INCREASING AVAILABILITY UNDER MUTUAL EXCLUSION CONSTRAINTS WITH DYNAMIC VOTE REASSIGNMENT

Daniel Barbara
Hector Garcia-Molina
Annemarie Spauster

CS-TR-056-86

November 1986

Increasing Availability under Mutual Exclusion Constraints with Dynamic Vote Reassignment

Daniel Barbara

Universidad Simon Bolivar
Departamento de Matematicas y Ciencia de la Computacion
Caracas, Venezuela

Hector Garcia-Molina
Annemarie Spauster

Computer Science Department
Princeton University
Princeton, NJ 08540

ABSTRACT

Voting is used commonly to enforce mutual exclusion in distributed systems. Each node is assigned a number of votes and only the group with a majority of votes is allowed to perform a restricted operation. This paper describes techniques for dynamically reassigning votes upon node or link failure, in an attempt to make the system more resilient to future failures. We focus on autonomous methods for achieving this, i.e., methods that allow the nodes to make independent choices about changing their votes and picking new vote values, rather than group consensus techniques that require tight coordination among the remaining nodes. Protocols are given which allow nodes to install new vote values while still maintaining mutual exclusion requirements. The lemmas and theorems to validate the protocols are presented. A simple example shows how to apply the method to a database object-locking scheme; the protocols, however, are versatile and can be used for any application requiring mutual exclusion. In addition, policies are presented that allow nodes to autonomously select their new vote values. Simulation results are presented comparing the autonomous methods to static vote assignments and to group consensus strategies. These results demonstrate that under high failure rates, dynamic vote reassignment shows great improvement over a static assignment of votes in terms of availability. In addition, many autonomous methods for determining a new vote assignment yield almost as much availability as a group consensus method and at the same time are faster and more flexible.

November 3, 1986

Increasing Availability under Mutual Exclusion Constraints with Dynamic Vote Reassignment

Daniel Barbara

Universidad Simon Bolivar
Departamento de Matematicas y Ciencia de la Computacion
Caracas, Venezuela

Hector Garcia-Molina
Annemarie Spauster

Computer Science Department
Princeton University
Princeton, NJ 08540

1. INTRODUCTION

In distributed systems, the mutual exclusion problem is solved by ensuring that two or more nodes or subsets of the nodes are not performing a restricted operation concurrently. Some applications that require a mutual exclusion mechanism are, for instance, database transaction commit [Gray78], coordinator election [Garc82a] and n-modular redundant computation. Another use of mutual exclusion occurs in replicated data management, where it can be used to provide consistency among copies[Bern81,Davi85].

Aside from ensuring mutual exclusion for the application at hand, a technique for implementing mutual exclusion may have other critical objectives. For example, it may be important that it work in a partitioned network. At the same time, it is generally desirable for the method to provide high system availability. In other words, the system should be able to perform the restricted operations as much of the time as possible. For instance, when applied to replicated data management, it is during partitioning that mutual exclusion is crucial: if two noncommunicating groups of nodes both perform updates on the database, the data copies will diverge. Not desirable, however, is a method that too often leaves all nodes unable to perform

This work has been supported by NSF Grants DMC-8351616 and DMC-8505194, New Jersey Governor's Commission on Science and Technology Contract 85-990660-6, and grants from DEC, IBM, NCR, and Concurrent Computer corporations.

updates. Ideally, we would like a mutual exclusion mechanism that achieves our secondary goals as well.

1.1 Solution

Voting is one well-studied technique for providing mutual exclusion in a distributed system. [Giff79,Thom79,Barb84] In its simplest form, each node is assigned a number of votes and only a node or group of nodes that can collect a majority of votes is allowed to perform the restricted operation. When the vote assignment is kept static, it is not difficult to see that mutual exclusion can be guaranteed. Further, voting is resilient to partitions. As an example of this method, consider the four node system illustrated in Figure 1. (We ignore the connectivity of the network; for the purposes of the example it is irrelevant). In this system, each node has been assigned one vote, except for node *d* which has received 2 votes. Now assume the system is partitioned into two groups, one with nodes $\{a, b\}$ and the second with nodes $\{c, d\}$. The nodes in the second group have a majority of votes (3 out of 5) and are allowed to perform a restricted operation, since they are assured that no other group is concurrently in the same situation.

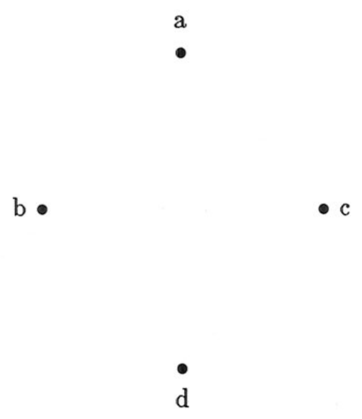


Figure 1

Notice that certain partitions can make it impossible for any group to perform the restricted operation. In our example, if three groups are formed, $\{a, b\}$, $\{c\}$, and $\{d\}$, then no group has a majority. In this case we say the system has *halted*. The network does not necessarily have to partition to produce a halted state. If nodes with a majority of votes fail and the available nodes cannot accurately detect this, the system is rendered inoperative since it is impossible to collect votes from the nodes that are down.

Stated simply, under voting work cannot be done if a majority of votes cannot be collected. To achieve higher availability without compromising mutual exclusion, we need to minimize the likelihood of these undesirable halted states. To accomplish this, we can initially assign the votes intelligently (e.g., give the nodes that are more reliable or better interconnected more votes), as suggested in [Barb84]. For a simple instance, in our example, it was more sensible to assign one of the nodes two votes instead of giving all four nodes just one vote. (With just a total of four votes in the system, at least three nodes are needed to form a majority. If instead one node has two votes then two nodes form a majority if one of them is the node with the extra vote. In addition, it is still true that any three nodes form a majority. It is in general always better for the total number of votes in the system to be odd rather than even.) In addition to a good initial assignment, we propose to *dynamically reassign* the votes in the presence of failures in order to make the current majority group more resilient to future failures. As an example of this technique, consider once again the system of Figure 1, with its vote assignment: $v_a = v_b = v_c = 1$ and $v_d = 2$, where v_i represents the number of votes of node i. Assume that a partition separates node d from nodes a, b and c. Nodes a, b and c can still collect a majority of votes while d cannot. However, if a second partition occurs, separating node c from a and b, the system will be halted; no transactions may be processed. However, we can reduce the likelihood of halting if we increase the votes of group $\{a, b, c\}$ *before* the second partition occurs. That is, after any failure, the majority group (if any) dynamically reassigns the votes in order to increase its voting power and increase the system's chances of surviving subsequent failures.

In our example, nodes a, b and c may opt for reconfiguring the votes during the first partition. For instance, a new vote assignment could be $v_a = v_b = v_c = 5$. Node d is unaware of the change and remains with $v_d = 2$ votes. (As a matter of fact, since d is not in the majority group, it *cannot* change its votes.) In this way, the second partition will find nodes a and b with 10 votes out of 17, forming a majority group and the system will not be halted. After the second partition, the new majority group of $\{a, b\}$ could reassign itself new votes of $v_a = 15$ and $v_b = 5$ in order to tolerate even a third partition. We refer to this method as ***dynamic vote reassignment.***

1.2 Basic Methodologies

We categorize dynamic vote reassignment by two general methods.

- **Group Consensus**: The nodes in the active (majority) group agree upon the new vote assignment using either a distributed algorithm or by electing a coordinator to perform the task. Nodes outside the majority group do not receive any votes.
- **Autonomous Reassignment**: Each node makes it own decision about changing its votes and picking a new vote value, without regarding the rest of the nodes. Before the change is made final, though, the node must collect a majority of votes.

The tradeoff between these two is accuracy versus time and information requirements. Group consensus relies on the nodes collecting information about the current system topology and using it to decide on a new global vote assignment. Because the new assignment reflects the state of the system, it is an "intelligent" one that is more resilient to future failures. Coming up with this new assignment, though, requires a complicated algorithm. In addition, the topological information must be accurate to achieve a good assignment. Autonomous reassignment, on the other hand, simply requires that each node decide independently when it is the right time to change its own votes and what its new vote value should be. The node then attempts to get the recognition of the majority in much the same way it would for any other event requiring a majority. Each node uses its own view of the system to decide on its new vote value, with its primary goal being to claim for itself all or part of the voting power of a node (or nodes) that have been separated from the majority group. Because the view at each node may be different and because each node operates independently, the global vote assignment may not be as good as under group consensus. However, this method is quicker, simpler and more flexible.

Implementing either form of dynamic vote reassignment requires solving two main problems. The first is one of *policy*. The policy is the mechanism for selecting a new vote value. After a new value is picked, the node must "install" it, making sure it is in a majority group that is allowed to change its votes. We refer to this algorithm as the *protocol*. Within the protocol, we must ensure that nodes with a majority of the "old" votes are informed of each change. When deciding if a majority exists, the algorithm must not get confused between the "old" and the "new" votes. We will see that it is the protocol that ensures mutual exclusion, the chosen policy is an attempt to get as good a new assignment as possible.

To clarify this distinction, we refer again to our previous example. During the first partition, nodes *a*, *b* and *c* decide to take on 4 more votes, increasing their vote values to 5. This is a policy decision: the nodes take on twice the number of votes of the disconnected node. Node *d* may in fact also implement the policy and decide on a good new vote value for itself. This is where the protocol is important. Only nodes *a*, *b* and *c* should be able to install the change; node *d*'s attempt should fail.

In this paper we concentrate on the autonomous method for dynamic vote reassignment. The group consensus method does not require new study in great detail. Techniques for determining a vote assignment given a topology have been studied [Barb84] as have algorithms for achieving agreement [Garc82a,Garc82b]. We return to the group consensus approach in Section 5 where we present simulation results comparing autonomous methods to group consensus and to a static assignment of votes. These results demonstrate that autonomous methods do yield almost as much availability as the group consensus approach. In light of this and the speed and simplicity which autonomous methods offer, we believe autonomous vote reassignment is a viable alternative to group consensus which deserves the further attention we give it here.

1.3 Assumptions and Other Work

An important feature of our method is that it makes no assumptions about how quickly or accurately nodes must detect failures or partitions. We assume each node has its own view of the state of the network (this view indicates which nodes are up and which are down). Among the nodes, however, these views may be inconsistent. In other words, we allow the possibility that a node (or nodes) is incorrect about the state of a link or other node. With our protocols, such inconsistencies may lead to suboptimal assignments, but at no point is mutual exclusion compromised. In addition, we allow for lost messages and for out-of-order messages. We do not, however, tolerate Byzantine failures, i.e., nodes cannot be insane, all failures are fail-stop. It is perhaps unreasonable to expect any mutual exclusion mechanism to tolerate such behavior, since an insane node can simply violate mutual exclusion on its own. By the same token, we assume that messages that do arrive at their destinations are intact, not garbled. We also assume that each node has available some non-volatile storage.

