Computer Science
Technical Report

Colorado State University

Mutual Exclusion Between Neighboring Nodes in a Tree That Stabilizes Using Read/Write Atomicity*

Gheorghe Antonoiu[1] and Pradip K. Srimani[1]

May 27, 1998

Technical Report CS-98-106

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

* A somewhat abridged version will appear in he Proceedings of **Euro-Par-98**, Soutampton, UK, 1–4 September 1998

Mutual Exclusion Between Neighboring Nodes in a Tree That Stabilizes Using Read/Write Atomicity*

Gheorghe Antonoiu[1] and Pradip K. Srimani[1]

Department of Computer Science, Colorado State University, Ft. Collins, CO 80523

Abstract. Our purpose in this paper is to propose a new protocol that can ensure mutual exclusion between neighboring nodes in a tree structured distributed system, i.e., under the given protocol no two neighboring nodes can execute their critical sections concurrently. This protocol can be used to run a serial model self stabilizing algorithm in a distributed environment that accepts as atomic operations only send a message, receive a message an update a state. Unlike the scheme in [1], our protocol does not use time-stamps (which are basically unbounded integers); our algorithm uses only bounded integers (actually, the integers can assume values only 0, 1, 2 and 3) and can be easily implemented.

1 Introduction

Because of the popularity of the serial model and the relative ease of its use in designing new self-stabilizing algorithm, it is worthwhile to design lower level self-stabilizing protocols such that an algorithm developed for a serial model can be run in a distributed environment. This approach was used in [1] and can be compared with the layered approach use in networks protocol stacks. The advantage of such a lower level self-stabilizing protocol is that it makes the job of self-stabilizing application designer easier; one can work with the relatively easier serial model and does not have to worry about message management at lower level. Our purpose in this paper is to propose a new protocol that can be used to run a serial model self stabilizing algorithm in a distributed environment that accepts as atomic operations only send a message, receive a message an update a state. Unlike the scheme in [1], our protocol does not use time-stamps (which are basically unbounded integers); our algorithm uses only bounded integers (actually, the integers can assume values only 0, 1, 2 and 3) and can be easily implemented. Our algorithm is applicable for distributed systems whose underlying topology is a tree.

It is interesting to note that the proposed protocol can be viewed as a special class of self-stabilizing distributed mutual exclusion protocol. In traditional distributed mutual exclusion protocols, self-stabilizing or non self-stabilizing (for references in self-stabilizing distributed mutual exclusion protocols, see [2] and for non self-stabilizing distributed mutual exclusion protocols, see [3, 4]), the objective is to ensure that only one node in the system can execute its critical section at any given time (i.e., critical section execution is mutually exclusive from *all other* nodes in the system; the objective in our protocol, as in [1], is to ensure that a node executes its critical section mutually exclusive from its *neighbors* in the system graph (as opposed to all nodes in the system), i.e., multiple nodes can execute their critical sections concurrently as long as they are not neighbors to each other; in the critical section the node executes an atomic step of a serial model self-stabilizing algorithm.

* A somewhat abridged version will appear in he Proceedings of **Euro-Par-98**, Soutampton, UK, 1–4 September 1998

In section 2 we describe our model precisely while in section 3 we show that the balance unbalance protocol of [2] is not self-stabilizing and describe an alternative scheme to make it self-stabilizing to motivate the development of our self-stabilizing protocol for a tree graph in section 3. We conclude the paper in section 4.

2 Model

We model the distributed system, as used in this paper, by using an undirected graph $G = (V, E)$. The nodes represent the processors while the symmetric edges represent the bidirectional communication links. We assume that each processor has its unique *id*. Each node x maintains one or more local state variables and one or more local state vectors (one vector for each local variable) that are used to store copies of the local state variables of the neighboring nodes. Each node x maintains an integer variable S_x denoting its *status*; the node also maintains a local state vector LS_x where it stores the copies of the status of its neighbors (this local state vector contains d_x elements, where d_x is the degree of the node x, i.e. x has d_x many neighbors); we use the notation $LS_x(y)$ for the local state vector element of node x that keeps a copy of the local state variable S_y of neighbor node y. The *local state* of a node x is defined by its local state variables and its local state vectors. A node can both read and write its local state variables and its local state vectors; it can only read (and not write) the local state variables of its neighboring nodes and it does neither read nor write the local state vectors of other nodes. A configuration of the entire system or a system state is defined to be the vector of local states of all nodes.

