Then, D would point to E when it is added to the waiting list, and E would be the last processor on the list instead of D.

The interesting aspect of the path compression algorithm is the use of incomplete information.

- Incomplete information: A way to improve the performance of a distributed algorithm is to loosen the requirement that the state of all processors be consistent. Instead, a processor can be required to store only a "good guess" about the current global state. To make such a scheme work, a distributed operation must be able to recover from mistakes due to incomplete information, and processors must be able to update their state regularly.
- Path compression: The path compression algorithm is a very effective use of incomplete information. A processor does not know where the token is, but knows a processor that has more recent information about where the token is held. Recovery is automatic, by repeating the operation on processors that have better and better information. Information update is easy because the requesting processor represents a better guess about the future token holder than any current guess. In addition to being a good example of making effective use of incomplete information, the path compression algorithm is a useful algorithm structure in its own right.

10.2 Election

We next turn our attention to the problem of election, which is broadly defined as getting a set of processors to agree on a leader. We often call the leader the coordinator and the other processors the participants. The leader typically knows the participants it leads. The coordinator and the participants are called the group.

Election is used when the distributed computation needs a unique, centralized coordinator. For example, some replicated data schemes use a primary copy and several secondary copies, where the data held by the primary copy is by definition the up-to-date data. A distributed computation might use a coordinator to assign subtasks to the participating processors.

The problem with assigning a unique and vital role to a single processor is the lack of fault tolerance. If the leader fails, the computation stops. The purpose of an election is to find another coordinator, which will determine the state of the system and restart the computation.

In some ways, election is similar to synchronization, since all processors must come to an agreement about who owns a token. However, in an election all participants must know who owns the token (i.e., who is the leader), while in synchronization the nontoken holders only need to know that they do not hold the token. Also, synchronization algorithms are designed to work well in the absence of failures, and failure handling is usually an afterthought. We explore the issue in the exercises. In contrast, election is

usually performed only if there is a failure, so failure handling must be an integral part of the protocol.

We will examine the classic election algorithms of Garcia-Molina. Several variations of election have been proposed, but the Garcia-Molina algorithm best defines and handles the possible failures.

Garcia-Molina devised two algorithms, using two different failure models. The first algorithm works under the fail-stop model and makes use of the failure assumptions to achieve simplicity. The second algorithm works under a more general, and a much more realistic, failure assumption. Since the failure model is complex, even stating the correctness conditions of the second algorithm requires care. However, the second algorithm is applicable to systems that one generally encounters.

Both algorithms make assumptions about the environment in which they operate. The most significant assumption is that each node has access to some permanent storage that survives node failures. This storage is used to record version numbers and ensure that they are strictly increasing. Typically, the node will save its state to a hard disk drive, although another possibility is nonvolatile semiconductor memory. This assumption does not require that the node have a local disk drive; a remote file system can serve as well. In addition, the algorithms assume that when a processor fails, it halts all processing (i.e., failures will not cause a node to behave erratically).

10.2.1 The Bully Algorithm

The bully algorithm makes an additional two assumptions about the computing environment:

1. The message delivery subsystem delivers all messages within T_m seconds of the sending of the message (message propagation time).
2. A node responds to all messages within T_p seconds of their delivery (message handling time).

These two assumptions allow the bully algorithm to build a reliable failure detector: If a processor doesn't respond to a message within $T = 2T_m + T_p$ seconds, it must have failed. All processors that look for this failure will detect it, and the failed processor will know that it failed when it recovers. These assumptions let us simplify the description of what the election algorithm must accomplish, because it can establish a global property. Distributed systems with these tight timing constraints are called synchronous systems.

Each processor keeps track of the state information shown in Algorithm Listing 10.16. When a node recovers from a failure, it sets Status to Down and stays in that state until the node begins an election to determine the coordinator (possibly itself). Whenever a node starts the election, it sets Status to Election. When the election is finished, the processors need to reestablish a common state (i.e., ensure that nodes that have recovered from a failure learn of the changes to the system state, etc.), so the processors enter the Reorganization status. After receiving the new common state, the nodes enter the Normal status and are again ready to proceed with the computation.

