Leader Election in Distributed Systems with Crash Failures

Scott D. Stoller*
Dept. of Computer Science, Indiana University, Bloomington, IN 47405, USA

17 July 1997

Abstract

Leader election is an important problem in distributed computing. Garcia-Molina's Bully Algorithm is a classic solution to leader election in synchronous systems with crash failures. This paper shows that the Bully Algorithm can be easily adapted for use in asynchronous systems. First, we re-write the Bully Algorithm to use a failure detector, instead of explicit time-outs; this yields a modular solution to leader election in synchronous systems. Second, we show that minor modifications to that algorithm yield a simple and efficient solution to leader election in asynchronous systems with crash failures. We point out a flaw in Garcia-Molina's specification of leader election in asynchronous systems, propose a revised specification, and show that the modified Bully Algorithm satisfies this specification.

Keywords: leader election, synchronous, asynchronous, crash failures, failure detector

1 Introduction

In a classic paper, Garcia-Molina specifies the leader election problem for synchronous and asynchronous distributed systems with crash failures and gives an elegant algorithm for each type of system; these algorithms are called the Bully Algorithm and Invitation Algorithm, respectively [GM82]. Leader election is an important problem in fault-tolerant distributed computing. It is closely related to the primary-backup approach (since choosing a primary replica is like electing a leader), an efficient form of passive replication. It is also closely related to group communication [Pow96], which (among other uses) provides a powerful basis for implementing active replication. For example, the group communication system in Amoeba [KT91, KT92] uses Garcia-Molina's Invitation Algorithm [GM82] to reconfigure a group after crashes. As another example, the group membership algorithms in Horus [FvR95, vRBM96] and Ensemble [Hay97] can be seen as a combination of Garcia-Molina's Bully Algorithm (for handling crashes) and Invitation Algorithm (for merging partitions of a group).

This paper shows that the Bully Algorithm can be easily adapted for use in asynchronous systems. The resulting algorithm, which we call the *Asynchronous Bully Algorithm*, is simple and message-efficient: in common cases, it uses half as many messages as the Invitation Algorithm. We obtain the Asynchronous Bully Algorithm in two steps. First, we re-write the Bully Algorithm to use a failure detector [CT94], instead of explicit time-outs. A failure detector is a module that reports crashes of other nodes. Re-writing the Bully Algorithm in this way has three benefits. First, the modularity

*Email: *stoller@cs.indiana.edu*. Tel.: 812-855-7979. Fax: 812-855-4829. Web: *http://www.cs.indiana.edu/~stoller/*

facilitates use of different failure detection mechanisms in different systems. Second, it helps clarify the ways in which synchrony assumptions are used, since only the implementation of the failure detector depends on them. Third, although the re-written algorithm is still intended for synchronous systems, it is easily adapted for use in asynchronous systems.

The specification of leader election in synchronous systems says (roughly) that the system always reaches a state in which all operational nodes agree on the leader. In asynchronous systems (with crash failures), agreement problems of this kind are unsolvable [FLP85], so a weaker specification must be used. Garcia-Molina gives such a specification, by incorporating the idea of *groups*: a group is a collection of nodes that agree on a leader. However, Garcia-Molina's specification is unintentionally strong, since contrary to his claim, the Invitation Algorithm does not satisfy it. Furthermore, Garcia-Molina's specification is undesirably strong for some systems, since it sometimes forces nodes that cannot directly communicate to be in the same group. We propose a new specification for leader election in asynchronous systems, which requires that the number of groups not exceed the number of fully-connected components needed to cover the reachability graph. This specification allows nodes that cannot directly communicate to be in the same group, but never forces this.

Finally, we adapt the Bully Algorithm for use in asynchronous systems. The effect of asynchrony is reflected by considering failure detectors that sometimes raise "false alarms" (*i.e.*, report failure of an operational node). Only minor modifications are needed in the other parts of the algorithm to deal with these false alarms. We show that the modified algorithm, which we call the *Asynchronous Bully Algorithm*, satisfies our specification.

2 Failure Detectors

The failure detector (FD) module detects and reports crashes. When the FD detects that node i is down, it raises a signal $\langle \mathrm{DownSig}, i \rangle$. These signals are the only outputs of the FD; the FD does not report recoveries, since those are easily detected by the application when it receives a message from a recovering node.

The FD has inputs $\mathrm{Start}_{\mathrm{FD}}(i)$ and $\mathrm{Stop}_{\mathrm{FD}}(i)$, which start and stop, respectively, monitoring of node i. If node i is already being monitored, $\mathrm{Start}_{\mathrm{FD}}(i)$ "resets" the monitoring of node i. We illustrate the meaning and usefulness of this with an example. Suppose node i crashes, and the FD on node j reports this by raising $\langle \mathrm{DownSig}, i \rangle$. Suppose node i then recovers and sends a message to node j. Node j receives that message and "believes" that node j has recovered. Suppose node i crashes again. The FD on node j, having already reported that node i is down, would not normally be required to raise $\langle \mathrm{DownSig}, i \rangle$ again. If the client (of the FD) on node j wants to receive updated information about the status of node i, it can call $\mathrm{Start}_{\mathrm{FD}}(i)$ to "reset" monitoring of node i. More precisely, after an invocation of $\mathrm{Start}_{\mathrm{FD}}(i)$, if node i is down, then the FD is required to raise $\langle \mathrm{DownSig}, i \rangle$, regardless of whether it raised $\langle \mathrm{DownSig}, i \rangle$ before the most recent invocation of $\mathrm{Start}_{\mathrm{FD}}(i)$. Furthermore, to ensure that the FD reports up-to-date information, we require that the client receives $\langle \mathrm{DownSig}, i \rangle$ *only* if node i is down after the most recent invocation of $\mathrm{Start}_{\mathrm{FD}}(i)$.

