Self-stabilizing Mutual Exclusion and Group Mutual Exclusion for Population Protocols with Covering

Joffroy Beauquier, Janna Burman

Self-stabilizing Mutual Exclusion and Group Mutual Exclusion for Population Protocols with Covering

Joffroy Beauquier[1⋆] and Janna Burman[2⋆⋆]

[1] LRI, University Paris-Sud 11, France. joffroy.beauquier@lri.fr
[2] MASCOTTE, INRIA, I3S (CNRS/University of Nice Sophia-Antipolis), France.
janna.burman@inria.fr.

Abstract. This paper presents and proves correct two *self-stabilizing* deterministic algorithms solving the *mutual exclusion* and the *group mutual exclusion* problems in the model of *population protocols* with *covering*. In this variant of the population protocol model, a *local fairness* is used and bounded state anonymous mobile agents interact in pairs according to constraints expressed in terms of their *cover times*. The cover time is an indicator of the "time" for an agent to communicate with all the other agents. This indicator is expressed in the number of the pairwise communications (*events*) and is unknown to agents. In the model, we also assume the existence of a particular agent, the *base station*. In contrast with the other agents, it has a memory size proportional to the number of agents. We prove that without this kind of assumption, the mutual exclusion problem has no solution.
The algorithms in the paper use a *phase clock* tool. This is a synchronization tool that was recently proposed in the model we use. For our needs, we extend the functionality of this tool to support also phases with unbounded (but finite) duration. This extension seems to be useful also in the future works.

Keywords: distributed algorithms, mobile agent networks, population protocols, cover times, self-stabilization, synchronization, (group) mutual exclusion

1 Introduction

Population protocols is an elegant communication model [2] specially designed for large cheap sensor networks with resource-limited mobile agents. In the original model, each agent is represented by a finite state machine. Agents are anonymous and move in an asynchronous way. When two agents come into range of each other (meet), they can exchange information (communicate). It is important to note that in this model, a type of *global fairness* condition (in the sense of, e.g., [3, 17]) is imposed on the scheduler. According to this condition, a configuration that can be reached infinitely often during the execution is reached infinitely often. This is in contrast to a weaker type of condition called *local fairness* which is generally assumed in the theoretical literature on distributed computing. Informally, with local fairness, in an infinite fair execution, each

⋆ The work of this author was partially supported by grants from Grand Large project, INRIA Saclay.
⋆⋆ The work of this author was supported by Chateaubriand grant from the French Government.

agent satisfying certain conditions is given a turn infinitely often. In contrast to local fairness, global fairness brings an effect of randomization of the scheduling and hence has the ability to circumvent many impossibility results. For example, self-stabilizing algorithms which are proved correct in [3] will fail if local fairness is assumed (refer to [3, 17] for examples and a more detailed discussion on fairness).

In the considered model of population protocols with covering [8], a type of local fairness is used. According to this fairness, there is an indicator called *cover time* associated to each agent. A cover time of an agent x, $\mathbf{cv}_x$, is the minimum (unknown to agents) number of global pairwise interactions (or events) that should happen in the system for being certain that x has met each other agent. A scheduler schedules the next event according to the cover times of agents. Even though this type of the local fairness condition seems relatively strong, many problems stay impossible to solve in such a model (see, e.g., [6,9]). In this work, we prove a similar impossibility result stating that in the model with only bounded state agents, mutual exclusion has no solution (Sec. 6). This result can be easily extended to the group mutual exclusion problem as well. For circumventing the impossibility result, as in [6, 9, 10], we introduce a special agent, the base station (BS). We assume that BS has a memory size proportional to the number of the other system agents.

Like in the original model of population protocols [2], we assume that the number of agents in a system is unknown, that the agents, except BS, are anonymous (no identifiers, uniform code) and have only a few bits of memory (independent of the number of agents). As in [2], we assume a complete communication topology, where each agent has the ability to communicate sometime with any other agent.

