AN EFFICIENT QUORUM STRUCTURE FOR DISTRIBUTED MUTUAL EXCLUSION

BY

SURAKIT TANAVUTIKAI

Bachelor of Business Administration

Institute of Technology

and Vocational Education

Bangkok, Thailand

1988

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirement for
the Degree of
MASTER OF SCIENCE
July, 1993

OKLAHOMA STATE UNIVERSITY

AN EFFICIENT QUORUM STRUCTURE FOR

DISTRIBUTED MUTUAL EXCLUSION

Thesis Approved:

Huizhu Lu

Thesis Advisor

J Chandler

Mitchell L Neilsen

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express my appreciation to my major advisor, Dr. Huizhu Lu, for her encouragement, advice, and guidance throughout my graduate program. I would also like to thank my committee members, Dr. K. M. George and Dr. John P. Chandler for their assistance.

Appreciation is also expressed to Dr. Mitchell L. Neilsen for his expert guidance and much valued counsel during the work of this thesis.

Most of all, I would like to express a special thank you to my parents, Somkeat and Pensri Tanavutikai, for their endless support, encouragement, and countless sacrifices throughout my academic career. I also want to thank the other members of my family for their support.

Finally, I want to thank my friends and all the people who make this thesis possible.

TABLE OF CONTENTS

Chapter page

I. INTRODUCTION 1

Motivation 3
Terminology 3
Outline of the thesis 4

II. LITERATURE REVIEW 6

Majority consensus 6
√N algorithm for mutual exclusion 7
Coterie 10
Tree quorums 11

III. PROPOSED APPROACH 15

Introduction 15
Finding generators 16
Forming quorums 21
Simulation details 23
Special case 26

IV. ANALYSIS AND RESULTS 28

Analysis 28
Results 32
Complexity 48

V. CONCLUSIONS 49

Conclusions 49
Future work 50

REFERENCES 51

APPENDIXES 54

APPENDIX A - INPUT QUORUMS FOR MAEKAWA'S ALGORITHM AND THE PROPOSED METHOD USED IN THE ANALYSIS SECTION 55

APPENDIX B - ANALYSIS AND SIMULATION PROGRAMS 61

LIST OF TABLES

Table page

I. Expected size of quorums of the four methods 35

II. Comparison of the availabilities of the four methods with N = 5 37

III. Comparison of the availabilities of the four methods with N = 7 39

IV. Comparison of the availabilities of the four methods with N = 9 41

V. Comparison of the availabilities of the four methods with N = 11 43

VI. Comparison of the availabilities of the four methods with N = 13 45

VII. Comparison of the availabilities of the four methods with N = 15 47

LIST OF FIGURES

Figure page

1. Result quorums of N = 13 using Maekawa's algorithm 8
2. Result quorums of N = 8 using Maekawa's algorithm 9
3. A given tree for generating tree quorums 12
4. Representing quorums by using binary digits 23
5. Recursive procedure for forming generators 24
6. Pseudocode for finding quorums 25
7. An example of the substitution method 27
8. Original tree 31
9. Divided subtree 31
10. Graph comparing the expected size of quorums of the four methods 34
11. Graph comparing the availabilities of the four methods with N = 5 36
12. Graph comparing the availabilities of the four methods with N = 7 38
13. Graph comparing the availabilities of the four methods with N = 9 40
14. Graph comparing the availabilities of the four methods with N = 11 42
15. Graph comparing the availabilities of the four methods with N = 13 44
16. Graph comparing the availabilities of the four methods with N = 15 46

CHAPTER I

INTRODUCTION

A distributed system is a system that consists of a set of computers connected by a communication network. Its objective is to provide low cost availability and consistency of resources. Many distributed systems use replication to increase availability of the resources. As the number of resources increases, it becomes more difficult to provide consistency, especially when failures occur in some parts of the system. In a distributed system with replicated data, each node in the system may store identical information. To ensure consistency between the copies, the system must not allow two or more write operations to be performed simultaneously; otherwise, the system may have different copies of data. When updating data at a node, any other nodes in the system should be able to notice those changes. When a read operation is performed, the system has to make sure that the read operation reads the latest version of data. It is necessary to have mutual exclusion mechanisms to control consistency of resources in distributed systems. A mutual exclusion mechanism that has low communication cost and works even when nodes and communication lines have failed is preferred.

There are many methods available to achieve mutual exclusion in distributed system. Thomas proposed a majority consensus approach [Thomas 79]. Gifford presented weighted voting [Gifford 79]. Weighted voting requires votes to be assigned to each node. To achieve mutual exclusion, a set of nodes that has at least a sum of votes equal to a read (write) threshold must be obtained in order to perform a read (write) operation. Such a set of nodes is called a quorum. Maekawa proposed a $\sqrt{N}$ algorithm for mutual exclusion [Maekawa 85]. The algorithm uses a logical structure based on finite projective planes to find quorums for mutual exclusion. Garcia introduced the concept of coterie [Garcia 85]. The paper gives definitions of dominated and nondominated coteries. Nondominated coteries are more tolerant to node and communication line failures. Agrawal proposed an efficient and fault-tolerant solution for distributed mutual exclusion [Agrawal 91]. The algorithm selects nodes in a tree to form quorums. Neilsen introduced composition as a method for constructing coteries [Neilsen 92a]. Composition combines nonempty structures to construct new larger structures. The result of combining nondominated coteries is a new larger nondominated coterie.

This thesis discusses some of the previous works and a new method to construct quorums for distributed mutual exclusion. The new method uses a difference set algorithm to construct generators. Then, it uses the generators to form quorums. The constructed quorums ensure distributed mutual exclusion.

Motivation

In many distributed systems it is necessary to have a mutual exclusion mechanism that works even when nodes fail or the communication lines are broken [Garcia 85]. For instance, in a system that manages replicated data, it is difficult to provide consistency of the replicated data. Users may update data at different nodes (stations) at any time. When failed nodes recover, the data at those nodes may be obsolete. These situations can lead to inconsistency between the replicated data. Thus, a mutual exclusion mechanism that works when nodes or communication lines fail is necessary. Quorum-based protocols have been proposed to effectively tolerate node and communication line failures in distributed systems. Sets of nodes (quorums) that have at least one node in common with each other can guarantee mutual exclusion. It is necessary to have algorithms to construct quorums that can achieve mutual exclusion. The constructed quorums should resist node and communication line failures and require low communication cost. Therefore, this thesis presents an algorithm to construct such quorums.

Terminology

i. Mutual exclusion: If a process P_i is executing in its critical section, then no other process can be executing in its critical section.

ii. Node: a computer in a network (distributed systems).

iii. Quorum: a set of nodes.

iv. N: number of nodes in a distributed system.

v. E: number of nodes in a quorum.

vi. Generator: a set of nodes that can be used to generate quorums. It is also a quorum.

vii. Perfect difference set: Given a number P, a perfect difference set is a set of numbers such that differences between two members of the set can be used to represent every number from 1 to P-1 modulo P. That is, every number from 1 to P-1 can be obtained in one and only one way as the difference of two members of the set [Blattner 68].

viii. Difference set: Given a number P, a difference set is a set of numbers such that differences between two members of the set can be used to represent every number from 1 to P-1 modulo P. Every number from 1 to P-1 is not necessarily obtained in one and only one way as the difference of two members of the set.

Outline of the Thesis

In Chapter II, outlines of the various approaches towards quorum structures for mutual exclusion are given. In Chapter III, the new method to construct quorums for distributed mutual exclusion is proposed. The chapter includes simulation details of the proposed method. The special case of results is discussed as well. Chapter IV discusses analysis and results of the proposed method. An analytical comparison between the proposed method and the

previous method is given. Finally, in Chapter V, conclusions and future work are outlined.

CHAPTER II

LITERATURE REVIEW

This section discusses the previous works of [Thomas 79], [Maekawa 85], [Garcia 85], and [Agrawal 91] on how to define quorums and how to achieve mutual exclusion in distributed systems in more detail.

Majority Consensus

Thomas gives some advantages and disadvantages aspects of having copies of data at a number of network nodes (stations). The first advantage is increased data accessibility. The second is less delay when accessing data because the data is stored at a number of nodes. Thus, it may not be necessary to ask for the data from other nodes. Finally, the nodes in the network share an equal amount of the processing load. There are some disadvantages, such as higher cost for extra devices and problems of maintaining consistency of the copies in the system. The majority consensus algorithm uses $\lceil (N+1)/2 \rceil$ nodes, where N = number of nodes in the system, to perform mutual exclusion. It uses a timestamp to control data consistency. The system updates or accesses data that have the latest timestamp. Majority consensus is a simple and elegant method to achieve mutual exclusion, but it imposes a high communication cost.

$\sqrt{N}$ Algorithm for Mutual Exclusion

Maekawa proposed a mutual exclusion algorithm that uses only $c\sqrt{N}$ messages, where c is a constant between 3 to 5 and N is the number of nodes in a distributed system. The algorithm generates quorums based on finite projective planes. Quorums which have n+1 nodes are said to be of order n. In every known example of a finite projective plane, the order n is of the form p^k, where p is a prime number and k is a positive integer [Blattner 68]. The order n of the form p^k has n^2+n+1 points.

Let $E = n+1$ be number of nodes in each quorum.

Let N = the number of nodes in a distributed system.

Suppose that N = 7, we can say that:

$$N = n^2+n+1$$

$$7 = n^2+n+1 \qquad \text{(substitute N by 7)}$$

so, $\quad n = 2$

Thus, $\quad E = n+1 = 3$

There are four properties used to define quorums in Maekawa's algorithm:

i. For any quorums Qi and Qj, $1 \le i,j \le N$ and $i \ne j$, $Qi \cap Qj \ne \emptyset$.

ii. Qi, $1 \le i \le N$, always contains i.

iii. All quorums are the same size.

iv. Quorums that contain i, $1 \le i \le N$, contain all j, $1 \le j \le N$, $j \ne i$. For example, from Figure 1, quorums that contain node 1 should contain nodes 2, 3,..., 13.

Property ii reduces the number of sending and receiving messages. Properties iii and iv are included to have a truly distributed algorithm [Maekawa 85]. Figure 1 is an example of quorums for N = 13 (implies that E = 4).

For some number N, there may not exist a correspondence order n (nodes in a quorum - 1) of the form p^k. The algorithm finds an n′, where n′ is the smallest number that is larger than n, that has the form p^k. Then, it uses n′ to form quorums that satisfy the four properties above. After that, the algorithm cuts off quorum Qj, where j > N. Then, it replaces nodes that are greater than N and appear in Qi, where i ≤ N, with nodes that are smaller than or equal to N.

Q_1 = {1,2,3,4}

Q_5 = {1,5,6,7}

Q_8 = {1,8,9,10}

Q_{11} = {1,11,12,13}

Q_2 = {2,5,8,10}

Q_6 = {2,6,9,12}

Q_7 = {2,7,10,13}

Q_{10} = {3,5,10,12}

Q_3 = {3,6,8,13}

Q_9 = {3,7,9,11}

Q_{13} = {4,5,9,13}

Q_4 = {4,6,10,11}

Q_{12} = {4,7,8,12}

Figure 1. Result quorums of N = 13 using Maekawa's algorithm

For instance, let N = 8. It is impossible to have $n = p^k$. The algorithm found that $n' = 3$ is the smallest number that is larger than n (calculating value of n by replacing value of N in the equations above, giving n = 2.193). Using $n' = 3$ instead of n in the equation above implies that N = 13 ($N=3^2+3+1$). The quorums of N = 13 are the same as quorums in Figure 1. Then, the algorithm needs to cut the overhead quorums Qj, where j > N. That means the algorithm cuts quorums Q_9, Q_{10}, Q_{11}, Q_{12}, and Q_{13}. After that, it replaces nodes 9, 10, 11, 12, and 13 that appear in $Q_1..Q_8$ by nodes 4, 5, 6, 7, and 8 respectively. Figure 2 gives the result quorums of N = 8.

$Q_1 = \{1,2,3,4\}$

$Q_5 = \{1,5,6,7\}$

$Q_8 = \{1,8,4,5\}$

$Q_2 = \{2,5,8,6\}$

$Q_6 = \{2,6,4,7\}$

$Q_7 = \{2,7,5,8\}$

$Q_3 = \{3,6,8\}$

$Q_4 = \{4,6,5\}$

Figure 2. Result quorums of N = 8 using Maekawa's algorithm

Note that, in Q_3 node 13 is supposed to be replaced by node 8, but node 8 is already an element of Q_3. Thus, it is not necessary to add another node 8 to Q_3. Q_4 is the same situation as Q_3.

Maekawa's algorithm requires only $O(\sqrt{N})$ nodes in each quorum for mutual exclusion. This number is optimal for distributed algorithms [Maekawa 85]. It is much better than those of [Thomas 79]. While N is of the form n^2+n+1 (remember that n is the number of nodes in a quorum -1) and n is of the form p^k, the constructed quorums are symmetric. This means that it is a truly distributed system. However, if n is not of the form p^k, the constructed quorums are not balanced. Another disadvantage of the algorithm is that it provides a small number of quorums (equal to N) for mutual exclusion.

Coterie

Garcia-Molina and Barbara define properties of coteries that are very useful in distributed mutual exclusion.

Let U be the set of nodes in a system. A set of quorums C is a coterie under U if and only if:

i. $G \in C \Rightarrow G \neq \emptyset$ and $G \subseteq U$.

ii. (Intersection property) if G, H $\in$ C, then G and H must have at least one node in common.

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

Property ii can guarantee mutual exclusion. Property iii reduces the redundancy of quorums that provide solutions to mutual exclusion problems. Following is an example of a coterie.

Let U = {1,2,3}

C = {{1,2}, {1,3}, {2,3}}

C is a coterie under U. It consists of quorums {1,2}, {1,3}, and {2,3}. C has all the three properties above.

There are two kinds of coteries: dominated and nondominated coteries. Let C be a coterie under U. C is dominated if and only if there exist a quorum $G \subseteq U$ such that

i. G is not a superset of any quorum in C.

ii. G has the intersection property. This means that for all quorums $H \in C$, $G \cap H \neq \emptyset$.

A coterie C under U is dominated if there is another coterie, D, under U that dominates C. If there is no such coterie, then C is nondominated. Nondominated coteries are more fault-tolerant to node and communication line failures than the coteries they dominate.

Let U = {1,2,3,4}

Let C and D be coteries under U.

C = {{1,2,3}, {1,2,4}, {1,3,4}, {2,3,4}}

D = {{1,2}, {1,3}, {1,4}, {2,3,4}}

Obviously, D dominates C; each quorum in D dominates C. D resists more fault than C. For instance, if the system separates into two groups: {1,2} and {3,4}, there is one active group under D ({1,2}) but none under C.

Tree Quorums

Agrawal and El Abbadi combined the idea of logical structures and coteries to develop an efficient and fault-tolerant solution to mutual exclusion problems, called tree quorums. The tree quorum selects nodes in a binary tree to

form quorums. It starts selecting nodes from the root and ending with any of the leaves. If a path from the root to a leaf has an inaccessible leaf, then that path cannot form a quorum. If a nonleaf node in a path is inaccessible, then paths starting with children of the failing node and ending with leaves are used instead. Agrawal showed that the algorithm not only works with binary trees but also works with trees in which each nonleaf node has degree d [Agrawal 91]. Neilsen showed that the algorithm can be applied to any tree in which each nonleaf node has at least two children [Neilsen 92a]. They also proved that the algorithm produces a nondominated coterie. Following is an example of how to select nodes from a given tree in Figure 3.

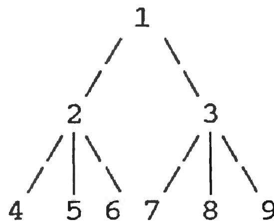

Figure 3. A given tree for generating tree quorums

The following assumption can be made from the given tree:

If all nodes in the tree are accessible, quorums can be built as {1,2,4}, {1,2,5}, {1,2,6}, {1,3,7}, {1,3,8}, {1,3,9}.

If node 1 is inaccessible, nodes 2 and 3 (children of node 1) will be used instead. The constructed quorums are {2,3,4,7}, {2,3,4,8}, {2,3,4,9}, {2,3,5,7}, {2,3,5,8},

{2,3,5,9}, {2,3,6,7}, {2,3,6,8}, and {2,3,6,9}.

If node 2 is inaccessible, the quorum in the path is {1,4,5,6}.

If node 3 is inaccessible, {1,7,8,9} is the quorum in the path.

If nodes 1 and 2 are inaccessible, the constructed quorums are {3,4,5,6,7}, {3,4,5,6,8}, and {3,4,5,6,9}.

If nodes 1 and 3 are inaccessible, the constructed quorums are {2,4,7,8,9}, {2,5,7,8,9}, and {2,6,7,8,9}.

Finally, if nodes 1, 2, and 3 have failed, then {4,5,6,7,8,9} is a quorum.

The collection of quorums is a coterie.

Tree quorum algorithm provides several choices of quorums to a node requesting mutual exclusion. This is a good feature of the algorithm. When a node malfunctions, any other nodes that occur in the same quorums with that node still can achieve mutual exclusion because they may appear in other quorums. For instance, in the coterie above, if node 1 is inaccessible, all quorums that contain node 1 cannot achieve mutual exclusion. Nodes 2,3,4,5,6,7, 8, and 9 can still achieve mutual exclusion. In the best case the algorithm requires permissions from only $\lceil \log N \rceil$ nodes. In the worst case it requires $\lceil (N+1)/2 \rceil$ nodes (for binary trees) [Agrawal 91]. The algorithm requires more nodes to achieve mutual exclusion when nodes in the upper levels of the tree are inaccessible. For example, if node 1 is inaccessible, the algorithm requires at least 4 nodes to perform mutual exclusion. Nodes bear different

responsibility to control mutual exclusion. Some nodes appear more and some nodes appear less in the quorums. Thus when nodes that appear in more quorums are not working, there are fewer working quorums left for mutual exclusion. For example, if node 2 is inaccessible, there are only 9 quorums left. If node 4 is inaccessible, there are 14 quorums left to provide mutual exclusion.