We formally specify the behavior of the FD using linear-time temporal logic [MP92] with the following predicate symbols:

up_i holds when node i is operational.

$start(i)$ holds when the client calls $\text{Start}_{\text{FD}}(i)$.

$stop(i)$ holds when the client calls $\text{Stop}_{\text{FD}}(i)$.

$downSig(i)$ holds when the client receives signal $\langle \text{DownSig}, i \rangle$.

In synchronous systems with reliable communication, complete and accurate failure detection is possible. Completeness means that if node i is being monitored and is down, then eventually node i recovers, or failure of node i is reported, or the client calls $\text{Start}_{\text{FD}}(i)$ or $\text{Stop}_{\text{FD}}(i)$. Formally, for all nodes i:[1]

$$\Box(start(i) \Rightarrow \Box(\neg up_i \Rightarrow \Diamond(up_i \vee downSig(i) \vee start(i) \vee stop(i)))). \tag{1}$$

The disjunct up_i in (1) means that the FD is not required to report that a node is down if the node eventually recovers; this simplifies the implementation of the FD, especially if rapid sequences of failures and recoveries are possible.

Accuracy means that $\langle \text{DownSig}, i \rangle$ is received only if node i was actually down after the most recent invocation of $\text{Start}_{\text{FD}}(i)$. This can be expressed concisely in temporal logic using the temporal operator $\mathcal{W}$ (read "unless") [MP92]; informally, $p\,\mathcal{W}\,q$ means that either p holds henceforth, or q eventually holds and p holds continuously until q holds. Formally, the accuracy requirement is, for all nodes i:

$$\Box(start(i) \Rightarrow \neg downSig(i)\,\mathcal{W}\,\neg up_i) \;\wedge\; \Box(stop(i) \Rightarrow \neg downSig(i)\,\mathcal{W}\,start(i)). \tag{2}$$

The second conjunct just says that $\langle \text{DownSig}, i \rangle$ is not received when node i is not being monitored.

The *latency* of a FD is the maximum time from when a node crashes and is being monitored until $\langle \text{DownSig}, i \rangle$ is received by the FD's client, provided the crashed node does not recover within that time. For example, suppose the FD has a latency of 3 time units. If node i crashes at time 45, and node j calls $\text{Start}_{\text{FD}}(i)$ at time 50, and node i remains down until at least time 53, then node j receives $\langle \text{DownSig}, i \rangle$ by time 53.

One motivation for encapsulating the failure detection mechanism in a module with a well-defined interface and semantics is to facilitate use of different failure detectors in different situations. For example, the simplest implementation of a FD is for each node to periodically send "Are you alive?" messages to each node being monitored, and to raise $\langle \text{DownSig}, i \rangle$ if a reply is not received in the expected time. A slightly more complicated approach is for each node i, when it starts monitoring node j, to tell node j to periodically send "I'm alive" messages to node i. This uses fewer messages and reduces the latency of the FD. A more complicated approach, based on an *attendance list* [Cri91, CS95],

[1]The temporal operator $\Box$ means "henceforth" or "always", and the temporal operator $\Diamond$ means "eventually". See [MP92] for details.

is to construct a logical ring and periodically circulate a token around it. If a node does not see the token within the expected time, then one or more failures have occurred; "Are you alive?" messages can be used to pinpoint the failures. This approach uses fewer messages if multiple nodes are being monitored by multiple nodes, though at the expense of increased detection latency.

3 Specification of Leader Election for Synchronous Systems

The specification has two parts, corresponding roughly to safety and liveness. The safety requirement asserts that nodes never disagree on the leader. We assume each node has a local variable *ldr* indicating its leader. Since it is impossible to make all nodes change *ldr* at the same instant, we introduce another variable, *status*. When *status* equals Norm, the node is in the normal mode of operation, and the value of *ldr* is significant; when *status* has any other value, a new leader is being elected. We require agreement only among nodes whose *status* is Norm. We use subscripts to distinguish local variables of different nodes; for example, $status_i$ and ldr_i are local variables of node i. Thus, we require:

SLE1: At all times, for all operational nodes i and j, if $status_i = \text{Norm}$ and $status_j = \text{Norm}$, then $ldr_i = ldr_j$.

Liveness requires that the system eventually enter a state in which the leader is operational and all operational nodes have status Norm; such states are characterized by the predicate *ldrElected*, defined by

$$ldrElected \;=\; (\forall i : status_i = \text{Norm} \wedge up_{ldr_i}). \tag{3}$$

When must such a state be reached? Repeated crashes and recoveries can prevent any protocol from making progress. Thus, we require:

SLE2: For a given system, there exists a constant c such that if no failures or recoveries occur for a period of at least c, then by the end of that period, the system reaches a state satisfying *ldrElected*. Furthermore, the system remains in that state as long as no failures or recoveries occur.

SLE1 and SLE2 are equivalent to Garcia-Molina's specification (Assertions 1 and 2 in [GM82]), except that we have simplified the problem slightly by omitting discussion of the re-distribution application-level tasks that might be needed following election of a new leader. Restoring this aspect of the problem is straightforward.

