Title

Optimizing and Evaluating Algorithms for Replicated Data Concurrency Control

Authors

Kumar, A

Segev, A

Publication Date

1989-02-01

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

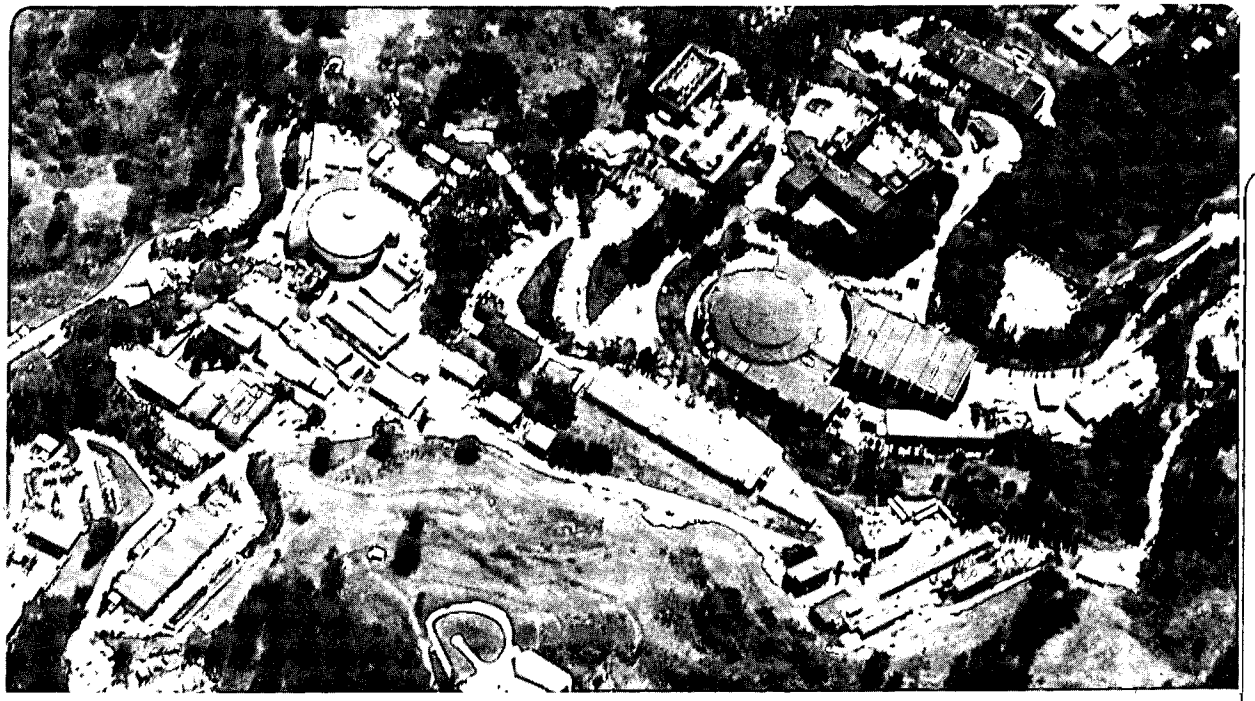
Information and Computing Sciences Division

To be presented at the Ninth International Conference on Distributed Computing Systems, Newport Beach, CA, June 5–9, 1989

Optimizing and Evaluating Algorithms for Replicated Data Concurrency Control

A. Kumar and A. Segev

February 1989



Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

DISCLAIMER

LBL-25435

Optimizing & Evaluating Algorithms For Replicated Data Concurrency Control

Akhil Kumar and Arie Segev

School of Business Administration
University of California

and

Computing Science Research & Development
Information & Computing Sciences Division
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, California 94720

February 1989

Proceedings of the 9th International Conference on Distributed Computing Systems, Newport Beach, June 1989

This work was supported by the Director, Office of Energy Research, Applied Mathematical Sciences Research Program, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098 and by an Arthur Andersen & Company Doctoral Dissertation Fellowship.

OPTIMIZING AND EVALUATING ALGORITHMS FOR REPLICATED DATA CONCURRENCY CONTROL

Akhil Kumar and Arie Segev

School of Business Administration and
Computer Science Research Dept, Lawrence Berkeley Lab
University of California
Berkeley, Ca., 94720

Abstract

Techniques for optimizing a static voting type algorithm are presented. Our optimization model is based on minimizing communications cost subject to a given availability constraint. We describe a semi-exhaustive algorithm for solving this model. This algorithm utilizes a novel signature-based method for identifying equivalent vote combinations, and also an efficient algorithm for computing availability. Static algorithms naturally have the advantage of simplicity; however, votes and quorum sizes are not allowed to vary. Therefore, the optimized static algorithm was compared against the available copies method, a dynamic algorithm, to understand the relative performance of the two types of algorithms. We found that if realistic reconfiguration times are assumed, then no one type of algorithm is uniformly better. The factors that influence relative performance have been identified. The available copies algorithm does better when the update traffic ratio is small, and the variability in the inter-site communications cost is low.

1. Introduction

A replicated data environment is one in which multiple copies of a file are present. By replicating data, system reliability may be increased to satisfy the high up-time requirements in several real-time applications, such as banking and airlines. Clearly, if copies of a file reside on several computers with independent failure modes, then the file system would be more reliable. The disadvantage, however, is that the copies must be

This research was supported by the Applied Mathematics Sciences Research Program of the Office of Energy Research U.S. Department of Energy under contract DE-AC03-76SF00098 and by an Arthur Andersen & Co. Foundation Doctoral Dissertation Fellowship.

kept mutually consistent by synchronizing transactions at different sites so that a global serialization order is ensured. For instance, two independent transactions must not be allowed to simultaneously update different copies of the same file. Hence, the concurrency control algorithm becomes more complex and also more expensive to implement. The additional communications and processing cost arises because several rounds of messages must be exchanged with other sites while implementing the algorithm.

