Elections in a Distributed Computing System

HECTOR GARCIA-MOLINA, MEMBER, IEEE

Abstract—**After a failure occurs in a distributed computing system, it is often necessary to reorganize the active nodes so that they can continue to perform a useful task. The first step in such a reorganization or reconfiguration is to elect a coordinator node to manage the operation. This paper discusses such elections and reorganizations. Two types of reasonable failure environments are studied. For each environment assertions which define the meaning of an election are presented. An election algorithm which satisfies the assertions is presented for each environment.**

Index Terms—**Crash recovery, distributed computing systems, elections, failures, mutual exclusion, reorganization.**

I. Introduction

A DISTRIBUTED system is a collection of autonomous computing nodes which can communicate with each other and which cooperate on a common goal or task [4]. For example, the goal may be to provide the user with a database management system, and in this case the distributed system is called a distributed database [16].

When a node fails or when the communication subsystem which allows nodes to communicate fails, it is usually necessary for the nodes to adapt to the new conditions so that they may continue working on their joint goal. For example, consider a collection of nodes which are processing sensory data and trying to locate a moving object [18]. Each node has some sensors which provide it with a local view of the world. The nodes exchange data and together decide where the object is located. If one of the nodes ceases to operate, the remaining nodes should recognize this and modify their strategy for locating the object. A node which neighbors the failed node could try to collect sensory data for the area which was assigned to the failed node. Another alternative would be for the remaining nodes to use a detection algorithm which is not very sensitive to "holes" in the area being studied. Or the nodes could decide to switch to such an algorithm when the failure occurs. If enough nodes fail, the remaining nodes may decide that they just cannot perform the assigned task, and may select a new or better suited job for themselves.

There are at least two basic strategies by which a distributed system can adapt to failures. One strategy is to have software which can operate continuously and correctly as failures occur and are repaired [9]. (In the previous example, this would correspond to using an algorithm which can detect the object even when there are holes in the data.) The second alternative is to temporarily halt normal operation and to take some time out to *reorganize* the system. During the reorganization period, the status of the system components can be evaluated, any pending work can either be finished or discarded, and new algorithms (and possibly a new task) that are tailored to the current situation can be selected. The reorganization of the system is managed by a *single* node called the *coordinator*. (Having more than one node attempting to reorganize will lead to serious confusion.) So as a first step in any reorganization, the operating or active nodes must *elect* a coordinator. It is precisely these elections we wish to study in this paper.

In this paper we will not study the first strategy of continuous operation. This does not mean that we think that the second strategy of reorganization is superior. Which strategy is best depends on the requirements of the application and on the failure rate. If failures are very common, it will probably not pay off to reorganize. If it is not possible to stop performing the task for even a few seconds (or milliseconds?), then clearly continuous operation is a necessity. On the other hand, algorithms for continuous system operation will in all likelihood be more complex than algorithms that can halt when a failure is encountered. Thus, operation between failures should be more efficient if reorganizations are allowed.

In this paper we discuss election protocols in the context of failures, but notice that election protocols can also be used to start up a system initially, or to add or remove nodes from the system [11]. Thus, when the nodes in the system are initially turned on, they will automatically elect a coordinator and start operating, just as if they were recovering from a failure.

The intuitive idea of an election is very natural. We have a number of nodes which talk to each other. After some deliberation among the nodes, a single node is elected the coordinator. When the election terminates, there is only one node that calls itself the coordinator, and all other nodes know the identity of the coordinator. After the election, the coordinator can start the reorganization of the system, after which normal operation can resume.

However, when one attempts to translate the natural idea of an election to a concrete algorithm for performing the election, several interesting issues arise. For example, notice that after a node is elected coordinator, some or all of its constituents may fail. So what does it mean to be coordinator if you really cannot tell who you are coordinating? How can the election protocol cope with failures during the election itself? When certain types of failures occur, it may be impossible to guarantee that a single node calls itself a coordinator. How can these cases be handled? Furthermore, in some situations (like after the communication subsystem is partitioned) we may wish to have more than one coordinator.

Manuscript received January 7, 1981; revised July 17, 1981. This work was supported in part by the National Science Foundation under Grant ECS-8019393.

The author is with the Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ 08544.

This paper will try to answer some of these questions. In Section II we study the types of failures that can affect an election. In Sections III and IV we consider two failure environments and design election protocols for these environments. In Section III we assume that the communication subsystem never fails and that nodes do not temporarily suspend their processing without a full failure. In Section IV these assumptions are relaxed. For the two environments considered, assertions are presented to define what being a coordinator means. The assertions can then be used to show that the election protocols work properly.

The election of a coordinator is basically a synchronization of parallel processes problem, and thus there is a considerable amount of related work [2], [7], [17], etc. When there are no communication failures, electing a coordinator is similar to the mutual exclusion of processes. That is, we can think of being coordinator as having entered a special critical region. In the election, the nodes (i.e., processes) compete with each other so that only one node enters the critical region and becomes coordinator. However, even if there are no communication failures there are important differences between an election protocol and a mutual exclusion protocol, and these differences will be listed in Section III. When communication failures occur, the concept of an election diverges from the mutual exclusion concept, for in a partitioned system we may not wish to prevent several nodes from becoming coordinators.

Several researchers have dealt specifically with election protocols [5], [11], [12], [19]. In these cases the election protocol is embedded in other application algorithms, so that only a very small portion of the papers is devoted to the election itself. We believe that election protocols are important enough to merit an extensive discussion. Furthermore, in this paper we concentrate on the types of failures that can be handled by the protocols and in studying the assertions that must hold throughout the elections.

II. The Assumptions

The types of failures that can occur in a distributed system play a crucial role in election protocols. Thus, a first step in understanding election protocols is to understand the types of failures that can occur in the system.

Trying to protect against all possible types of failures is impractical, so we start by stating that we will not consider certain types of failures which are either very rare or very hard to protect against. Specifically, we make these three assumptions.

Assumption 1: All nodes cooperate and use the same election algorithm.

Assumption 2: The election algorithm at each node must make use of certain software facilities. These facilities include a local operating system and a message handler. We assume that there are no "software bugs" in these facilities and that they indeed offer the proper services.

Assumption 3: If a node i receives a message M from node j, then that message M was sent by node j to node i at some earlier time. That is, we assume that the communication subsystem will not spontaneously generate messages.

There are other types of failures whose probability of occurrence can be lowered to acceptable levels with some elementary error correction and avoidance techniques. To simplify our discussion, we will assume that these techniques are in use and that we will never encounter these failures. This frees our minds to concentrate on the rest of the failure types. Specifically, we assume the following.

Assumption 4: All nodes have some (possibly limited) "safe" storage cells. Data stored in these safe cells survive any failure of the node. A failure during an update to a safe cell cannot ruin its contents. Either the update is performed correctly leaving the cell with its new value, or the update is never performed. The "state vector" which is used by the election protocol and described later is kept in safe storage. (Safe cells can be implemented by double writing to the appropiate hardware. See [10].)

Assumption 5: When a node fails (i.e., when its processor or its nonsafe memory fails), then node immediately halts all processing [10]. Some time later, the processor at the node is reset to some fixed state and execution resumes from that fixed state. (None of the previously executing processes is started on recovery. Data in nonsafe storage may be lost during such a failure.) We assume that a failure cannot cause a node to deviate from its algorithms and behave in an unpredictable fashion. Any hardware failure that does not stop a node entirely must be detected and must be converted into a "full fledged" crash before the node is able to affect any other system component. (This can be achieved by adding redundancy to the hardware and to the data structures used.)

Assumption 6: There are no transmission errors. That is, if a message M is sent and received, then the received message is precisely M. Notice that we make no assumptions as to whether M is received at all. This assumption only states that if M is received, then it is received correctly. (Transmission errors can be detected and corrected by adding redundancy to M [1].)

