A Quorum-Based Self-Stabilizing Distributed Mutual Exclusion Algorithm

Mikhail Nesterenko
Department of Mathematics and Computer Scienece
Kent State University
Kent, OH 44240
mikhail@mcs.kent.edu

Masaaki Mizuno
Deptartment of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506
masaaki@cis.ksu.edu

September 16, 2001

Abstract

In this paper we present a self-stabilizing quorum-based distributed mutual exclusion algorithm. Our algorithm is designed for an asynchronous message-passing model. The algorithm scales well since it has constant synchronization delay and its message complexity is proportional to the square root of the number of processes in the system. The algorithm tolerates message loss. The algorithm places few assumptions on timeouts needed for its implementation. All this allows for a ready implementation of the algorithm on practical distributed architectures.

1 Introduction

The problem of mutual exclusion (MX) was first discussed by Dijkstra [9]. A *distributed mutual exclusion* algorithm is designed to work in a system where processes do not share memory or a clock. Such algorithms have been studied extensively [21, 23, 25]. The two main performance metrics MX algorithms designers try to optimize are message complexity and synchronization delay [25]. *Message complexity* is the number of messages needed for one critical section (CS) access. *Synchronization delay* is the number of causally related message exchanges required for one CS entry. Message complexity measures the overhead introduced by an MX algorithm and synchronization delay measures the algorithm's efficiency.

There are three well-known algorithms that exhibit constant (i.e. independent of the number of processes in the system) synchronization delay: Lamport's [16], Ricart-Agrawala's [22], and Maekawa's [18]. All three algorithms use Lamport's logical clock [16]. In the first two algorithms each process communicates with all other processes. Therefore, the message complexity of these algorithms is proportional to the number of processes in the system.

In Maekawa's algorithm a process is assigned a subset of processes called a *quorum*. Maekawa [18] presented a technique of generating quorums such that their size is proportional to the square root of the number of processes in the system. A process communicates with its quorum only. Thus, the message complexity of Maekawa's algorithm is also proportional to the square root of the number of processes in the system. Since a process in a quorum-based algorithm does not communicate with all other processes directly, a deadlock is possible. Maekawa's algorithm incorporates deadlock avoidance measures. Sanders has reported that Maekawa's algorithm is still prone to deadlock [23]. Further corrections to Maekawa's algorithm are discussed in [8, 24]. The research in quorum-based MX algorithms has continued. Examples of recent developments in the area can be found in [1, 3].

The notion of self-stabilization in the context of distributed systems was introduced by Dijkstra [10]. A system is *self-stabilizing* (SS) with respect to a set of *legitimate states* if, starting from an arbitrary initial state, the system is guaranteed to arrive at a legitimate state and remain in the legitimate states thereafter. Thus, an SS system does not need to be initialized and it is able to recover from transient faults.

A significant number of SS algorithms studied in the literature assumes a serial (C-daemon) execution model where a process can read the state of its neighbors and update its own state in one atomic step. This model is easy to reason about due to high-atomicity of actions: relatively few interleavings of these actions have to be considered. However, implementing actions of high-atomicity in real distributed architectures is not straightforward. Conversely, the algorithms designed for a message-passing model can be easily implemented since this model assumes low-atomicity of program actions [17]. Katz and Perry [15] presented an approach of adding stabilization to an arbitrary message-passing algorithm by repeatedly taking a snapshot of global state and resetting the system if an illegitimate state is detected. However, the overhead incurred by this transformation motivates the design of efficient stabilizing versions of individual algorithms. Several self-stabilizing MX algorithms are described in the literature [6, 7, 11]. They assume a shared-memory model and have synchronization delay proportional to the size of the system. Mizuno et al [19] and Nesterenko [20] presented SS algorithms for message-passing model based on Lamport's and Ricart-Agrawala's algorithms.

In this paper we present a self-stabilizing MX algorithm based on Maekawa's algorithm. We call our algorithm $\mathcal{SSM}$. It uses an asynchronous message-passing model. In legitimate states, the algorithm has the same message complexity and synchronization delay as Maekawa's. $\mathcal{SSM}$ has a few other attractive properties. In particular, $\mathcal{SSM}$ is designed to tolerate message loss. That is, losing a message in a communication channel does not take the system into an illegitimate state. This makes $\mathcal{SSM}$ more robust. $\mathcal{SSM}$ streamlines deadlock avoidance of Maekawa's algorithm and (unlike Maekawa's algorithm) the correctness of $\mathcal{SSM}$ does not explicitly depend on using Lamport's logical clocks to generate timestamps.

Our algorithm uses timeouts. Gouda and Multari [12] showed that a self-stabilizing algorithm in a message-passing model has to have timeouts. We place few assumptions on the timeouts used by $\mathcal{SSM}$ – we assume that timeout actions are always enabled and can be executed at any time. The algorithm is proven to work correctly regardless of the timeout length. Thus, the implementers of $\mathcal{SSM}$ can vary the timeout length to accommodate the requirements of a particular system without violating the correctness of the algorithm.

Gouda and Multari [12] also demonstrated that an algorithm in a message-passing system has to have infinite number of states. $\mathcal{SSM}$ uses unbounded timestamps. However, in the concluding section we outline the possibility of using finite-space timestamps if additional assumptions are placed on the execution model.

The rest of the paper is organized as follows. We define the distributed mutual exclusion problem and provide an overview of Maekawa's algorithm in Section 2. We then define the model, syntax and semantics that we use to describe

$\mathcal{SSM}$ in Section 3. We describe $\mathcal{SSM}$ in Section 4 and prove its correctness in Section 5. We discuss of the performance of $\mathcal{SSM}$, implementation details of $\mathcal{SSM}$, and further avenues of research in Section 6.

2 Mutual exclusion problem and Maekawa's algorithm

2.1 Distributed mutual exclusion problem

A distributed system consists of processes that do not share memory or a clock. The processes communicate asynchronously by exchanging messages through communication channels. A process executes a certain portion of code called the *critical section* (CS). A process exits the CS in finite amount of time. After exiting the CS the process may or may not execute the CS again. When a process wants to execute the CS, this process is in *CS contention.* A distributed mutual exclusion algorithm ensures that the following two properties hold: *safety* – no two processes execute the CS at the same time; *liveness* – a process requesting to execute the CS is eventually allowed to do so.

2.2 Maekawa's algorithm

The algorithm presented in this paper is based on non-SS MX algorithm proposed by Maekawa [18]. Maekawa's algorithm works as follows. Each process in the system has one permission for CS entry. A process can grant this permission to other processes. Each process P_i is assigned a quorum K_i of processes in the system. A process can enter the CS when it collects permissions from all processes in its quorum. The quorums are assigned such that every two quorums have at least one process in common. For example, for the system of three processes: P_1, P_2, and P_3, the quorums may be: $K_1 = \{P_1, P_2\}$, $K_2 = \{P_2, P_3\}$, and $K_3 = \{P_1, P_3\}$. Since each process has only one permission to give, mutual exclusion is guaranteed. Timestamps are used to resolve CS contention if multiple processes request the CS at the same time.

