ message to the TM of the requesting node.

   Else if $t$ equals $cc-time(x)$ and the NODE number of the new request is less than that of the last blocked request, then the new request will be rejected immediately by sending a $cc-tm(x, cc-time(x), vt, v, false)$ message to the TM of the requesting node.

   Else if the lock is free ($nr = 0$ and $nw = 0$) and the timestamp of the new request is greater than that of the last blocked request (when $op$ equals read, $t > cc-time(x)$ or $t = cc-time(x)$ and the NODE number of the new request is greater than that of the last blocked request; when $op$
equals write, \( t > cc\text{-}time(x) \), CC grants the permission\((x,t,vt,v)\) to the arriving request and sets \( nr \) to 1 if \( op \) is read. Otherwise CC sets \( nw \) to 1, and resets \( cc\text{-}time(x) \) to \( t \). \( vt \) and \( v \) are the version number and the value of the latest permanent update on \( x \) known by this node.

Else if the lock is not free, both the new request and the current blocked request are read operations, \( t = cc\text{-}time(x) \) and the NODE number of the new request is greater than the smallest NODE number of the current blocked requests, then CC grants permission to the new request if the queue for \( x \) is empty and increases \( nr \) by one. Otherwise, \( t \) must be less than the timestamp of the head of the queue before CC can grant permission to the new request.

Else CC puts the new request in the queue for \( x \) by timestamp order.

Note that for each operation request on \( x \) from a site \( i \), CC only grants permission to the most recent request and deletes any earlier requests in the queue for \( x \). We do so by keeping track of the largest timestamp of received requests on \( x \) from \( i \).

3: When a release\((x,vt,v,f)\) message for a data item \( x \) arrives, CC decreases the value of \( nr \) by one if the current blocked request is read, otherwise CC decreases \( nw \) by one. If \( f \) is true and the current blocked request is write, then the permanent update of \( x \) takes place and CC stores \( vt \) and \( v \) in the stable storage. Also each node keeps the newest version number and value for each data item \( x \) when a release\((x,vt,v,true)\) message is received.

When \( nr = 0 \) and \( nw = 0 \), the lock for \( x \) becomes free again. If the local queue for \( x \) is empty, then CC sets lock to free. Otherwise, CC
grants permission to the head of the queue for $x$. If the granted request is a read operation and there are $k$ successive read requests with the same local time before the first write request in the queue for $x$, then CC grants permissions to all $k$ read requests and resets $cc\text{-}time(x)$, $cc\text{-}node(x)$, and $nr$ to the timestamp $t$, NODE number of the last blocked request, and $k+1$ respectively. The lock remains not free.

4: When a $permission(x,t,vt,v)$ with $t$ equalling the current local time for a data item $x$ arrives, CC checks if it is the last permission required for $x$. If the answer is yes, then CC sends a $cc\text{-}tm(x,t,vt,v,true)$ message to the local TM notifying it of the success of that operation. Otherwise, CC records the permission in $quorum\text{-}array$.

Note that if the request on $x$ is read, then $v$-field of the permission with the largest version number $vt$ from all the members of $x$'s quorum will contain the most recent value of $x$.

5: When a $permission(x,t,vt,v)$ with timestamp $t$ which less than the local time arrives, CC sends $release(x,t,vt,v,false)$ message immediately because the permission has been granted to an earlier aborted operation request.

**Lemma 6.1:** No two conflicting operation requests on the same data item will be granted by a node simultaneously.

**Proof:** When a new operation request on a data item $x$ arrives, if the timestamp of the new request is less than that of the last blocked request, the new request will be rejected. Otherwise, there are four possibilities to consider depending on whether or not the lock for a data item $x$ is free and the queue for $x$ is empty. Recall that
each member of the quorum of a request operating on $x$ maintains a queue for $x$ during current transaction processing. All requests in the queue are ordered by their timestamps. In the first case where the lock for $x$ is free and there is no outstanding request in the queue, CC grants permission to the new request immediately and sets the lock for $x$ to not free. In the second case where the lock $x$ becomes free while some outstanding requests are waiting in the queue, CC puts the new request into the queue. Then CC grants permissions to all read requests with smaller timestamps than the first write request in the queue if the queue head is a read request. Otherwise, CC grants permission to the single write request which is the queue head. The lock for $x$ is set to not free. In the third case where the lock is not free and there is no outstanding request in the queue, if the current blocked request is a write, the new request will be placed in the queue. Otherwise, either CC grants permission to the new request if the timestamp of the new read request is greater than that of the current blocked request, or CC puts the new request into the queue. The lock remains not free. In the fourth case where the lock is not free and the queue is not empty, if the current blocked request is a write, the new request will be placed in the queue. Otherwise, either CC grants permission to the new request if the timestamp of the new request is greater than the smallest timestamp of the current blocked requests and is less than that of the queue head or CC puts the new request into the queue. The lock remains not free.