Several popular methods for replicated data concurrency control are based on the formation of quorums [BERN87, DAVI85]. We refer to such methods as "voting-type" or quorum consensus (QC) class of algorithms [GIFF79, THOM79]. These algorithms may be categorized into two broad types: static and dynamic. In the static algorithms the assignment of votes to sites is fixed a priori, while in the dynamic ones, the votes and quorum sizes are allowed to vary dynamically. Our objective in this paper is to develop techniques for optimizing the assignment of votes in a static algorithm and compare the relative performance of an optimized static algorithm against a dynamic algorithm. Static algorithms have the advantage of simplicity and ease of implementation, and consequently, if the performance of the two types of algorithms is comparable, then static algorithms would be preferred.

In a previous paper [KUMA88], we have introduced failure tolerance as a measure of reliability and described a model for integrating failure tolerance with communications cost. The objective function in the model there was to minimize communications cost subject to a failure tolerance constraint. Here we introduce **availability** as a more realistic measure of reliability, and incorporate it in an optimization model. Then, we describe in detail new solution techniques for the model, and finally, compare an optimized static algorithm against a dynamic algorithm.

Availability is defined in probabilistic terms as follows: Given n copies of a file such that each is up with probability, p_i, the **availability,** A is the probability that both a read and a write quorum can be formed. Consider an example where 3 copies of a file reside at different sites. If each p_i is 0.9, and the size of a quorum is 2, then the overall availability of the file is computed as:

$$A = 3\times(0.9)^2\times 0.1 + (0.9)^3 = 0.97.$$

This means that by replicating a file at 3 sites instead of 1, the availability can be increased from 0.9 to 0.97. One can similarly show that if n is 5, and the quorum size is 3, then A increases to 0.991.

Our optimization model is as follows:

Model 1

Minimize Communications Cost

such that:

Availability $> cut-off$

The common feature in voting-type algorithms [GIFF79] is that each site, i is assigned a vote, v_i, and in order to perform various operations quorums must be formed by assembling votes. To perform a read (or write) operation, a transaction must assemble a read (or write) quorum of sites such that the votes of all the sites in the quorum add up to a predefined threshold, Q_r (or Q_w). The basic principle behind the algorithm is that the sum of these two thresholds must exceed the total sum of all votes, i.e.,

$$\text{Invariant 1:} \qquad \sum_i v_i < Q_r + Q_w$$

Hence a read and a write operation cannot proceed simultaneously. Moreover, the write threshold is larger than half the sum of all votes, i.e.,

$$\text{Invariant 2:} \qquad \sum_i v_i < 2 \times Q_w$$

Thus, two write operations are prevented from proceeding simultaneously. It is important to note that the above two invariants do not enforce unique values upon Q_r and Q_w. Furthermore, the v_i's do not have to assume unique values. Hence, several alternative sets of solutions for these variables exist. The read-one write-all method is a special case of the quorum consensus method with each v_i and Q_r equal to 1, and Q_w equal to n (assuming there are n copies of the file). This leads to better performance for queries at the expense of poorer performance for updates and works well in an environment where a very large fraction of the transactions are queries.

The basic QC algorithm described above is a static algorithm because the votes and quorum sizes are fixed a priori. This restriction does not apply to dynamic algorithms, and in these the votes and quorum sizes are adjusted suitably as sites fail and recover. Examples of such algorithms are: the **Missing Writes method** [EAGE81, EAGE83], and the **Virtual Partition method** [ELAB85]. In the Missing Writes method, the read-one write-all method is implemented when all the sites in the network are up; however, if any site goes down, the size of the read quorum is increased while that of the write quorum is reduced. After the failure is repaired it reverts to the old scheme. In the virtual partition algorithm, each site maintains a view consisting of all the sites that it can communicate with, and within this view, it implements the read-one write-all method.

Other quorum based methods of the dynamic type are: the **vote reassignment method** [BARB86], the **quorum adjustment method** [HERL87], and the **dynamic voting algorithm** [JAJO87]. In the vote reassignment and the quorum adjustment methods, sites that are up can change their votes on detecting failures of other sites by following a certain protocol. In the dynamic voting method, additional information regarding the number of sites at which the most recent update was performed is stored. This makes it possible to shrink the size of a quorum dynamically if site failures or partitions occur. For instance, consider an object replicated at 5 sites designated as A, B, C, D and E, and further assume that the most recent update to this object was performed at sites A, B, and C. If a subsequent link failure isolates site A from B and C, the latter two sites can still continue to perform updates because they contain a majority among the sites at which the most recent update was performed. Therefore, this method allows updates to take place even with fewer than a majority of the sites available.

Another dynamic algorithm is the **available copies** method [BERN84]. In this method, query transactions can read from any single site while update transactions must write to all the sites that are up. Moreover, in order for the algorithm to work correctly, each transaction must perform two additional steps called missing writes validation and access validation. In the first, a transaction must ensure that all copies it tried to, but could not, write are still unavailable. In the second step, a transaction must ensure that all copies it read or wrote are still available. The **primary copy** algorithm [STON79] is based on designating one copy as a primary copy, and each transaction must update it before committing. Later, updates are spooled to the other copies also. If the primary copy fails, then a new primary copy is designated.