Maekawa's algorithm consists of two parts: basic algorithm and deadlock avoidance algorithm. Three types of messages are used in Maekawa's basic algorithm: `request`, `grant`, and `release`. When process P_i wants to enter the CS, it obtains a new timestamp value ts_i. P_i then sends `request` carrying ts_i to every process in K_i. Each process P_j maintains a queue Q_j of requests. When P_j receives `request` from P_i, P_j enqueues the request in Q_j in the increasing order of timestamps. P_j sends `grant` to the process at the head of Q_j.

When P_i receives `grant`s from all processes in K_i, it can enter the CS. When P_i exits the CS, it sends `release` to every process in K_i. Upon the receipt of `release` from P_i, P_j removes P_i's entry from Q_j and sends `grant` to the process whose request is at the head of Q_j.

In Maekawa's algorithm a process may grant its permission out of timestamp order which results in a process with a lower timestamp waiting for the process with a higher timestamp. Such a permission has to be recalled to eliminate deadlocks. For example, let processes P_i, P_j, and P_k be such that $P_j \in K_i$ and $P_j \in K_k$ but $P_i \notin K_k$ and $P_k \notin K_i$. If both P_i and P_k try to enter the CS, they do not contact each other directly. Rather, P_i and P_k request the permission to enter the CS from P_j. Suppose P_i and P_k want to enter the CS and the timestamp of P_i's request is greater than the timestamp of P_k's request (i.e. $ts_k < ts_i$.) If `request` from P_i reaches P_j earlier than `request` from P_k, P_j grants permission to P_i. In this case P_k has to wait for P_i to enter the CS. It is, however, possible that P_k receives a permission from P_m ($P_m \in K_k$) which forces another process P_n ($P_m \in K_n$) wishing to enter the CS to wait. This wait-for relation can form a cycle which leads to a deadlock.

Maekawa's algorithm avoids cycles in the wait-for relation by ensuring that a process with a lower timestamp does not wait for a process with a higher one. Thus, the wait-for relation is kept in the timestamp order. Messages `inquire`, `yield`, and `failed` are used to avoid a deadlock. If a process that granted its permission to enter the CS receives `request` from another process with a smaller timestamp, then the permission is recalled to avoid a deadlock. Suppose P_j granted the permission to P_i and receives a request from P_k such that $ts_k < ts_i$, then P_j sends `inquire` to P_i. If P_i has not already entered the CS when it receives `inquire`, it replies with `yield`. When P_j receives `yield`, it sends `grant` to P_k keeping P_i's request in Q_j. Alternatively, if P_j receives `request` from P_i and P_j already granted its permission to anther process P_k and $ts_k < ts_i$, then P_j sends `failed` to P_i to inform it that the permission cannot be granted at this time.

Maekawa proposed to use quorums generated from finite projective planes. The size of such a quorum is $\lceil\sqrt{N}\rceil$ where N is the number of processes in the system. Thus, the message complexity of the algorithms is $3\sqrt{N}$ if no deadlock avoidance is necessary; otherwise, it is approximately $5\sqrt{N}$. See [1, 18] for further discussion of quorum formation techniques.

3 Model, syntax and semantics

Model. A distributed system consists of a set of *processes* $\mathcal{P}$. The processes have unique identifiers 1 through N that are not subject to change. The processes communicate only by exchanging messages through channels. Each pair of processes P_i and P_j is connected by a channel Ch_{ij} that passes messages from P_i to P_j (i.e. the system is fully connected.) A channel is an infinite capacity FIFO queue of messages.

Syntax. Each process P_i contains a set of variables (called local variables of the process) and a set of actions. We do not declare variables in the program but rather describe them informally. *Constants* are read-only variables. A process has the following syntax.

process P_i

$$*[\quad \langle \mathit{action} \rangle \; [] \; \cdots \; [] \; \langle \mathit{action} \rangle \quad]$$

Each action has the syntax:

$$\langle \mathit{guard} \rangle \longrightarrow \langle \mathit{statement\text{-}sequence} \rangle$$

An action is either *external* or *internal*. An external action abstracts the reaction of the environment. An internal action is used for cooperation between the processes of the system. The guard of an external action is an external-statement. The guard of an internal action is either a binary *predicate* over local variables, a receive-statement, or a timeout-statement. $\langle \mathit{statement\text{-}sequence} \rangle$ consists of assignment statements, if-then-else statements, send-statements and function-calls. No receive or timeout-statements can appear in a statement sequence.
A receive-statement has the syntax:

$$\mathbf{receive}\ \langle \mathit{message\text{-}type} \rangle \mathbf{(} \langle \mathit{variables} \rangle \mathbf{)}\ \mathbf{from}\ \langle \mathit{process} \rangle$$

where $\langle \mathit{message\text{-}type} \rangle$ is the type of the received message; $\langle \mathit{variables} \rangle$ – a list of variables that assume the values carried by the message; and $\langle \mathit{process} \rangle$ – the identifier of the process that sent the message.
A send-statement has the syntax:

$$\mathbf{send}\ \langle \mathit{message\text{-}type} \rangle \mathbf{(} \langle \mathit{expressions} \rangle \mathbf{)}\ \mathbf{to}\ \langle \mathit{process} \rangle$$

where $\langle \mathit{message\text{-}type} \rangle$ is the type of the message to be sent; $\langle \mathit{expressions} \rangle$ – a list of expressions whose values the message is to carry; and $\langle \mathit{process} \rangle$ – the identifier of the destination process.
A timeout-statement has the syntax:

$$\mathbf{timeout(} \langle \mathit{process} \rangle \mathbf{)}$$

A function-call has the syntax:

$$\langle \mathit{function\text{-}name} \rangle \mathbf{(} \langle \mathit{argument\text{-}list} \rangle \mathbf{)}$$

where $\langle \mathit{function\text{-}name} \rangle$ is the name of the function; $\langle \mathit{argument\text{-}list} \rangle$ is a (possibly empty) list of arguments that the function accepts. The arguments are separated by commas. If the function has a return value, it appears on the right-hand side of an assignment statement. If the function does not have a return value, it appears as a separate statement.