Chow & Johnson. Distrib. OS & Alg. | Addison Wesley, 1997.

ALGORITHM LISTING 10.16

State information for election.

Status	One of {Down,Election,Reorganization,Normal}.
Coordinator	The current coordinator of the node.
Definition	The state information for the task being performed.

We are now ready to state the correctness condition of the bully algorithm.

Correctness Assertion 1: Let G be a consistent state. Then, for any pair of nodes p_i and p_j, the following two conditions hold in G:

1. If $\text{Status}_i \in \{\text{Normal}, \text{Reorganization}\}$ and $\text{Status}_j \in \{\text{Normal}, \text{Reorganization}\}$ then $\text{Coordinator}_i = \text{Coordinator}_j$.
2. If $\text{Status}_i = \text{Normal}$ and $\text{Status}_j = \text{Normal}$ then $\text{Definition}_i = \text{Definition}_j$.

The correctness assertion for the bully algorithm states that if two nodes think that they are in working order, they agree on who is the coordinator and on the state of the system. So, the coordinator can direct the execution of the system, and the state of the system will not diverge in different processors. However, the coordinator cannot assume that all processors will obey its orders, since some of the processors might have failed. We will see algorithms that handle this problem in a later chapter.

We need another correctness assertion to guarantee that assertion 1 is not vacuously satisfied. For example, an algorithm that keeps all processors in the election state is "correct" according to assertion 1 but obviously is not an algorithm that we would want to use. So, we need to guarantee that the election algorithm makes progress (liveness).

Correctness Assertion 2: Let G be a consistent state. Then, the following two properties are eventually true in any run with no further failures starting in G:

1. There is a node i such that $\text{State}_i = \text{Normal}$ and $\text{Coordinator}_i = i$.
2. For every other nonfailed node j, $\text{State}_j = \text{Normal}$ and $\text{Coordinator}_j = i$.

The idea behind the bully algorithm is that we can preselect which processor should be the leader. That way, we avoid a lot of squabbling. We can assign a *priority* to each of the nodes so that the priority of a node is known to all other nodes. In an election, a node first checks to see if the higher-priority nodes are failed. If so, the node knows that it should be the leader; therefore the node "bullies" the other nodes into accepting its leadership. Since we have assumed that our failure detection is reliable, a low-priority node will not attempt to establish itself as the leader while a higher-priority node is still unfailed.

The bully algorithm makes use of two global variables, shown in Algorithm Listing 10.17 in addition to the ones used for the correctness assertion.

ALGORITHM LISTING 10.17

ıl variables used by the bully algorithm.

Up	A set containing names of processors known to be in the group.
halted	Identity of the processor that notified you of the current election.

An election is initiated by the Coordinator_Timeout procedure if a node does not hear from the coordinator for a suspiciously long time and by the Recovery procedure when the node recovers from a failure (Algorithm Listing 10.18). In addition, the coordinator periodically checks the state of other nodes (Algorithm Listing 10.19). If the coordinator detects that a failed processor has recovered, or that a processor in the group has failed, it will form the group again by calling an election. It might seem drastic to call an election when you learn of a recovered or failed processor, but the participants in the group will need to reorganize their computations when the recovered processor joins the group. We note that the coordinator might use a different mechanism to detect failures and recoveries (i.e., timeouts in the supported protocol or "let me join" messages). The important point is that there is some mechanism by which the coordinator learns of failed and recovered processors in a timely manner.

The election procedure, shown in Algorithm Listing 10.20 (on p.381), determines if a better leader (i.e., higher-priority node) exists. If so, the node waits for the higher-priority node to initiate the election. Otherwise the node attempts to establish itself as the coordinator by sending "enter-election" messages to the lower-priority processors. When the new coordinator establishes the new group, it will try to use an accurate picture of the processors that are nonfailed participants. If one of the participants in Up does not respond to the protocol messages, it must have failed. So, a new election is called to ensure an accurate group. This is not strictly necessary for correctness.