Assumption 7: All messages from node i to node j which do arrive at j are processed at j in the same order in which they were sent. (This can be enforced by adding sequence numbers to messages.) Again, this assumption does not imply that if node i sends messages $M1$, $M2$, $M3$ in that order, then node j has to process $M1$, $M2$, $M3$. For example, node j can simply process $M1$, $M3$ and declare $M2$ as not received.

Finally, we will make two additional assumptions. These two assumptions will make an election in a distributed system similar to our intuitive notions of an election. These assumptions also simplify the design of the election protocol. (These two assumptions will be relaxed in Section IV where we discuss elections in environments where these assumptions do not hold.)

Assumption 8: The communication subsystem does not fail. Specifically, we assume that the communication subsystem has a fixed time limit T in which it guarantees that a message will be delivered if the destination node is active. That is, if node i wishes to send a message M to node j, node i hands M to the communication subsystem. If after T seconds node i does not receive an acknowledgment from the communication subsystem stating that M was received by j, then node i knows *for certain* that node j has failed.

Assumption 9: A node never pauses and always responds to incoming messages with no delay. (This assumption is closely related to Assumption 8.) In other words, not responding "fast" to incoming messages is also considered a failure. Hence, if node j delays the receipt of message M and there is any possibility that a node somewhere may have waited T seconds without hearing about M's receipt, then node j should automatically "fail." That is, all executing processes should be halted, the state of the node reset, and the recovery procedure should be initiated.

Assumptions 8 and 9 are reasonable assumptions for certain classes of distributed systems, and thus it is interesting to study elections in such a simplified environment. For example, if the communication lines are very reliable (like the Ethernet [13]) or if the communication network has high connectivity, it seems fair to assume that the probability of a communication failure is negligible. Special hardware (e.g., timers) can be added to the nodes so that they can effectively discover any delays and convert them into failures.

Elections in an environment where Assumptions 1-9 hold will be studied in Section III. Since there are systems where it is not possible to make Assumptions 8 and 9, elections in the more general case where only Assumptions 1-7 hold will be discussed in Section IV.

To close this section we will introduce some notation to describe the components of the state vector of a node. The *state vector* of node i, $S(i)$, is a collection of safe storage cells which contain data which is crucial for the election and application algorithms. The principal components of the state vector $S(i)$ are as follows.

$S(i) \cdot s$—*The Status of Node i:* The status can be one of "Down," "Election," "Reorganization," or "Normal." When node i has stopped because of a failure, we consider $S(i) \cdot s$ = "Down" regardless of what actually is stored in that cell. As soon as node i recovers from the failure, node i is reset to a fixed state where $S(i) \cdot s$ equals "Down." Thus, $S(i) \cdot s$ will be equal to "Down" from the instant node i fails to the instant when the election begins. When node i is participating in an election, $S(i) \cdot s$ is set to "Election." If $S(i) \cdot s$ = "Reorganization," then node i is participating in a reorganization. A reorganization follows the election of the coordinator. Nodes in the "Reorganization" state know the identity of the coordinator, but do not yet know what task they will be performing. When $S(i) \cdot s$ = "Normal," node i is in normal operation working on the application task.

$S(i) \cdot c$—*The Coordinator According to Node i:* If $S(i) \cdot c = i$, then node i considers itself a coordinator.

$S(i) \cdot d$—*The Definition of the Task Being Performed:* This usually includes the application algorithms being used, a list of the participating nodes, and other state information.

III. Elections with No Communication Failures and No Node Pauses

In this section we study elections in an environment where no communication failures occur and where nodes never pause. Specifically, we assume that Assumptions 1-9 hold. These assumptions eliminate the possibility that a node may halt for an arbitrary amount of time and then resume processing where it left off as if nothing had occurred. This is turn simplifies the election protocol. We will show that in this environment it is possible to have election algorithms which guarantee that exactly one coordinator will be elected.

As an initial step, we wish to specify the rules that an election protocol must follow in order that it be considered a "good" protocol. To see what one rule should be, suppose that we have two nodes i and j, and that at some instant in time $S(i) \cdot s$ = "Reorganization" = $S(j) \cdot s$ and $S(i) \cdot c$ is not equal to $S(j) \cdot c$. Clearly, this situation is undesirable because we have two nodes that have not agreed upon the coordinator. But the situation where $S(i) \cdot s$ = "Down" or $S(j) \cdot s$ = "Down" and where $S(i) \cdot c$ is not equal to $S(j) \cdot c$ is acceptable because one of the nodes is not doing anything. So in general we have the following assertion.

Assertion 1: At any instant in time, for any two nodes i and j in the distributed system, the following must hold.

a) If ($S(i) \cdot s$ = "Reorganization" or $S(i) \cdot s$ = "Normal") and if ($S(j) \cdot s$ = "Reorganization" or $S(j) \cdot s$ = "Normal"), then $S(i) \cdot c = S(j) \cdot c$.

b) If $S(i) \cdot s$ = "Normal" and $S(j) \cdot s$ = "Normal," then $S(i) \cdot d = S(j) \cdot d$.

The fact that $S(i) \cdot d$ must equal $S(j) \cdot d$ under normal operation does not mean that all nodes are assigned exactly the same set of functions. Notice that $S(i) \cdot d$ could be of the form "If you are node i then $\cdots$, if you are node j then $\cdots$, etc." (In Assertion 1, like in the rest of the paper, we use the concept of time in an intuitive fashion. The notion of time can be formalized with the concept of logical time introduced by Lamport [8]. For example, Assertion 1 could be reworded as the requirement that the following event *not* occur in the system. Node i at logical time T_i sends a message to node j with T_i, $S(i) \cdot s$, $S(i) \cdot c$ and $S(i) \cdot d$. Node j receives the message when its logical clock reads T_j, and notices that $T_j < T_i$ and either condition a) or b) of Assertion 1 is violated.)

(This assertion is similar to the one given by [12]. However, here we require that the assertion hold at all times, not just at the end of the election. This stronger assumption avoids the problems of having some nodes believe that they are coordinators before the entire protocol has finished everywhere.)

Assertion 1 defines the consistent state vectors and should be satisfied by an election protocol. This simple assertion also tells us what it means to be a coordinator. Once node i knows that it is a coordinator (i.e., when $S(i) \cdot s$ = "Reorganization" or "Normal" and $S(i) \cdot c = i$), it knows that any other active node (i.e., nodes with "Reorganization" or "Normal" status) will recognize it as the coordinator. Furthermore, as long as node i does not fail and keeps its status of "Reorganization" or "Normal," no other node can become coordinator. But this is all that a coordinator knows. At any time, the rest of the nodes in the system may fail, so that at no time can a coordinator know for sure which nodes are active and considering it the coordinator. This means that if the coordinator has some actions that it wishes a certain set of active nodes to perform, it cannot simply issue the necessary commands, as would appear possible from the intuitive idea of a coordinator. Instead, the coordinator must use a *two phase commit* protocol [6] to guarantee that the actions are performed by all nodes in the set (or not performed by any node).

In addition to Assertion 1, there is a second condition which should be satisfied by the election protocol. Suppose that at a given time we observe that there is no coordinator in the system. We would hope that after a finite time a new coordinator would be elected. Unfortunately, this is not possible, for a string of failures could keep any election protocol from ever electing a coordinator. So instead we require that if no failures interfere with the election, then a coordinator will eventually be elected.

Assertion 2: If no failures occur during the election, the election protocol will eventually transform a system in any state to a state where:

a) there is a node i with $S(i) \cdot s$ = "Normal" and $S(i) \cdot c = i$, and

b) all other nodes j which are not failed have $S(j) \cdot s$ = "Normal" and $S(j) \cdot c = i$.