Two recent papers [Davc85, Abba86] also address increasing availability under mutual exclusion requirements. Their techniques attempt to change the required majority needed to perform restricted operations when the state of the network changes, instead of changing the actual votes assigned to the nodes. In [Davc85], the authors rely on accurate views of the current network state. This in essence means that when a failure or recovery of a node or link occurs, every node recognizes it instantaneously. In [Abba86], the algorithm relies on coordinating a consistent view among the nodes. Also, in [Abba86] the algorithm is specific to the application of improving replicated database reliability. In contrast, our protocols provide a flexible method for increasing system availability in any application requiring mutual exclusion, without requiring the system to have special features that coordinate status information continually.

In this paper, we examine in detail dynamic vote reassignment and its effect on availability, focusing on the autonomous method. The next section is devoted to two sets of protocols and proofs of correctness. In Section 3, we demonstrate the applicability of autonomous reassignment to a simple replicated database object-locking scheme which shows the ease with which the method can be used for any problem requiring mutual exclusion. In Section 4, we present various policies to be used in conjunction with the protocols. In addition, in Section 5, we discuss simulation results that demonstrate the increased availability provided by dynamic vote reassignment over a static assignment. We also compare the performance of autonomous methods to a group consensus technique. We end in Section 6 with some conclusions.

2. THE PROTOCOLS

The protocols for autonomous vote reassignment are what guarantee mutual exclusion. Once a node picks a new vote value, a ***vote changing*** protocol is invoked to install the change. The vote changing protocol uses the ***vote collecting*** protocol to ensure that enough votes have been collected to validate the change. In addition, the vote collecting protocol is used for all other events requiring majority approval.

Here we present two sets of vote changing and vote collecting protocols, corresponding to two different scenarios. Under Scenario One, nodes are only allowed to increase their votes. Under Scenario Two, nodes can autonomously increase or decrease their votes. The implications of these two scenarios will be evident when we discuss policies in Section 4. For each scenario, we include a proof of correctness.

2.1 Scenario One

For Scenario One there are two protocols, one for vote collecting and one for vote increasing. To utilize them, we must establish (in stable storage) at each node i the vector V_i where $V_i[j]$ indicates the number of votes of node j according to node i. This vector represents what node i believes the current global vote assignment to be. We use the notation v_k to indicate the votes of a node k as determined upon vote collecting.

Protocol P1. Vote Collecting

Assume node i is collecting votes to decide upon an event. Each node j that can communicate with i will send its voting vector. Let G be the set of nodes from which i has received votes (including i itself). Node i decides upon the votes of node k (v_k) using the following rules:

a) If i received the vector V_k from k, then $v_k = V_k[k]$. Also, if $V_k[k] > V_i[k]$, then i should modify its entry $V_i[k]$ to be equal to $V_k[k]$. (The reason for this last step will be clear later.)

b) If i does not receive V_k within a certain time period (perhaps k cannot communicate with i), then $v_k = \max_{j \in G}(V_j[k])$. In this way, k is considered to have the largest number of votes recorded among the nodes that have voted. In addition, i modifies its entry $V_i[k]$ to be equal to v_k. This way, the largest value of v_k gets propagated.

Using the v_k values, node i can determine if it has a majority of votes. That is, the total number of votes will be computed by i as

$$TOT = \sum_{all\ k} v_k$$

and the votes received will be

$$\sum_{j \in G} v_j$$

If this last sum represents a majority in TOT, then node i has a majority. □

We now describe the protocol for vote increasing. Essentially, it is nothing more than a commit protocol to change the number of votes that a node has. It must ensure that a group of nodes with a majority of votes agree with this change and record it. In fact, any commit protocol that uses two or three phases will serve this purpose. However, there are two facts that allow us to simplify this protocol:

- No negative acknowledgments will be produced. That is, no node is to vote against the increase of votes of another node.
- Assume that a node increasing its votes becomes separated from the participants on the protocol, and these participants registered the change before it committed. This situation does not become dangerous, since the worst that can happen is the participants consider themselves to not have a majority in a future event.

These two observations allow us to reduce the protocol to a one phase protocol in which a node indicates to the rest of the sites its intention of increasing its votes and waits for the acknowledgments. The initiator only makes the change effective when it receives acknowledgments from nodes with a majority of votes, but the rest of the nodes record the change immediately. The protocol is as follows:

Protocol P2. Vote increasing.

The initiator (node i)

a) Send the change to the rest of the nodes with which node i can communicate.

b) Wait for a majority of acknowledgments to arrive (whether or not a majority of votes has been received by node i is determined by following protocol *P1*), and then make the change permanent in the local voting vector, that is update $V_i[i]$.

The participants

Upon receiving the change, register it in the local voting vector (update $V_j[i]$) and send acknowledgment to the initiator in the manner described in Protocol *P1*. □

The algorithm for the initiator can be optimized for the case in which i is not connected to a group of nodes with a majority of votes. By timing out the responses, the node may cancel the vote increase if it does not receive enough votes after a certain time period. Note that this is not essential, since the node will not make the change permanent in its local vector until enough votes are received and therefore it will keep voting with its old number of votes. (For simplicity, we will not explicitly deal with cancelled increases in our proofs. However, the proofs are still valid since a cancelled increase logically has no effect. Successful increases will continue to be valid in spite of the cancellation.)

To prove these protocols correct, a first approach might be to totally order the vote increments and show that each increment occurs fully aware of all those that preceded it. This would simply guarantee that each time a majority of votes is collected, the node collecting votes is aware of the latest vote value at every other node. Protocols *P1* and *P2*, however, allow two or more vote increases to occur concurrently. Since running the protocol can be affected by such things as network transmission delays and load factors, we cannot guarantee that two vote increases, one at node a initiated before one at node b, will finish in the same order as they are started. We cannot even pick the one that starts first (or the one that ends first) and claim that all those that start (end) later are aware of the first.

Node	votes on	votes on	votes on	votes on
(x) 1	I_1 at z	I_2 at x		
2	I_1 at z	I_2 at x		
3	I_1 at z	I_2 at x		
(y) 4		I_2 at x	I_3 at y	
5		I_2 at x	I_3 at y	
6		I_2 at x	I_3 at y	
7			I_3 at y	I_1 at z
8			I_3 at y	I_1 at z
(z) 9			I_3 at y	I_1 at z

Time --->

Figure 2

To illustrate this situation consider the following example of the vote increasing process. Assume we have a nine node system where each node initially has 1 vote. (We leave aside the relevant connectivity.) Distinguish nodes 1, 4 and 9 as x, y and z, respectively. Say node z initiates vote increment I_1 to 3 votes and nodes 1, 2 and 3 record the vote change in their vote vectors and vote on it (by sending votes to z) with 1 vote each. Then, node x initiates a vote increment I_2 to 3 votes and receives votes from nodes 2, 3, 4, 5, 6 and itself, each with 1 vote. Note that I_2 is aware of I_1 since nodes 1, 2 and 3 each passed their vote vectors and node x incorporated this information in accordance with the protocols. (We will use the notation I_1 --> I_2 for this occurrence and say I_2 *sees* I_1. We will give a more precise definition of "-->" later.) Now, node x assumes z has 3 votes. Node x determines it has 6 votes out of 11, a majority, and I_2 is complete. Meanwhile, I_1 is still pending. Now node y initiates increment I_3 and receives votes from 5, 6, 7, 8, 9 and its own, at 1 vote each. So, I_2 --> I_3 via nodes 4, 5, 6; y assumes x has 3 votes and determines it has a majority of 6 votes out of 11. I_3 is completed.

Finally, nodes 7 and 8 vote on *z*'s increment I_1. I_3 --> I_1 via nodes 7, 8 and 9, so *z* assumes *y* has 3 votes and determines it needs 6 votes out of 11, which it gets, and I_1 is approved. (See Figures 2 and 3.) Note that for this example, even though I_1 finished last, it wasn't aware of I_2. In addition, I_3 did not see I_1 and I_2 did not see I_3. It is thus not possible to totally order these vote increments. A total ordering could be guaranteed by protocols with higher overhead (such as a group consensus technique) but is unnecessary. In fact, Protocols *P1* and *P2* allow for a vote increment to occur even though the node's view of the vote assignment may be incorrect. We will show, however, that this is not dangerous. We can prove that vote increments occur with ***enough*** knowledge of each other so that any increment that is approved is safe, safe in terms of guaranteeing mutual exclusion.

node	view	votes rec'd
1	1	1
2	1	1
3	1	1
4	1	1
5	1	1
6	1	1
7	1	--
8	1	--
9	3	--

(a)

node	view	votes rec'd
1	3	--
2	1	--
3	1	--
4	1	1
5	1	1
6	1	1
7	1	1
8	1	1
9	1	1

(b)

node	view	votes rec'd
1	1	1
2	1	1
3	1	1
4	3	--
5	1	--
6	1	--
7	1	1
8	1	1
9	1	1

(c)

Figure 3: View and votes received when vote change completes
(a) at *x* when I_2 completes;
(b) at *y* when I_3 completes;
(c) at *z* when I_1 completes.

To proceed with the proofs, we will order the vote increments according to the global time at which they are initiated. The timing of the increments is actually irrelevant and serves solely as a notational convenience. For two increments that are initiated at the same time, one is chosen arbitrarily to precede the other. We refer, then, to vote increment *j* as I_j, where *j* is a positive integer. Increment I_j occurs at node $node(I_j)$ and represents a change of votes to new vote value $v(I_j)$. The node initiating the successful increase I_j does so by collecting a majority of votes from a group of nodes which we call $MAJG_{I_j}$. This is group G in Protocol *P1*. The sum of the votes obtained from $MAJG_{I_j}$ is called $MAJV_{I_j}$. In addition, we often refer to the nodes that were not in the voting group and call them $ming_{I_j}$ with vote total $minv_{I_j}$. The total $minv_{I_j}$ is determined as in step (b) of Protocol *P1*. It does not necessarily accurately reflect the vote values of nodes in $ming_{I_j}$. Note that for any such increment I_j, $MAJG_{I_j} \cap ming_{I_j} = \emptyset$ and $MAJV_{I_j} > minv_{I_j}$ as prescribed by Protocol *P1*.

In addition, for an increment I_j we distinguish between the previous vote increments that I_j uses as votes to collect a majority and those I_j is aware of but does not necessarily use. Say I_k changes the votes of node $node(I_k) = z$ to $v(I_k)$. When I_j collects votes using Protocol $P1$, it determines v_z, the number of votes to use for node z:

if $v_z \geq v(I_k)$ then I_k --> I_j and we say I_j *sees* I_k,

if $v_z = v(I_k)$ and $z \in MAJG_{I_j}$ then I_k => I_j and we say I_j *uses* I_k.

