Lock Based Self-Stabilizing Distributed Mutual Exclusion Algorithms

Masaaki Mizuno* Mikhail Nesterenko Hirotsugu Kakugawa†

Dept. of Comp. and Info. Sciences
Kansas State University
Manhattan, Kansas 66506
{masaaki,mikhail}@cis.ksu.edu

Dept. of Elect. Engineering
Hiroshima University
Higashi-Hiroshima, Japan
kakugawa@se.hiroshima-u.ac.jp

Abstract

In 1974, Dijkstra introduced the notion of self-stabilization and presented a token circulation distributed mutual exclusion (DMX) protocol as the first self-stabilizing (SS) algorithm. Since then, many variations of SS DMX algorithms have been presented. Most, if not all, of these algorithms impose stronger assumptions on their execution environments than those provided by common distributed systems. Independently, non SS DMX algorithms have been studied extensively in the last 15 years. This paper presents two SS DMX algorithms that are based on existing non SS DMX algorithms: one is based on a link-locking algorithm and the other is on a node-locking algorithm. Our algorithms assume execution environments that are close to those provided by common distributed systems. Furthermore, they provide better synchronization delays than token circulation SS DMX algorithms. We have implemented our algorithms and tested them with various initial configurations.

Keywords: *distributed systems, fault-tolerance, mutual exclusion, self-stabilization*

1 Introduction

A distributed system consists of a set of processes (nodes) and communication links that connect the processes. The processes cooperate with each other by exchanging messages through the links. Dijkstra introduced the notion of *self-stabilization* in the context of distributed systems [4]. A system is defined to be self-stabilizing (SS) with respect to a set of legitimate states if, regardless of its initial state, the system is guaranteed to arrive at a legitimate state in a finite number of steps and will never leave legitimate states after that. Thus, an SS system need not be initialized and is able to recover from transient failures by itself. Here, transient failures are failures that may change a state of a system to a non-legitimate state. They include corrupting local states of processes, message channels, and shared memory.

The first SS program that Dijkstra introduced was a mutual exclusion protocol based on token circulation in a ring network. In this algorithm, the execution environment has stronger assumptions than environments that common distributed systems can provide; therefore, the protocol may not run in a real distributed system without major modifications. Because of the historical perspective that Dijkstra's algorithm provided and the importance of their role in distributed systems, many SS distributed mutual exclusion (DMX) algorithms have been proposed. Some of these algorithms are designed based on more relaxed assumptions on their execution environments [2, 5]. However, most of these assumptions are still stronger than environments that common distributed systems provide.

Independently, non SS DMX algorithms have been studied extensively in the last 15 years. Almost all proposed algorithms assume a fully connected network, and much effort has been made to improve two performance characteristics under this assumption: the number of messages required for a critical section (CS) entry and (2) synchronization delay, which is the number of rounds of message passing required for one node to leave the CS and another to enter the CS. Several taxonomic studies and generalizations of the algorithms have been published [10, 12, 14]. Generally, the algorithms are classified into two major categories: token based and permission based (also called non-token based) algorithms. We use the term "lock based" to repre-

*This work was supported in part by the National Science Foundation under Grant CCR-9201645.

†This work was supported in part by the Telecommunications Advancement Foundation.

sent the latter category. In general, token based algorithms require less messages than lock based algorithms. However, it tends to be more difficult for token based algorithms to recover from failures since not only node failures but also token loss/duplication must be considered. We classify lock based algorithms into two sub-classes: link-locking algorithms and node-locking algorithms. Ricart and Agrawala's [11] and Carvalho-Roucairol's [3] algorithms are in the former class, and Lamport's [7] and Maekawa's [8] algorithms are in the latter class.

In this paper, we present two lock based SS DMX algorithms; one is based on the link-locking approach and conceptually similar to Ricart and Agrawala's algorithm, and the other is based on the node-locking approach and similar to Lamport's algorithm. In the process of the development, we carefully sorted out differences and similarities between the link-locking and node-locking algorithms. Then, we developed our SS DMX algorithms based on this study. As a result, the two algorithms are structurally different, but interestingly, details of the algorithms are almost identical. As a price of self-stabilization, they generally require more message overhead than Ricart and Agrawala's and Lamport's algorithms.

Some error recovery methods have been presented for some of these non SS DMX algorithms. They assume such errors as node failures and communication line failures. On the other hand, these methods do not consider many transient errors. Thus, self-stabilization provides a complementary approach to other methods of fault tolerance [13].

Unlike many other SS DMX algorithms, our algorithms are not based on token circulation. Thus, their synchronization delays are better than many proposed algorithms. More importantly, our algorithms assume execution environments that are close to those provided by common distributed systems. We have actually implemented both algorithms on our distributed/parallel algorithm development system [9]. Both implementations were tested with many different initial configurations.

The organization of the paper is as follows: Section 2 reviews lock based non SS algorithms. In particular, differences and similarities between Ricart and Agrawala's and Lamport's algorithms are discussed. Section 3 presents our SS DMX algorithms in detail. The correctness proofs of the algorithms are given in Section 4, and Section 5 discusses performance of the algorithms.

2 Review of lock based non SS DMX algorithms

2.1 System model

In most non SS DMX algorithms, the following assumptions are made: A distributed system consists of a set of n nodes $\mathcal{P} = \{P_1, \cdots, P_n\}$ that are connected by a communication network. There is no shared memory in the system. Thus, the nodes exchange information only via message passing. The network is assumed to be logically fully connected; that is, for every pair of nodes P_i and P_j ($i \neq j$) in $\mathcal{P}$, there exists a bidirectional logical communication link, denoted $L_{i,j}$ or $L_{j,i}$. The network guarantees error-free FIFO delivery with bounded message delay. Note that in our SS protocols described in Section 3, this assumption is relaxed and the network is assumed to have transient errors, even though it still guarantees FIFO delivery.