Receive, send, timeout, and assignment statements can be parameterized over a set of processes. For example, let X be a set of processes $\{P_1, \cdots, P_k\}$. Then:

$$(P_i \in X)\mathbf{receive} \langle \mathit{message\text{-}type} \rangle \mathbf{(} \langle \mathit{variables} \rangle \mathbf{)}\ \mathbf{from}\ P_i \longrightarrow \cdots$$

denotes:

$$\begin{array}{l} \textbf{receive}\langle \textit{message-type}\rangle(\langle \textit{variables}\rangle)\ \textbf{from}\ P_1 \longrightarrow \cdots \\ \qquad\quad [\!] \cdots [\!] \\ \textbf{receive}\langle \textit{message-type}\rangle(\langle \textit{variables}\rangle)\ \textbf{from}\ P_k \longrightarrow \cdots \end{array}$$

Semantics. A *state* of the system is defined by one value for each local variable in each process in the system, and a value (i.e. a sequence of messages) for each channel in the algorithm [12]. An action whose guard is **true** in some state of the system is *enabled* in this state. The *execution* of an action moves the system from one state to another.

A timeout-statement and an external-statement are **true** in every state of the system. Modeling timeouts and environment reaction this way allows greater flexibility in the algorithm implementation since the algorithm works correctly regardless of the length of the timeout or environment influence.

A receive-statement is **true** when a message of the appropriate type is at the head of an incoming channel. When an action containing a receive-statement is executed, the message is removed from the channel and the variables specified in the receive-statement assume the values carried by the message. The execution of a send-statement results in appending the message to the tail of the channel to the destination process. To model lossy channels, we assume that a send may fail. In this case no message is appended to the channel. When a function is called, it carries out its specified task. If a function returns a value, it is assigned to the variable on the left-hand side of the assignment statement.

A *computation* is an infinite fair sequence of states such that for each state s_i, the state s_{i+1} is obtained by executing the command of an action that is enabled at s_i. Since a self-stabilizing algorithm can start from an arbitrary state, the set of computations defined by the algorithm is *suffix-closed.* That is, any suffix of a computation of the algorithm is also a computation of this algorithm. We restrict our attention to only infinite computations because the algorithm we present has internal actions that are enabled in any state of the system. We make the following fairness assumptions about the computations we consider:

- if an internal action is enabled in all but finitely many states of a computation, then this action is executed in this computation infinitely many times;
- if infinitely many **send**s of the same message are attempted on the same channel, infinitely many must succeed.

These fairness assumptions allow us to state the following proposition.

Proposition 1 If, in a certain computation, there is a sequence of processes $P_{i_0}, P_{i_1}, \cdots, P_{i_n}$ and a sequence of messages $m_{i_0}, m_{i_1}, \cdots, m_{i_n}$ such that:

- P_{i_0} has an action enabled in every state of the computation; when this action is executed, P_{i_0} sends the same message m_{i_0} to P_{i_1};

- when a process P_{i_k} in the sequence (except for possibly P_{i_0} and P_{i_n}) receives a message $m_{i_{k-1}}$ from $P_{i_{k-1}}$, the process always sends message m_{i_k} to $P_{i_{k+1}}$;

then P_{i_n} receives infinitely many messages m_{i_n}.

Let $\mathcal{A}$ be an algorithm and R and S be state predicates on the states of $\mathcal{A}$. R is *closed* if every state of the computation of $\mathcal{A}$ that starts in a state conforming to R also conforms to R. R *converges* to S in $\mathcal{A}$ if R is closed in $\mathcal{A}$, S is closed in $\mathcal{A}$, and any computation starting from a state conforming to R contains a state conforming to S. $\mathcal{A}$ stabilizes to R iff **true** converges to R in $\mathcal{A}$.

4 Description of $\mathcal{SSM}$

4.1 Modules, variables, functions, and messages

Modules. The guarded commands of the algorithm are grouped in two modules: requester and arbiter. These modules perform independent functions: *requester* obtains permissions from the processes in the quorum, *arbiter* grants permissions to other processes. Since the functions of the requester and arbiter modules of the same process are separate, they do not share variables. Requester and arbiter are shown in Figures 1 and 2 respectively.

Timestamps and rounds. $\mathcal{SSM}$ uses logical *timestamps* and *round numbers*. We state the assumptions about them here and relegate the discussion of their implementation to Section 6. Timestamp values are drawn from a totally ordered infinite domain. Timestamps are used to order CS access requests of different processes. When a process wants to enter the CS, it obtains a timestamp. This timestamp is a *request timestamp*. The timestamps of different requesters are compared and the requester with the lowest timestamp is granted the permission to enter the CS. Timestamps are *unique*: a request timestamp of one process cannot be equal to a request timestamp of another process even in illegitimate states.

Requests may arrive out of timestamp order. To avoid deadlock, permission to enter the CS granted to a process may need to be recalled. To keep track of recalls $\mathcal{SSM}$ uses round numbers. Round numbers are drawn from a totally ordered infinite domain.

Requester variables and functions. When discussing variables of multiple processes we attach the identifier of the process to the variable as a subscript. For example process P_i has variable v_i.

Requester contains the following variables and constants:

- K – constant, holds the quorum of process P_i;
- L – holds the set of processes that granted P_i the permission to enter the CS;

```
process P_i

*[
(r1)    external ⟶
            if ¬needcs then
                needcs := true
                L := ∅
                ts := newts()
                (P_k ∈ K) send request(ts) to P_k
  []
(r2)    (P_j ∈ K) receive grant(mts) from P_j ⟶
            if needcs ∧ ts = mts then
                L := L ∪ {P_j}
                if L = K then
                    /* execute CS */
                    needcs := false
                    (P_k ∈ K) send release to P_k
  []
(r3)    (P_j ∈ K) receive inquire(mts, mrd) from P_j ⟶
            if needcs ∧ ts = mts then
                L := L \ {P_j}
                send yield(ts, mrd) to P_j
  []
(r4)    (P_j ∈ K) receive failed(mts) from P_j ⟶
            if needcs ∧ ts = mts then
                L := L \ {P_j}
  []
(r5)    (P_j ∈ K) timeout(P_j) ⟶
            if needcs then
                send request(ts) to P_j
            else
                send release to P_j
```

Figure 1: Requester module of process P_i

```
        []
(a1)      (Pj ∈ AK) receive request(mts) from Pj ⟶
              if ¬empty() ∧ firstid() = lockid() ∧ lockts() > mts then
                  round := newrd()
              update(Pj, mts)
              if firstid() = Pj then
                  if lockid() = Pj then
                      send grant(lockts()) to lockid()
                  else
                      send inquire(lockts(), round) to lockid()
              else
                  send failed(mts) to Pj
        []
(a2)      (Pj ∈ AK) receive yield(mts, mrd) from Pj ⟶
              update(Pj, mts)
              if lockid() = Pj ∧ round = mrd then
                  lockfirst()
                  send grant(lockts()) to lockid()
        []
(a3)      (Pj ∈ AK) receive release from Pj ⟶
              if lockid() = Pj then
                  delete(Pj)
                  if ¬empty() then
                      send grant(lockts()) to lockid()
              else
                  delete(Pj)
        ]
```

Figure 2: Arbiter module of process P_i

- *needcs* – indicates whether P_i wants to enter the CS; **true** if P_i is in CS contention and **false** otherwise;
- *ts* – stores the timestamp of P_i's request for CS access;
- mts, mrd – respectively hold the timestamp and round carried by the received message.