Note that I_k => I_j implies I_k --> I_j. Also, notice that for I_k => I_j it is not necessary that $k < j$. If I_j was initiated before I_k but I_k finished first and then voted on I_j, then I_k => I_j but $k > j$. By Protocol *P2*, however, I_k must have completed before it was used for I_j. Also, since votes only increase, the latest vote value at a node implicitly represents all the previous increments at that node. Consider two increments I_j and I_k such that $node(I_j) = node(I_k)$ and I_j and I_k occur one right after the other at that node. Then, I_j => I_k and I_k --> I_m (I_m at any node) implies I_j --> I_m.

We establish a graphical representation for the set of vote increases occurring in the system. As an example, let I_0 signify the original assignment. If I_1 uses only the original assignment, then I_0 => I_1. Also, I_0 --> I_1, but for convenience we omit these redundant arcs. If I_1 is subsequently used for I_2, along with some original votes, then this is depicted as in Figure 4(a). Also, I_1 and I_2 may be used for I_3 as in Figure 4(b). We also indicate in Figure 4(b) that I_3 has

$$\begin{tikzcd}I_0 \arrow[r, Rightarrow] \arrow[rr, Rightarrow, bend right] & I_1 \arrow[r, Rightarrow] & I_2\end{tikzcd}$$

(a)

$$\begin{tikzcd}I_0 \arrow[r, Rightarrow] \arrow[rr, Rightarrow, bend right] \arrow[rrr, bend left] & I_1 \arrow[r, Rightarrow] \arrow[rr, Rightarrow, bend right] & I_2 \arrow[r, Rightarrow] & I_3\end{tikzcd}$$

(b)

Figure 4

seen the original assignment, that is I_0 --> I_3. Any set of increments can be represented by such a directed graph with source I_0 and two types of arcs. It is sometimes convenient to just consider the "uses" (=>) arcs and say they constitute a path to a vote increment I_k if there is a path from I_0 to I_k consisting of "uses" arcs. We define the length of such a path to be the number of increments on it (excluding I_k), which is simply the number of "uses" arcs.

With this machinery we can prove that Protocols *P1* and *P2* have a property that is sufficient to guarantee mutual exclusion. We have already stated that we cannot simply say I_1 --> I_2 --> I_3 --> ... where I_1 occurs before I_2, I_2 before I_3, etc. We can prove, however, that I_j --> I_k or I_k --> I_j for all pairs of vote increments I_j and I_k. (In the example of Figures 2 and 3, I_1 --> I_2, I_2 --> I_3 and I_3 --> I_1.) Intuitively, we are stating that between any pair of vote increases at least one knows of the other. For an application requiring mutual exclusion this property extends to any vote collecting event. In this way, no node will perform an action that conflicts with another since for two conflicting actions at least one node will know of the conflict.

We now present the lemmas and theorems concerning Scenario One. First we prove an important property of the directed graphs.

Lemma 2.1.1: If we exclude the "sees" (-->) arcs between increments, the resulting graph (the directed graph representing the set of paths leading to an increment) is acyclic.

Proof: The proof is by contradiction. First note that I_k => I_x implies that I_k finished collecting votes using Protocol *P2* before I_x collected a majority using Protocol *P2*. Hence, I_k finished executing *P2* before I_x finished executing *P2*. Now, suppose there exists a cycle in the graph, I_k => I_1 => ... => I_m => I_k. This implies that I_k finished before I_1,..., before I_m, before I_k, an impossibility. □

Next we prove three properties of the vote increments.

Lemma 2.1.2: If I_x --> I_j and I_j => ...=>...=>I_k, then I_x --> I_k.

Proof: The proof is obvious by the properties of Protocol *P1*. If I_x --> I_j, then further uses of I_j are given the I_x information, in particular, I_k receives information about I_x. In effect, I_x --> I_k. □

Lemma 2.1.3: If two vote increments I_j and I_k occur such that $MAJG_{I_j} \cap MAJG_{I_k} \neq \varnothing$, then either I_j --> I_k or I_k --> I_j.

Proof: The lemma is obvious by the properties of Protocol *P1*. Say x is the node $MAJG_{I_j}$ and $MAJG_{I_k}$ have in common. If x votes on I_j first then I_j --> I_k via node x. Similarly, if x votes on I_k first then I_k --> I_j via node x. Node x must do one or the other first, so the lemma is true. □

Lemma 2.1.4: If two increments I_j and I_k occur such that I_j has seen the vote increments I_k uses to collect a majority and I_k has seen the vote increments I_j uses to collect a majority, then $MAJG_{I_j} \cap MAJG_{I_k} \neq \varnothing$.

Proof: Say $MAJG_{I_j} \cap MAJG_{I_k} = \varnothing$. Then, $MAJG_{I_j} \subseteq ming_{I_k}$ and $MAJG_{I_k} \subseteq ming_{I_j}$. For a node $x \in ming_{I_j}$ and $x \in MAJG_{I_k}$, the vote value for node x that I_j sees is greater than or equal to the value that I_k uses, since I_j has seen what I_k uses and votes only increase. This implies $minv_{I_j} \geq MAJV_{I_k}$. Certainly $MAJV_{I_j} > minv_{I_j}$ for I_j to have occurred, so $MAJV_{I_j} > minv_{I_j} \geq MAJV_{I_k}$. We can argue similarly for a node y, $y \in ming_{I_k}$ and $y \in MAJG_{I_j}$, yielding $MAJV_{I_k} > minv_{I_k} \geq MAJV_{I_j}$. But now we have reached a contradiction, so $MAJG_{I_j} \cap MAJG_{I_k} \neq \varnothing$. □

Theorem 2.1.1: For all pairs of vote increments, I_j, I_k, either I_j --> I_k or I_k --> I_j (or both).

Proof: The proof uses a double induction.

Induction One is on the number of vote increments, i:

Basis: 1 vote increment.

Obviously, I_1 sees its own increment, so I_1 --> I_1.

Inductive Hypothesis One.

Assume the theorem is true for a set of i increments.

Show true for a set of i+1 vote increments.

There are three steps.

1). If we consider the directed graph formed by the i+1 increments, it is acyclic by Lemma 2.1.1. Therefore, it has one vertex with no outgoing edges (by the properties of directed acyclic graphs), corresponding to one increment that has not been used by any of the other i increments. Call one such unused increment I_f. We first prove the following fact.

Fact: If a vote increase, I_f, of δ votes occurs at node $node(I_f)$, but no other node uses I_f to increase its own votes, then we can remove I_f from the set of increments without preventing any of the remaining vote increases.

Proof: Consider another vote increase I_k that doesn't use I_f. If I_k doesn't see I_f either, then I_k has no knowledge of I_f and acts as though I_f never occurred, so the lemma is true immediately. If I_f --> I_k, then there are two cases. *Case 1:* $node(I_f) \in ming_{I_k}$. Certainly $MAJV_{I_k} > minv_{I_k}$ for I_k to increase its votes. If we remove increment I_f then $minv_{I_k}$ is decreased by δ, $minv_{I_k}{}' = minv_{I_k} - \delta$ but still $MAJV_{I_k} > minv_{I_k}{}'$ so still I_k occurs. *Case 2:* $node(I_f) \in MAJG_{I_k}$. Then, I_f had not been installed at $node(I_f)$ when $node(I_f)$ voted on I_k. $node(I_f)$ voted with an older vote value; $v_{n(I_f)}$ at $node(I_k)$ is computed to be the older value. If I_f is removed, $v_{n(I_f)}$ does not change, so neither does the vote total at $node(I_k)$. □

With this fact, we can remove I_f from the set of $i+1$ vote increments without preventing any of the other vote increases. By Inductive Hypothesis One, for all the remaining i increments the theorem is true. It remains then to show that either I_m --> I_f or I_f --> I_m where I_m is any one of the remaining i increments.

2). Order the remaining i increments by the length of the longest path from the original vote assignment to the increment.

3). Show that for I_m with a longest path of length p, either I_m --> I_f or I_f --> I_m by *Induction Two* on the path length, p.

Basis: maximum path length = 1.

Let I_m be an increment with maximum path length 1. Then, I_m uses only the original assignment (I_0 => I_m). By Inductive Hypothesis One, for any vote increase I_{f^*} on a path to I_f, either I_{f^*} --> I_m or I_m --> I_{f^*}. If I_m --> I_{f^*} for some I_{f^*}, then I_m --> I_f by Lemma 2.1.2 and we are done. Else, I_{f^*} --> I_m for all such I_{f^*}. Certainly I_f has seen the original assignment. Thus, both I_m and I_f have seen the increments that the other is using, so the basis is true by Lemmas 2.1.3 and 2.1.4.

Inductive Hypothesis Two.

Assume true for maximum path length p.

Show true for maximum path length p+1.

Let I_m be an increment with a maximum path length $p+1$. Once again we know by Inductive Hypothesis One that for any vote increase I_{f^*} on a path to I_f, either I_{f^*} --> I_m or I_m --> I_{f^*}. We know by Inductive Hypothesis Two that for all I_{m^*} on this path to I_m, either I_{m^*} --> I_f or I_f --> I_{m^*}. Once again, if for any I_{f^*} on the paths to I_f, I_m --> I_{f^*} then I_m --> I_f by Lemma 2.1.2 and we are done. Similarly, if for one of I_{m^*}, I_f --> I_{m^*} then I_f --> I_m by Lemma 2.1.2

and we are done. Otherwise, I_{m^*} --> I_f for each such I_{m^*} and I_{f^*} --> I_m for each such I_{f^*}. But here again, both I_f and I_m have seen the increments that the other is using. So, the proposition of step 3 is true by Lemmas 2.1.3 and 2.1.4.

Since the proposition is true for I_m of any path length, the theorem is true. □

By Theorem 2.1.1, we have a "weak" property for dynamic vote reassignment using Protocols *P1* and *P2*. Immediately, we have the following corollary:

Corollary 2.1.1: Let t be the time when increment I_j receives a majority of votes. Then any increment I_k that initiates Protocol *P2* after time t will see I_j, i.e., I_j --> I_k. □

By Corollary 2.1.1, once an increment I_j "commits", we can be sure that all other increments that completely follow it will see I_j. However, we cannot say anything like this about concurrent increments. It is these concurrent increments that prevent us from applying a total ordering to the vote increments. We show in Section 3 that when the protocols are used for an application the weak property is sufficient and we can allow these concurrent vote increasing events without compromising mutual exclusion.

