Proactive Leader Election in Asynchronous Shared Memory Systems

M C Dharmadeep*, K Gopinath**

Computer Science and Automation,
Indian Institute of Science,
Bangalore, India. 500012

Abstract. In this paper, we give an algorithm for fault-tolerant proactive leader election in asynchronous shared memory systems, and later its formal verification. Roughly speaking, a leader election algorithm is proactive if it can tolerate failure of nodes even after a leader is elected, and (stable) leader election happens periodically. This is needed in systems where a leader is required after every failure to ensure the availability of the system and there might be no explicit events such as messages in the (shared memory) system. Previous algorithms like DiskPaxos[1] are not proactive.

In our model, individual nodes can fail and reincarnate at any point in time. Each node has a counter which is incremented every period, which is same across all the nodes (modulo a maximum drift). Different nodes can be in different epochs at the same time. Our algorithm ensures that per epoch there can be at most one leader. So if the counter values of some set of nodes match, then there can be at most one leader among them. If the nodes satisfy certain timeliness constraints, then the leader for the epoch with highest counter also becomes the leader for the next epoch(stable property). Our algorithm uses shared memory proportional to the number of processes, the best possible. We also show how our protocol can be used in clustered shared disk systems to select a primary network partition. We have used the state machine approach to represent our protocol in Isabelle HOL[3] logic system and have proved the safety property of the protocol.

1 Introduction and Motivation

In certain systems, a leader is required after every failure to ensure progress of the system. This can be solved in shared memory systems by electing a leader every period. This problem admits a trivial solution: let T mod n be the leader for T^{th} epoch, where n is the number of nodes. There are problems with this solution.

* email: dharma@csa.iisc.ernet.in
** email: gopi@csa.iisc.ernet.in

– Electing a different leader for each epoch is costly. For example, in clustered shared disk systems, recovery has to be done every time primary network partition changes. We need a "stable" leader election[7]: failures of nodes other than the leader should not change the leader.
– Failed nodes are also elected as leaders in some epochs.

In this paper, we use the concept of a proactive leader election in the context of asynchronous shared memory systems. In such an election, a leader renews a lease every so often. The stable leader election[7], appropriate in asynchronous network systems, differs from proactive leader election, as in the latter no failures are needed to trigger leader election; leaders are "elected" on a regular beat. This is needed in the systems we are interested in such as clustered shared disk systems where there are no interrupts to notify events that happen through the shared medium; all the communication is through shared disks.

Network partition is a possibility in a clustered shared disk system. A primary network partition is selected to ensure consistency of data on the shared disk. This is ensured by using a rule such as: the number of nodes in the primary network partition is at least $\lfloor \frac{n}{2} \rfloor + 1$, where n is the total number of nodes in the cluster.

Another way of selecting a primary network partition is by the use of fence devices. Fencing is used in clustered shared disk systems to prevent nodes in the non-primary network partitions from accessing disks (to ensure mutual exclusion for shared resources such as disks). Examples of fencing methods are: reboot nodes not in primary partition, disable ports corresponding to nodes not in the primary partition in the network, etc. However, even in this case, there is still a chance that all nodes get fenced. Moreover, once a primary partition is selected, if the nodes in primary network partition fail, then the system comes to halt even if there are nodes that are alive in a different partition.

Consider a simple case of a network containing two nodes.

– Suppose we assign unequal weights to the nodes and require the total weight of nodes in the primary partition to be strictly greater than that of the smaller one. If the node with higher weight dies, the system comes to halt in spite of the node with lower weight being alive.
– Suppose fencing is used. Let us suppose fencing succeeds after a network partition and a primary partition is selected. Now if the only node in primary network partition dies, then the system comes to halt.

Networks with higher number of nodes can also run into similar situations.

In this paper, we give an algorithm for fault-tolerant proactive leader election in asynchronous shared memory systems, and later its formal verification. For example, in the second case, because of leases used in our algorithm, even if the node in the primary partition dies and the other node has already been fenced, in the subsequent epochs, the previously fenced node will get elected as the leader and the system becomes available again. Our work has been inspired by

Paxos[2] and DiskPaxos[1] protocols; however, these protocols do not have the lease framework[1] incorporated into them.

We assume a model in which individual nodes can fail and reincarnate (this happens if the fencing method is reboot in clustered shared disk systems) at any point in time. Nodes have access to reliable shared memory[2]. There is no global clock accessible to all the nodes, but each node has access to a local counter which is incremented every T secs (whose accuracy has to be within the limits of a drift parameter) and restarts the protocol; for the current leader, this is extending the lease. The set of nodes which have the same counter value are said to be in the same epoch. It is quite possible that at a given instant, different nodes are in different epochs. In this paper, we propose a protocol that elects a leader for each epoch. We guarantee that there is at most one leader per epoch. Moreover, if the nodes in the system satisfy certain timeliness conditions, then the leader for the epoch with highest counter value also becomes the leader for the next epoch.