The requester uses function $newts()$. This function returns a greater timestamp every time it is invoked. The timestamps returned by $newts()$ can grow infinitely large. That is, given an arbitrary timestamp ts_a (not necessarily of the same process) $newts()$ returns a timestamp greater than ts_a in a finite number of invocations.

Arbiter variables and functions. Arbiter contains the following variables and constants:

- AK – constant, contains the *anti-quorum* set of P_i [4]. That is: $AK \equiv \{P_j | P_i \in K_j\}$;
- $round$ – holds the round number of the permission recall;
- mts, mrd – hold the timestamp and round of the received message.

The arbiter uses function $newrd()$. This function returns a greater round number every time it is invoked. The arbiter of P_i maintains a queue Q of CS requests from processes in the anti-quorum of P_i. Requests from processes not in AK cannot be stored in Q. The requests in Q are stored in timestamp order. In any state one request of the queue is always marked as locked. This is the request of the process which holds the permission granted by P_i. Q is manipulated through the functions described below.

- $delete(P_j)$ – deletes the request of P_j from Q if such request is present in the queue. If P_j's request is marked as locked and there are other requests in the queue, the first such request is marked as locked when P_j's request is deleted.
- $empty()$ – returns **true** if Q is empty, and **false** otherwise.
- $firstid()$ – returns the identifier of the process whose request is at the head of Q. If Q is empty, it returns a value that is not equal to any process identifier in AK.
- $lockfirst()$ – marks the first request in the queue as locked.
- $lockid()$ – returns the identifier of the process whose request is locked. If Q is empty, it returns a value that is not equal to any process identifier in AK.
- $lockts()$ – returns the timestamp of the process whose request is locked. If Q is empty the return value is unspecified.
- $update(P_j, ts_j)$ – puts the request into Q in the timestamp order. If Q was empty before the operation, the request is marked as locked. If there is a request from P_j in Q with a different timestamp, the old request is removed and the new one is added; if the old request was marked as locked, the request at the head of the queue is marked as locked.

Messages. Six types of messages are used in the algorithm. The requester module sends the following messages:

- `request` – informs the arbiter of a process in K that CS entry is requested;
- `release` – announces CS exit;
- `yield` – gives up the permission.

The arbiter module sends the following messages:

- `grant` – gives the permission;
- `inquire` – tries to recall the permission granted out of timestamp order;
- `failed` – informs requester that the permission cannot be granted.

4.2 Operation of $\mathcal{SSM}$

In this subsection we informally discuss the operation of $\mathcal{SSM}$. We discuss the behavior of the algorithm in legitimate states only. We subdivide the legitimate behavior of $\mathcal{SSM}$ into two parts: (1) basic operation – requests are received by processes in timestamp order and no permission recall is necessary; and (2) deadlock avoidance – permission recall is used. We analyze the algorithm's stabilization from an illegitimate state when we prove the algorithm's correctness in Section 5.

Basic. When P_i needs to enter the CS, its requester module tries to obtain the permission to enter the CS from every process in K_i. The requester joins CS contention by executing the external action (refer to $r1$ in Figure 1). This action is always enabled. However, it is not subject to the fairness assumptions and it may never be executed throughout the computation. Thus, the process may or may not request CS entry.

When the requester of P_i joins CS contention ($r1$), it sets $needcs_i$, empties the set of collected permissions L_i, obtains a new timestamp for the request, stores it in ts_i and sends `request` to all processes in K_i. The message carries the timestamp of the request.

When the arbiter of $P_j \in K_i$ receives a request from P_i ($a1$), the arbiter places the request in Q_j. If P_i is at the head of Q_j, the arbiter sends `grant` back to P_i. If there is a request in Q_j with timestamp smaller than P_i's, P_j sends `failed` to P_i.

When the requester of P_i receives `grant` from P_j with timestamp matching ts_i ($r2$), the requester adds P_j to L_i. If the requester obtains permissions from all processes in K_i, the requester executes the CS. After CS execution, the requester sends `release` to all processes in K_i. When requester receives `failed` from P_j it deletes P_j from L_i.

When the arbiter of P_j receives `release` from P_i ($a3$), it deletes P_i's request from Q_j and sends `grant` to the process whose request is now at the head of Q_j.

Deadlock avoidance. Due to the lack of direct communication between each process in $\mathcal{SSM}$ the permission for CS entry may be granted out of timestamp order even in legitimate states (cf. the discussion of the deadlock avoidance of Maekawa's algorithm in Section 2.2.) To avoid a deadlock, such permission is recalled.

Suppose the arbiter of P_j grants the permission to P_i with timestamp ts_i and receives `request` from process P_k with timestamp ts_k such that $ts_k < ts_i$. Process P_j obtains a new round number and sends `inquire` to P_i ($a1$). This message carries the new round number. We illustrate the necessity of round numbers in the appendix. When P_i receives `inquire`, it removes P_j from L_i and sends `yield` back to P_j ($r3$). When P_j receives `yield`, P_j sends `grant` to the process at the head of Q_j ($a2$).

Thus, a process requesting to enter the CS with a smaller timestamp does

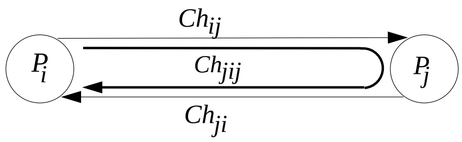

Figure 3: Channel notation

not wait for a process with a greater timestamp and there is no possibility of deadlock. Note that an arbiter may grant the permission to a process and then recall it multiple times before this process enters the CS.

5 Correctness of $\mathcal{SSM}$

In this section we first introduce the notation that simplifies the presentation of the proof. Then we demonstrate that $\mathcal{SSM}$ stabilizes to a certain invariant ($I_{\mathcal{SSM}}$) and show that this invariant guarantees that the algorithm satisfies safety and liveness properties of a solution to the MX problem.