Next, our model assumes read/write atomicity of [2] (as opposed to the composite read/write atomicity as in [5, 6]). An atomic step (move) of a processor node consists of either reading the a local state variable of one of its neighbors (and updating the corresponding entry in the appropriate replica vector), or some internal computation, or writing of one of its local state variables; any such move is executed in finite time. Execution of a move by a processor may be interleaved with moves by other nodes – in this case the moves are concurrent. We use the notation $\mathcal{N}(x) = \{n_x[1], \ldots . n_x[d_x]\}$ to denote the set of neighbors of node x.

The process executed by each node x consists of an infinite loop of a finite sequence of moves:

$$R_x^1 \ : \ LS_x(n_x[1]) = S_{n_x[1]}$$

$$R_x^2 \ : \ LS_x(n_x[2]) = S_{n_x[2]}$$

$$\cdots \ : \ \cdots$$

$$R_x^{d_x} \ : \ LS_x(n_x[d_x]) = S_{n_x[d_x]}$$

$$CS_x \ : \ \text{if } \Phi_x \textbf{ then } \text{execute CriticalSection}$$

$$W_x \ : \ S_x = f_x(v_x)$$

In the first part of the loop the local state variables of all neighbors are read and stored in the local vector elements $LS_x(y), y \in \mathcal{N}(x)$; the node x then evaluates a predicate Φ_x (involving variables S_x and $LS_x(y), y \in \mathcal{N}(x)$) and if the predicate is true, executes its critical section (CS); lastly, the node x computes its new local state variable as a function of its current local state (S_x and $LS_x(y), y \in \mathcal{N}(x)$) and writes on its local state variable S_x. Note that each move in the sequence is an atomic operation; on a single node these moves are serial, while moves on different nodes may be concurrent. We also assume a distributed message passing paradigm: to read the state of a neighbor, a node sends a query message to its neighbor and receives a reply in finite time; when a node x receives a query message, it sends back (to the requesting node) the last written value on S_x; note that since individual read and write operations are atomic, any read request from any neighbor always reads the most currently recorded state S_x of a node x.

Remark 1. The objective of mutual exclusion is that no two neighboring nodes can execute the critical section **concurrently**. So, the mutual exclusion definition in a distributed system requires a common (global) time for all nodes in the system; the physical time may be used for this purpose though this time may not be explicitly known to the participating nodes.

Any operation (move) by a node is executed in a physical time interval $(x, y]$; unused time intervals may or may not exist between two consecutive operations. The time intervals for the operations executed by the same node are always disjunctive, but the time intervals for operations executed at different nodes may overlap; in this case the operations are concurrent. To read the state of a neighbor, a node sends a query message to its neighbor and updates its local register when the response is received (Note that it takes finite time for the request to reach the neighbor and it also takes finite time for the response to reach back the requesting node)[1].

Definition 1. *The read operation R_x^i, performed by node x, **reads from** the operation W_y, performed by node $y = n_x[i]$ if the value used to update $LS_x(n_x[i])$ is written by W_y.*

Consider an arbitrary execution (sequence of moves by the nodes) in the distributed system. We need to generate a serial execution equivalent to the distributed execution. We can use the time of completion for the moves to generate a total order $<$, on the set of all moves in the system (node id is used to resolve ties). However, this ordering relation does not have the property that any read operation reads from the corresponding precedent write operation.

We note that in case of a read operation, the value read by the requesting node depends on when the request reaches the neighbor; the delay for the response to go back to the requesting node does not matter; thus we can define the completion time of the read move by the time the request reaches the neighbor.