The rest of the paper is organized as follows. In section 2, we describe the related work. In section 3, we describe the model and algorithm informally. In section 4, we give details of the algorithm. In section 5, we discuss the encoding in Isabelle and the main invariant used in proving the safety property. In section 6, we discuss implementation issues. And we conclude with section 7.

2 Related Work

It is is well-known that there is no algorithm to solve consensus in asynchronous systems [13,14,15]. Failure detectors have been proposed to solve the problem in weaker models. The failure detector Ω for asynchronous network systems can be used to implement a leader oracle. Roughly speaking, the leader oracle running at each node outputs a node which it thinks is operational at that point in time. Moreover, if the network stabilizes after a point, then there is a time after which all operational nodes output the same value. Implementations of the the leader oracle and failure detectors in asynchronous network systems augmented with different kinds of assumptions are described in several works including [2,9,10,11].

The leader oracle (Ω) augmented with view numbers introduced in [7] is similar to the problem considered here. Here, in addition, a view changes if either the current leader, or the network links or both do not satisfy certain timeliness conditions. But we are interested in asynchronous shared memory systems that are more suited for clustered shared disk systems. Consensus in asynchronous shared memory systems with various failure detectors is studied

[1] The Paxos paper mentions the use of leases but no details are given.

[2] We believe this not to be a serious constraint. There are methods for implementing fault tolerant wait free objects from atomic read/write registers[5]; they can be used for constructing reliable shared memory. Also, in clustered shared disk systems, shared "memory" can be realized with some extra effort by using hot swappable mirrored disk (RAID) devices with hot spares.

in [12]. DiskPaxos[1], which has inspired this work, is similar to our protocol except that it does not have the lease framework.

Light weight leases for storage centric coordination is introduced in [6] that requires $O(1)$ shared memory (independent of the number of nodes). Their paper assumes a model similar to the timed asynchronous model [18] with the safety property involving certain timeliness conditions. We prove the safety property of our protocol assuming asynchronous shared memory model but it requires n units of shared memory. This matches the lower bound in Chockler and Malkhi[16], where they introduce a new abstract object called *ranked register* to implement the Paxos algorithm and show that it cannot be realized using less than n read/write atomic registers.

3 An Informal Description of the Model and Algorithm

We consider a distributed system with $n > 1$ processes. Each node has a area allocated in the shared memory (actually, a shared disk) to which only it can write and other nodes can read. Processes can fail and reincarnate at any point in time. We call a system stable[3] in $[s, t]$ if at all times between s and t the following hold:

- The drift rate between any two nodes or drift rate of any node from real time is bounded by δ.
- The amount of time it takes for a single node to read a block from the shared memory and process it or write a block to the shared memory is less than r secs.
- The time it takes to change the state after a read or write is negligible compared to r.

We require the second assumption, because in our case shared disk serves as the shared memory. We assume that the system is stable infinitely often and for a sufficient duration so that leader election is possible. We assume that each of the nodes know the value of δ and r. Each node has access to a local timer which times out every T secs. We assume $T >> 3nr(1 + \delta)$; this will be motivated later.

The counter value of each node is stored in local memory as well in the shared memory; it is first written to the shared memory area and then written to the local memory. The counter values of all nodes are initialized to 0. When a process reincarnates, it reads the counter value from its shared memory area and then starts the timer. When the timer at a node expires, it increments its counter value and restarts the timer.

Each node is associated with a node id which is drawn from the natural number set. We assume that each node knows the mapping *nodeid* between the shared memory addresses and the node ids. We assume that the shared memory is reliable[1].

[3] Please note that this is different from the meaning of stable in "stable leader election". Context should make clear what is being meant.

3.1 Safety Property

The safety property of the protocol requires that if a node with counter value v becomes leader, the no other node with counter value v ever becomes leader.

3.2 Informal Description of Algorithm

Each block in shared memory allocated to a node consists of a counter value, a ballot number and proposed leader node id. When the counter of a node is incremented, it starts the protocol for the new epoch. During each of the phases, if a node finds a block with higher counter value, it sleeps for that epoch.

- In Phase 0, each node reads the disk blocks of all other nodes and moves to phase 1.
- In Phase 1, a node writes a ballot number to the disk. It chooses this ballot number that is greater than any ballot number read in the previous phase. If none of the blocks read after writing to the disk have a higher ballot number, the node moves to phase 2. If the node finds a block with higher ballot number, it restarts Phase1. This phase can viewed as selecting a node which proposes the leader node id.
- In Phase 2, a node proposes the node id of the leader and writes it to disk. This value is chosen so that all nodes that have finished the protocol for the current epoch agree on the same value. If none of the blocks read after writing the proposed value to the disk have a higher ballot number, the node completes the protocol for this epoch. If there is a block with a higher ballot number, it goes back to Phase1. If the proposed value is same as that of this node id, this node is the leader. Otherwise, it sleeps for this epoch.