CHAPTER III

PROPOSED APPROACH

Introduction

The proposed method is a new method to construct quorums that can be effectively used in distributed mutual exclusions. The construction of quorums includes two procedures:

i. Finding generators by using a difference set algorithm. The difference set algorithm is given in the next section.

ii. Forming quorums by using the generators from step i.

The proposed method uses the difference set algorithm to find generators because the difference set algorithm provides generators that can be used to construct quorums. The constructed quorums have at least one node in common with each other which guarantees mutual exclusion. The difference set algorithm requires a small number of nodes to form a generator. The number of nodes needed to form a generator are the same as the number of nodes needed to form a quorum. Thus, quorums constructed from the generator consist of a small number of nodes as well. This means that each quorum requires low message cost to achieve mutual exclusion.

The proposed method requires a small number of nodes ($O(\sqrt{N})$) in each quorum and can still achieve mutual exclusion. The proposed solution also provides a reasonable level availability.

Finding Generators

The proposed method requires knowing the number of nodes in a distributed system, N. It finds generators from a given N by using a difference set algorithm. There are three steps to find generators:

Step I. Finding the number of nodes in a quorum, E.

Step II. Finding all possible generators for a given N.

Step III. Applying the difference set algorithm to all possible generators from Step II to obtain generators.

Step III requires knowledge of a perfect difference set, a difference set, and their applications. But it is important to understand how the perfect difference set and the difference set work before the last step because formulas in Step I also involve the perfect difference set and the difference set.

Perfect Difference Set and Difference Set

This section explains how to obtain the perfect difference set and the difference set. A method that is used to find the perfect difference set is called the perfect difference set algorithm. A method that is used to

find the difference set is called the difference set algorithm. The difference sets are used as generators.

Perfect Difference Set. An easy way to explain how the perfect difference set algorithm works is by giving an example.

Let N = 7.

Let E = 3 (Step I in the following section explains how to find E).

Let S = {1,2,4} (a set that contains E elements of nodes in N).

To see that S is a perfect difference set, consider the following differences; all arithmetic is modular.

```
          1
          2
          4
       -------
          2       (4 - 2)
          3       (4 - 1)
          1       (2 - 1)
       -------    end of the first half
          6       ((1 - 2) mod 7)
          4       ((1 - 4) mod 7)
          5       ((2 - 4) mod 7)
       =======    end of the second half
```

The differences (2,3,1,6,4, and 5) are every number from 1 to N-1. The set S is called a perfect difference set because the result of the method, every number from 1 to N-1, can be obtained in one and only one way as the difference of two members of the set S [Blattner 68]. After applying the perfect difference set algorithm, not all sets that contain E elements of nodes in N can produce a number from 1 to N-1. For example, let N = 7, let E = 3, and let set S = {1,2,3}. The differences are computed:

```
      1
      2
      3
   -------
      1       (3 - 2)
      2       (3 - 1)
      1       (2 - 1)
   -------    end of the first half
      6       ((1 - 2) mod 7)
      5       ((1 - 3) mod 7)
      6       ((2 - 3) mod 7)
   =======    end of the second half
```

The result is the numbers 1, 2, 1, 6, 5, and 6 which do not include every number form 1 to N-1. Thus, the set S = {1,2,3} is not a perfect difference set.

Difference Set. The reason for introducing the difference set algorithm is that a perfect difference set does not exist for most values of N. Thus, generators cannot be formed for those N. The difference set algorithm relaxes a condition in the perfect difference set algorithm in order to find difference sets for different values of N, E, and S. After applying the difference set algorithm to any set S that contains E elements of nodes in N, if a set S results in every number from 1 to N-1, not necessarily obtained in one and only one way, then the set S is a difference set (generator). For example, let N = 5, let E = 3, and let set S = {1,2,3}.

```
      1
      2
      3
   -------
      1       (3 - 2)
      2       (3 - 1)
      1       (2 - 1)
   -------    end of the first half
      4       ((1 - 2) mod 5)
      3       ((1 - 3) mod 5)
      4       ((2 - 3) mod 5)
   =======    end of the second half
```

The result is the numbers 1, 2, 1, 4, 3, and 4, which include every number from 1 to N-1. Thus, the set S is a difference set. Note that there is more than one way to find the differences between two members in the set S that includes every number from 1 to N-1. The difference set is used as a generator. Thus the set {1,2,3} is a generator.

Step I

Given N nodes in a distributed system, the proposed method finds a number of nodes, E, that are needed to form a quorum. Referred back to the perfect difference set section, the result of the perfect different set algorithm is N-1 numbers. From a given N, after applying the algorithm, the value of E can be computed. The algorithm chooses two elements out of E elements to find the difference between the two elements. All possibilities of choosing two elements out of E elements is equal to $\binom{E}{2}$. The differences are computed up to two of $\binom{E}{2}$ times. From the above information, the following formulas can be derived:

$$N = 2\binom{E}{2} + 1$$

$$N = E^2 - E + 1$$

$$E^2 - E + 1 - N = 0$$

Thus, $$E = \frac{1 \pm \sqrt{1 - 4(1-N)}}{2}$$

Since E is positive,

$$E = \frac{1 + \sqrt{4N - 3}}{2}$$

Since E is discrete the proposed method sets $E = \left\lceil \frac{1 + \sqrt{4N - 3}}{2} \right\rceil$. E is rounded up in order to cover N that does not produce any perfect difference set, but does produce difference set. An example of calculation E from the given N = 5 is given below.

$$E = \left\lceil \frac{1 + \sqrt{4N - 3}}{2} \right\rceil$$

$$E = \left\lceil \frac{1 + \sqrt{4(5) - 3}}{2} \right\rceil$$

$$E = \left\lceil \frac{1 + \sqrt{17}}{2} \right\rceil$$

$$E = \left\lceil 2.56 \right\rceil$$

$$E = 3.$$

From the example, N = 5 cannot be used to produce any perfect difference set, but can be used to produce difference sets which are generators.

Step II

After calculating the value of E in Step I, the propose method uses the given N and the calculated E values to find all possible generators. Forming all possible generators is the same as choosing E elements out of N nodes. For example, if N is equal to 5, then E will be equal to 3 (from Step I). The total number of possible generators is

$$\binom{N}{E} = \binom{5}{3} = 10$$

All possible generators are {1,2,3}, {1,2,4}, {1,2,5}, {1,3,4}, {1,3,5},{1,4,5}, {2,3,4}, {2,3,5}, {2,4,5}, and {3,4,5}.

Step III

In Step III, all the possible generators from Step II is used to find generators. The difference set algorithm is applied to the possible generators. Generators are those possible generators such that any number from 1 to N-1 is the difference of a pair of nodes in the possible generator. Other possible generators are discarded.

Forming Quorums

To form quorums, the proposed method uses the constructed generators from the previous section to form quorums as follow:

Step I. Forming quorums by giving successors to nodes in the constructed generators.

Step II. Check for the intersection property between the generator and existing quorums before constructing quorums from the generators.

Step I

Assigning N-1 successor nodes to each node in a generator. Each successor is constructed by adding 1,2,...,N-1 to the nodes in the generator using modular arithmetic.

For example, let N = 5. Then E = 3 and the set {1,2,3} is a generator. The constructed quorums are as follow:

```
1   2   3   4   5
2   3   4   5   1
3   4   5   1   2.
```

From the third row, 4 is from 3+1, 5 is from 3+2, 1 is from 3+3-5, and 2 is from 3+4-5. The quorums are {1,2,3}, {2,3,4}, {3,4,5}, {4,5,1}, and {5,1,2}.

Step II

Before constructing quorums from another generator, the proposed method checks if the new generator intersects all of the currently constructed quorums. If it intersects all the constructed quorums, then it forms a new group of quorums as mentioned in Step I. The new group of quorums is added to the existing quorums. If the new generator does not intersect all of the existing quorums, then ignore it. Step II is applied to all of the constructed generators.

An example of performing step II is given below. Let set {1,2,4} be another generator of N = 5. The already constructed quorums are {1,2,3}, {2,3,4}, {3,4,5}, {4,5,1}, and {5,1,2}. The generator {1,2,4} intersects all the existing quorums. Thus, quorums are constructed from the generator {1,2,4} as follow:

1	2	3	4	5
2	3	4	5	1
4	5	1	2	3.

After that, the new constructed quorums are added to the existing quorums. The result quorums are {1,2,3}, {2,3,4}, {3,4,5}, {4,5,1}, {5,1,2}, {1,2,4}, {2,3,5}, {3,4,1}, {4,5,2}, and {5,1,3}.

Simulation Details

The approach towards constructing quorums discussed in the Finding generators and Forming quorums sections is simulated on Sequent S81 - DYNIX/ptx. The code is written in C. All the programs, including the analysis programs, are in Appendix B.

The simulation uses bits of unsigned long integers to represent nodes. For instance, if a quorum contains nodes 1, 2, and 5, the integer 19 will be assigned to an unsigned long integer variable to represent the quorum. Figure 4 shows how this scheme works.

Values	4294967295	...	64	32	16	8	4	2	1
Binary digits	0	...	0	0	1	0	0	1	1
Bits	32	...	7	6	5	4	3	2	1

Figure 4. Representing quorums by using binary digits

The simulation program uses bit N+1 to check the upper bound of N. Thus, it can simulate value of N up to 31 nodes. Operations in the main procedure are as follow:

main procedure

Getting N

Finding E

Finding generators

Finding quorums.

Initially, the simulation program bounds the value of N in between 3 and 31, inclusively. Then, it computes the

value of E , Step I. Step II and Step III are performed recursively through every possible generator. Result generators are kept in a binary tree. The recursive procedure for this part is shown in Figure 5.

An example of all possible generators of N = 3 runs through the recursive procedure in the Figure 5 are {1,2}, {1,3}, and {2,3}. The difference set algorithm, function PDS(), is applied to all the possible generators. If the function PDS() returns TRUE, that means the possible

```
Find_generators(N,E)
{
  G = 0; /*unsigned long integer that represents a quorum */
  Depth = 1;
  loop_begin = 1;
  loop_end = E - 1;
 Recursive_find_generators(N,E,Depth,loop_begin,loop_end,G);
}

Recursive_find_generators(N,E,Depth,loop_begin,loop_end,G)
{
   if (Depth <= E)
   {
      for (I=loop_begin; I <= (N - loop_end); I++)
      {
           G = G + Power(BASE,(I-1));
           if (Depth == E)
           {
              if (PDS(N,E,G))
              {
                 if(Search(&generators_tree,G,N)==NOTFOUND)
                    Generators_tree(&generators_tree, G);
              }
           }
           else
              Recursive_find_generators
                 (N, E, Depth+1, I+1, loop_end-1, G);

           G = G - Power(BASE,(I-1));
      }
   }
   else
       return(DONE);
}
```

Figure 5. Recursive procedure for forming generators

generator is a generator (new generator). The function Search() searches existing generators in the generators tree. If the function Search() returns NOTFOUND, that means the existing generators do not produce quorums that are equal to the new generator. Then, the new generator is inserted into the generator's tree by calling the function Generators_tree().

The function Power() calculates integer value that has binary digits representing nodes in a quorum. The BASE variable is defined as integer 2. That means the function Power() returns the integer that has the form power of 2.

After constructing all generators, the simulation program starts to form quorums from the constructed generators. Figure 6 shows the pseudocode for this part.

```
Find_Quorums()
{
     Outside_recursive_inorder(&generators_tree);
}
Outside_recursive_inorder(&generators_tree);
{
     Outside_recursive_inorder(&left_subtree);
     Inside_recursive_inorder(generator, &generators_tree);
     Outside_recursive_inorder(&right_subtree);
}
Inside_recursive_inorder(generator,&generators_tree);
{
     Inside_recursive_inorder(&left_subtree);
     Build_quorums(generator, &quorums_tree);
     Inside_recursive_inorder(&right_subtree);
}
Build_quorums(generator, quorums_tree)
{
     If the generator intersects all existing quorums in the
     quorums tree, then the generator is used to form
     quorums and add the new generated quorums to the
     quorums tree, or else ignore it.
}
```

Figure 6. Pseudocode for finding quorums

The idea of the pseudocode in Figure 6 is the same as doing nested for loops but with the generator's tree in inorder manner. The Build_quorum() performs the same as Step I and Step II in the Forming quorum section.

Special Case

After running the simulation program, it appeared that for some values of N such as 20, 29, and 30, the proposed method does not provide any quorums. The results after applying the difference set algorithm to those N are not every number from 1 to N-1. This means that generators for those N do not exist. Thus, quorums cannot be formed.

To solve this problem, a substitution method similar to those of Maekawa is introduced. First, let N be the number that cannot be used to generate quorums. Next, the replacing method finds an N′, where N′ is the smallest number that is larger than N, that can be used to generate quorums. Then, the proposed method is applied to construct quorums for N′. Finally, the substitution method is used to replace nodes in the constructed quorums that are greater than N with nodes that are smaller than or equal to N. The resulting quorums after substitution have at least one node in common with each other. Thus, the resulting quorums can still achieve mutual exclusion. An example of the substitution method with N = 29 is given in Figure 7.

```
====================
N = 31     E = 6
====================
====================
GENERATOR(S) :
1 2 5 11 13 18
END.
--------------------
QUORUMS :
  1  2  5 11 13 18     2  3  6 12 14 19     3  4  7 13 15 20
  4  5  8 14 16 21     5  6  9 15 17 22     6  7 10 16 18 23
  7  8 11 17 19 24     8  9 12 18 20 25     9 10 13 19 21 26
 10 11 14 20 22 27    11 12 15 21 23 28    12 13 16 22 24 29

 13 14 17 23 25 30    14 15 18 24 26 31     1 15 16 19 25 27
  2 16 17 20 26 28     3 17 18 21 27 29     4 18 19 22 28 30
  5 19 20 23 29 31     1  6 20 21 24 30     2  7 21 22 25 31
  1  3  8 22 23 26     2  4  9 23 24 27     3  5 10 24 25 28
  4  6 11 25 26 29     5  7 12 26 27 30     6  8 13 27 28 31
  1  7  9 14 28 29     2  8 10 15 29 30     3  9 11 16 30 31
  1  4 10 12 17 31
END.
====================
```

The substitution method modifies quorums of N = 31 by replacing nodes 30 and 31 with nodes 28 and 29, respectively. The result is as follows:

```
====================
N = 29     E = 6
====================
====================
GENERATOR(S) :
1 2 5 11 13 18
END.
--------------------
QUORUMS :
 1  2  5 11 13 18     2  3  6 12 14 19     3  4  7 13 15 20
 4  5  8 14 16 21     5  6  9 15 17 22     6  7 10 16 18 23
 7  8 11 17 19 24     8  9 12 18 20 25     9 10 13 19 21 26
10 11 14 20 22 27    11 12 15 21 23 28    12 13 16 22 24 29
13 14 17 23 25 28    14 15 18 24 26 29     1 15 16 19 25 27
 2 16 17 20 26 28     3 17 18 21 27 29     4 18 19 22 28
 5 19 20 23 29        1  6 20 21 24 28     2  7 21 22 25 29
 1  3  8 22 23 26     2  4  9 23 24 27     3  5 10 24 25 28
 4  6 11 25 26 29     5  7 12 26 27 28     6  8 13 27 28 29
 1  7  9 14 28 29     2  8 10 15 29 28     3  9 11 16 28 29
 1  4 10 12 17 29
END.
====================
```

Figure 7. An example of the substitution method

CHAPTER IV

ANALYSIS AND RESULTS

Analysis

There are two important aspects of analyzing quorum structures:

i. Message cost

ii. Availability

In distributed systems, quorums should have a low number of nodes. The fewer number of nodes required to form a quorum, the lower number of messages required to obtain mutual exclusion. Availability of forming quorums (reliability) is another important aspect. Availability is used to measure performance of algorithms that are used to construct quorums. Algorithms that provide higher availability are preferred.

In this thesis, the analysis of the message cost and the availability of forming quorums are based on [Maekawa 85], [Agrawal 91], and [Neilsen 92c] papers. The thesis analyzes and compares the two important aspects of the majority consensus, the tree quorums, Maekawa's algorithm, and the proposed method.

Message Cost

The number of nodes required to achieve mutual exclusion in the majority consensus are $\lceil (N+1)/2 \rceil$. In the tree quorums, the number of nodes required to form a quorum are highly dependent on nodes in the upper levels of a tree. A formula for calculating the average number of nodes required to form a quorum given in Agrawal's paper is used. The formula is as follows:

$$C_{l+1} = f(C_l + 1) + (1 - f)(2C_l),$$

where C_l is the average number of nodes required to form quorum in a tree of level l,

f is the fraction of quorums that include the root of level l+1, and

C_0 is equal to one.

Maekawa's algorithm requires $O(\sqrt{N})$ nodes. The proposed method also requires $O(\sqrt{N})$ nodes.

Availability

The thesis measures availability by assigning equal probability of node operations to each node in the system. This is done in order to compare clearly the results among each algorithm. The assigned probabilities start from 0.05 to 0.95.

