CSP Practical 98.4

The "Bully" algorithm

The following description of a distributed election algorithm is a slightly edited version of one on p. 306 of *Distributed Systems* by Colouris, Dollimore and Kindberg. (You can find an example and a picture of it in operation there.)

The bully algorithm This is used to allow a group of processors to have an agreed *coordinator* from amongst them, despite the fact that processors may at various times fail and revive, The aim is always to have them agree that the co-ordinator is the highest indexed processor that is working. We assume that communication is reliable, but processes can fail at any time (including during the election procedure that the algorithm represents).

There are three types of message in this algorithm:

```
channel election:Proc.Proc

channel answer:Proc.Proc

channel coordinator:Proc.Proc
```

where `Proc={1..N}`, say, and all these represent synchronisations between a sender process (the first index) and a receiver.

An `election.n.m` message is used to announce an election (from `n` to `m`); an `answer` message is sent in response to an election message; and a `coordinator` is sent to announce the identity of the new coordinator. A process begins an election when it notices that the coordinator has failed[1] . Let us therefore assume that there are four further channels:

[1] The statement in the description of the algorithm that a process notices that its coordinator is a little vague, and begs a lot of detail about the mechanism. I used the two channels `test` and `ok` that are declared here: any process can decide to send a `test` message to its coordinator, who only sends `ok` back if it is a properly working coordinator. This raises the question of what happens if a process tests its coordinator but the latter, though working, does not think is *is* the coordinator! You could quite legitimately (in my opinion) program your system so that it might make such tests successful, and/or use a different protocol for testing, such as having a pair of signals that any process can perform,`working.n.m` or `broken.n.m` depending on what state it is in.

```
channel fail, revive:Proc

channel test,ok:Proc.Proc
```

The first two are communicated when a process fails or revives (allowing us to see what the current status of each process is). The last two are used by one process to test another, and to give a response to such a test.

To begin an election, a process sends an `election` message to all those with a higher index. It then awaits an `answer` message in response. If none arrives within time `T1`, the processor considers itself the coordinator (as it thinks all the higher ones must be dead), and sends a `coordinator` message to all processes with lower identifiers announcing this fact. Otherwise, the process waits a further limited period (`T2`), expecting to receive a coordinator message as described in the next paragraph, before beginning another election (if no such message is received).

If a process receives a coordinator message, it records the identifier of the coordinator contained within it (i.e., the sender of that message) and treats that process as the coordinator.

If a process receives an `election` message, it sends back an `answer` message and begins another election, unless it has begun one already.

When a failed process is `revived`, it begins an election. If is has the highest process identifier, then it will decide that it is the coordinator, and announce this to the other processes. Thus is will become the coordinator, even though the current coordinator is functioning. It is for this reason that the the algorithm is called the *bully* algorithm.

The object of this practical is to specify and implement this algorithm to see if it meets the specifications apparently implied.

Part 1: Specification

Assume all of the above actions, except for `fail`, `revive` and `test` have been hidden as internal, and do not therefore play a part in our specifications. Assume, furthermore, that there are additional channels

```
channel tock

channel leader:Proc.Proc
```

the first showing the passage of time, the second an external channel of each process showing who it believes is the current coordinator, which it is prepared to communicate on whenever it is neither failed nor in mid-election.

Express the following conditions as specification processes which could be checked against your future implementation (i.e., we will test to see if the implementation, with all but the above events hidden, refines these processes).

(a) Whenever no process has revived for `T3` time units, and each `fail` has been followed by some `revive` or `test` event (the two sorts that could set an election going) at least `T3` timed units ago, then if any node believes it is the coordinator (i.e., communicates `leader.n.n`) it is the highest "living" one.

    Note that this process should allow all communications from its alphabet at all times except at those times when it is limiting who thinks it is the co-ordinator, at which times it should only allow the proper `coordinator.i` *event. So you will have to give your process, in all states, a parameter that tells it which processes are alive and which dead; and in some states you will have to give it a* `tock`*-counting parameter that allows it to tell whether it should be imposing a constraint or not.*

(b) Under the same conditions as in (a), then no node believes that any node other than the "true" one (the highest "living" one) is the co-ordinator.

(c) Under the same conditions, every living node believes that someone is the leader (though they need not agree; this is effectively saying that no election is taking place). (This is a failures specification, while the other two are trace specifications.)