Finally, there is one more issue we need to address concerning Scenario One and Protocols *P1* and *P2*. Some dynamic vote reassignment protocols can run into a type of deadlock anomaly. It occurs when two nodes in the majority group concurrently attempt to increase their votes, but are unable to get a majority of confirming votes. As an example, we return to Figure 1 and the original vote assignment

$$V_a[a] = 1, V_b[b] = 1, V_c[c] = 1, V_d[d] = 2$$

Assume that nodes a and b are trying to increase their votes from 1 to 5. Both nodes may have communicated their intentions to the rest of the nodes, but they may be waiting for acknowledgments from nodes with a majority of votes to make the change permanent. At this point, each node has a view of the votes in the system that looks like this:

For node a,

$$V_a[a] = 1, V_a[b] = 5, V_a[c] = 1, V_a[d] = 2$$

For node b,

$$V_b[a] = 5, V_b[b] = 1, V_b[c] = 1, V_b[d] = 2$$

In this situation, both nodes determine that there are a total of 9 votes throughout the system. Both will proceed to acknowledge the vote increment of the other, sending with its acknowledgment its current number of votes. For instance, a may send an acknowledgment to b with 1 vote. When b counts the number of votes received in the acknowledgments, it will have received 3 votes (one from a, b and c) out of a total of 9 and not be able to proceed. The same situation may occur meanwhile at a. If the counting of votes is not done carefully, both nodes will be precluded from changing votes even though they have the potential to do so.

The algorithms we have developed handle the counting of votes received in order to avoid this "deadlock" anomaly. By rule (b) of Protocol *P2* a node does not make a vote change permanent until it has been acknowledged with a majority of votes. In our example, both a and b will send their vote values under the current (pre-change) vote assignment. Furthermore, by rule (a) of Protocol *P1*, node a will use $v_b = V_b[b]$ and node b will use $v_a = V_a[a]$ as the vote values for b and a, respectively. The majority will be calculated using the old assignment and both nodes will determine that they have a majority and can increase their votes.

It is not hard to see that the deadlock anomaly can be avoided for any number of concurrently executing vote increases. More important, this feature of the vote collecting protocol benefits the application which uses autonomous vote reassignment. When a node is collecting votes to approve any action that requires mutual exclusion, concurrently executing vote increases will not prevent it from collecting a majority of votes.

2.2 Scenario Two

We now consider Scenario Two, where votes can increase or decrease. As expected, including the capability to decrease votes adds complexity to the protocols. Under Scenario One, when a node y collected votes, if a node x did not vote, y could determine an appropriate vote value for x by looking at the largest value for x among the voting nodes. This largest value was always the most recent among the group. Now that votes can decrease, this is no longer true. Instead, we must include new information with vector V_j to represent the age of a vote value. In addition, the nodes must do more work to guarantee that vote changes are propagated properly through the system. The new age information and vector V_j must be passed along with requests for votes to install a vote increase and with indication of a vote decrease. We will see, though, that unlike vote increasing, decreasing votes does not require approval by a group of nodes with a majority of votes. Instead, a node decreasing its votes simply informs other nodes of its decision. This should seem intuitive, since a decrease in votes means a node is giving up power and thus is not endangering mutual exclusion.

Protocol P3. Vote collecting.

Assume node i is collecting votes to decide upon an event. In this case, each voting node j will send i two vectors, the voting vector V_j and a version vector N_j. The entries $N_j[k]$ represent the version number for the value $V_j[k]$.

a) If i received V_k, then $v_k = V_k[k]$. Also, change $V_i[k]$ to $V_k[k]$ and $N_i[k]$ to $N_k[k]$ if either of the following two conditions apply:

$V_k[k] > V_i[k]$ or

$V_k[k] < V_i[k]$ and $N_k[k] > N_i[k]$.

The first condition is simply that of Scenario One. $V_k[k] > V_i[k]$ indicates that k has increased its votes since i last determined $V_i[k]$. The version number is irrelevant in this case since it provides no additional information. In the second case, $V_k[k] < V_i[k]$ indicates that either k has decreased its votes or an increase at k has not yet been approved or has been timed out. (Node i has been informed of the increase.) If an increase at k has not been approved, k must send $V_k[k] < V_i[k]$ (see Protocols *P4* and *P5*), but i should not alter $V_i[k]$ since k's increase may be approved at a future time. If, however, $N_k[k] > N_i[k]$ then $V_k[k]$ reflects a later decrease of votes at k or a failed vote increase attempt and this new information should be recorded.

b) If i does not receive V_k, then $v_k = V_j[k]$ for j such that $N_j[k] = \max_{j \in G}(N_j[k])$. That is, i assumes the newest value among the voting group for the vote value of node k. In addition, i modifies its entry $V_i[k]$ to equal v_k and $N_i[k]$ to equal $N_j[k]$. □

Protocol P4. Vote increasing.

The same as *P2*, except that:

- The initiator sends V_i and N_i along with its vote increase. Upon successfully collecting a majority of votes, the initiator increases $N_i[i]$ by one.
- The participants register $N_j[i]$ as $N_i[i]$ plus one and update their vectors V_j and N_j as necessary. □

Protocol P5. Vote decreasing.

The initiator (node i)

Update its own entry, $V_i[i]$, to the new value. Add one to $N_i[i]$. Send the vectors V_i, N_i to the participants.

The participants

Upon receiving the vectors, update the resident vectors V_j and N_j as necessary. □

Before presenting the proofs for Scenario Two, we define some terms and make some changes in notation. We refer to a vote increase as I_j, a decrease as D_j and a vote change (increase or decrease) simply as θ_j. Once again, for an increment I_j we distinguish between the previous operations that I_j uses to collect a majority and those I_j is aware of but does not necessarily use. Say θ_k changes the votes of node $node(\theta_k) = z$ to $v(\theta_k)$ with version number $n(\theta_k)$. When I_k collects votes using Protocol *P3*, it determines v_z, the number of votes to use for node z and n_z the version number for that vote:

if $n_z \geq n(\theta_k)$ then $\theta_k \dashrightarrow I_j$ and we say I_j *sees* θ_k,

if $n_z = n(\theta_k)$ and $z \in MAJG_{I_j}$ then $\theta_k \Rightarrow I_j$ and we say I_j *uses* θ_k.

The only difference with Scenario One is that if θ_k --> I_j, I_j may see a vote value larger *or smaller* than that installed by θ_k. In the previous scenario, the value seen by I_j would always be larger or equal.

Vote decrements do not see or use other operations in the same sense that vote increments do. (This is simply because decrements do not require vote collecting.) As a matter of fact, the only relationship we have to keep track of for vote decrements is immediate precedence. Let D_j be a decrement that installed version number $n(D_j)$ at $node(D_j)$. Operation θ_k immediately precedes D_j, $\theta_k \sim\sim> D_j$, if θ_k installed version $n(\theta_k) = n(D_j) - 1$ at $node(D_j)$.

In our proofs it will be convenient to group together each increment I_j with all the decrements at $node(I_j)$ that it precedes and that occur before the next increment at $node(I_j)$. If $I_j \sim\sim> D_1 \sim\sim> \ldots \sim\sim> D_m$ (and there is no D_{m+1} such that $D_m \sim\sim> D_{m+1}$), we call $VS_j = \{I_j, D_1, \ldots, D_m\}$ the vote set of I_j. We also extend our "uses" relationship to vote sets. We say increment I_k uses VS_j if I_k uses any of I_j, D_1, ... D_m. Similarly, we say that VS_k uses VS_j if I_k, the increment in VS_k, uses VS_j.

It will also be convenient to refer to all the operations that occur at a given node after some specific vote change θ_j takes place. We will represent this set by $post(\theta_j)$. Note that all operations in $post(\theta_j)$ occur at $node(\theta_j)$ and have version numbers greater than $n(\theta_j)$.

In our directed graph representation for vote changes, we represent vote sets as ovals enclosing the operations involved. "Uses" relationships involving vote sets are represented by "uses" arrows (=>) to or from the ovals. Within ovals, the $\sim\sim>$ "immediately precedes" notation ($\sim\sim>$) applies. In addition, "sees" arcs (-->) can occur between increments as before. As mentioned earlier, the only relationship relevant to vote decrements is immediate precedence, so we do not include "sees" arcs from increments to decrements; however, we do include "sees" arcs from decrements to increments. So, for example, if the initial assignment is used for I_1 and then $node(I_1)$ decreases its votes just twice this is depicted as in Figure 5. Here we have five nodes, I_0, I_1, D_1, D_2 and I_2, and three vote sets, VS_0, VS_1 and VS_2. One of I_1, D_1 or D_2 in VS_1 is being used for another vote increase, I_2. Finally, we still define the length of a path to an increment to be the number of increments on it. That is, a path consists of "uses" and "precedes" arcs, but only "uses" arcs are counted in the length. For example, in Figure 5 the path length to I_2 is two.

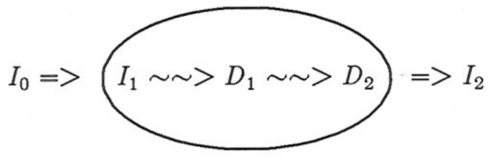


Figure 5

With these changes, we can now prove that under Scenario Two, for all pairs of vote increases I_j, I_k, either I_j --> I_k or I_k --> I_j using the definition of --> for Scenario Two. (We will henceforth use the Scenario Two definition.) The following four lemmas correspond to those used for the proof of Scenario One. First we present a stronger version of Lemma 2.1.1.

Lemma 2.2.1: Under Scenario Two, if we exclude the "sees" arcs between increments, the resulting graph (the directed graph representing the various relationships between vote operations) is acyclic.

Proof: As before, I_k => I_x implies that I_k "committed" before I_x. Since decrements are committed unilaterally by the originating node, D_k => I_x or D_k --> I_x implies that D_k committed before I_x. Finally, θ_k ~~> D_x means that at $node(\theta_k)$, θ_k completed before D_x. Since there are no cycles in the committed relationship, the graph can have no cycles. □

In addition, we make the following extension to Lemma 2.1.2.

Lemma 2.2.2: If I_x --> I_j and VS_j => ... => ...=> I_k then I_x --> I_k.

Proof: By Protocol *P3*, a voting node must send its vectors *V* and *N*. In addition, a local vector is updated only upon notification (via other vectors) of a more recent value at some node. Hence, every vote set in the chain from VS_j => ... => ... => I_k sees I_x explicitly or sees some $post(I_x)$, so I_x --> I_k. □

Lemma 2.1.3 comes over directly. We present it here as Lemma 2.2.3 without proof.

Lemma 2.2.3: Under Scenario Two, if two vote increments I_j and I_k occur such that $MAJG_{I_j} \cap MAJG_{I_k} \neq \emptyset$, then either I_j --> I_k or I_k --> I_j.

Lemma 2.2.4 makes a weaker statement than Lemma 2.1.4 but is sufficient for the proof of Theorem 2.2.1.

Lemma 2.2.4: If two increments I_j and I_k occur such that I_j sees the vote increments in the vote sets that I_k is using and I_k sees the vote increments in the vote sets that I_j is using, then I_j --> I_k or I_k --> I_j.