We note that the fairness condition expressed in terms of the bounded cover times is supported by recent experimental and analytical studies. The assumption that an agent communicates with all other agents periodically, within a bounded period of real time, has been experimentally justified for some types of mobility such as the human or animal mobility within a bounded area or mobility with "home coming tendency" (the tendency to return periodically to some specific places, e.g., agents' homes). In these cases, a statistical analysis of experimental data sets confirms this assumption. These data sets concern students in a campus [1,24], participants in a network conference [25], visitors at Disney World and more. All exhibit the fact that the *Inter Contact Time* (ICT - the time period between two successive contacts of the same two mobile agents) follows a so called truncated Pareto distribution [11,20,23]. In particular, this involves that ICT is practically bounded. Thus, ICT is also bounded when measured in the number of events. In our model, the cover time of an agent can be expressed in terms of ICTs. Hence, a cover time can be indeed bounded in practice.

The algorithms presented in this paper are self-stabilizing [14]. Such algorithms have the important property of operating correctly (except for some finite period), regardless of their initial configuration. In practice, self-stabilizing algorithms adjust themselves automatically to any change or corruption of the network components (excluding the algorithm's code). Those changes are assumed to cease for some sufficiently long time. Self-stabilization is considered here for two reasons. First, mobile agents are generally fragile, subject to failures and hard to initialize. Second, systems of mobile agents are by essence dynamic, some agents may leave the system while new ones

can be introduced. Self-stabilization is a well adapted framework for dealing with such situations.

In this paper, for the model of population protocols with covering, we present self-stabilizing deterministic solutions for two basic and classical problems in distributed computing, mutual exclusion [21, 26] (Sec. 4) and group mutual exclusion - GME [18, 22] (Sec. 5). Self-stabilizing mutual exclusion in population protocols has been addressed in [3] and in [12]. In [3], a solution is given for Dijkstra-style token circulation in oriented ring communication topology and assumes global fairness (it does not work in the considered case of local fairness). In [12], a deterministic solution assuming local fairness is given. However, it uses strong schemes of "mobile agents" and of oracles. In [6], a simple self-stabilizing algorithm for GME is proposed for the same model as here. However, in this solution, agents can spend only some bounded predefined period in the *critical session* (CS). This is in contrast with the classical definition of GME, according to which they are allowed to spend an unbounded but finite amount of time in CS. Refer to Sec. 2.3 for the specifications we consider for mutual exclusion and for GME.

We design our algorithms in combination with the phase clock tool proposed in [6]. This synchronization tool provides an easy way to organize protocol execution into phases. However, the tool in [6] allows to define and run only phases with bounded duration. In order to be able to grant to an agent an unbounded (but finite) amount of time in CS, we extend the phase clock with the mechanism allowing it to manage also the unbounded duration phases (Sec. 3).[3]

The technique we use for stabilizing the algorithms is similar to the one used in [5, 15]. The variables of the algorithm are regularly reset to predetermined values and an instance of a protocol is repeatedly executed. Note that the automatic self-stabilizing transformer of [9], which uses the same technique, cannot be useful here, because it applies only to static problems.

The problems are not as simple as they appear at first sight, because of the self-stabilization requirement, unknown population size, asynchrony and the anonymity of agents, in particular, when fairness of access to a resource is concerned.

2 Model and Problem Specifications

Basically, the model is as in [2], with the addition of the cover times (Sec. 2.2) and BS.

2.1 Transition System

Let A be the set of all the agents in system $\mathcal{S}$, where the (population) size of $\mathcal{S}$ is $|A| = \mathbf{n}$, and $\mathbf{n}$ is unknown to agents. Among the agents, there may be a distinguishable one, the *base station* (BS), which is said *non-mobile*[4] (and which can have an unbounded memory in contrast with the other agents). All the other agents are bounded state, anonymous (have uniform codes and no identifiers) and are referred as *mobile*.

[3] Note however that this extension does not imply an automatic or easy extension of the GME solution in [6], mainly due to the requirements of CS access fairness in GME.

[4] If BS is actually mobile, it will not change the analysis in this paper.