[1] To avoid the deadlock we assume that query messages are processed by a distinct process at each node; the main process and the communication process may use any local locking mechanism to serialize the access to the local state variables

Definition 2. *Consider an arbitrary execution. The normal form execution is an execution that has (i) the same start and completion time for all write and CS operations (ii) the completion time for every read operation is the time when the query messages arrived at the corresponding neighbor.*

Remark 2. Since the critical sections are executed at the same time in both executions, it is enough to prove that our mutual exclusion algorithm works for normal form executions only.

Thus, we can assume that all executions are in a normal form. We now use the time of completion for the moves to generate a total order $<$, on the set of all moves in the system (node id is used to resolve ties). This total order relation has the property that any read operation of node x from node y, **reads from** the immediately preceding (in the sense of relation $<$) write operation of node y. This property allow us to define a global system state such that when a node moves the new system state depends only on the present state of the system and the move (this property is somewhat similar with Markov property for the probabilistic systems).

Since the operations at a node are executed in the order they appear in the infinite loop, they also preserve the same order in the serial execution. However, the presented algorithms work if a weaker assumption is made, the fairness of an execution.

Definition 3. *An infinite execution is fair if it contains a infinite number of actions for any type.*

Remark 3. The purpose of our protocol is to ensure mutual exclusion between neighboring nodes. Each node x can execute its critical section iff the predicate Φ_x is true at node x. Thus, in a legitimate system state, mutual exclusion is ensured iff

$$\Phi_x \Rightarrow \forall y \mid y \text{ is a neighbor of } x, \quad \neg\Phi_y$$

i.e., as long as node x executes its CS, no neighbor y of node x can execute its CS ($\Phi_y \mid (y$ is a neighbor of $x)$ is false).

3 Self-Stabilizing Mutual Exclusion Protocol for Two Processes Without Shared Memory

The purpose of our protocol is to ensure self-stabilizing mutual exclusion between neighboring nodes that do not share any common memory and that can communicate only by exchanging messages. To be more precise, each node has a critical section in its code and the node shall be able to execute the critical section in a mutually exclusive fashion with the other node; to execute its critical section each node computes a predicate of its state and the state of the other node, and if the predicate is true it executes the critical section.

The concept of a global legitimate (legal) state (or a set of such states) of the system is an integral part of any self-stabilizing algorithm since that defines the

desired property of the system. Thus, the legitimate state is defined by what we desire from the system. Our objective in the present paper is to ensure mutual exclusion between any two neighboring nodes (processes); so in any legitimate state of the system, if one node is executing its critical section, the other node cannot execute its critical section, or in other words, if the predicate is true at one node, the predicate at other node is not true or cannot be true until the first node completes execution of its critical section.

3.1 The Balance Unbalance Protocol [2] is not Self-Stabilizing

The well known algorithms [7–9, 3, 4] for mutual exclusion between two processes are not applicable in our case since each of them either uses a shared variable or is not self-stabilizing. The balance-unbalance algorithm, [10, 2], does not use any shared variable, does ensure mutual exclusion, but is not self-stabilizing. The two processes in the algorithm [2] roughly execute the following loops:

Process A:
R_a: $LS_a(b) = S_b$;
CS_a: **if**$(S_a = LS_a(b))$ **then** Execute Critical Section
W_a: **if**$(LS_a(b) = S_a)$ **then** $S_a = (S_a + 1) \bmod 2$
Process B:
R_b: $LS_b(a) = S_a$;
CS_b: **if**$(S_b \neq LS_b(a))$ **then** Execute Critical Section
W_b: **if**$(S_b \neq LS_b(a))$ **then** $S_b = (S_b + 1) \bmod 2$

Note that S_a and $LS_a(b)$ are local variables to process A; process A can write on both S_a and $LS_a(b)$ and process B can only read S_a. Similarly, S_b and $LS_b(a)$ are local variables to process B: process B can write on both S_b and $LS_b(a)$ and process A can only read S_b. Note that the processes use read/write atomicity and they are not uniform (two processes execute different codes). Also observe that the variables maintained by the processes are binary variables. If the process A sees a balanced link, $(LS_a(b) = S_a)$, it unbalances it; if the process B sees a unbalanced link, $(LS_b(a) \neq S_b)$, it balances it.