The election protocol makes use of several different types of queries about remote machines. The queries are handled by a thread that executes the algorithm in Algorithm Listing 10.21 (on p.382) and waits to answer the queries (and perhaps change the state of

ALGORITHM LISTING 10.18

Algorithms to initiate an election.

```
Coordinator_Timeout()
    if State = Normal or State = Reorganization
        send(Coordinator,AreYouUp),timeout=T
        wait until Coordinator sends (AYU_answer), timeout=T
        On timeout
            Election()

Recovery()
    State = Down
    Election()
```

ALGORITHM LISTING 10.19

Algorithm used by the coordinator to check the state of other processors.

```
Check()
    if State = Normal and Coordinator=Self
        for every other node j,
            send(j,AreYouNormal)
            wait until j sends (AYN_answer; status), timeout=T
                if (j ∈ Up and status=False) or j ∉ Up
                    Election()
                    return()
```

the protocol). One of the entries, Enter_Election, has the side effect of making the node enter the Election state. Since the coordinator is no longer well defined, the protocol must stop the computation that the node is performing, using the the procedure stop_processing(). For example, if the node was a copy of a replicated server, the node would stop accepting requests on the behalf of clients. In addition, the node must stop running its own election if it is executing one. The stop_processing() routine can be implemented in many ways (sending signals, setting flags in shared memory, etc.), and the systems issues must be carefully thought out.

Receiving an Enter_Election message means that a higher-priority processor is running its own election and should be elected as the leader. Although the node has already checked the higher-priority nodes and determined that they have failed, a higher-priority node might recover from its failure during the election.

Informally, the bully algorithm works because a node n makes certain that every other node has heard about its election before declaring itself the winner. It might seem that the Enter_Election messages can be combined with the Set_Coordinator messages to save a round of communication. But, to ensure that correctness assertion 1 is satisfied, no nonfailed processor can still be in the Normal state.

An execution of the bully algorithm is illustrated in Figure 10.8 (on p.382). In stage 1, p_3 tests all higher-priority nodes. After the timeout, p_3 knows that in consistent cut c_1 there is no higher-priority processor that is executing in the Normal state. In stage 2, p_3 tests all lower-priority nodes with an Enter_Election message. This message causes the recipient to enter the Election state. So, at cut c_2, p_3 knows that there is no processor in the Normal state, and there is no processor acting as a coordinator. Processor p_3 proposes itself as the coordinator and at cut c_3 it knows that all processors in Up are in the Reorganization state and believe that p_3 is the coordinator. Finally, at cut c_4, processor p_3 knows that all processors in Up are in the Normal state.

Correctness assertion 1 follows from the way that a processor that proposes itself as the coordinator first finds a consistent cut in which no processors is in the Reorganization or the Normal state. If two processors are concurrently trying to elect themselves the coordinator, one will have higher priority. Any progress toward election made by the lower-priority processor will be erased by the higher-priority processor when the higher-priority processor establishes cut c_2.

ALGORITHM LISTING 10.20

Bully election algorithm.

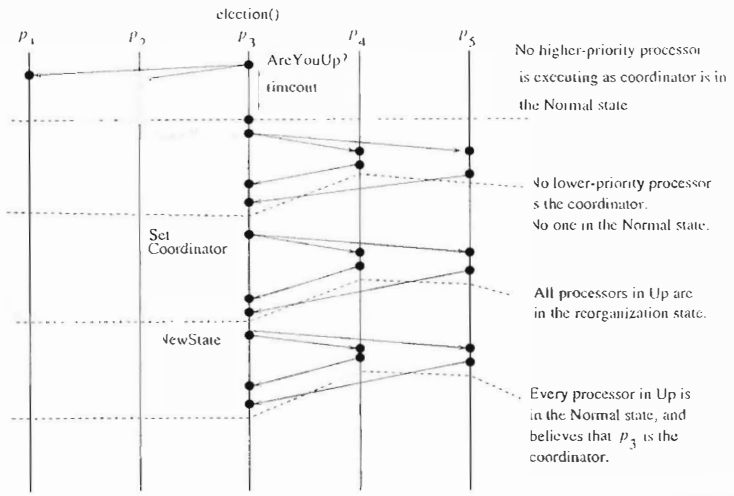
```
Election()
    highest = True
    For every higher-priority processor p
        send(p,AreYouUp)
        wait up to T seconds for (AYU_answer) messages
            AYU_answer (sender) :
                highest = False
    If highest = False
        return()

    State = Election
    halted = Self
    Up = {}
    For every lower-priority processor p
        send(p,Enter_Election)
        wait up to T seconds for (EE_answer) messages
            EE_answer (sender) :
                Up = Up ∪ {sender}

    num_answers = 0
    Coordinator = Self
    State = Reorganization
    for each p in Up
        send(p,Set_Coordinator; Self)
        wait up to T seconds for (SC_answer) messages
            SC_answer (sender):
                num_answers + +
        if num_answers < |Up|
            Election()
            return()

    num_answers = 0
    for each p in Up
        send(p,New_State; Definition)
        wait up to T seconds for (NS_answer) messages
            NS_answer (sender):
                num_answers + +
        if num_answers < |Up|
            Election()
            return()

    State = Normal
```