4 The Bully Algorithm for Synchronous Systems

The Bully Algorithm is designed for systems with the following properties. The system comprises a fixed set of nodes and a communication network. Nodes may crash and recover; other types of failure are assumed not to occur. Each node has access to a small amount of stable storage (*i.e.*, storage whose contents survives crashes). Nodes communicate by sending messages. Communication is FIFO. For synchronous systems, we assume also that communication is reliable.

We use integers to identify nodes: the set of node identifiers is

$$\mathrm{ID} = \{1, 2, \ldots, N\}, \tag{4}$$

where N is the number of nodes. The basic idea in the Bully Algorithm is that the operational node with the highest priority becomes the leader. For simplicity, we use node identifiers as priorities: lower numbers correspond to higher priorities, as in UNIX.[2]

Each node i has a status, initially Norm. If node i detects failure of its leader, then it sets its status to Elec_1, indicating that it is in stage 1 of organizing an election. In stage 1, node i checks whether nodes in $\mathrm{lesser}(i)$ are operational. If some of them are operational, node i waits, giving those higher-priority nodes a chance to become leader. If none of them are operational (*i.e.*, if node i receives $\langle \mathrm{DownSig}, j \rangle$ for all $j \in \mathrm{lesser}(i)$), then node i sets its status to Elec_2, indicating that it is in stage 2 of organizing an election. In stage 2, node i prepares nodes in $\mathrm{greater}(i)$ for a new leader by sending them Halt messages. When a node receives a Halt message, it sends an Ack message and sets its status to Wait, indicating that it is waiting for the outcome of an election. If a node with status Wait detects failure of the node that halted it, then it starts an election itself.

When a node i organizing stage 2 of an election has received an acknowledgment from or failure notification for each node in $\mathrm{greater}(i)$, then it becomes the leader, setting its status to Norm and sending a Ldr message to each node in $\mathrm{greater}(i)$ from which it received an acknowledgment. When those nodes receive Ldr messages from node i, they accept node i as the new leader and set their status to Norm.

Each of the messages described above is tagged with an *election identifier*, indicating which election the message is part of. An election identifier is a tuple containing the identifier of the node that started the election, that node's incarnation number (which is kept on stable storage and incremented on each recovery), and a sequence number (which is incremented for each election). If an Ack or Ldr message arrives that doesn't contain the expected election identifier, the message is ignored.

The Bully Algorithm, re-written to use failure detectors, appears in Figure 1. We call this the $\mathrm{Bully}_{\mathrm{FD}}$ Algorithm. It is written in reactive style, using the **On** statement to specify code to be executed when a message or signal is received. The **Periodically**(τ) statement specifies code to be executed periodically, with period τ. Each node starts by executing the **On recovery** statement.

We use the following notation. The modifier **stable** indicates that a variable is stored on stable storage. The statement **send** m **to** j sends a message m to node j. For a set S of nodes, **send** m **to** S abbreviates a loop that sends m to each node in S. Atomicity is not implied: the sender might send m to a subset of S and then crash. Similarly, for a set S of nodes, $\mathrm{Start}_{\mathrm{FD}}(S)$ and $\mathrm{Stop}_{\mathrm{FD}}(S)$ abbreviate the obvious loops.

The expressions $S \oplus x$ and $S \ominus x$ denote set S with element x inserted or removed, respectively. The C-style assignment statements $S \oplus\!= x$ and $S \ominus\!= x$ update the value of S by adding or removing

[2]In [GM82], higher numbers correspond to higher priorities. Our convention has the (potential) benefit of allowing new nodes to be integrated without changing the leader or the priority of other nodes.

element x, respectively. The operator π_1 returns the first component of a tuple.

The significant differences between our Bully$_{\rm FD}$ Algorithm and the original Bully Algorithm [GM82] are:

1. We use a failure detector, instead of explicit time-outs. To see that Start$_{\rm FD}$ and Stop$_{\rm FD}$ are used correctly, note that whenever node i would be waiting for a reply from node j in the Bully Algorithm, node j is being monitored by node i's FD in the Bully$_{\rm FD}$ Algorithm. Note that procedure Timeout in the Bully Algorithm is, in effect, incorporated into the code for handling $\langle \text{DownSig}, j\rangle$ in the Bully$_{\rm FD}$ Algorithm.

2. In stage 1 of an election, we check concurrently (rather than sequentially) whether the nodes in lesser(i) are operational. This optimization is independent of the use of a failure detector but would be awkward to express using Garcia-Molina's RPC-style communication primitive.

3. We include an election identifier in each message, to avoid confusion caused by messages that get delayed in the network. Outside of the FD, we do not assume a bound on message latency.

4. We omit the additional round of communication used in [GM82] to re-distribute application-level tasks. Restoring this additional round of communication is straightforward.

Correctness of the algorithm is independent of the value of τ and the values of the volative variables when a node recovers. The proofs that the Bully$_{\rm FD}$ Algorithm satisfies SLE1 and SLE2 are very similar to the proofs of Theorems A1 and A2 in [GM82] and are therefore omitted. Only item 2 above has a non-trivial impact on the proof, and it is easy to see from Garcia-Molina's proof that the status of the nodes in lesser(i) can be checked in any order. For the Bully$_{\rm FD}$ Algorithm, the constant c in SLE2 is

$$c_S = \max(\tau + 2\delta, \tau_{\rm FD}) + (n-1)\max(2\delta, \tau_{\rm FD}) + \delta, \tag{5}$$

where δ is the maximum message latency, $\tau_{\rm FD}$ is the latency of the failure detector, and (as shown in the pseudo-code) τ is the period with which Norm? messages are sent.