Proof: If $MAJG_{I_j} \cap MAJG_{I_k} \neq \emptyset$, then Lemma 2.2.3 applies and I_j --> I_k or I_k --> I_j. Therefore, assume that $MAJG_{I_j} \cap MAJG_{I_k} = \emptyset$. If for every node $x \in MAJG_{I_j}$, v_x at $node(I_j) \leq v_x$ at $node(I_k)$, and for every $y \in MAJG_{I_k}$, v_y at $node(I_k) \leq v_y$ at $node(I_j)$, then the proof of Lemma 2.1.4 carries over and we have a contradiction. Therefore, we must have at least one node where this does not hold. Say it is an $x \in MAJG_{I_j}$ where v_x at $node(I_j) > v_x$ at $node(I_k)$.

The situation at this point is as follows (See Figure 6). At node x some vote set, VS_α, has been used by I_j. Let θ_α be the specific operation used, i.e., $\theta_\alpha \in VS_\alpha$ and θ_α => I_j. (θ_α could be I_x, the increment in VS_α, or one of the decrements.) When x receives the request for votes from $node(I_j)$ the version number at x is $n(\theta_\alpha)$. The new votes for I_j and their version numbers are recorded at x at this time. Hence, whenever x informs other nodes of an operation in $post(\theta_\alpha)$, it will also inform them of I_j. That is, if θ_β --> I_k and $\theta_\beta \in post(\theta_\alpha)$, then I_j --> I_k.

Hence, to complete our proof, all we have to do is show that there is such a θ_β. By our hypothesis, I_α --> I_k (I_α is the increment of VS_α). This means that I_k has seen at least through $n(I_\alpha)$. However, the value v_x actually used by I_k was smaller than that produced by θ_α. Therefore, I_k must have also seen some $\theta_\beta \in post(\theta_\alpha)$, and by our argument above I_j --> I_k. □

Next is our main theorem.

Theorem 2.2.1: Under Scenario Two, for all pairs of vote increments, I_j, I_k, either I_j --> I_k or I_k --> I_j (or both).

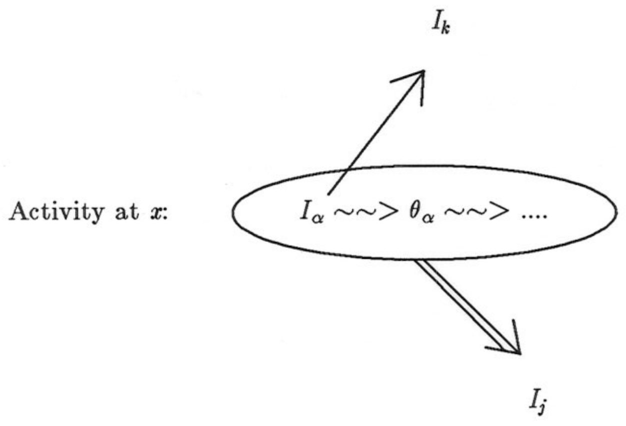


Figure 6

Proof: The proof of Theorem 2.2.1 proceeds in much the same way as that of Theorem 2.1.1. There are only two significant changes. The first is in the induction step of the outer induction. Here we have a graph with *i+1* increments and we want to reduce it to one with *i* increments. To do this we use Lemma 2.2.1. The lemma tells us that there must either be a decrement node with no outgoing arcs or an increment node with only outgoing "seen by" arcs. If it is a decrement, then it was not seen or used by any operation, so it is safe to remove it. We continue to remove such decrements until we have an increment with only outgoing "seen by" arcs. The fact shown in the proof of Theorem 2.1.1 then tells us that this increment can be safely removed. The removed increment becomes the I_f for the rest of the proof.

The second change is in the inner induction. Here we use Lemmas 2.2.2 and 2.2.4, but do not need Lemma 2.2.3. Also, as mentioned earlier, the path length ignores decrements. To illustrate the changes, we prove the induction step in the new scenario.

We wish to show that an increment I_m, on a path of length *p+1* from the original assignment, has been seen by or sees the removed increment I_f. By the first inductive hypothesis we know that for any increase I_f* on the paths to I_f, either I_f* --> I_m or I_m --> I_f*. By inductive hypothesis two, for all I_m* on the paths to I_m, I_m* --> I_f or I_f --> I_m*. If for any I_f*, I_m --> I_f*, then by Lemma 2.2.2 I_m --> I_f and we are done. Similarly, if for any I_m*, I_f --> I_m* then I_f -->

I_m (Lemma 2.2.2) and we are done. Otherwise, $I_m{}^* \dashrightarrow I_f$ for each such $I_m{}^*$ and $I_f{}^* \dashrightarrow I_m$ for each such $I_f{}^*$. This means that I_f has seen all the increments in all the vote sets I_m is using, and I_m has seen all the increments in all the vote sets I_f is using. Therefore, by Lemmas 2.2.3 and 2.2.4, either $I_m \dashrightarrow I_f$ or $I_f \dashrightarrow I_m$. □

Finally, it is easy to see that the deadlock anomaly can be avoided using Protocols *P3*, *P4* and *P5* for Scenario Two.

3. A SIMPLE APPLICATION

In Section 2, we proved a weak property of the protocols for Scenarios One and Two. Here, we claim that it is enough to provide mutual exclusion in a distributed system; that is, any application that uses static votes can use dynamic voting instead. We will not prove this general claim; however, we will show that mutual exclusion is preserved in a simple example. It is easy to use the same ideas for other applications.

The application we consider is locking objects in a database system for mutually exclusive access. The goal of locking is as follows. Say an object x exists in the system. Transaction T performs $lock(x)$. When $lock(x)$ is complete, T has exclusive access to x. T performs $unlock(x)$ to release the lock.

First we consider how *lock* and *unlock* can be implemented in the case of a static vote assignment. Please keep in mind that this is just a simple description that adheres well to our model. Much more efficient implementations exist but are not discussed here. For each object x in the system, each node has a local "lock" implemented as a log. A $granted(T)$ entry in the log records the fact that the local lock was granted to transaction T. (Transactions are identified by a timestamp and the id of the node where the $lock(x)$ action initiated.) Similarly, a $released(T)$ entry indicates that the lock has been released. If every *granted* entry has a matching *released* entry, then the local lock for x is available. Otherwise, the lock is held by the transaction with the unmatched *granted*. For simplicity, let us assume that logs are stored on stable storage.

When a transaction T at node a wants to perform a $lock(x)$, it sends $lock-request(x)$ messages to all nodes. Each receiving node b checks the request against its lock log. (Node a also acts as a receiving node.) If the lock can be granted at b, b enters $granted(T)$ in its log and replies *yes* with its assigned number of votes. If the initiator a receives a majority of votes, the $lock(x)$ is successful and transaction T has a systemwide lock on object x.

If the log for x at node b indicates that the local lock is not available, then node b replies *no*. Node b also forwards the log for x to node a, to insure that node a is informed of the pending lock. When node a fails to receive a majority of votes, or when it receives a *no* message, it aborts the request. In this case a performs $unlock(x)$ by sending $unlock-request(x)$ messages to all nodes. It is not necessary to collect votes for $unlock(x)$. (As the nodes receive these messages, they add $released(T)$ entries to their logs.) If node a received logs from other nodes, it merges them with its own, to have an up to date view of what is locked. (Duplicate entries are removed in the merge process.)

If T accomplished $lock(x)$, when T no longer needs object x, T proceeds as in the abort case. To ensure that all nodes eventually unlock, we can assume that a node periodically checks if a lock has been granted for a "long" period of time. If so, it can ask the initiator if it missed an $unlock-request(x)$.

We refer to the above method as the *one-phase locking protocol*. In the case of static votes it is easy to see that the protocol ensures mutual exclusion. Suppose both T_i and T_j have initiated $lock(x)$ requests and neither has initiated a $release(x)$ operation. If both transactions get a majority of votes, then there must be one node that replied *yes* to both requests. (All majority groups intersect in a static vote assignment.) Since this is not possible, then at most one of the requests could have been successful. Note that if we represent the lock operations by L_i and L_j, then we can say that either L_i sees L_j (in which case T_i is aborted), L_j sees L_i (T_j is aborted), or both see each other (both transactions are aborted). Using our earlier notation, we have that either L_i --> L_j or L_j --> L_i (or both).

Although the mechanism we have described guarantees mutual exclusion, it of course has drawbacks. It may lead to starvation and blocking. Local locks are implemented inefficiently. However, we have chosen it to illustrate simply how a protocol can be extended to operate with dynamic vote reassignment. (The technique for making the transition to dynamic voting which we are about to present would also work for a more efficient locking protocol, but the proofs would not be as obvious.)

As one might expect, we have to do some extra work to enjoy the benefits of dynamic voting. Lock information must be propagated as the voting power shifts. Consider the following simple scenario, using again Figure 1 and initial vote assignment $V_a[a] = V_b[b] = V_c[c] = 1$ and $V_d[d] = 2$. Say that node c initiates transaction T_c that requires a lock on object x. Node c sends $lock-request(x)$ messages to all other nodes. Assume that nodes c and d approve the request immediately, so c gets 3 votes out of 5, and the $lock(x)$ is successful. Let us also assume that node a never receives the lock request message from c. Now say that node a, for some reason, successfully increases its votes to $V_a[a] = 100$ and no other nodes increase their votes. Now, of course, a in effect has all the voting power in the system. Next a partition occurs, and a becomes isolated. Node a now can act independently, granting itself any locks available at a since it has 100 votes out of 104. Nodes b, c and d cannot collect a majority and are left to work only with locks they have already obtained. T_c then can continue to use object x. (T_c has not performed $unlock(x)$ yet.) Node a, however, has no knowledge of the $lock(x)$ at c (remember - the $lock-request(x)$ message sent to a was lost) and can now $lock(x)$ independently. Say a transaction T_a does just that. Now there are two locks concurrently held on the same object, certainly an undesirable situation. To avoid this, when node a increased its votes, it should have received the lock logs of the voting nodes, thus propagating the lock information.

The procedure for dynamic voting, then, is as follows. When a node a initiates Protocol *P2* to increase its votes, an acknowledging node, b, must send its lock log along with its votes. Node a, then, must integrate the lock information with its own, adding any locks or unlocks it has missed. The one-phase commit protocol for the static case can be used to obtain locks, except that votes are counted using Protocol *P1* (or *P3*). Note that the integration step may make it appear to a node that two (or more) locks are held on the same object by transactions at different nodes. This is due to the nature of the one-phase locking protocol. This does not mean that two different transactions have performed a successful *lock*() on the same object concurrently. At most one such *lock*() is approved, at least one will be backed out.

We can now prove the following corollary.