Since a timestamp consists of a local time and a node identifier and no two node identifiers are identical, no conflicting operation requests will be granted by a node at the same time.

**Theorem 6.1:** No conflicting operation requests on the same data item are executed
at the same time.

**Proof:** Each operation request must collect the permissions from all members of its nca quorum before performing the operation. From Theorem 5.3, each pair of nca quorums for operation requests on the same data item have at least one node in common. As a result of Lemma 6.1, no conflicting operation requests on the same data item are executed at the same time.

**Theorem 6.2:** All conflicting operation requests on the same data item are executed in their timestamps order.

**Proof:** As a result of Lemma 6.1 and the fact that each node grants permission of a data item to all conflicting operation requests in their timestamp order.

**Lemma 6.2:** Each transaction execution satisfies atomic commitment.

**Proof:** All operation requests of a transaction are put in *transaction-array* when the transaction processing starts. If an operation succeeds, the transaction manager records this information in the *transaction-array*. If an operation request fails, the transaction manager updates its local lock as large as necessary, releases all granted permissions, and resends all requests with a new local time again. This process will be repeated until all operation requests succeed. Only at that time does the permanent update take place. As a result, the effects of the transaction execution will be "all or nothing"; no partial effects will be reflected on the database.

**Theorem 6.3:** All transactions having conflicting requests on the same data item are executed in their timestamp order.

**Proof:** As a result of Theorem 6.2 and Lemma 6.2.
Lemma 6.3: The concurrent transaction processing satisfies serializability.

Proof: Transactions submitted in the same time period are executed in their time stamp order. Also, the updated results are passed from one transaction to other transactions through the common node of their nca quorums. As a consequence of Theorem 6.3, the concurrent execution of transactions submitted in the same time period produces the same database state as some serial execution of the same transactions.

Theorem 6.4: Transactions are executed atomically.

Proof: The theorem is an easy consequence of Lemma 6.2 and Lemma 6.3.

Theorem 6.5: The algorithm is deadlock free.

Proof: Assume that there is a waiting cycle of transactions: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$. In that case there exists an operation $r_i$ in each transaction $T_i$ of the cycle such that $r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow \ldots \rightarrow r_n \rightarrow r'_1$, where $r_1$ and $r'_1$ may not refer to the same operation. Since the timestamp of an operation request waiting in the queue must smaller than that of the current blocked request. The timestamp of the first $r_1$ is less than that of the last $r_1$ which is impossible because all operation requests of the same transaction have identical timestamp.

Theorem 6.6: The algorithm is starvation free.

Proof: All transaction submitted in the previous time period must succeed before executing any transaction submitted in the next time period. Thus, a transaction will eventually succeed in a finite amount of time.
6.4 Suggested Time Section Techniques

In our algorithm transactions are executed atomically. That means all member operations of a transaction must commit atomically, and the concurrent execution of a set of transactions must generate the same effects as a serial execution of the same transactions. Let \( T_0 \) be a transaction to which an operation request \( o_i \) belongs. The request \( o_i \) can be rejected due to an operation request \( o_j \) with a larger timestamp only when \( o_j \) has arrived at a common node before \( o_i \). This rejected operation request will cause \( T_0 \) to be aborted, releasing all its granted permissions, and resetting the local time of \( T_0 \) to \( \text{timestamp}(T_0) + 1 \) if the new timestamp is larger than the original. This result is applicable to any pair of transactions which request permissions for the same data item from a common node. As a consequence, the timestamp of a transaction will be updated continually until the timestamp is large enough that all its member operations will succeed at the same time. The timestamp of an operation request consists of the local time and NODE number of the site to which the transaction was submitted. Since no two NODE numbers are identical, a node can grant permission for a data item to only one of a set of conflicting requests at a time. A maximal set of transactions \( T \) is a set of transactions \( \{T_1, T_2, T_3, \ldots T_k\} \) in the current time section such that \( T_i \) and \( T_{i+1} \) access at least one data item in common for \( 1 \leq i \leq k - 1 \), but \( T \) is not a subset of other maximal sets of transactions. \( TP \) is this subset of \( T \) to which a transaction \( T \) belongs if and only if at least one of its member operations is the first to receive permission for some shared data item after the current time section starts. Since there is no deadlock, the transaction \( T_q \in TP \) with the largest timestamp will eventually succeed, followed by the success of at least one transaction with \( \text{timestamp}(T_q) + 1 \), which in turn is followed by the success of transactions with
timestamp(T_q) + 2, and so on.