2.2 Taxonomy of lock based algorithms

We classify the lock based algorithms into the following two sub-categories:

1. Link-locking algorithms: In order for a node P_i to enter its CS, it must lock all the communication links incident to P_i ("P_i's link" for short). A link connecting two nodes can be locked by either of these two nodes. If a node P_i has locked all of its links, there cannot be another node P_j that also locks all of its links since $L_{i,j}$ can be locked by either P_i or P_j, but not both. This property guarantees mutual exclusion.

   Ricart and Agrawala's [11] and Carvalho-Roucairol's [3] algorithms are in this category.

2. Node-locking algorithms: When a node P_i wants to enter its CS, it tries to lock a set of nodes, called P_i's *request set* (denoted R_i). Request sets of any pair of nodes in $\mathcal{P}$ have at least one node in common. Each node can be locked only by at most one node at a time. Thus, if a node P_i has locked all the nodes in R_i, there cannot be another node P_j that also locks all the nodes in R_j because of the relation $R_i \cap R_j \neq \emptyset$.

   Maekawa presented a node-locking algorithm in [8]. He presented a method of constructing efficient request sets from finite projective planes. The algorithm is general and works with any other request sets. Lamport's algorithm [7] is also in this category. In Lamport's algorithm, the request set of every node is $\mathcal{P}$. In both algorithms, each node P_i maintains a queue Q_i of requests to resolve lock contention. Note that link-locking algorithms generally do not need queues since each contention involves only two nodes which are connected by the link.

In this paper, we present two SS DMX algorithms. One is based on the link-locking approach and is conceptually similar to Ricart and Agrawala's algorithm. Another is based on the node-locking approach and is similar to Lamport's algorithm. Before presenting our algorithms, we review common and distinct features of Ricart and Agrawala's and Lamport's non SS algorithms.

In both algorithms, each node maintains a Lamport's logical clock [7]. Every message carries a clock value, called the timestamp of the message, which is the value of the clock at the time when the message is sent. When a node P_i requests to enter its CS, it sends request messages that carry the timestamp to all other nodes. The timestamp carried by the request messages is called the timestamp of the request. In the rest of the section, ts_i denotes the timestamp of the most recent request by P_i.

In both algorithms, if multiple nodes compete to lock a link or a node, the node that has issued a request with the smallest timestamp obtains the lock[1]. Therefore, the requests are granted in timestamp order[2]. This guarantees deadlock-freedom.

The next two subsections review features specific to each algorithm.

2.3 Ricart and Agrawala's algorithm

Two types of messages are exchanged between every pair of nodes: request and release messages. If a node P_i wants to enter its CS, it must lock the links $L_{i,j}$, for all $P_j \in (\mathcal{P} - \{P_i\})$.

In order for P_i to lock link $L_{i,j}$, it sends a request message to P_j. When P_j receives the request, it sends a reply message to P_i only if P_i should lock the link; that is, either P_j is not requesting to lock $L_{i,j}$ or P_j is requesting but $ts_i < ts_j$. If P_j has already owned a lock on $L_{i,j}$ or is requesting and $ts_j < ts_i$, it defers to send a reply back to P_i until it (obtains and) releases the lock. Therefore, when P_i receives a reply from P_j, it knows that it has obtained a lock on $L_{i,j}$. If P_i receives replies from all P_j in $(\mathcal{P} - \{P_i\})$, it can enter its CS.

2.4 Lamport's algorithm

Three types of messages are exchanged between every pair of nodes: request, reply, and release messages. If a node P_i wants to enter its CS, it must lock all the nodes $P_j \in \mathcal{P}$. Note that P_i must lock P_i itself, too.

In order for P_i to lock a node P_j, P_i sends a request message to P_j. When P_j receives the request, it enqueues the request in Q_j in timestamp order. If $i \neq j$, P_j sends a reply message.

P_j is locked by P_i if (1) P_i's request is at the top of Q_j and (2) it is guaranteed that no further request with a smaller timestamp will arrive at P_j. The latter condition is satisfied if P_j has received any message with a timestamp greater than ts_i from every node except for P_i. Note that reply messages are to guarantee that this condition will hold at the requesting node in finite time.

Since P_i has sent request messages with the same timestamp to all the nodes in $\mathcal{P}$, if these two conditions hold at one node, they have held or will hold at every node; that is, each node either locks all the nodes or does not lock any node. Therefore, P_i can enter its CS as soon as it detects that P_i has locked itself.

When P_i exits its CS, it sends release messages to all the nodes. When P_j receives a release message from P_i, it dequeues P_i's request from Q_j.

3 Our SS DMX algorithms

3.1 Basics

For our SS algorithms, we make the same assumptions on the underlying distributed system model as described in Section 2.1, except that transient failures may occur in the nodes and the network. In both of our protocols, each node P_i contains two processes: an application process A_i and a synchronization process S_i. A_i executes an application program that issues requests to enter its CSs. S_i communicates with other S_j s via message passing to obtain necessary locks.

A_i and S_i communicate with each other using two shared variables: "A_request$_i$" and "S_state$_i$." A_request$_i$ is *maintained* by A_i and *exported* to S_i; that is, A_i can read and modify A_request$_i$, but S_i can only read A_request$_i$. A_request$_i$ is assigned a value either ACQUIRE or RELEASE. S_state$_i$ is maintained by S_i and exported to A_i. It is assigned a value LOCKED, WAITING, or IDLE.

Handshaking synchronization is carried out between A_i and S_i:

1. When A_i wants to enter its CS, it sets A_request$_i$ to ACQUIRE.
2. When S_i detects A_request$_i$ = ACQUIRE, it sets S_state$_i$ to WAITING.
3. When S_i obtains all the necessary locks, it sets S_state$_i$ to LOCKED.
4. When A_i detects S_state$_i$ = LOCKED, it enters the CS.
5. When A_i leaves the CS, it sets A_request$_i$ to RELEASE.
6. Then, S_i sets S_state$_i$ to IDLE.