It can be easily shown that the protocol is not self-stabilizing, i.e., there is no proper subset of states such that for any initial state the system reaches that subset after a finite number of moves.

3.2 Self-Stabilizing Balance Unbalance Protocol for a Pair of Processes

One possible approach to make the above balance unbalance protocol self-stabilizing is presented in [2]; this approach uses shared variables. We present an alternative approach without using any shared variables. The structure of the two processes A and B remain the same, i.e. infinite loop at each processor consists of an atomic read operation, critical section execution if certain predicate is true and an atomic write action. As before, S_a and $LS_a(b)$ are two local variables maintained by process A ($LS_a(b)$ is the variable maintained at process A to store a copy of the state of its neighbor process B); process A can write on both S_a

and $LS_a(b)$ and process B can only read S_a. Similarly, S_b and $LS_b(a)$ are local variables to process B: process B can write on both S_b and $LS_b(a)$ and process A can only read S_b. The difference is that the variables are now ternary, i.e., they can assume values 0, 1 or 2. The proposed algorithm is shown in Figure 1:

Process A	Process B
R_a: $LS_a(b) = S_b$;	R_b: $LS_b(a) = S_a$;
CS_a : **if**$(S_a = 0)$ **then** Execute Critical Section	CS_b : **if**$(S_b = 1)$ **then** Execute Critical Section
W_a: **if**$(LS_a(b) = S_a)$ **then** $S_a = S_a + 1 \bmod 3$	W_b: $S_b = LS_b(a)$;

Fig. 1. Self-Stabilizing Balance Unbalance Protocol for a Pair of Processes

Since each of the processes A and B executes an infinite loop, after a R_a action the next "A" type action is W_a, after a W_a action the next "A" type action is R_a, after a R_b action the next "B" type action is W_b, after a W_b action the next "B" type action is R_b and so on.

Remark 4. We are not interested in the actions CS_a or CS_b since they do not change the system state as far as our algorithm for mutual exclusion is concerned (we are interested in proving the mutual exclusive execution of the critical sections by the two processes in a self-stabilizing way, but do not care what is done in the critical section); we assume execution of critical section by either process takes finite time.

Remark 5. An execution of the system is an infinite execution of the processes A and B and hence an execution of the system contains an infinite number of each of the actions from the set $\{R_a, W_a, R_b, W_b\}$; thus, the execution is *fair*.

The system may start from an arbitrary initial state and the first action in the execution of the system can be any arbitrary one from the set $\{R_a, W_a, R_b, W_b\}$. Note that the global system state is defined by the variables S_a and $LS_a(b)$ in process A and the variables S_b and $LS_b(a)$ in process B.

Remark 6. When a process makes a move (the system executes an action), the system state may or may not be changed. For example, in a system state where $S_a \neq LS_a(b)$, the move W_a does not modify the system state, i.e., the system remains in the same state after the move.

Definition 4. *A move (action) that modifies the system state is called a* **modifying action**.

Our objective is to show that the system, when started from an arbitrary initial state (possibly illegitimate), converges to a legitimate state in finite time (after finitely many actions by the processes). We introduce a new binary relation.

Definition 5. *We use the notation $x \succeq y$, if $x = y$ or $x = (y+1) \bmod 3$, where $x, y \in Z_3$.*

Remark 7. The relation $\succeq$ is neither reflexive, nor transitive, nor symmetric. For example, $1 \succeq 0$, $2 \succeq 2$, $2 \not\succeq 0$, $2 \succeq 1$, $0 \not\succeq 1$, etc.