Assertions 1 and 2 define the desired characteristics of an election protocol under Assumptions 1–9. The next step is to exhibit a protocol with these properties. There are actually many such protocols. As was mentioned in the Introduction, the election of a coordinator is very similar to the mutual exclusion of parallel processes. That is, in an election the various nodes (i.e., processes) attempt to enter a special critical region. Once a node enters the region, all other nodes are excluded and the successful node becomes the coordinator. Since both the election and mutual exclusion problems are similar, many of the techniques and ideas used for mutual exclusion [2], [3], [7], [15], [17], etc., can also be applied to the design of election protocols. Nevertheless, the mutual exclusion protocols should not be used directly for elections because there are some important differences between the two problems.

1) In an election fairness is not important. An elected functioning coordinator has no need to let other nodes become coordinators. In a general mutual exclusion algorithm, all nodes should eventually be able to enter the critical region. Since the election protocol does not have to be fair, it can be simpler.

2) The election protocol must properly deal with the case of a coordinator failing. On the other hand, most mutual exclusion algorithms assume that the process in the critical region (i.e., the coordinator) will not fail.

3) A new coordinator must inform all active nodes that it is the coordinator. (That is, all $S(i) \cdot c$ must be set to identify the coordinator.) In a mutual exclusion algorithm, the nodes not in the critical region have no need to know what node is in the region.

A. The Bully Election Algorithm

The election algorithm we present here is called the *Bully Algorithm* because the node with the highest identification number forces the nodes with smaller identification numbers into accepting it as coordinator. The idea of using node identification numbers as priorities has been used in other algorithms [5], [11], [12], [19], but the specifics of our algorithm and the proof of correctness are different from the other algorithms.

In order to avoid getting lost in the details, we will leave the precise description of the election algorithm and the proof that Assertions 1 and 2 are satisfied for Appendix I. Here we will only give an informal description of the algorithm. Keep in mind that the Bully Algorithm (as well as the algorithms in [11], [12], and [19]) only works when there are no communication failures and when nodes do not pause (i.e., Assumptions 1–9 hold).

As mentioned earlier, each node in the system is assigned at system creation time a unique identification number. This identification number is used as a priority by the Bully Algorithm, so that the node with the highest priority (i.e., with the highest identification number) out of the nodes that are participating in the election will become coordinator. Let us assume that the identification numbers are between 1 and n, the number of nodes in the system.

In order to simplify the explanation, temporarily assume that once a node fails, it remains inactive forever. Thus, a failed node will never come up and will not interfere with any ongoing elections. (Later, we will study how nodes can recover from failures.) Now let us look at a node with identification number i which wishes to start an election to become coordinator. (For example, node i may have discovered that the old coordinator was down and is trying to replace it as coordinator.)

The election protocol for node i is divided into two parts. First, node i attempts to contact all nodes with higher priority, that is, all nodes with identification number j such that $i < j \leq n$, where n is the number of nodes in the system. If any of these nodes respond, then node i gives up its bid to become coordinator and waits until the node with higher priority becomes the new coordinator. (After waiting for some time without hearing from some coordinator, node i should restart the election protocol from scratch.) On the other hand, if all nodes with higher priority do not respond after the time limit of T seconds, then node i can guarantee that they have all failed. (See Assumption 8.) By our temporary assumption, these failed nodes will remain so. Thus, for $i < j \leq n$, $S(j) \cdot s$ will be "Down" and Assertion 1 will be satisfied from now on as far as the higher priority nodes are concerned.

As soon as node i completes the first part of the protocol, it has assured itself the job of new coordinator. The higher priority nodes are out of operation, while the lower priority nodes cannot complete the first part of the algorithm because node i (a higher priority node for them) is active. Thus, only node i will enter the second part of the election protocol and become coordinator. Of course, node i could fail at any point during the second part of the protocol. In this case node i loses the job of coordinator, just as any coordinator loses its job when it fails.

To inform all the lower priority nodes that it is the coordinator, node i in the second part of the election algorithm must proceed in two steps. (If only one step were used, Assertion 1 could be violated.) Since node i does not know the contents of the state vectors of the lower priority nodes, it first forces these nodes into "similated failures." By sending a "halt" message to node k, node i makes node k stop any processing and set $S(k) \cdot s$ to "Election." Once all nodes are in a known state, the state vectors can be modified without ever violating Assertion 1. So in the second step (of the second part of the protocol), node i sends out "I am elected" messages to the lower priority nodes. When a node k receives such a message, it sets $S(k) \cdot c$ to i and then sets $S(k) \cdot s$ to "Reorganization." (Before node k processes the "I am elected" message from node i, it checks that the last "halt" message was also sent by node i.) Now that the election is over, node i also sets $S(i) \cdot c$ to i and $S(i) \cdot s$ to "Reorganization." During reorganization, node i distributes the new algorithms to the nodes (i.e., $S(k) \cdot d$) and all status are changed to "Normal."

In the description of the protocol so far, we have assumed that failed nodes did not recover. Clearly, nodes should recover and we need a recovery strategy that does not destructively interfere with an ongoing election. There are various options for this, but it turns out that the simplest one is to let a recovering node attempt to become the coordinator using the same protocol we have described. It all works out nicely, for a recovering node, in following the election protocol, will halt all lower priority nodes which may be in the process of becoming coordinators. So the algorithm we described for the case where nodes never recover is actually the complete Bully Algorithm.

Theorem 1: In an environment where Assumptions 1–9 hold, the Bully Algorithm (whose details are given in Appendix I) satisfies Assertions 1 and 2.

Informal Proof: See Appendix I. □

IV. Elections with Communication Failures and Node Pauses

In the previous section we discussed distributed system elections under some strong failure assumptions. We assumed that the communication subsystem never failed (Assumption 8) and that nodes never paused (Assumption 9). In this section we will study elections in the more general environment where only Assumptions 1–7 hold. We will find that in this environment it is necessary to redefine the meaning of an election.

Once Assumptions 8 and 9 are relaxed, there are many types of failures which can take place. To illustrate some of the possible failures which we wish the election protocol to handle, we will give some examples. The nodes in the system may be partitioned into two or more groups of isolated nodes, where the nodes within a group can communicate with each other but are unable to communicate with nodes in other groups. Some nodes may be able to send out messages but not receive them, or vice versa. Two nodes i and j may be able to communicate with a third node, but at the same time nodes i and j may be unable to communicate with each other. A node may stop its processing at any time for an arbitrary period of time, and then resume its processing exactly where it left off as if nothing had happened, and so on.

With such failures it is impossible to elect a coordinator in the sense of Assertions 1 and 2. That is, a node can never guarantee that it is the only node which considers itself a coordinator. For example, after a system partition, the coordinator of one group of nodes cannot (and may not wish to) force other groups of nodes into not having other coordinators. We thus have the following result.

Theorem 2: In a distributed system which does not satisfy Assumptions 8 and 9, there is no election protocol which satisfies Assertions 1 and 2.

Informal Proof: If either one of the assumptions does not hold, we may have a node i in the system with $S(i) \cdot s$ = "Normal" and $S(i) \cdot c = i$ which does not respond to the messages from the other nodes (because of a communication failure or because node i has simply paused for a while). From the point of view of the rest of the nodes, node i may have set $S(i) \cdot s$ to "Down," so to satisfy Assertion 2 they must elect a new coordinator. But if a new coordinator is elected, Assertion 1 will be violated. □

If we wish to have elections in an environment where Assumptions 8 and 9 do not hold, we must redefine what is meant by an election and by a coordinator. A coordinator cannot be unique throughout the system, so we think of electing a coordinator only among the nodes which can communicate with each other. Unfortunately, the term "nodes which can communicate with each other" is rather ill defined, for such a group can change as failures occur and are repaired. So how can we tell who a coordinator is coordinating? There is also the problem of dealing with nodes which "migrate" from one group of nodes to another group. That is, suppose that we have a group of nodes which all think that node i is their "local" coordinator. At any point in time, a new node might pop into this group from another group without the new node realizing that it has switched groups. The new node should not be allowed to operate in the group coordinated by node i without some type of new election and reorganization.