ALGORITHM LISTING 10.21

Algorithm executed by the thread that monitors the election.

```
Monitor_Election()
    while (True)
        wait for a message.
            case AreYouUp (sender):
                send(sender,AYU_answer)
            case AreYouNormal (sender):
                if State = Normal, send(sender,AYN_answer; True)
                    else send(sender,AYN_answer; False)
            case Enter_Election (sender) :
                State = Election
                stop_processing()
                stop the election procedure if it is executing
                halted = sender
                send(sender,EE_answer)
            case Set_Coordinator (sender, newleader) :
                if State = Election and halted = newleader
                    Coordinator = newleader
                    State = Reorganization
                send(sender,SC_answer)
            case New_State (sender, newdef):
                if Coordinator = sender and state = Reorganization
                    Definition = newdef
                    State = Normal
                send(NS_answer)
```

FIGURE 10.8 Execution of the bully algorithm.



10.2.2 The Invitation Algorithm

The bully algorithm is simple, but it makes a strong use of the fact that timeouts can accurately detect failed processors. Through an arbitrary timing glitch (lost messages, overfull buffers, temporary overloads at the processors), the bully algorithm can elect two leaders. The timeouts T can be made so large that a failure to respond to a timeout almost certainly means that it is impossible to communicate with the processor in question. However, such large timeouts mean that the election algorithm can take a very long time to execute. Furthermore, the node that fails to respond might not have failed, because a network partition might have occurred.

If we admit the possibility that the network and the processors might experience arbitrary processing delays (i.e., delete assumptions 1 and 2), it no longer makes sense to talk about the global coordinator. If there is a network partition, a coordinator will (should?) appear in both partitions. Two processors in the same partition might not be able to communicate with each other and thus both declare themselves the coordinator. If one cannot make safe assumptions about the timing of system events, one is developing algorithms for an asynchronous system.

Since declaring a global leader is fraught with peril, we need to reconsider what we mean by "coordinator." A coordinator is needed to guide a group of processors through a distributed computation. Hence it makes sense to tie the coordinator to the group that it coordinates. In addition to leading the group, the coordinator needs to ensure that the participants in the group are working according to the same plan. A coordinator might form the same group twice but with a different plan the second time around (for an example, see the dynamic quorum change protocols in Chapter 12). For this reason, it is convenient to make unambiguous the group identity with a group number (i.e., a sequence number). All members of the group agree on the group number, and the same group number is never used twice. We include the group number as part of the defining state of a node, as shown in Algorithm Listing 10.22.

Next we need to revise the correctness assertions. The first assertion states that all members of the same group agree on the identity of the coordinator:

Correctness Assertion 3: Let G be a consistent state. Then, for any pair of nodes p_i and p_j, the following two conditions hold in G:

1. If $\text{Status}_i \in \{\text{Normal, Reorganization}\}$, $\text{Status}_j \in \{\text{Normal, Reorganization}\}$, and $\text{Group}_i = \text{Group}_j$, then $\text{Coordinator}_i = \text{Coordinator}_j$.
2. If $\text{Status}_i = \text{Normal}$, $\text{Status}_j = \text{Normal}$, and $\text{Group}_i = \text{Group}_j$, then $\text{Definition}_i = \text{Definition}_j$.