We assume that A_i does not issue a new request (by setting A_request$_i$ to ACQUIRE again) until it detects S_state$_i$ = IDLE.

[1] Node identifiers are used for tie-breaking.

[2] In Maekawa's algorithm, this property may not hold. Therefore, the algorithm is incorporated with a deadlock avoidance mechanism [8].

As in non SS lock based algorithms, each S_i maintains a Lamport's logical clock[3] [7], and every message carries a timestamp. S_i has a function "new_timestamp$_i$()" that returns a new timestamp value which is greater than any timestamps that S_i has seen. A_i maintains a variable "req_ts$_i$," which is updated to new_timestamp$_i$() when A_i issues a new request. Thus, req_ts$_i$ holds the timestamp value associated with the most recent request.

Only one type of messages is exchanged between nodes. Each message carries a mode as well as a timestamp. A mode is either Need_Reply or Need_No_Reply. When a node P_i receives a message with mode Need_Reply from P_j, P_i must send a message back to P_j. When P_i receives a message with Need_No_Reply, P_i need not return any message.

Each S_i has $n-1$ interval timers, one for each $P_j, i \neq j$. Each interval timer has a counter which is decremented by one at every clock tick. When a counter value becomes 0, the timer interrupts S_i. When interrupted, S_i knows which interval timer has caused the interrupt and sends a message to the associated node. Whenever S_i sends a message to a node P_j, the associated counter is automatically initialized to τ. This mechanism guarantees that S_i sends at least one message to each P_j in every τ time[4]. A timeout mechanism is necessary to implement SS protocols [6].

S_i sends messages in the following four occasions:

1. When A_i makes a new request, S_i sends messages with Need_Reply to all nodes[5];
2. When A_i makes a request to exit its CS, S_i sends messages with Need_No_Reply to all nodes.
3. When P_i receives a message with Need_Reply from another node P_j, S_i sends a message back to P_j with Need_No_Reply; and
4. When the interval timer associated with a node P_j expires, S_i sends a message to P_j with Need_No_Reply.

While A_i is requesting, all messages that S_i sends carry the req_ts$_i$ value. While A_i is not requesting, every message that S_i sends carries a value of new_timestamp$_i$(). Messages sent with req_ts$_i$ are considered to be request messages or messages that confirm S_i's current requests. The new_timestamp$_i$() values carried by other messages guarantee that in case the receiving processes hold some pending requests, S_i will *lose* lock contention to the nodes that issued these requests.

[3] Node identifiers are used for tie-breaking. It is assumed that node identifiers are not affected by any transient failures.

[4] We assume that the time required to send a message is negligible.

[5] More precisely, the definition of "all nodes" differs in the two algorithms.

```
Coordinator thread c_i
RA01 loop
RA02  if (A_request_i=ACQUIRE) then
RA03     if (S_state_i=IDLE) then
RA04        if (for all k, lock_state_i[k] =IDLE) then
RA05           req_ts_i := new_timestamp_i();
RA06           S_state_i := WAITING fi;
RA07     else
RA08        if (for all k, lock_state_i[k] =LOCKED) then
RA09           S_state_i := LOCKED
RA10        else S_state_i := WAITING; fi; fi
RA11  else    /* A_request_i=RELEASE */
RA12     S_state_i := IDLE fi
RA13 endloop

Link thread l_i[j]
RB01 *[ S_state_i=WAITING and lock_state_i[j]=IDLE →
RB02      lock_state_i[j] := WAITING;
RB03      send_request()
RB04   []received message(msg_ts, mode) from P_j →
RB05      if S_state_i≠IDLE then
RB06         if (msg_ts > req_ts_i) then
RB07            lock_state_i[j] := LOCKED
RB08         else lock_state_i[j] := WAITING fi; fi;
RB09      if (mode=Need_Reply) then
             inform_timestamp(P_j) fi
RB10   []S_state_i=IDLE and lock_state_i[j] ≠ IDLE →
RB11      lock_state_i[j] := IDLE;
RB12      send_release()
RB13   []timeout(P_j) →inform_timestamp(P_j)
RB14   ]

RC01 procedure send_request()
RC02    send(req_ts_i, Need_Reply) to P_j

RD01 procedure send_release()
RD02    send(new_timestamp_i(), Need_No_Reply) to P_j

RE01 procedure inform_timestamp(P_j)
RE02    if (lock_state_i[j] =IDLE) then
RE03       send(new_timestamp_i(), Need_No_Reply) to P_j
RE04    else send(req_ts_i, Need_No_Reply) to P_j fi
```

Figure 1. Link Locking Algorithm

3.2 A link-locking approach

Conceptually, the protocol is similar to Ricart and Agrawala's algorithm. Figure 1 shows the details of the algorithm. Statement $*[G_1 \rightarrow S_1[] \cdots []G_m \rightarrow S_m]$ in the figure represents an infinite loop construct, where each G_k is a predicate called a *guard* and S_k is a (compound) statement. In each iteration, all the guards are evaluated. Then, one guard is nondeterministically selected among the guards that are evaluated to true, and the associated statement is executed.

A synchronization process S_i is implemented by n independent threads: one coordinator thread c_i and array of $(n-1)$ lock threads $l_i[1..n, i \neq j]$. Coordinator thread c_i coordinates the behavior of the lock threads. Lock thread $l_i[j]$ locks link $L_{i,j}$.

In addition to variables req_ts$_i$ and S_state$_i$, S_i maintains array variable "lock_state$_i[1..n]$" (lock_state$_i[i]$ is not used).

Variable lock_state$_i[j]$ is assigned a value IDLE, WAITING, or LOCKED by $l_i[j]$.