In the following section we will talk about a set of maximal sets of transaction $P = \{T^1, T^2, T^3, \ldots, T^w\}$ in the current time section. If $T^i \in P$, $T^P_i$ is the corresponding $TP$ of $T^i$, and $T^q_i$ be the corresponding $T_q$ of $TP^i$, then $MaxT_q$ will be $max(timestamp(T^q_i))$ for any $T^q_i$ in the current time section. When a node received a list NODES of competing nodes on a data item $x$, it knows the number of competing nodes on $x$, and stores this number in an integer called $xrequest(x)$. As a result, at the time $MaxTime = MaxT_q + max(xrequest(x)) + 1$ for all $x$ involved in the current time section, we can guarantee that all transactions submitted in the current time section can be completed.

**Theorem 6.7:** All the transaction submitted in the current time section will be complete before $MaxTime$.

**Proof:** Suppose that $T^i \in P$ is a maximal set of transactions $\{T_1, T_2, \ldots, T_k\}$ such that $T_i$ and $T_{i+1}$ access at least one data item in common for $1 \leq i \leq k-1$. Also, $T^P_i$ is this subset of $T^i$ to which a transaction $T$ belongs if and only if at least one of its member operations is the first to receive permission for some shared data item after the current time section starts. Since there is no deadlock, the transaction $T^i_q \in T^P_i$ with the largest timestamp will eventually succeed, followed by the success of at least one transaction with $timestamp(T^i_q + 1)$, which in turn is followed by the success of transactions with $timestamp(T^i_q) + 2$, and so on. Thus, before $timestamp(T^i_q) + max(xrequest(x^i))$ for all $x^i$ involved in $T^i$, all the transactions in $T^i$ will complete their transactions. Consequently, all the transactions submitted in the current time section will be completed by $max(timestamp(T^i_q) + max(xrequest(x^i)))$, which is less than $max(timestamp(T^i_q)) + max(xrequest(x)) = MaxTime$ for all $T^i \in P$ and $x$.
involved in the current time section.

However, it is possible that all transactions submitted in the current time section complete before MaxTime if some transactions complete their processing concurrently. In order to start the next time section as soon as possible, \( x_{request}(x) \) will be passed along with the permission for \( x \) and decreased by one when a transaction operating on \( x \) succeeds. Also, \( x_{nca}(x) \) will send the list of competing nodes on \( x \) to all the nodes in the network so that all the nodes know the NODE numbers of all participating nodes. When a transaction whose \( x_{request}(x) \) is the number of competing nodes on \( x \) has succeeded, the transaction is one member of \( TP_i \) for some \( T_i \) and its resident node will send its timestamp to other nodes in the network so that MaxTime can be computed. If after MaxTime a node still not receives the success message of a participating node, then it can send a message to the “unsuccessful” node to check either the message is lost or the node already failed. Based on the reason, the node takes an appropriate action.

On the other hand, when a transaction with \( x_{request}(x) = 0 \) has succeeded, the resident node knows that all the nodes in NODES for \( x \) have succeeded and will broadcast the success messages of all competing nodes on \( x \) to all other nodes in the network (Use information in NODES). As a result, the next time section can start at the moment when all the participating nodes have succeeded.

Another suggested approach uses semaphores to control the number of transactions allowed in a time section. When limit number of transactions finish, the next time section starts. Transactions belonging to the next time section wait at \( P(wait) \) until the current time section ends.

\[
\text{sem mutex = 1; wait = 0;}
\]
\[
\text{int n = 0;}
\]
\texttt{P(mutex);}  
\texttt{if } \texttt{n} \texttt{ > } \texttt{limit} \texttt{ then } \{ \texttt{V(mutex);} \texttt{ P(wait);} \}  
\texttt{n} = \texttt{n} + \texttt{1;} \texttt{ V(mutex)}  

\{Transaction Processing\}

\texttt{P(mutex);}  
\texttt{if } \texttt{n} \texttt{ < } \texttt{limit} \texttt{ then } \texttt{V(mutex);}  
\texttt{else for i := 1 to n do}  
\texttt{V(wait);}  
\texttt{n} = \texttt{0;} \texttt{ V(mutex);}  

An alternative semaphore approach uses semaphores to control the beginning of a time section but not the number of transactions allowed in a time section. When all transactions submitted before the current time section began are finished, the next time section begins, with all transactions submitted during the previous time section being processed. This approach is more efficient because the submitted transactions can start as soon as the current time section ends, even if fewer than \texttt{limit} transactions were completed in the previous time section.