Corollary 2.2: Mutual exclusion under the lock scenario is preserved using Protocols *P1* and *P2*: no two nodes can concurrently hold a lock on the same object.

Proof: Note that a successful $lock(x)$ event, L_i, and a successful vote incrementing every, I_j are very similar. Both use the same vote collecting protocol and propagate vote and lock information. Thus, we can now speak of a general event E, where E is a lock or a vote increment action. Using our earlier notation we can define the "has used" and "has seen" relationships:

if I_j => E_k, then E_k has used vote increment I_j,

if E_j --> E_k, then E_k has seen event E_j.

Using basically the proof of Theorem 2.1.1 (or 2.2.1) we can show that either E_j --> E_k or E_k --> E_j for all pairs of events E_j, E_k. The rest of the proof proceeds by simple contradiction. Say two nodes hold locks on the same object concurrently, corresponding to events E_a and E_b and say E_a --> E_b (E_b --> E_a works analogously). Then, the node initiating event E_b knew of the conflicting lock granted for event E_a. E_b, then, could not have been approved under the one-phase commit protocol described above. □

In summary, when using dynamic voting for a particular application, information that must endure partitions (e.g., the fact that an object is locked or the values that a committed transaction has installed in the database) must be given to a node that is increasing its votes. This ensures that any node collecting a majority in the future will obtain the same information from the node with more votes than it would have obtained directly from the nodes that participated in the vote increase.

4. POLICIES

Now that we know how a node can install a vote change, we need a systematic way for the node to pick a new vote value. In this section we describe various policies which can be used to make this decision. It is important to keep in mind that the protocols guarantee mutual exclusion. The policies are merely attempting to get a good assignment for the system that is resilient to future failures. It is possible that at times the nodes will not adhere strictly to the chosen policy because of misinformation about the state of the network. This may lead to a poor assignment. It will not, however, endanger mutual exclusion. We consider first vote increasing policies which are divided into two basic strategies:

- **Alliance techniques.** After a failure (or group of failures), *all* the nodes in the active group increase their votes.
- **Overthrow techniques.** After a failure (or group of failures), *one* node in the active group takes on more votes.

4.1 The Overthrow Technique

Vote increasing under the overthrow technique is straightforward. Consider a system in which node x has gone down, while the rest of the nodes are still up. (This can be considered as a partition of the system into two groups, with x in one group and the rest of the nodes in the other.) Let v_x be the number of votes that node x has. Let TOT be the total number of votes in the system and MAJ the majority of votes. Assuming TOT is odd, $MAJ = (TOT+1)/2$. If node a is the node supplanting x, the new number of votes for a, $v_a{}'$ will have to be such that it covers the voting power that a had before (v_a), plus the voting power of x, plus the increase in the total number of votes. If a increases its votes by $2v_x$, the total number of votes will be $TOT' = TOT + 2v_x$ and $MAJ' = MAJ + v_x$. It can be shown that all the majority groups that used x can be formed using a instead:

- If a group G had V_G votes and contained a and x, the group $G' = G - \{x\}$, will have $V_{G'} = V_G + 2v_x - v_x \geq MAJ + v_x = MAJ'$. Therefore, G' is a majority group under the new vote assignment.

- If a group G had V_G votes and contained x, but not a, the group $G' = G \cup \{a\} - \{x\}$ will have $V_G{}' = V_G + v_a + v_x > MAJ'$.

Of course, any other group in which x does not participate may suffer loss of voting power and need the help of node a to complete the majority, but the basic goal of supplanting x is achieved.

Deciding which node should increase its votes for the node(s) that are no longer in the active group can be accomplished, for example, by using a priority mechanism. The nodes can be initially ranked and the node in the majority group with the highest priority can increase its votes. Instead, a token passing mechanism can be used, where the node with the token increases its votes. In any case, the method does not need to be foolproof. If problems such as communication delays arise and nodes pick a vote value that is not accurate or several nodes supplant an excluded node, the worst that can happen is the new assignment is not as good as it could be. It will not lead to more than one active group.

4.2 The Alliance Technique

There are many variations of the alliance technique. We describe three here. In general, we want to give each node a fraction of the voting power of a node that has been excluded from the majority group. As in the overthrow technique, we want to be sure to give out at least $2v_x$ votes in the majority group, enough to counteract those votes that node x holds plus the number of votes node x could have contributed if it were in the active group. Of course, we can always assign a surplus of votes to each node. One possibility is to assign $2v_x$ votes to every member of the active group; or, we can assign v_x votes to each member of the active group, and assign $2v_x$ votes when there is just one node left. Another possibility is to spread the $2v_x$ votes out. Say N = the number of nodes in the majority group. Then, give each node in the active group $\lceil 2v_x/N \rceil$ votes (henceforth referred to simply as $2v_x/N$). If need be, N can be estimated by the nodes. This may not be as good as possible in terms of resilience to failures, but is certainly not dangerous. No matter what the strategy, we have to be careful when there are only 2 nodes left in the majority group. In that situation, it is senseless to give each node the same number of votes, since if they lose communication with each other, their extra votes will only cancel each other out and no group may have a majority. Instead, it is better to pick one node and give it $2v_x$ votes. We can use a priority mechanism to handle this case.

4.3 Examples

We illustrate these techniques with an example. Consider again the system of Figure 1, but with initial vote assignment $v_a = 6$, $v_b = v_c = v_d = 5$. Assume that node a gets disconnected from the rest, leaving $\{b, c, d\}$ as the active group. Using the overthrow techniques, if we say node b is of lowest rank or has the token (depending on the strategy used), the new vote assignment will be

$$v_a = 6,\ v_b = 17,\ v_c = v_d = 5.$$

since we give node b $2v_a$ votes.

Now consider the three alliance techniques. If we give each node $2v_a$ votes we have

$$v_a = 6,\ v_b = v_c = v_d = 17.$$

If we give each node v_a votes we have

$$v_a = 6,\ v_b = v_c = v_d = 11.$$

If we give each node $2v_a/N$ votes, N = the number of nodes in the majority group, we have the assignment

$$v_a = 6,\ v_b = v_c = v_d = 9.$$

There are obvious differences between these assignments. For instance, the first two have node(s) with a higher number of votes than the last two. In general, the size of the votes will grow much faster using the first two techniques than in the last two. We will discuss how to handle this problem shortly. Also, the first assignment gave much power to one node, which can be a disadvantage if this node fails. Also important is the amount of message traffic these assignments incurred. Since only one node got votes in the overthrow case, this technique required fewer messages. But just looking at the assignments tells us little about their relative performance. This is just one 4-node system and set of failures, certainly not representative. A later presentation of simulation results will give us a better means of comparison.

4.4 Balance of Power

Of course, whenever a node is excluded from the majority and other nodes increase their votes, the balance of voting power is disturbed. What we need are techniques for maintaining the original vote distribution of the system. There are two possibilities:

(1) A node that has been out of the active group can "catch up" when it returns to the active group, in other words, increase its votes.

(2) When a node that has been out of the active group returns, the node(s) that increased their votes because of its absence can relinquish them, i.e., decrease their votes.

These two strategies are directly related to the two scenarios discussed in the Protocols section. Method (1) requires only the vote collecting and increasing protocols of Scenario One. Method (2) necessitates the added capability to decrease votes as in Scenario Two. Another factor in choosing between catching up and decreasing votes is the effect of each on availability. This is discussed in the section on simulation results. In addition, the policy chosen for increasing votes affects which of these two is preferable, as we shall see.

4.4.1 Catch up Strategies

When a node x is separated from the active group each node in the active group takes on more votes, depending on the policy chosen for vote increasing and the value v_x. When node x returns to the active group it should increase its votes by at least as many as other nodes did when x was initially excluded. In addition, node x can keep track of the last majority group that it participated in. When it becomes part of an active group again, it can determine which nodes were in the previous majority group with x, but are not in the present majority group. These nodes caused vote increases since the last active group that included node x. Node x can then increase its votes as if it were present when those nodes were initially excluded. This allows node x to pick up most (but maybe not all) of the vote changes that occurred while it was excluded from the active group. It may not, for example, increase its votes for a node, y, that has failed after x failed, but repaired before x repaired.

For example, referring again to Figure 1, suppose we have the initial vote assignment $v_a = 6$, $v_b = v_c = v_d = 5$. Assume we are using the $2v_x$ alliance technique with catch up. Say once again that node a goes down and nodes b, c, d get 17 votes each as in the example of Section 4.3. Then, say node b fails and nodes c and d take on 34 more votes each yielding the assignment

$$v_a = 6,\ v_b = 17,\ v_c = v_d = 51.$$

Now, node a returns and wishes to catch up. It takes 2 times its own votes automatically. Then, it checks to see who is in the active group and notes that node b is not. Since node b was in the last active group a was in, b must have become excluded in the interim. So, node a takes $2 \times v_b$ votes as well, which it has learned from node c (or d) is 17. So, node a gains 46 votes for a total of 52. The final assignment is

$$v_a = 52,\ v_b = 17,\ v_c = v_d = 51.$$

and node a has caught up.

Clearly, this technique is not relevant to overthrow methods. If only one node is taking on votes for a disconnected node, then a returning node does little to rebalance the voting power by catching up to just one node. A second consideration is that we can not let votes increase forever. Eventually the nodes must decide to decrease their votes. After some node hits a predefined threshold vote value it can attempt to initiate some consensus technique to bring all the votes back down to their original values. This may require that all nodes be in the active group and that other processing wait until the votes are reset. This is not too severe, though, since this reconfiguration should not need to be done too often.

4.4.2 Decrease Strategies

Decreasing votes avoids the problem of votes getting too big. To implement this, each node must keep track of how many votes it took on when some other node became excluded. When the node returns to the active group, the nodes can decrease their votes by the appropriate amount. This requires that each site maintain a table with an entry for each other node indicating the vote change. This method applies to either overthrow or alliance techniques.

In the next section we present performance results for the techniques described along with an analysis of the advantages and disadvantages of each method.

5. PERFORMANCE RESULTS

As discovered in the last section we have many policies to choose from for an implementation of dynamic vote reassignment. The next step is to determine their usefulness. Several questions require our attention: Is dynamic vote reassignment much better than retaining a static assignment? Is group consensus much more resilient than the autonomous techniques? Is one autonomous technique better than the rest? Is the topology of the network relevant? In an attempt to answer these questions we simulated the policies of the previous section.

5.1 Methods

Each experiment uses an event driven simulator where the events consist of node and link failures, and repairs. Times between failures (and repairs) are exponentially distributed. The failure rates are assumed to be very high in order to focus our attention on how well the system adapts. In other words, if we choose typical failure rates (e.g., each component is down 1% of the time), the system will halt rarely and the vote reassignment policy will have little effect on average reliability measures. However, our goal is to minimize disruptions *during* failures, so instead we zero in on a failure period when the system is unstable, by selecting high failure rates.