Handshaking synchronization is carried out among A_i, c_i, and all $l_i[j]$s. When c_i finds that A_request$_i$ has become ACQUIRE, it sets S_state$_i$ to WAITING (refer to RA06 in Figure 1). Note that at this moment, c_i updates req_ts$_i$ to a value of new_timestamp$_i$() (RA05). When lock thread $l_i[j]$ finds that S_state$_i$ has become WAITING (RB01), it sets lock_state$_i[j]$ to WAITING (RB02) and sends a request message to P_j (RB03). Lock_state$_i[j]$ is set to LOCKED when $l_i[j]$ has obtained the lock (RB07). When c_i finds that all entries of lock_state$_i$[ ] become LOCKED, it sets S_state$_i$ to LOCKED to inform A_i that A_i can enter its CS (RA09).

When c_i finds A_state$_i$ = RELEASE, it sets S_state$_i$ to IDLE (RA12). When Lock thread $l_i[j]$ finds that S_state$_i$ has become IDLE, it sets lock_state$_i[j]$ to IDLE (RB11). Note that c_i can work on a new request (by setting req_ts$_i$ to new_timestamp$_i$() and S_state$_i$ to WAITING) only after it confirms that all lock_state$_i[j]$s have become IDLE (RA04).

The algorithm differs from Ricart and Agrawala's algorithm in that a reply is always sent back immediately whenever a process receives a request message (a message with Need_Reply) (RB09). Upon a receipt of the reply, the requesting node checks whether it has locked the link by comparing the timestamps (RB04-RB06). Since there are no pending reply messages, when A_i exits its CS, each $l_i[j]$ sends a message with a new_timestamp$_i$() value to $l_j[i]$ (RB12) to grant a lock to $l_j[i]$ in case $l_j[i]$ has a pending request.

3.3 A node-locking approach

Conceptually, this protocol is similar to Lamport's algorithm. Figure 2 presents details of the algorithm for a synchronization process S_i. It is almost identical to the algorithm for $l_i[j]$ in Figure 1. Differences are:

1. Each S_i is implemented as one process, rather than multiple threads; and
2. S_i maintains a queue Q_i of requests.

Three operations are applied to Q_i.

1. update($Q_i, \langle j, ts_j\rangle$): The operation enqueues request $\langle j, ts_j\rangle$ in Q_i in timestamp order, where $\langle j, ts_j\rangle$ indicates that the request is issued by P_j and its associated timestamp is ts_j. If there is another request $\langle j, ts'_j\rangle$ by P_j in Q_j, it deletes $\langle j, ts'_j\rangle$ before $\langle j, ts_j\rangle$ is enqueued. If there is a garbage request $\langle k, ts_k\rangle$ in Q_i such that $k < 1$ or $n < k$, it deletes the request.
2. topid(Q_i) and topts(Q_i): These operations return the node identifier and the timestamp of the request at the top of Q_i, respectively.

```
Synchronization Process S_i
LA01  *[ A_request_i=ACQUIRE and S_state_i=IDLE →
LA02       req_ts_i := new_timestamp_i();
LA03       S_state_i := WAITING;
LA04       send_request()
LA05    []received a message(msg_ts, mode) from P_j →
LA06       update(Q_i, ⟨msg_ts, P_j⟩);
LA07       if A_request_i=ACQUIRE then
LA08          if (topid(Q_i)=P_i and topts(Q_i)=req_ts_i) then
LA09             S_state_i := LOCKED
LA10          else S_state_i := WAITING fi fi;
LA11       if (mode=Need_Reply) and (i ≠ j) then
              inform_timestamp(P_j) fi
LA12    []A_request_i=RELEASE and S_state_i≠IDLE →
LA13       S_state_i := IDLE;
LA14       send_release()
LA15    []timeout(P_j) →inform_timestamp(P_j)
LA16    ]

LB01  procedure send_request()
LB02     for all nodes in 𝒫 do send(req_ts_i, Need_Reply)

LC01  procedure send_release()
LC02     tmp_ts := new_timestamp_i();
LC03     for all nodes in 𝒫 do
            send(tmp_ts, Need_No_Reply)

LD01  procedure inform_timestamp(P_j)
LD02     if (S_state_i=IDLE) then
LD03        send(new_timestamp_i(), Need_No_Reply) to P_j
LD04     else send(req_ts_i, Need_No_Reply) to P_j fi
```

Figure 2. Node Locking Algorithm

Note that there is no dequeue operation. Once S_i has received a message from each node, Q_i will always hold one request from every node. Therefore, as soon as the top of Q_i changes, S_i can grant a new lock to the request that has just moved to the top of Q_i.

When a process A_i requests to enter its CS, S_i sends messages with the same timestamp, req_ts$_i$, to all the nodes, including itself (refer to LA04:LB01-LB02 in Figure 2). Thus, each requesting node either locks all the nodes or does not lock any node, and a requesting node S_i knows that it has successfully locked all the nodes when S_i finds its own request at the top of Q_i (LA07-LA09).

When A_i exits its CS, S_i sends messages with a value of new_timestamp$_i$() to all the nodes (LA14:LC01-LC03). These messages are to delete entries $\langle j, ts_j\rangle$ at the top of all the queues; that is, their role is the same as that of release messages in Lamport's algorithm.

4 Correctness of the algorithms

4.1 Assumptions

The following assumptions are made:

- At each application process A_i, if A_request$_i$ = ACQUIRE and S_state$_i$ = LOCKED, then A_request$_i$ will become RELEASE in finite time.

- Each synchronization process S_i guarantees a weakly fair scheduling. That is, when S_i executes statement $*[g_1 \rightarrow S_1 [] \cdots [] g_m \rightarrow S_m]$, if a guard g_i becomes true and thereafter remains true, the associated statement S_i will be executed in finite time.