4 The Algorithm

Each node's status (*nodestatus*) can be in one of the five states: *Suspended, Dead, Leader, PreviousLeader, Participant.*

- A node is in *Suspended* state, if it withdrew from the protocol for the current epoch.
- A node is in *Dead* state, if it has crashed.
- A node is in *Leader* state, if it is the leader for the current epoch.
- A node is in *PreviousLeader* state, if it was the leader for the previous epoch and is participating in the protocol for the current epoch.
- A node is in *Participant* state, if it is participating in the protocol for the current epoch and is not the leader for the previous epoch.

A block in the shared memory location allocated to a node is of form *(ctrv_d, pbal, bal, val)*, where *ctrv_d* is the counter value of the node (in the shared memory (actually, *disk*)) which has write permission to it, *pbal* is the proposed ballot number of that node, and *bal* is equal to the proposed ballot number *pbal*

for which *val* was recently set. After each read or write to the shared memory, depending of whether some condition holds or not, the system moves from one phase to another. In each of the phases, *phase0*, *phase1* and *phase2*, before reading the blocks from the shared memory, each node clears its existing blocks read in the previous phase. To make our algorithm concise, we have used the phrase *"Node n rereads disk blocks of all nodes"* in each of the phases; this operation need not be atomic in our model.

We use $disk\ s\ n$ to represent the block of node n in the shared memory in state s. Also, let $blocksRead\ s\ n$ represent the blocks of nodes present at node n in state s. The state of the system is made up of: *state* of each of the nodes, *blocks* of each of the nodes in shared memory, *phase* of each of the nodes, the counter value of each of the nodes and *blocksRead* of each of the nodes.

Let $A(disk\ s\ n)$ denote the projection of component A of block $disk\ s\ n$. For example, $pbal(disk\ s\ n)$ denotes the *pbal* component of block $disk\ s\ n$. Similarly, $A(B :: set)$ denotes the set composed of projection of component A of all blocks in set B. We use $ctrv\ s\ n$, $nodestatus\ s\ n$ and $phase\ s\ n$ to denote the counter value, state and phase of node n in state s respectively. Note that *ctrv* is the value at the node whereas *ctrv_d* is the value at the disk.

There is an implicit extra action in each phase (omitted in the given specification for brevity): $Phase\{i\}Read\ s\ s'\ n\ m$, which says that node n in $phase\{i\}$ reads the block of node m and the system moves from state s to state s'. State variables not mentioned in a state transition below remain unchanged across the state transition.

For facilitating the proof, we use a history variable *LeaderChosenAtT*, but actually not needed in the algorithm: $LeaderChosenAtT\ s\ t = k$ if k is the leader for epoch t in state s. Also, $LeaderChosenAtT\ s'\ t = LeaderChosenAtT\ s\ t$ unless the value for t is changed explicitly.

Phase0

Action: Node n (re)reads disk blocks of all nodes including itself. Changes the state to *Suspended* if there exists a node with higher counter value. Otherwise, moves to phase 1. Formally,

Case: $\exists\ br \in blocksRead\ s\ n.\ ctrv\_d(br) > ctrv\ s\ n$
Outcome: $nodestatus\ s'\ n = Suspended.$

Case: $\neg\exists\ br \in blocksRead\ s\ n.\ ctrv\_d(br) > ctrv\ s\ n.$
Outcome: $phase\ s'\ n = 1$

Phase1

Action: Write a value greater than *pbal*s of all blocks read in previous phase to the disk. Formally,
$disk\ s'\ n = (ctrv\ s\ n,\ pbal',\ bal(disk\ s\ n),\ val(disk\ s\ n))$
where $pbal' = \text{Max}(pbal(blocksRead\ s\ n)) + 1.$
Node n rereads disk blocks of all the nodes.

If there exists a block with higher counter value, move to *Suspended* state. If there exists a block with higher *pbal*, restart phase 1. Otherwise, move to phase 2. Formally,

Case: $\exists\ br\ \in\ blocksRead\ s\ n.\ ctrv\_d(br)\ >\ ctrv\ s\ n.$
Outcome: $nodestatus\ s'\ n\ =\ Suspended.$

Case: $\exists\ br\ \in\ blocksRead\ s\ n.\ ctrv\_d(br)\ =\ ctrv\ s\ n$
$\&\ ((pbal(br)\ >\ pbal(disk\ s\ n))$
$\mid\ (pbal(br)\ =\ pbal(disk\ s\ n)$
$\&\ nodeid(br)\ >\ n))$
Outcome: Node n restarts *Phase1.*

Case: $\neg\exists\ br\ \in\ blocksRead\ s\ n.\ ctrv\_d(br)\ >\ ctrv\ s\ n$
$\mid\ ((ctrv\_d(br)\ =\ ctrv\ s\ n)$
$\&\ (pbal(br)\ >\ pbal(disk\ s\ n)))$
$\mid\ ((ctrv\_d(br)\ =\ ctrv\ s\ n)$
$\&\ (pbal(br)\ =\ pbal(disk\ s\ n))$
$\&\ (nodeid(br)\ >\ n))$
Outcome: $phase\ s'\ n\ =\ 2\ .$