\texttt{sem mutex := 1;} \texttt{wait := 0;}  
\texttt{int n, d := 0;}  
\texttt{P(mutex);}  
\texttt{if } \texttt{d} = \texttt{0} \texttt{ and } \texttt{n} = \texttt{0} \texttt{ then}  
\texttt{\{} \texttt{n := n + 1;} \texttt{ V(mutex);} \texttt{;} \texttt{\}}  
\texttt{else if } \texttt{n} > \texttt{0} \texttt{ then}  
\texttt{\{} \texttt{d := d + 1;} \texttt{ V(mutex);} \texttt{ P(wait);} \texttt{;} \texttt{\}}  

\{Transaction processing\}

\texttt{P(mutex);}  
\texttt{n := n - 1;}  
\texttt{if } \texttt{n} = \texttt{0} \texttt{ then}  
\texttt{for i := 1 to d do}  
\texttt{V(wait);}  
\texttt{n := d;} \texttt{ d := 0;}
V(mutex);

The final suggested time section technique is similar to message flooding. Starting from the root, "start" signals flood down the logical tree until reaching the leaves. When all the transactions submitted to a node and its descendants have finished, the node will send a "done" signal back up the tree. When the root has received "done" signals from all its children, it starts the next time section by repeating the same message flooding technique. For a partitioned network, the root will try to send "start" signals to its children. If one of them has failed, the root will send "start" signals to the children of the failed node. Similarly, after a node receives a "start" signal, it will try to send out "start" signals to its children. If one of them has failed, the node will send "start" signals to the children of the failed node. This recursive procedure works for a failed child node also. If none of a node's descendants are alive, the node will send back a "done" signal after all the transactions submitted to the node have finished. Otherwise, a "done" signal will be sent back up the tree when the node has received "done" signals back from all the nodes to which it has sent "start" signals, and the node has finished its transaction. The "done" signal is sent to the node from which a "start" signal was received. If the root fails, the node with smallest NODE number will replace it and start the next time section by using the same technique.

The first suggested approach performs best when the number of requests on each data item is constant. In such a case, most transactions will succeed almost at the same time. Otherwise, some nodes may have to wait very long if the difference between the numbers of requests on different data items is very large. For example, if there are three transactions \( T_1, T_2, T_3 \) submitted to nodes a1, a2, and a3, with local
time 2, 3, and 5 respectively. Both $T_1$ and $T_2$ operate on one single data item $x$, while $T_3$ operates on data items $x$ and $z$. Suppose that $T_3$ succeeds first and $p$ is the longest time needed to complete a read or write operation. The current time section requires up to $3p$ time to complete its transactions if all operations are writes, but only $p$ time to complete if all operations are reads.

Consider another example where $T_2$ operates on $y$ instead of $x$, and $T_3$ operates on $x$ and $y$ instead of $x$ and $z$. In this arrangement, the number of operations on both $x$ and $y$ will be the same. As a result, the current time section completes up to $2p$ time if all operations are writes, and $p$ time if all operations are reads.

Unlike the first approach which considers the number of requests on each data item, the second approach requires the number of transactions per time unit to be as stable as possible. Otherwise, the average waiting time of transactions will be large. In a real system, a copy of the suggested algorithm will reside on each site. When a transaction is submitted to a node, the node will send messages to notify all the nodes in the network. Consider the case where the number of transactions submitted at different time units varies and the predefined number of transactions allowed per time section is $\text{limit} = 2$. If five transactions $T_1, T_2, T_3, T_4, T_5$ are submitted in the current time section subsequently, then $T_3$ and $T_4$ must wait for $T_1$ and $T_2$ to complete, and $T_5$ must wait for other four transactions to complete. One problem with this approach is that if after the first transaction in the current time section has been submitted, there is a long time before the other transaction submitted, then the first transaction must wait unnecessarily. Thus, the second suggested approach performs best when the number of transactions submitted per time unit is constant.

