An Efficient Method for Mutual Exclusion in Truly Distributed Systems

Hao Chen and Jian Tang
Department of Computer Science
Memorial University of Newfoundland
St. John's, Nfld, A1C 5S7, Canada

Abstract

Many operations in a distributed system require mutual exclusion to guarantee correctness. Quorum methods have been widely proposed for implementing mutual exclusion. Majority voting is the best known quorum method. It has the merit of simplicity, but may incur high message overhead. Tree algorithm is an efficient structured quorum method to the mutual exclusion problems. The quorums generated by a tree algorithm are smaller on the average than those by a majority voting. However, the tree algorithm enforces a highly biased treatment to the nodes at different levels. This affects its performance in a distributed system where the nodes have similar characteristics. We propose a new structured quorum method which treats the nodes more evenly than the tree algorithm yet still preserves a satisfactory availability. We believe that this method is desirable for implementing mutual exclusion in a truly distributed system.

1 Introduction

Many operations in a distributed system require mutual exclusion to guarantee correctness. We term these operations restricted operations. Examples of restricted operations include updates on replicated data, naming of distributed objects, and atomic commitment. Mechanisms guaranteeing mutual exclusion should be both resilient and efficient. Resiliency usually implies high availability of resources in the face of failures, while efficiency implies low overhead incurred by performing restricted operations.

The issues relating to mutual exclusion have been studied extensively [1, 2, 3, 7, 13, 16]. Among the solutions suggested, the methods based on *quorums* have been widely accepted as effective mechanisms for implementing mutual exclusion. In a quorum method, a set of groups, called quorums, is predefined either directly or indirectly. At any time, the execution of a restricted operation is allowed only if a quorum can be constructed by following some specific rules. In general, quorums must have intersection property to ensure mutual exclusion. In [5], the authors study the general properties of a quorum set. A paradigm for optimizing the availability of quorum sets is proposed in [14].

The majority voting [6, 15] is the best known quorum method. It assigns a number votes to each node, and allows only those nodes that can collect a majority of votes to perform a restricted operation. Since at any time, there is only one majority in the system, mutual exclusion is guaranteed. The merit of the majority voting scheme is its simplicity, simple to understand and simple to implement. Its shortcoming is just that: it always requires a majority to make the operation succeed. This may incur a high communication overhead.

To reduce the cost, some methods have been proposed which are based on a logical structure of the network [1, 4, 8, 9, 11, 17]. In [11], the author associated with each site a set of sites and all such sets pairwise intersect. The sizes of these sets can be $\sqrt{n}$, compared with $\lceil \frac{n+1}{2} \rceil$ required by a majority voting. The drawback of this scheme is that it exhibits a very low availability when n gets larger. In [4, 9], the authors use 'grid' as the logical structure. In [8], the author uses a hierarchy to define quorum. Both methods are used mainly for synchronizing the read and write of replicated data. The idea of using tree as the logical structure for achieving mutual exclusion is suggested in [1]. This method has a very low cost in the best case. It is also easy to implement, since at runtime, each site only maintains a tree representing the logical structure required by the algorithm. However, the costs of the quorums are highly unevenly distributed in the quorum set for the tree algorithms. The nodes at the higher levels are heavily favored than those at the lower levels. (See section 2.1.) Consequently, in a distributed system where the nodes have similar characteristics, its performance in terms of either availability or the cost will be affected.

In this paper, we propose a new structured quorum method. It is based on a special logical structure, called *Triangular Net*. Our algorithm treats the nodes more evenly than the tree algorithm does and works better in the average case. We believe that this is a favorable property for a truly distributed system.

The rest of this paper is organized as follows. In section 2, we give a motivation to our algorithm. In section 3, we outline the model for our discussion. In section 4, we present the proposed algorithm and establish its correctness. In section 5, we analyze our algorithm by comparing its availability and cost with the majority voting and the tree algorithm. In section 6, we outline the extension of our algorithm to the general case. We conclude the paper by summarizing the main results.

1063-6927/94 $03.00

2 Motivation

2.1 Tree quorum algorithms

The tree quorum algorithms logically organize nodes as a tree. (We only consider binary tree for easy presentation. The extension to the general case is straightforward.) It constructs a quorum by selecting a root-to-leaf path in the tree. If such a path exists, then it is a quorum. If no such path exists due to node failures, then to form a quorum the failed node in some path must be (recursively) substituted for a collection of the paths with each starting from one of its children and ending at a leaf node. As an example, consider the four-level binary tree in Figure 1.

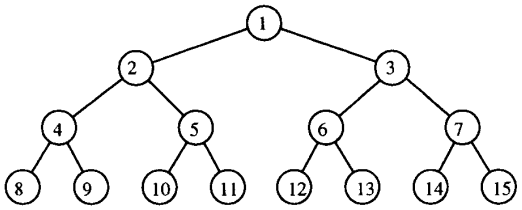

Fig 1. A binary tree used by the tree algorithm

If all nodes in set {1,2,5,10} are functioning, then this set is constructed as a quorum since the nodes in the set form a root-to-leaf path. If node 1 fails, but all nodes in set {2,5,10,3,6,12} are functioning, this set will also be constructed as a quorum, since 2,5,10 and 3,6,12 are the paths starting from the two children, nodes 2 and 3, of node 1 and ending at leaf nodes, 10 and 12, respectively. Finally, if in the above set, node 3 fails, but two more nodes, 7 and 14, are functioning, then it is still possible to construct a quorum, namely, {2,5,10,6,12,7,14}. This is because nodes 6,12 and nodes 7,14 are paths from the children of the failed node 3.