- Each communication link $L_{i,j} (= \ L_{j,i})$ consists of two unidirectional channels, $ch_{i,j}$ and $ch_{j,i}$. P_i (P_j) sends messages to $ch_{i,j}$ ($ch_{j,i}$) and receives messages from $ch_{j,i}$ ($ch_{i,j}$).

- Initially, each communication channel $ch_{i,j}$ may contain finite number of garbage messages, and at any process P_i, its variables and the program counter may hold arbitrary values.

4.2 Safety property

4.2.1 Basics

To prove that a given algorithm is self-stabilizing, we define a predicate $\mathcal{T}$ on the global states of an execution H of the algorithm such that the safety property holds if $\mathcal{T}$ is true. Then, we show that H is self-stabilizing with respect to $\mathcal{T}$ by showing (1) Convergence: $\mathcal{T}$ holds in finite time t in any execution H, and (2) Closure: $\mathcal{T}$ remains true after t [6]. Note that while proving the convergence and closure properties, it is assumed that no transient errors will occur; that is, the initial state of the system is considered to be the state right after some transient error has occurred.

To formally define $\mathcal{T}$, we first define a global state. A global state of an execution is defined over states (values of local variables) of the nodes and states of the communication channels (contents of messages in the channels). By following the notation in [1], we introduce two auxiliary variables for each channel $ch_{i,j}$: sent$[ch_{i,j}]$ and rec$[ch_{i,j}]$. Both are unbounded sequences of messages. Sequence sent$[ch_{i,j}]$ contains all messages that P_i sent to $ch_{i,j}$, in the order they were sent. Sequence rec$[ch_{i,j}]$ contains all messages that P_j received from $ch_{i,j}$, in the order they were received.

Initially, each channel may hold some garbage messages. We define the initial value of sent$[ch_{i,j}]$ to be empty. That is, sent$[ch_{i,j}]$ only contains messages that P_i actually has sent (initial garbage messages are not included in sent$[ch_{i,j}]$). Sequence rec$[ch_{i,j}]$ is undefined while garbage messages remain in $ch_{i,j}$. It is defined and initialized to be empty when P_j has received all initial garbage messages in $ch_{i,j}$. Sequences sent$_t[ch_{i,j}]$ and rec$_t[ch_{i,j}]$ denote values of sent$[ch_{i,j}]$ and rec$[ch_{i,j}]$ at time t, respectively. If rec$[ch_{i,j}]$ is undefined at time t, rec$_t[ch_{i,j}]$ is undefined. Since we assume FIFO delivery in each channel, rec$[ch_{i,j}]$ is always a prefix of sent$[ch_{i,j}]$. The last message of sequence $s1$ is denoted by last$(s1)$.

We define a predicate $\mathcal{T}$ as follows[6]:

Definition ($\mathcal{T}$): $\mathcal{T}$ is true if for any pair of P_i and P_j in $\mathcal{P}$, rec$[ch_{i,j}]$ is defined and contains at least one message and the compound command corresponding to the receive guard for the first message (RB05-RB09, LA06-LA11) is executed.

The second part of this predicate may sound odd. Note, however, that a value of a program counter is also a part of a state.

Theorem 1: Both the link-locking and node-locking algorithms are self-stabilizing with respect to $\mathcal{T}$.

Proof: Since there is no loop other than the main loop $*[\cdots]$ in each algorithm (RB01-RB14, LA01-LA16), whichever statement the program counter is initially pointing to, it will go back to the beginning of the main loop in finite time. Since we assume a weakly fair scheduling policy, all initial garbage messages in each channel $ch_{i,j}$ connected to S_j will be received by S_j in finite time (RB04, LA05). Thus, all the sequences rec$[ch_{i,j}]$ will be defined in finite time.

Due to a timeout timer associated with each channel, S_i sends at least one message to each of its links, $ch_{i,j}$, in every τ time. Since the weakly fair scheduling is assumed, S_j receives messages in $ch_{i,j}$ in finite time. Thus, rec$[ch_{i,j}]$ will have at least one message and the command corresponding to the receive guard for the first message will be executed in finite time.

It is obvious that once $\mathcal{T}$ becomes true, it remains true. □

For a message m, let $ts(m)$ denote the timestamp carried by m. From the protocol, it is clear that the following proposition holds:

Proposition 1: After $\mathcal{T}$ becomes true, for any pair of processes P_i and P_j in $\mathcal{P}$, for any consecutive messages m_1 and m_2 in sent$[ch_{i,j}]$ or $\mathrm{rec}[ch_{i,j}]$, $ts(m_1) \leq ts(m_2)$. □

An application process A_i is in its CS only when ((S_state$_i$ = LOCKED) $\wedge$ (A_request$_i$ = ACQUIRE)) holds. Let req_ts$_i^x$ denote the timestamp associated with the request for the P_i's x^{th} CS entry. We prove that the mutual exclusion holds in an execution of each algorithm.

4.2.2 The link-locking algorithm

Lemma 1: After $\mathcal{T}$ becomes true, at any time t when (lock_state$_i[j]$ = LOCKED) $\wedge$ (A_request$_i$ = ACQUIRE) holds for the P_i's x^{th} CS entry, req_ts$_i^x$ $< ts$(last(rec$_t[ch_{j,i}]$)).

Proof: Lock thread $l_i[j]$ changes $\mathrm{lock\_state}_i[j]$ to LOCKED for the x^{th} CS entry when (1) S_state$_i$ = WAITING (thus, A_request$_i$ = ACQUIRE for the x^{th} CS entry) and (2) $l_i[j]$ receives a message m from P_j such that req_ts$_i^x$ $< ts(m)$. Hence, if $l_i[j]$ does not receive any message until t after it receives m, then m = last(rec$_t[ch_{j,i}]$) and the lemma holds.

[6] For the link-locking algorithm, we will introduce another predicate $\mathcal{T}_l$ that will become true after $\mathcal{T}$ becomes true.