When the number of transactions in each time section is unstable, the alternative semaphore approach can be used. It provides great flexibility, allowing a varying
number of transactions to be executed in each time section. As soon as the transactions submitted before the current time section started have finished, the next time section starts. In the previous example, all five transactions could execute in a single time section if they were all submitted before the previous time section started.

Finally, the last suggested approach is preferable to the other approaches when the system configuration changes frequently. Since messages flood down and up a tree along active nodes, the approach can adapt dynamically to changes in system configuration without extra effort. Also, unlike other approaches that broadcast messages to all the nodes in the network, this approach requires each active node only to communicate with its nearest active neighbor, which reduces message traffic. However, if transactions succeed from the root to leaves rather from the leaves to the root, the algorithm will suffer unnecessary delay due to “done” messages having to sent back up the tree.
CHAPTER 7

ANALYSIS

In order to understand the performance of our algorithm, a simulation model program in SR has been designed for networks containing seven and fifteen sites. Each site consists of a transaction manager, a concurrency controller, and a replica controller. The only communication between sites is through message exchanges. Messages are delivered in the order in which they were sent. No message is lost, duplicated or collapsed in the communication link.

First, the general performance of the algorithm was studied by running the program for a network containing fifteen sites, each of them containing five data items. Tables 1 to 12 show the results of varying the percentage of read operations in a transaction over the values 10%, 50%, 90%, the percentage of competing nodes in the network over the values 10%, 30%, 50%, 70%, 90%, and the number of operations contained in a transaction from 2 to 5. For each individual case the program was run 200 times, each simulating an entire time section containing \((\text{the percentage of competing nodes}) \times 15\) transactions. All transactions were distributed randomly in the network. The type of a member operation was decided randomly depending on the percentage of read operations in each transaction. After completing all 200 time sections for an individual case, the average number of messages per operation, the restart rate, the throughput per 1000 millisecond, and the response time were computed.

The results in Figures 7.1 through 7.3 show that the average number of mes-
sages per successful operation increases as the percentage of competing nodes in a network increases no matter how many operations are contained in a transaction. Also, the difference between the average numbers of messages per successful operation when transactions contain three, four, and five operations becomes smaller as the percentage of reads in a transaction increases. In addition, as the percentage of read operations contained in a transaction increases from 10% to 50%, the average number of messages per successful operation decreases when the transaction contains no more than \([5/2]\) operations. Otherwise, the average number of messages increases.

![Figure 7.1: The average number of messages for 10% read operations.](image1)

![Figure 7.2: The average number of messages for 50% read operations.](image2)
The reason for this comes from the congestion of operation requests at the common node of each pair of quorums. Especially when we store 5 data items at each site and all the transactions contain 4 or 5 data items, the probability that different transactions ask permissions for the same data item from a common node is very high. Since a node grants permissions to several read requests simultaneously in only one of the following situations, many requests are likely to be rejected: first, both the current blocked request(s) and the newly arrived request are reads and have the same local time and the NODE number of the newly arrived request is greater than that of the current blocked request(s). Second, even when the lock of the data item requested is free, the newly arrived read request is still required to contain a greater timestamp than the last blocked request before the permission can be granted. In this way, it is guaranteed that an obsolete value will not be accessed by a read request. Third, if the lock of a data item $x$ becomes free again while a sequence of operation requests waits in the queue for $x$, the node will grant permissions to all read requests with the same smallest local time that precede the first write request in the queue when the queue head is a read request. As a result, if two read requests with the same local
time as the last blocked request arrive at a node in a different order, the result may be totally different. Either both of them gain their permissions at the same time or one gets permission and the other is rejected.

In most cases, the average number of messages per successful operation for transactions containing different numbers of operations decreases when the percentage of reads in a transaction increases from 50% to 90%. Below 50%, concurrent reads do not significantly affect the average number of messages for each successful operation.

Figure 7.4: The average restart rate per transaction for 10% read operations.

Figure 7.5: The average restart rate per transaction for 50% read operations.
Similar results for restart rates for transactions containing different numbers of operations are shown in Figures 7.4 through 7.6. The restart rate goes up as the percentage of competing nodes increases, and the difference in restart rates between transactions containing different numbers of operations reduces as read operations increases. Also, when the percentage of reads in a transaction increases, the restart rate goes down for a transaction containing 2, 3, or 4 operations. However, if a transaction contains 5 operations, the restart rate goes up as the percentage of read operations increases from 10% to 70%. After that the restart rate goes down.