Clearly, in a tree quorum algorithm, nodes at the higher levels bear more weight than those at the lower levels when constructing a quorum. In the previous example, a single node at level 0 (i.e., node 1) always worth at least three lower level nodes in terms of the capability of forming a quorum. In the general case, a single node s at level i always worth at least $h-i-1$ lower level nodes, where h is the total number of levels in the tree. In other words, if node s fails, any quorum in which node s participated requires at least $h-i-1$ lower level nodes in lieu of s to remain a quorum. This means that single failure of level i nodes always increases the quorum size by at least $h-i-1$, which may be substantial when i is small. We believe that this has a negative impact on the average size of a quorum. Another impact is on the availability. Since nodes at different levels have very different capabilities of forming a quorum, the overall availability will be affected if all nodes have similar failure characteristics (which we believe are the most common cases). The factors that contribute to this shortcoming have to do with the structure of a tree. Firstly, for a tree structure different paths initiated at distinct nodes at the same level will never intersect. Thus whenever a high level node must be substituted by the union of the paths in its two subtrees, the size of the union is the sum of the sizes of the individual paths. In other words, every node in the paths contribute to the increased size. Secondly, the height and the width of the bottom level in a tree differ enormously (the ratio is approximately $log_2 n{:}n$ where n is the total number of nodes). As a result, the quorums formed along the bottom can be an order of magnitude larger than the quorums formed along the height.

2.2 Outline of a triangular net structure (TNS)

The goal of our algorithm is twofold. One is to preserve the logical clarity and simplicity as well as the essential properties for mutual exclusions, such as the intersection and minimality properties of quorums and non-dominance of a quorum set. The other is to diminish the discrepancy of the nodes at different levels in terms of the capabilities of forming a quorum in the hope of reducing the cost and enhancing availability. We achieve these goals by organizing the nodes in such a way that the shortcoming of a tree structure illustrated in the last section can be improved. To this end, we organize all nodes into a triangular net structure where the height and the width of the leaf level are roughly the same. Also two subtrees of a node intersect each other. Shown in Figure 2 is a typical TNS, where each node has two children and a distinguished node, node 1, is the root of the TNS.

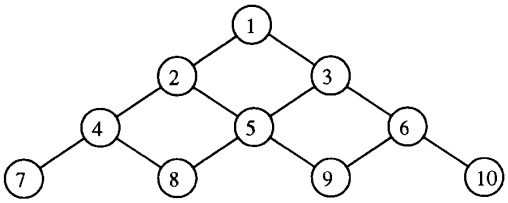

Fig 2. A TNS with four levels

Note that as shown in Figure 2, in a TNS except for the nodes in the two outer-most paths, each node has two parents. In a TNS, the higher level nodes have the similar capabilities to the lower level nodes in forming a quorum. For example, in the TNS of Figure 2, {1,2,5,8} is a quorum. If node 1 fails, it can be replaced by a single lower level node 3, resulting in another quorum {2,3,5,8}. In this case, node 1 and node 3 have the same capability of forming a quorum. Furthermore, quorums constructed along either the height or the bottom are roughly the same in size. For example, groups {1,2,5,8} and {7,8,9,10} are the two quorums constructed along the height and the bottom, respectively, in the TNS of Figure 2. (See section 4.2 for the detail of triangular net quorum algorithm.)

3 The model

Nodes and links are the basic components in a distributed system. Due to failures, or other exceptional reasons, the normal functions of a component may be disrupted. Two states, *up* and *down* are used to simulate this phenomena. At any instance of time, a component can be either up or down. A node that is up can send messages to and receive messages from the

other nodes and perform operations. A link that is up can deliver messages between its two adjacent nodes. We assume a down component simply stops functioning.

The execution of some operations requires the participation of a group of nodes. When this happens, the operation is first initiated at a node. We call the node where the operation is initiated the coordinator for the operation, and the other nodes in the group participants. The coordinator must ask for the permission from all participants in the group before it is allowed to carry out the operation. If all participants grant the permission, the operation is performed, otherwise, it is rejected. We say that an operation requires mutual exclusion if any two disjoint groups of nodes are disallowed to perform the operation in parallel. A natural way of ensuring mutual exclusion is to allow the operations to be performed only by a set of quorums which pairwise intersect. Thus, in order to perform the operations, the coordinator must obtain the permissions from the participants which form a quorum in the set.

4 The triangular net quorum (TNQ) algorithm

In this section, we give a detailed description of the triangular net quorum algorithm. We first give a formal definition of TNS. We then illustrate how a TNQ algorithm works based on the TNS structure. For easy presentation, we only consider the simple case. We will extend it to the general case in later sections.

4.1 Triangular net structure

As outlined in section 2, a TNS is a hierarchical structure which consists of a number of levels. We will use the convention that these levels are numbered as 0, 1,$\cdots$, in a top-down fashion.