Suppose that $l_i[j]$ receives further messages before t. For any such message m' (= last(rec$_t[ch_{j,i}]$)), $ts(m) \leq ts(m')$ holds from Proposition 1. Therefore, the lemma holds. □

Proposition 2: After $\mathcal{T}$ becomes true, at any time t when (lock_state$_i[j]$ = LOCKED) holds for the P_i's x^{th} CS entry, $ts(m) \leq$ req_ts$_i^x$ for any message m in sent$_t[ch_{i,j}]$.

Proof: While ($\mathrm{lock\_state}_i[j]$ = LOCKED) holds for the x^{th} CS entry, the value in req_ts$_i$ remains equal to req_ts$_i^x$. Note that this is true even if A_i leaves the x^{th} entry and immediately makes a request for the $(x+1)^{st}$ entry while ($\mathrm{lock\_state}_i[j]$ = LOCKED) still holds (RA04). Therefore, during this period, all the messages that $l_i[j]$ sends carry timestamp req_ts$_i^x$. Thus, by Proposition 1, the proposition holds. □

In the link-locking protocol, unfortunately, $\mathcal{T}$ is not enough to guarantee mutual exclusion. Thus, we define the following $\mathcal{T}_l$.

Definition ($\mathcal{T}_l$): For any P_i and P_j in $\mathcal{P}$, ((S_state$_i$ = LOCKED) $\wedge$ (A_request$_i$ = ACQUIRE)) $\Rightarrow$ (lock_state$_i[j]$ = LOCKED).

Lemma 2: After $\mathcal{T}$ becomes true, $\mathcal{T}_l$ will become true in finite time; that is, the link-locking algorithm is self-stabilizing with respect to $\mathcal{T}_l$.

Proof: In the initial state, there may exist more than one node, P_i, such that A_request$_i$ = ACQUIRE, S_state$_i$ = LOCKED, and lock_state$_i[j]$ = LOCKED for all $P_j(i \neq j)$. Let P_s be the node with the smallest req_ts$_s$ among such nodes. When $\mathcal{T}$ holds for the first time at u, all but P_s, say $P_i(s \neq i)$, such that A_request$_i$ = ACQUIRE and S_state$_i$ = LOCKED will have lock_state$_i[j]$ = WAITING because of statement RB08. Thus, $\mathcal{T}_l$ may not hold. However, all such P_i will execute RA10 and S_state$_i$ will become WAITING in finite time, say time $t_l(u \leq t_l)$. At t_l, $\mathcal{T}_l$ holds for P_s.

We show that after t_l, $\mathcal{T}_l$ remains true. After t_l, for any P_i, a coordinator thread c_i changes S_state$_i$ to LOCKED only after it has detected (lock_state$_i[j]$ = LOCKED) for all of its lock threads. By Lemma 1, once lock_state$_i[j]$ becomes LOCKED, it will never be changed to WAITING (RB06-RB08). Thus, $l_i[j]$ changes lock_state$_i[j]$ to IDLE only after it detects (S_state$_i$ = IDLE). Therefore, (S_state$_i$ = LOCKED) $\Rightarrow$ (lock_state$_i[j]$ = LOCKED), and the Lemma holds. □

Theorem 2: After $\mathcal{T}_l$ becomes true, mutual exclusion holds in an execution of the link-locking algorithm.

Proof: In the proof of Lemma 2, we have shown that when $\mathcal{T}_l$ becomes true at t_l, at most one node is in its CS; namely, the node P_s with the smallest req_ts$_s$.

We show that at any time after t_l, for any pair of processes P_i and P_j in $\mathcal{P}$, ((S_state$_i$ = LOCKED) $\wedge$ (A_request$_i$ = ACQUIRE)) $\wedge$ ((S_state$_j$ = LOCKED) $\wedge$ (A_request$_j$ = ACQUIRE)) = false. Let $\mathcal{R}$ be ((lock_state$_i[j]$ = LOCKED) $\wedge$ (lock_state$_j[i]$ = LOCKED)). From Lemma 2, it is enough to show that $\mathcal{R}$ = false.

Suppose, on the contrary, at time t, A_i is in its x^{th} CS entry and A_j is in its y^{th} CS entry and therefore $\mathcal{R}$ holds. Since rec$_t[ch_{i,j}]$ is a prefix of sent$_t[ch_{i,j}]$, from Lemma 1 and Proposition 2, req_ts$_i^x$ $< ts$(last(rec$_t[ch_{j,i}]$)) $\leq$ req_ts$_j^y$ $< ts$(last(rec$_t[ch_{i,j}]$)) $\leq$ req_ts$_i^x$. This is a contradiction. Thus, mutual exclusion holds at all times. □

4.2.3 The node-locking algorithm

We show that unlike the link-locking algorithm, as soon as $\mathcal{T}$ becomes true, mutual exclusion holds in an execution of the node-locking algorithm.

Lemma 3: After $\mathcal{T}$ becomes true, at any time t when (S_state$_i$ = LOCKED) $\wedge$ (A_request$_i$ = ACQUIRE)) holds for the P_i's x^{th} CS entry, req_ts$_i^x$ $< ts$(last(rec$_t[ch_{j,i}]$)).

Proof: S_i changes S_state$_i$ from WAITING to LOCKED for the x^{th} CS entry at some time s when $((\mathrm{topid}(Q_i) = i) \wedge (\mathrm{topts}(Q_i) = \mathrm{req\_ts}_i^x))$ is detected. Recall that Q_i holds one request $\langle j, ts_j\rangle$ from every P_j $(i \neq j)$ behind P_i's own request, where $ts_j = ts$(last(rec$_s[ch_{j,i}]$)). Since req_ts$_i^x$ $<$ ts(last(rec$_s[ch_{j,i}]$)), if S_i does not receive any message from P_j in between s and t, last(rec$_s[ch_{j,i}]$) = last(rec$_t[ch_{j,i}]$) and the lemma holds.