These problems can be solved in a simple way if we introduce the notion of group number for a group of nodes. Any group of nodes that has somehow gotten together to elect a coordinator and works on a common task or goal is identified with a unique *group number.* All messages exchanged by the nodes of a group contain the group number, so that a node can simply ignore messages that were generated by foreign groups. For this to work, all group numbers must be unique and nodes cannot forget their current group number. (As we will see later, not all messages from foreign groups can be ignored.)

Let us assume that node i keeps its group number in its state vector $S(i)$. We will call the group number of node i $S(i) \cdot g$. Using group numbers, we can state the assertion that must be satisfied by the election protocols. All nodes in the same group should have the same coordinator. More specifically, we have the following assertion.

Assertion 3: At any instant in time, for any two nodes i and j in the distributed system, the following must hold.

a) If ($S(i) \cdot s$ = "Reorganization" or $S(i) \cdot s$ = "Normal") and ($S(j) \cdot s$ = "Reorganization" or $S(j) \cdot s$ = "Normal") and $S(i) \cdot g = S(j) \cdot g$, then $S(i) \cdot c = S(j) \cdot c$.

b) If $S(i) \cdot s = S(j) \cdot s$ = "Normal" and $S(i) \cdot g = S(j) \cdot g$, then $S(i) \cdot d = S(j) \cdot d$.

Notice that Assertion 1 can be obtained from the new assertion if we think of all nodes belonging to a single fixed group. Assertion 3 (like Assertion 1) defines the meaning of being a coordinator. In an environment where Assumptions 8 and 9 hold, a coordinator can be sure that any node which does not consider it the coordinator is down. Now that these assumptions do not hold, a coordinator knows even less than before. All a coordinator can tell is that any nodes which consider themselves part of its group must consider it the coordinator. But nodes in the group are free to change groups at any time, leaving the coordinator with little to coordinate.

We also need an assertion similar to Assertion 2 that tells us that a coordinator should be elected in any group of nodes, unless failures interfere. Before stating the new assertion, we must decide whether nodes which only have one way communication or which can only communicate with a subset of the nodes in the group should be required to be members of the group. Since it is very hard to carry on an election with nodes which can only receive or only send messages, or with nodes which cannot receive or send mesages to certain nodes, we choose not to make any requirements for these nodes. These "partially failed" nodes may or may not end up with the same group number and coordinator as the other nodes in the group. Thus, we have the following assertion.

Assertion 4: Suppose that we have a set of operating nodes R which all have two way communication with all other nodes in R. That is, for the nodes in R Assumptions 8 and 9 hold. Also, assume that there is no superset of R with this property. If no node failures occur during the election, the election algorithm will eventually transform the nodes in set R from any state to a state where:

a) there is a node i in R with $S(i) \cdot s =$ "Normal" and $S(i) \cdot c = i$,

b) and for all other nodes j in R, $S(j) \cdot s =$ "Normal," $S(j) \cdot c = i$ and $S(j) \cdot g = S(i) \cdot g$.

As in Section III, we have no fairness requirement for the election protocols. We do not care which node is elected coordinator, and the coordinator may remain coordinator as long as it pleases.

A. The Invitation Election Algorithm

We will now present an electron algorithm which satisfies Assertions 3 and 4 in an environment where only Assumptions 1-7 hold. A description and proof of correctness of the protocol is given in Appendix II. Here we will give an informal description of the algorithm.

In the Bully Algorithm (of Section III-A) the active node with the highest priority "forced" all other nodes to refrain from trying to become coordinator, so that the highest priority node was assured success in the election. In our current failure environment such an approach simply does not work because several nodes may believe that they are the highest priority node. So instead of forcing nodes into a given state, nodes who wish to become coordinator will "invite" other nodes to join it in forming a new group. A node receiving an invitation can accept or decline it. When a node accepts an invitation, it modifies its state vector to join the new group.

To simplify the algorithm, we can make a receiving node form a new group with itself the coordinator and only member. Then the entire election protocol boils down to the protocol for merging or combining various groups of nodes into a new group. To illustrate what we mean, consider the following example. A group of 6 nodes is operating with node 6 as coordinator. Then node 6 crashes, leaving the other five nodes able to communicate with each other without a coordinator.

As each of the five nodes realizes that the coordinator is down, they each form a new group. The coordinator of each group periodically looks around the system to see if there are other groups which might want to join it. To prevent all group coordinators from sending out invitations at once, a priority mechanism is used so that lower priority nodes defer sending out their invitations for a longer period of time. This, of course, does not really prevent all conflicts; it only reduces the probability of a conflict. So, in the example let us assume that both nodes 4 and 5, believing that they are each the coordinator with highest priority, send out invitations to the rest of the coordinators. Say that node 3 accepts the invitation of node 4, while nodes 1 and 2 accept the invitation of node 5. Then two new groups would form (with new group numbers): one group with nodes 3 and 4 and node 4 coordinating, and a second group with node 5 coordinating nodes 1, 2, and 5. Sometime later, one of the two coordinators invites the other one to join its group, so the system of five operating nodes ends up being in a single group with a single coordinator.

Group numbers allow the system to evolve in a simple and well-defined fashion. They identify the groups as well as all messages which emanate from the groups. The only property needed by group numbers is uniqueness. The group numbers can be generated by each new coordinator as follows. Every node has a unique identification number, as well as a counter for the number of new groups generated by the node. (The counter and node identification are kept in safe storage. See Assumption 4.) A new group number is generated by incrementing the counter by one and appending the node identification to the result. (We can think of group numbers as timestamps [8], [20] for the group generation time.)

The two main ideas in the Invitation Algorithm are that recovering nodes form new single node groups, and that coordinators periodically try to combine their group with other groups in order to form larger groups. The details of the algorithm are given in Appendix II.

Theorem 3: In an environment where Assumptions 1-7 are satisfied, the Invitation Algorithm satisfies Assertions 3 and 4.

Informal Proof: See Appendix II. □

B. Comments on the Invitation Algorithm

There are many variations and improvements on the basic Invitation Algorithm. For example, a coordinator could decline invitations if it does not trust the new coordinator or if it is simply "not interested" in forming a new group. Most of these variations are straightforward and will not be discussed here. We believe that this algorithm can also be used in other failure environments. For example, it might be better to use the Invitation Algorithm even when Assumptions 8 and 9 hold in the system because this algorithm is more general and not much more complicated than the Bully Algorithm.

In some applications certain operations should not be performed simultaneously by isolated groups of nodes. For example, in a distributed database with replicated data, we may not want isolated groups of nodes to at the same time update their copy of the replicated data because the data will then diverge [16]. One common way to enforce this restriction is to require that if a group is going to perform the restricted operation, it must have as members a majority of the nodes in the entire system (where the total numberof nodes is known and constant). This guarantees that at most one group performs the restricted operation.

Restricting some operations is part of the reorganization phase which follows the election of the coordinator. That is, if there are some restricted operations, a newly elected coordinator should count the members of the group and decide if the operation is allowed. The allowed operations and application algorithms are then distributed to the member nodes to be stored in $S(i) \cdot d$. By the way, notice that by requiring a majority of nodes it is possible to end up with a system where no group of nodes is allowed to perform the restricted operation. Also, notice that the restricted operation (as all other operations) must be performed with a two phase commit protocol [6] to ensure that the group still contains a majority of nodes when the operation is actually performed.

In some applications it is also necessary to keep track of the history of the groups that were formed. A history for a group of nodes contains the group number, the identification number of the coordinator and all member nodes, plus a history of the previous groups that the nodes participated in. Thus, the history of a group forms a graph with groups as vertices and going back to the initial group that created the system. A history graph for a new group can be compiled by the coordinator and distributed to the member nodes for safekeeping (in $S(i) \cdot d$). The history information can be useful during reorganization time in order to "clean up" pending or conflicting work of the previous groups. For example, in a distributed file system histories can be used to detect conflicting modifications to files [14].