Figure 7.7: The throughput per 1000 milliseconds for 10% read operations.
Since the more that nodes compete to access the same data item, the smaller the throughput is, in Figures 7.7 through 7.9 when the percentage of competing nodes increases, the throughput decreases. However, in the atomic transaction execution environment the percentage of read operations must reach a certain point, depending on the number of operations in the transaction and the percentage of competing nodes, before concurrent reads can actually affect the system throughput. If fewer than 50% of the nodes in a network are competing and each transaction contains no more than \( \left\lceil \frac{5}{2} \right\rceil \) operations, the throughput increases as the percentage of read operations increases from 10% to approximately 50%, and then decreases as the per-
percentage of reads increases from 70% to 90%. In contrast, when such transaction contain more than \([5/2]\) operations, the throughput decreases at first, then increases as the percentage of reads increases above approximately 50%. If the percentage of competing nodes exceeds 50%, then the throughput increases as the percentage of reads increases no matter how many operations are contained in a transaction.

Finally, since the shorter response time produces higher throughput, the opposite of the throughput results can be expected in Figures 7.10 through 7.12.

When transactions containing more operations are running, each member operation can be expected to wait longer in a queue. During the period that an operation request is waiting in the queue, a new duplicate request with a larger timestamp, resent because of the abortion of another member operation, may arrive. The concurrency controller will ignore the earlier one and grant permission to the later one. Thus, the more operations contained in a transaction the longer its waiting time in the queue, which reduces the number of member operations of that transaction that will be aborted. In turn, the restart rate becoming smaller causes the average number of messages per operation to become smaller.

![Figure 7.10: The average response time per transaction for 10% read operations.](image)
During the simulation process we tried using $R$-timestamp($x$) for a read operation and $\max(W$-timestamp($x$), $R$-timestamp($x$)) for a write operation as in [11] to update the local time before a transaction operating on $x$ starts. $R$-timestamp is the largest timestamp of all reads on $x$ and $W$-timestamp is the largest timestamp of all writes on $x$. The results show that the greater percentage of read operations contained in a transaction, the worse the performance is. A transaction containing more read operations aborts more frequently than one containing more write operations. The reason is that a transaction containing a read operation on a data item $x$ uses only $R$-timestamp to update its local time, not considering any write operation.
that occurs on $x$ after the last read on $x$. The resulting timestamp is smaller than $cc\text{-}time(x)$, causing the read request to be rejected and the transaction to be aborted.

On the other hand, a transaction containing a write operation on $x$ considers both $R\text{-}timestamp$ and $W\text{-}timestamp$. The resulting timestamp equals $cc\text{-}time(x) + 1$, which is the timestamp we want.

7.1 Further Analysis

In order to understand the effects of having various numbers of data items resident at each site of the network, we ran the program for different data distributions. Four cases were considered: two data items, three data items, four data items, and five data items stored at each site of a network. The number of operations in a transaction varied from one up to the maximum number of data items stored at any site. Samples were selected by varying the percentage of active nodes in the network from 10% to 100%, and the percentage of read operations of each transaction from 10% to 100%. For each individual case we ran the program 200 times and computed its average number of messages, restart rate, throughput, and response time twice for networks containing seven and fifteen sites.

The regression analysis was done using the linear model:

$$Y = B0 + B1 \times NOP + B2 \times PERACT + B3 \times PEREAD.$$ 

$Y$ is the average number of messages per successful operation, the restart rate, the response time of a transaction, or the throughput per 1000 milliseconds. NOP is the number of operations contained in a transaction, PERACT is the percentage of competing nodes in a network, PEREAD is the percentage of read operations in each transaction.
The regression results show that the average number of messages per successful operation is affected by the number of operations contained in a transaction and the percentage of competing nodes in the network, but not by the percentage of read operations in a transaction. When more data items reside at each site, the number of operations contained in a transaction has less influence on the average number of messages, while the percentage of competing nodes has greater influence. The same results were found for the restart rate. However, the regression results for throughput are different, in that both the number of operations and percentage of reads contained in a transaction have greater influence on throughput as more data items reside at each site. Also, it is surprising that the response time per transaction is significantly affected only by the percentage of competing nodes in a network. However, when more data items reside at each site, the regression results for the response time are the opposite of throughput, that is the number of operations in a transaction and the percentage of competing nodes have less influence on the response time.
CHAPTER 8