Population protocols can be modeled as transition systems. We adopt the common definitions of the following: *state* of an agent (a vector of the values of its variables), *configuration* (a vector of states of all the agents), *transitions* (possible atomic steps of agents and their associated state changes), *execution* (a possibly infinite sequence of configurations in which each element follows its predecessor by a transition). For the formal definitions, refer, e.g., to [26]. We refine and add some terms below.

A *local transition system* of an agent x is defined by a set of states and a set of transitions between states.

An event (x, y) is a pairwise communication (meeting) of two agents x and y. During an event, a transition of the form $(\pi_x, \pi_y) \rightarrow (\pi'_x, \pi'_y)$ is executed, where π_x and π_y are the states of x and y before the event, and π'_x and π'_y are the states of x and y after the event (the states after the event may not change). We assume symmetrical communication during (x, y), and (x, y) is an unordered pair. However, the results in the paper can be adapted to asymmetrical communications, in which the state of only x (or y) is modified during an event (x, y).

We extend the transitions between states to configurations as follows. Without loss of generality and as in [2], we assume that no two events happen "simultaneously". Then, there is a transition between two configurations C and C', iff there is a transition $(\pi_x, \pi_y) \rightarrow (\pi'_x, \pi'_y)$, from C and resulting in C', for some two agents x and y, and such that the states of all the other (than x and y) agents are identical in C and C'. Note that each execution corresponds to a unique sequence of events.

Intuitively, it is convenient to view executions as if a *scheduler* (an adversary) "chooses" which two agents participate in the next event. Formally, a scheduler $\mathcal{D}$ is a predicate on the sequences of events. A *schedule* of $\mathcal{D}$ is a sequence of events that satisfies predicate $\mathcal{D}$. To each schedule s corresponds a unique execution of the system (if it is deterministic, and possibly several executions, if it is not). We say that this (or these) execution(s) is (are) *induced* by the schedule s. Let a *period* or a *segment* of a schedule (or of an execution) s be any consecutive sequence of events.

For some l ($\in \mathbb{N}_0$) and agent x, let l *local events at* x, denoted $[l]^x$, be l consecutive (from x's "point of view") events in which agent x participates. This stands in contrast to l *global events* (or just events) which are l consecutive events in an execution. Note that if $[l]^x$ events occurred, then at least l global events occurred.

As in [26], a specification $\mathcal{P}$ of a problem is a predicate on the executions. We say that a system (algorithm) $\mathcal{S}$ solves a specification $\mathcal{P}$, iff any execution of $\mathcal{S}$ satisfies $\mathcal{P}$.

A transition system (or an algorithm) is said to be *self-stabilizing* for a specification $\mathcal{P}$ iff there exists a subset of the set of configurations, called *legitimate configurations*, such that starting from an arbitrary configuration, any execution reaches a legitimate configuration and any execution starting from a legitimate configuration satisfies $\mathcal{P}$. When an execution reaches a legitimate configuration, we say that a system (algorithm) *stabilizes* for $\mathcal{P}$ or just that *stabilization* has occurred. The maximum number of events until stabilization is the *stabilization time* of the protocol. More formal definitions can be found in [26].

2.2 The Cover Time Property (Covering)

Definition 1. *Given a system $\mathcal{S}$ with $\mathbf{n}$ agents, a vector $\overline{\mathbf{cv}} = (\mathbf{cv}_1, \mathbf{cv}_2, \ldots, \mathbf{cv}_\mathbf{n})$ of positive integers (the* cover times*) and a scheduler $\mathcal{D}$, $\mathcal{D}$ (and any of its schedules) is said to satisfy the* cover time property *for $\overline{\mathbf{cv}}$ (in $\mathcal{S}$), if and only if any segment of $\mathbf{cv}_i$ ($\forall i : i \in \{1 \ldots \mathbf{n}\}$) consecutive events of each schedule of $\mathcal{D}$ contains at least one event of an agent x_i with every other agent.*

- *Any* execution *of $\mathcal{S}$ under such a scheduler is said to* satisfy the cover time property for $\overline{\mathbf{cv}}$.
- *The minimum cover time value in $\overline{\mathbf{cv}}$ is denoted by $\mathbf{cv_{min}}$ and the maximum one by $\mathbf{cv_{max}}$. A* fastest / slowest *agent x has $\mathbf{cv}_x = \mathbf{cv_{min}}$ / $\mathbf{cv}_x = \mathbf{cv_{max}}$.*

Note that agents are not assumed to know cover times. They are (usually) unable to store them as their memory is bounded (note that each cover time depends on $\mathbf{n}$).