(d) Under the same conditions, every living node believes that the "true" leader really is. (This is effectively the conjunction of the previous three specifications.)

(The assumption here is that any node can choose to `test` its coordinator whenever it likes. It is clearly unreasonable to expect nodes to have the right answer when either too little time has passed since a disruptive event or when no test of the coordinator has occurred that would have found it dead. This is why we have assumed that the `test` events have been left visible.)

Do you think this is a reasonable specification?

Part 2: Implementation

You should implement the algorithm. The result should be deadlock free, and satisfy an appropriate timing consistency check (see notes on timing below).

Given the freedom with which the nodes potentially run, it seems to be necessary, for the avoidance of deadlock, to make the nodes continually receptive of almost all the messages that might be directed at them by another node. This even applies when the node has failed: see the examples below.

You will find it necessary to decide what the algorithm does in subtle situations not directly catered for in the English description: for example, what happens when a node receives an `election` message when it is in the middle of distributing the message that it is the coordinator. *Following the text too literally and putting an* `answer` *in led my version into a curious misbehaviour; the right answer seems to be to reply with the appropriate* `coordinator` *message.*

Timing I recommend taking a minimalist approach to introducing timing into this system. In particular, in my version I always assumed that any response a process could generate to a message, or sequence of messages to announce an election or new coordinator, happen before the next `tock`. Clearly this could be weakened, but you would get a more complex model with more states. What I did have to do to avoid some oddities and failures of timing consistency was to assume that no process generates more than one `test` of the coordinator in any time interval, and that at least one `tock` occurs between any process failing and reviving.

Examples In order to give you a start, the following are a few states from my version of this protocol. You can copy them if you like, or follow your own ideas. The first shows a process that believes it is the coordinator:

```
RunAsCoord(n) = tock -> RunAsCoord(n)
                [] test?k:below(n)!n -> ok.n.k -> RunAsCoord(n)
                [] coordinator?k:above(n)!n -> Running'(n,k)
                [] election?k:below(n)!n ->
                                answer.n.k -> BeginElection(n)
                [] fail.n -> Failed(n)
                [] leader.n.n -> RunAsCoord(n)
```

Note the ways it lets time pass but always responds “instantly” to any test that is made of it.

The second state is when the process has just failed:

```
Failed(n) = tock -> Failed'(n)
            [] coordinator?k:above(n)!n -> Failed(n)
            [] election?k:below(n)!n -> Failed(n)
            [] test?k:below(n)!n  -> Failed(n)
            [] ok?k:above(n)!n  -> Failed(n)
            [] answer?k:above(n)!n  -> Failed(n)
            [] fail.n -> Failed(n)

Failed'(n) = tock -> Failed'(n)
            [] coordinator?k:above(n)!n -> Failed'(n)
            [] election?k:below(n)!n -> Failed'(n)
            [] test?k:below(n)!n  -> Failed'(n)
            [] ok?k:above(n)!n  -> Failed'(n)
            [] answer?k:above(n)!n  -> Failed'(n)
            [] fail.n -> Failed(n)
            [] revive.n -> (if n==N-1
                             then BecomeCoordinator(n)
                             else BeginElection(n))
```

The difference between the `Failed(n)` and `Failed'(n)` is just that, in the first one, no revival is possible because it is too soon since the failure. I used the sets `above(n)` and `below(n)` to represent the sets of processes with higher and lower indices, as all the communication channels of the algorithm point exclusively "upstream" or "downstream".

Part 3: Analysis

I do not insist that your implementation meets all the specifications set out in Part 1. What I would like to see is the trace (before the hiding of the internal events) you get when any of them fails for reasonable values of `T1`, `T2` and `T3`. (Because of the simplified timing assumptions I made it was OK to have `T1=T2=1` though really better to have `T1=1` and `T2=2`, but you may have done things differently.)

What you should set out to achieve is that in any failure you find can point the finger back at the original text of the algorithm rather than at your own implementation.

If you do find failures of the specifications (as, for several of them, I did with my implementation), discuss your findings and say whether you think the algorithm is at fault or the specification, or what?

I found that I could handle systems up to size 4 easily, and 5 reasonably, and that interesting failures (both of the deadlock and consistency checks, and the Part 1 specifications) often appeared as I increased `N` from 3 to 4.

A.W.R.
February 1998