Definition: For $h \geq 0$, an $h+1$ level binary TNS (We will omit the term 'binary' when no confusion is possible.) is a collection of interconnected nodes arranged by levels such that for all $i, 0 \leq i \leq h$:

1. There are exactly $i+1$ nodes, denoted by $s_{i0}, \cdots, s_{i,i}$, at level i;

2. For all i, j, $0 \leq i \leq h-1$ and $0 \leq j \leq i$, node $s_{i,j}$ has two children $s_{i+1,j}$ and $s_{i+1,j+1}$ at level $i+1$.

For the above definition, we call the node at level 0 the root of the TNS, the nodes at level h leaves. For all $i, 0 \leq i \leq h$, we call nodes s_{i0} and $s_{i,i}$ side nodes. Figure 3 is the general structure of the TNSs.

Since each node in a TNS represents an actual network node, it can be only in one of the two states, up and down. From the functional point of view, all nodes that are down are the same. However, from the viewpoint of a TNQ algorithm, all down nodes do not exhibit the same characteristics in forming a quorum: some down nodes may have enough up successors to form a quorum, while the others do not. To simulate such a scenario, we need two additional states.

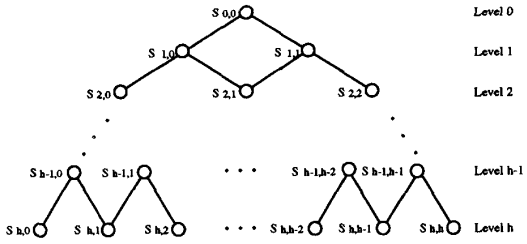

Fig 3. The general structure of a (binary) TNS

Definition: A node s is *open* if the following conditions hold:

1. if s is a leaf, then s is up;

2. if s is an internal node, then either s is up and one of its children is open or s is down and both of its children are open.

otherwise, we say that node s is *closed*.

For easy references, we will call a state up or down a *UD-state*, and a state open or closed an *OC-state*.

Our intention here is to use the OC-state to signify the significance of a node to any quorum which the TNQ algorithm can construct based on the current network state. Specifically, a node being closed signifies that it is insignificant to any quorums constructed on the current network state. Let us still consider the TNS in Figure 2. Suppose nodes 2,3,5,4 and 9 are up, and all the rest are down. By definition, node 9 is open. This in turn implies node 5 is open, which in turn implies nodes 2 and 3 are open. Now consider node 1, which is presumably down. Since both of its children are open, it is open too. It can be verified that except for these five nodes, all the rest are closed. Note that even though node 4 is up, it is closed since both of its children are closed. Our algorithm in section 4.2 will tell us that the only quorum that can be constructed is {2,3,5,9}. Apparently, all the closed nodes are insignificant to this quorum. For example, none of the closed nodes 4, 7 and 8 is part of this quorum. Now, assume a different network state where the nodes that are up are 1, 4, 5 and 6. It is easy to verify that for this state none of the nodes is open. Accordingly, our algorithm will not be able to construct any quorum.

4.2 The algorithm

The input to the algorithm is the complete set of network nodes organized into a TNS, as well as the states of each node (i.e., either up or down). For clarity, we present our algorithm as a two pass process. In the first pass, The procedure *Mark* marks all the nodes in the TNS rooted at t as either open or closed. This information will later be used by function *Formquorum* in the second pass to determine how to construct a quorum[1]. Both procedures work recursively. The logic underlying procedure *Mark* directly follows from the definition of an OC-state. It first checks if t has already been marked. This is because subtrees may overlap and a node may be visited by many instances of recursive calls. The actual marking actions proceed

[1] In the real implementation, it is not difficult to combine these two passes into a single pass if doing so is deemed desirable.

from the bottom to the top. Procedure *Formquorum* will be called only if the root of the TNS is open. It returns a quorum in the TNS rooted at node t. Note that with the exception of leaf nodes, *Formquorum* will always bypass a node with both children open (i.e., does not include it into the quorum) and go straight to process its children. In this case, it does not even care if the parent node is up. This *children-first* approach is different from the *parent-first* approach used by the tree quorum algorithm. (In the parent-first approach, if it is open the parent will never be bypassed unless it is down.) In general, children-first approach tends to construct a quorum along the bottom while parent-first approach along the height. From the structure of TNS where the two subtrees of a node always overlap, a parent-first approach does not automatically ensure minimality. However, by using children-first approach, it can be proven that the minimality is always guaranteed.

```
ALGORITHM:
/*input: a TNS rooted at node T */
  Mark(T);
  IF T is marked as closed THEN
     stop; /* no quorum can be formed*/
  ELSE
     Formquorum(T);

PROCEDURE Mark(t:NODE)
BEGIN
  IF t is not marked THEN
     IF (t is a leaf) THEN
       IF (t is up) THEN
         mark t as open;
       ELSE mark t as closed;
     ELSE {
       Mark(t.left);
       Mark(t.right);
       IF (t is up) THEN
         IF ((t.left is marked as open) OR
             (t.right is marked as open)) THEN
           mark t as open;
         ELSE mark t as closed;
       ELSE IF ((t.left is marked as open) AND
             (t.right is marked as open)) THEN
           mark t as open;
         ELSE mark t as closed;}
END;

FUNCTION Formquorum(t:NODE)
/* note: t is assumed to be open */
BEGIN
  IF (t is a leaf) THEN
       RETURN({t});
  ELSE
     IF ((t.left is marked as open) AND
        (t.right is marked as open)) THEN
          RETURN(Formquorum(t.left) ∪
                 Formquorum(t.right));
     ELSE IF (t.left is marked as open) THEN
          RETURN({t}∪Formquorum(t.left));
       ELSE
          RETURN({t}∪Formquorum(t.right));
END;
```

