PROTOCOLS FOR DYNAMIC VOTE REASSIGNMENT

Daniel Barbara
Hector Garcia-Molina
Annmarie Spauster

CS-TR-037-86

May, 1986

PROTOCOLS FOR DYNAMIC VOTE REASSIGNMENT

Daniel Barbará
Hector Garcia-Molina
Annemarie Spauster

Computer Science Department
Princeton University
Princeton, New Jersey 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. Protocols are given which allow nodes to select new vote values autonomously while still maintaining mutual exclusion requirements. The lemmas and theorems to validate the protocols are presented, along with proof of correctness. 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. Also included is a brief discussion of simulation results.

1. Introduction

In distributed systems, voting is used commonly to provide a mutual exclusion mechanism that works under catastrophic failures like partitions. Each node is a priori assigned a number of votes, and only the group with a majority of votes is allowed to perform a restricted operation [Davi82, Giff79, Thom79, Garc82]. This restricted operation may be, for instance, the update of a replicated database.

For example, consider the four node system illustrated in Figure 1. Each node manages a copy of a replicated database. Each node has been assigned one vote, except for node d which has received 2 votes. Now assume that 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 update the database, since they are assured that no other group is concurrently in the same situation. (The majority is actually checked within the updating transaction's commit protocol [Skee82]. If the reader is not familiar with such protocols, one can simply think of the nodes within a group as updating the database in unison.)

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 and hence no one can update the database. This is an undesirable situation since it makes the data unavailable for updates everywhere in the system.

To minimize the likelihood of these undesirable *halted* states, 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]. In addition, we can attempt to dynamically reassign the votes in the presence of failures, which is what we advocate in this paper. 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.

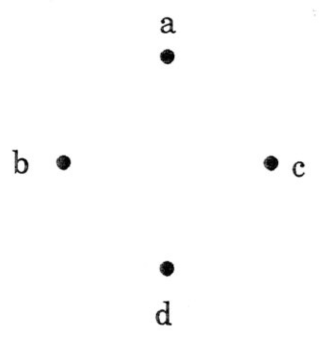


Figure 1

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 a 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. When the partitions are repaired, the nodes that have proportionately less votes (e.g., d) can attempt to increase their votes.

In our approach, we want vote change decisions to be *autonomous*, without requiring group consensus (such as electing a coordinator or using a distributed algorithm). Group consensus techniques can select very good assignments but necessitate tight coordination among the members of the group. Also, nodes need good knowledge of the current system topology and state. On the other hand, autonomous vote changes are much simpler and more flexible. Each node decides independently if it should attempt to change its votes and determines on its own what its new vote value should be. In addition, the node does not require complete or accurate information about the state of the system. In a sense, the node makes an educated guess about the best number of votes to have, based on what it assumes is the state of the network. Its primary goal is to claim for itself all or part of the voting power of a node (or nodes) that have been separated from the majority group. For instance, in our example, after the first failure nodes a and b decide to increase their votes by 4, i.e., twice the number of votes that the disconnected node d had. When c fails, node a does the same thing and takes on 10 more votes. Node b, on the other hand, does not increase its votes after the c failure because it recognizes that there are only two nodes left in the majority group. In that case, it is best to give one of the nodes the majority of votes. However, if node b does not recognize this fact, it may try to increase its votes, too, and may succeed. This does not lead to the best assignment, but it still yields a valid one, i.e., one that guarantees mutual exclusion among updaters. We refer to this method as autonomous reconfiguration and concentrate here on this strategy.

We use the term *policy* to refer to the mechanism for autonomously selecting the new assignment. Note that selecting a new vote value is only half of the problem. After the selection, the node must "install" the new value, making sure that it is in a majority group that is allowed to change its votes. In our example, when node d is disconnected from the rest of the system, its policy may dictate that it try to increase its voting power to compensate for the loss of nodes a, b, and c, but it should be unable to install this or any change. We refer to the algorithm for installing a change, i.e., for making a vote transition 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.

In this paper we examine in detail the technique of autonomous vote reassignment, concentrating on the protocols and their proof of correctness. We will also demonstrate the applicability of autonomous vote reassignment to a simple database object-locking scheme. Our protocols, however, are versatile and can be used for any application that requires mutual exclusion (e.g. replicated data management, coordinator election, majority transaction commit protocols, etc.).