V. Conclusions

In this paper we discussed elections in distributed computing systems. We discovered that the meaning of an election depends on the types of failures that can occur, so we studied elections in two representative failure environments. For each environment we postulated assertions which define the concept of an election and which must be satisfied by election algorithms. An election algorithm for each of the environments was also presented.

An election algorithm is an important component of a crash resistant distributed system, but it is just one of the many such components. The election algorithm interacts with the rest of the application components in various ways, and it is impossible to design a complete election algorithm without designing the rest of the system. In this paper we only studied the basic concepts and alternatives for election algorithms, and much more work is required before these ideas can be properly integrated into a complete system.

Appendix I

The Bully Election Algorithm

This appendix describes the Bully Election Algorithm which operates in an environment where Assumptions 8 and 9 hold. Before giving the details of the algorithm, we will briefly describe the message passing and operating system facilities available at each node in the distributed system.

The main way in which users of the system interact with the local operating system is with the CALL statement. The format of this statement is

CALL proc (i, parameters), ONTIMEOUT (t): *stmt*.

Execution of this statement by a process causes procedure "proc" to be executed at node "i" with the given parameters. This possibly remote procedure invocation is managed by the operating system. If node i is the local node, then the statement is interpreted as a standard procedure call. If node i is a remote node, then the operating system suspends the calling process and sends out a messsage to node i requesting the execution of the procedure. When the remote node receives the message, it schedules a process to execute procedure "proc." (The scheduled process is executed concurrently with other processes at the remote node.) When the process completes, an acknowledgement message is sent back ot the originating node with any results produced. (All parameters are value result.) The local operating system then causes the calling process to resume execution after the call statement, just as if the call had been a local procedure call.

While the calling process is suspended, the local operating system keeps track of the elapsed time. If "t" seconds go by without an acknowledgment from node "i," then the calling process is started up and statement "*stmt*" is executed. (A response to the call statement which arrives after the timeout occurred is ignored.)

There are some procedure calls which are handled differently. These are calls to *immediate* procedures. From the point of view of the calling operating system, an immediate call is handled the same way. When a remote node i receives a message to execute an immediate procedure, the node i operating system itself executes the requested procedure, instead of creating a new process to execute it. With an immediate procedure, the communication subsystem and the node i operating system guarantee that a response will be received by the calling node within a maximum of T seconds. (No such guarantee is made with the other procedures.) Thus, if a process P executes the statement

CALL *foo*(i, *parms*), ONTIMEOUT (T): s

where "*foo*" is an immediate procedure and if statement "s" is executed, then process P knows for sure that node i is down. (See Assumptions 8 and 9.) To make it possible for the the operating system to perform immediate procedures fast, these procedures should be short and should not include CALL statements.

Procedures which can be invoked with a CALL statement are defined with the declaration:

```
[IMMEDIATE] PROCEDURE proc (i, parameters);
    Procedure code;
```

The first parameter i is the node at which the procedure is being executed.

The application procedures (i.e., everything other than the operating system and the election procedures) can also use the CALL statement. However, the operating system delays execution of all application procedures until the status of the node $S(i) \cdot s$, is "Normal."

(Lampson and Sturgis [10] describe how a call statement similar to the one we have described can be implemented in a system where Assumptions 4 and 5 hold. The main idea is that the operating system must uniquely number all outgoing call messages so that the responses can be identified when they return. The counter for numbering procedure calls must be kept in a safe cell.)

The operating system also provides a facility to simulate a node failure. This facility is invoked by the statement

CALL STOP.

When this statement is executed by a procedure P, the operating system stops the execution of all processes. The process running procedure P is allowed to finish P and then is also halted. The scheduler is reset to indicate that no processes exist at the node. The stop statement does not affect the operating system. It continues to receive and process messages. After the execution of a stop statement, new processes can be stated up, either be remote nodes (through CALL statements) or by procedures Check and Timeout which are next described.

Periodically, the operating system calls a special procedure called *Check.* This procedure checks the status of the distributed system, and if anything is "wrong" starts up an election. The code for this procedure is given later with the rest of the election algorithm.

Similarly, when the operating system notices that a certain predefined amount of time goes by without hearing from the coordinator, procedure *Timeout* is automatically started up. This procedure will try to contact the last known coordinator, and if it cannot, it will take appropriate action.

When a node i fails, variable $S(i) \cdot s$ is set to "Down" and all processing stops. We assume that data in the state vector $S(i)$ is not lost due to the failure. When a node recovers, the operating system is started up, and a process is started up to execute procedure *Recovery.* Procedure Recovery can also be invoked by the application programs when they discover that the coordinator is down.

As discussed in the main body of this paper, the state vector at node i, $S(i)$ contains a status indicator $S(i) \cdot s$, the identity of the coordinator $S(i) \cdot c$, and the definition of the task being performed $S(i) \cdot d$. We also include the following in the state vector.

$S(i) \cdot h$: This is the identity of the last node which caused node i to halt. This identity is used to distinguish messages from the node which is currently trying to become coordinator from messages of nodes which previously tried.

$S(i) \cdot Up$: A set of node identification numbers. These are the nodes which node i believes to be in operation.

Since processes at a node can be executed concurrently, we must be careful about interference among processes which manipulate the state vector $S(i)$. We will assume that there is a mutual exclusion mechanism controlling access to the state vector of a node. In the election algorithm the code in the critical regions where the state vector is accessed will be enclosed in double brackets [[,]]. In other words, when a process is executing code within these brackets, we assume that any other processes wishing to enter their critical regions are delayed. If a CALL STOP statement is executed while a process P is in its critical region, then the operating system waits until P exits the critical region before suspending it. After a node failure, the mutual exclusion mechanism is reset so that new processes may enter their critical regions.