Availability of the majority consensus is measured as follows:

Let $K = \lfloor N/2 \rfloor$

Let P = Probability that nodes are operational

Availability = Probability (K+1 nodes are operational)

+
.
.
.
+

Probability (N nodes are operational)

$$= \binom{N}{K+1} P^{(K+i)}(1-P)^{(N-(K+i))} + \ldots + \binom{N}{N} P^{N}(1-P)^{(N-N)}$$

Availability of the tree quorums, Maekawa's algorithm, and the proposed method are measured by using methods in the Neilsen's paper. The following are the methods used.

The probability that only the nodes in a quorum Q are operational, is defined by:

$$Pr(Q,U) = \prod_{i \in Q}(P_i) \prod_{i \in (U-Q)}(1-P_i) \qquad (1)$$

where P_i is the steady-state probability that node i is operational, U is all nodes in a system. The availability of coterie C is defined by

$$Avail(C) = \sum_{Q \in A(C)} Pr(Q,U) \qquad (2)$$

where A(C) is the corresponding acceptance set. The acceptance set is the set of all subsets of U that contain a quorum of C.

The constructed quorums of Maekawa's algorithm and the proposed method are also coteries. Thus,the definitions (1)

and (2) above can be applied to measure the availability of the two algorithms.

The availability of the tree quorums is measured a little different. A tree that is used to form quorums is divided into subtrees each of which has tree nodes. Supposed that the tree consists of five nodes then it can be divided into subtrees, as shown in Figure 8 and Figure 9.

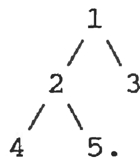

Figure 8. Original tree

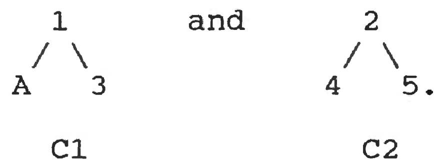

Figure 9. Divided subtrees

Note that A is a new node that represents the subtree C2. This method is from the composition by Neilsen, [Neilsen 92a], [Neilsen 92b], and [Neilsen 92c].

To compute the availability of the tree, first, use tree quorum algorithm to construct quorums from each subtree. Then, use the definitions (1) and (2) above to compute the availability of each subtree. The probability of the new node(s), A, is the probability of subtree(s) it represents.

Let the probability of each node in the tree equal to 0.9. Quorums of the subtree C2 are { {2,4},{2,5},{4,5} }. N of C2 is {2,4,5}.

$$A(C2) = (\{2,4\},\{2,5\},\{4,5\},\{2,4,5\})$$

$$Pr(\{2,4\}) = (0.9)(0.9)(0.1) = 0.0810$$

$$Pr(\{2,5\}) = (0.9)(0.1)(0.9) = 0.0810$$

$$Pr(\{4,5\}) = (0.1)(0.9)(0.9) = 0.0810$$

$$Pr(\{2,4,5\}) = (0.9)(0.9)(0.9) = 0.7290$$

$$\therefore Avail(C2) = 0.0810+0.0810+0.0810+0.7290 = 0.9720.$$

Since node A represents the subtree C2, then, the probability of node A is equal to Avail(C2) = 0.9720.

Quorums of the subtree C1 are { {1,A},{1,3},{A,3} }. N of C1 is {1,A,3}.

$$A(C1) = (\{1,A\},\{1,3\},\{A,3\},\{1,A,3\})$$

$$Pr(\{1,A\}) = (0.9)(0.972)(0.1) = 0.0875$$

$$Pr(\{1,3\}) = (0.9)(0.028)(0.9) = 0.0227$$

$$Pr(\{A,3\}) = (0.1)(0.972)(0.9) = 0.0875$$

$$Pr(\{1,A,3\}) = (0.9)(0.972)(0.9) = 0.7873$$

$$\therefore Avail(C1) = 0.0875+0.0227+0.0875+0.7873 = 0.9850.$$

Results

Results of the simulation are separated into two aspects, the same as the analysis section. The results are shown by graphs in Figure 10 to Figure 16. Table I to Table VII show the source data that are used to plot graphs in Figure 10 to Figure 16, respectively.

Results of Message Cost

Figure 10 and Table I show the comparisons of numbers of nodes needed to form a quorum in the majority consensus, the tree quorums, Maekawa's algorithm, and the proposed method. The numbers used in the tree quorums are the average numbers as mentioned in the analysis section.

The graph in Figure 10 shows that the proposed method uses fewer nodes to form a quorum than majority consensus. Comparing to the tree quorums, if less than seventy-five percent of the root of each level in a tree were used to form quorums, the average quorum size of the tree quorums would be larger than the quorum size of the proposed method.

Results of Availability

Figure 11 to Figure 16 and Table II to Table VII show the comparisons of the availabilities of majority consensus, the tree quorums, Maekawa's algorithm, and the proposed method with differences N.

The results indicate that, if quorums of the proposed method are constructed from more than one generator (N=5,9,15), its availability becomes similar to the availability of the majority consensus when the probability is greater than 0.75, better than the availability of the tree quorums when the probability is greater than 0.65, and better than Maekawa's algorithm when the probability is greater than 0.5.

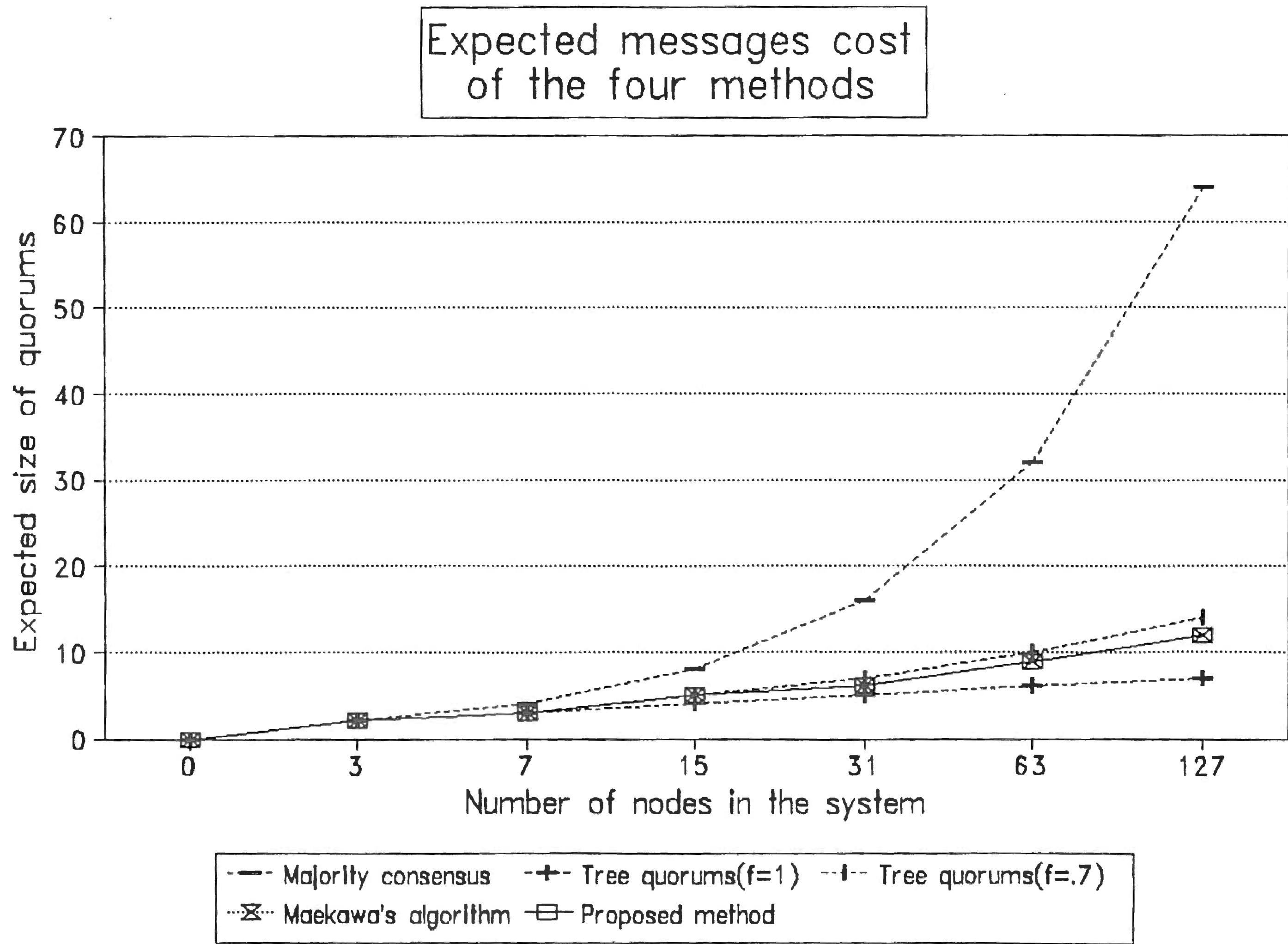

Figure 10. Graph comparing the expected size of quorum of the four methods

TABLE I

EXPECTED SIZE OF QUORUMS OF THE FOUR METHODS

Methods	N = 3	N = 7	N = 15
Majority consensus	2	4	8
Tree quorums(f=1)	2	3	4
Tree quorums(f=.7)	2	3	5
Maekawa's algorithm	2	3	5
Proposed method	2	3	5

TABLE I (Continued)

Methods	N = 31	N = 63	N = 127
Majority consensus	16	32	64
Tree quorums(f=1)	5	6	7
Tree quorums(f=.7)	7	10	14
Maekawa's algorithm	6	9	12
Proposed method	6	9	12

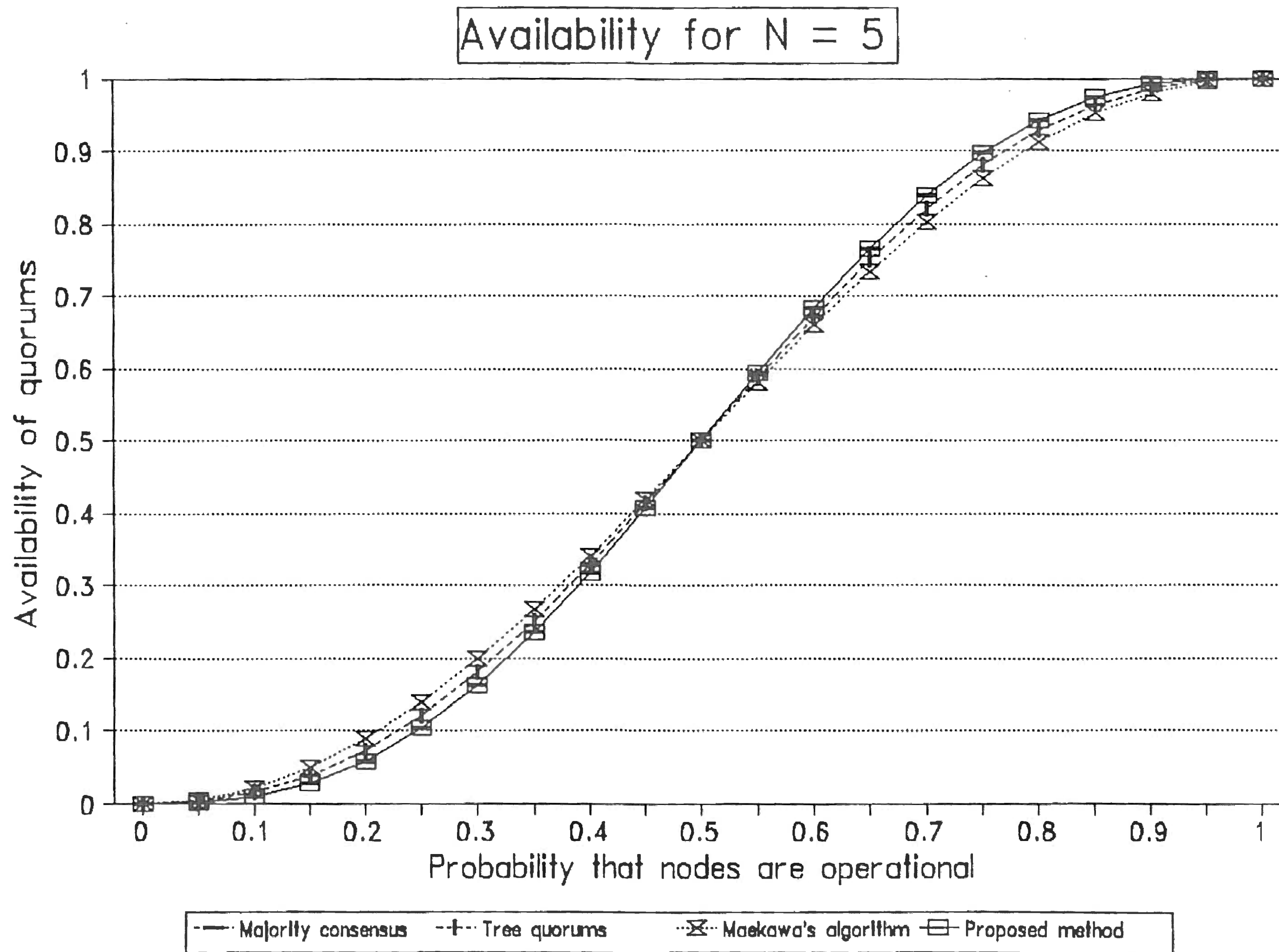

Figure 11. Graph comparing the availabilities of the four methods with N = 5

TABLE II

COMPARISON OF THE AVAILABILITIES OF THE FOUR METHODS WITH N = 5

PROBABILITY	MAJORITY CONSENSUS	TREE QUORUMS	MAEKAWA'S ALGORITHM	PROPOSED METHOD
0.05	0.0002	0.0032	0.0052	0.0012
0.10	0.0086	0.0150	0.0215	0.0086
0.15	0.0266	0.0380	0.0494	0.0266
0.20	0.0579	0.0733	0.0886	0.0579
0.25	0.1035	0.1211	0.1387	0.1035
0.30	0.1631	0.1807	0.1984	0.1631
0.35	0.2352	0.2507	0.2662	0.2352
0.40	0.3174	0.3290	0.3405	0.3174
0.45	0.4069	0.4130	0.4191	0.4069
0.50	0.5000	0.5000	0.5000	0.5000
0.55	0.5931	0.5870	0.5809	0.5931
0.60	0.6826	0.6710	0.6595	0.6826
0.65	0.7648	0.7493	0.7338	0.7648
0.70	0.8369	0.8193	0.8016	0.8369
0.75	0.8965	0.8789	0.8613	0.8965
0.80	0.9421	0.9267	0.9114	0.9421
0.85	0.9734	0.9620	0.9506	0.9734
0.90	0.9914	0.9850	0.9785	0.9914
0.95	0.9988	0.9968	0.9948	0.9988

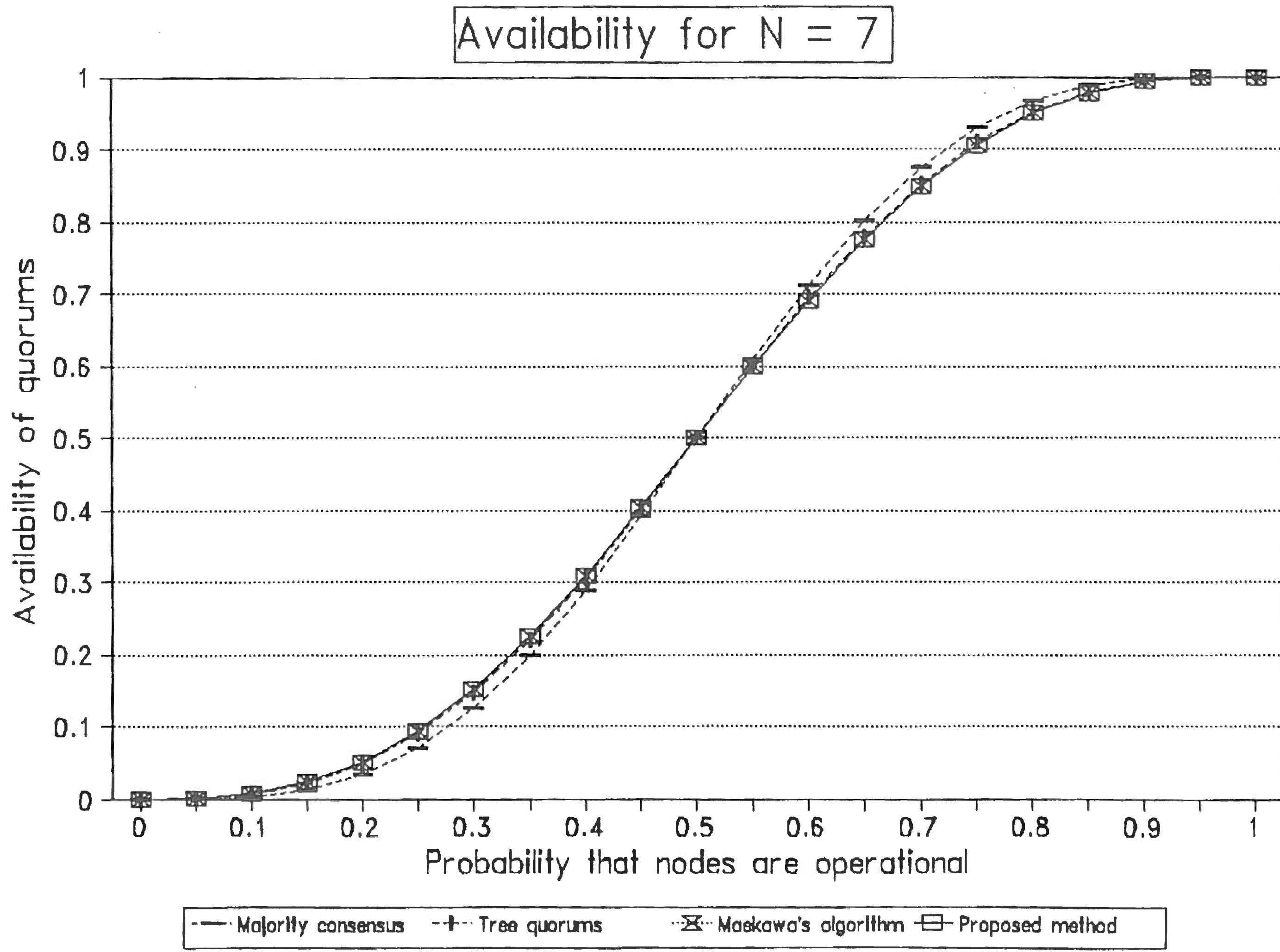

Figure 12. Graph comparing the availabilities of the four methods with N = 7

TABLE III

COMPARISON OF THE AVAILABILITIES OF THE FOUR METHODS WITH N = 7

PROBABILITY	MAJORITY CONSENSUS	TREE QUORUMS	MAEKAWA'S ALGORITHM	PROPOSED METHOD
0.05	0.0002	0.0008	0.0009	0.0009
0.10	0.0027	0.0062	0.0068	0.0068
0.15	0.0121	0.0208	0.0223	0.0223
0.20	0.0333	0.0480	0.0506	0.0506
0.25	0.0706	0.0903	0.0936	0.0936
0.30	0.1260	0.1483	0.1520	0.1520
0.35	0.1998	0.2210	0.2246	0.2246
0.40	0.2898	0.3064	0.3092	0.3092
0.45	0.3917	0.4008	0.4023	0.4023
0.50	0.5000	0.5000	0.5000	0.5000
0.55	0.6083	0.5992	0.5997	0.5997
0.60	0.7102	0.6936	0.6909	0.6909
0.65	0.8002	0.7790	0.7754	0.7754
0.70	0.8740	0.8517	0.8480	0.8480
0.75	0.9294	0.9097	0.9064	0.9064
0.80	0.9667	0.9519	0.9495	0.9495
0.85	0.9879	0.9792	0.9777	0.9777
0.90	0.9973	0.9938	0.9932	0.9932
0.95	0.9998	0.9992	0.9991	0.9991