It will be clear that for any application, our protocols provide increased protection against partitions. Partitions may not be too common on simple networks (e.g., an ethernet), but modern networks are more and more often composed of heterogeneous and evolving collections of networks. In these networks, partitions are more common, since a gateway failure or a software bug can disconnect two networks. Also, in many cases, a simple host failure cannot be distinguished from a real partition (where that host is disconnected from the network). Thus, it is becoming more important to protect against such disasters.

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. We believe that allowing for such errors reflects the reality of distributed computing networks.

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 the next section we present protocols for implementing autonomous vote reassignment, along with a proof of correctness. In Section 3 we demonstrate how dynamic vote changing can be used for applications requiring mutual exclusion via a simple example. We have also studied the availability provided by autonomous vote reassignment through detailed simulations. As expected, it yields substantially higher availability than static vote assignments. Surprisingly, it also yields almost as much availability as the group consensus approach, but with less overhead. The simulation results, as well as choices for policies, are briefly discussed in Section 4. (A full discussion of the policies and the simulation results appears in [Barb86].)

2. Protocols

In this section we present two protocols that work together to provide for autonomous vote reassignment in a distributed system. One is a *vote collecting* protocol that is used whenever a node is collecting votes to perform an action that requires mutual exclusion. The second is a *vote changing* protocol and is used when a node wants to adjust its vote value. Vote changing is an event requiring a majority, so vote collecting is invoked whenever vote changing is performed. In this paper the protocol for vote changing only allows nodes to increase their votes. (A group consensus technique can be used to reset to the original vote assignment. Alternatively, we can allow nodes to decrease their votes autonomously. This involves further protocols that are not discussed in this paper.)

We start by establishing some notation and terminology. We establish 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 v_k to indicate the votes of a node k as determined upon vote collecting. We now present the protocols for vote collecting and vote increasing.

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 $v_k = \max(V_j[k])$ for $j \in G$. In this way, k is considered to have the largest number of votes recorded among the nodes that have voted. At the same time, 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 a majority of acknowledgments, 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. □

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 votes.

Before presenting formal proofs, we point out some key aspects of Protocols *P1* and *P2*. One important observation is that we cannot apply a total ordering to vote increments such that a vote increase is aware of all those that occur before it in the ordering. Several nodes may be running the protocol concurrently. Running the protocol is undoubtedly affected by such factors as network transmission delays and load factors at participating nodes. As a result, 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. 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.

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 $n(I_j)$ 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 $MAJG_{I_j}$. The sum of the votes obtained from $MAJG_{I_j}$ is called $MAJV_{I_j}$. In addition, we often refer to the the nodes that were not in the voting group and call them $ming_{I_j}$ with vote total $minv_{I_j}$. Note that for any such increment I_j, $MAJG_{I_j} \cap ming_{I_j} = \emptyset$ and $MAJV_{I_j} > minv_{I_j}$.

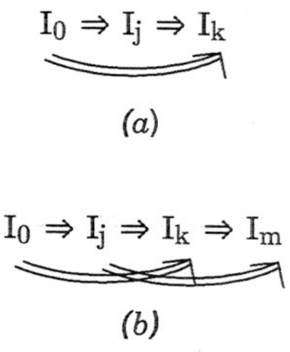
In addition, for an increment I_j we distinguish between the previous vote increments that I_j uses as votes to collect a majority and those I_j is aware of but does not necessarily use. Say I_k changes the votes of node $n(I_k) = z$ to $v(I_k)$. When I_j collects votes using Protocol *P1*, it determines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

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

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

Note that $I_k \Rightarrow I_j$ implies $I_k \rightarrow I_j$. Also, notice that for $I_k \Rightarrow I_j$ 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 $I_k \Rightarrow I_j$ but $k > j$. By Protocol *P2*, however, I_k must have finished 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 $n(I_j) = n(I_k)$ and I_j and I_k occur one right after the other at that node. Then, $I_j \Rightarrow I_k$ and $I_k \rightarrow I_m$ (I_m at any node) implies $I_j \rightarrow I_m$.