In [GARC84, GARC85], the concept of coteries is introduced as an alternative to quorums. A coterie is defined as a collection of intersecting minimal subsets of sites. It is shown that there exist several coteries which do not have a corresponding vote assignment, and therefore, the solution space of vote assignments is a subset of the coterie solution space. It is also shown that the problems of finding an optimal vote assignment and an optimal coterie assignment in order to maximize availability have exponential complexity.

Two types of failures can occur: link failures and site failures. We will assume that a link failure that does not cause a partition is transparent because every site can continue to access all other sites. Further, it is assumed that partitions don't occur, i.e., if a site does not respond within a certain time-out interval, it is down.

The organization of this paper is as follows. In Section 2, we define basic vote assignment concepts, used later in the paper. Then, in Section 3, we analyze the complexity of the vote assignment problem, while Section 4 turns to a semi-exhaustive algorithm for solving the vote assignment problem. In Section 5, we devise an efficient algorithm for computing availability and illustrate it with an example. Finally, computational results for the optimized QC algorithm and the available copies algorithm are presented in Section 6.

2. Vote Assignment Definitions and Concepts

In this section, we shall define some basic concepts in the context of the vote assignment problem and shall use them through the remainder of the paper.

2.1. Vote Assignment versus Vote Combination

At the outset, a distinction must be drawn between a **vote assignment** and a **vote combination.** A vote combination for n sites is an n-tuple consisting of n elements. It becomes a **vote assignment** when each element in this combination is assigned to a **specific** site. Therefore, if each element in a vote combination is unique, then there are n! specific assignments for that combination and they may be produced by enumerating all permutations of the vote combination. On the other hand, if duplicates are present in a vote combination, then the number of corresponding assignments is less than n!. In the extreme case if all elements in the combination are equal, then only one vote assignment exists. For example, in a 5-site problem, the vote combination (1,1,1,1,1) is also a vote assignment because only one permutation exists for this combination of votes. On the other hand, the combination (5,4,3,2,1) has 5! (or 120) different assignments.

Without loss of generality, we shall represent a vote combination as a non-increasing sequence of votes, $V = (v_i, \cdots, v_n)$ where:

$$v_1 \geq v_2 \geq \cdots \geq v_j \geq \cdots \geq v_n$$

Notationally, we shall represent a vote assignment, $\overline{V}$, in a similar way; however, in an assignment $\overline{V}$, the i^{th} element of the vector specifically represents the vote given to site i and therefore, the elements occur in site-number order.

2.2. Quorum Size

The size of a quorum, q is the minimum number of votes needed to make a majority among all votes. For simplicity, a read and a write quorum are required to be equal since the system is considered unavailable if either quorum cannot be formed. (Note, however, that we allow non-equal votes).

2.3. Corresponding Assignment for a Vote Combination

A vote combination can also be viewed as an assignment in which the i^{th} element is assigned to site i. Such an assignment will be called the **corresponding assignment** for a vote combination. For example, $\overline{V}$=(5,4,2,2,1) is the corresponding assignment for the combination, V=(5,4,2,2,1). For brevity, the term "availability of a vote combination" will refer to the availability of the corresponding assignment for the vote combination.

2.4. Equivalent Vote Assignments and Combinations

Two vote assignments, $\overline{V}_1$ and $\overline{V}_2$ are **equivalent** if for every group of sites in $\overline{V}_1$ that can form a quorum, the same group of sites in $\overline{V}_2$ can also form a quorum and vice versa. For example, it is easy to verify that (1,1,1,1,1) and (2,2,2,2,1) are equivalent vote assignments. In the former case, q is 3, while in the latter it is 5. We define two vote combinations, V_1 and V_2 to be equivalent if their corresponding assignments are equivalent.

The following result is derived from our definition of equivalence.

Theorem 1: The availability of two equivalent vote assignments, $\overline{V}_1$ and $\overline{V}_2$ is always equal for all sets of p_i's, where p_i is the reliability of site i.

Proof : Given n sites there are exactly 2^n alternative states representing all combinations of up and down sites. In any given state, a quorum can be formed if the sum of the votes of all up sites is at least q. Moreover, the probability that the system is in a specific state is computed as the product of all p_i's for all up sites i, and of all $(1-p_j)$'s for all down sites j. Consequently, the availability of a vote assignment may be computed by first identifying those states in which a quorum can be formed, and aggregating

the probability that the system is in one of these states.† From the above definition of equivalence it follows that the collection of states in which a quorum can be formed is identical for $\overline{V}_1$ and $\overline{V}_2$, and hence, the availability is equal for both assignments (for all sets of p_i's).

Corollary 1: The availability of two equivalent vote combinations, V_1 and V_2 is always equal for all sets of p_i's where p_i is the reliability of site i.

The corollary follows directly from Theorem 1 and Section 2.3. These results are used in Procedure SE, described in Section 4.1.

2.5. Dominating Vote Assignments

An assignment, $\overline{V}_1$ **dominates** another assignment, $\overline{V}_2$ if for each set of sites in $\overline{V}_2$ that can form a quorum, a quorum can always be formed by the same or a smaller set of sites in $\overline{V}_1$; however, the converse is not true. This means that $\overline{V}_1$ is superior to $\overline{V}_2$ because it would have a greater availability.

It has been shown in [GARC85] that an assignment in which the total number of votes is even is always dominated by a vote assignment in which the total number of votes is made odd by assigning 1 extra vote. For example, if n is 4, an "odd" assignment represented by (2,1,1,1) is superior to the "even" assignment (1,1,1,1). We shall, therefore, consider only those assignments (or combinations) in which the total number of votes assigned is an odd number.