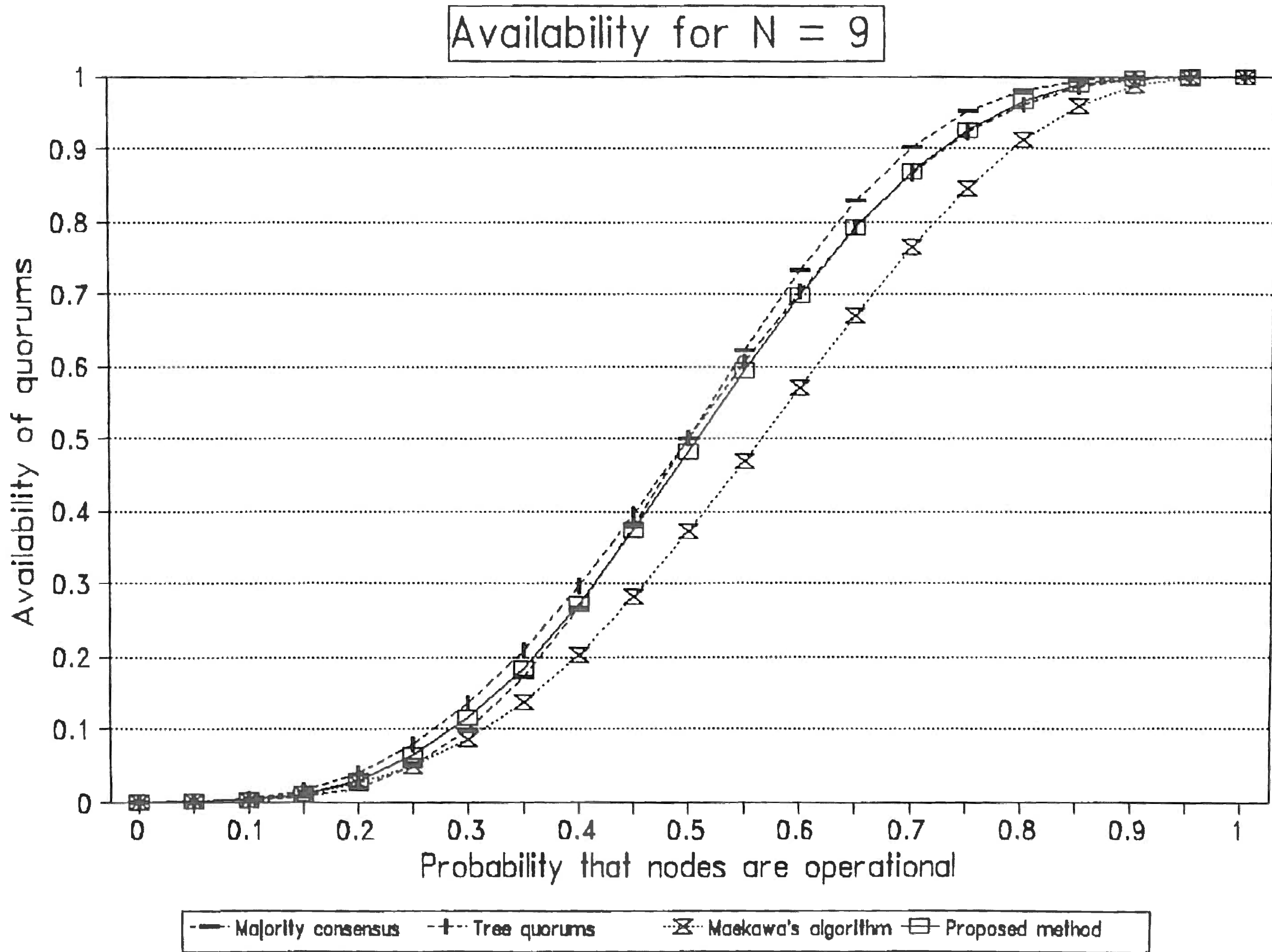

Figure 13. Graph comparing the availabilities of the four methods with N = 9

TABLE IV

COMPARISON OF THE AVAILABILITIES OF THE FOUR METHODS WITH N = 9

PROBABILITY	MAJORITY CONSENSUS	TREE QUORUMS	MAEKAWA'S ALGORITHM	PROPOSED METHOD
0.05	0.0000	0.0005	0.0003	0.0002
0.10	0.0009	0.0046	0.0026	0.0023
0.15	0.0056	0.0164	0.0096	0.0103
0.20	0.0196	0.0400	0.0240	0.0290
0.25	0.0489	0.0788	0.0487	0.0628
0.30	0.0988	0.1346	0.0859	0.1146
0.35	0.1717	0.2076	0.1371	0.1850
0.40	0.2666	0.2955	0.2024	0.2725
0.45	0.3786	0.3948	0.2812	0.3735
0.50	0.5000	0.5000	0.3711	0.4824
0.55	0.6214	0.6052	0.4689	0.5927
0.60	0.7334	0.7045	0.5703	0.6976
0.65	0.8283	0.7925	0.6703	0.7909
0.70	0.9012	0.8654	0.7635	0.8679
0.75	0.9511	0.9212	0.8450	0.9260
0.80	0.9804	0.9600	0.9106	0.9651
0.85	0.9944	0.9836	0.9578	0.9874
0.90	0.9991	0.9954	0.9861	0.9972
0.95	1.0000	0.9995	0.9981	0.9998

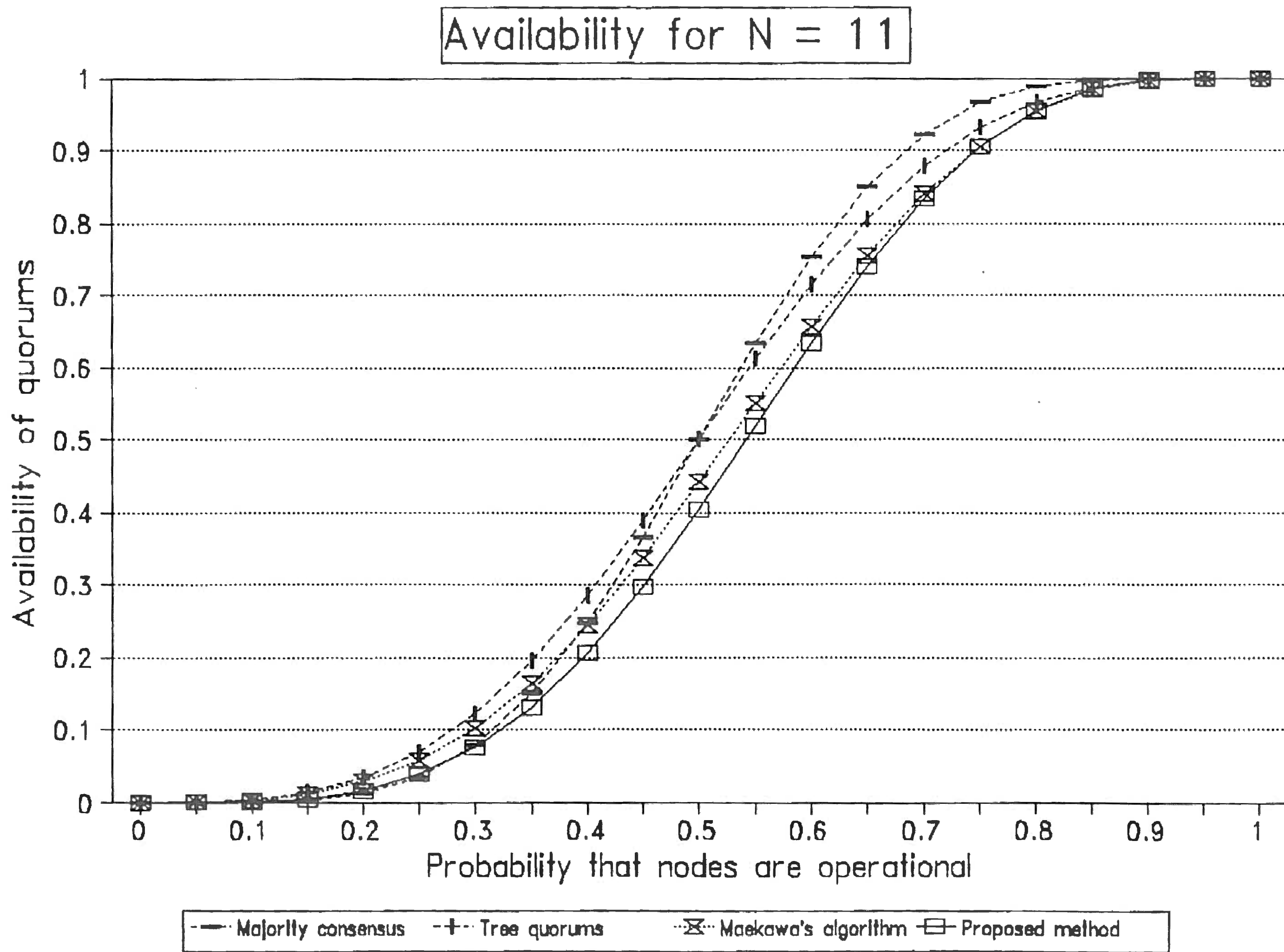

Figure 14. Graph comparing the availabilities of the four methods with N = 11

TABLE V

COMPARISON OF THE AVAILABILITIES OF THE FOUR METHODS WITH N = 11

PROBABILITY	MAJORITY CONSENSUS	TREE QUORUMS	MAEKAWA'S ALGORITHM	PROPOSED METHOD
0.05	0.0000	0.0004	0.0003	0.0001
0.10	0.0002	0.0036	0.0028	0.0011
0.15	0.0027	0.0131	0.0107	0.0054
0.20	0.0117	0.0334	0.0277	0.0165
0.25	0.0343	0.0687	0.0574	0.0386
0.30	0.0782	0.1221	0.1025	0.0756
0.35	0.1487	0.1947	0.1647	0.1306
0.40	0.2465	0.2849	0.2436	0.2048
0.45	0.3669	0.3888	0.3371	0.2969
0.50	0.5000	0.5000	0.4409	0.4033
0.55	0.6331	0.6112	0.5496	0.5180
0.60	0.7535	0.7151	0.6565	0.6332
0.65	0.8513	0.8053	0.7551	0.7408
0.70	0.9218	0.8779	0.8395	0.8333
0.75	0.9657	0.9313	0.9063	0.9053
0.80	0.9883	0.9666	0.9532	0.9550
0.85	0.9973	0.9869	0.9817	0.9837
0.90	0.9997	0.9964	0.9953	0.9964
0.95	1.0000	0.9996	0.9996	0.9997

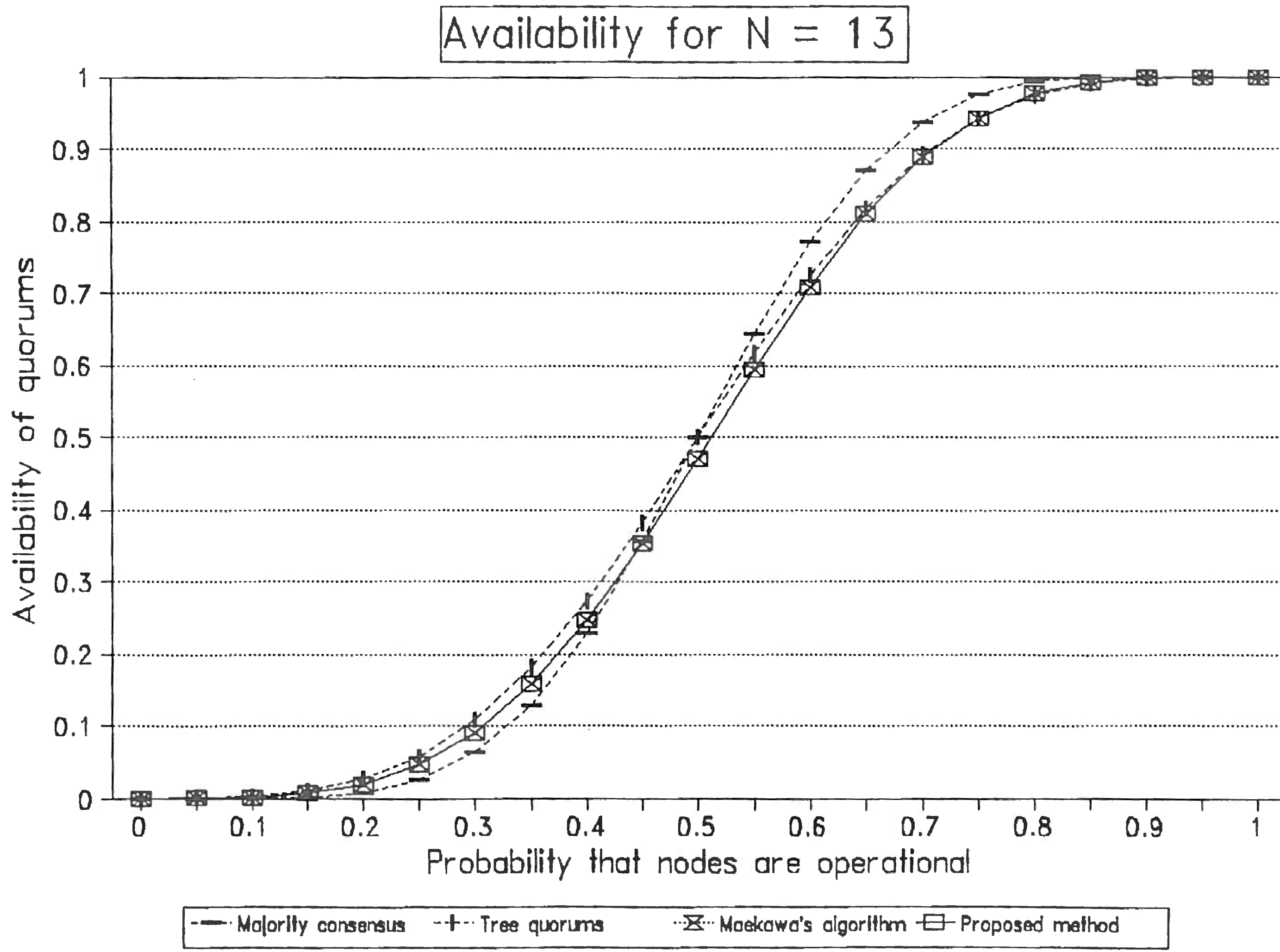

Figure 15. Graph comparing the availabilities of the four methods with N = 13

TABLE VI

COMPARISON OF THE AVAILABILITIES OF THE FOUR METHODS WITH N = 13

PROBABILITY	MAJORITY CONSENSUS	TREE QUORUMS	MAEKAWA'S ALGORITHM	PROPOSED METHOD
0.05	0.0000	0.0002	0.0001	0.0001
0.10	0.0001	0.0022	0.0013	0.0013
0.15	0.0013	0.0094	0.0065	0.0065
0.20	0.0070	0.0264	0.0199	0.0199
0.25	0.0243	0.0583	0.0467	0.0467
0.30	0.0624	0.1094	0.0917	0.0917
0.35	0.1295	0.1817	0.1580	0.1580
0.40	0.2288	0.2743	0.2459	0.2459
0.45	0.3563	0.3828	0.3525	0.3525
0.50	0.5000	0.5000	0.4714	0.4714
0.55	0.6437	0.6172	0.5937	0.5937
0.60	0.7712	0.7257	0.7094	0.7094
0.65	0.8705	0.8183	0.8096	0.8096
0.70	0.9376	0.8906	0.8882	0.8882
0.75	0.9757	0.9417	0.9431	0.9431
0.80	0.9930	0.9736	0.9762	0.9762
0.85	0.9987	0.9906	0.9925	0.9925
0.90	0.9999	0.9978	0.9986	0.9986
0.95	1.0000	0.9998	0.9999	0.9999