Suppose that S_i receives further messages at u $(s < u \leq t)$. From Proposition 1, for any such message, $ts(\mathrm{last}(\mathrm{rec}_s[ch_{j,i}])) \leq ts(\mathrm{last}(\mathrm{rec}_u[ch_{j,i}]))$ holds. Thus, the lemma holds. □

Lemma 4: After $\mathcal{T}$ becomes true, for the P_i's x^{th} CS entry, ((S_state$_i$ = LOCKED) $\wedge$ (A_request$_i$ = ACQUIRE)) $\Rightarrow ((\mathrm{topid}(Q_i) = i) \wedge (\mathrm{topts}(Q_i) = \mathrm{req\_ts}_i^x))$.

Proof: S_i changes S_state$_i$ from WAITING to LOCKED for the x^{th} CS entry only after it detects condition $(\mathrm{topid}(Q_i) = i) \wedge (\mathrm{topts}(Q_i) = \mathrm{req\_ts}_i^x)$ while (A_request$_i$ = ACQUIRE). By Proposition 1, condition $(\mathrm{topid}(Q_i) = i) \wedge (\mathrm{topts}(Q_i) = \mathrm{req\_ts}_i^x)$ remains true until S_i receives a message from itself with a new timestamp req_ts$_i^{x+1}$. This occurs after S_i detects that A_request$_i$ has been changed from ACQUIRE to RELEASE. Thus, the lemma holds. □

It is clear that the following proposition holds.

Proposition 3: After $\mathcal{T}$ becomes true, at any time t when (S_state$_i$ = LOCKED) $\wedge$ (A_request$_i$ = ACQUIRE) holds for the P_i's x^{th} CS entry, $ts(m) \leq$ req_ts$_i^x$ for any message m in sent$_t[ch_{i,j}]$. □

Theorem 3: After $\mathcal{T}$ becomes true, mutual exclusion holds in an execution of the node-locking algorithm.

Proof: Because of statement LA10, when $\mathcal{T}$ becomes true for the first time at u, at most one node, say P_s, with the smallest req_ts$_s$ can have condition (A_request$_s$ = ACQUIRE $\wedge$ S_state$_i$ = LOCKED).

We show that any time after u, for any pair of processes P_i and P_j in $\mathcal{P}$, ((S_state$_i$ = LOCKED) $\wedge$ (A_request$_i$ = ACQUIRE)) $\wedge$ ((S_state$_j$ = LOCKED) $\wedge$ (A_request$_j$ = ACQUIRE)) = false. Let $\mathcal{R}$ be $((\mathrm{topid}(Q_i) = i)$

$\wedge$ $(\text{topts}(Q_i) = \text{req\_ts}_i^x)) \wedge ((\text{topid}(Q_j) = j) \wedge (\text{topts}(Q_j) = \text{req\_ts}_j^y))$. From Lemma 4, it is enough to show that $\mathcal{R}$ = false for any x and y.

Suppose, on the contrary, $\mathcal{R}$ holds at time t. That is, A_i is in its x^{th} CS entry and A_j is in its y^{th} CS entry at time t. Since each node has its request in every queue, $\langle i,\ ts(\text{last}(\text{rec}_t[ch_{j,i}]))\rangle$ is in Q_i and $\langle j,$ $ts(\text{last}(\text{rec}_t[ch_{i,j}]))\rangle$ is in Q_j. Sequence $\text{rec}_t[ch_{i,j}]$ is a prefix of $\text{sent}_t[ch_{i,j}]$, and requests are sorted in timestamp order in both queues. Thus, from Lemma 3 and Proposition 3, $\text{req\_ts}_i^x < ts(\text{last}(\text{rec}_t[ch_{j,i}])) \leq \text{req\_ts}_j^y < ts(\text{last}(\text{rec}_t[ch_{i,j}])) \leq \text{req\_ts}_i^x$. This is a contradiction. Therefore, mutual exclusion holds at all times. □

4.3 Liveness property

4.3.1 The link-locking algorithm

Theorem 4: After $\mathcal{T}_l$ becomes true, if ((A_request$_i$ = ACQUIRE) $\wedge$ (S_state$_i$ $\neq$ LOCKED)) holds for the P_i's x^{th} CS entry, S_state$_i$ will becomes LOCKED in finite time.

Proof: (S_state$_i$ $\neq$ LOCKED) iff ((S_state$_i$ = IDLE) or (S_state$_i$ = WAITING)). State ((A_request$_i$ = ACQUIRE) $\wedge$ (S_state$_i$ = IDLE)) will become ((A_request$_i$ = ACQUIRE) $\wedge$ (S_state$_i$ = WAITING)) as soon as statement RA06 is executed. The only case in which ((A_request$_i$ = ACQUIRE) $\wedge$ (S_state$_i$ = WAITING)) remains forever is that there exists at least one lock thread $l_j[i]$ from which $l_i[j]$ will not receive a message with timestamp ts_j such that req_ts$_i^x$ $< ts_j$. Due to timeout timers, $l_i[j]$ and $l_j[i]$ exchange messages at least once every τ time.

There are three cases to consider:

1. S_state$_j$ is IDLE: $l_j[i]$ sends a message to $l_i[j]$ with a value of new_timestamp$_j$(). Thus, req_ts$_i^x$ < ts$_j$ must hold after $l_j[i]$ receives at least one message that carries timestamp req_ts$_i^x$ from $l_i[j]$.

2. S_state$_j$ is LOCKED for the P_j's y^{th} CS entry: ((A_request$_j$ = ACQUIRE) $\wedge$ (S_state$_j$ = LOCKED)) and req_ts$_j^y$ $<$ req_ts$_i^x$ hold. By the assumption, A_request$_j$ will become RELEASE in finite time and, as a result, S_state$_j$ will become IDLE. Then, case 1 applies.