† A more efficient method for computing availability is discussed in Section 5.

2.6. Active Votes

A vote, v_j assigned to site j in an assignment, $\overline{V}$ is said to be an **active** vote if there exists at least one quorum of sites which includes site j such that if site j is withdrawn, the remaining sites do not form a quorum.

In the assignment (4,2,2,2,1), v_5 does not contribute to the formation of any quorum because all quorums must include at least two of the 4 other sites. Therefore, v_5 is an inactive vote and this assignment is treated as an invalid 5-site assignment. For an n-site problem, we shall restrict ourselves to those assignments in which each vote is active.

3. Complexity of the Vote Assignment Problem

In [GARC85] it has been shown by heuristic arguments based on visualizing a hypercube that an upper bound on the total number of vote assignments for n sites is 2^{n^2}. Here we give another derivation which is intuitively simpler.

For n sites, there are 2^n possible groups corresponding to all possible combinations involving the inclusion and exclusion of each site. From these 2^n groups, we may arbitrarily choose any n groups that should form a quorum, and write n simultaneous equations with n variables. Each system of n equations may then be solved for the v_i's. Of course, some of the systems of equations may not have any solution. However, this allows us to place an upper bound on the number of vote assignments as $\binom{2^n}{n}$. This expression represents the number of ways in which n elements may be chosen from a universe of 2^n elements. The complexity of the above expression is $O(2^{n^2})$.

Not only does this number include several systems of equations which have no solution, but it also involves a large number of permutations of a given vote combination.

For instance, (5,3,3,1,1), (3,1,1,5,5), (1,1,3,3,5), etc., are examples of permutations of the vote combination, (5,3,3,1,1), and each such permutation is included in the upper bound above. For these two reasons, it is possible that a tighter upper bound may exist for the total number of unique combinations, but has not yet been found. In the next section, we develop an algorithm that enumerates a very large number of vote combinations, and also implement this algorithm to show the number of assignments that exist for $n \leq 9$.

4. A Semi-Exhaustive (SE) Algorithm

In this section we shall describe a procedure (Procedure SE) to generate non-equivalent vote combinations, and then use it to develop an algorithm (Algorithm SE-A) to solve model 1. Algorithm SE-A produces alternative assignments for each vote combination generated by Procedure SE. For each assignment, the algorithm evaluates the cost and availability, and selects the minimum cost assignment which meets the reliability criterion.

4.1. Procedure SE

This procedure starts with an initial vote combination, and further new combinations are generated by incrementing the votes in the current combination. To make the procedure more efficient, we devise a method to identify and discard a combination which is equivalent to one already produced. Thus, only non-equivalent vote combinations are generated.

The initial combination is:

For odd n : v_i's $= 1$ for all i,
For even n : $v_1 = 2$, v_i's $= 1$ for $i = 2, \cdots, n$

Since each v_i must be a positive integer strictly greater than 0, this is obviously the combination with the smallest odd sum of votes.

Next, we generate new combinations by considering all ways in which 2 votes may be added to the current combination. (A combination is **new** if it is not equivalent to a combination which has already been produced). This will give us a set of v_i's which add up to $n+2$ or $n+3$ depending on whether n is odd or even, respectively. At each subsequent stage, we consider only the new combinations produced in the previous stage and use them to produce further combinations. For example, say n is 5, and the starting combination is (1,1,1,1,1). At the next step, the total number of votes must be increased by 2, and the alternative combinations that result are (3,1,1,1,1), and (2,2,1,1,1). Each of these combinations will have to be checked for equivalence against the combination (1,1,1,1,1). If found to be new, then it is added to the existing list of combinations.

Each successive iteration of the while-loop in the main section of Procedure SE (described below) corresponds to finding new combinations by adding r more votes to each current N vote combination. Ordinarily N is incremented by 2 in each successive iteration and r remains at 2; however, if at any iteration no new combination is found, then in subsequent iterations r is incremented by 2 and N is kept fixed until a new combination is realized. At this point, N is incremented by the current value of r. There are two stopping rules. The first rule is that the total sum of votes becomes greater than a pre-specified maximum, while the second rule is that r becomes larger than 25. The second rule corresponds to a situation in which no new combination is found for a gap of 25 votes, a reasonably large interval. Such a gap would arise if no new combination is found even after 12 successive iterations.

The availability of each prospective new combination is computed by applying Algorithm AVAIL (to be described in Section 5) to its corresponding assignment. The value of availability so computed is called the **signature** for the combination. Since, as shown in Section 2.4, two equivalent combinations will always have the same availability, this gives us a convenient way of eliminating redundant vote combinations. Each new signature is checked against an array of signatures, and if not found in it, it is added to this array. A collision occurs if the signature already exists in the array, and in this case the combination is discarded. Since Procedure SE is used only to generate non-equivalent combinations, an arbitrary set of p_i's can be used for this purpose (the actual p_i's are, however, required in Algorithm SE-A to compute the real availability, as described in Section 4.3). Although the converse of Theorem 1, i.e., if two assignments have the same availability then they must be equivalent has not been proven, it has been verified experimentally that false collisions are prevented by choosing site reliabilities such that no two p_i's are equal. Therefore, this rule should be observed while selecting p_i's for use in Procedure SE.

The main steps are as follows:

Procedure SE

1. (Initialization). The starting combination is $V = (v_1, \cdots, v_n)$ such that:

   For odd n : v_i's = 1 for all i; Sum of votes, N = n.
   For even n : $v_1 = 2$, v_i's = 1 for $i = 2, \cdots, n$; N = n+1.