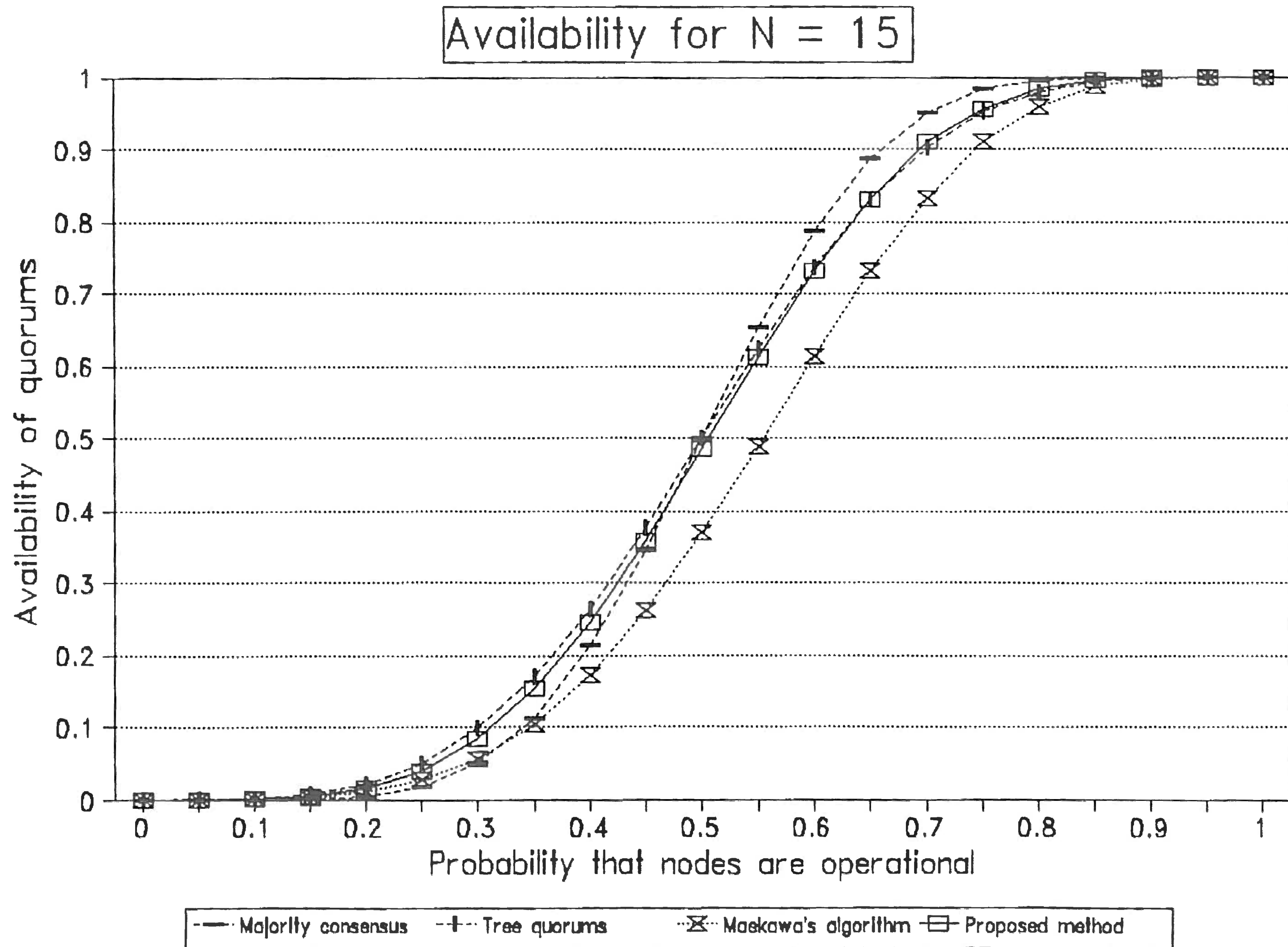

Figure 16. Graph comparing the availabilities of the four methods with N = 15

TABLE VII

COMPARISON OF THE AVAILABILITIES OF THE
FOUR METHODS WITH N = 15

PROBABILITY	MAJORITY CONSENSUS	TREE QUORUMS	MAEKAWA'S ALGORITHM	PROPOSED METHOD
0.05	0.0000	0.0001	0.0000	0.0000
0.10	0.0000	0.0013	0.0006	0.0006
0.15	0.0006	0.0066	0.0032	0.0041
0.20	0.0042	0.0206	0.0107	0.0151
0.25	0.0173	0.0493	0.0269	0.0398
0.30	0.0500	0.0978	0.0564	0.0843
0.35	0.1132	0.1694	0.1036	0.1527
0.40	0.2131	0.2639	0.1716	0.2456
0.45	0.3465	0.3768	0.2609	0.3589
0.50	0.5000	0.5000	0.3689	0.4847
0.55	0.6535	0.6232	0.4896	0.6124
0.60	0.7869	0.7361	0.6138	0.7306
0.65	0.8868	0.8306	0.7311	0.8300
0.70	0.9500	0.9023	0.8317	0.9095
0.75	0.9827	0.9508	0.9085	0.9548
0.80	0.9958	0.9794	0.9592	0.9828
0.85	0.9994	0.9935	0.9866	0.9954
0.90	1.0000	0.9987	0.9974	0.9993
0.95	1.0000	0.9999	0.9999	1.0000

If quorums of the proposed method are constructed from a generator (N=7,11,13), its availability is similar to the tree quorums and Maekawa's algorithm when the probability is greater than 0.75.

In Maekawa's algorithm, if there exists p^k of order n, then the availabilities of Maekawa's algorithm and the proposed method are the same (N=7,13).

Complexity

Time complexity of the proposed method includes two parts. The first part is the complexity for finding generators. The second part is the complexity for forming quorums. The complexity for finding generators is $O(N^{\sqrt{N}})$, where N is the number of nodes in the system. The $N^{\sqrt{N}}$ is calculated from applying the difference set algorithm $\binom{N}{E}$ times, $E = O(\sqrt{N})$. The complexity for forming quorums is $O(n^2)$, where n is the number of generators. n^2 is calculated from doing a nested loop in the generator's tree. Thus, the complexity of the proposed method is $O(N^{\sqrt{N}} + n^2)$.

CHAPTER V

CONCLUSIONS

Conclusions

A new method of constructing quorums in distributed systems has been proposed. The proposed method uses the difference set algorithm to construct generators. Then, it uses the constructed generators to form quorums. The resulting quorums have the property that each quorum has at least one node in common with all other quorums. This property can be used to guarantee mutual exclusion in a distributed system.

The proposed method requires fewer nodes to form a quorum than majority consensus and tree quorums when a fraction of quorums that include root, f, is less than 0.75. When the proposed method constructs quorums from more than one generator, the proposed method is more flexible and reliable than Maekawa's algorithm in that it provides more choice of quorums to perform mutual exclusion and gives higher availability of forming quorums.

Regardless of constructing quorums from one or more generators, the proposed method gives a balanced load to all nodes in the system, in most cases.

Future Work

The proposed method applies the difference set algorithm to all of the possible generators of a given N in order to find generators. The time complexity for finding generators is $O(N^{\sqrt{N}})$. It would be useful to have a more efficient algorithm to compute the possible generators for a given N.

REFERENCES

[Agrawal 89]
Agrawal, Divyakant, and El Abbadi, Amr, "An Efficient Solution to the Distributed Mutual Exclusion Problem," Proceedings of the 8th ACM Symposium on Principles of Distributed Computing, 1989, pp. 193-200.

[Agrawal 90]
Agrawal, Divyakant, and El Abbadi, Amr, "Exploiting Logical Structures in Replicated Databases," Information Processing Letters, Vol. 33, January 1990, pp.255-260.

[Agrawal 91]
Agrawal, Divyakant, and El Abbadi, Amr, "An Efficient and Fault-Tolerant Solution for the Distributed Mutual Exclusion," ACM Transactions on Computer Systems, Vol. 9, No. 1, February 1991, pp. 1-20.

[Barbara 86]
Barbara, Daniel, and Garcia-Molina, Hector, "The Vulnerability of Vote Assignments," ACM Transactions on Computer Systems, Vol. 4, No. 3, August 1986, pp. 187-213.

[Bhargava 87]
Bhargava, Bharat K., Concurrency Control and Reliability in Distributed Systems, Van Nostrand Reinhold Company Inc., 1987.

[Blattner 68]
Blattner, John W., Projective Plane Geometry, Holden-Day, Inc., 1968, pp. 12-46.

[Cheung 90]
Cheung, Shun Yan, Ammar, Mostafa H., and Ahamad Mustaque, "The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data," IEEE 6th International Conference on Data Engineering, 1990, pp.438-445.

[Garcia 85]
Garcia-Molina, Hector, and Barbara, Daniel, "How to Assign Votes in a Distributed System," Journal of the ACM, Vol. 32, No. 4, October 1985, pp.841-860.

[Gifford 79]
Gifford, David K., "Weighted Voting for Replicated Data," Proceedings of the 7th ACM Symposium on Operating Systems Principles, 1979, pp.150-162.

[Kumar 91]
Kumar, Akhil, "Hierarchical Quorum Consensus : A New Algorithm for Managing Replicated Data," IEEE Transactions on Computers, Vol. 40, No. 9, September 1991, pp. 996-1004.

[Lamport 78a]
Lamport, Leslie, "The Implementation of Reliable Distributed Multiprocess systems," Computer Networks, Vol. 2, 1978, pp. 95-114.

[Lamport 78b]
Lamport, Leslie, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, Vol. 21, No. 7, July 1978, pp. 558-565.

[Maekawa 85]
Maekawa, Mamoru, "A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems," ACM Transactions on Computer Systems, Vol. 3, No. 2, May 1985, pp. 145-159.

[Mullender 89]
Mullender, Sape J., Distributed Systems, Addison-Wesley Publishing Company, 1989.

[Neilsen 92a]
Neilsen, Mitchell L., and Mizuno, Masaaki, "Coterie Join Algorithm," IEEE Transaction on Parallel and Distributed Systems, Sept 1992, pp. 582-590.

[Neilsen 92b]
Neilsen, Mitchell L., Mizuno, Masaaki, and Raynal, Michel, "A General Method to Define Quorums," IEEE 12th International Conference on Distributed Computing Systems, Yokohama, Japan 1992, pp. 657-664.

[Neilsen 92c]
Neilsen, Mitchell L., and Mizuno, Masaaki, "Availability analysis of composite coteries," IEEE 11th International Phoenix Converence on Computers and Communications, 1992, pp. 759-765.

[Ricart 81]
Ricart, Glenn, and Agrawala, A. K., "An Optimal Algorithm for Mutual Exclusion in Computer Networks," Communication of the ACM, Vol. 24, No. 1, January 1981, pp. 9-17.

[Thomas 79]
Thomas, Robert H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979, pp. 180-209.

APPENDIXES

APPENDIX A

INPUT QUORUMS FOR MAEKAWA'S ALGORITHM AND THE PROPOSED METHOD USED IN THE ANALYSIS SECTION

INPUT DATA FOR MAEKAWA'S ALGORITHM

```
==========
N = 5
==========
QUORUMS:
1 2 3
1 4 5
2 4 0
2 5 0
3 5 4
END.
==========


==========
N = 7
==========
QUORUMS:
1 2 3
1 4 5
1 6 7
2 4 6
2 5 7
3 4 7
3 5 6
END.
==========


==========
N = 9
==========
QUORUMS:
1 2 3 4
1 5 6 7
1 8 9 6
2 5 8 7
2 6 9 8
2 7 6 9
3 6 8 9
3 7 9 0
4 6 0 7
END.
==========


==========
N = 11
==========
QUORUMS:
1 2 3 4
1 5 6 7
1 8 9 10
1 11 10 0
```

```
2 5 8 11
2 6 9 10
2 7 10 11
3 5 10 0
3 6 8 11
3 7 9 11
4 6 10 11
END.
==========


==========
N = 13
==========
QUORUMS:
1 2 3 4
1 5 6 7
1 8 9 10
1 11 12 13
2 5 8 11
2 6 9 12
2 7 10 13
3 5 10 12
3 6 8 13
3 7 9 11
4 5 9 13
4 6 10 11
4 7 8 12
END.
==========


==========
N = 15
==========
QUORUMS:
1 2 3 4 5
1 6 7 8 9
1 10 11 12 13
1 14 15 10 11
2 6 10 14 12
2 7 11 15 13
2 8 12 10 14
2 9 13 11 15
3 6 11 0 14
3 7 10 0 15
3 8 13 15 12
3 9 12 14 13
4 6 12 15 0
4 7 13 14 0
5 7 12 11 0
```

INPUT DATA FOR THE PROPOSED METHOD

```
====================
N = 5     E = 3
====================
GENERATOR(S) :
1 2 3
1 2 4
END.
--------------------
QUORUMS:
  1  2  3     2  3  4     3  4  5     1  4  5
  1  2  5
  1  2  4     2  3  5     1  3  4     2  4  5
  1  3  5
END.
====================


====================
N = 7     E = 3
====================
GENERATOR(S) :
1 2 4
END.
--------------------
QUORUMS:
  1  2  4     2  3  5     3  4  6     4  5  7
  1  5  6     2  6  7     1  3  7
END.
====================


====================
N = 9     E = 4
====================
GENERATOR(S) :
1 2 3 5
1 2 4 5
1 2 4 6
END.
--------------------
QUORUMS:
  1  2  3  5     2  3  4  6     3  4  5  7     4  5  6  8
  5  6  7  9     1  6  7  8     2  7  8  9     1  3  8  9
  1  2  4  9
  1  2  4  5     2  3  5  6     3  4  6  7     4  5  7  8
  5  6  8  9     1  6  7  9     1  2  7  8     2  3  8  9
  1  3  4  9
  1  2  4  6     2  3  5  7     3  4  6  8     4  5  7  9
  1  5  6  8     2  6  7  9     1  3  7  8     2  4  8  9
  1  3  5  9
END.
====================
```

```
====================
N = 11      E = 4
====================
GENERATOR(S) :
1 2 3 6
END.
--------------------
QUORUMS:
  1  2  3  6      2  3  4  7      3  4  5  8      4  5  6  9
  5  6  7 10      6  7  8 11      1  7  8  9      2  8  9 10
  3  9 10 11      1  4 10 11      1  2  5 11
END.
====================
```

```
====================
N = 13      E = 4
====================
GENERATOR(S) :
1 2 5 7
END.
--------------------
QUORUMS:
  1  2  5  7      2  3  6  8      3  4  7  9      4  5  8 10
  5  6  9 11      6  7 10 12      7  8 11 13      1  8  9 12
  2  9 10 13      1  3 10 11      2  4 11 12      3  5 12 13
  1  4  6 13
END.
====================
```

```
====================
N = 15      E = 5
====================
GENERATOR(S) :
1 2 8 10 13
1 2 4 7 8
1 2 4 7 11
1 2 4 8 13
1 2 4 11 13
END.
--------------------
QUORUMS:
  1  2  8 10 13      2  3  9 11 14      3  4 10 12 15
  1  4  5 11 13      2  5  6 12 14      3  6  7 13 15
  1  4  7  8 14      2  5  8  9 15      1  3  6  9 10
  2  4  7 10 11      3  5  8 11 12      4  6  9 12 13
  5  7 10 13 14      6  8 11 14 15      1  7  9 12 15
  1  2  4  7  8      2  3  5  8  9      3  4  6  9 10
  4  5  7 10 11      5  6  8 11 12      6  7  9 12 13
  7  8 10 13 14      8  9 11 14 15      1  9 10 12 15
  1  2 10 11 13      2  3 11 12 14      3  4 12 13 15
  1  4  5 13 14      2  5  6 14 15      1  3  6  7 15
  1  2  4  7 11      2  3  5  8 12      3  4  6  9 13
  4  5  7 10 14      5  6  8 11 15      1  6  7  9 12
```

```
   2  7  8 10 13      3  8  9 11 14      4  9 10 12 15
   1  5 10 11 13      2  6 11 12 14      3  7 12 13 15
   1  4  8 13 14      2  5  9 14 15      1  3  6 10 15
   1  2  4  8 13      2  3  5  9 14      3  4  6 10 15
   1  4  5  7 11      2  5  6  8 12      3  6  7  9 13
   4  7  8 10 14      5  8  9 11 15      1  6  9 10 12
   2  7 10 11 13      3  8 11 12 14      4  9 12 13 15
   1  5 10 13 14      2  6 11 14 15      1  3  7 12 15
   1  2  4 11 13      2  3  5 12 14      3  4  6 13 15
   1  4  5  7 14      2  5  6  8 15      1  3  6  7  9
   2  4  7  8 10      3  5  8  9 11      4  6  9 10 12
   5  7 10 11 13      6  8 11 12 14      7  9 12 13 15
   1  8 10 13 14      2  9 11 14 15      1  3 10 12 15
END.
====================
```