During the analysis, we consider all the executions that satisfy the cover time property for a given vector of cover times. However, for some vectors of cover times, there is no schedule satisfying the cover time property (take, for instance, the vector $(4, 6, 11, 11)$). A *vector of cover times $\overline{\mathbf{cv}}$ is acceptable* if and only if there exists at least one schedule satisfying the cover time property for $\overline{\mathbf{cv}}$. In the sequel, we will only consider acceptable vectors of cover times.

Remark 1. Let $\overline{\mathbf{cv}}$ be a vector of cover times in system $\mathcal{S}$. Assume that in $\mathcal{S}$, there exists a schedule s of length $\mathbf{cv_{min}}$ containing at least once every possible event. Then, the infinite schedule s_∞ resulting from the infinite iteration of s satisfies the cover time property of $\mathcal{S}$.

In our algorithms, for the purpose of event counting and for being able to use the phase clock tool from [6], BS should be able to estimate the upper bounds on the values of $\mathbf{cv_{min}}$ and $\mathbf{cv_{max}}$ in the system. The self-stabilizing algorithms that estimate $\mathbf{cv_{min}}$ and $\mathbf{cv_{max}}$ are presented in [9]. They are executed at BS and stabilize in $O([\mathbf{cv_{min}}]^{BS})$ events. We assume that those algorithms provide the upper bounds of $\mathbf{cv_{min}}$ and $\mathbf{cv_{max}}$ in variables $\mathbf{cv}^*_{\mathtt{min}}$ and $\mathbf{cv}^*_{\mathtt{max}}$ respectively.

2.3 Specifications

2.3.1 The Mutual Exclusion Problem We adopt here the presentation of the mutual exclusion problem in [21]. Thus, it is assumed that each mobile agent has a section of code programmed to use some shared resource. This section of code is called a *critical section* (CS). For the mutual exclusion, it is required that at most one mobile agent executes CS at any given time. An execution of CS is preceded by an *entry section* and followed by an *exit section*. An agent executing its entry section is said to ask for entering CS. After entering CS, an agent is assumed to leave it after some finite but unbounded period. Agents have to deliver the requests one at a time. Once a request is delivered, an agent has to access its CS before making another request.

Definition 2 (Specification of Mutual Exclusion).

– ***Safety:*** *In any configuration, no two agents are executing CS.*

– ***Bounded waiting:*** *There exists some bound B such that when an agent asks for entering CS, it can be preceded by at most B other agents, but eventually enters CS.*

The problem of mutual exclusion can be also presented and solved by introducing a token that circulates between the agents and grants the access to CS (see, e.g., [14]). Thus, the mutual exclusion problem is sometimes named a problem of *token circulation*. Our solution to the problem use the circulating token as well.

2.3.2 The Group Mutual Exclusion Problem (GME) GME was introduced by Joung [22]. The problem deals with sharing $\mathbf{r}$ mutually exclusive resources between $\mathbf{n}$ processors (agents, in our case). There has been some discussions about a precise specification of GME. We adopt here the one given in [18], which is the most precise. As in the mutual exclusion problem, each mobile agent has a section of code called a critical section (CS), which is preceded by an entry section and followed by an exit section. After entering CS, an agent is assumed to leave it after a finite but unbounded period. In [18], an agent wishing to enter CS is said to request a *session*. Sessions represent resources. One resource can be used simultaneously by an arbitrary number of mobile agents, but two or more resources cannot.[5] Thus, agents that have requested different sessions cannot be in CS simultaneously, but agents that have requested the same session can. Each agent requests only one session at a time and it cannot request another session while a request it made is still pending. In [18], the remaining part of the code of an agent, which is outside the critical section, the entry section and the exit section, is called a *noncritical section* (NCS).