Phase2

Action: Write the proposed leader node id to the disk, where the node id is chosen as follows: if no other node with same counter value has proposed a value, set it to this node id; otherwise, set it to the value of the block with highest *bal* whose proposed value is non-zero. Formally,

$disk\ s'\ n\ =\ (ctrv\ s\ n,\ pbal(disk\ s\ n),\ pbal(disk\ s\ n),\ proposedv)$
where $proposedv\ =$
n if $(\forall\ br\ \in\ blocksRead\ s\ n.\ ctrv\_d(br)\ =\ ctrv\ s\ n$
$\longrightarrow\ val(br)\ =\ 0)$
else
m where $(m\ =\ val(br)$
$\&\ bal(br)\ =\ \mathrm{Max}(bal(\{br|\ br\ \in\ blocksRead\ s\ n$
$\&\ ctrv\_d(br)\ =\ ctrv\ s\ n$
$\&\ val(br)\ \neq\ 0\})))$

Node n rereads the blocks of all the nodes.
If there exists a node with higher counter value, move to *Suspended* state. If there exists a node with higher *pbal* restart from phase 1. Otherwise, if the proposed node id is same as the id of this node, this node is the leader. If the proposed node id is not same as the id of this node, move to *Suspended* state. Formally,

$\underline{Case:}\ \exists\ br\ \in\ blocksRead\ s\ n.\ ctrv\_d(br)\ >\ ctrv\ s\ n.$
$\underline{Outcome:}\ nodestatus\ s'\ n\ =\ Suspended.$

$\underline{Case:}\ \exists\ br\ \in\ blocksRead\ s\ n.\ ctrv\_d(br)\ =\ ctrv\ s\ n$
$\&\ ((pbal(br)\ >\ pbal(disk\ s\ n)$
$|\ (pbal(br)\ =\ pbal(disk\ s\ n)$
$\&\ nodeid(br)\ >\ n))$
$\underline{Outcome:}$ Node n restarts *Phase1*.

$\underline{Case:}\ \neg\exists\ br\ \in\ blocksRead\ s\ n.\ ctrv\_d(br)\ >\ ctrv\ s\ n$
$|\ ((ctrv\_d(br)\ =\ ctrv\ s\ n)$
$\&\ (pbal(br)\ >\ pbal(disk\ s\ n))$
$|\ ((ctrv\_d(br)\ =\ ctrv\ s\ n)$
$\&\ (pbal(br)\ =\ pbal(disk\ s\ n))$
$\&\ (nodeid(br)\ >\ n))$
$\underline{Outcome:}$ if $val(disk\ s\ n)\ =\ n$
then $nodestatus\ s'\ n\ =\ Leader$
$\&\ LeaderChosenAtT\ s'\ (ctrv\ s\ n)\ =\ n$
else $nodestatus\ s'\ n\ =\ Suspended$).

Fail

$\underline{Outcome:}\ nodestatus\ s'\ n\ =\ Dead$

ReIncarnate

$\underline{Outcome:}\ nodestatus\ s'\ n\ =\ Suspended$

IncrementTimer

$\underline{Case:}$Node n timer expires.
If this node is the leader in previous epoch, update the counter value on disk and move to phase 2. Otherwise, update the counter value on disk, reset *bal* and *val* on disk and move to phase 0.Formally,

$\underline{Outcome:}$ if $nodestatus\ s\ n\ =\ Leader$
then $nodestatus\ s'\ n\ =\ PreviousLeader$
$disk\ s'\ n\ =$
$(ctrv\ s\ n\ +\ 1,\ pbal(disk\ s\ n),\ bal(disk\ s\ n),$
$val(disk\ s\ n))$
$\&\ \ ctrv\ s'\ n\ =\ ctrv\ s\ n\ +\ 1$
$\&\ \ phase\ s'\ n\ =\ 2$
else $nodestatus\ s'\ n\ =\ Participant$
$\&\ disk\ s'\ n\ =\ (ctrv\ s\ n\ +\ 1,\ pbal(disk\ s\ n),\ 0,\ 0)$
$\&\ \ ctrv\ s'\ n\ =\ ctrv\ s\ n\ +\ 1$
$\&\ \ phase\ s'\ n\ =\ 0$

Note that with our protocol, it is quite possible that a particular block on disk and the block corresponding to it in *blocksRead* of some node do not match. But this doesn't compromise the safety property mentioned below. However, for

any node n, if the block corresponding to $disk\ s\ n$ is in its $blocksRead$, it will be same as that of $disk\ s\ n$.

safety property: $LeaderChosenAtT\ s\ t \neq\ 0\ \longrightarrow$
$\forall\ s',\ m.\ ((\ m\ \neq\ LeaderChosenAtT\ s\ t$
$\&\ ctrv\ s'\ m\ =\ t)$
$\longrightarrow\ nodestatus\ s'\ m \neq\ Leader)$

The safety property of the protocol says that if a node with counter value T becomes leader then no other node with counter value T ever becomes leader.