APPENDIX B

ANALYSIS AND SIMULATION PROGRAMS

ANALYSIS PROGRAM FOR THE MAJORITY CONSENSUS ALGORITHM

```
/************************************************************
     The file makefile is used to compile all the programs
that are used to calculate availability in the majority
consensus algorithm.  The compiled file is "run".  Note
that -lm is for the include math.h. To run the program
enter "run N Pr", where N (nodes in a system) is an integer
number and Pr (probability that nodes are operational) is a
float number.
************************************************************/

run: main.o avail.o get_N_Pr.o N_pick_K.o power_f.o
     cc main.o avail.o get_N_Pr.o N_pick_K.o power_f.o \
          -lm -o run
main.o:        general.h main.c
     cc  -g -c main.c
avail.o:     general.h avail.c
     cc  -g -c avail.c
get_N_Pr.o:  general.h get_N_Pr.c
     cc  -g -c get_N_Pr.c
N_pick_K.o:  general.h N_pick_K.c
     cc  -g -c N_pick_K.c
power_f.o:   general.h power_f.c
     cc  -g -c power_f.c


/************************************************************
     The file general.h is used to define variables that
will be used throughout the availability analysis of
majority consensus simulation program.  The general.h is an
include file that is included in every other program of the
analysis majority consensus.
************************************************************/

#include  <stdio.h>
#include  <stdlib.h>
#include  <string.h>
#include  <math.h>

#define   MAX_N         31
#define   MIN_N         3
#define   PRINT_AVAIL   7


/************************************************************
     The procedure main() is the main driver of the
availability analysis of the majority consensus.  It calls
procedures Get_N_Pr() and Availability().
************************************************************/

#include  "general.h"

main (argc, argv)
```

```
int   argc;

char *argv[];
{
int    N, K;
double Pr;

/***  Get input N and Pr. (probability)  ***/
      Get_N_Pr(argc, argv, &N, &Pr);

/***  Take lower bound of K  ***/
      K = (N/2);
      Availability(N, K, Pr);
}


/************************************************************
     The procedure Get_N_Pr() is called by the procedure
main().  It is used to get the value of N and value of
probability that will be used in the procedure
Availability().
************************************************************/

#include  "general.h"

Get_N_Pr(argc, argv, N, Pr)
int    argc;
char   *(*argv);
int    *N;
double *Pr;
{
int    i;

      if (argc < 3)
      {
         printf("\n*****Missing value of N or Pr*****\n");
         printf("Enter  run N Pr  ");
         printf("(N is an integer, Pr is a float
(probability))\n\n");
         exit(0);
      }
      i = 0;
      while (argv[1][i])
      {
         if ((argv[1][i] < '0') || (argv[1][i] > '9'))
         {
          printf("\n*****%s is not an
integer*****\n",argv[1]);
          printf("Enter  run N Pr  ");
          printf("(N is an integer, Pr is a float
(probability))\n\n");
            exit(0);
         }
         i++;
      }
```

```
     *N = atoi(argv[1]);

      if (((*N) > MAX_N) || ((*N) < MIN_N))
      {
         printf("\n%d is out of range (%d..%d)\n ",(*N),
                MIN_N, MAX_N);
         printf("Enter  run N Pr  ");
       printf("(N is an integer, Pr is a float
(probability))\n\n");
         exit(0);
      }

/*** Checking Pr's value  ***/

     *Pr = atof(argv[2]);

      if ((*Pr > 1.0) || (*Pr == 1.0) || (*Pr == 0.0))
      {
         printf("\n*****Pr should not be > or = 1.0 ");
         printf("and should not be = 0.0 *****\n");
         printf("Enter  run N Pr  ");
         printf("(N is an integer, Pr is a float
(probability))\n\n");
         exit(0);
      }
}


/************************************************************
     The procedure Availability() is called by the
procedure main().  It is used to calculate the availability
of the majority consensus algorithm.
***********************************************************/

#include  "general.h"

Availability(N, K, Pr)
int     N, K;
double  Pr;
{
double   Avail, Tot_avail;
double   _p_Kplus1, _1minusp_K;
double   Prob;
int      I;

     printf("N = %d\n",N);
     printf("K = %d\n",K);
     printf("Pr = %1.2f\n",Pr);

     I = 0;
     Prob = 0;
     Avail = 0.0;
     Tot_avail = 0.0;
     for (I = 1; (N-(K+I)) >= 0 ; I++)
     {
```

```
          N_pick_K(N, (K+I), &Prob);
          _p_Kplus1 = 0.0;
          Power_float(Pr,(K+I), &_p_Kplus1);
          _1minusp_K = 0.0;
          Power_float((1.0-Pr),(N-(K+I)), &_1minusp_K);
          Avail = Prob * _p_Kplus1 * _1minusp_K;
          Tot_avail = Tot_avail + Avail;
          printf("(%d pick %d) * ",N, (K+I));
          printf("(%1.2f power of %d) * ", Pr, (K+I));
          printf("(%1.2f power of %d) = %1.10f \n", (1-Pr),
                (N-(K+I)), Avail);
     }
     printf("Availability = %1.10f\n",Tot_avail);
}


/************************************************************
     The procedure N_pick_K() is called by the procedure
Availability().  It is used to calculate all possible
choices of choosing K out of N.
************************************************************/

#include  "general.h"

N_pick_K(N, K, Prob)
int  N, K;
double *Prob;
{
int  i, D;

    *Prob = 1;
     if ((N-K) > K)
     {
        D = K;
        for (i = N; i > (N-K); i--)
        {
            (*Prob) = (*Prob) * i;
            if (D > 1)
            {
               (*Prob) = (*Prob) / D;
                 D--;
            }
        }
        for (i = D; i > 1; i--)
            (*Prob) = (*Prob) / i;
     }
     else
     {
        D = (N-K);
        for (i = N; i > K; i--)
        {
            (*Prob) = (*Prob) * i;
            if (D > 1)
            {
               (*Prob) = (*Prob) / D;
```

```
                D--;
            }
        }
        for (i = D; i > 1; i--)
            (*Prob) = (*Prob) / i;
    }

}


/************************************************************
     The procedure Power_float() is called by the procedure
Availability().  It is used to calculate the power of a
given base variable.  The power value is a float number.
************************************************************/

#include  "general.h"

Power_float(base, exp, P)
double base;
int    exp;
double *P;
{
int    i;

    if (exp <= 0)
       *P = 1.0;
    else
    {
       *P = base;
       for (i=2; i <= exp; i++)
       {
           (*P) = (*P) * base;
       }
    }
}
```

OUTPUTS OF THE PROGRAM WITH N = 5 , Pr = 0.9 AND 0.95

```
N = 5
K = 2
Pr = 0.90
(5 pick 3) * (0.90 power of 3) * (0.10 power of 2) =
  0.0729000000
(5 pick 4) * (0.90 power of 4) * (0.10 power of 1) =
  0.3280500000
(5 pick 5) * (0.90 power of 5) * (0.10 power of 0) =
  0.5904900000
Availability = 0.9914400000


N = 5
K = 2
Pr = 0.95
(5 pick 3) * (0.95 power of 3) * (0.05 power of 2) =
  0.0214343750
(5 pick 4) * (0.95 power of 4) * (0.05 power of 1) =
  0.2036265625
(5 pick 5) * (0.95 power of 5) * (0.05 power of 0) =
  0.7737809375
Availability = 0.9988418750
```

ANALYSIS PROGRAM FOR THE TREE QUORUM ALGORITHM

```
/************************************************************
     The file makefile is used to compile all the programs
used in calculating availability of the tree quorums
algorithm.  The compiled file is "run".  Note that -lm is
used for the include math.h.  To run the program enter "run
N Pr", where N (nodes in a system) is an integer number and
Pr (probability that nodes are operational) is a float
number.
************************************************************/

run: main.o avail.o get_N_Pr.o p_all.o
     cc main.o avail.o get_N_Pr.o p_all.o -lm -o run

main.o:        general.h main.c
     cc  -g -c main.c
avail.o:     general.h avail.c
     cc  -g -c avail.c
get_N_Pr.o:  general.h get_N_Pr.c
     cc  -g -c get_N_Pr.c
p_all.o:   general.h p_all.c
     cc  -g -c p_all.c


/************************************************************
     The file general.h is used to define variables that
will be used throughout the availability analysis of the
tree quorums algorithm.  The general.h is an include file
that is included in every other program of the analysis tree
quorums.
************************************************************/

#include  <stdio.h>
#include  <stdlib.h>
#include  <string.h>
#include  <math.h>

#define   MAX_N         31
#define   MIN_N         3
#define   THREE         3
#define   BASE          2
#define   PRINT_AVAIL   7


/************************************************************
     The procedure main() is the main driver that is used to
calculate availability of the tree quorums algorithm.  It
calls procedures Get_N_Pr() and Availability().
************************************************************/

#include  "general.h"

main (argc, argv)
```

```
int   argc;
char *argv[];
{
int    N;
double Pr;

/***  Get input N, and Pr (probability)  ***/
      Get_N_Pr(argc, argv, &N, &Pr);

      printf("N  = %d\n",N);
      printf("Pr = %f\n",Pr);

      Availability(N, Pr);

}


/*************************************************************
     The procedure Get_N_Pr() id called by the procedure
main().  It is used to get the value of N and value of
probability that will be used in the procedure
Availability().
*************************************************************/

#include  "general.h"

Get_N_Pr(argc, argv, N, Pr)
int    argc;
char   *(*argv);
int    *N;
double *Pr;
{
int    i;

    if (argc < 3)
    {
       printf("\n*****Missing value of N or Pr*****\n");
       printf("Enter  run N Pr  ");
       printf("(N is an integer, Pr is a float
(probability))\n\n");
       exit(0);
    }

    i = 0;
    while (argv[1][i])
    {
       if ((argv[1][i] < '0') || (argv[1][i] > '9'))
       {
          printf("\n*****%s is not an
integer*****\n",argv[1]);
          printf("Enter  run N Pr  ");
          printf("(N is an integer, Pr is a float
(probability))\n\n");
          exit(0);
       }
```

```
        i++;
     }

   *N = atoi(argv[1]);

    if (((*N) > MAX_N) || ((*N) < MIN_N) || (((*N)%2) == 0))
    {
       printf("\n%d is out of range (%d..%d) ",(*N),
               MIN_N, MAX_N);
       printf(" or %d is an even number\n",*N);
       printf("Enter  run N Pr  ");
       printf("(N is an integer, Pr is a float
(probability))\n\n");
       exit(0);
    }

/*** Checking Pr's value  ***/

   *Pr = atof(argv[2]);

    if ((*Pr > 1.0) || (*Pr == 1.0) || (*Pr == 0.0))
    {
       printf("\n*****Pr should not be > or = 1.0 ");
       printf("and should not be = 0.0 *****\n");
       printf("Enter  run N Pr  ");
       printf("(N is an integer, Pr is a float
(probability))\n\n");
       exit(0);
    }
}


/************************************************************
     The procedure Availability() is called by the
procedure main().  It is used to calculate the availability
of the tree quorums algorithm.  It is a recursive procedure
that calculates availability of each level of a tree.
************************************************************/

#include  "general.h"

double Recu_avail();

Availability(N, Pr)
int     N;
double  Pr;
{
double   Pr_left, Pr_right;
int      Root;

   Root = 1;
   Pr_left = Pr;
   Pr_right = Pr;
   Recu_avail(N, Root, Pr, Pr_left, Pr_right);
}
```

```
/************************************************************
     The procedure Recu_avail is the recursive part of the
procedure Availability().
************************************************************/

double Recu_avail(N, Root, Pr, Pr_left, Pr_right)
int      N, Root;
double   Pr, Pr_left, Pr_right;
{
double   Avail, Tot_avail;
double   Pr_l, Pr_r;
unsigned long int  P;
int  i, I;

   Pr_l = Pr_left;
   Pr_r = Pr_right;
   if ((Root*4) < N)
   {
      Pr_l = Recu_avail(N, Root*2, Pr, Pr_left, Pr_right);
      if ((Root*4+2) < N)
       Pr_r = Recu_avail(N, Root*2+1, Pr, Pr_left,
  Pr_right);
   }
   Tot_avail = 0.0;
   I = 0;
   P = 1;                      /*** {a,b}, {a.c} ***/
   for (i=1; i<THREE; i++)
   {
       P = P + 2;
       Avail = 1.0;
       printf("Pr({%d %d}) = ",Root, Root*2+I);
       Print_avail_all(P, Pr, Pr_l, Pr_r, &Avail);
       Tot_avail = Tot_avail + Avail;
       I++;
    }

    P = 6;                     /*** {b,c} ***/
    Avail = 1.0;
    printf("Pr({%d %d}) = ",Root*2, Root*2+1);
    Print_avail_all(P, Pr, Pr_l, Pr_r, &Avail);
    Tot_avail = Tot_avail + Avail;

    P = 7;                     /*** {a,b,c} ***/
    Avail = 1.0;
    printf("Pr({%d %d %d}) = ",Root, Root*2, Root*2+1);
    Print_avail_all(P, Pr, Pr_l, Pr_r, &Avail);
    Tot_avail = Tot_avail + Avail;

    printf("Availability = %1.20f\n",Tot_avail);

    return(Tot_avail);
}
```

```
/**************************************************************
     The procedure Print_avail_all() is called by the
procedure Availability().  It is used to print availability
of each subtree.
**************************************************************/

#include  "general.h"

Print_avail_all(P, Pr, Pr_l, Pr_r, Avail)
unsigned long int P;
double  Pr, Pr_l, Pr_r;
double *Avail;
{
double  Pr_temp;
int     i;

     Pr_temp = Pr;
     for (i=1; i <= THREE; i++)
     {
         if ((P%2) == 1)
         {
            (*Avail) = (*Avail) * Pr_temp;
            printf("(%f)",Pr_temp);
         }
         else
         {
            (*Avail) = (*Avail) * (1.0 - Pr_temp);
            printf("(%f)",(1.0-Pr_temp));
         }
         Pr = Pr_l;
         Pr_l = Pr_r;
         Pr_temp = Pr;

         P= P >> 1;
      }
      printf(" = %f \n",(*Avail));
}
```

OUTPUTS OF THE PROGRAM WITH N = 5, Pr = 0.9 AND 0.95

```
N  = 5
Pr = 0.900000
Pr({2 4}) = (0.900000)(0.900000)(0.100000) = 0.081000
Pr({2 5}) = (0.900000)(0.100000)(0.900000) = 0.081000
Pr({4 5}) = (0.100000)(0.900000)(0.900000) = 0.081000
Pr({2 4 5}) = (0.900000)(0.900000)(0.900000) = 0.729000
Availability = 0.97200000000000009000
Pr({1 2}) = (0.900000)(0.972000)(0.100000) = 0.087480
Pr({1 3}) = (0.900000)(0.028000)(0.900000) = 0.022680
Pr({2 3}) = (0.100000)(0.972000)(0.900000) = 0.087480
Pr({1 2 3}) = (0.900000)(0.972000)(0.900000) = 0.787320
Availability = 0.98496000000000006000


N  = 5
Pr = 0.950000
Pr({2 4}) = (0.950000)(0.950000)(0.050000) = 0.045125
Pr({2 5}) = (0.950000)(0.050000)(0.950000) = 0.045125
Pr({4 5}) = (0.050000)(0.950000)(0.950000) = 0.045125
Pr({2 4 5}) = (0.950000)(0.950000)(0.950000) = 0.857375
Availability = 0.99275000000000002000
Pr({1 2}) = (0.950000)(0.992750)(0.050000) = 0.047156
Pr({1 3}) = (0.950000)(0.007250)(0.950000) = 0.006543
Pr({2 3}) = (0.050000)(0.992750)(0.950000) = 0.047156
Pr({1 2 3}) = (0.950000)(0.992750)(0.950000) = 0.895957
Availability = 0.99681125000000004000
```