   Also, initialize:

   the final solution set, S = $\emptyset$,
   the solution set for a total of N votes, $S_N = \{V\}$
   the p_i values from input data,
   the signature array, SA=0.

2. (Main section)

```
r = 2.
while (N+r < N_max and r < 25){
   for each combination, V in S_N {
      add r more votes to V to create combination, V_temp .
      compute availability, A_temp for V_temp .
      If A_temp nomem SA array{
            add V_temp to S_N+r .
            add A_temp to SA array.
                                  }
                                   }
   If (S_N+r ≠ ∅)
            r = 2.  N = N + 2.
    else
            r = r + 2.

                  }
```

3. The final solution set, S is obtained by concatenating all non-empty sets, S_i.

4.2. Implementing Procedure SE

We implemented this algorithm and Table 1 shows the number of unique vote combinations that were found for various values of n. For n equal to 7, no vote combinations were found for N_{max} between 44 and 69, at which point the program terminated. When n was 8 and 9, the program was interrupted at N_{max} equal to 50 and 45 respectively. We repeated this experiment for two different sets of probability values and found the same result in each case. This was done to eliminate the possibility of two non-equivalent combinations having the same signature.

It should be reemphasized that the numbers in Table 1 represent vote combinations and not specific vote assignments. For each combination, there are potentially n! assignments if each vote is unique. An exhaustive algorithm to solve the problem must consider all the specific assignments corresponding to each vote combination, and therefore, multiplying the numbers in Table 1 by n! gives a better appreciation for the real

complexity of the problem.

4.3. A Vote Assignment Algorithm

We shall now describe Algorithm SE-A to solve model 1. Procedure SE is used within this algorithm to generate a set of vote combinations, and then all possible assignments of each combination are evaluated to find the best solution.

Algorithm SE-A

1. Produce Set S of vote combinations from Procedure SE.

2. Produce set P of vote assignments by permuting all combinations in S.

3. (Main section)

$min\text{-}cost = \infty$
For each assignment, $\overline{V}$ in set P
  if (AVAIL ($\overline{V}$) $\geq$ cut-off and COMPUTE_COST ($\overline{V}$) $<$ min-cost){
    $min\text{-}cost$ = cost.
    $min\text{-}assignment = \overline{V}$.
  }

# of Sites(n)	Maximum Sum of Votes	# of Combinations
1	-	1
2	-	1
3	-	1
4	-	1
5	-	4
6	-	19
7	-	133
8	50	>2071
9	45	>7603

Table 1: Number of unique vote combinations for various n

4. The best solution is given by *min−assignment* .

COMPUTE_COST($\overline{V}$)

1. For each site i, determine the set of other sites to be included in i's quorum, Q_i by:

   (a) first ordering all other sites j in ascending order of $\frac{c_{ij}}{v_j}$, and
   (b) then choosing the first k sites from this sequence such that:

$$v_i + \sum_k v_k \geq q$$

2. The cost of assignment, $\overline{V}$ is:

$$\sum_i \sum_{k \in Q_i} (q_i + u_i) \times c_{ik}$$

3. return (cost)

The above algorithm is conceptually straightforward but computationally intensive. It utilizes procedures COMPUTE_COST and AVAIL to compute respectively the cost and availability of an assignment.

The **COMPUTE_COST** procedure is required to determine the total communications cost for assignment $\overline{V}$, given the query and update volumes (q_i and u_i respectively) of each site in addition to the c_{ij}'s. The major step is to determine the set of sites, Q_i which site i must communicate with in order to form a quorum. This subproblem can be formulated as a knapsack problem [BUDN77] and the technique described in step 1 is a heuristic solution method for it. Once the Q_i's are known, the total cost is computed as in step 2. Now we shall turn to describe Procedure AVAIL.

5. Computing Availability

In this section, we devise an algorithm for computing availability when several copies of a file exist, and the vote assigned to each site, v_i and its reliability, p_i are known. This algorithm is used by Algorithm SE-A described above. One simple method for computing availability is by enumerating all possible combinations of up and down sites, identifying those combinations in which a quorum can be formed, and computing the aggregate probability of all such combinations. This is clearly an inefficient scheme. Here we describe a more efficient algorithm and show that it is considerably less expensive than complete enumeration. A formal problem definition and the details of our algorithm are given in the following section. Subsequently, an example will be given to demonstrate the algorithm.

5.1. Algorithm Description

Problem Definition: Compute the availability for a vote assignment represented by the vector $\overline{V} = (v_1, v_2, \cdots, v_n)$, and a quorum of size q. The reliability, or the probability that site i is up, is denoted by p_i. Without loss of generality, it is assumed that:

$$v_1 \geq v_2 \geq \cdots \geq v_j \geq \cdots \geq v_n$$

Our Algorithm **AVAIL** for computing availability is based on first constructing a tree which we call the quorum subset tree. Each branch in this tree corresponds to the inclusion and exclusion of a certain site in a quorum and by following the path from a leaf node to the root one can generate alternative subsets. The procedure for constructing this tree is called **BUILD-TREE.** We shall describe this procedure first, and then use it as a subroutine in Algorithm AVAIL.