5.1 Notation

Recall that a channel is a queue of messages sent by one process to another. Ch_{jij} denotes a queue of messages composed by appending Ch_{ij} to the tail of Ch_{ji}. This notation is illustrated in Figure 3. We drop the parentheses in functions that maintain the arbiter queue. For example, we use $lockid$ for $lockid()$. Most of the discussion in this section is going to focus on the communication between the arbiter part of one process and the requester part of another. Unless noted otherwise we consider the requester part of P_i and the arbiter part of P_j. We drop the subscript of a variable when it is clear to which process it belongs. For example, L means L_i. We also use ts^i_j to denote the timestamp of P_i's request stored in Q_j. The receive-actions in P_i make non-trivial changes only if the timestamp carried by the message received matches ts_i. When it is clear from the context we ignore messages with timestamps different from ts_i. We use $P_i \in Q_j$ to denote that P_i's request is in Q_j.

5.2 Stabilization

In this section we state a predicate $I_{\mathcal{SSM}}$ that defines the legitimate states of $\mathcal{SSM}$ and show that the algorithm stabilizes to $I_{\mathcal{SSM}}$. To show stabilization we first prove Lemmas 1 and 2. These lemmas define predicates R_1 and R_2 and show that $\mathcal{SSM}$ stabilizes to these predicates. Predicates R_1 and R_2 contain $I_{\mathcal{SSM}}$. They establish the order of round numbers and timestamps of messages in channels between the requester of one process and the arbiter of another in the legitimate states of $\mathcal{SSM}$.

Theorem 1 demonstrates that $\mathcal{SSM}$ stabilizes to $I_{\mathcal{SSM}}$. We group the states of $I_{\mathcal{SSM}}$ into five sets A through E. To prove the closure of $I_{\mathcal{SSM}}$ we show that the execution of each action of $\mathcal{SSM}$ either keeps the system in the same set or moves it to another one of the sets. To prove convergence we show that the system eventually reaches one of the sets. Lemma 3 aids in proving convergence.

Lemma 1 $\mathcal{SSM}$ stabilizes to the following predicate.

`inquire` and `yield` in Ch_{iji} are in non-decreasing order of the round values they carry and the round value of any such message is no greater than $round_j$ (R_1)

Proof: **(sketch)** $round_j$ changes only when $newrd$ is called (see $a1$). Function $newrd$ returns a greater timestamp each time it is invoked. Therefore, the value of $round_j$ can only increase.

When P_j sends `inquire`, it attaches the value of $round_j$ to the message. P_i sends `yield` only when it receives `inquire` ($r3$). The `yield` carries the round value received in the `inquire`. Therefore, after P_i receives every `inquire` that is initially present in Ch_{ji} and P_j receives every `yield` that P_i generates in response to such `inquire`, R_1 holds. □

Lemma 2 $\mathcal{SSM}$ stabilizes to the following predicate.

messages carrying timestamps in Ch_{jij} are in
non-decreasing timestamp order; and
the timestamp of any such message is no greater than ts_i; and
if P_j has P_i's request then (R_2)
  ts_j^i is no greater than ts_i and
  the timestamp of a message in Ch_{ij} is no smaller than ts_j^i and
  the timestamp of a message in Ch_{ji} is no greater than ts_j^i

Proof: **(sketch)** We observe that ts_i can only increase. P_i attaches the value of ts_i to every `yield` and `request` message it sends. Thus, eventually the messages in Ch_{ij} are in non-decreasing order of their timestamps and every such timestamp is no greater than ts_i.

P_j sends a message to P_i only when it has P_i's request in Q_j. P_j stores the timestamp value of P_i's request in ts_j^i. P_j attaches the value of ts_j^i to every `grant` and `inquire` it sends to P_i.

When the messages in Ch_{ij} are in non-decreasing order of timestamps and P_j receives at least one such message, ts_j^i never decreases and ts_j^i is no greater than the timestamp of ts_i or any message in Ch_{ij}. If ts_j^i can only increase, the messages in Ch_{ji} are in non-decreasing order of their timestamps and every such timestamp is no greater than ts_j^i.

To complete the proof we show that either a message carrying a timestamp reaches P_j or P_i's request is removed from Q_j. To that end, we observe that $r5$ is always enabled at P_i. If $needcs$ is set, P_i sends `request`; if $needcs_i$ is cleared – `release`. By the fairness assumption such send eventually succeeds. If P_j receives `request` from P_i it updates ts_j^i. If it receives `release` it removes P_i's request from Q_j. □

We group legitimate states of $\mathcal{SSM}$ with respect to the pair (P_i, P_j) in 5 disjoint sets.
Informally:

A is a set of states where P_i does not wish to enter the CS;
B — a request to enter the CS is sent but not received by P_j;
C — P_j granted P_i the permission to enter the CS and ts^i_j is the smallest in Q;
D — P_j granted P_i the permission but P_j has a request with the timestamp smaller than ts_i;
E — P_i's request is queued by P_j but the permission to enter the CS is not granted to P_i.

The sets are formally defined by the following predicates. Note that only messages with timestamps matching the timestamp of the request are considered.

$$\neg needcs \qquad (A)$$

$$\begin{array}{l} needcs \wedge (P_i \notin Q \vee (P_i \in Q \wedge ts_i \neq ts^i_j)) \wedge (P_j \notin L) \wedge \\ \text{no } \texttt{release} \text{ following } \texttt{request} \text{ in } Ch_{ij} \text{ and} \\ \text{no messages in } Ch_{ji} \end{array} \qquad (B)$$

$$\begin{array}{l} needcs \wedge (P_i \in Q) \wedge (ts_i = ts^i_j) \wedge (lockid = P_i) \wedge (firstid = P_i) \wedge \\ \text{no } \texttt{release} \text{ in } Ch_{ij} \end{array} \qquad (C)$$

$$\begin{array}{l} needcs \wedge (P_i \in Q) \wedge (ts_i = ts^i_j) \wedge (lockid = P_i) \wedge (firstid \neq P_i) \wedge \\ \text{no } \texttt{release} \text{ in } Ch_{ij} \text{ and} \\ \text{no } \texttt{grant} \text{ following } \texttt{inquire}(mrd) \text{ in } Ch_{ji} \text{ where } mrd = round \text{ and} \\ \text{if } \texttt{yield}(mrd),\ mrd = round \text{ in } Ch_{ij} \text{ then} \\ \qquad P_j \notin L \text{ and no } \texttt{grant} \text{ in } Ch_{ji} \end{array} \qquad (D)$$

$$\begin{array}{l} needcs \wedge (P_j \in Q_i) \wedge (ts_i = ts^i_j) \wedge (P_j \notin L) \wedge (lockid \neq P_i) \wedge \\ \text{no } \texttt{release}\ Ch_{ij} \text{ and} \\ \text{no } \texttt{grant} \text{ in } Ch_{ji} \end{array} \qquad (E)$$