To simplify the simulation we assume that the vote reassignment is done instantaneously. In other words, if an event causes a new vote assignment, the vote changes occur before the next event does. This approach will yield worst case results for autonomous reconfiguration techniques as compared to group consensus since we presume that group consensus vote increases take much longer in practice than autonomous ones.

The topology and the connectivity of the communication network play an important role in dynamic vote reassignment. Of course, we must limit the number of networks studied, and Figure 7 presents the six 5-node systems for which we give results in this paper. The networks represent a spectrum of connectivity.

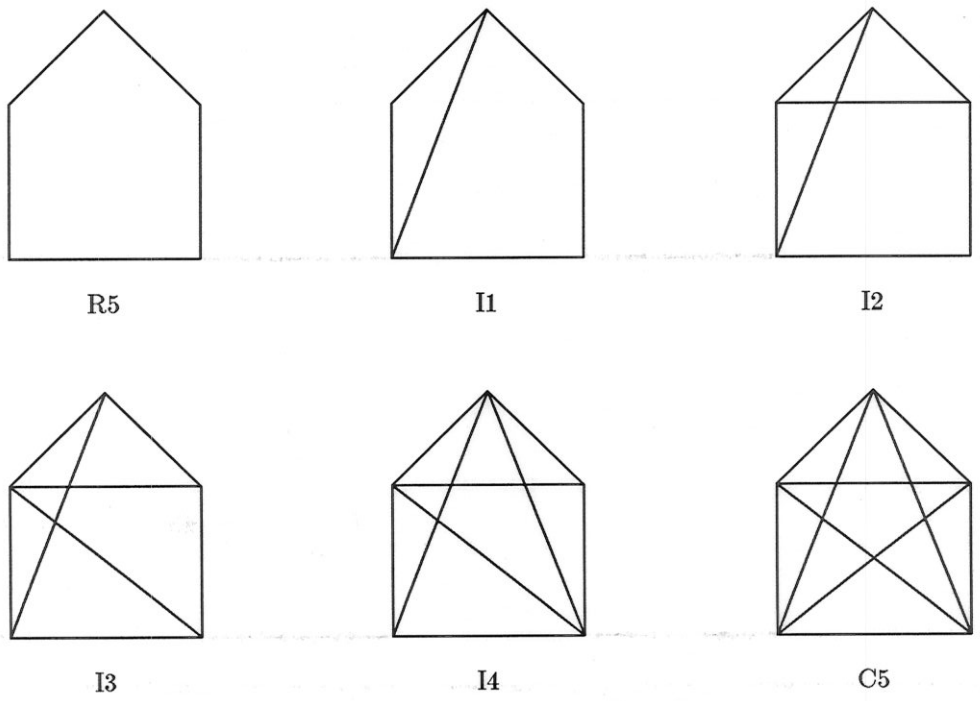


Figure 7

All results obtained a 95% confidence interval with a width of 10 percentage points. Each policy was simulated for 200 time units for the six network topologies shown in Figure 7. Three different failure and repair scenarios were considered:

(1) Nodes and links fail and repair at a rate of 20.

(2) Nodes fail and repair at a rate of 20; links don't fail.

(3) Links fail and repair at a rate of 20; nodes don't fail.

Nine strategies were simulated:

(1). Alliance: $2v_x$ votes with vote catchup
(2). Alliance: v_x votes with vote catchup
(3). Alliance: $2v_x/N$ votes with vote decreasing
(4). Alliance: $2v_x$ votes with vote decreasing
(5). Alliance: v_x votes with vote decreasing
(6). Overthrow: ranking with vote decreasing
(7). Overthrow: token passing with vote decreasing
(8). Group consensus
(9). Static Vote Assignment

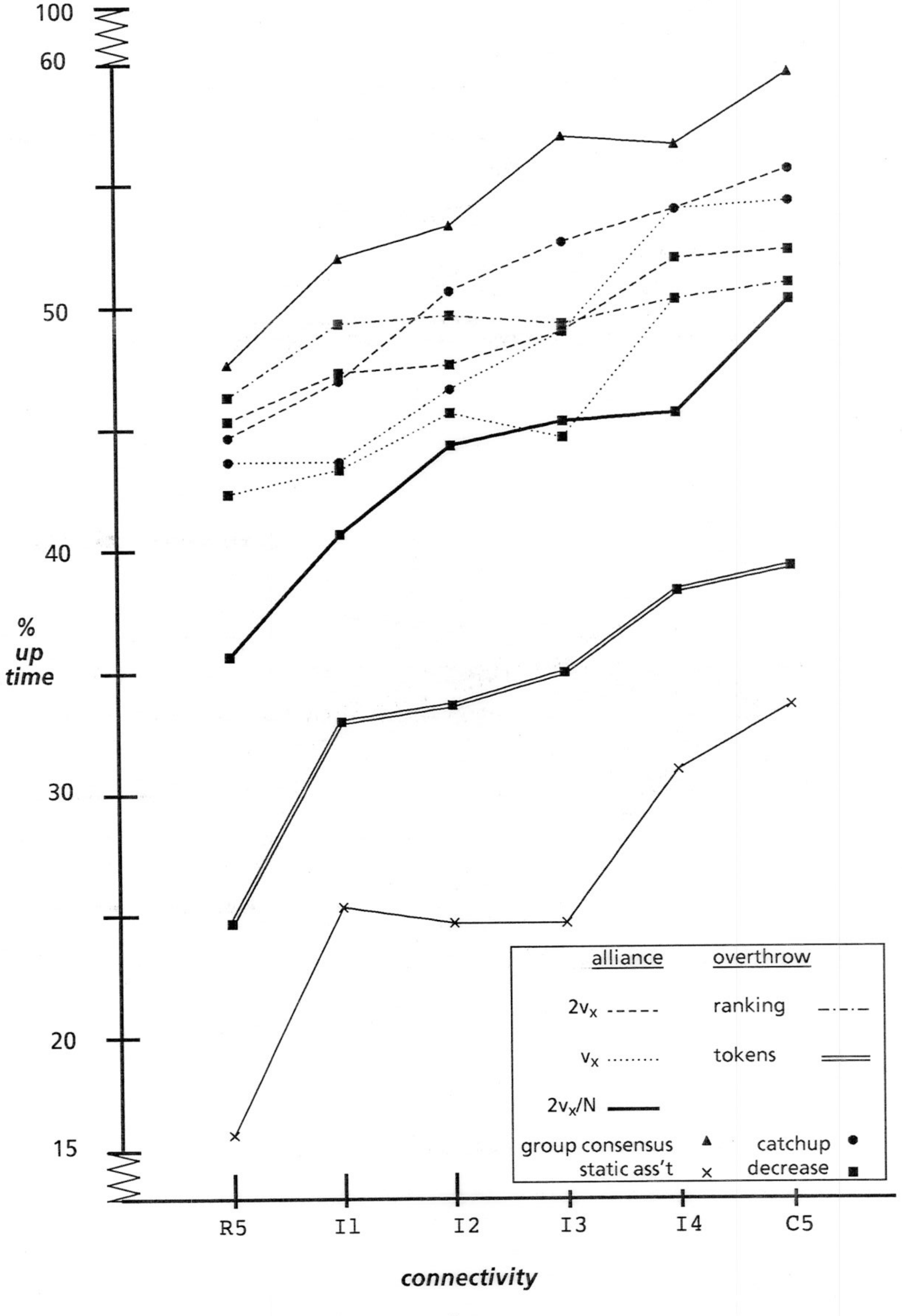
The overthrow technique combined with vote catch up was not implemented. As mentioned earlier, under the overthrow method, votes are not evenly distributed when a node is excluded, hence a node catching up catches up to one other node and does little to reestablish the initial vote assignment. Vote catch up was not implemented for the alliance technique with $2v_x/N$ votes either, since it would be difficult in practice for the returning node to know how many nodes were in the active group when it was excluded.

For each strategy, the nodes start off with an initial assignment determined by the topology of the network. Votes are distributed according to the number of links incident to a node. If need be, an extra vote is added to the node with the largest number of incident edges to make the total number of votes odd. See [Barb84] for the details and rationale of this technique. The static vote assignment strategy retains this same assignment throughout a run. The group consensus technique readjusts the votes after each failure according to this algorithm.

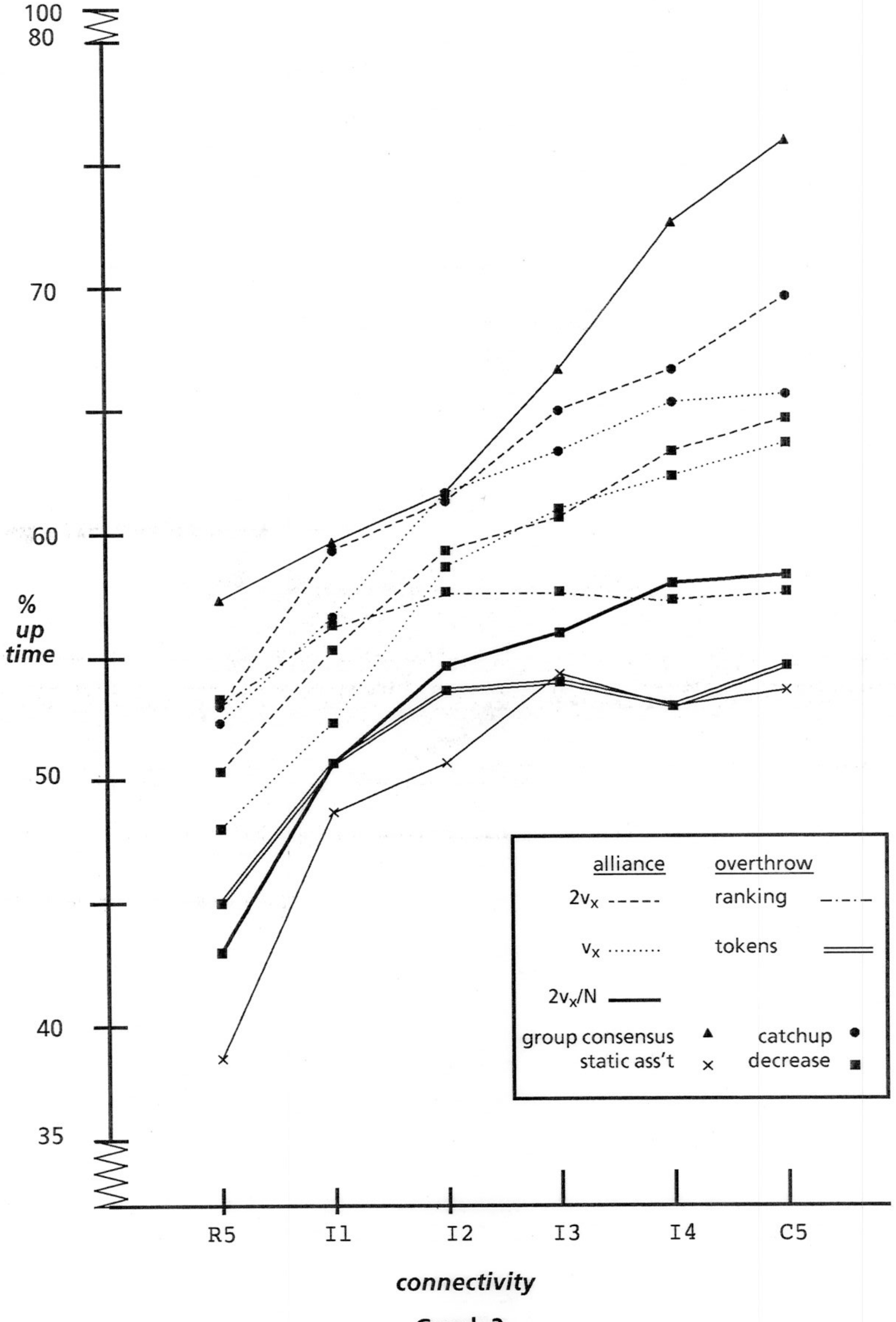
The strategies are compared on the basis of mean percent uptime, the average over all runs of the percentage of time that some group in the system could perform operations. For purposes of illustration, this is displayed graphically in Graphs 1, 2 and 3 for the three scenarios, even though the connectivity axis is not continuous. (Note that up times are relatively low, but remember that we are looking at a failure period only.)