To ensure liveness of the protocol in Timed Asynchronous Model, one can use leader election oracle mentioned in [6] with Δ equal to T, and δ equal to r, to choose a node in *IncrementTimer*. If a node is not elected by the leader oracle, it sleeps for approximately $3nr(1+\delta)$ secs and then starts the protocol. If the leader oracle succeeds in electing a single leader, that particular node has to write to its block and read all other blocks, at most thrice, so the execution would take at most $3nr(1+\delta)$ in a stable period. Actually, this number can be reduced to $(n+1)r(1+\delta)$, if the output of the leader oracle in previous epoch is same as that of leader's id for current epoch. This is because the previous epoch leader directly moves to *phase2* after it increments its timer. So, if no other node is in *Participant* state when the previous epoch leader is participating in the protocol, it at most has to write to its block twice and read blocks of all other nodes once. This would take at most $(n+1)r(1+\delta)$ secs in a stable period. While proving the safety property of the protocol in asynchronous model, the timing constraints are not required. Hence, in the actual specification of the algorithm in Isabelle, we have not encoded the timing constraints.

5 Encoding in Isabelle and its proof

We have used the state machine approach to specify the protocol in Isabelle[3]. The encoding of the state machine in Isabelle is similar to the one given in [4]. Note that in *phase1* and *phase2*, we first write to the disk and then read the blocks of all nodes. Furthermore, in the specification of the algorithm above, we have used the phrase *"Node n restarts from phase1"*. We have realized this by associating a boolean variable $diskWritten$ with each node. We require it to be $true$, as a precondition for any of the cases to hold. When a node writes to the disk in *phase1* or *phase2*, it sets $diskWritten\ s'\ n$ to $true$ and sets $blocksReads'\ n$ to empty set. In addition, we require all blocks to be read as a precondition for any of the cases to hold. And by a node n restarting from phase 1, we mean that $diskWritten\ s'\ n$ is set to $false$ and $phase\ s'\ n$ is set to 1.

In the specification of the protocol in Isabelle, we have three phases while we had only two phases in the informal description. This is not essential, but we have done it for better readability of the specification. In the 3^{rd} phase, a node does nothing except changing its state to *Leader* or *Suspended*. Furthermore, in

the specification for *phase0*, we deliberately split the case 2 of *phase0* into two cases anticipating optimizations later.

The proof is by method of invariants and bottom-up. However unlike [1], the only history variable we have used is *LeaderChosenAtT*, where *LeaderChosenAtT(t)* is the unique leader, if any exists, for the epoch t, otherwise it is zero. The specification of the protocol, the invariants used and the lemmas can be found in [17]. The proof of the lemmas is quite straightforward, but lengthy because of the size of the protocol.

The main invariant used in the proof is the $AFTLE\_INV4$ & $AFTLE\_INV4k$. $AFTLE\_INV4$ requires that, if a node is in phase greater than 1 and has written its proposed value to the disk, then either $MaxBalInp$ is true or there exists a block br, either in *blocksRead*, or on disk which it is about to read, which will make this node to restart from $phase1$. Formally,

$$
\begin{aligned}
AFTLE\_INV4\ s\ \equiv\ & \\
& \forall\ p.\ ((phase\ s\ p\ >=\ 2) \\
& \qquad \&\ (diskWritten\ s\ p\ =\ True))\ \longrightarrow \\
& ((MaxBalInp\ s\ (bal(disk\ s\ p))\ p\ val(disk\ s\ p)) \\
& |\ (\exists\ br.\ ((br\ \in\ blocksRead\ s\ p) \\
& \qquad \&\ Greaterthan\ br\ (disk\ s\ p))) \\
& |\ (\exists\ n.\ ((\neg hasRead\ s\ p\ n) \\
& \qquad \&\ Greaterthan\ (disk\ s\ n)\ (disk\ s\ p)))
\end{aligned}
$$

where

- $Greaterthan\ br\ br'\ \equiv$

$$
\begin{aligned}
& ((ctrv\_d(br) > ctrv\_d(br')) \\
& |\ ((ctrv\_d(br) = ctrv\_d(br')) \\
& \quad \&\ (pbal(br) > pbal(br'))) \\
& |\ ((ctrv\_d(br) = ctrv\_d(br')) \\
& \quad \&\ (pbal(br) = pbal(br')) \\
& \quad \&\ (id(br) > id(br'))))
\end{aligned}
$$