Definition 3 (Specification of Group Mutual Exclusion).

– ***Mutual exclusion:*** *If two agents are in CS at the same time, then they request the same session.*
– ***Lockout freedom:*** *If an agent enters its entry section, then it eventually enters CS.*
– ***Bounded exit:*** *If an agent enters the exit section then it enters NCS within a bounded number of its own steps.*
– ***Concurrent entering:*** *If an agent x requests a session and no other agent requests a different session, then x enters CS within a bounded number of its own steps.*

3 Self-stabilizing Phase Clock Tool

The algorithms we propose use the self-stabilizing bounded phase clock designed in [6]. The specification it satisfies is given below and follows a conventional definition of a phase clock (see, e.g., [4, 13]) with some adaptation (as the *frequency of progress* condition) to better suit the model of population protocols with covering.

Definition 4 (Specification of a Phase Clock [6]).
A bounded phase clock (the clock size $\mathbf{K}$ is definable) provides each agent x with a clock/phase *value in the variable* $\texttt{clock}_x$ *subject to the following conditions.*

[5] In the following, session is also referred as the *period* during which agents can share the same resource.

– ***Progress:*** *In any execution, every variable* $\texttt{clock}$ *is updated infinitely often and each time, according to the assignment statement* $\texttt{clock} := (\texttt{clock} + 1) \mod \mathbf{K}$ *only.*
– ***Frequency of progress:*** *In any execution, after every update of* $\texttt{clock}_x$*, the next update cannot happen before* $\beta(\texttt{clock}_x)$ *events, where* β *is a predefined function.*
– ***Asynchronous unison:*** *In any configuration reached by an execution, the clock values of any two agents differ by no more than* 1 *(* $\mod \mathbf{K}$*). That is, for any two agents* x *and* y*, the following predicate is true:* $(\texttt{clock}_x = \texttt{clock}_y) \vee$
$(\texttt{clock}_x = (\texttt{clock}_y + 1) \mod \mathbf{K}) \vee (\texttt{clock}_y = (\texttt{clock}_x + 1) \mod \mathbf{K})$

In the phase clock algorithm of [6], agents synchronize their clocks with BS. According to the algorithm, BS is the only agent that increments the clock value, which is then propagated to the other agents. A maximal period (a segment of an execution) during which $\texttt{clock}_{BS} = p$ (for some p) is called *phase* p. A period between two increments of the clock at BS is called a *complete phase* p if after the first increment $\texttt{clock}_{BS} = p$. *Incomplete phases* arise from a bad (faulty) initialization.

Remark 2. A useful property of the phase clock after stabilization is that during a complete phase p, there is a period of at least $\beta(p)$ events, where all the agents have the same clock value p (Lem. 7 [6]).
In addition, by Lem. 1 [6], every phase p is bounded, if $\beta(p)$ is bounded. Every complete phase p lasts $[\beta(p) + \min(2 \cdot \mathbf{cv}^*_{\texttt{min}}, \mathbf{cv}^*_{\texttt{max}})]^{BS}$ events.

Extending the Phase Clock of [6]. To use the phase clock of [6], one should define the size of the clock $\mathbf{K}$ (the number of the required phases) and the duration β for every phase. However, the ability to define only a bounded phase duration does not seem to be good enough for our purposes here. For being able to grant an agent a finite but unbounded period in CS, we should be able to define an unbounded phase that could be preempted externally, after some finite period. Then, the *progress* condition (in Def. 4) could be still satisfied. Thus, we slightly extend the phase clock of [6] to provide this functionality. We change appropriately the *frequency of progress* condition in the specification.