ANALYSIS PROGRAM FOR MAEKAWA'S ALGORITHM AND THE PROPOSED METHOD

```
/************************************************************
     The file makefile is used to compile all the programs
that are used in availability analysis of the Maekawa's
algorithm and the proposed method.  The complied file is
"run".  Note that -lm is used for the include math.h.  To
run the program enter "run N Pr", where N (nodes in a
system) is an integer number and Pr (probability that nodes
are operational) is a float number.
************************************************************/

run: main.o avail.o change_N.o convert_Q.o find_E.o \
          get_N_Pr.o p_all.o p_avail.o power.o
     cc main.o avail.o change_N.o convert_Q.o find_E.o \
          get_N_Pr.o p_all.o p_avail.o power.o -lm -o run

main.o:        general.h main.c
     cc  -g -c main.c
avail.o:     general.h avail.c
     cc  -g -c avail.c
change_N.o:  general.h change_N.c
     cc  -g -c change_N.c
convert_Q.o: general.h convert_Q.c
     cc  -g -c convert_Q.c
find_E.o:    general.h find_E.c
     cc  -g -c find_E.c
get_N_Pr.o:  general.h get_N_Pr.c
     cc  -g -c get_N_Pr.c
p_all.o:   general.h p_all.c
     cc  -g -c p_all.c
p_avail.o:   general.h p_avail.c
     cc  -g -c p_avail.c
power.o:     general.h power.c
     cc  -g -c power.c


/************************************************************
     The file general.h is used to define variables that
will be used throughout the availability analysis of
Maekawa's algorithm and the proposed method.  It is an
include file that includes in every other program of the
analysis Maekawa's algorithm and the proposed method.
************************************************************/

#include  <stdio.h>
#include  <stdlib.h>
#include  <string.h>
#include  <math.h>

#define   MAX_N        31
#define   MIN_N        3
#define   INPUT        4
```

```
#define   OUTPUT       10
#define   BASE         2
#define   ONE          1.0
#define   TWO          2.0
#define   THREE        3.0
#define   FOUR         4.0
#define   PRINT_AVAIL  5


/************************************************************
     The procedure main() is the main driver that is used to
calculate availability of Maekawa's algorithm and the
proposed method.
************************************************************/

#include  "general.h"

FILE *I, *O;

main (argc, argv)
int   argc;
char *argv[];
{
int    N, E;
double Pr;
char   Input[INPUT];
char   Output[OUTPUT];

/***  get input N and probability Pr  ***/
      Get_N_Pr(argc, argv, &N, &Pr);

/***  find number of nodes in a quorum from the
      input N  ***/
      Find_E(N, &E);

/***  change integer N to a string in order to use the
      string to open an input file  ***/
      strcat(Input,"");
      Change_N_to_string(N, Input);
      if((I = fopen(Input,"r")) == NULL)
      { printf("Cannot open input file\n"); exit(0); }
      if((O = fopen("CONV","w")) == NULL)
      { printf("Cannot open input file\n"); exit(0); }

/***  represent quorums with unsigned long integers  ***/
      Convert_Q(I, O, E);

      fclose(O);
      fclose(I);

      system("sort -n CONV > Conv");
      system("rm CONV");

      if((I = fopen("Conv","r")) == NULL)
      { printf("Cannot open input file\n"); exit(0); }
```

```
        strcat(Output,"o");
        strcat(Output,Input);
        if((O = fopen(Output,"w")) == NULL)
        { printf("Cannot open output file\n"); exit(0); }

        fprintf(O,"N  = %d\n",N);
        fprintf(O,"E  = %d\n",E);
        fprintf(O,"Pr = %1.2f\n",Pr);

/***    calculate availability  ***/
        Availability(N, E, Pr, I, O);

        system("rm Conv");
        fclose(I);
        fclose(O);
}


/************************************************************
     The procedure Get_N_Pr() is called by the procedure
main().  It is used to get the value of N and the
probability Pr that are used in the procedures main() and
Availability().
************************************************************/

#include  "general.h"

Get_N_Pr(argc, argv, N, Pr)
int     argc;
char    *(*argv);
int     *N;
double *Pr;
{
int     i;

        if (argc < 3)
        {
           printf("\n*****Missing value of N or Pr*****\n");
           printf("Enter  run N Pr  ");
           printf("(N is an integer, Pr is a float
(probability))\n\n");
           exit(0);
        }

        i = 0;
        while (argv[1][i])
        {
           if ((argv[1][i] < '0') || (argv[1][i] > '9'))
           {
            printf("\n*****%s is not an
integer*****\n",argv[1]);
            printf("Enter  run N Pr  ");
            printf("(N is an integer, Pr is a float
(probability))\n\n");
            exit(0);
```

```
         }
         i++;
      }

     *N = atoi(argv[1]);

      if (((*N) > MAX_N) || ((*N) < MIN_N))
      {
         printf("\n%d is out of range (%d..%d).\n",(*N),
                MIN_N, MAX_N);
         printf("Enter  run N Pr  ");
         printf("(N is an integer, Pr is a float
(probability))\n\n");
         exit(0);
      }

/*** Checking Pr's value  ***/

     *Pr = atof(argv[2]);

      if ((*Pr > 1.0) || (*Pr == 1.0) || (*Pr == 0.0))
      {
         printf("\n*****Pr should not be > or = 1.0 ");
         printf("and should not be = 0.0 *****\n");
         printf("Enter  run N Pr  ");
         printf("(N is an integer, Pr is a float
(probability))\n\n");
         exit(0);
      }
}


/************************************************************
     The procedure Find_E() is called by the procedure
main().  It is used to find E, number of nodes needed to
form a quorum, from a given N.
************************************************************/

#include  "general.h"

Find_E(N, E)
int  N;
int *E;
{
double  N_double, E_double;
float   E_float;

/*** convert integer to double ***/
     N_double = N;

/*** equation and conversion***/
     E_double = ceil((ONE + sqrt((FOUR * N_double)
                  - THREE)) / TWO);
/*** convert double to float ***/
     E_float  = (float)E_double;
```

```
/*** convert float to integer ***/
    *E = (int)E_float;
}


/************************************************************
     The procedure Change_N_to_string() is called by the
procedure main().  It is used to change integer value to
string.
************************************************************/

#include  "general.h"

Change_N_to_string(N, Input)
int   N;
char  *Input;
{
int   Inp;
char  temp_input[INPUT];
int   i, j, count;

      count = 0;
      strcpy(temp_input,"");
      while (N > 0)
      {
         Inp = N%10;
         N = N/10;
         switch (Inp) {
            case 0: strcat(temp_input,"0"); break;
            case 1: strcat(temp_input,"1"); break;
            case 2: strcat(temp_input,"2"); break;
            case 3: strcat(temp_input,"3"); break;
            case 4: strcat(temp_input,"4"); break;
            case 5: strcat(temp_input,"5"); break;
            case 6: strcat(temp_input,"6"); break;
            case 7: strcat(temp_input,"7"); break;
            case 8: strcat(temp_input,"8"); break;
            case 9: strcat(temp_input,"9"); break;
         }
         count++;
      }
      j = 0;
      for (i = (count-1) ; i >= 0 ;i--)
      {
          Input[j] = temp_input[i];
          j++;
      }
      Input[j] = '\0';
}
```

```
/*************************************************************
     The procedure Convert_Q() is called by the procedure
main().  It is used to convert each node in a quorum to an
unsigned ling integer number.
*************************************************************/

#include  "general.h"

Convert_Q(I, O, E)
FILE  *I, *O;
int    E;
{
unsigned long int  Q;
unsigned long int  Q_in;
int       i;

      fscanf(I,"%u",&Q_in);
      while (!feof(I))
      {
          Q = 0;
          Q = Q + Power(BASE,(Q_in-1));
          for (i = 1; i < E; i++)
          {
               fscanf(I,"%u",&Q_in);
               Q = Q + Power(BASE,(Q_in-1));
          }
          fprintf(O,"%u\n",Q);
          fscanf(I,"%u",&Q_in);
      }
}


/*************************************************************
     The procedure Availability() is called by the procedure
main().  It is used to calculate availability of Maekawa's
algorithm and the proposed method.
*************************************************************/

#include  "general.h"

Availability(N, E, Pr, I, O)
int     N, E;
double  Pr;
FILE   *I, *O;
{
unsigned long int  Q;
unsigned long int  P, Ac, Ac_start;
double    Avail, Tot_avail;
int       i, j;

   P = Power(BASE,N) - 1;
   Tot_avail = 0.0;
   Ac_start = 1;
   fscanf(I,"%u",&Q);
```

```
   if (!feof(I))
   {
      Ac_start = Q;
      for (Ac = Ac_start; Ac <= P; Ac++)
      {
          while (!feof(I))
          {
              if ((Q & (~Ac)) == 0)   /*** a super set ***/
              {
                 Avail = 1.0;

                 if (N <= PRINT_AVAIL)
                     Print_avail_all(N, Ac, Pr, &Avail, 0);
                 else
                     Print_avail(N, Ac, Pr, &Avail, 0);

                 Tot_avail = Tot_avail + Avail;
                 fseek(I,0,2);
              }
              fscanf(I,"%u",&Q);
          }
          fseek(I,0,0);
      }
      fprintf(O,"Availability = %1.10f\n",Tot_avail);
   }
}


/************************************************************
     The function Power() is used to calculate the power of
a given base variable.
************************************************************/

#include  "general.h"

unsigned long int Power(base, exp)
int   base;
int   exp;
{
unsigned long int  P;
int   i;

      if (exp < 0)
         return(0);

      P = 1;
      for (i=1; i <= exp; i++)
      {
          P = P * base;
      }
      return(P);
}
```

```
/***********************************************************
     The procedure Print_avail_all() is called by the
procedure Availability().  It is used to print full format
of the result availabilities.
***********************************************************/

#include  "general.h"

Print_avail_all(N, Ac, Pr, Avail, O)
int      N;
unsigned long int Ac;
double  Pr;
double *Avail;
FILE   *O;
{
unsigned long int Ac_temp;
int  i, j;

     Ac_temp = Ac;
     fprintf(O,"Pr({ ");
     for (i=1; i <= N; i++)
     {
         if ((Ac_temp%2) == 1)
            fprintf(O,"%d ",i);
         Ac_temp = Ac_temp >> 1;
     }
     fprintf(O,"}) = ");
     Ac_temp = Ac;
     for (i=1; i <= N; i++)
     {
         if ((Ac_temp%2) == 1)
         {
            (*Avail) = (*Avail) * Pr;
            fprintf(O,"(%1.2f)",Pr);
         }
         else
         {
            (*Avail) = (*Avail) * (1.0 - Pr);
            fprintf(O,"(%1.2f)",(1.0-Pr));
         }
         Ac_temp = Ac_temp >> 1;
     }
     fprintf(O," = %1.10f \n",(*Avail));
}


/***********************************************************
     The procedure Print_avail() is called by the procedure
Availability().  It is used to print short format of the
result availabilities.
***********************************************************/

#include  "general.h"

Print_avail(N, Ac, Pr, Avail)
```

```
int     N;
unsigned long int Ac;
double  Pr;
double *Avail;
{
int  i, j;

     for (i=1; i <= N; i++)
     {
         if ((Ac%2) == 1)
            (*Avail) = (*Avail) * Pr;
         else
            (*Avail) = (*Avail) * (1.0 - Pr);

         Ac = Ac >> 1;
     }
}
```

OUTPUTS OF THE PROGRAM WITH N = 5, Pr = 0.9 AND 0.95
(THE PROPOSED METHOD)

```
N  = 5   E  = 3   Pr = 0.90
Pr({ 1 2 3 }) = (0.90)(0.90)(0.90)(0.10)(0.10) = 0.0072900
Pr({ 1 2 4 }) = (0.90)(0.90)(0.10)(0.90)(0.10) = 0.0072900
Pr({ 1 3 4 }) = (0.90)(0.10)(0.90)(0.90)(0.10) = 0.0072900
Pr({ 2 3 4 }) = (0.10)(0.90)(0.90)(0.90)(0.10) = 0.0072900
Pr({ 1 2 3 4 }) = (0.90)(0.90)(0.90)(0.90)(0.10) = 0.0656100
Pr({ 1 2 5 }) = (0.90)(0.90)(0.10)(0.10)(0.90) = 0.0072900
Pr({ 1 3 5 }) = (0.90)(0.10)(0.90)(0.10)(0.90) = 0.0072900
Pr({ 2 3 5 }) = (0.10)(0.90)(0.90)(0.10)(0.90) = 0.0072900
Pr({ 1 2 3 5 }) = (0.90)(0.90)(0.90)(0.10)(0.90) = 0.0656100
Pr({ 1 4 5 }) = (0.90)(0.10)(0.10)(0.90)(0.90) = 0.0072900
Pr({ 2 4 5 }) = (0.10)(0.90)(0.10)(0.90)(0.90) = 0.0072900
Pr({ 1 2 4 5 }) = (0.90)(0.90)(0.10)(0.90)(0.90) = 0.0656100
Pr({ 3 4 5 }) = (0.10)(0.10)(0.90)(0.90)(0.90) = 0.0072900
Pr({ 1 3 4 5 }) = (0.90)(0.10)(0.90)(0.90)(0.90) = 0.0656100
Pr({ 2 3 4 5 }) = (0.10)(0.90)(0.90)(0.90)(0.90) = 0.0656100
Pr({ 1 2 3 4 5 }) = (0.90)(0.90)(0.90)(0.90)(0.90) =
0.5904900
Availability = 0.9914400000


N  = 5   E  = 3   Pr = 0.95
Pr({ 1 2 3 }) = (0.95)(0.95)(0.95)(0.05)(0.05) = 0.0021434
Pr({ 1 2 4 }) = (0.95)(0.95)(0.05)(0.95)(0.05) = 0.0021434
Pr({ 1 3 4 }) = (0.95)(0.05)(0.95)(0.95)(0.05) = 0.0021434
Pr({ 2 3 4 }) = (0.05)(0.95)(0.95)(0.95)(0.05) = 0.0021434
Pr({ 1 2 3 4 }) = (0.95)(0.95)(0.95)(0.95)(0.05) = 0.0407253
Pr({ 1 2 5 }) = (0.95)(0.95)(0.05)(0.05)(0.95) = 0.0021434
Pr({ 1 3 5 }) = (0.95)(0.05)(0.95)(0.05)(0.95) = 0.0021434
Pr({ 2 3 5 }) = (0.05)(0.95)(0.95)(0.05)(0.95) = 0.0021434
Pr({ 1 2 3 5 }) = (0.95)(0.95)(0.95)(0.05)(0.95) = 0.0407253
Pr({ 1 4 5 }) = (0.95)(0.05)(0.05)(0.95)(0.95) = 0.0021434
Pr({ 2 4 5 }) = (0.05)(0.95)(0.05)(0.95)(0.95) = 0.0021434
Pr({ 1 2 4 5 }) = (0.95)(0.95)(0.05)(0.95)(0.95) = 0.0407253
Pr({ 3 4 5 }) = (0.05)(0.05)(0.95)(0.95)(0.95) = 0.0021434
Pr({ 1 3 4 5 }) = (0.95)(0.05)(0.95)(0.95)(0.95) = 0.0407253
Pr({ 2 3 4 5 }) = (0.05)(0.95)(0.95)(0.95)(0.95) = 0.0407253
Pr({ 1 2 3 4 5 }) = (0.95)(0.95)(0.95)(0.95)(0.95) =
0.7737809
Availability = 0.9988418750
```