Definition 6. *Consider the ordered sequence of variable $(S_a, LS_b(a), S_b, LS_a(b))$; a system state is* **legitimate** *if (i) $S_a \succeq LS_b(a) \wedge LS_b(a) \succeq S_b \wedge S_b \succeq LS_a(b)$ and (ii) if at most one pair of successive variables in the previous sequence are unequal.*

Example 1. For example, $\{S_a = 1,\ LS_a(b) = 0,\ S_b = 0,\ LS_b(a) = 1\}$ is a legitimate state while $\{S_a = 2,\ LS_a(b) = 1,\ S_b = 1,\ LS_b(a) = 0\}$ is not.

Remark 8. It is easy to note that for a legitimate system state, only four possibilities exist: (1) $S_a = LS_b(a) = S_b = LS_a(b)$, (2) $S_a = LS_b(a) + 1,\ LS_b(a) = S_b = LS_a(b)$, (3) $S_a = LS_b(a),\ LS_b(a) = S_b + 1,\ S_b = LS_a(b)$, (4) $S_a = LS_b(a) = S_b,\ S_b = LS_a(b) + 1$. Note: all additions are modulo 3.

Theorem 1. *In a legitimate state the two processes A and B execute their critical sections in mutual exclusive way, i.e., if process A is executing CS then process B cannot execute CS and vice versa, i.e. $S_a = 0 \Rightarrow S_b \neq 1$ and $S_b = 1 \Rightarrow S_a \neq 0$.*

Proof. The proof is obvious since in a legitimate state $S_a \succeq LS_b(a) \wedge LS_b(a) \succeq S_b \wedge S_b \succeq LS_a(b)$ and at most one pair of successive variables in the sequence can be unequal.

Theorem 2. *Any arbitrary move from the set $\{R_a, W_a, R_b, W_b\}$ made in a legitimate state of the system leads to a legitimate state after the move.*

Proof. Since there are only four possible moves, it is easy to check the validity of the claim. For example, consider the move R_a; the variable $LS_a(b)$ is affected; if $LS_a(b) = S_b$ before the move, this move does not change the system state; if $LS_a(b) \neq S_b$ before the move, then the system state before the move must satisfy $S_a = LS_b(a) = S_b$ (since it is legitimate) and after the move it will satisfy $S_a = LS_b(a) = S_b = LS_a(b)$ (hence, the resulting state is legitimate).

Lemma 1. *Any infinite fair execution contains infinitely many modifying actions (see Definition 4).*

Proof. By contradiction. Assume that after a finite number of moves S_a does not change its value anymore. Then after a complete loop executed by process B, $LS_b(a) = S_a$ and $S_b = S_a$. In the next loop the process A must move, which contradicts our assumption. It is easy to see that if S_a changes its value infinitely many times, any other variable changes its value infinitely many times.

Lemma 2. *For any given fair execution and for any initial state, a state such that three variables from the set $\{S_a,\ LS_b(a),\ S_b,\ LS_a(b)\}$ are equal each other is reached in a finite number of moves.*

Proof. Consider the first move that modifies the state of S_a. After this move $S_a \neq LS_a(b)$. To change again the value of S_a, the $LS_a(b)$ variable must change its value and become equal to S_a. But $LS_a(b)$ always takes the value of S_b. Since S_a change its value infinitely many times a state such that $S_a = LS_a(b)$ and $LS_a(b) = S_b$ is reached in a finite number of moves.

Lemma 3. *For any given fair execution and for any initial state, a state such that $S_a \neq LS_a(b)$, $S_a \neq S_b$, $S_a \neq LS_b(a)$, is reached in a finite number of moves.*

Proof. Since we use addition modulo 3, the variables S_a, $LS_b(a)$, S_b, $LS_a(b)$, can have values in the set $Z_3 = \{0, 1, 2\}$. When three variables from the set $\{S_a, LS_b(a), S_b, LS_a(b)\}$ are equal to each other, Lemma 2, there is a value $i \in Z_3$ such that $S_a \neq i$, $LS_a(b) \neq i$, $S_b \neq i$, $LS_b(a) \neq i$. When $LS_b(a)$, S_b, $LS_a(b)$ change their values, they only copy the value of one variable in the set $\{S_a, LS_b(a), S_b, LS_a(b)\}$. Thus, when S_a reaches for the first time the value i, the condition $S_a \neq LS_a(b)$, $S_a \neq S_b$, $S_a \neq LS_b(a)$ is met.