We now present the procedures that make up the Bully Algorithm. The procedures are given in an informal Algol-like language. Comments are enclosed in double angular brackets ⟨⟨, ⟩⟩.

```
IMMEDIATE PROCEDURE AreYouThere($i$);
  BEGIN
  ⟨⟨This procedure is used by remote nodes to discover if node
    $i$ is operating. Nothing is actually done at node $i$.⟩⟩
  END AreYouThere;

IMMEDIATE PROCEDURE AreYouNormal($i$, answer);
  BEGIN
  ⟨⟨This procedure is called by coordinators to discover the
    state of a node.⟩⟩
  [[IF $S(i) \cdot s$ = "Normal" THEN answer := "Yes"
    ELSE answer := "No";]]
  END AreYouNormal;

IMMEDIATE PROCEDURE Halt($i, j$);
  BEGIN
  ⟨⟨When node $i$ receives this message, it means that node $j$
    is trying to become coordinator.⟩⟩
  [[$S(i) \cdot s$ := "Election"; $S(i) \cdot h := j$;
  CALL STOP;]]
  END Halt;

IMMEDIATE PROCEDURE NewCoordinator($i, j$)
  BEGIN
  ⟨⟨Node $j$ uses this procedure to inform node $i$ that node $j$
    has become coordinator.⟩⟩
  [[IF $S(i) \cdot h = j$ AND $S(i) \cdot s$ = "Election" THEN
    BEGIN
    $S(i) \cdot c := j$; $S(i) \cdot s$ := "Reorganization";
    END;]]
  END NewCoordinator;

IMMEDIATE PROCEDURE Ready($i, j, x$);
  BEGIN
  ⟨⟨Coordinator $j$ uses this procedure to distribute the new
    task description $x$.⟩⟩
  [[IF $S(i) \cdot c = j$ AND $S(i) \cdot s$ = "Reorganization" THEN
    BEGIN
    $S(i) \cdot d := x$; $S(i) \cdot s$ := "Normal";
    END;]]
  END Ready;

PROCEDURE Election($i$);
  BEGIN
  ⟨⟨By executing this procedure, node $i$ attempts to become
    coordinator. First step is to check if any higher priority
    nodes are up. If any such node is up, quit.⟩⟩
  FOR $j := i + 1, i + 2, \cdots, n - 1, n$ DO
    BEGIN ⟨⟨$n$ is number of nodes in system.⟩⟩
    CALL AreYouThere($j$), ONTIMEOUT($T$):
            NEXT ITERATION;
        RETURN;
        END;
⟨⟨Next, halt all lower priority nodes, starting with this
    node.⟩⟩
[[CALL STOP;
  $S(i) \cdot s$ := "Election"; $S(i) \cdot h := i$; $S(i) \cdot$ Up := { };]]
FOR $j := i - 1, i - 2, \cdots, 2, 1$ DO
    BEGIN
    CALL Halt ($j, i$), ONTIMEOUT($T$) : NEXT ITERATION;
    [[$S(i) \cdot$ Up := {$j$} UNION $S(i) \cdot$ Up;]]
    END;
⟨⟨Now node $i$ has reached its "election point." Next step is to
    inform nodes of new coordinator.⟩⟩
[[$S(i) \cdot c := i$; $S(i) \cdot s$ := "Reorganization";]]
FOR $j$ IN $S(i) \cdot$ Up DO
    BEGIN
    CALL NewCoordinator($j, i$), ONTIMEOUT($T$):
        BEGIN
        CALL Election ($i$); RETURN;
        ⟨⟨Node $j$ failed. The simplest thing is to restart.⟩⟩
        END;
    END;
⟨⟨The reorganization of the system occurs here. When done,
    node $i$ has computed $S(i) \cdot d$, the new task description. ($S(i)$
    $\cdot d$ probably contains a copy of $S(i) \cdot$ Up.) If during the
    reorganization a node in $S(i) \cdot$ Up fails, the reorganization
    can be stopped and a new reorganization started.⟩⟩
FOR $j$ IN $S(i) \cdot$ Up DO
    BEGIN
    CALL Ready ($j, i, S(i) \cdot d$), ONTIMEOUT ($T$):
        BEGIN
        CALL Election ($i$); RETURN;
        END;
    END;
[[$S(i) \cdot s$ := "Normal";]]
END Election;


PROCEDURE Recovery($i$);
    BEGIN
    ⟨⟨This procedure is automatically called by the operating
        system when a node starts up after a failure.⟩⟩
    $S(i) \cdot h$ := UNDEFINED; ⟨⟨e.g., set it to $-1$⟩⟩
    CALL Election($i$);
    END Recovery;


PROCEDURE Check($i$);
    BEGIN
    ⟨⟨This procedure is called periodically by the operating
        system.⟩⟩
    IF [[$S(i) \cdot s$ = "Normal" AND $S(i) \cdot c = i$]] THEN
        BEGIN ⟨⟨I am coordinator. Check if everyone else is
        normal.⟩⟩
        FOR $j := 1, 2, \cdots, i - 1, i + 1, \cdots, n$ DO
            BEGIN
            CALL AreYouNormal($j$, $ans$), ONTIMEOUT($T$) :
                                NEXT ITERATION;
            IF $ans$ = "No" THEN
                BEGIN CALL Election($i$); RETURN; END;
            END;
        END;
    END Check;

PROCEDURE Timeout($i$);
    BEGIN
    ⟨⟨This procedure is called automatically when node $i$ has
        not heard from the coordinator in a "long" time.⟩⟩
    IF [[$S(i) \cdot s$ = "Normal" OR $S(i) \cdot s$ "Reorganization']]
        THEN
        BEGIN
        ⟨⟨Check if coordinator is up.⟩⟩
        CALL AreYouThere($S(i) \cdot c$), ONTIMEOUT($T$) :
                        CALL Election($i$);
        END
    ELSE CALL Election($i$);
    END Timeout;
```

Theorem A1: The Bully Algorithm under Assumptions 1-9 satisfies Assertion 1.

Informal Proof: (As we mentioned earlier, we use the concept of time in an intuitive fashion. This proof can be formalized with the ideas in [8].) We divide the proof into two parts. First, we show that part a) of Assertion 1 holds then we show part b).

a) We must show that any two nodes which are in a "Reorganization" or "Normal" state must consider the same node coordinator. Notice that this constraint can only be violated in the Bully Algorithm when a node i switches its state to "Reorganization" with a new value for $S(i) \cdot c$. (When a node switches to "Normal" state, the coordinator $S(i) \cdot c$ is not changed, so this transition is not important here.) We will now show by contradiction that a node i making the critical transition will never violate the constraint in part a) of Assertion 1.

Suppose that there is a node i which changes $S(i) \cdot s$ to "Reorganization" with $S(i) \cdot c = j$ ($j \geq= i$) at some time t_1. At that same time there is another node k with $S(k) \cdot s$ equals to "Reorganization" or "Normal" and with $S(k) \cdot c = m$, where m is different from j ($m \geq= k$).

Since node i is making $S(i) \cdot c$ equal to j, node j must have successfully reached an "election point" in the immediate past. (The election point of node j is when that node completes the second FOR loop in procedure Election.) Let us look at the election of node j which is closest to time t_1. Going back in time from time t_1, say that at time t_2 node reached its election point, at time t_3 node i was halted (with procedure Halt) by node j (setting $S(i) \cdot h = j$), and at time t_4 node j started the election protocol (where $t_4 < t_3 < t_2 < t_1$).

Since we are looking at the last election of j before t_1, node j did not fail and was not halted (by procedure Halt) between times t_4 and t_2. We also know that between times t_3 and t_1 node i did not fail and did not receive any other Halt messages (else $S(i) \cdot h$ would have been reset and the t_1 transition could not have occurred). Also, notice that at some time t_m between times t_4 and t_2, node j assured itself that node m was halted. (If $m > j$, node j makes sure that node m is down. If $m \leq j$, then node j sends a halt message to node m.)

In addition to the election of node j, node m was also being elected. Let t_e be the time at which the election point of the last m election occurred. Notice that t_e occurred before the current time t_1, else node k would not have $S(k) \cdot c = m$. That is, $t_e < t_1$. Now consider the following two cases.

Case 1 [*of Part a*)]: $m > i$. In this case $t_m < t_3$. (See algorithm.) Node m also sends a Halt message to node i, but this must have occurred before t_3 because node i did not receive Halt messages after t_3. Since $j \geq i$, node m assured itself that j was down also before time t_3. That is, some time between t_m and t_3 node j was halted. But this is a contradiction because the period between t_m and t_3 is included in the time between t_4 and t_2 in which we know that node j did not halt.

Case 2 [*of Part a*)]: $m \leq i$. In this case $t_m > t_3$. This implies that the election of m took place entirely between the times t_3 and t_1. This in turn means that at some point in this interval node m made sure that node i had failed. But if node i failed between t_3 and t_1, then $S(i) \cdot h$ was changed to a value different from j, and the transition at time t_1 did not take place. This is a contradiction. (If $j = i$, $t_1 = t_2$ and node m also prevents the t_1 transition from occurring. If $m = i$, the transition at time t_1 does not take place either.)

b) The proof of part b) of Assertion 1 is also by contradiction. Suppose that at time t_1 a node i with $S(i) \cdot c = j$ and $S(i) \cdot d = x$ switches its state to $S(i) \cdot s$ = "Normal." Also, suppose that at the same time a second node k (k different from i) has $S(k) \cdot s$ = "Normal," $S(k) \cdot c = m$, and $S(k) \cdot d = y$. By part a) of this proof, $m = j$. This means that both x and y were distributed by the same coordinator j in different elections. Without loss of generality, say that y was distributed after the second j election. Before y is sent out by calls to procedure Ready, node j resets all nodes to either "Down" or "Election" states, so that any node like node k with the old value of x must not be in "Normal" state. This is a contradiction. (Recall that messages between node j and other nodes are delivered in the order in which they were sent. See Assumption 7.) □

Theorem A2: The Bully Algorithms under Assumptions 1-9 satisfies Assertion 2.