3. S_state$_j$ is WAITING for the P_j's y^{th} CS entry: If we assume that lock_state$_i[j]$ does not become LOCKED after $l_i[j]$ receives a message from $l_j[i]$, req_ts$_j^y$ < req_ts$_i^x$ must hold. State lock_state$_j[i]$ must become LOCKED after $l_j[i]$ receives one message from $l_i[j]$. If all entries of lock_state$_j$[ ] become LOCKED, S_state$_j$ become LOCKED and case 2 applies.

   If there are other lock threads in P_j that are still in state WAITING, this discussion applies inductively: There must exist a process that has issued a request with the smallest timestamp. All the threads of this process will obtain locks without waiting. Then, for the process that has issued a request with the second smallest timestamp, case 2 applies. In this way, requests are granted in timestamp order.

Hence, S_state$_i$ will become LOCKED in finite time. □

4.3.2 The node-locking algorithm

Theorem 5: After $\mathcal{T}$ becomes true, if ((A_request$_i$ = ACQUIRE) $\wedge$ (S_state$_i$ $\neq$ LOCKED)) holds for the P_i's x^{th} CS entry, S_state$_i$ will becomes LOCKED in finite time.

Proof: The only reason ((A_request$_i$ = ACQUIRE) $\wedge$ (S_state$_i$ = WAITING)) remains forever is that there exists at least one process S_j from which S_i will not receive a message with timestamp ts_j such that req_ts$_i^x$ $< ts_j$. Thus, S_j's request $\langle j, \text{req\_ts}_j\rangle$ is ahead of S_i's request $\langle i, \text{req\_ts}_i^x\rangle$. However, by the same discussion as in the proof of Theorem 4, S_j will eventually send a timestamp which is greater than req_ts$_i^x$, and S_state$_i$ will become LOCKED in finite time. □

5 Performance Evaluation

Traditionally, the performance of non SS DMX algorithms is compared on the basis of synchronization delay and a number of messages required for a CS entry. In both the link-locking and node-locking algorithms, the synchronization delay is one message, which is the same as those of both Ricart and Agrawala's and Lamport's non SS DMX algorithms. If no timeout occurs, both algorithms require $3*(n-1)$ messages for each CS entry: $(n-1)$ messages for a request, $(n-1)$ messages for replies, and $(n-1)$ messages for an exit. This number is the same as that of Lamport's algorithm.

A timeout occurs when a node does not send a message to another node for some predefined time period (let τ denote the timeout period). Timeouts are necessary for the system to recover from transient failures. Since each timeout yields one outgoing message, it is desirable to reduce the number of timeouts. This may be done by increasing the value of τ. However, it will also increase recovery time from a transient failure.

We conducted performance evaluation on our implementation of the algorithms by using a simulation feature of the underlying development system [9]. In this section, we show the result based on the following execution parameters[7]:

1. the number of nodes (denoted by n): 10 (they are fully connected by a network);

[7] Parameters (2)-(4) use exponential distribution with the average time units given in parentheses

Table 1. Average numbers of messages

	τ value		
	10	100	200
Link-Locking	121.3	31.1	27.9
Node-Locking	111.0	33.0	27.9

2. network transmission latency : (2 time units);
3. execution time inside a CS [denoted by *MX_TIME*]: (10 time units); and
4. execution time outside a CS: (50 time units).

We run the programs with three τ values: (1) *MX_TIME*$(= 10)$, (2) *MX_TIME*$* n(= 100)$, and (3) *MX_TIME*$* n * 2(= 200)$. Table 1 shows the numbers of messages exchanged for each CS entry that are averaged over 1000 time units. Even though the τ value is chosen to be large (*MX_TIME*$* n * 2$), the number of messages for a CS entry is still slightly greater than $27(= 3 * (n - 1))$. This is because (1) a few timeouts still occurred; and (2) some extra messages were exchanged at the beginning of computation for the system to stabilize.

6 Conclusion

In this paper, we have presented two lock based self-stabilizing distributed mutual exclusion algorithms: one is a link-locking algorithm and the other is a node-locking algorithm. Self-stabilization provides a complementary approach to some existing error recovery methods that were developed for non self-stabilizing mutual exclusion algorithms. Unlike most, if not all, of the existing self-stabilizing mutual exclusion algorithms, our algorithms assume execution environments that are close to those provided by common distributed systems. Furthermore, they provide better synchronization delays than token circulation self-stabilizing mutual exclusion algorithms. We have implemented the algorithms and tested them with various initial configurations.

References

[1] Andrews, G.R. *Concurrent Programming, Principles and Practice*. Benjamin/Cummings Publishing Co., 1991.

[2] Brown, G.M., Gouda, M.G., and Wu, C. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38(6):845–852, 1989.

[3] Carvalho, S.F. and Roucairol, G. On mutual exclusion in computer networks. *Communications of the ACM*, 26(2):146–147, 1983.

[4] Dijkstra, E.W. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[5] Dolev, S., Israeli, A., and Moran, S. Self stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[6] Gouda, M.G. and Multari, M.J. Stabilizing communication protocols. *IEEE Tran. on Computers*, 40(4):448–458, 1991.

[7] Lamport, L. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–564, 1978.

[8] Maekawa, M. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985.

[9] Mizuno, M. *A Distributed/Parallel Algorithm Development System (software)*. Kansas State University, 1994.

[10] Raynal, M. A simple taxonomy for distributed mutual exclusion algorithms. *ACM Operating Systems Review*, 25(2):47–50, 1991.

[11] Ricart, G. and Agrawala, A.K. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[12] Sanders, B. The information structure of distributed mutual exclusion algorithms. *ACM Transactions on Computer Systems*, 5(3):284–299, 1987.

[13] Schneider, M. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[14] Singhal, M. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18:94–101, 1993.