We define the invariant $I_{\mathcal{SSM}}$ as follows. For any two processes P_i and P_j such that $P_j \in K_i$ the following holds: $R_1 \wedge R_2 \wedge (A \vee B \vee C \vee D \vee E)$. These sets and transitions between the sets are shown on the diagram in Figure 4. The execution of an action either keeps the system in the same set (loopback transition) or moves it to a different set. For example, when *needcs* is not set and the system conforms to A, the execution of $r1$ moves the system to B. Indeed, if Predicate R_2 holds in A then the timestamps of messages in C_{jij} is no greater than ts_i and if P_j has P_i's request then $ts_i \geq ts^i_j$. Thus, when P_i increases ts_i,

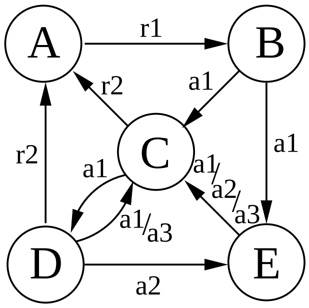

Figure 4: Sets of legitimate states and inter-set transitions

and sends `request`, there can be no `release` following it. Furthermore, there can be no messages carrying the new value of ts_i in C_{ji} and the value of ts^i_j is strictly less than ts_i. Therefore, the system conforms to B. To simplify the diagram, we do not show loopback transitions in Figure 4.

Lemma 3 If $needcs_i$ is set in every state of a computation then the computation has a state conforming to either C or E.

As the liveness theorem later shows the lemma holds vacuously – every process eventually enters CS. However, proving this lemma lets us simplify the stabilization proof.

Proof: The outline of the proof is as follows. We first prove that if $needcs_i$ is set, every process in AK_j executes the CS only finitely many times. Then we show that if P_i has the lowest timestamp among processes in AK_j that wish to enter the CS, then the system eventually reaches C; and if P_i's timestamp is not the lowest the system reaches E.

Since $needcs_i$ is set in every state of the computation, P_i never executes the CS. Thus, the value of ts_i never changes. Let cts_i be value of ts_i during the computation. When $needcs_i$ is set, P_i does not send `release` at all and sends `request` carrying cts_i infinitely often. Thus, there are only finitely many states in the computation where Q_j does not contain cts_i.

If a process $P_k \in AK_j$ executes the CS continuously then it has to alternate executing $r1$ and $r2$. Each time $r1$ is executed, the value held by ts_k increases. Thus, P_k either eventually stops executing the CS or ts_k becomes greater than cts_i. When P_k executes $r1$, it empties L_k. If $P_j \notin L_k$, to enter the CS P_k has to receive `grant`(mts) from P_j where $mts = ts_k$. P_j sends `grant` to P_k only when a request with the corresponding timestamp is in Q_j. Note that if Q_j contains cts_i and $ts_k > cts_i$ then P_j does not send `grant` to P_k. Therefore, every process P_k in AK_j executes the CS only finitely many times.

If a process executes the CS only finitely many times, the timestamp value held at every process eventually stops changing. Let σ_1 be the suffix of the computation where the timestamp of each process remains constant and cts_k – the timestamp of every process $P_k \in AK_j$ in σ_1 . Let W be the set of processes in AK_j such that every process $P_k \in W$ has $needcs_k$ set in σ_1. Correspondingly, $AK_j \setminus W$ is the set such that every $P_k \in (AK_j \setminus W)$ has $needcs_k$ cleared in σ_1.

If P_k is in W, then P_k does not send `release` in σ_1 and sends `request` infinitely often ($r5$). Therefore, there are only finitely many states in σ_1 where Q_j does not contain cts_k and has `release` in Ch_{kj}. Conversely, if $P_k \in (AK_j \setminus W)$ then P_k sends only `release` in σ_1. Thus, there are only finitely many states in σ_1 where Q_j has P_k's request.

Thus, there is a suffix σ_2 of the computation with the following properties. In every state of σ_2 the queue Q_j contains the request of every process in W and none from $AK_j \setminus W$. Furthermore, in every state of σ_2 the channel from a process in W does not contain `release` and the channel from a process in $AK_j \setminus W$ does not contain `request`.

There are two cases two consider: (1) cts_i is the smallest among the timestamps in W, and (2) there is a timestamp in W smaller than cts_i. Let us consider the first case. We now show that there are only finitely many states where $lockid_j \neq P_i$. Note that $lockid_j$ cannot be assigned any other value but P_i during σ_2. Suppose $lockid_j = P_k \neq P_i$ in the first state of σ_2. A process has $r5$ enabled in every state. Since P_i is in W, P_i sends `request` to P_j infinitely often. When P_j receives `request` from P_i, it sends `inquire` to P_k. When P_k receives `inquire` from P_j, it replies with `yield`. By Proposition 1, P_j eventually receives `yield` from P_k. When P_j receives `yield`, it sets $lockid_j = P_i$. This state conforms to C.

Let us now consider the case where cts_i is not the smallest among the timestamps of processes in W. We show that eventually $lockid_j \neq P_i$ and $P_j \notin L_i$. Again, when P_i executes $r5$, `request` is sent to P_j. When P_j receives `request` from P_i, it sets $lockid_j \neq P_i$ and replies with `failed`. By Proposition 1, P_i eventually receives `failed` from P_j. When P_i receives `failed` it deletes P_j from L_i. This state conforms to E. ☐

Theorem 1 (Stabilization) $\mathcal{SSM}$ stabilizes to $I_{\mathcal{SSM}}$.

Proof:
Closure. Note that any legitimate state conforms to R_1 and R_2 and belongs to one of the sets defined by the predicates: $A, \cdots, E$. Note also that these sets are disjoint. To prove closure we demonstrate that if a state is legitimate, then the execution of any action either keeps the system in the same set or moves it to one of the others. We show this for predicates A and C. The argument for the closure of the other predicates is similar.

Let s_1 be the state of the system conforming to either A or C and s_2 – the state of the system after the execution of an arbitrary action. We consider each predicate separately.

A: The only action whose execution violates A is $r1$. Action $r1$ empties

L, assigns new timestamp to ts_i, and sends `request` carrying the value of ts_i. Since R_2 holds, in s_1 the timestamps of messages in Ch_{jij} as well as ts^i_j are smaller than the new ts_i. Therefore s_2 conforms to B.