Theorem 3. *For any given fair execution, the system starting from any arbitrary state reaches a legitimate state in finitely many moves.*

Proof. The system reaches a state such that $S_a \neq LS_a(b)$, $S_a \neq S_b$, $S_a \neq LS_b(a)$ in a finite number of moves, Lemma 3. Let $S_a = i \in Z_3$ Since $LS_b(a) \neq i$, $S_b \neq i$ and $LS_a(b) \neq i$, S_a can not change its value until $LS_a(b)$ becomes equal to i, $LS_a(b)$ can not become equal to i until S_b becomes equal to i and S_b can not become equal to i until $LS_b(a)$ becomes equal to i. Thus, S_a can not modify its state until a legitimate state is reached.

4 Self-Stabilizing Mutual Exclusion Protocol for a Tree Network Without Shared Memory

Consider an arbitrary rooted tree; the tree is rooted at node r. We use the notation d_x for the degree of node x, $n_x[j]$ for the j-th neighbor of the node x, $\mathcal{N}(x) = \{n_x[1], \ldots . n_x[d_x]\}$ for the set of neighbors of node x, $\mathcal{C}(x)$ for the set of children of x and P_x for the parent of node x; since the topology is a tree each node x knows its parent P_x and for the root node r, P_r is Null.. As before, each node x maintains a local state variable S_x (readable by its neighbors) and a local state vector LS_x used to store the copies of the states of its neighbors; we use notation $LS_x(y)$ to denote the component of the state vector LS_x that stores a copy of the state variable S_y of node y, $\forall y \in \mathcal{N}(x)$. All variables are now modulo 4 integers (we explain the reason later). We assume that each node x maintains a height variable H_x such that $H_r = 0$ for the root node and for $\forall x \neq r$, H_x is the number of edges in the unique path from node x to the root node. It is easy to see that if the root node sets $H_r = 0$ and any other node x sets its H_x to $H_{P_x} + 1$ (level of its parent plus 1), the height variables will correctly indicate the height of each node in the tree after a finite number of moves, starting from

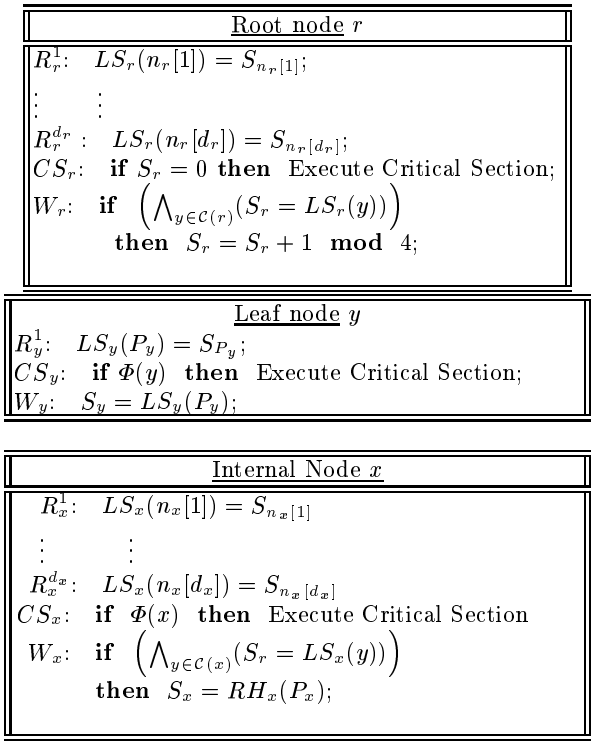

Fig. 2. Protocol for an Arbitrary Tree

any illegitimate values of those variables. To avoid cluttering the algorithm, we do not include the rules for handling H_x variable in our algorithm specification. As before, the root node, internal nodes as well as the leaf nodes execute infinite loops of reading the states of neighbor(s), executing critical sections (if certain predicate is satisfied) and writing its new state. The protocols (algorithms) for root, internal nodes and leaf nodes are shown in Figure 2 where the predicate $\Phi(x)$ is:

$$\Phi(x) = (S_x = 0 \wedge (H_x \ is \ even)) \vee (S_x = 2 \wedge (H_x \ is \ odd))$$

Note, as before, the state of a node x is defined by the variable S_x and the vector LS_x; the global system state is defined by the local states of all participating nodes in the tree.