COMPARISON BETWEEN TREE QUORUM ALGORITHM AND NCA QUORUM ALGORITHM

The latest development in tree quorum algorithms [3] gives the user two degrees of freedom in choosing quorums, which is a milestone in the development of tree quorum techniques. Each read (write) quorum can choose \( l_r \) (\( l_w \)) levels and \( w_r \) (\( w_w \)) children. Starting from the root, a successful read quorum in a failure-free network is constructed by selecting the root and \( w_r \) of its children; for each selected child, \( w_r \) of its children; and so on to depth \( l_r \). A write quorum can be constructed similarly. Unfortunately, in order to satisfy the intersection property of a coterie, the choice of read and write quorums of a data item is not totally without restriction. In fact, the requirements of \( l_r + l_w > h \) and \( w_r + w_w > d \) not only affect the size of a quorum, but also the number of faults that the tree quorum algorithm can tolerate. On the other hand, the nca quorum algorithm computes a quorum for each operation request dynamically and utilizes the nearest common ancestor of two nodes to enforce the intersection property. The read quorum and write quorum of the same data item are independent in size and self adapted to system configuration changes, freeing the user from selecting appropriate quorums when the system configuration changes. Since all the elements \( l_r, l_w, w_r, w_w, h, \) and \( d \) affect the performance of the tree quorum algorithm, it is difficult to decide what is the general case of the tree quorum algorithm. Agrawal and El Abbadi have used majority tree, read root, and log write instances of the tree quorum algorithm to study its performance. We will follow
them in comparing the performances of the nca quorum algorithm and tree quorum algorithm.

In a quorum-based algorithm, the message cost of an operation is proportional to the size of the read quorum or write quorum. Thus in order to compare the expected cost of the nca quorum algorithm and the tree quorum algorithm, we will generate read quorums and write quorums by using both algorithms in various system configurations simultaneously. In Tables 8.1, 8.2, and 8.3 the lower bounds for read and write quorums are shown above the slash line and the upper bounds are shown below the slash line. The results in Table 8.1 show that the nca quorums have smaller lower bound and upper bound for both read quorums and write quorums than the majority tree instances of tree quorums. Also, from Table 8.2 we can conclude that nca quorums have the same lower bound and upper bound for read quorums as for the read root instances of tree quorums. However, nca quorums have much smaller lower bound and upper bound for write quorums. In the log write instances of tree quorums, nca quorums have the same lower bound and upper bound for both read quorums and write quorums as tree quorums. In general, nca quorums have smaller quorum size than tree quorums.

<table>
<thead>
<tr>
<th>(n)</th>
<th>Tree Quorum</th>
<th>Nca Quorum</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n = 4)</td>
<td>1/2 3/3</td>
<td>1/2 2/2</td>
</tr>
<tr>
<td>(n = 13)</td>
<td>3/6 3/6</td>
<td>2/4 2/4</td>
</tr>
<tr>
<td>(n = 40)</td>
<td>3/12 7/14</td>
<td>2/8 3/6</td>
</tr>
<tr>
<td>(n = 121)</td>
<td>7/28 7/28</td>
<td>3/12 3/12</td>
</tr>
<tr>
<td>(n = 364)</td>
<td>7/56 15/60</td>
<td>3/24 4/16</td>
</tr>
<tr>
<td>(n = 1093)</td>
<td>15/120 15/120</td>
<td>4/32 4/32</td>
</tr>
<tr>
<td>(n = 3280)</td>
<td>15/240 31/248</td>
<td>4/64 5/40</td>
</tr>
</tbody>
</table>

Table 8.1: Lower/upper bounds of majority tree quorums and corresponding nca quorums for a ternary tree containing \(n\) nodes.
Furthermore, the nca quorum algorithm is more resilient to site and communication failures. If the read quorum of a data item \( x \) has length \( l \) and width \( w \) and a transaction contains only one read or write operation on \( x \), then when using the tree quorum algorithm, a read quorum can vary from \((w^l - 1)/(w - 1)\) to \(w^{l-1} \times (w^l - 1)/(w - 1)\) and a write quorum can vary from \([(d - w + 1)^{h-l+1} - 1]/(d - w)\) to \((d - w + 1)^{h-l} \times [(d - w + 1)^{h-l+1} - 1]/(d - w)\). However, if the transaction contains both read and write operations on \( x \), the read quorum size and write quorum size can be as large as \(w^{h-l+1} \times (w^l - 1)/(w - 1)\) and \((d - w + 1)^l \times [(d - w + 1)^{h-l+1} - 1]/(d - w)\).