We also establish a representation for the set of vote increases that lead up to a later vote increase. As an example, let I_0 signify the original assignment. If I_j uses only the original assignment, then $I_0 \Rightarrow I_j$. If I_j is subsequently used for I_k, along with some original votes, then this is depicted as in Figure 2(a). Also, I_j and I_k may be used for I_m as in Figure 2(b). Converting increments to vertices and $\Rightarrow$ to directed edges, we obtain a one-to-one correspondence with a directed graph with source I_0. Any set of increments can be so represented by a connected directed graph. We say that a set of paths lead up to a vote increment and define the length of any such path to be the number of increments on it. This is simply the path length from I_0 in the directed graph.


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

(a)

$$\begin{tikzcd} I_0 \arrow[r, Rightarrow] \arrow[rr, Rightarrow, bend right] & I_j \arrow[r, Rightarrow] \arrow[rr, Rightarrow, bend right] & I_k \arrow[r, Rightarrow] & I_m \end{tikzcd}$$

(b)

Figure 2

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 $I_1 \rightarrow I_2 \rightarrow I_3 \rightarrow \cdots$ where I_1 occurs before I_2, I_2 before I_3, etc. We can prove, however, that $I_j \rightarrow I_k$ or $I_k \rightarrow I_j$ for all pairs of vote increments I_j and I_k. 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 and for two conflicting actions at least one node will know of the conflict.

We now present the lemmas and theorems concerning vote increments and then apply them to a simple scheme for managing replicated data under mutual exclusion requirements.

First we prove an important property of the directed graphs.

Lemma 2.1: The directed graph representing the set of paths leading to an increment I_j is acyclic.

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

Next we prove three properties of the vote increments.

Lemma 2.2: If $I_x \rightarrow I_j$ and $I_j \Rightarrow \cdots \Rightarrow \cdots \Rightarrow I_k$ then $I_x \rightarrow I_k$.

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

Lemma 2.3: 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 $MAJG_{I_j} \cap MAJG_{I_k} \neq \emptyset$.

Proof: Say $MAJG_{I_j} \cap MAJG_{I_k} = \emptyset$. 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. 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 $MAJG_{I_j} \cap MAJG_{I_k} \neq \emptyset$. □

Lemma 2.4: If two vote increments I_j and I_k occur such that $MAJG_{I_j} \cap MAJG_{I_k} \neq \emptyset$, then either $I_j \rightarrow I_k$ or $I_k \rightarrow I_j$.

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 $I_j \rightarrow I_k$ via node x. Similarly, if x votes on I_k first then $I_k \rightarrow I_j$ via node x. Node x must do one or the other first, so the lemma is true. □

Next is our main theorem.

Theorem 2.1: For all pairs of vote increments, I_j, I_k, either $I_j \rightarrow I_k$ or $I_k \rightarrow I_j$ (or both).

Proof: The proof uses a double induction.

<u>Induction One</u> is on the number of vote increments, *i*:

Basis: 1 vote increment.

Obviously, I_1 sees its own increment, so $I_1 \rightarrow I_1$.

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. 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 $n(I_f)$, 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 $I_f \rightarrow I_k$, then there are two cases. <u>Case 1</u>: $n(I_f) \in ming_{I_k}$. 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. <u>Case 2</u>: $n(I_f) \in MAJG_{I_k}$. Then, I_f had not been installed at $n(I_f)$ when $n(I_f)$ voted on I_k. $n(I_f)$ voted with an older vote value; $v_{n(I_f)}$ at $n(I_k)$ 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 $n(I_k)$. □

With this fact, we can remove I_f from the set of *i+1* increments without preventing any of the other vote increases. By Inductive Hypothesis 1, for all the remaining *i* increments the theorem is true. It remains then to show that either $I_m \rightarrow I_f$ or $I_f \rightarrow I_m$ 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 $I_m \rightarrow I_f$ or $I_f \rightarrow I_m$ by <u>Induction Two</u> 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 ($I_0 \Rightarrow I_m$). By

Inductive Hypothesis One, for any vote increase I_{f^*} on a path to I_f, either $I_{f^*} \rightarrow I_m$ or $I_m \rightarrow I_{f^*}$. If $I_m \rightarrow I_{f^*}$ for some I_{f^*}, then $I_m \rightarrow I_f$ by Lemma 2.2 and we are done. Else, $I_{f^*} \rightarrow I_m$ 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 2.3 and 2.4.