Definition 7. *Consider a link or edge (x, y) such that node x is the parent of node y in the given tree. The state of a link (x, y), in a given system state, is defined to be the vector $(S_x, LS_y(x), S_y, LS_x(y))$. The state of a link is called* **legitimate** *iff $S_x \succeq LS_y(x) \wedge LS_y(x) \succeq S_y \wedge S_y \succeq LS_x(y)$ and at most one pair of successive variables in the vector $(S_x, LS_y(x), S_y, LS_x(y))$ are unequal.*

Definition 8. *The system is in a legitimate state if all links are in a legitimate state.*

Theorem 4. *For an arbitrary tree in any legitimate state, no two neighboring processes can execute their critical sections simultaneously.*

Proof. In a legitimate state (when the H variables at nodes have stabilized) for any two neighboring nodes x and y, we have (either L_x is even & L_y is odd), or (L_x is odd & L_y is even). Hence, $\Phi(x)$ and $\Phi(y)$ are simultaneously (concurrently) true iff $S_x = 2$ and $S_y = 0$ or $S_x = 0$ and $S_y = 2$. But since link (x, y) is in a legitimate state, such condition can not be met; hence, two neighboring nodes cannot execute their critical sections simultaneously.

Lemma 4. *Consider an arbitrary link (x, y). The node x modifies the value of the variable S_x infinitely many times, if and only if the node y modifies the value of the variable S_y infinitely many times.*

Proof. If node x modifies S_x finitely many times, then after a finite number of moves the value of S_x is not modified anymore. The next complete loop of node y after the last modification of S_x, makes $LS_y(x) = S_x$ and $LS_y(x)$ is not be modified by subsequent moves. Hence, after at most one modification, S_y remains unchanged. Conversely, if node y modifies S_y finitely many times then after a finite number of moves the value of S_y is not modified anymore. The next complete loop of node x after the last modification of S_y, makes $LS_x(y) = S_y$ and $LS_x(y)$ is not be modified by subsequent moves. Hence, after at most one modification, S_x remains unchanged.

Lemma 5. *For any fair execution, variable S_r at root node r is modified infinitely many times.*

Proof. By contradiction. Assume that the value of S_r is modified finitely many times. Then, after a finite number of moves, the value S_y for each child y of r will not be modified anymore, Lemma 4. Repeating the argument, after a finite number of moves no S or LS variables for any node in the tree may be modified. Consider now the leaf nodes. Since the execution is fair, the condition $S_z = LS_z(P_z)$ must be met for each leaf node z. If this condition is met for leaf nodes it must be met for the parents of leaf nodes too. Repeating the argument, the condition must be met by all nodes in the tree. But, if this condition is met, the root node r modifies its S_r variable in its next move, which is a contradiction.

Lemma 6. *For any fair execution and for any node x, the variable S_x is modified infinitely many times.*

Proof. If node x modifies its variable S_x finitely many times, its parent, say node z, must modify its variable S_z only finitely many times, Lemma 4. Repeating the argument, the root node also modifies its variable S_r finitely many times, which contradicts Lemma 5.