- $MaxBalInp$ requires that, if the proposed value of node n is val, then any other node with same counter value as that of n and $(bal, nodeid)$ greater than that of node n, has val as its proposed value. Formally,

$$
\begin{aligned}
& MaxBalInp\ s\ b\ m\ val\ \equiv \\
& \qquad (\forall\ n.\ ((val > 0) \\
& \qquad\quad \&\ (ctrv\ s\ n = ctrv\ s\ m) \\
& \qquad\quad \&\ ((bal(disk\ s\ n) > b) \\
& \qquad\quad\quad |\ ((bal(disk\ s\ n) = b) \\
& \qquad\quad\quad\quad \&\ (n > m))))\ \longrightarrow \\
& \qquad\quad val(disk\ s\ n) = val) \\
& \qquad \&\ (\forall\ n.\ (\forall\ br.\ ((val > 0) \\
& \qquad\quad\quad \&\ (br\ \in\ blocksRead\ s\ n)
\end{aligned}
$$

$$
\begin{aligned}
&\& \ (ctrv\ s\ m = ctrv\ s\ n) \\
&\& \ (ctrv\_d(br) = ctrv\ s\ n) \\
&\& \ ((bal(br) > b) \\
&\quad |\ ((bal(br) = b) \\
&\qquad \& \ (nodeid(br) > m))) \ \longrightarrow \\
&val(br) = val)))
\end{aligned}
$$

– $hasRead\ s\ p\ q\ \equiv$

$$(\exists\ br\ \in\ (blocksRead\ s\ p).\ nodeid(br)\ =\ q)$$

$AFTLE\_INV4k$ requires that, if a node n is not the leader in previous epoch, then for any node distinct from n which is in phase greater than 1 and whose counter value is less than that of n, one of the following hold: its *pbal* is less than *pbal* of n, it moves to *Suspendedor Dead* state, moves to phase 1. Formally,

$AFTLE\_INV4k\ s\ \equiv$

$$
\begin{aligned}
\forall\ p.\ (\forall\ n.\ &((n \neq p) \\
&\& \ (ctrv\ s\ n > ctrv\ s\ p) \\
&\& \ (phase\ s\ p >= 2) \\
&\& \ (diskWritten\ s\ p) \\
&\& \ (val(disk\ s\ p) = p) \\
&\& \ ((phase\ s\ n > 1) \\
&\quad |\ ((phase\ s\ n = 1) \\
&\qquad \& \ (diskWritten\ s\ n)))) \ \longrightarrow \\
&((pbal(disk\ s\ n) > pbal(disk\ s\ p)) \\
&\ |\ (pbal(disk\ s\ n) = pbal(disk\ s\ p) \\
&\ \ \& \ (n > p)) \\
&\ |\ (\exists\ br\ \in\ blocksRead\ s\ p.\ Greaterthan\ br\ (disk\ s\ p)) \\
&\ |\ (\neg hasRead\ s\ p\ n)))
\end{aligned}
$$

First we proved that the invariant holds for the initial state and then we proved that if the invariant holds before a state transition, then it also holds after a state transition.

The first part of the invariant $AFTLE\_INV4$ is similar to the main invariant in [1]. The second part $AFTLE\_INV4k$ is new. We could not prove $AFTLE\_INV4$ by itself; we had to strengthen it by adding $AFTLE\_INV4k$ to be able to be prove it. The place where this invariant is needed is in *Increment-Timer*. The need for strengthening arises due to the one round optimization in the protocol. If a node A is the leader for epoch T, another node B is the leader for epoch $T+1$ with *pbal* smaller than that of A's and A increments its counter value and moves to *Phase2*, then this invariant could be violated. This is what is ruled out by $AFTLE\_INV4k$. $ATFLE\_INV4k$ says that if node A is the leader for a particular epoch, then any node other than A, which has a counter value greater than that of A and which had written to the disk in *Phase1*, has

pbal greater than that of A. We could not prove $AFTLE\_INV4k$ alone either. When *incrementTimer* event occurs, if two nodes with same *ctrv_d* are leaders in s, then this invariant could be violated. This is exactly what is ruled out by $AFTLE\_INV4$.

The following are the only assumptions we used in the proof, apart from the axioms that each of the possible values of *nodestatus* are distinct from one another. Let us denote the set of all *nodeids* by S.

$$S \neq \{\}, \quad finite\ S, \quad s \in S \longrightarrow s \neq 0$$

Note that as a consequence, our protocol holds even if the number of nodes participating in the protocol is 1. But, in this case, leader election is trivial. We need the second assumption because we are often required to use the following rule which had that assumption as one of the premises.

$$finite\ A;\ A \neq \{\};\ x \in A \Longrightarrow x <= Max\ A$$

We have used HOL-Complex logic instead of just HOL logic of Isabelle anticipating use of *real set* later.