5 Specification of Leader Election in Asynchronous Systems

In an asynchronous system, it is impossible to satisfy SLE1 and SLE2; this follows from the FLP impossibility result [FLP85]. Following Garcia-Molina [GM82], we weaken the safety requirement by introducing the notion of groups. Since it is impossible to ensure that all nodes agree on the leader, we consider algorithms that organize the system into disjoint groups such that all members of a group agree on the group's leader. We assume each node has a local variable *grp* identifying the group it is currently in. Thus, the analogue of SLE1 is:

ALE1: At all times, for all operational nodes i and j, if $status_i$ = Norm and $status_j$ = Norm and $grp_i = grp_j$, then $ldr_i = ldr_j$.

```
var $status$ : {Norm, Elec$_1$, Elec$_2$, Wait}
    $ldr$ : ID
    $elid$ : ID $\times$ Nat $\times$ Nat
    $down$ : $Set$(lesser($i$))
    $acks$ : $Set$(greater($i$))
    $nextel, pendack$ : Nat
stable var $incarn$ : Nat

procedure StartStage1()
$status :=$ Elec$_1$
$elid := \langle i, incarn, nextel \rangle$
$nextel := nextel + 1$
$down := \emptyset$
if $i = 1$ then StartStage2()
else Start$_{\mathrm{FD}}$(lesser($i$)) fi

On $\langle$Halt$, t\rangle$ from $j$ :
Halting($t, j$)

procedure Halting($t, j$) :
$down \ominus= j$
Start$_{\mathrm{FD}}$($j$)
$elid := t$
$status :=$ Wait
send $\langle$Ack$, t\rangle$ to $j$

On $\langle$DownSig$, j\rangle$ :
if $j \in$ lesser($i$) then
    $down \oplus= j$
    if ($status$ = Norm $\wedge$ $j = ldr$)
        $\vee$($status$ = Wait $\wedge$ $j = \pi_1(elid)$) then
        StartStage1()
    else if $status$ = Elec$_1$ $\wedge$ $down \supseteq$ lesser($i$) then
            StartStage2()
         fi
    fi
else (* $j \in$ greater($i$) *)
    if $status$ = Elec$_2$ $\wedge$ $j = pendack$ then
        ContinStage2()
    fi
fi

procedure StartStage2()
$status :=$ Elec$_2$
$acks := \emptyset$
$pendack := i$
ContinStage2()

procedure ContinStage2()
if $pendack < N$ then
    $pendack := pendack + 1$
    Start$_{\mathrm{FD}}$($pendack$)
    send $\langle$Halt$, elid\rangle$ to $pendack$
else (* I'm the leader *)
    $ldr := i$
    $status :=$ Norm
    send $\langle$Ldr$, t\rangle$ to $acks$
fi

On $\langle$Ack$, t\rangle$ from $j$ :
if $status$ = Elec$_2$ $\wedge$ $t = elid$ $\wedge$ $j = pendack$ then
    $acks \oplus= j$
    ContinStage2()
fi

On $\langle$Ldr$, t\rangle$ from $j$ :
if $status$ = Wait $\wedge$ $t = elid$ then
    $ldr := j$
    $status :=$ Norm
    Stop$_{\mathrm{FD}}$(ID $\ominus$ $i$)
    Start$_{\mathrm{FD}}$($ldr$)
fi

Periodically($\tau$) :
if $status$ = Norm $\wedge$ $ldr = i$ then
    send $\langle$Norm?$, elid\rangle$ to greater($i$)
fi

On $\langle$Norm?$, t\rangle$ from $j$ :
if $status \neq$ Norm then
    send $\langle$NotNorm$, t\rangle$ to $j$
fi

On $\langle$NotNorm$, t\rangle$ from $j$ :
if $status$ = Norm $\wedge$ $ldr = i$ $\wedge$ $t = elid$ then
    StartStage1()
fi

On recovery :
$incarn := incarn + 1$
StartStage1()
```

Figure 1: Bully$_{\mathrm{FD}}$ Algorithm. Code executed by node i.

ALE1 corresponds to Assertion 3 in [GM82].

If we added the requirement that there always be at most one group, then ALE1 would be equivalent to SLE1. We allow multiple groups but require that the number of groups be as small as is reasonable. Formalizing this requirement is slightly tricky. Garcia-Molina attempts to formalize it by requiring (roughly) that if a set of nodes can all communicate with each other, then they end up in the same group. To express this more precisely, we introduce some terminology, based on [CS95, Cri96]. Two nodes are *connected* in a given time interval if all messages sent between them during that time interval are delivered within δ time units, where δ is a known constant. Two nodes are *disconnected* in a given time interval if all messages sent between them during that time interval are lost. Two nodes are *partially connected* in a given time interval if they are neither connected nor disconnected during that time interval. Garcia-Molina's Assertion 4 can be paraphrased as follows:

> *Assertion 4.* Suppose there is a set R of nodes which are operational and pairwise connected for the duration of an election. Suppose also that there is no superset of R with this property. Then the election leaves the system in a state in which: (a) there is a node i in R with $status_i =$ Norm and $ldr_i = i$, and (b) for every other node j in R, $status_j =$ Norm and $ldr_j = i$ and $grp_j = grp_i$.