SIMULATION PROGRAM FOR THE PROPOSED METHOD

```
/**********************************************************
     The file makefile is used to compile all the programs
that are necessary for the simulation program.  The complied
file is "run".  Note that -lm is for the include math.h.  To
run the simulation enter "run N", where N (nodes in a
system) is an integer number.
**********************************************************/

run: main.o clr_int.o find_E.o find_G.o  \
          find_Q.o free_mem.o g_tree.o get_N.o \
          initialize.o pds.o power.o search.o shift_L.o
     cc main.o clr_int.o find_E.o find_G.o find_Q.o \
          free_mem.o g_tree.o get_N.o initialize.o \
          pds.o power.o search.o shift_L.o -lm -o run

main.o:        general.h main.c
     cc  -g -c main.c
clr_int.o:   general.h clr_int.c
     cc  -g -c clr_int.c
find_E.o:    general.h find_E.c
     cc  -g -c find_E.c
find_G.o:    general.h find_G.c
     cc  -g -c find_G.c
find_Q.o:    general.h find_Q.c
     cc  -g -c find_Q.c
free_mem.o:  general.h free_mem.c
     cc  -g -c free_mem.c
g_tree.o:   general.h g_tree.c
     cc  -g -c g_tree.c
get_N.o:     general.h get_N.c
     cc  -g -c get_N.c
initialize.o: general.h initialize.c
     cc  -g -c initialize.c
pds.o:       general.h pds.c
     cc  -g -c pds.c
power.o:     general.h power.c
     cc  -g -c power.c
search.o:    general.h search.c
     cc  -g -c search.c
shift_L.o:   general.h shift_L.c
     cc  -g -c shift_L.c


/**********************************************************
     The file general.h is used to define variables and
structures that will be used throughout the simulation
program.  The general.h is an include file that is included
in every other program of the simulation.
**********************************************************/

#include  <stdio.h>
#include  <stdlib.h>
```

```
#include  <string.h>
#include  <math.h>

#define   MAX_N        31
#define   MIN_N        3
#define   BASE         2
#define   ONE          1.000000
#define   TWO          2.000000
#define   THREE        3.000000
#define   FOUR         4.000000
#define   DONE         1
#define   FOUND        1
#define   NOTFOUND     0
#define   GENERATE     1
#define   NOTGENERATE  0
#define   NEW_LINE     4
#define   PRINT_Q      31

typedef struct Generators{
   struct Generators *lptr;
   unsigned long int  gene;
   struct Generators *rptr;
}Generators;

typedef struct {
        Generators *Header;
}GENE;

GENE    Gene_tree, Quorum_tree;


/************************************************************
     The procedure main() is the main driver for the
simulation program.  It start from calling procedures
Get_N(), Find_E(), Initialize_tree(), Find_generators(),
Find_Quorums(), and, finally, Free_mem().
************************************************************/

#include "general.h"

main (argc, argv)
int   argc;
char *argv[];
{
int   N, E;
int   i;

/***  Get input N (number of node in a distributed
      system  ***/
      Get_N(argc, argv, &N);
      printf("N = %d\n",N);

/***  Find number of nodes in a quorum from the
      input N  ***/
      Find_E(N, &E);
```

```
      printf("E = %d\n",E);

/***  Initialize header of a generator tree  ***/
      Initialize_tree(&Gene_tree);

/***  Find all possible generators and keep them in
      a binary tree ***/
      Find_generators(N, E);

/***  Find all quorums from the constructed generators  ***/
      Find_Quorums(N, E, &Gene_tree, &Quorum_tree);

/***  Free memory after finish the simulation processes ***/
      Free_mem(&Gene_tree);
}


/************************************************************
     The procedure Get_N() is called by the procedure
main().  It is used to bound the value of N.  The value of N
can be between 3 and 31.
************************************************************/

#include "general.h"

Get_N(argc, argv, N)
int  argc;
char *(*argv);
int  *N;
{
int   i;

      if (argc < 2)
      {
         printf("\nMissing value of N.\n");
         printf("Enter  run N   (N is an integer).\n");
         exit(0);
      }

      i = 0;
      while (argv[1][i])
      {
         if ((argv[1][i] < '0') || (argv[1][i] > '9'))
         {
            printf("\n%s is not an integer.\n",argv[1]);
            printf("Enter  run N   (N is an integer).\n");
            exit(0);
         }
         i++;
      }

      *N = atoi(argv[1]);

      if (((*N) > MAX_N) || ((*N) < MIN_N))
      {
```

```
        printf("\n%d is out of range (%d..%d).\n",(*N),
               MIN_N, MAX_N);
        printf("Enter  run N   (N is an integer).\n");
        exit(0);
     }
}


/************************************************************
     The procedure Find_E() is called by the procedure
main(). It is used to compute value of E from a given N.
E is returned to the calling function.
************************************************************/

#include "general.h"

Find_E(N, E)
int  N;
int *E;
{
double  N_double, E_double;
float   E_float;

/*** convert integer to double ***/
     N_double = N;

/*** equation and conversion***/
     E_double = ceil((ONE + sqrt((FOUR * N_double)
                  - THREE)) / TWO);

/*** convert double to float ***/
     E_float  = (float)E_double;

/*** convert float to integer ***/
    *E = (int)E_float;
}


/************************************************************
     The procedure Initialize_tree() is called by the
procedures main() and Find_quorums().  It is used to
initialize heads of the generator's tree and quorum's tree.
************************************************************/

#include  "general.h"

Initialize_tree(T)
GENE   *T;
{
      (*T).Header = NULL;
}
```

```
/***********************************************************
     The procedure Find_generators() is called by the
procedure main().  It is used to find generators from all
the possible quorums.  The generators are kept in a binary
tree.  This procedure is a recursive procedure.
***********************************************************/

#include "general.h"

Find_generators(N, E)
int   N, E;
{
unsigned long int G;
int   Depth, Loop_begin, Loop_end;

      Depth = 1;
      Loop_begin = 1;
      Loop_end = E - 1;
      G = 0;
      Recu_find_gene(N, E, Depth, Loop_begin, Loop_end, G);
}


/***********************************************************
     The Recu_find_gene is the recursive part of the
procedure Find_generators().
***********************************************************/

Recu_find_gene(N, E, Depth, Loop_begin, Loop_end, G)
int  N, E;
int  Depth, Loop_begin, Loop_end;
unsigned long int G;
{
int  I, P;

     if (Depth <= E)
     {
        for (I = Loop_begin; I <= (N - Loop_end); I++)
        {
            G = G + Power(BASE,(I-1));
            if (Depth == E)
            {
               P = PDS(N, E, G);
               if (P == GENERATE)
               {
                  if (Search(&Gene_tree, G, N) == NOTFOUND)
                  {
                     Generators_tree(&Gene_tree, G);
                  }
               }
            }
            else
               Recu_find_gene(N, E, Depth+1, I+1,
                                   Loop_end-1, G);
            G = G - Power(BASE,(I-1));
```

```
         }
      }
      else
         return(DONE);
}


/************************************************************
     The function Power() is used to calculate power of a
given base variable.
************************************************************/

#include  "general.h"

unsigned long int Power(base, exp)
int   base;
int   exp;
{
unsigned long int  P;
int   i;

      P = 1;
      for (i=1; i <= exp; i++)
      {
          P = P * base;
      }
      return(P);
}


/************************************************************
     The function PDS() is called by the procedure
Find_generators().  It performs perfect difference set to
all possible quorums.  If a possible quorum is a perfect
difference set, then the PDS() return GENERATE (the quorum
is a generator).
************************************************************/

#include "general.h"

PDS(N, E, G)
int   N, E;
unsigned long int G;
{
int  Nodes_in_Q[MAX_N];
int  Result_pds[MAX_N];
unsigned long int G_temp;
int  i, j, k;
int  dif, count;

     Clear_int_ary(Nodes_in_Q);
     Clear_int_ary(Result_pds);
     count = 0;
     G_temp = G;
     for (i=1; i <= N; i++)
```

```
    {
        if ((G_temp%2) == 1)
           Nodes_in_Q[++count] = i;
        if (count == E)
           i = N+1;
        G_temp = G_temp >> 1;
    }
    for (i=E; i > 1; i--)
    {
        k = i-1;
        for (j=k; j > 0; j--)
        {
            dif = Nodes_in_Q[i] - Nodes_in_Q[j];
            Result_pds[dif] = 1;
            Result_pds[N-dif] = 1;
        }
    }
    for (i=1; i < N; i++)
    {
        if (Result_pds[i] != 1)
           return(NOTGENERATE);
    }
    return(GENERATE);
}


/************************************************************
       The procedure Clear_int_ary() is called by the
procedure PDS().  It is used to initialize array of integer.
************************************************************/

#include  "general.h"

Clear_int_ary(INT)
int INT[MAX_N];
{
int  i;

     for (i=0; i < MAX_N; i++)
         INT[i] = 0;
}


/************************************************************
     The function Search() is called by the procedure
Find_generators().  It is used to search for a duplicate key
(generator) in the generator's tree.  It also is a recursive
procedure.
************************************************************/

#include "general.h"

Search(T, G, N)
GENE     *T;
unsigned long int G;
```

```
{
int i;
int RET;

    RET = NOTFOUND;
    for (i=1; i<N; i++)
    {
        G = Shift_left(G, N);
        RET = Recu_search(&((*T).Header), G);
        if (RET == FOUND)
            i = N;
    }
    return(RET);
}


/************************************************************
     The function Recu_search() is the recursive part of
the function Search().
************************************************************/

Recu_search(sub_root, G)
Generators **sub_root;
unsigned long int G;
{
     if (*sub_root != NULL)
     {
        if ((**sub_root).gene == G)
        {
            return(FOUND);
        }
        else if ((**sub_root).gene > G)
        {
           return(Recu_search(&((**sub_root).lptr), G));
        }
        else if ((**sub_root).gene < G)
        {
           return(Recu_search(&((**sub_root).rptr), G));
        }
     }
     else
        return(NOTFOUND);
}


/************************************************************
     The function Shift_left() is used to shift bits in
unsigned long integers one position to the left.
************************************************************/

#include  "general.h"

unsigned long int  Shift_left(G, N)
unsigned long int  G;
int   N;
```

```
{
unsigned long int  bound_G;

      bound_G = Power(BASE,N) - 1;
      G = G << 1;
      if (G > bound_G)
      {
         G = G - Power(BASE,N);
         G = G + Power(BASE,0);
      }
      return(G);
}


/************************************************************
     The procedure Generators_tree is called by the
 procedure Find_generator().  It is a recursive procedure
that is used to add a generator to the generator's tree.
************************************************************/

#include "general.h"

Generators_tree(T, G)
GENE  *T;
unsigned long int G;
{
    Recu_g_tree(&((*T).Header), G);
}


/************************************************************
     The procedure Recu_g_tree() is the recursive part of
the procedure Generators_tree().  It finds an appropriate
place to add a generator in the generator's tree.
************************************************************/

Recu_g_tree(sub_root, G)
Generators **sub_root;
unsigned long int G;
{
Generators *newnode;

   if ((*sub_root) == NULL)
   {
/*** get new node ***/
     if ((newnode = (Generators*)malloc(sizeof(Generators)))
== NULL)
      {
          printf("Address of new node error\n");
          return(DONE);
      }

      (*newnode).gene = G;
      (*newnode).lptr = NULL;
      (*newnode).rptr = NULL;
```

```
/*** copy address of new node to its parent ***/
      *sub_root = newnode;
      return(DONE);
    }

/*** check duplicate key ***/
    if ((**sub_root).gene == G)
    {
       printf("Found duplicated key \n");
       return(DONE);
    }

/*** insert new key to the left of the tree ***/
    if ((**sub_root).gene > G)
    {
       Recu_g_tree(&((**sub_root).lptr), G);
       return(DONE);
    }

/*** insert the key to the right of the tree ***/
    if ((**sub_root).gene < G)
    {
       Recu_g_tree(&((**sub_root).rptr), G);
       return(DONE);
    }
}


/************************************************************
     The procedure Find_quorums is called by the procedure
main().  It is used to construct quorums from generators in
the generator's tree.  The set of generators that can be
used to construct quorums, satisfy the intersection
property, are kept in quorum's tree and will be printed out
as the output of the simulation program.
************************************************************/

#include  "general.h"

FILE  *O;

Find_Quorums(N, E, GT, QT)
int   N, E;
GENE *GT, *QT;
{
     if((O = fopen("output","w")) == NULL)
     { printf("Cannot open output file\n"); exit(0); }
     fprintf(O,"===================\nN = %d     E = %d\n",
             N, E);
     fprintf(O,"===================\n");
     if ((*GT).Header == NULL)
     {
        fclose(O);
        return(DONE);
     }
```

```
     Outside_recu_inorder(&((*GT).Header), &(*GT),
                          &(*QT), N);
     fclose(O);
}


/***********************************************************
     The procedure Outside_recu_inorder() is called by the
procedure Find_Quorums().  It is a recursive procedure that
calls another recursive procedure, Recu_find_Q_inorder().
The main reason is to perform nested loop to the generator's
tree to find set of generators that can be used to form
quorums which have the intersection property.  Then, the
procedure prints the set of generators, kept in the quorum's
tree.  It also prints the constructed quorums out of the set
of generators.
***********************************************************/

Outside_recu_inorder(sub_out, GT, QT, N)
Generators **sub_out;
GENE  *GT;
GENE  *QT;
int    N;
{
Generators  *newnode;
unsigned long int G;
int  S;

     if (*sub_out != NULL)
     {
         if ((**sub_out).lptr != NULL)
            Outside_recu_inorder(&((**sub_out).lptr),
                                 &(*GT), &(*QT), N);

/***  Initialize header of a quorum tree  ***/
         Initialize_tree(&(*QT));
/***  get new node  ***/
     if ((newnode = (Generators*)malloc(sizeof(Generators)))
== NULL)
       {    printf("Address of new node error\n");
  return(DONE);   }
         (*newnode).gene = (**sub_out).gene;
         (*newnode).lptr = NULL;
         (*newnode).rptr = NULL;
         (*QT).Header = newnode;

         Recu_find_Q_inorder(&((*GT).Header), &(*QT), N);

       fprintf(O,"====================\nGENERATOR(S)
:\n");
         Print_Generators(&(*QT), N);
         if (N <= PRINT_Q)
         {
          fprintf(O,"END.\n--------------------
\nQUORUMS(S) :\n");
```

```
            Print_Quorums(&(*QT), N);
        }
        Free_mem(&(*QT));
        fprintf(O,"END.\n===================\n");

        if ((**sub_out).rptr != NULL)
           Outside_recu_inorder(&((**sub_out).rptr),
                                     &(*GT), &(*QT), N);
    }
}


/************************************************************
     The procedure Recur_find_Q_inorder() is called by the
procedure Outside_recu_inorder().  It is the inside nested
loop.
************************************************************/

Recu_find_Q_inorder(sub_in, QT, N)
Generators **sub_in;
GENE  *QT;
int    N;
{
Generators *Q;

     if (*sub_in!= NULL)
     {
        if ((**sub_in).lptr != NULL)
           Recu_find_Q_inorder(&((**sub_in).lptr),
                                    &(*QT), N);
        Q = (*QT).Header;
        if ((*Q).gene != (**sub_in).gene)
           Build_quorums((**sub_in).gene, &(*QT), N);

        if ((**sub_in).rptr != NULL)
           Recu_find_Q_inorder(&((**sub_in).rptr),
                                    &(*QT), N);
     }
}


/************************************************************
     The procedure Build_quorums() is called by the
Recu_find_Q_inorder.  It is used to find generators that can
be used to form quorums (the generators that intersect all
the existing quorums).  If it found such a generator, then
it keeps the generator in the quorum's tree.
************************************************************/

Build_quorums(In_G, QT, N)
unsigned long int In_G;
GENE *QT;
int   N;
{
Generators  *Q, *newnode, *current;
```

```
unsigned long int In_Q;
int I;

/*** get new node ***/
 if ((newnode = (Generators*)malloc(sizeof(Generators)))
  == NULL)
   {    printf("Address of new node error\n");
        return(DONE);    }
   (*newnode).gene = In_G;
   (*newnode).lptr = NULL;
   (*newnode).rptr = NULL;

   Q = (*QT).Header;
   do {
        In_Q = (*Q).gene;
        for (I=1; I <= N; I++)
        {
            if ((In_Q & In_G) == 0)
            {
               free(newnode);
               return(DONE);
            }
            In_Q = Shift_left(In_Q, N);
        }
        current = Q;
        Q = (*Q).rptr;
   } while (Q != NULL);

   (*current).rptr = newnode;
}


/************************************************************
     The procedure Print_Generators() is called by the
procedure Outside_recu_inorder().  It is used to print the
set of generators, kept in the quorums tree, that are used
to form quorums that have the intersection property.
************************************************************/

Print_Generators(QT, N)
GENE *QT;
int   N;
{
Generators *Q;
unsigned long int P;
int   i;

      Q = (*QT).Header;
      while(Q != NULL)
      {
         P = (*Q).gene;
         for (i=1; i <= N; i++)
         {
             if ((P%2) == 1)
                fprintf(O,"%d ",i);
```

```
            P = P >> 1;
         }
         fprintf(O,"\n");
         Q = (*Q).rptr;
      }
}


/************************************************************
     The procedure Print_Quorums is called by the procedure
Outside_recu_inorder().  It is used to print quorums from
the set of generators, in the quorum's tree.
************************************************************/

Print_Quorums(QT, N)
GENE *QT;
int   N;
{
Generators *Q;
unsigned long int P1, P2;
int   i, j;
int   newline;

      newline = 0;
      Q = (*QT).Header;
      while(Q != NULL)
      {
         P1 = (*Q).gene;
         for (i=1; i <= N; i++)
         {
             P2 = P1;
             for (j=1; j <= N; j++)
             {
                 if ((P2%2) == 1)
                    fprintf(O,"%3d",j);
                 P2 = P2 >> 1;
             }
             newline++;
             if (newline == NEW_LINE)
             {
                 fprintf(O,"\n");
                 newline = 0;
             }
             else
                 fprintf(O,"   ");
             P1 = Shift_left(P1, N);
         }
         fprintf(O,"\n");
         newline = 0;
         Q = (*Q).rptr;
      }
}
```

```
/************************************************************
     The procedure Free_mem() is called by the procedure
main() and the procedure Outside_recu_inorder().  It is a
recursive procedure that is used to free memory from the
generator's tree and quorum's tree.
************************************************************/

#include "general.h"

Free_mem(T)
GENE *T;
{
      if ((*T).Header == NULL)
         return(DONE);
      else
         Recu_free_mem(&(*T).Header);
}


/************************************************************
     The procedure Recu_free_men is the recursive part of
the procedure Free_mem().
************************************************************/

Recu_free_mem(sub_root)
Generators **sub_root;
{
    if ((**sub_root).lptr != NULL)
       Recu_free_mem(&((**sub_root).lptr));
    if ((**sub_root).rptr != NULL)
       Recu_free_mem(&((**sub_root).rptr));

/*** return node to memory manager ***/
    free(*sub_root);

    return(DONE);
}
```

OUTPUTS OF THE SIMULATION PROGRAM WITH N = 5, 7, AND 9

```
====================
N = 5      E = 3
====================
GENERATOR(S) :
1 2 3
1 2 4
END.
--------------------
QUORUMS :
  1  2  3     2  3  4     3  4  5     1  4  5
  1  2  5
  1  2  4     2  3  5     1  3  4     2  4  5
  1  3  5
END.
====================


====================
N = 7      E = 3
====================
GENERATOR(S) :
1 2 4
END.
--------------------
QUORUMS :
  1  2  4     2  3  5     3  4  6     4  5  7
  1  5  6     2  6  7     1  3  7
END.
====================


====================
N = 9      E = 4
====================
GENERATOR(S) :
1 2 3 5
1 2 4 5
1 2 4 6
END.
--------------------
QUORUMS :
  1  2  3  5     2  3  4  6     3  4  5  7     4  5  6  8
  5  6  7  9     1  6  7  8     2  7  8  9     1  3  8  9
  1  2  4  9
  1  2  4  5     2  3  5  6     3  4  6  7     4  5  7  8
  5  6  8  9     1  6  7  9     1  2  7  8     2  3  8  9
  1  3  4  9
  1  2  4  6     2  3  5  7     3  4  6  8     4  5  7  9
  1  5  6  8     2  6  7  9     1  3  7  8     2  4  8  9
  1  3  5  9
END.
====================
```

VITA 2

SURAKIT TANAVUTIKAI

Candidate for the degree of

Master of Science

Thesis: AN EFFICIENT QUORUM STRUCTURE FOR DISTRIBUTED MUTUAL EXCLUSION

Major Field: Computer Science

Biographical:

Personal Date: Born at Ayutthaya, Thailand, on July 27, 1965, the son of Somkeat and Pensri Tanavutikai.

Education: Graduated from the Bangkok Commercial Campus in March 1986; received Bachelor of Business Administration Degree with a major in Information Systems from Institute of Technology and Vocational Education in March 1988; Completed requirements for the Master of Science degree at Oklahoma State University in July 1993.

Professional Experience: Programmer, Private Electric Authority of Thailand, June, 1988, to February 1989.