Definition 5 (New *frequency of progress* condition).
Frequency of progress: *In any execution, after every update of* $\texttt{clock}_x$*, the next update cannot happen before* $\beta(\texttt{clock}_x)$ *events, if* $\beta(\texttt{clock}_x) \neq \infty$*, where* β *is a predefined function. Otherwise (*$\beta(\texttt{clock}_x) = \infty$*), the next update happens in a finite period.*

To satisfy the new specification, we adopt the same implementation of the phase clock as in [6] and we add only two things. First, we allow to define an unbounded duration for a phase p, by defining $\beta(p) = \infty$. Second, we design a user interface procedure called $\texttt{switch}()$ which, when invoked, causes the phase clock to switch to the next phase in a finite period. The $\texttt{switch}()$ procedure is implemented as follows: $\texttt{switch}() = \{\texttt{event\_ctr} \leftarrow \min(2 \cdot \mathbf{cv}^*_{\texttt{min}}, \mathbf{cv}^*_{\texttt{max}})\}$, where $\texttt{event\_ctr}$ is the counter variable of the phase clock in [6] which is managed by BS and holds the number of the events that are still to count before the next switch (increment) of $\texttt{clock}_{BS}$. This counter is decremented on each event of BS with some agent. Thus, whenever a user

calls switch(), in $[\min(2 \cdot \mathbf{cv}^*_{\mathtt{min}}, \mathbf{cv}^*_{\mathtt{max}})]^{BS}$ events, the phase clock switches to the next phase. Note that in switch(), event_ctr is not set to 0. This is to ensure that each phase lasts at least $\min(2 \cdot \mathbf{cv}^*_{\mathtt{min}}, \mathbf{cv}^*_{\mathtt{max}})$ events, as required in [6].

Remark 3. It is easy to verify that the correctness proofs of the new phase clock are similar to those in [6], with the only difference that now a phase may be defined to have some unbounded but finite duration. To guarantee the stabilization of the phase clock, the user should ensure that if for some p, $\beta(p) = \infty$, then phase p is indeed finite (e.g., ensure that switch() is called after a finite period).

The Extended Phase Clock in a Composition. Our algorithms for mutual exclusion (Sec. 4) and for group mutual exclusion (Sec. 5) use the extended phase clock tool as a module. The (group) mutual exclusion module reads only the clock variables (clock) of the phase clock module and invokes the switch() procedure when appropriate. The modules are composed in a *strict interleaving*. That means that during each event, the codes of both modules are executed one after the other. This composition is not *fair* in the sense of [19] or [26], so that general results about *fair composition* cannot be applied. However, the main result remains true and comes from Remark 3. That is, for proving that the composition of the two modules is self-stabilizing, it suffices to prove the self-stabilization of the (group) mutual exclusion module, assuming that the phase clock is already stabilized. In particular, this implies that one should first prove that every phase is finite.
For the phase clock module used in our algorithms, we define $\mathbf{K} = 3$ (that is, $\mathtt{clock} \in \{0, 1, 2\}$), $\beta(0) = \mathbf{cv}^*_{\mathtt{max}}$, $\beta(1) = \infty$ and $\beta(2) = \mathbf{cv}^*_{\mathtt{min}}$. Thus, to prove stabilization, the first step is to prove that phase 1 is finite (see Lem. 1 and Lem. 7 [7]).

4 A Self-stabilizing Solution to Mutual Exclusion

In this section we present a self-stabilizing algorithm, Alg. 1, solving the mutual exclusion problem in the model of population protocols with covering (for any given acceptable vector of cover times) and with BS. As there is no such algorithm if all agents have a bounded state (Corollary 1), we assume that BS has a memory size proportional to the number of agents. The codes of the mobile agents are identical, but BS has a special code. Our solution uses a phase clock tool as explained in Sec. 3.

First, we describe the algorithm once stabilization has occurred. It operates by infinite iteration of a succession of three phases. A first phase (phase 0; lines 6-8), called a *request phase*, has a phase duration (see Sec. 3) of $\mathbf{cv}^*_{\mathtt{max}}$ events. An agent requesting CS ($\mathtt{state} = \mathbf{request}$), delivers a request to BS (and becomes $\mathbf{registered}$) when it meets BS during phase 0 and its clock is equal to $0(\mod 3)$. During this phase, BS counts the number of requests (in a variable req_ctr). Each request phase is followed by a finite but unbounded phase, which is controlled by BS and called an *access phase*. During an access phase, BS gives the token to the first registered agent visiting it (lines 9-14), waits for the token to be returned (remember that as a basic assumption, an agent uses its CS for a finite time), and then decrements req_ctr (lines 15-20). Then, BS waits for another registered agent and so on, until req_ctr goes down to 0. Then, a

sweeping phase (having a phase duration of $\mathbf{cv}^*_{\mathtt{min}}$ events) resets all the counters and the states of all the agents (to a $\mathbf{neutral}$ state) and a new request phase can begin (lines 34-43).