At first glance, this assertion seems to require that two nodes end up in the same group only if they are connected—in other words, this requirement seems not to specify whether nodes that are partially connected or disconnected must end up in the same group. Surprisingly, this is not always true. To see why, consider the system shown in Figure 2. The edge between nodes 1 and 2 indicates that those nodes are connected. Similarly, nodes 2 and 3 are connected, but nodes 1 and 3 are disconnected.[3] The dotted lines indicate two sets $R_1 = \{1, 2\}$ and $R_2 = \{2, 3\}$ that meet the preconditions of Assertion 4. Thus, Assertion 4 requires that nodes 1 and 2 end up in the same group, and that nodes 2 and 3 end up in the same group, and hence that all three nodes end up in the same group, even though nodes 1 and 3 are not connected.

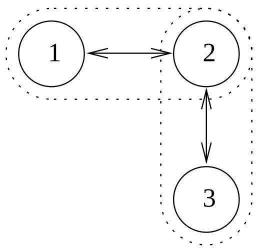

Figure 2: Example showing that Assertion 4 sometimes requires disconnected (or partially-connected) nodes to be in the same group.

For some applications, it is desirable for a group to contain only nodes that are pairwise connected. Thus, in some cases, Assertion 4 is an undesirably strong requirement. Furthermore, as we show in

[3]Garcia-Molina explicitly considers non-transitive connectivity [GM82, page 52].

Appendix A, the Invitation Algorithm does not satisfy Assertion 4, despite Theorem A4 in [GM82].

We propose the following weaker requirement, to formalize the idea that there should be as few groups as possible without forcing disconnected nodes to be in the same group. We say that the system is *stable* in a time interval if, during that interval, no crashes or recoveries occur and every pair of nodes is either connected or disconnected (*i.e.*, none are partially connected). When a system is stable, we define its *connectivity graph* to be the undirected graph $\langle \mathrm{ID}, E \rangle$, where $(i, j) \in E$ iff nodes i and j are connected. We require that the number of groups be at most the minimum number of fully-connected components needed to cover the connectivity graph.[4] Thus:

> **ALE2:** For a given system, there exists a constant c such that if the system is stable for a period of at least c, then by the end of that period, the system reaches a state satisfying *ldrsElected* and such that the number of groups is at most the minimum number of fully connected components needed to cover the system's connectivity graph. Furthermore, the system remains in that state as long as the system remains stable.

where

$$\mathit{ldrsElected} \;=\; (\forall i : \mathit{status}_i = \mathrm{Norm} \wedge \mathit{up}_{\mathit{ldr}_i} \wedge \mathit{grp}_{\mathit{ldr}_i} = \mathit{grp}_i). \tag{6}$$

Note that two fully-connected components are needed to cover the graph in Figure 2, so ALE2 is satisfied even if nodes 1 and 3 are in different groups.

A weaker requirement, which does not constrain the number of groups in the stable configuration, is:

> **ALE2′:** For a given system, there exists a constant c such that if the system is stable for a period of at least c, then by the end of that period, the system reaches a state satisfying *ldrsElected*. Furthermore, the system remains in that state as long as the system remains stable.

6 Failure Detectors in Asynchronous Systems

In an asynchronous system, complete and accurate failure detection is impossible; this, too, follows from the FLP impossibility result [FLP85]. Thus, a FD for an asynchronous system is required to satisfy completeness requirement (1) but not accuracy requirement (2). However, we do require that when the system is stable, failure detection is accurate, with maximum latency τ_{FD}. More precisely, if the system is stable during the interval $[t_1, t_2]$, and if $\langle \mathrm{DownSig}, j \rangle$ is received at some time t in the interval $[t_1 + \tau_{\mathrm{FD}}, t_2]$, then j was down at some time in the interval $[t - \tau_{\mathrm{FD}}, t]$.

In asynchronous systems, flexibility in choosing an appropriate FD is important, since the uncertainty of failure detection in asynchronous systems leaves many reasonable options. The FD implementations sketched in Section 2 can still be used, with time-out periods based the latency within which most messages are delivered. The choice of time-out period depends on the cost to the application of

[4]A set of fully-connected components *cover* a graph if they contain every node in it.

inaccurate failure detections. In systems where this cost is particularly high, cooperative failure detection, in which (roughly speaking) a node is declared faulty only if a certain number of nodes are unable to communicate with it, may be desirable.

7 The Asynchronous Bully Algorithm

Modifying the Bully_{FD} Algorithm to work in an asynchronous system is straightforward. As long as a node believes its leader is alive, it rejects participating in elections that would give it a lower-priority leader. Also, when a leader i is halted in stage 2 of an election organized by a higher-priority leader j, node i informs its followers (*i.e.*, nodes with $\mathit{status} = \text{Norm} \wedge \mathit{ldr} = i$) that it is no longer a leader; this is necessary because communication failures might cause its followers not to learn about j's election. A convenient way for node i to inform its followers that it is no longer their leader is by "playing dead". We assume the failure detector has additional inputs $\text{PlayDead}_{\text{FD}}(j)$, which causes the calling node to appear dead to the FD of node j, and $\text{PlayAlive}_{\text{FD}}(j)$, which cancels the effect of $\text{PlayDead}_{\text{FD}}(j)$. Thus, if a node i calls $\text{PlayDead}_{\text{FD}}(k)$, and node k's FD is monitoring node i, then within time τ_{FD}, node k receives $\langle \text{DownSig}, i \rangle$. For a set S of nodes, $\text{PlayDead}_{\text{FD}}(S)$ and $\text{PlayAlive}_{\text{FD}}(S)$ abbreviate the obvious loops.