Still consider the TNS in Figure 2. Suppose nodes 2, 3, 4, 5, 6, 7 and 8 are up and all the rest are down. In the first pass, the procedure *Mark* will mark each node as either open or closed as shown in Figure 4.a. (The small letters o and c attached to the nodes denote OC-states open and closed, respectively. The broken-lined circles denote the nodes that are down.) In the second pass, *Formquorum* will chose {3,5,7,8} as the quorum. Suppose node 9 comes up but node 7 goes down. Procedure *Mark* will make the TNS as depicted in Figure 4.b. *Formquorum* will construct quorum {4,6,8,9}. Now suppose node 10 comes up and node 3 goes down, resulting in the state of Figure 4.c. Accordingly, the algorithm will construct quorum {4,8,9,10}.

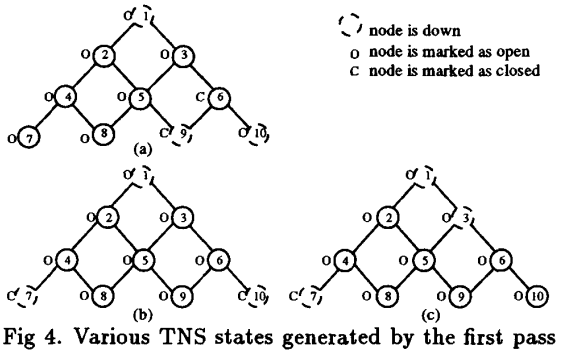

Fig 4. Various TNS states generated by the first pass

From this example, we have the following observations: 1. among the three quorums constructed by the TNQ algorithm based on the different network states, none of them is a subset of the others; 2. they pairwise intersect; 3. the differences in quorum sizes are small. In the following sections, we will prove that the first two statements are in fact true in all cases. The third statement describes the property of the TNQ algorithm which underlies its low cost.

4.3 Correctness

In this section, we establish the basic properties of the TNQ algorithm that are essential to a 'good' mutual exclusion mechanism. These include the following:

1. Minimality: $\forall G,\ H \in S,\ G \not\subseteq H$;
2. Intersection: $\forall G,\ H \in S,\ G \cap H \neq \phi$;
3. Non-dominance[2]: $\forall G$, if $G \cap H \neq \phi$ for all $H \in S$, then $\exists Q \in S$ such that $G \supseteq Q$.

These properties are established through the following lemmas and theorems. Due to the space limitation, we will only provide the proofs for Theorem 2 and Theorem 3, deemed they are less obvious than the others.

In what follows, we use T to denote the root of the TNS for which we establish the properties. We use S_T to denote the set of all quorums that can possibly be constructed by the TNQ algorithm with a particular TNS rooted at T. For a node p in the TNS, we use $p.left$ and $p.right$ to denote the left and right child of p, respectively. Let R and S be two sets of groups of

[2] In some literature, the quorum set which satisfies the three properties listed here is called ND-coterie. The term 'non-dominance' follows from term 'non-dominated' coterie.

nodes, we define $R \otimes S \equiv \{U \cup V : U \in R \ \& \ V \in S\}$. In a reasonable abuse of symbols, we sometimes use a letter to represent both the root and the entire structure of a TNS. When this happens, we will precede the letter by the term 'node' or 'TNS' to indicate its actual meaning. We will use the term, TNS state, to denote the TNS in which each node is associated with a UD-state and an OC-state (i.e., all nodes have been marked.)

We first introduce two lemmas which will be used in the subsequent proofs.

Lemma 1: S_T can be written as $S_T = G_1 \cup G_2 \cup G_3$ where $G_1 = \{\{T\}\} \otimes S_{T.left}$, $G_2 = \{\{T\}\} \otimes S_{T.right}$ and $G_3 \subseteq S_{T.left} \otimes S_{T.left}$.

Lemma 2: For any $g \in S_T$, g can be constructed by the TNQ algorithm with a TNS state M where $\forall s \in g, \forall r \notin g$, s is up and r is down in M.

THEOREM 1. The set of all possible quorums constructed by a TNQ algorithm has the intersection property.

THEOREM 2. S_T has the minimality property.

Proof: Let $g \in S_T$ be an arbitrary quorum, and $x \in g$ be an arbitrary node. Assume $q = g - \{x\}$. We now prove that $q \notin S_T$. Assume the contrary. By Lemma 2, g can be constructed in a TNS state, say M_1, such that $\forall s \in g$(s is up in M_1) and $\forall s \notin g$(s is down in M_1). Likewise, q can be constructed in a TNS state, say M_2, such that $\forall s \in q$(s is up in M_2) and $\forall s \notin q$(s is down in M_2). Note that M_1 and M_2 defer only in the state of node x.