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 $I_{f^*} \rightarrow I_m$ or $I_m \rightarrow I_{f^*}$.. We know by Inductive Hypothesis Two that for all I_{m^*} on this path to I_m, either $I_{m^*} \rightarrow I_f$ or $I_f \rightarrow I_{m^*}$. Once again, if for any I_{f^*} on the paths to I_f, $I_m \rightarrow I_{f^*}$ then $I_m \rightarrow I_f$ by Lemma 2.1 and we are done. Similarly, if for one of I_{m^*}, $I_f \rightarrow I_{m^*}$ then $I_f \rightarrow I_m$ by Lemma 2.1 and we are done. Otherwise, $I_{m^*} \rightarrow I_f$ for each such I_{m^*} and $I_{f^*} \rightarrow I_m$ 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 2.3 and 2.4.

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

Finally, there is one more issue we need to address. 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 by 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 importantly, 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. We look at the interaction of autonomous vote reassignment with an application in more detail in the next section.

3. A Simple Application

In Theorem 2.1 we proved a "weak" property for dynamic vote reassignment using Protocols *P1* and *P2*. We claim that this is enough to provide mutual exclusion in a distributed system; that is, any application (e.g., transaction commit) that worked with static votes will work with dynamic votes. 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 system wide lock on object x.

If the log for x at node b indicates that the local lock is not available, then the 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.

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 $L_i \rightarrow L_j$ or $L_j \rightarrow L_i$ (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 in an inefficient way. 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, except that 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 $V_a[a] = V_b[l] = V_c[c] = 1$ 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 send 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*. 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 doesn't 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 2.1: 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 event, 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 $I_j \Rightarrow E_k$, then E_k has used vote increment I_j

if $E_j \rightarrow E_k$, then E_k has seen event E_j.

Using basically the proof of Theorem 2.1 we can show that either $E_j \rightarrow E_k$ or $E_k \rightarrow E_j$ 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 $E_a \rightarrow E_b$ ($E_b \rightarrow E_a$ 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 anyone 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

In the interest of completeness we provide a short discussion on policies, the choice a node has for picking a new vote value, and briefly mention simulation results. Again, see [Barb86] for a complete description.

There are many policies that can be implemented for autonomous reassignment. In general, we divide them into two categories: *alliance* techniques and *overthrow* techniques. Consider a node x with v_x votes that becomes disconnected from the network and for purposes of the example assume all nodes detect this. Under alliance techniques all remaining nodes, say there are N of them, will take on more votes. For example, each node could increase its votes by $2v_x$ or by $2v_x/N$. An overthrow technique might use a priority scheme where the highest priority node increases its votes, perhaps by $2v_x$.

Our simulation compared various techniques with a static assignment of votes and with a group consensus technique. We based our comparison on the percentage of time the system was up, in other words, some connected group could collect a majority of votes and therefore perform restricted operations. The simulation provided a very dynamic system; in other words, failures and repairs of nodes and links occurred very frequently. This achieved a very low system uptime overall, but allowed us to zero in on just the situation where reassigning votes will help: when the network is especially volatile. We considered 5-node networks of varying connectivity. As one might expect, dynamic reassignment provided much higher availability than the static assignment did. In fact, system uptime increased by a factor of 2 or 3 under the unstable conditions. Surprisingly, autonomous techniques did not perform much worse than group consensus did. This is particularly encouraging since we think autonomous techniques are faster and easier to implement. Of the autonomous techniques, in general, alliance worked better on high connectivity networks while overthrow achieved better results in low connectivity cases.

5. Conclusions

We have presented a set of protocols that provide 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. Furthermore, the 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. Simulation results have shown that autonomous reassignment shows much improvement over a static assignment of votes and is a viable alternative to dynamically reassigning votes using a group consensus technique.

6. Acknowledgments

The authors would like to thank Rafael Alonso, Kenneth Salem and the reviewers for their many helpful comments and suggestions.

References

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

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

[Barb86] D. Barbará, H. Garcia-Molina, and A. Spauster, "Policies for Dynamic Vote Reassignment", *Proc. Sixth International Conference on Distributed Computing Systems*, May 1986, to appear.

[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.

[Davi82] S. Davidson, "Evaluation of an Optimistic Protocol for Partitioned Distributed Database Systems", Technical Report 299, Department of Electrical Engineering and Computer Science, Princeton University, May 1982.

[Garc82] 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.

[Skee82] D. Skeen, *Crash Recovery in a Distributed Database System*, PhD Thesis, ERL Memo M82/45, University of California, Berkeley, May 1982.

[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.