In the protocol specification, we chose *nodeids* from the natural number set. In spite of that, we had to state that none of the nodeid's is equal to 0 as a axiom. Furthermore, for each state transition, we had to mention the state variables that do not change along with those that change. There are some results which we could not prove using Isabelle, like

$$nodestatus\ s \neq (nodestatus\ s)(n := Leader) \Rightarrow nodestatus\ s\ n \neq Leader$$

which was created during the proof of a lemma by a method named *auto*. In such cases, we had to backtrack to find a alternate path which avoids such a situation. Furthermore, we had to explicitly prove and pass certain results to the theorem prover because it could not recognize these patterns. (The method *auto* could prove these results.)

One such example is the following.

$$(\forall\ x \in P.\ Q(x)) \Longrightarrow (\forall\ x.\ (x \in P) \longrightarrow Q(x))$$

More such examples can be found in the proof given in [17].

6 Selecting a Primary Network Partition in Clustered Shared Disk Systems

One can use the above protocol to select the primary network partition in clustered shared disk systems. In the following discussion, we assume that the nodes in the same network partition are loosely time synchronized, i.e., modulo the drift parameter. Once consensus is reached on node id of the leader, each node can check if the node id is present in its membership set. If it is present, it knows that it is part of the primary partition.

Note that in the asynchronous shared memory model, it is quite possible in our protocol that two different nodes in two different network partitions become leaders for different epochs at the same time instant (due to drift), although likely to happen only infrequently in practice. But, in clustered shared disk systems, once a primary partition is selected, nodes in network partitions other than the

primary partition are fenced before the recovery is done. So even if two nodes from two different network partitions become leaders at the same time, at most one network partition would access the disk.

With existing methods, fencing does not work always correctly. In the process of implementing the protocol on Redhat Cluster GFS, we realized that there is a way in which fencing can always be made to work with Brocade fibre channel switches[4] that allow only one admin telnet login at a time. So each node can login into every switch first in a predefined order (for example in the order in which they appear in the configuration file), then check if it has been fenced in any switch, if so logout from all switches and return a error; otherwise fence all the nodes which are not in its partition, unfence ones in its partition, and once finished then logout of all the switches. Although this method works with Brocade switches, it need not work in general. Note that even this method can fail if the only node in the current primary partition fails in a two node cluster.

Our protocol requires some set of disks to be outside the fencing domain which it can use as the shared memory. We think such a scenario is not rare because when different nodes are accessing different disks, no fencing is required between them. If fencing uses the Brocade switch property, when a leader gets elected for a new epoch, it can use the fencing method mentioned above with the modification that before returning an error it unfences itself. Example two node network and the fencing method is illustrated in Figure 1.

Note as a consequence of the safety property, if the highest *ctrv_d* in the system is T, then recovery/fencing would have been done at most T times. Furthermore, once a primary partition is selected and the leader in the primary partition is in the epoch with highest counter value and no more partitions/failures occur in the primary partition, then the *leader* for this epoch will be elected as the leader for the next epoch if the system is in stable period. In this case, fencing and recovery need not be done again. Furthermore, one more optimization that could be done in *Phase0*: when a node finds that there exists a block with higher *pbal* or same *pbal* from a higher node id, it changes its state to *Suspended*. We believe similar optimizations can be done in *Phase1* and *Phase2* too. But this would require that once a node is selected as a leader, it inform the nodes in its partition through the network which otherwise can be avoided assuming nodes in same network partition are (loosely) time synchronized.

7 Conclusion

In this paper, we have given a protocol for proactive leader election in asynchronous shared memory systems. We have specified the protocol and proved the safety property of the protocol using Isabelle [3] theorem prover. We have also shown how one can use the protocol to choose a primary network partition in clustered shared disk systems. As a part of future work, we intend to specify the leader oracle protocol mentioned in [6] using Isabelle and also use it to prove

[4] Fibre Channel (FC) is a specialized data link layer for storage devices. FC switches are similar in function to gigabit ethernet switches.