Since $x \in g$ and g can be constructed by the TNQ algorithm with M_1, x must be open. Thus x either is a leaf node or has exactly one open child.

Since it is down in M_2, if x is a leaf node, then it must be closed in M_2. If it is not a leaf node, according to the definition of an OC-state, it must also be closed in M_2. Note that x must not be the root, since otherwise q could not possibly be constructed from M_2. Let y be one of its open parents on which the recursive call is made when g is constructed from M_1. (Note that one of the parents of such kind must exist since otherwise no recursive call could be made on node x and therefore node x would not have been included in g.) Let z be the other child of y. If z is open in M_1, we claim y must be down in M_1. Assume not. Then y is up in M_1. Clearly, $y \notin g$. Since x is closed while z is still open in M_2, (Note that the OC-state of z will not be affected by the state change of x.) $y \in q$, a contradiction to the definition of q. Thus y is down in M_1 and therefore in M_2. Since x is closed in M_2, y is closed in M_2. We have proven that if z is open in M_1, then y is closed in M_2. On the other hand, if z is closed in M_1, then z will remain closed in M_2. Again, since node x too is closed in M_2, y is closed in M_2.

Now let u be the parent of y on which the recursive call is made when g is constructed from M_1. The same argument as above can prove that u is closed in M_2. We can repeat this process until we reach the root of M_2. This means that no quorum can be constructed within M_2, a contradiction to the assumption that q can be constructed in M_2. □

THEOREM 3: S_T has the non-dominance property.

Proof: We prove the theorem by induction on the number h of levels in the TNS.

Base: $h = 1$. There is only one node T in the TNS. Our algorithm will generate $S_T \equiv \{\{T\}\}$. Obviously this set is complete.

Inductive step: $h > 1$. Again we write S_T as the union of the following three subsets.

1. $G_1 = \{\{T\}\} \otimes S_{T.left}$
2. $G_2 = \{\{T\}\} \otimes S_{T.right}$
3. $G_3 \subseteq S_{T.left} \otimes S_{T.right}$

Assume g is an arbitrary group such that g intersects all the groups in S_T. Two cases are possible.

Case 1: $T \in g$. Thus g must intersect all the groups in G_3. If g intersects all the groups in $S_{T.left}$, then by the induction hypothesis, $g \supseteq g_1$ for some $g_1 \in S_{T.left}$. Thus $g \supseteq \{T\} \cup g_1$, which is in the set G_1. Now assume g does not intersect all the groups in $S_{T.left}$. Assume $p_1 \in S_{T.left}$ such that $g \cap p_1 = \phi$. We claim that g intersects all the groups in $S_{T.right}$. If not, let $p_2 \in S_{T.right}$ such that $g \cap p_2 = \phi$. Thus $g \cap (p_1 \cup p_2) = \phi$. By Lemma 2, p_1 and p_2 can be constructed respectively from the TNS states M_1 and M_2, where $\forall s \in p_1, \forall r \notin p_1$, s is up and r is down in M_1, and $\forall s \in p_2, \forall r \notin p_2$, s is up and r is down in M_2. We define a TNS state M for structure T as follows: $\forall s \in p_1 \cup p_2, \forall r \notin p_1 \cup p_2$, s is up and r is down in M. Clearly, $\forall s$, s is up in M_1 or $M_2 \Rightarrow s$ is up in M. Note that nodes $T.left$ and $T.right$ are open in M_1 and M_2, respectively, since otherwise p_1 or p_2 could not be constructed. Clearly, they will remain open in M. Thus node T is open in M. This means that a quorum can be constructed from M. Let p denote this quorum. Thus $p \subseteq G_3$. Since all the nodes not in $p_1 \cup p_2$ are down in M, we have $p \subseteq p_1 \cup p_2$. This implies $g \cap p = \phi$. This contradicts to our assumption that g intersects all the groups in G_3. Thus g intersects every group in $S_{T.right}$. Using the similar arguments to those in the first half of this case, we can prove that g is a superset of some group in G_2.

Case 2: $T \notin g$. Thus g intersects all groups in $S_{T.left} \cup S_{T.right}$. By the induction hypothesis, $\exists g_1 \in S_{T.left}, g \supseteq g_1$ and $\exists g_2 \in S_{T.right}, g \supseteq g_2$. Thus $g \supseteq g_1 \cup g_2$. Using the similar arguments to those in case 1, we can prove $\exists p \in G_3, p \subseteq g_1 \cup g_2$. Thus $g \supseteq p$. □

5 Analysis of triangular net quorum algorithm

In this section, we analyze the performance of a TNQ algorithm by comparing its availability and cost with another two influential algorithms for mutual exclusion, majority voting and tree quorum algorithm.

5.1 Availability

We assume that each node has independent failure mode, with the probability p of being up. We define the availability of a mutual exclusion algorithm to be the probability that a quorum can be constructed by the algorithm.

Assume there are n nodes. For majority voting algorithm, The size of a quorum is always $\lceil (n+1)/2 \rceil$.