C: The execution of $r1$ does not violate C because $needcs$ is set. Let us consider $r2$. If P_i receives `grant` from P_j (or any other process) then P_i updates L and possibly executes the CS. The update of L does not violate C and the execution of the CS moves the system into A. The execution of $r3$, $r4$, or $r5$ does not violate C. Let us consider $a1$. If P_j receives `request` from P_i, then P_j sends `grant` to P_i which does not violate C. If P_j receives `request`(mts) from another process P_k there can be two possibilities. If $mts > ts_i$ then P_j sends `failed` to P_k which does not violate C. If $mts < ts_i$ then P_j assigns a new value of round to $round$ and sends `inquire` to P_i. Since R_1 holds in s_1, the round values carried by messages in Ch_{iji} are smaller than the new value of $round$. Therefore s_2 conforms to D. The execution of $a2$ does not violate C. Let us consider $a3$. P_j cannot receive `release` from P_i if s_1 conforms to C. Receiving `release` from other processes does not violate C.

Convergence. We show that if R_1 and R_2 hold then the system eventually conforms to one of the five predicates $A, \cdots, E$. If a computation has a state where $needcs_i$ is cleared then such state conforms to A. If $needcs_i$ is set in every state of the computation, then by Lemma 3 this computation has a state conforming to either C or E. □

5.3 Safety

Theorem 2 (Safety) In any state conforming to $I_{\mathcal{SSM}}$ at most one process has an enabled action that executes critical section.

Proof: Let us consider a process P_i that has the action that executes the CS enabled at some state conforming to $I_{\mathcal{SSM}}$. Let P_j be any process such that $P_j \in K_i$.

Process P_i executes the CS ($r2$) only if $L_i = K_i$ and $needcs_i$ is set. P_i cannot execute the CS when the system is in A because $needcs_i$ is cleared in A. The system cannot be in either B or E because both sets imply $P_j \notin L_i$ and none of the transitions from of B or E execute $r2$. Thus, for P_i to execute the CS the system needs to be in either C or D. Note that both C and D imply that $lockid_j = P_i$. This means that the system cannot be in C or D with respect to some process $P_k \in AK_j$ and P_j. That is, if P_i has the action that executes the CS enabled, none of the other processes in AK_j do. This argument applies to every arbiter in P_i's quorum. The theorem follows. □

5.4 Liveness

Lemma 4 When $I_{\mathcal{SSM}}$ holds, if P_i has $needcs_i$ set then either the system eventually reaches a state where there is P_k such that $needcs_k \wedge (ts_k < ts_i)$ or P_i executes the CS.

Proof: If the system reaches a state where there is P_k wishing to enter the CS with smaller timestamp than ts_i, the lemma holds trivially. We shall prove that if the computation does not contain such a state then P_i enters the CS. We shall consider one process P_j such that $P_j \in K_i$. Since P_i wishes to enter the CS, it sends `request` to P_j. Thus, eventually P_j has P_i's request and $ts_i = ts_j^i$.

If a process $P_k \neq P_i$ such that $P_k \in AK_j$ wishes to enter the CS, it sends `request` to P_j; if not – it sends `release`. The timestamp carried by `request` is greater than ts_i. Thus, the P_k's request is either eventually placed behind P_i's request in Q_j or removed for the rest of the computation. Eventually there will be no `request` or `yield` messages carrying timestamps smaller than ts_i in channels leading to P_j. Therefore, P_i's request moves to the head of Q_i and stays there until P_i executes the CS.

If P_i's request is at the head of Q_j, the system can be in C or E. If the system is in E and $firstid = P_i$, then there is a process $P_k \neq P_j$ such that $lockid = P_k$. P_k and P_j are in D. This process may execute the CS and move to A. In this case it sends `release` to P_j. If P_k never requests CS access again, then P_j eventually receives `release` and P_i and P_j move to C. If P_k requests CS access again, then P_j eventually receives `request` from P_k with a timestamp greater than ts_i. In this case P_i and P_j also move to C.

Processes P_i and P_j remain in C until the CS is executed. When the system is in C, P_i sends `request`. When P_j receives this message, it replies with `grant`. When P_i receives `grant` it adds P_j to L_i. Note that P_j remains in L until P_i executes the CS. Applying similar argument to all processes in K_i we conclude that eventually P_i collects the permissions from all these processes in L_i and enters the CS. □

Theorem 3 (Liveness) When $I_{\mathcal{SSM}}$ holds, if a process wishes to enter the CS then this process eventually executes the CS.

Proof: Let us assume the opposite: there is a computation that starts in a state conforming to $I_{\mathcal{SSM}}$ such that $needcs_i$ is set and P_i never executes the CS. This means that the value of ts_i does not change throughout the computation. Let cts_i be the value ts_i holds.

If a process executes the CS infinite number of times throughout the computation, then eventually the timestamp that this process obtains becomes greater than cts_i. Assume that there is a process P_k that enters the CS only finitely many times and then has $needcs_k$ set for the rest of the computation and $ts_k < cts_i$. There must be a process P_l with the lowest timestamp among such processes. This means that the computation has a suffix where every process wishing to enter the CS has a timestamp greater than ts_l. P_l conforms to the conditions of Lemma 4. This means that it eventually executes the CS and clears $needcs_l$ which contradicts our assumption that such process exists.

Therefore there must be a suffix of the computation where every process wishing to enter the CS has the timestamp greater than cts_i. Applying Lemma 4 again we show that contrary to our initial assumption P_i eventually enters the CS. □

6 Performance evaluation, implementation and further research.

Performance evaluation. We evaluate the algorithm's performance only after it stabilizes. Also, we discount messages generated by timeouts. The performance of our algorithm differs with the *load* – the number of processes in CS contention simultaneously. If the load is low, it takes 3 messages for one CS entry per quorum member (`request`, `grant`, and `release`). If the size of the quorum is proportional to $\sqrt{N}$, the message complexity is proportional to $3\sqrt{N}$. If the load is high, the arbiter sends `failed` to the requester when the arbiter has a request with lower timestamp. Thus the message complexity is proportional to $4\sqrt{N}$. If a significant number of requests arrive out of timestamp order some permissions have to be recalled. It takes two messages per arbiter to revoke a permission (`inquire` and `yield`). If the permission needs to be recalled `failed` is not sent to the process with the smaller timestamp. That is, the worst case scenario is as follows. Process P_i sends `request` to every process in K_i. Process P_j has to recall the permission. It sends `inquire` to the process P_k currently holding the permission. Process P_k replies with `yield`. Process P_j sends `grant` to the P_i, which enters the CS. After exiting the CS, P_i sends `release` to P_j. Therefore, the message complexity with high load and high rate of out-of-order requests is proportional to $5\sqrt{N}$.

Implementation issues. In this paper we assume that timestamps are objects that are totally ordered and that timestamps returned by successive invocations of $newts()$ can grow infinitely large. We also assume that timestamps of each process cannot be equal to timestamps of another process even in illegitimate states. Such timestamps can be implemented by infinite integer counters. If the timestamps of two processes have the same counter value, process identifiers can be used to break a tie. To ensure that timestamps are unique in illegitimate states, the arbiter can discard the messages timestamped with a process identifier that does not match the sender's identifier; similarly the requester can discard the messages timestamped with process identifier different from its own.