5.2. Analysis

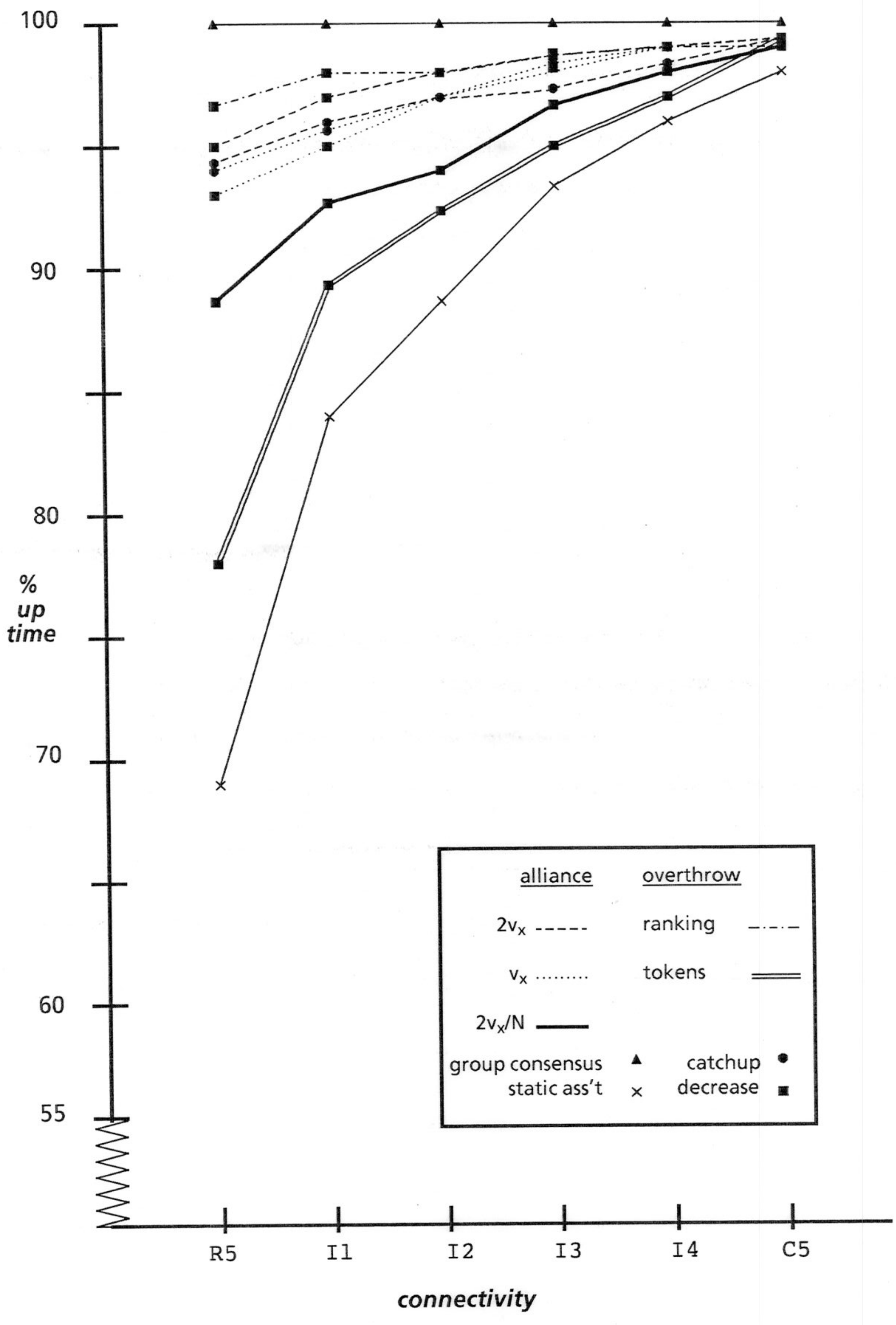
Consider first Graph 1, indicating results for both node and link failures and repairs. Certainly, any of the reconfiguration strategies proved more effective at keeping the system operative than retaining a static assignment. Group consensus showed a slight gain over the auto-



Graph 1
Both nodes and links fail.



Graph 2
Only nodes fail



Graph 3
Only links fail

nomous techniques in all cases. Among the alliance techniques, assigning $2v_x$ votes with catching up yielded the best results, especially in the intermediate cases, although $2v_x$ with vote decreasing worked as well for graphs R5 and I1. In general, though, catching up yielded better results than its decreasing counterpart for higher connectivity graphs. Alliance $2v_x$ performed better than v_x under the same balancing strategy. Use of the alliance policy with $2v_x/N$ votes showed relative improvement as the connectivity increased.

Overthrow using token passing did not work as well as the other reconfiguration strategies. This is not surprising since when the token ends up with some node not in the majority, no node gets extra votes. Overthrow using ranking, however, performed well. It was in fact slightly better than the alliance techniques for graphs R5 and I1, but worse for the I2, I3, I4 and C5 graphs. This is not surprising in light of work done in [Barb84] on static assignment, which asserts that vote distribution is not as advisable for rings and other low connectivity networks; a singleton assignment is preferable. But higher connectivity graphs perform better under vote distribution. Otherwise, $2v_x$ with decreasing appears to be a good choice.

As expected, all strategies yielded higher percent uptimes for the case of no edge failures (Graph 2). The gain to be had from any type of reconfiguring was less than that of the previous case, but still substantial. Once again, we see that of the alliance techniques, $2v_x$ with catching up gave a slight improvement over the other alliance techniques in most cases and catching up in general is better than vote decreasing. Use of the $2v_x/N$ alliance technique was the worst of the alliance group. Overthrow with ranking performed comparably to alliance for rings and low connectivity graphs but lost out as more links were added. The use of tokens with the overthrow technique did not perform as badly relative to the other techniques as it did for the previous scenario.

Group consensus again performed better than all autonomous techniques, however not significantly better than $2v_x$ alliance for the intermediate graphs, and only slightly for the ring. It was substantially better for the complete graph case.

Looking at the third scenario in Graph 3, where nodes do not fail and links fail and repair at the same rate, we see a large increase in the effectiveness of the static assignment. In fact for I3, I4 and C5, it is questionable that we should pursue a dynamic means of vote assignment. The most striking aspect of these results, though, is the perfect performance of group consensus. Since it is tailored to the topology of the system and the vote changes were immediate, it was not affected by the edge failures.

Here again, token passing in overthrow and $\frac{2v_x}{N}$ alliance showed a slightly poorer performance. The other alliance techniques performed nearly equivalently. The overthrow with ranking policy showed just a slight improvement over alliance for the less dense networks.

6. CONCLUSIONS

We have presented a technique for achieving higher availability in a distributed system operating under mutual exclusion constraints. Using voting as the basic mutual exclusion mechanism, upon failure of a node or partitioning of the network, nodes can reassign themselves new votes dynamically, in order to survive future failures. A *policy* is chosen which the nodes use to determine a new vote value. A set of *protocols* allow each node to initiate this vote change autonomously. The method is simple and fast and does not require accurate detection of failures and partitions at the sites. In addition, our method is flexible, it can be used for any application that requires mutual exclusion. The simulation results we have presented show that autonomous reassignment shows substantial improvement over a static assignment of votes and is a viable alternative to dynamically reassigning votes using a group consensus technique.

7. ACKNOWLEDGMENTS

The authors would like to thank Rafael Alonso and Ken Salem for their many helpful comments and suggestions.

8. REFERENCES

[Abba86] A. Abbadi, S. Toueg, "Availability in Partitioned Replicated Databases," *Proc. Principles of Database Systems Symposium*, 1986.

[Barb84] D. Barbara, H. Garcia-Molina, "Optimizing the Reliability Provided by Voting Mechanisms," *Proc. Fourth International Conference on Distributed Computing Systems*, October 1984, pp. 340-346.

[Bern81] P. Bernstein, N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185-221.

[Davc85] D. Davcev, W. Burkhard, "Consistency and Recovery Control for Replicated Files", *Proc. Tenth ACM Symposium on Operating Systems Principles*, December 1985, pp. 87-96.

[Davi85] S. Davidson, H. Garcia-Molina, D. Skeen, "Consistency in Partitioned Networks", *ACM Computing Surveys*, Vol. 17, No. 3, September 1985, pp. 341-370.

[Garc82a] H. Garcia-Molina, "Elections in a Distributed Computing System", *IEEE Transactions on Computers*, C-31, No. 1, January 1982, pp. 48-59.

[Garc82b] H. Garcia-Molina, "Reliability Issues for Fully Replicated Distributed Databases", *IEEE Computer*, Vol. 15, No. 9, September 1982, pp. 34-42.

[Giff79] D.K. Gifford, "Weighted Voting for Replicated Data", *Proceedings Seventh Symposium on Operating System Principles*, December 1979, pp. 150-162.

[Gray78] J.N. Gray, "Notes on Database Operating Systems", in *Operating Systems: An Advanced Course*, Springer-Verlag, New York, 1978, pp. 393-481.

[Thom79] R.H. Thomas, "A Majority Consensus Approach to Concurrency Control", *ACM Transactions on Database Systems*, Vol. 4, No. 2, June 1979, pp. 180-209.