Thus the availability is

$$A_{maj} = \sum_{i=\lceil (n+1/2) \rceil}^{n} \binom{n}{i} p^i (1-p)^{n-i}.$$

The availability of the *tree* quorum algorithm is obtained from the following recursive relation. Let A_{h+1} be the availability of the tree algorithm for a tree of level $h+1$ where $h \geq 0$. We have

$A_1 = p$;

$A_{h+1} = pA_h(1-A_h) + p(1-A_h)A_h + pA_h^2 + (1-p)A_h^2$, i.e., $A_{h+1} = 2pA_h + (1-2p)A_h^2$.

The calculation of the availability by TNQ algorithm is more complex due to the fact that subtrees of a node always overlap in the TNS. Thus the probabilities that a quorum can be constructed in the subtrees of a node are not independent of each other. At this time, we do not know how to use a closed form or a recursive relation to specify this availability in general case. To calculate the availability, the following method is used. Clearly, the availability of a TNQ algorithm is the probability that the root of the associated TNS is open. In a TNS with $h+1$ levels, for all $i, 0 \leq i \leq h$, there are exactly $i+1$ nodes at level i. Let S_i be the set of all possible combinations of OC-states (open/close) and R_i the set of all possible combinations of UD-states (up/down) of the nodes at level i. For convenience, we simply call the members of these two sets states. Clearly, S_i and R_i each contains 2^{i+1} different states. Note that from the way the OC-state is defined, for $0 \leq i \leq h-1$, each state in S_i is the union of the two states with one in R_i and the other in S_{i+1}, and the union of any two states with one in R_i and the other in S_{i+1} forms a state in S_i, with the exception that at the leaf level, we have $R_h = S_h$.

The algorithm we use to compute the availability of the TNQ algorithm is briefly described as follows. Given the probability p that each node in the TNS is up. Let P_{ij} and Q_{ij} be the probabilities that the jth states in R_i and S_i occur, where $1 \leq j \leq 2^{i+1}$. (We assume there is an order on the elements in S_i and R_i.) Since the jth state in R_i is just a collection of the UD-states of all the nodes at level i and the UD-states of different nodes are independent of each other, P_{ij} can be computed immediately (i.e, independently of any other levels). The algorithm computes Q_{ij} for all $i, j, 0 \leq i \leq h, 1 \leq j \leq 2^{i+1}$ level by level in a bottom up fashion. From the notes we made at the end of the last paragraph, Q_{ij} is the product of P_{ik} and $Q_{i+1,l}$, assuming the union of the kth state in R_i and the lth state in S_{i+1} gives the jth state in S_i. When Q_{01} and Q_{02} are finally generated, whichever of the two corresponds to the probability that the root is open is the availability of the TNQ algorithm.

Using the algorithm described above, we evaluated the availabilities of TNQ algorithm and the majority voting algorithm for 6, 15 and 28 nodes. We evaluated the availabilities of the tree algorithm for 7, 15 and 31 nodes, since these numbers are the closest to those used by the TNQ algorithm while maintaining the balanced tree structure. In each case, we let the node probabilities vary from 0.5 to 1 with a step size 0.0025. In all cases but 6 nodes, the majority voting has the highest availability. For TNQ and tree algorithm, it is interesting to note that there is a turning point of the node probability which is close to 0.7. The tree algorithm has better availability below this point while the TNQ algorithm does better above it. We have listed ten sample data obtained for each of the three algorithms in cases of 15 and 28 nodes (15 and 31 nodes for the tree algorithm) in Table 1 and Table 2, respectively. The left-most column in each table is the list of the sample node probabilities based on which the availabilities of the three algorithms are evaluated. We use the double horizontal lines to indicate the estimated locations of the turning points. (The complete data is available from the authors of this paper.) Note that this comparison is made in cases where the TNQ algorithms use less nodes than the tree algorithm does.

5.2 The sizes of the quorums

In this section, we will compare the maximum, minimum and average sizes of the quorums constructed by the three algorithms.

For a system of n nodes, maximum, minimum and the average sizes of the quorums constructed by the majority voting algorithm are always $\lceil (n+1)/2 \rceil$. For a tree algorithm, the maximum and the minimum quorum sizes are $(n+1)/2$ and $log_2 n$, respectively. To obtain the average quorum size for the tree algorithm, let n_i and s_i be the total number of quorums and the average quorum sizes for a tree with i levels. We have the following recursive relations:

$n_0 = 1$;

$s_0 = 1$;

$n_{i+1} = 2 * n_i + n_i^2$;

$s_{i+1} = (2*(s_i+1)*n_i + 2*s_i*n_i^2)/n_{i+1}$.