Lemma 7. *Consider an arbitrary node z ($\neq r$). If all links in the path from r to z are in a legitimate state, then these links remain in a legitimate state after an arbitrary move by any node in the system.*

Proof. Let (x, y) be an link in the path from r to z. Since (x, y) is in a legitimate state, we have $S_x \succeq LS_y(x) \wedge LS_y(x) \succeq S_y \wedge S_y \succeq LS_x(y)$ and at most one pair of successive variables in the sequence $(S_x, LS_y(x), S_y, LS_x(y))$ are unequal. We need to consider only those system moves (executed by nodes x and y) that can change the variables S_x, $LS_y(x)$, S_y, $LS_x(y)$. Considering each of these moves, we check that legitimacy of the link state is preserved in the resulting system state. The read moves (that update the variables $LS_y(x)$, $LS_x(y)$) obviously preserve legitimacy. To consider the move W_x, we look at two cases differently:

Case 1 ($\mathbf{x} = r$): When W_x is executed, S_x can be modified (incremented by 1) only when $S_x = LS_x(y)$. Thus, since the state is legitimate, a W_x move can increment S_x only under the condition $S_x = LS_y(x) = S_y = LS_x(y)$ and after the move the link (x, y) remains in a legitimate state.

Case 2 ($\mathbf{x} \neq r$): Since the link (t, x) (where node t is the parent of x, i.e., $t = P_x$) is in a legitimate state, we have $LS_x(t) = S_x$ or $LS_x(t) = (S_x + 1) \bmod 4$. When W_x is executed, S_x can be modified only by setting its value equal to $LS_x(t)$; hence, after the move the link (x, y) remains in a legitimate state.

Lemma 7 shows that if a path of legitimate links from the root to a node is created the legitimacy of the links in this path is preserved for all subsequent states. The next lemma shows that a new link is added to such a path in finite time.

Lemma 8. *Let (x, y) be a link in the tree. If all links in the path from root node to node x are in a legitimate state, then the link (x, y) becomes legitimate in a finite number of moves.*

Proof. First, we observe that the node y modifies the variable S_y infinitely many times, Lemma 6. Then, we use the same argument as in the proof of Theorem 3 to show that the link (x, y) becomes legitimate after a finite number of moves.

Theorem 5. *Starting from an arbitrary state, the system reaches a legitimate state in finite time (in finitely many moves).*

Proof. The first step is to prove that all links from the root node to its children become legitimate after a finite number of moves. This follows from the observation that each child x of the root node modifies its S variable infinitely many times and from an argument similar to the argument used in the proof of Theorem 3. Using Lemma 8, the theorem follows.

5 Conclusion

In this paper we have proposed a self-stabilizing protocol for ensuring mutually exclusive critical section execution from neighboring nodes in a tree structured

message passing distributed system. The proposed protocol can be readily used to provide a suitable run time or execution time environment to run self stabilizing algorithms developed for serial model. Our protocol is id-based and does not use any shared variable as opposed to the self-stabilizing traditional mutual exclusion algorithm [2] which is anonymous and does use shared link registers. Our protocol is based on read/write atomicity of operations like [2] and operates under a distributed demon. To compare our algorithm with that of [1], we do not use any time stamp to implement the scheme; one immediate improvement is that we use only bounded integers (in fact, the variables in our scheme can have only values 0, 1, 2 and 3) while time stamps in [1] are essentially unbounded integers. It'd be interesting to extend the concepts developed in this paper to an arbitrary system graph.

References

1. M. Mizuno and H. Kakugawa. A timestamp based transformation of self-stabilizing programs for distributed computing environments. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG'96)*, volume 304–321, 1996.
2. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.
3. M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, Cambridge MA, 1986.
4. P. K. Srimani and S. R. Das, editors. *Distributed Mutual Exclusion Algorithms*. IEEE Computer Society Press, Los Alamitos, CA, 1992.
5. M. Flatebo, A. K. Datta, and A. A. Schoone. Self-stabilizing multi-token rings. *Distributed Computing*, 8:133–142, 1994.
6. S. T. Huang and N. S. Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 1993.
7. E. W. Dijkstra. Solution of a problem in concurent programming control. *Communication of the ACM*, 8(9):569, September 1965.
8. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):107–118, August 1974.
9. L. Lamport. The mutual exclusion problem: Part II – statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.
10. H. S. M. Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Inf. Processing Letters*, 8(2):91–95, 1979.