The root of the tree is labeled as level 1, and nodes at lower levels are numbered successively. An information triple is maintained at each node as follows:

(votes included, votes excluded, votes remaining)

where:

votes included (VI): total votes of sites included so far

votes excluded (VE): total votes of sites excluded so far

votes remaining (VR): total votes still to be assigned

Based on this (VI, VE, VR) triple, a decision is made as to whether a particular node is "fathomed". If a node is fathomed, then no further branching is done from there. Otherwise, the branching process is repeated. At an unfathomed level i node, we consider the effect of including or excluding site i by constructing two branches: one corresponding to including site i and the other corresponding to excluding site i. The main steps of procedure **BUILD_TREE** are described below. Figure 1 shows a tree that has been constructed from this algorithm for $\overline{V} = (5,3,3,1,1)$.

BUILD_TREE

1. (initialization)

   $i = 1$

   The root is marked by the triple (0,0,W), where W is the sum of all votes. Next, two branches are constructed from the root: one corresponding to the inclusion of site 1 (the "include 1" branch) and the other corresponding to excluding site 1 (the "exclude 1" branch). The nodes at the end of these two branches represent level 2 of the tree.

2. At each level $i+1$ node, the triple at the node is computed from the i^{th} level parent node in the following manner. If the branch leading to the node is an inclu-

sion branch, then:

$\mathrm{VI} = (VI)_p + v_i$

$\mathrm{VE} = (VE)_p$

$\mathrm{VR} = (VR)_p - v_i$

(The subscript p denotes the parent node values while the unsubscripted VI, VR, and VE represent the child node values).

On the other hand, if the branch is an exclusion branch, then the new values are computed as:

$\mathrm{VI} = (VI)_p$

$\mathrm{VE} = (VE)_p + v_i$

$\mathrm{VR} = (VR)_p - v_i$

3. (Fathoming Step) This step is repeated for all nodes at level $i+1$.

We consider 4 cases:

CASE 1: $VI \geq q$

If VI for a new node is greater than or equal to q, then this node is marked as "fathomed - type 1".

CASE 2: $VI < q$ and $VI+VR = q$

If $VI+VR$ for the new node is equal to q, then this means that all the remaining sites must be included in order to create a quorum. In this case, we put a note on the node to indicate that sites $i+1, i+2, \cdots, n$ must be included, and mark the node as "fathomed - type 1".

CASE 3: $VE > W-q$

This means that enough sites have been excluded already to preclude the formation of a quorum from the remaining sites. Hence, this node is marked as "fathomed -

type 2".

CASE 4: $VI < q$ and $VI + VR > q$

This node cannot be fathomed.

4. If all nodes at level $i+1$ have been fathomed, then the tree construction is complete, and the algorithm stops, else

```
    {
i=i+1.
At each unfathomed level i node, construct
   an "include i" and an "exclude i" branch
go to step 2.
    }
```

Now we list the steps of Algorithm AVAIL and a detailed description follows.

Algorithm AVAIL

1. First construct a tree from procedure **BUILD_TREE** described above.

2. Set Availability, A = 0.

3. Next, for each "fathomed - type 1" node, do {

```
P = 1.
Traverse tree upwards from the node and
  Do Until root is reached {
    If there is an "include i" in the path,
      P = P × p_i ,
    else
      If there is an "exclude i" in the path,
        P = P(1 - p_i ).
    If current node is not the root, traverse the next higher path.
                        }
A = A + P
                                    }
```

4. The Availability of the file system is given by A.

The Availability is computed by following the path from each "fathomed -- type 1" node to the root backwards. Each "include i" along a path corresponds to p_i and each

"exclude i" corresponds to $(1-p_i)$. The product of the p_i's and $(1-p_i)$'s is computed along each such path, and aggregating the individual products gives the availability. The following example will demonstrate this algorithm.

5.2. An Example

Here we describe an example to illustrate Algorithm AVAIL.

Example 1: Compute the availability for the following $\overline{V}$ vector:

$$\overline{V} = (5,3,3,1,1)$$

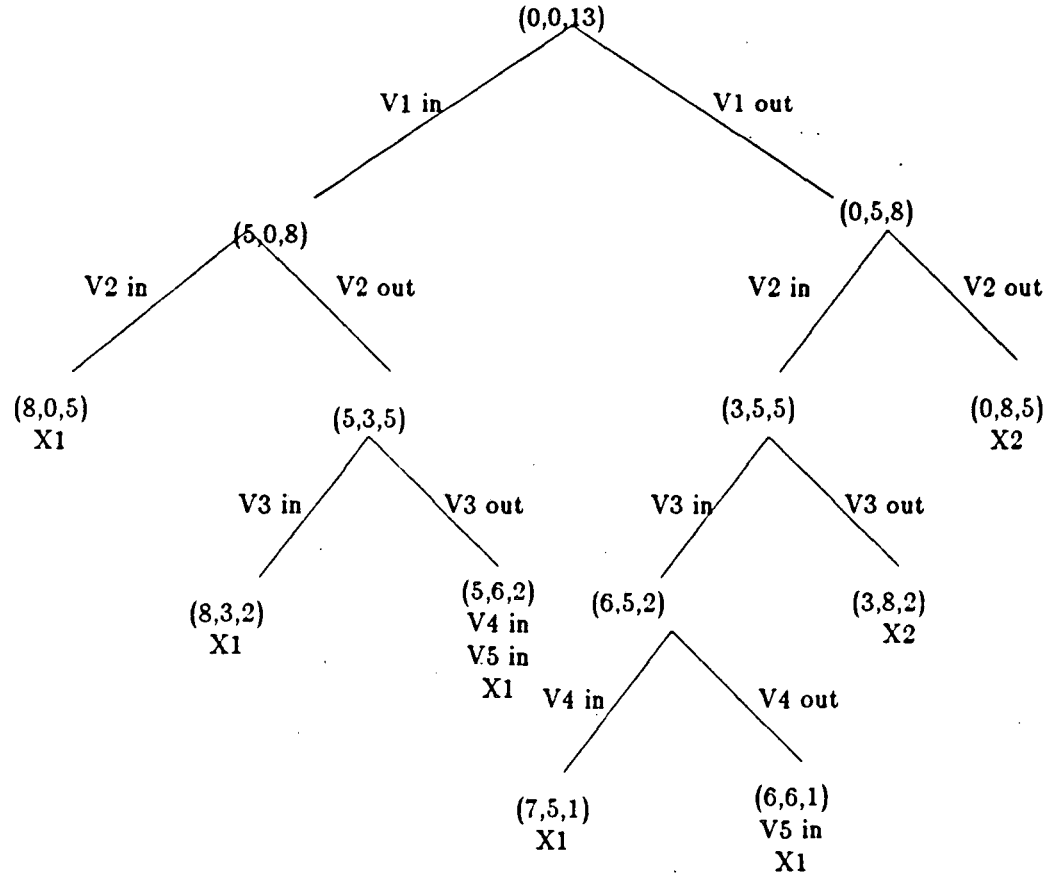
The first step in implementing Algorithm AVAIL is to construct the quorum subset tree. This is shown in Figure 1. Using this tree, the availability for this example is computed as:

$$A = p_1p_2 + p_1(1-p_2)p_3 + p_1(1-p_2)(1-p_3)p_4p_5 + (1-p_1)p_2p_3p_4 + (1-p_1)p_2p_3(1-p_4)p_5.$$

6. Experimental Results

We implemented the semi-exhaustive algorithm and compared its performance against the available copies algorithm [BERN84] which is a dynamic algorithm. The results are presented in this section.

Algorithm SE-A was implemented for 7-site example networks. For each network, we first generate an inter-site unit communications cost matrix. The costs are obtained from uniform distributions in which the range is allowed to vary. The following uniform distributions were used: U(1,1), U(1,5), and U(1,10). For each distribution, the availability cut-off was varied from 0.93 to 0.99 in intervals of 0.01. At each cut-off value, the corresponding communications cost was computed using Algorithm SE-A. The results of these computations are presented in Table 2. The total traffic volumes and the reliability of each site, p_i are also given in this table.



X1 -- Fathomed, Type 1

X2 -- Fathomed, Type 2

Figure 1: Quorum subset tree for Example 1

A	U(1,1)	U(1,5)	U(1,10)
0.93	39	95	101
0.94	39	96	101
0.95	44	101	107
0.96	44	111	120
0.97	53	119	140
0.98	62	128	146
0.99	78	167	240

Table 2: Static Algorithm: Minimum Communications Cost for 3 cost distributions (7 sites)
$\overline{P}$ = (0.91,0.90,0.89,0.87,0.86,0.85,0.84)
Total Traffic Volumes = (5,7,4,9,1,5,8)

To make a comparison against the available copies algorithm we need to compute the communications cost and availability again. In the static case, the read and write quorums were required to be equal as discussed in Section 2.2; therefore, the total communications cost does not depend on the write ratio (ρ_w), defined as the fraction of all transactions that are updates. In the dynamic case, however, this is not true, and the communications cost would vary for different values of ρ_w. In Table 3, we have computed the communications cost for different values of ρ_w. The 3 columns correspond to the 3 different distributions for the unit inter-site communications cost as in Table 2. The rows of Table 3 correspond to different values of ρ_w. The total traffic volumes were the same as in Table 2; however, the read and write traffic components were varied depending upon ρ_w.

The availability in the dynamic case is computed differently than in the static case. We now turn to describe our method for performing this computation. In the available copies algorithm, transactions running at the time a site fails or recovers may have to be aborted, and restarted. This will result in a time delay during which period the system

will be unavailable. We call this interval a reconfiguration interval, t_{recon}. It is assumed that each site has a mean time to failure (MTTF) and a mean time to repair (MTTR). For simplicity, all sites are assigned the same value of MTTF and also of MTTR. To make the analysis tractable, it is reasonable to further assume that each site will fail and recover once, on the average, in an MTTF + MTTR cycle, and at each failure and recovery point, the system will be unavailable for t_{recon} length of time. Thus, if availability is defined as the fraction of time for which the system is available during one cycle, it may be expressed as:

$$1 - \frac{2 \times t_{recon} \times n}{MTTF + MTTR}$$

In order to make the comparison against the optimized static algorithm, MTTF was set at 6 hours, while MTTR was set to 44 minutes for each site. This translates to a reliability of 0.874 in probabilistic terms which is the average p_i for the 7 sites in Table 2. Using a t_{recon} value of 30 secs, and setting n to 7, the availability is 0.98.

ρ_w	U(1,1)	U(1,5)	U(1,10)
0.05	12	34	73
0.1	24	75	141
0.15	36	108	199
0.2	48	139	256
0.3	72	213	422
0.4	96	282	502
0.5	120	364	675

Table 3: Dynamic Algorithm: Communications Cost versus ρ_w for 3 cost distributions (7 sites)
$\overline{P}$ = (0.91,0.90,0.89,0.87,0.86,0.85,0.84)
Total Traffic Volumes = (5,7,4,9,1,5,8)
A = 0.98

Several interesting conclusions may be drawn from the results in Tables 2 and 3. First, if cost minimization is the main objective, then the dynamic algorithm is superior when ρ_w is below a cut-off. However, this cut-off becomes smaller as the range of variation of c_{ij} increases. For instance, if the distribution chosen is U(1,1), then the cut-off is 0.15, while for distributions U(1,5) and U(1,10), the cut-off reduces to 0.10 and 0.05 respectively. This means that as the variation in c_{ij} increases, the dynamic algorithm becomes less attractive if cost minimization is the main objective.