We now analyze the sizes for the quorums by the TNQ algorithm. It follows from the way a TNS is marked that the root will be open if all the leaf nodes are up. For a TNS with n nodes, there are approximately $\sqrt{2n}$ leaf nodes. Thus the minimum quorum size for the TNQ algorithm is at most $\sqrt{2n}$. For the maximum and the average sizes, we can only rely on the experimental results due to the complication caused by the overlap of the subtrees of a node. We have evaluated TNSs with the sizes ranging from 3 to 28. The method for our evaluation is based on the enumeration. The results show that the maximum size generated by the TNQ algorithm is slightly larger than the tree algorithm. For example, in case of 15 nodes, the maximum size by the TNQ algorithm is 9

Table 1: Availabilities when 15 nodes are used

Probab.	Tree alg.	TNQ alg.	Maj.alg
0.5350	0.586881	0.585572	0.608726
0.5850	0.703873	0.701325	0.749973
0.6350	0.804545	0.801980	0.860720
0.6850	0.883253	0.881760	0.934645
0.7350	0.938493	0.938440	0.975475
0.7375	0.940667	0.940680	0.976815
0.7850	0.972582	0.973501	0.993238
0.8350	0.990407	0.991434	0.998825
0.8850	0.997755	0.998303	0.999907
0.9350	0.999775	0.999882	0.999998

Table 2: Availabilities when 31 nodes are used by tree and 28 nodes are used by others

Probab.	Tree alg.	TNQ alg.	Maj.alg
0.5500	0.646689	0.643741	0.635560
0.6000	0.774970	0.771155	0.813154
0.6500	0.872822	0.870531	0.926422
0.6975	0.935023	0.935012	0.977673
0.7000	0.937527	0.937624	0.979236
0.7500	0.974164	0.975709	0.996218
0.8000	0.991495	0.992996	0.999626
0.8500	0.998006	0.998732	0.999985
0.9000	0.999743	0.999990	0.999999
0.9500	0.999992	0.999999	0.999999

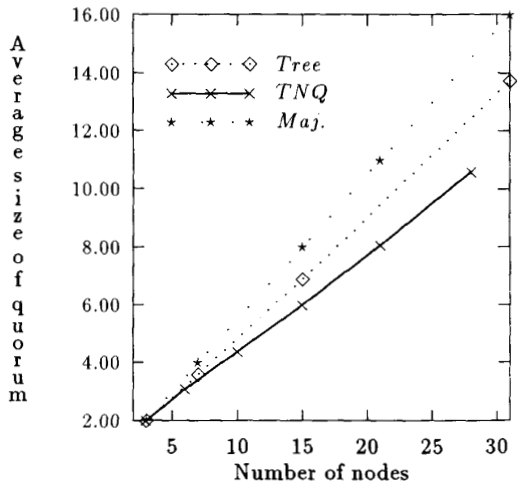

Fig 5. The average quorum sizes for the 3 algorithms

and that by the tree algorithm is 8. In case of 28 nodes (31 nodes for the tree algorithm), the maximum size by the TNQ is the same as that by the tree algorithm, both being 16. However, the average size by the TNQ algorithm is constantly smaller than that by the tree algorithm. This has been depicted in Figure 5. Our data also indicates that in all cases the average quorum sizes by the TNQ algorithm is about 11% smaller than that by the tree algorithm. Among the three algorithm, the majority voting algorithm has the highest average cost.

The above result implies that the quorum sizes generated by the TNQ algorithm are more evenly distributed in the quorum space than that by the tree algorithm. In addition, our data also shows that the nodes at different levels have more even capabilities of forming a quorum under the TNQ algorithm than that under the tree algorithm. For example, under the TNQ algorithm with 15 nodes, the root participates in 96 out of a total of 258 quorums. The average size of the quorums in which the root participates is 5.375, while the average size of the quorums in which the root does not participate is 6.377. On the other hand, under the tree algorithm also with 15 nodes, the root only participates in 30 out of a total of 255 quorums. The average size of the quorums the root participates is 4.6, compared with the average size of 7.2 of the quorums the root does not participate.

6 Generalization

The TNS we have discussed so far allows an internal node to have only two children. This restriction can be lifted to make the TNQ algorithm more general. Note that if a parent has more then two children, then a child may also have more than two parents. The assignment of the parent nodes to each child node must be properly distributed. In general, suppose each internal node has m children and, counted from the left the nodes at level i are s_{i0}, s_{i1},$\cdots$,s_{iq}. Then the TNS will be organized in such a way that s_{i0} and s_{iq} each has one parent, s_{i1} and $s_{i,q-1}$ each has two parents,$\cdots$, $s_{i,m-2}$ and $s_{i,q-m+2}$ each has $m-1$ parents, and for all $p, m-1 \leq p \leq q-m+1$, $s_{i,p}$ has m parents. Shown in Figure 6 is an extended three-level TNS where each internal node has three children.

Let $r, \frac{m+1}{2} \leq r \leq m$ be an integer. We define the OC-states for a node s as follows.

Definition: A node s is open if the following conditions hold:

1. if s is a leaf node, then s is up;
2. if s is an internal node, then either it is up and has at least $m-r+1$ open children, or it has at least r open children.

otherwise it is closed.

A natural way to design the TNQ algorithm for the extended TNS is to simulate that for the binary TNS. That is, the first pass defines the OC-state for each node and the second pass constructs the quorum. While the first pass here is a close analog of that for the restricted case, the second pass has an aspect which was not present in the previous case, namely, nondeterminism. Now, starting with the root of the TNS, if a node has at least r open children, then the the TNQ issues recursive calls to *any* r open children of the node. The quorums returned is the union of the quorums returned by these r calls. If a node has less than r but at least $m-r+1$ open children then if itself is up then the recursive calls will be made on any $m-r+1$ open children of the node. The quorum returned will be the union of the quorums returned by these $m-r+1$ calls with the node itself included. Unlike the second pass in section 4.2 where the action is completely deterministic, here it may return different quorums for the same TNS state. This non-determinism has an impact on the quorum set generated. We have found that while this simple version of TNQ can guarantee intersection property, it cannot guarantee minimality property. Briefly, this is because even though a call on any individual parent only generates the minimum number of recursive calls (i.e., either r or $m-r+1$) on its children, the calls on two or more parents which share some children may aggregately generate more than the minimum calls on the children of one of them. To guarantee minimality, more sophisticated TNQ is needed. We are currently studying this issue and will present the results in a future paper.