ALGORITHM LISTING 10.22

Additional state variable required for asynchronous election.

oup	Identifier of the group that the processor belongs to.

We also need a liveness property. Since we now allow for communication errors, we might need to restrict ourselves to the set of nodes that can actually communicate with each other.

Correctness Assertion 4: Let R be a maximal set of nodes that can communicate in consistent state G_0. Then, the following two conditions are eventually true in any run starting at G_0 such that R remains a maximal set of nodes that can communicate and no further failures occur:

1. There is a node $p_i \in R$ such that $\text{State}_i = \text{Normal}$ and $\text{Coordinator}_i = p_i$.
2. For every other nonfailed node $p_j \in R$, $\text{State}_j = \text{Normal}$ and $\text{Coordinator}_j = p_i$.

Correctness assertion 3 is actually very easy to satisfy. When processor p wishes to establish itself as the coordinator, it creates a unique group identifier and targets a set of potential participants (including itself). Next, p suggests to the potential participants that they join the new group, with p as the coordinator. To join the new group, the participants accept p's suggestion. Since the group identifier is unique, correctness assertion 3.1 is automatically satisfied. When the new group is formed, p distributes the new definition to the participants in the group. Correctness assertion 3.2 is automatically satisfied if the participant only accepts new definitions from its current group coordinator, and only in the Reorganization state.

The main difficulty in implementing an election algorithm is to ensure that correctness assertion 4 is satisfied. There are many possible ways to run the election. In this section, we will generally follow the procedure suggested by Garcia-Molina but will point out where modifications are possible.

Since it is possible that there is more than one coordinator, the bully algorithm might not be able to ensure correctness assertion 4. The problem is that if the coordinators compete for participants, they might repeatedly steal participants from each other and no progress will be made toward forming a new group. It might make more sense to try to get the competing coordinators to agree to merge their groups into a single group. This is the approach taken by the invitation algorithm – one coordinator invites the other to merge into one group. By repeatedly merging groups, eventually there will be a single global group.

Groups that can communicate need to coalesce into larger groups. Periodically, the group coordinator searches for other groups, using the Check procedure shown in Algorithm Listing 10.23, and the coordinators it finds are added to the Others set. If another coordinator is found, the node will try to merge the other coordinator's group into its own. The other coordinator will be doing the same thing. To avoid livelock, the node delays for a time between detecting a new coordinator and acting on the information. Note that unlike the case in the bully algorithm, not receiving a response within T seconds does not mean anything in particular. In fact, the node might receive messages from the last time it sent AreYouCoordinator messages. In the Check procedure, this will not cause a problem. However, in other procedures we will need to match replies with requests.

ALGORITHM LISTING 10.23

Algorithm used by a coordinator to find the coordinators of other groups.

```
Check()
    If State = Normal and Coordinator = Self
        Others = {}
        for every other node P,
            send(P,AreYouCoordinator)
            wait up to T seconds for (AYC_answer) messages
                AYC_answer (sender; is_coordinator)
                    if is_coordinator = True
                        Others = Others ∪ sender
        if Others = {}
            return()
        Wait for a time inversely proportional to your priority
        Merge(Others)
```

We note that many optimizations can be made to the Check procedure. For example, we can try to assign priorities to the coordinators. If a high-priority coordinator receives an AreYouCoordinator from a lower-priority coordinator, the high-priority coordinator can make a note of the low-priority coordinator and attempt to merge the lower-priority coordinator's group into its own in a short period of time. The low-priority coordinator will not add the high-priority coordinator to the Others set, and will delay executing the Merge procedure for a while. If a participant receives an AreYouCoordinator message, it can answer with the identity of its coordinator. However, all that is required is that a coordinator searches out and attempts to merge with other groups. The implementer can experiment with this part of the protocol to find the most effective techniques.