On the other hand, if the cost of the dynamic algorithm is compared against its static counterpart which gives the same availability, then the above ρ_w cut-offs increase to 0.25, 0.2, and 0.1 respectively. Therefore, the ρ_w space within which the dynamic algorithm does better becomes larger. Clearly, ρ_w is a critical factor in choosing between the static and dynamic algorithms.

Secondly, while different static vote assignments lead to various values of availability and communications cost, in the dynamic case there is one availability value, 0.98. Table 2 shows that the static method can give a higher availability of 0.99. Therefore, if availability maximization is the main goal then the static technique seems to outperform the dynamic one. Of course, t_{recon} and n are critical parameters in computing the availability from the formula above, and if these are both decreased, then availability would naturally increase.

7. Conclusions

An optimization model was developed for the problem of assigning votes to sites so as to minimize the communications cost subject to a given availability constraint. This problem has exponential complexity and no efficient solution procedure that would run in reasonable time is known. (Complete enumeration is an obvious solution method,

though clearly not a feasible one). A semi-exhaustive algorithm to solve this problem was discussed in detail and also implemented. This algorithm employs an efficient technique for computing the availability of a vote assignment. The signature concept was used to prune the exponential space of vote combinations and to generate only non-equivalent combinations.

Finally, the optimized static algorithm was compared against the available copies method, a dynamic algorithm. It was found that no one type of algorithm uniformly dominates the other. Ranges over which each type of algorithm does better were determined. Dynamic algorithms were better for a small value of the write ratio, ρ_w, and low variability in the inter-site communications cost. On the other hand, static algorithms were better if the goal was to maximize availability. Also, if the goal was to minimize cost, then the range of ρ_w values over which the dynamic algorithm does better is very small. Finally, ease of implementation must also be considered, and on this account static algorithms are a big winner.

This study clearly shows that a more detailed analysis of static algorithms against dynamic algorithms would be a very useful exercise. In this paper, the treatment of dynamic algorithms has been restricted to just one type, the available copies method. Future work is anticipated to evaluate other dynamic algorithms, perhaps using more elaborate models. Further work is also needed to develop more efficient heuristic solution methods to the vote assignment problem.

References

[BARB86] Barabara, D., Garcia-Molina, H., and Spauster, A., "Protocols for Dynamic Vote Reassignment", Technical Report, Department of Computer Science, Princeton University, May 1986.

[BERN84] Bernstein, P.A., and Goodman, N., "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases", ACM Transactions on Database Systems 9(4), pp 596-615, December 1984.

[BERN87] Bernstein, P., Hadzilacos, V., and Goodman, N., Concurrency Control and Recovery in Database Systems, Addison Wesley Publishing Co., 1987.

[DAVI85] Davidson, S. B., Garcia-Molina, H., and Skeen, D., "Consistency in Partitioned Networks", ACM Computing Surveys 17(3), pp 341-370, September 1985.

[EAGE81] Eager, D.L., "Robust Concurrency Control in Distributed Databases", Technical Report CSRG #135, Computer Systems Research Group, University of Toronto, October 1981.

[EAGE83] Eager, D.L., and Sevcik, K.C., "Achieving Robustness in Distributed Database Systems", ACM Trans. Database Syst. 8(3), pp 354 - 381, September 1983.

[ELAB85] El Abbadi, A., Skeen, D., and Cristian, F., "An Efficient, Fault-Tolerant Protocol for Replicated Data Management", Proc. 4th ACM SIGACT-SIGMOD Symp, on Principles of Database Systems, pp 215 - 228, Portland, Oregon, March 1985.

[GARC84] Garcia-Molina, H., and Barbara, D., "Optimizing the Reliability Provided by Voting Mechanisms", Proc. 4th International Conference on Distributed Computing Systems, pp 340-346, May 1984.

[GARC85] Garcia-Molina, H., and Barbara, D., "How to Assign Votes in a Distributed System", Journal of ACM, Vol. 32, No. 4, pp 841-860, October 1985.

[GIFF79] Gifford, D.K., "Weighted Voting for Replicated Data", Proc. 7th ACM SIGOPS Symp. on operating Systems Principles, pp 150 - 159, Pacific Grove, CA, December 1979.

[HERL87] Herlihy, M., "Dynamic Quorum Adjustment for Partitional Data", ACM Trans. on Database Systems, Vol 12, No 2, pp 170-194, June 1987.

[JAJO87] Jajodia, S. and Mutchler, D., "Dynamic Voting", Proc. 1987 ACM SIGMOD, pp 227-238, San Francisco, CA, May 1987.

[KUMA88] Kumar, A., and Segev, A., "Optimizing Voting-Type Algorithms for Replicated Data", Lecture Notes in Computer Science, Vol 303, J.W. Schmidt, S. Ceri and M. Missekoff (eds.), pp 428-442, Springer-Verlag, March 1988.

[STON79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed Ingres", IEEE Transactions on Software Engineering 3(3), pp 188-194, May 1979.

[THOM79] Thomas, R. H., "A Majority Consensus Approach to Concurrency Control", ACM Trans. on Database Systems 4(2), pp 180-209, June 1979.

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
1 CYCLOTRON ROAD
BERKELEY, CALIFORNIA 94720