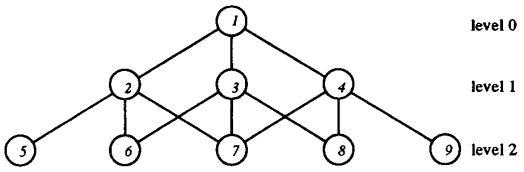

Fig 6. A three level ternary TNS

The extended TNS has the smaller height but wider bottom than the binary TNS if roughly the same number of nodes are used. Thus it decreases the minimum quorum size. It is not clear to us at this time how the maximum and average quorum sizes in a general TNQ defer from those in a binary TNS. Our conjecture is that these sizes will also decrease since it provides more overlapping among the subtrees of a node. Whether and how this conjecture can be verified will be an interesting topic for the future study.

7 Conclusion

In this paper, we describe the triangular net algorithm for achieving mutual exclusion. Our algorithm is based on organizing the network nodes into a triangular net structure. Like a tree structure, it contains a number of levels and the nodes at different levels are associated by parent-child relationship. Unlike a tree structure, however, different children may share the same parents. It is because of this increased sharing our algorithm possesses some desirable properties which a tree algorithm does not have. We show that our algorithm provides a more uniform treatment to the nodes, which we believe is desirable for a truly distributed system. We show that our algorithm has a good average case behavior. We compare the performance of our algorithm with the majority voting and the tree algorithm. The results show that our algorithm has a better average cost than both of them. When the node probability is above the turning points, our algorithm provides a higher availability than the tree algorithm.

References

[1] D. Agrawal and A. EL Abbadi, "An efficient solution to the distributed mutual exclusion problem," *ACM Trans. on Comput. Syst.*, Vol. 9, No. 1, pp. 1-20, 1991.

[2] M. Ahamad and M. H. Ammar, "Performance characterization of quorum consensus algorithm for replicated data", *IEEE Trans. Software Engineering*, Vol. 17, No. 4, pp. 492-496, 1992.

[3] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources", *Proc. 2nd Intl. Conf. on Software Engineering*, pp. 562-570, 1976.

[4] S. Y. Cheung, M. H. Ammar and M. Ahamad, "The grid protocol: A high performance scheme for maintaining replicated data", *Proc. Intl. Conf. on Data Engineering*, pp. 438-445, 1990.

[5] H. Garcia-Molina and H. Barbara, "How to assign votes in a distributed system", *J. ACM*, Vol. 32, No. 4, pp. 841-860, 1985.

[6] D. K. Gifford, "Weighted voting for replicated data", *Proc. 7th Symp. on Operating Syst. Principles*, pp. 150-162, 1979.

[7] J. M. Helary, N. Plouzeau and M. Raynal, "A distributed algorithm for mutual exclusion in an arbitrary network", *Computer J.*, Vol. 31, No. 4, pp. 289-295, 1988.

[8] A. Kumar, "Hierarchical quorum consensus: A new algorithm for managing replicated data", *IEEE Trans. Comput.*, Vol. 40, No. 9, pp. 996-1004, 1991.

[9] A. Kumar and S. Y. Cheung, "A high availability $\sqrt{N}$ hierarchical grid algorithm for replicated data", *Inform. Process. Lett.*, Vol. 40, No. 6, pp. 311-316, 1991.

[10] L. Lamport, "The implementation of reliable distributed multiprocess systems", *Comput. Networks*, Vol. 2, No. 2, pp. 95-114, 1978.

[11] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems", *ACM Trans. on Comput. Syst.*, Vol. 3, No. 2, pp. 145-159, 1985.

[12] F. B. Schneider, "Synchronization in distributed programs", *ACM Trans. on Program. Lang. Syst.*, Vol. 4, No. 2, pp. 125-148, 1982.

[13] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm", *ACM Trans. on Comput. Syst.*, Vol. 3, No. 4, pp. 344-349, 1985.

[14] J. Tang and N. Natarajan, "Obtaining coteries that optimize the availability of replicated databases", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 5, No. 2, pp. 309-321, 1993.

[15] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases", *ACM Trans. on Database Syst.*, Vol. 4, No. 2, pp. 180-209, 1979.

[16] J. L. Van de Snepscheut, "Fair mutual exclusion on a graph of processes", *Distribute. Comput.*, Vol. 2, No. 2, pp. 113-115, 1987.

[17] C. Wu and G. G. Belford, "The triangular lattice protocol: A highly fault tolerant and highly efficient protocol for replicated data", *IEEE 11th Proc. Symp. on Reliable Distributed Syst.*, pp. 66-73, 1992.