Election identifiers play the role of group identifiers, *i.e.*, the variable *grp* is identified with the variable *elid*. The modified algorithm, which we call the *Asynchronous Bully Algorithm*, appears in Figure 3. Much of the code is the same as in Figure 1, but we have:

1. Modified the action for Halt messages to reject participating in elections that would give the node a lower-priority leader. Modified procedure Halting to call $\text{PlayDead}_{\text{FD}}$.

2. Added an action to handle Rej messages.

3. Modified the action for Norm? messages received from j, so that NotNorm is sent even if $\mathit{status} = \text{Norm}$, if an election by j would give node i a higher-priority leader. Added the conjunct $j < \pi_1(\mathit{elid})$ in the first disjunct of the conditional to prevent node i from causing j to start an election that node i would reject.

4. Modified procedure StartStage1 and the action for $\langle \text{DownSig}, j \rangle$ to omit stage 1 of elections. This is possible because of the weaker safety requirement. Procedure StartStage2 is modified to keep track of the election identifier and to call $\text{PlayAlive}_{\text{FD}}$. Variable *down* was needed only for stage 1, so it has been eliminated.

Theorem 1. The Asynchronous Bully Algorithm satisfies ALE1 and ALE2′.

Proof. See Appendix B. ∎

Neither the Asynchronous Bully Algorithm nor the Invitation Algorithm satisfies ALE2. To see that the Asynchronous Bully Algorithm does not, consider the following scenario in a system with 6 nodes. Three nodes—say, 1, 2 and 3—are pairwise connected. Each of these nodes is connected to exactly one

```
var $status$ : {Norm, Elec$_1$, Elec$_2$, Wait}
    $ldr$ : ID
    $elid$ : ID $\times$ Nat $\times$ Nat
    $acks$ : $Set$(greater($i$))
    $nextel, pendack$ : Nat
stable var $incarn$ : Nat

On $\langle$Halt, $t\rangle$ from $j$ :
if ($status$ = Norm $\wedge$ $ldr < j$)
  $\vee$($status$ = Wait $\wedge$ $\pi_1(elid) < j$) then
  send $\langle$Rej, $t\rangle$ to $j$
else Halting($t, j$)
fi

procedure Halting($t, j$) :
PlayDead$_{FD}$(greater($i$))
Start$_{FD}$($j$)
$elid := t$
$status :=$ Wait
send $\langle$Ack, $t\rangle$ to $j$

On $\langle$DownSig, $j\rangle$ :
if ($status$ = Norm $\wedge$ $j = ldr$)
  $\vee$($status$ = Wait $\wedge$ $j = \pi_1(elid)$) then
  StartStage2()
else if $status$ = Elec$_2$ $\wedge$ $j = pendack$ then
        ContinStage2()
     fi
fi

procedure StartStage2()
PlayAlive$_{FD}$(greater($i$))
$elid := \langle i, incarn, nextel\rangle$
$nextel := nextel + 1$
$status :=$ Elec$_2$
$acks := \emptyset$
$pendack := i$
ContinStage2()

On $\langle$Rej, $t\rangle$ from $j$ :
if $status$ = Elec$_2$ $\wedge$ $j = pendack$ then
  ContinStage2()
fi

procedure ContinStage2()
if $pendack < N$ then
  $pendack := pendack + 1$
  Start$_{FD}$($pendack$)
  send $\langle$Halt, $elid\rangle$ to $pendack$
else (* I'm the leader *)
  $ldr := i$
  $status :=$ Norm
  send $\langle$Ldr, $t\rangle$ to $acks$
fi

On $\langle$Ack, $t\rangle$ from $j$ :
if $status$ = Elec$_2$ $\wedge$ $t = elid$ $\wedge$ $j = pendack$ then
  $acks \oplus= j$
  ContinStage2()
fi

On $\langle$Ldr, $t\rangle$ from $j$ :
if $status$ = Wait $\wedge$ $t = elid$ then
  $ldr := j$
  $status :=$ Norm
  Stop$_{FD}$(ID $\ominus$ $i$)
  Start$_{FD}$($ldr$)
fi

Periodically($\tau$) :
if $status$ = Norm $\wedge$ $ldr = i$ then
  send $\langle$Norm?, $elid\rangle$ to greater($i$)
fi

On $\langle$Norm?, $t\rangle$ from $j$ :
if ($status \neq$ Norm $\wedge$ $j < \pi_1(elid)$)
  $\vee$($status$ = Norm $\wedge$ $j < ldr$) then
  send $\langle$NotNorm, $t\rangle$ to $j$
fi

On $\langle$NotNorm, $t\rangle$ from $j$ :
if $status$ = Norm $\wedge$ $ldr = i$ $\wedge$ $t = elid$ then
  StartStage2()
fi

On recovery :
$incarn := incarn + 1$
StartStage2()
```

Figure 3: Asynchronous Bully Algorithm. Code executed by node i.

of the premaining three nodes vertices 4, 5 and 6—say, i is connected to $i+3$, for $i = 1, 2, 3$. The unique minimum clique cover in this case comprises $\{1,4\}$, $\{2,5\}$, and $\{3,6\}$. However, the Asynchronous Bully Algorithm would lead to a configuration in which nodes 1, 4, 5, and 6 are leaders. Finding an appropriate liveness requirement for these and similar leader election algorithms is still an open problem. ALE2$'$ is undesirably weak, because it is satisfied by trivial algorithms that put each node in a separate group. ALE2 is undesirably strong, because algorithms satisfying it must compute (in some way) a minimum clique cover, and the problem of finding a minimum clique cover is NP-complete [GJ79]. Note that the specifications in [CS95] assume a stronger notion of stability, namely, that connectivity is transitive when the system is stable; as a result, they avoid these difficulties.