If a node does not hear from its coordinator for a long period of time, the coordinator might have failed. The algorithm for handling a suspected failure of the coordinator is shown in Algorithm Listing 10.24. If the node still cannot get a response, the node declares its own group. Declaring a new group is performed by the recovery algorithm (Algorithm Listing 10.25), which is also invoked when a processor recovers from a failure. We note that this strategy is simple and legal (single processor groups are permitted by correctness assertion 3) but perhaps can be improved on. When a participant p finds that its coordinator has failed, it knows the identities of many other potential participants – the other participants in the group. Processor p can then use this information to try to quickly form a new group by sending invitations to all of the other participants of the old group. However, all that is required is that the failure of the coordinator be detected so that a new group can be formed.

Most of the work of the invitation algorithm is done in the Merge procedure, shown in Algorithm Listing 10.26. This procedure accepts a list of potential coordinators from the Check procedure and attempts to merge the groups of all known coordinators into one group. The node starts by forming its own group, with a new group number. Note that the group number is made unique by attaching a version number to the node name. All other

ALGORITHM LISTING 10.24

ected fa dinator.

```
Timeout()
    if Coordinator = Self
        Return()
    send(Coordinator,AreYouThere; Group)
    wait for AYC_answer, timeout is T
        On timeout,
            is_coordinator = False
        AYC_answer (sender; is_coordinator) :

    If is_coordinator = False
        Recovery()
```

known groups, and the members of the current group, are invited to join the new group. The code to handle acceptance messages is located in the Monitor_Election thread (which executes the algorithm shown in Algorithm Listing 10.28 on p.389) because the node must be able to respond to acceptance messages even while it is not merging groups. Finally, the new definition is distributed to the members of the new group. If a member of the group fails to acknowledge the new definition, the group needs to be formed again.

Since a node must be able to respond to Invitation messages at any time, these messages are handled by their own thread. The algorithm for this thread is shown in Algorithm Listing 10.27 (on p.388). A node will accept an invitation if it is in the normal state. If the node was the coordinator of its own group, it will extend the invitation to join the new group to the old group members. As in the bully algorithm, the election monitor responds to messages that can arrive at any time.

The execution of the invitation algorithm is illustrated in Figure 10.9 (on p.388). Processor p_1 executes the Merge procedure after collecting p_2 and p_3 as the processors to query. Processor p_2 accepts the invitation immediately. Processor p_3 was acting as a

ALGORITHM LISTING 10.25

Algorithm for recc

```
Recovery()
    State = Election
    stop_processing()
    Counter ++; Group = (Self | Counter)
    Coordinator = Self
    Up = {}
    State = Reorganization
    Description = (a single node task description)
    State = Norma
```


ALGORITHM LISTING 10.26

Algorithm by which a coordinator merges other groups into one.

```
Merge(Coordinator_set)
    if Coordinator = self and state = Normal
        State = Election
        stop_processing()
        Counter ++; Group = (Self | Counter)
        Coordinator = Self
        UpSet = Up
        Up = {}
        For each p in Coordinator_set,
            send(p,Invitation; Self,Group)
        For each p in Upset,
            send(p,Invitation; Self,Group)
        Wait for T seconds  // Answers are collected by the Monitor_Election thread.
        State = Reorganization
        num_answer = 0
        For each p in Up
            send(p,Ready; Group,Definition)
            wait up to T seconds for Ready_answer messages
            Ready_answer (sender; ingroup, new_group):
                if ingroup = True and new_group = Group
                    num_answer + +
        If num_answer < |Up|
            invoke Recovery()
        else
            state = Normal
```

coordinator itself, so p_3 forwards the invitation to p_4 and p_5. After T seconds, p_1 uses the set of responses it collected as the Up set. Because these processors explicitly accepted the invitation, p_1 knows that they have entered the Reorganization state. Processor p_1 replies with a Ready message, taking the participant processors into the Normal state. When all answers are received (within T seconds), p_1 knows that every processor in Up has accepted the group and that normal processing can resume.

The similarities and differences between the bully and the invitation algorithms highlight some issues we have seen before and will see again.

- Logical structure: The bully algorithm is simple and efficient because it imposes a logical structure on the processors. It might take a long time for groups to merge when using the invitation algorithm because there is no strongly defined structure on them.