Informal Proof: Take a system with a set of operating nodes and a set of failed nodes and assume that no further failures occur. Let node i be the operating node with highest priority.

If node i is not the coordinator already, it will try to become one. (If node i considers node j the coordinator, node j must be of higher priority than i and thus j must be down. When procedure Timeout is called, node i will realize this and start an election.) Since node i has the highest priority, its bid to become coordinator will be successful. Therefore, the system will eventually reach a state where $S(i) \cdot s$ = "Normal" and $S(i) \cdot c = i$. By following the Bully Algorithm we see that all other nodes j will have $S(j) \cdot s$ = "Normal" and $S(j) \cdot c = i$.

If node i already was the coordinator to begin with, then there might be some nodes k in the system with $S(k) \cdot s$ = "Election" or "Reorganization." This situation will be detected by coordinator i when procedure Check is called next. By calling another election, node i will correct the situation and take the system to the desired state. □

APPENDIX II

THE INVITATION ELECTION ALGORITHM

This appendix describes the Invitation Election Algorithm, which operates in an environment where Assumptions 1-7 hold. The operating system and message passing facilities available to the algorithm are similar to the ones used by the Bully Algorithm of Appendix I. However, there are a few differences which we now discuss.

When a remote CALL statement times out after waiting T seconds, the calling process can now make no inferences about the state of the nonresponding node. The value of T is now chosen so that if a calling process waits this amount of time without a response, then it is likely that the called node is down. For the Invitation Algorithmn we do not use immediate procedures since they are of little use in this environment.

Nodes should not normally respond to foreign call statements from nodes with a different group number than the local group number. To enforce this, the group number of the calling node is appended to each call message and checked by the receiving node. However, in the election algorithm a node sometimes has to respond to messages originating in different groups. Therefore, in the description of the election procedures we will assume that the group number checking mechanism is *not* applied to the election procedures. If a group number check is needed in these election procedures, the group number will be included as a parameter in the CALL statement and explicitly checked by the called procedure. (As before, we assume that application procedures are only executed at a node when its status is "Normal.")

As in Appendix I, we assume that procedure Timeout is automatically invoked by the operating system after a period of no coordinator communication. Similarly, procedure Check is called periodically and procedure Recovery is called when a node recovers from a crash.

As discussed in the main body of the paper, the state vector of each node i includes $S(i) \cdot g$, the group number of the node. The state vector also contains $S(i) \cdot$ Up, the list of nodes which are participating in the current group. ($S(i) \cdot$ Up can be considered to be a part of $S(i) \cdot d$, the task description.) A counter $S(i) \cdot$ counter which keeps track of the groups generated at node i is also included in the state vector. We assume that the data in the state vector $S(i)$ is not lost due to a crash.

We now present the procedures that make up the Invitation Algorithm in the same informal Algol-like language. Code in the critical region for the state vector, as before, is enclosed in double brackets [[,]]. Comments are enclosed in double angle brackets ⟨⟨ ⟩⟩.

```
PROCEDURE Check ($i$);
    BEGIN
⟨⟨This procedure is called periodically by the operating
  system.⟩⟩
  IF [[$S(i) \cdot s$ = "Normal" AND $S(i) \cdot c = i$]] THEN
    BEGIN ⟨⟨See if other groups exist for possible
    merge.⟩⟩
    TempSet := { }; ⟨⟨i.e., the empty set⟩⟩
    FOR $j := 1, 2, \cdots, i-1, i+1, \cdots, n$ DO
      BEGIN ⟨⟨$n$ is the number of nodes in the system⟩⟩
      CALL AreYouCoordinator($j$, $ans$), ONTIMEOUT($T$):
        NEXT ITERATION;
        IF $ans$ = "Yes" then TempSet := TempSet UNION
                $\{j\}$;
        END;
    IF TempSet = { } THEN RETURN;
    $p$ := MAXIMUM (TempSet);
    IF $i < p$ THEN "wait time proportional to $p - i$";
    ⟨⟨The above statement gives the node with highest pri-
      ority a chance to organize a merge first. This wait time
      should be large as compared to the time between calls
      to procedure Check.⟩⟩
    CALL Merge ($i$, TempSet);
    END;
  END Check;

PROCEDURE Timeout ($i$);
  BEGIN
  ⟨⟨This procedure is called when node $i$ has not heard from
    its coordinator in some time.⟩⟩
  [[MyCoord := $S(i) \cdot c$; MyGroup := $S(i) \cdot g$]]
  IF MyCoord = $i$ THEN RETURN
  ELSE
    BEGIN
    CALL AreYouThere(MyCoord, MyGroup, $i$, $ans$),
        ONTIMEOUT ($T$) : CALL Recovery ($i$);
    IF $ans$ = "No" THEN CALL Recovery ($i$);
    END;
  END Timeout;

PROCEDURE Merge ($i$, CoordinatorSet);
  BEGIN
  ⟨⟨This procedure will form a new group with node $i$ as co-
    ordinator and will invite coordinators in CoordinatorSet
    to join in.⟩⟩
  [[$S(i) \cdot s$ := "Election";
    ⟨⟨Above statement will prevent node $i$ from accepting
    invitations while this procedure is executed.⟩⟩
    CALL STOP;
    $S(i) \cdot$ counter := $S(i) \cdot$ counter + 1;
    $S(i) \cdot g$ := $i$ CONCATENATED $S(i) \cdot$ counter;
    $S(i) \cdot c$ := $i$; TempSet := $S(i) \cdot$ Up; $S(i) \cdot$ Up := { };]]
  FOR $j$ IN CoordinatorSet DO
    CALL Invitation ($j, i, S(i) \cdot g$), ONTIMEOUT ($T$) :
                NEXT ITERATION;
  FOR $j$ IN TempSet DO
    CALL Invitation ($j, i, S(i) \cdot g$), ONTIMEOUT ($T$) :
                NEXT ITERATION;
  "Wait a reasonable time for accept messages to come in"
  [[$S(i) \cdot s$ := "Reorganization";]]
  ⟨⟨The reorganization of the group with members $S(i) \cdot$ Up
    occurs here. When done, node $i$ has computed $S(i) \cdot d$,
    the new task description (which probably includes a copy
    of $S(i) \cdot$ Up). If during the reorganization a node in $S(i)$
    $\cdot$ Up does not respond, the reorganization can be stopped
    and a new reorganization attempted.⟩⟩
  FOR $j$ IN $S(i) \cdot$ Up DO
    CALL Ready ($j, i, S(i) \cdot g, S(i) \cdot d$), ONTIMEOUT ($T$) :
        CALL Recovery ($i$);
  [[$S(i) \cdot s$ := "Normal";]]
  END Merge;

PROCEDURE Ready ($i, j, gn, x$);
  BEGIN
  ⟨⟨Coordinator $j$ of group $gn$ is giving node $i$ the new task
    description $x$.⟩⟩
  [[IF $S(i) \cdot s$ = "Reorganization" AND $S(i) \cdot g = gn$
    THEN
    BEGIN
    $S(i) \cdot d := x$; $S(i) \cdot s$ := "Normal";
    END;]]
  END Ready;

PROCEDURE AreYouCoordinator($i$, $ans$);
  BEGIN
  ⟨⟨The calling node wishes to know if node $i$ is a coordinator
    in normal state, so that it can be invited to join a new
    group.⟩⟩
  [[IF $S(i) \cdot s$ = "Normal" AND $S(i) \cdot c = i$ THEN $ans$ :=
    "Yes"
    ELSE $ans$ := "No";]]
  END AreYouCoordinator;

PROCEDURE AreYouThere ($i, gn, j, ans$);
  BEGIN
  ⟨⟨Node $j$ wishes to find out if node $i$ is the coordinator of
    group $gn$ and considers node $j$ a member of the
    group.⟩⟩
  [[IF $S(i) \cdot g = gn$ AND $S(i) \cdot c = i$ AND $j \in S(i) \cdot$ Up THEN
    $ans$ : = "Yes"
    ELSE $ans$ := "No";]]
  END AreYouThere;

PROCEDURE Invitation ($i, j, gn$);
  BEGIN
  ⟨⟨Node $j$ invites node $i$ to join group $gn$.⟩⟩
  [[IF $S(i) \cdot s \neq$ "Normal" THEN RETURN; ⟨⟨decline invi-
    tation⟩⟩
    CALL STOP; Temp := $S(i) \cdot c$; TempSet := $S(i) \cdot$ Up;
    $S(i) \cdot s$ := "Election"; $S(i) \cdot c := j$; $S(i) \cdot g := gn$;]]
  IF Temp = $i$ THEN ⟨⟨forward invitation to my old mem-
    bers⟩⟩
    FOR $k$ IN TempSet DO
      CALL Invitation ($k, j, gn$), ONTIMEOUT($T$) :
                NEXT ITERATION;
  CALL Accept ($j, i, gn$), ONTIMEOUT ($T$) :
                CALL Recovery ($i$) ;
  [[$S(i) \cdot s$ := "Reorganization";]]
  END Invitation;

PROCEDURE Accept ($i, j, gn$);
  BEGIN
  ⟨⟨Node $j$ has accepted the invitation of node $i$ to join group
    $gn$.⟩⟩
  [[IF $S(i) \cdot s$ = "Election" AND $S(i) \cdot g = gn$ AND $S(i) \cdot c$
    $= i$ THEN
    $S(i) \cdot$ Up := $S(i) \cdot$ Up UNION $\{j\}$;]]
  END Accept;

PROCEDURE Recovery ($i$);
  BEGIN
  ⟨⟨Node $i$ is recovering from a failure.⟩⟩
```

```
[[$S(i) \cdot s$ := "Election"; CALL STOP;
    $S(i) \cdot$ counter := $S(i) \cdot$ counter + 1;
    $S(i) \cdot g$ := $i$ CONCATENATED $S(i) \cdot$ counter;
    $S(i) \cdot c$ := $i$; $S(i) \cdot$ Up := { };
    $S(i) \cdot s$ := "Reorganization";
    ⟨⟨A single node task description is computed and placed
        in $S(i) \cdot d$.⟩⟩
    $S(i) \cdot s$ := "Normal";]]
END Recovery;
```