The Invitation Algorithm also satisfies ALE1 and ALE2$'$, so it is interesting to compare the efficiency of the Asynchronous Bully Algorithm and the Invitation Algorithm. We compare the message complexities in two scenarios, both of which assume the system initially contains a single group.

The first scenario we consider is the steady-state, *i.e.*, the system is stable. We compare the numbers of messages used to monitor the status of other nodes. In the Invitation Algorithm, the leader periodically, with period τ, sends AreYouNormal to each other node, and those nodes reply Yes. So, the Invitation Algorithm sends $2(N-1)$ messages per time τ. In the Asynchronous Bully Algorithm, the leader periodically sends Norm? messages. Also, each node other than the leader monitors the leader for failures. Assuming the failure detector is implemented using "I'm alive" messages (as described in Section 2), the Asynchronous Bully Algorithm also sends $2(N-1)$ messages per time τ. If $\tau_{\text{FD}} \geq \tau + \delta$, then the "I'm alive" messages can be piggybacked on the Norm? messages, in which case the Asynchronous Bully Algorithm sends only $N-1$ messages per time τ.

In the second scenario, the leader crashes, but no other failures occur. Presumably this scenario is more likely than scenarios involving multiple failures. Note that scenarios in which a node other than a leader crashes are not interesting here, since neither the Invitation Algorithm nor the Asynchronous Bully Algorithm reconfigures the system after failure of a non-leader. We show below that in this scenario, in the average case, the Invitation Algorithm uses $N^2+O(N)$ messages, while the Asynchronous Bully Algorithm uses only $\frac{1}{2}N^2 + O(N)$ messages. In the worst case, these numbers double: the Invitation Algorithm uses $2N^2 + O(N)$ messages, while the Asynchronous Bully Algorithm uses only $N^2 + O(N)$ messages.

To simplify the analysis of the Invitation Algorithm, we assume that the interval between calls to procedure Check is greater than or equal to NT, where T is the time-out period; this assumption typically holds in systems with moderate numbers of nodes. In the average case, half of the nodes call procedure Check before node $n-1$ calls it, and each of those nodes sends $N-1$ AreYouCoordinator messages, for a total of $(N-1)^2/2$ AreYouCoordinator messages, plus a comparable number of responses. For the Asynchronous Bully Algorithm, in the average case, half of the nodes call procedure StartStage2 before node 2 calls it, and each of those nodes sends on average $(N-1)/2$ Halt messages, for a total of $(N-1)^2/4$ Halt messages, plus a comparable number of responses. In the worst case, all of the other nodes call procedure Check (or procedure StartStage2) before node $n-1$ (or node 2), so relative to the average case, twice as many messages are sent.

8 Related Work

The most closely related work is [GM82], which has been discussed above. There is considerable work on self-stabilizing algorithms for leader election (*e.g.*, [DIM91, ILS95]). Those algorithms operate under weaker assumptions about failures (essentially, a failure can cause an arbitrary state transition) and provide weaker guarantees. The weaker assumptions about failures force a weakening of the guarantees: if arbitrary state transitions can occur, then it is impossible to ensure that an invariant (such as SLE1 or ALE1) is preserved. Comparing synchrony assumptions, the situation is reversed: the self-stabilizing algorithms in [DIM91, ILS95] are based on synchronous communication, while we consider also unreliable asynchronous communication.

We assume (*cf.* Section 4) that each node has a unique identifier (*e.g.*, a network address). This assumption is appropriate for algorithms running above a network layer, such as IP. In particular, it is appropriate for the types of applications described in Section 1. In contrast, there is considerable work on algorithms for leader election in *uniform* systems (*e.g.*, [DIM91, ILS95]), in which all nodes with the same number of neighbors are identical (and therefore do not have unique identifiers).

References

[Cri91] Flaviu Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4(4), 1991.

[Cri96] Flaviu Cristian. Synchronous and asynchronous group communication. *Communications of the ACM*, 39(4):88–97, April 1996.

[CS95] Flaviu Cristian and Frank Schmuck. Agreeing on processor group membership in asynchronous distributed systems. Technical Report CSE95-428, University of California, San Diego, 1995. Available via http://www-cse.ucsd.edu/users/flaviu/publications.html.

[CT94] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report TR 94-1458, Cornell University, Department of Computer Science, 1994.

[DIM91] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform dynamic self-stabilizing leader election. In Sam Toueg, Paul G. Spirakis, and Lefteris Kirousis, editors, *Proc. 5th International Workshop on Distributed Algorithms (WDAG '91)*, volume 579 of *Lecture Notes in Computer Science*, pages 167–180. Springer-Verlag, 1991.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FvR95] Roy Friedman and Robbert van Renesse. Strong and weak virtual synchrony in Horus. Technical Report TR 95-1491, Cornell University, Department of Computer Science, 1995.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[GM82] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):47–59, January 1982.

[Hay97] Mark Hayden. The Ensemble Distributed Communication System, 1997. Documentation available via http://www.cs.cornell.edu/Info/Projects/Ensemble/index.html.