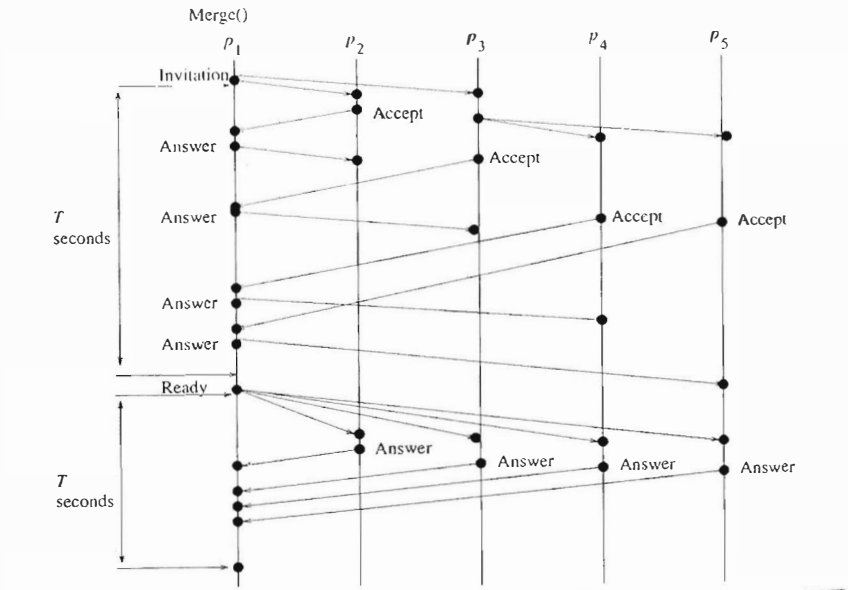
- Synchronous versus asynchronous: The bully algorithm uses the technique of flushing information from every other processor, so it feels similar to the distributed snapshot algorithm. It makes very strong use of its assumptions of bounded response time to

ALGORITHM LISTING 10.27

Algorithm executed by the thread that handles invitations.

```
Invitation()
    while True
        Wait for Invitation (new_coordinator; new_group):
        If State = Normal
            stop_processing()
            old_coordinator = Coordinator
            Upset = Up
            State = Election
            Coordinator = new_coordinator
            Group = new_group
            if old_coordinator = Self
                for each P in Upset
                    send(P,Invitation; Coordinator,Group)
            send(Coordinator,Accept; Group)
            wait up to T seconds for an Accept_answer(sender; accepted) message
                On timeout,
                    accepted = False
            if accepted is False invoke Recovery()
            State = Reorganization
```

FIGURE 10.9 Execution of the invitation algorithm.



ALGORITHM LISTING 10.28

Algorithm executed by the thread that monitors the election.

```
Election_Monitor()
    while True
        wait for a message
        Ready (sender; new_group,new_description)
            if Group = new_group and State = Reorganization
                Description = new_description
                State = Normal
                send(Coordinator,Ready_answer; True,Group)
            else
                send(sender,Ready_answer; False)
        AreYouCoordinator (sender)
            if State = Normal and Coordinator = Self
                send(sender,AYC_answer; True)
            else
                send(sender,AYC_answer; False)
        AreYouThere (sender; old_group)
            if Group = old_group and Coordinator = Self and sender in Up
                send(sender,AYT_answer; True)
            else
                send(sender,AYT_answer; False)
        Accept (sender; new_group)
            if State = Election and Coordinator = self and Group = new_group
                Up = Up ∪ sender
                send(sender,accept_answer; True)
            else
                send(sender,accept_answer; False)
```

build a simple algorithm. Unfortunately, such assumptions are not realistic in most systems.

The invitation algorithm works correctly in the presence of timing failures and thus is a practical algorithm. Note that in the new environment, stating the correctness of the algorithm becomes a much more difficult task. We will run into this difference between synchronous and asynchronous algorithms again, and the next chapter examines a fundamental difference between algorithms for synchronous and for asynchronous systems. Stating correctness criteria for asynchronous systems generally requires elaborate and perhaps unsatisfying statements.

- Relative consistency: In the synchronous system, correctness depends on *every* processor agreeing to a value. In the asynchronous system, correctness depends on getting a group of processors to agree to group membership and then to agree on a value. Since there may be several groups, consistency is relative to the group. Processors that do not communicate can be safely ignored.