### Table 8.2: Lower/upper bounds of read root quorums and corresponding nca quorums for a ternary tree containing \( n \) nodes.

<table>
<thead>
<tr>
<th>( n )</th>
<th>Tree Quorum</th>
<th>Nca Quorum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td>4</td>
<td>1/2</td>
<td>3/3</td>
</tr>
<tr>
<td>13</td>
<td>1/4</td>
<td>7/7</td>
</tr>
<tr>
<td>40</td>
<td>1/8</td>
<td>15/15</td>
</tr>
<tr>
<td>121</td>
<td>1/16</td>
<td>31/31</td>
</tr>
<tr>
<td>364</td>
<td>1/32</td>
<td>63/63</td>
</tr>
<tr>
<td>1093</td>
<td>1/64</td>
<td>127/127</td>
</tr>
<tr>
<td>3280</td>
<td>1/128</td>
<td>255/255</td>
</tr>
</tbody>
</table>

### Table 8.3: Lower/upper bounds of log write quorums and corresponding nca quorums for a ternary tree containing \( n \) nodes.

<table>
<thead>
<tr>
<th>( n )</th>
<th>Tree Quorum</th>
<th>Nca Quorum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td>4</td>
<td>1/3</td>
<td>2/2</td>
</tr>
<tr>
<td>13</td>
<td>1/9</td>
<td>3/3</td>
</tr>
<tr>
<td>40</td>
<td>1/27</td>
<td>4/4</td>
</tr>
<tr>
<td>121</td>
<td>1/81</td>
<td>5/5</td>
</tr>
<tr>
<td>364</td>
<td>1/243</td>
<td>6/6</td>
</tr>
<tr>
<td>1093</td>
<td>1/729</td>
<td>7/7</td>
</tr>
<tr>
<td>3280</td>
<td>1/2187</td>
<td>8/8</td>
</tr>
</tbody>
</table>
respectively, if all the nodes at the upper levels of a tree failed at the same time. If $t$ upper levels of the tree have failed, then since $l_r + l_w > h$, $t$ must be less than $h/2$ to guarantee the existence of both read and write quorums on $x$. In the worst case the read quorum size can increase to $w^{h/2} \times (w^{h/2+1} - 1)/(w - 1)$ and the write quorum can increase to $(d - w + 1)^{h/2+1} \times (d - w + 1)^{h/2} - 1)/(d - w)$ in a partitioned network. If $t$ is greater than $h/2$, then only one of the read and write quorums on $x$ is available. As a result, a transaction containing both read and write operations on $x$ will not succeed in this case. Thus, the tree quorum algorithm can not tolerate more than $w^{h/2} - 1$ failures if all the nodes with NODE numbers from 1...$w^{h/2} - 1$ fail. Unlike the tree quorum algorithm, the nca quorum algorithm can tolerate up to $n - 1$ failures in a network of $n$ nodes.
CHAPTER 9

CONCLUSION

The proposed replica control algorithm can tolerate up to \( n - 1 \) failures for a network containing \( n \) nodes, which is optimal. That each read quorum and write quorum is computed independently without considering the size of the other is our major contribution. Thus, both read quorum sizes and write quorum sizes can be expected to be constant in the lightly loaded system, at most \( \log_2 n \) for a failure-free network, and \( \lceil(n + 1)/2 \rceil \) for a partitioned network. Unlike Agrawal and El Abbadi's tree quorum, if read quorums have dimensions \( < l, w > \), then write quorum must have dimensions \( < h - l + 1, d - w + 1 > \), where \( h \) is the height and \( d \) is the degree of the tree. Thus, the size of a read quorum may as small as \( (w^l - 1)/(w - 1) \) if all copies in the quorum are from the upper levels of the tree. In the worst case, the size may increase to \( w^{l-1} \ast (w^l - 1)/(w - 1) \) when all copies in the quorum are from the lower levels of the tree. Similarly, the size of write quorum can vary from \( [(d-w+1)^{h-l+1} - 1]/(d-w) \) to \( (d-w+1)^{h-l} \ast [(d-w+1)^{h-l+1} - 1]/(d-w) \). Compared to the Agrawal and El Abbadi algorithm, the nca quorum algorithm requires fewer messages and has higher fault tolerance.
BIBLIOGRAPHY