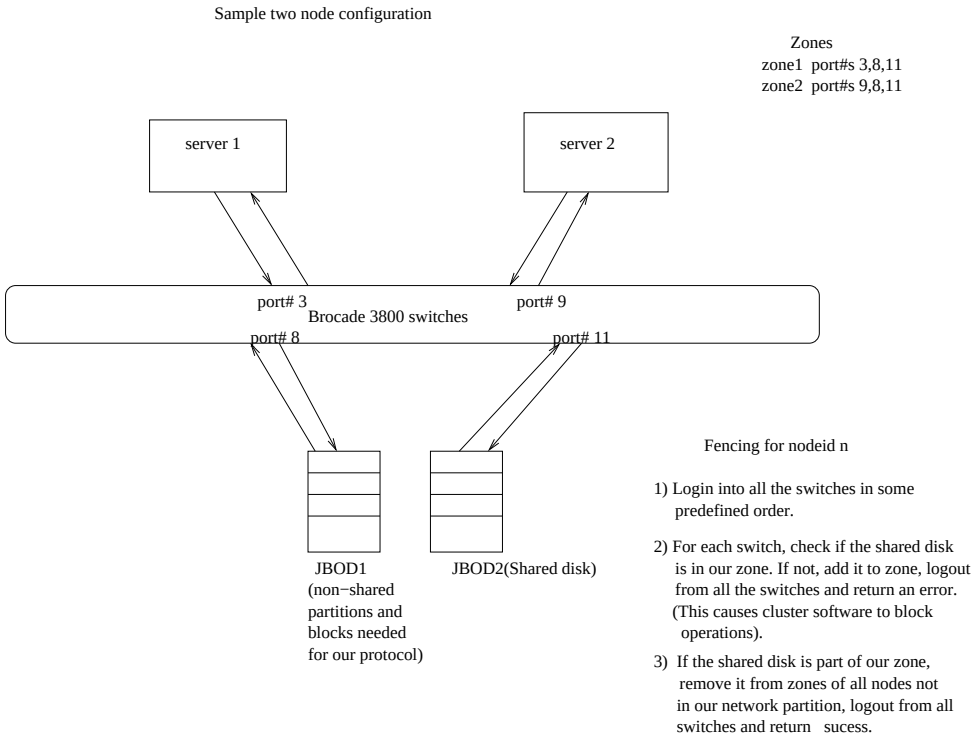

Fig. 1. Example two node cluster configuration

the liveness property of our protocol with the the leader oracle in timed asynchronous model. We also intend to incorporate the fencing part in the protocol and prove its correctness. A prototype implementation is currently in progress and we intend to experimentally understand the relationship between δ, r and the number of nodes n. The complete theory files along with the technical report are accessible at *http://agni.csa.iisc.ernet.in/~dharma/ATVA06/*.

8 Acknowledgements

We thank V.H.Gupta for reviewing the earlier draft. We thank anonymous reviewers for their comments.

References

1. E. Gafni and L. Lamport. "Disk Paxos," In Proceedings of the International Symposium on Distributed Computing, pages 330-344,2000.
2. L. Lamport. "The part-time parliament," ACM Transactions on Computer systems *16* (1998) 133-169.

3. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. "Isabelle/HOL – A Proof Assistant for Higher-Order Logic," volume 2283 of LNCS. Springer, 2002.
4. "http://afp.sourceforge.net/browser_info/current/HOL/DiskPaxos/"
5. Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. " Fault-tolerant wait-free shared objects," In Proceedings of the 33^{rd} Annual Symposium on Foundations of Computer Science, 1992.
6. Chockler, Gregory and Dahlia Malkhi. "Light-Weight Leases for Storage-Centric Coordination," MIT-LCS-TR-934 Publication Date: 4-22-2004.
7. Marcos K. Aquilera, Carole Delporte-Gallet, Huques Fauconnier and Sam Toueg. "Stable Leader Election," In Proceedings of the 15^{th} International Conference on Distributed Computing, 2001. Pages: 108-122.
8. Lampson, B. "How to build a highly available system using consensus," In *Distributed Algorithms*, ed. Babaoglu and Marzullo, LNCS 1151, Springer , 1996, 1-17.
9. R. De Prisco, B. Lampson, and N. Lynch. "Revisiting the Paxos algorithm," In Proceedings of the 11^{th} Workshop on Distributed Algorithms (WDAG), pages 11-125,Saarbrücken, September 1997.
10. M. Larrea, A. Fernández, and S. Arévalo. "Optimal implementation of the weakest failure detector for solving consensus," In Proceedings of the 19^{th} IEEE Symposium on Reliable Distributed Systems, SRDS 2000, pages 52-59, Nurenberg, Germany, October 2000.
11. F. Chu. "Reducing Ω to $\diamond W$," Information Processing Letters, 67(6):293-298, September 1998.
12. Wai-Kau Lo and Vassos Hadzilacos. "Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems," In Proceedings of the 8^{th} International Workshop in Distributed Algorithms, 1994. Pages: 280-295.
13. Michael J. Fischer, Nancy A. Lynch and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process," Journal of the Association for Computing Machinery, 32(2): 374-382, April 1985.
14. Danny Dolev, Cynthia Dwork and Larry Stockmeyer. "On the minimal synchronism needed for distributed consensus," Journal of the ACM, 34(1):77-97 , January 1987.
15. Michael C. Loui and Hosame H. Abu-Amara. "Memory requirements for agreement among unreliable asynchronous processes," In advances in Computer Research, volume 4, pages 163-183. JAI Press Inc., 1987.
16. G. Chockler and D. Malkhi. "Active Disk Paxos with Infinitely Many Processes," Proceedings of the 21^{st} ACM Symposium on Principles of Distributed Computing. (PODC), August 2002.
17. "http://agni.csa.iisc.ernet.in/~dharma/ATVA06/document.pdf"
18. F. Cristian and C. Fetzer. "The timed asynchronous system model," in Proceedings of the 28^{th} Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, June 1998, pp. 140-149.