Theorem A3: The Invitation Algorithm under Assumptions 1-7 satisfies Assertion 3.

Informal Proof: When a node i generates a new group number $S(i) \cdot g$, it is free to place any value in $S(i) \cdot c$ and $S(i) \cdot d$, since the number $S(i) \cdot g$ is unique and has never been used before. So the assertion is never violated in the process of generating new groups. When a node j stores a group number $S(i) \cdot g$ which it did not generate into $S(j) \cdot g$, then node j can only store $S(i) \cdot c$ and $S(i) \cdot d$ into its $S(j) \cdot c$ and $S(j) \cdot d$, respectively (where $S(i) \cdot c$ and $S(i) \cdot d$ were generated with $S(i) \cdot g$). Thus, the group number $S(i) \cdot g$ can never appear in any state vector with coordinator identification or task description which do not correspond. This means that Assertion 3 is satisfied. □

Theorem A4: The Invitation Algorithm under Assumptions 1-7 satisfies Assertion 4.

Informal Proof: Take a system with a set of operating nodes which can all communicate with each other and assume that no failures occur. Since there are no communication failures and nodes do not pause, no call statements will ever timeout.

Any nodes which are not participating in a normal group will eventually discover this (through procedure Timeout) and will form single node groups. Periodically, coordinators look for other coordinators and attempt to merge their groups. Since no new failures occur, merge attempts will either be successful or will leave the groups unchanged. By properly selecting the wait time in procedure Check, the possibility of repeated conflict between two groups attempting to merge can be eliminated, at least when no failures occur. (For example, if node i in procedure Check discovers a coordinator j with higher priority, then i can wait for at least $2P$ seconds, where P is the time between calls to procedure Check. This will give node j enough time to perform the merge without interference from node i.) Thus, eventually all operating nodes in the system will join a single group. □

Acknowledgment

Several useful ideas and suggestions were provided by S. Davidson, J. Kent, G. Le Lann, L. Svobodova, and the referees.

References

[1] E. R. Berlekamp, *Algebraic Coding Theory*, New York: McGraw-Hill, 1968.

[2] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. Ass. Comput. Mach.*, vol. 8, p. 569, Sept. 1965.

[3] C. A. Ellis, "Consistency and correctness of dupliicate database systems," in *Proc. 6th Symp. Operating Syst. Principles*, Nov. 1977, pp. 67-84.

[4] P. H. Enslow, "What is a distributed data processing system?," *Computer*, pp. 13-21, Jan. 1978.

[5] H. Garcia-Molina, "Performance of update algorithms for replicated data in a distributed database," Dep. Comput. Sci., Stanford Univ., Stanford, CA, Rep. STAN-CS-79-744, June 1979.

[6] J. N. Gray, "Notes on database operating systems," in *Advanced Course on Operating System Principles.* Munich, West Germany: Technical Univ., July 1977.

[7] L. Lamport, "A new solution of Dijkstra's concurrent programming problems," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 453-455, Aug. 1974.

[8] ——, "Time, clocks, and the ordering of events in a distributed system," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 558-564, July 1978.

[9] ——, "The implementation of reliable distributed systems," *Comput. Networks*, vol. 2, pp. 95-114, 1978.

[10] B. W. Lampson and H. E. Sturgis, "Crash recovery in a distributed data storage systems," Xerox PARC Rep., 1979.

[11] G. Le Lann, "Distributed systems—Towards a formal approach," in *Information Processing 77*, B. Gilchrist, Ed. Amsterdam, The Netherlands: North-Holland, 1977, pp. 155-160.

[12] D. A. Menasce, G. J. Popeck, and R. R. Muntz, "A locking protocol for resource coordination in distributed databases," *ACM Trans. Database Syst.* vol. 5, pp. 103-138, June 1980.

[13] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 395-404, July 1976.

[14] D. S. Parker *et al.*, "Detection of mutual inconsistency in distributed systems." in *Proc. 5th Berkeley Workshop Distributed Data Management Comput. Networks*, Feb. 1981, pp. 172-184.

[15] M. O. Rabin, "*N*-process synchronization by 4 log (base 2) *N*-valued shared variable," in *Proc. 21th Annu. Symp. Foundations Comput. Sci.*, Oct. 1980, pp. 407-410.

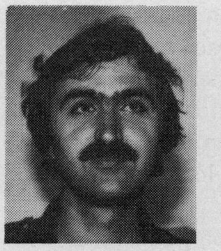
[16] J. B. Rothnie and N. Goodman, "A survey of research and development in distributed database management," in *Proc. 3rd VLDB Conf.*, Tokyo, Japan, 1977, pp. 48-62.

[17] F. B. Schneider, "Synchronization in distributed programs," Dep. Comput. Sci., Cornell Univ., Ithaca, NY, Tech. Rep. TR79-391, Jan. 1980.

[18] R. G. Smith, "The contract net protocol: High level communication and control in a distributed problem solver," in *Proc. 1st Int. Conf. Distributed Comput. Syst.*, Huntsville, AL, Oct. 1979, pp. 185-192.

[19] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRES," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 188-194, May 1979.

[20] R. H. Thomas, "A majority consensus approach to concurrency control," *ACM Trans. Database Syst.*, vol. 4, pp. 180-209, June 1979.



Hector Garcia-Molina (S'70-M'79) received the B.S. degree in electrical engineering from the Instituto Tecnologico de Monterrey, Monterrey, Mexico in 1974, and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA.

Currently, he is Assistant Professor in the Department of Electrical Engineering and Computer Science at Princeton University, Princeton, NJ. His research interests include distributed computing systems and database systems.

Dr. Garcia-Molina is a member of the Association for Computing Machinery.