[ILS95] Gene Itkis, Chengdian Lin, and Janos Simon. Deterministic, constant space, self-stabilizing leader election on uniform rings. In Jean-Michel Hélary and Michel Raynal, editors, *Proc. 9th International Workshop on Distributed Algorithms (WDAG '95)*, volume 972 of *Lecture Notes in Computer Science*, pages 288–302. Springer-Verlag, 1995.

[KT91] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the amoeba distributed operating system. In *Proc. IEEE 11th International Conference on Distributed Computing Systems (ICDCS)*, pages 222–230. IEEE Computer Society Press, 1991.

[KT92] M. Frans Kaashoek and Andrew S. Tanenbaum. Efficient reliable group communication for distributed systems. Rapport IR-295 IR-295, Faculteit Wiskunde en Informatica, Vrije Universiteit, 1992. Revised version available from ftp://ftp.cs.vu.nl/pub/papers/amoeba/group94.ps.Z.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag, 1992.

[Pow96] David Powell, guest editor. Special section on group communication. *Communications of the ACM*, 39(4):50–97, April 1996.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

Appendix A

We describe a scenario in which the Invitation Algorithm violates Assertion 4 of [GM82]. The system contains 3 nodes. Initially, all nodes are operating, and all pairs of nodes can communicate. All nodes are in the same group, and node 1 is the leader. The following events occur:

1. Node 1 crashes.

2. Nodes 2 and 3 each call Timeout and then Recovery; each forms a singleton group.

3. Node 1 recovers, but communication between nodes 1 and 3 has been lost. In other words, Assumptions 8 and 9 hold for the nodes in the set $\{1, 2\}$ and for the nodes in the set $\{2, 3\}$, but not for the nodes in the set $\{1, 2, 3\}$.

4. Node 1 calls Recovery and then Check. Since nodes 1 and 2 can communicate, node 1 calls Merge with CoordinatorSet $= \{2\}$.

The outcome is that nodes 1 and 2 are in one group, and node 3 remains in a singleton group. Note that this configuration is "stable", *i.e.*, assuming no more failures occur, these groups will not change.

Now consider whether Assertion 4 is satisfied. The set $\{2, 3\}$ satisfies the hypotheses about R in Assertion 4, so we have the requirements: (a) some node i in $\{2, 3\}$ is a coordinator, and (b) every other node j in $\{2, 3\}$ has node i as its coordinator. Requirement (b) is not satisfied, since node 2 has node 1 as its coordinator.

Appendix B

Proof-sketch of Theorem 1. Satisfaction of ALE1 follows easily from the observation that $status =$ Norm implies $ldr = \pi_1(elid)$. (Recall that grp and $elid$ are the same.)

For satisfaction of ALE2$'$, note first that if the system is stable, the algorithm changes state only a finite number of times, *i.e.*, it eventually reaches and thereafter remains in some state g. It is straightforward to show that g is reached within time c_A, where c_A is $O((N-1)\max(\tau + 2\delta, \tau_{\mathrm{FD}}))$.

We argue that g satisfies *ldrsElected*. First we show that for each node i, $status_i = norm$ in state g. Suppose a node i has $status_i \neq$ Norm at some time at least τ_{FD} into the interval of stability. Then $status_i$ is Elec$_2$ or Wait. If $status_i =$ Elec$_2$, then within total time c_A, either: (1) node i finishes its election (since each node j in greater(i) is either up and connected to node i, in which case node j replies to node i with an Ack or Rej message, or down or disconnected from node i, in which case the FD on node i raises $\langle \mathrm{DownSig}, j\rangle$) and sets its status to Norm, or (2) node i gets halted by a higher-priority node and sets its status to Wait, in which case the following argument applies. If $status_i =$ Wait, then either: (1) node i receives a Ldr message and sets its status to Norm, or (2) node $\pi_1(elid_i)$ gets halted by a higher-priority node and plays dead, in which case the FD on node i raises $\langle \mathrm{DownSig}, \pi_1(elid_i)\rangle$, and node i sets its status to Elec$_2$, so the argument for $status_i =$ Elec$_2$ applies, or (3) the Ldr message from node $\pi_1(elid_i)$ to node i gets lost, in which case those two nodes must be disconnected (since we are assuming the system is stable), so the FD on node i raises $\langle \mathrm{DownSig}, \pi_1(elid_i)\rangle$, and node i sets its status to Elec$_2$, so the argument for $status_i =$ Elec$_2$ applies. Alternation between the cases for $status_i =$ Elec$_2$ and $status_i =$ Wait can occur only $O(N)$ times, since a node participates only in elections that would give it a leader of strictly higher priority. Thus, eventually a "base case" is reached and node i sets its status to Norm.

Next we show that up_{ldr_i} holds. This follows immediately from the completeness requirement (1) for the FD and the code for handling $\langle \mathrm{DownSig}, j\rangle$. To see that the requirement $grp_{ldr_i} = ldr_i$ is satisfied, note that a (temporary) violation of this requirement arises only if node ldr_i is halted and obtains a new leader j, but node i is disconnected from node j and therefore does not participate in the election of j. When node ldr_i is halted by node j, it calls PlayDead$_{\mathrm{FD}}$(greater(ldr_i)), so the FD on node i raises $\langle \mathrm{DownSig}, ldr_i\rangle$, and node i gets a new leader, thus correcting the temporary violation of this requirement.