When a process does not request the CS for a long period of time, its counter may fall behind the other processes' counters. When such a process requests CS access, it obtains a timestamp that is smaller than timestamps used by other processes. The arrival of a request carrying such timestamp forces permission recall and results in inefficient behavior of $\mathcal{SSM}$. To avoid such situations, Lamport's logical clocks [16] can be used to synchronize the counters.

Since round numbers used by one arbiter are never compared with round numbers of any other arbiter, they do not have to be unique. Neither do they have to be synchronized. Thus, the round numbers can be implemented as just infinite counters.

Another issue we would like to address is the implementation of the timeouts. Weak fairness is the only constraint we place on the execution of the timeouts in $\mathcal{SSM}$. The timeouts are used for stabilization and recovery from message loss.

If a timeout is too long there are fewer overhead messages but it takes longer for the system to resubmit a lost message or to come to a legitimate state after a failure. Conversely, a shorter timeout creates higher overhead and faster stabilization. Thus, the implementer should select the timeout length based on the expected loads and the frequency of failures in the system. We also would like to observe that if a process does not request CS access the timeout is used for stabilization only. Thus, if the system failures are infrequent, the length of the timeout can be set differently depending on whether the process is in CS contention or out of it: the length of the timeout when process is not requesting the CS can be greater.

Further research. In our algorithm we used timestamps that can assume infinitely many values. This reflects the result presented by Gouda and Multari [12] that a self-stabilizing program in a message-passing system has to have an infinite number of states. Infinite variables have to be represented by finite counters in real distributed hardware. Since a self-stabilizing program has to start from an arbitrary initial state, it may start from a state where a finite counter is holding its maximum value. Then further execution of program actions leads to counter overflow which is not considered in the original program with infinite variables and may lead to the loss of stabilization [2].

Howell et al [14] demonstrated that the negative result in [12] is due to the selection of the execution model. Howell et al presented an alternative model that allows finite-space self-stabilizing programs. To make $\mathcal{SSM}$ more applicable in practice we would like to investigate the possibility of adapting $\mathcal{SSM}$ to this model.

$\mathcal{SSM}$ requires the quorums to be hard-coded in every process. Non-stabilizing algorithms that adjust quorums dynamically are described in the literature [5, 13]. Dynamic quorum adjustment makes quorum-based algorithms amenable to changes in network topology. Thus, we believe it would be appropriate to investigate the possibility of extending $\mathcal{SSM}$ to modify quorums as the topology of the network changes.

We conclude the paper by listing another attractive avenue of research. We believe that a relationship between the timeout length and time of stabilization warrants more detailed study. In this respect it is also worth investigating if $\mathcal{SSM}$ is sensitive to the scale of the fault: that is, if the algorithm is able to stabilize faster if the states of only few processes are corrupted.

References

[1] D. Agrawal, Ö. Eğecioğlu, and A. El Abbadi. Billiard quorums on the grid. *Information Processing Letters*, 64(1):9–16, October 1997.

[2] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Bounding the unbounded (distributed computing protocols). In *Proceedings IEEE INFOCOM 94 The Conference on Computer Communications*, pages 776–783, 1994.

[3] R. Baldoni and B. Ciciani. A class of high performance Maekawa-type algorithms for distributed systems under heavy demand. *Distributed Computing*, 8(4):171–180, 1995.

[4] D. Barbara and H. Garcia-Molina. Mutual exclusion in partitioned distributed systems. *Distributed Computing*, 1:119–132, 1986.

[5] B. Bhargava and S. Browne. Adaptable recovery using dynamic quorum assignment. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 231, Brisbane, Australia, August 1990.

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

[7] J.E. Burns. Self-stabilizing rings without demons. Technical Report GIT-ICS-87/36, Georgia Tech, 1987.

[8] R. Chow and T. Johnson. *Distributed Operating Systems and Algorithms.* Addison-Wesley Publishing Co., 1997.

[9] E. Dijkstra. *Cooperating Sequential Processes.* Academic Press, 1968.

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

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

[12] M.G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[13] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2), June 1987.

[14] R.R. Howell, M. Nesterenko, and M. Mizuno. Finite-state self-stabilizing protocols in message passing systems. In *Proceedings of the Fourth Workshop on Self-Stabilizing systems*, pages 62–69, 1999.

[15] S. Katz and K.J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

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

[17] L. Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4:59–68, 1990.

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

[19] M. Mizuno, M. Nesterenko, and H. Kakugawa. Lock-based self-stabilizing distributed mutual exclusion algorithms. In *Proceedings of the Sixteenth International Conference on Distributed Computing Systems*, pages 708–716, 1996.

[20] M. Nesterenko. *Designing Self-Stabilizing Algorithms for Practical Distributed Systems.* PhD thesis, Dept. of Computing and Information Sciences, Kansas State University, 1998.

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

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

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

[24] B. Sanders. Data refinement of mixed specifications. *Acta Informatica*, 35(2):91–129, 1998.

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

Appendix: why $\mathcal{SSM}$ needs round numbers

The following example demonstrates violation of the safety property in the absence of round numbers. We assume that the algorithm in this example is the same as the one given in Figures 1 and 2, except that the algorithm considered here does not use round numbers.

Let processes P_i, P_j, and P_k request CS entry. The relationship between the timestamps of their requests is as follows: $ts_k < ts_j < ts_i$. Let P_a be in K_i, K_j, and K_k. Consider the following scenario.

1. P_i, P_j, and P_k send `requests` to P_a. The `request` message from P_i reaches P_a.
2. P_a sends `grant` to P_i. The message reaches P_i. However, P_i still needs to obtain permissions from other processes in its quorum to enter the CS.
3. P_a receives `request` from P_j and sends `inquire` to P_i.
4. P_j times out and sends another `request` to P_a. P_a receives this request and sends another `inquire` to P_i. The first `inquire` reaches P_i. P_i replies with `yield`.
5. The second `inquire` reaches P_i. P_i sends another `yield`.
6. P_a receives the first `yield` from P_i and sends `grant` to P_j. P_j executes the CS and sends `release` to P_a.

7. P_a receives `release` from P_j and sends `grant` to P_i.
8. `request` from P_k reaches P_a. P_a sends `inquire` to P_i.
9. `yield` sent by P_i in Step 5 reaches P_a. P_a considers it to be the reply for `inquire` sent in Step 8 and sends `grant` to P_k.
10. `grant` sent in Step 7 reaches P_i, and `grant` sent in Step 9 reaches P_k.

In this state, both P_i and P_k enter the CS simultaneously, assuming that they received `grant`s from the other processes in their quorums.