For the protocol to be self-stabilizing, we use the technique of re-initializing periodically the variables of the algorithm. After the re-initialization in the sweeping phase, the algorithm executes as described above. However, due to a bad initialization of the variables, the execution could stay forever in phase 1 and never reach a sweeping phase, and hence, never reach stabilization. As $\mathtt{req\_ctr} \leq 0$ is the natural condition for ending phase 1 (line 32), we examine the cases in which `req_ctr` could stay strictly positive forever. We identify two such cases. The first one is when the number of the registered agents is lesser than the value of `req_ctr` (lines 20 and 32-33). The second case is when there is no token, neither at BS, nor at any mobile agent (consequently, BS will never be able to decrease `req_ctr` and switch to the sweeping phase; see lines 10,15, 20 and 32-33). We treat separately these two cases. We introduce a variable `no_req_evntctr` for checking that there are still registered agents (or agents in CS) when $\mathtt{req\_ctr} > 0$ (lines 21-25), and a variable `no_token_evntctr` for checking that the system is not in the second case (lines 26-30). These variables are event counters of the local events at BS. Each counter counts till $\mathbf{cv}^*_{\mathtt{max}}$ events to ensure that BS has met all the mobile agents during the last period, but neither the registered ones, nor the ones in CS or the ones in the CS exit section (that is, one of the cases above is satisfied). Then, BS calls `switch()` (line 32-33) to switch to the next (sweeping) phase to reset the algorithm and to stabilize.

4.1 Proving Correctness

First, according to Remark 3, we prove that phase 1 is finite. Recall that phases 0 and 2 are finite (and bounded) by the correctness of the phase clock (see Remark 2).

Lemma 1. *In Alg. 1, every phase 1 is finite.*

Proof: At the beginning of a phase 1 (complete or incomplete), there are two cases:
(A) there is at least one token either at BS or at some agent which is in state $\mathbf{in}$ or $\mathbf{out}$;
(B) there is no such agent holding a token (however, there may be an agent in a different state, holding a token).
In case (A) there are two possibilities:
(a) the number of the registered agents is greater or equal to the value of `req_ctr`;
(b) the number of the registered agents is strictly lesser than the value of `req_ctr`.
In case (a), if BS does not hold a token, then during the next event of BS with an agent in state $\mathbf{out}$ holding a token, $\mathtt{token}_{BS}$ becomes true (line 16) and `req_ctr` is decremented (line 20). From this point, BS can dispatch the token to different registered agents until $\mathtt{req\_ctr} \leq 0$ (lines 9-20). Then, the end of the phase condition, in line 32, triggers.
In case (b), the same happens, but once all the registered agents have visited BS and received the requested CS, `req_ctr` stays strictly positive. However, there are no more registered agents. Then, BS counts $\mathbf{cv}^*_{\mathtt{max}}$ events without seeing a registered agent (lines 21-25) and the condition $\mathtt{no\_req\_evntctr} \geq \mathbf{cv}^*_{\mathtt{max}}$, in line 32, causes the end of the phase.

Algorithm 1 Self-stabilizing Mutual Exclusion

Memory in a mobile agent $x \neq$ BS
$\texttt{token}_x$: *boolean*
$\texttt{state}_x \in \{\textbf{neutral}, \textbf{request}, \textbf{registered}, \textbf{in}, \textbf{out}\}$
Memory in BS
$\texttt{token}_{BS}$: *boolean*
$\texttt{req\_ctr}$: *integer*
$\texttt{no\_req\_evntctr}$: *integer*
$\texttt{no\_token\_evntctr}$: *integer*

1: **when** agent x enters its entry section **do**
2: $\quad \texttt{state}_x \leftarrow \textbf{request}$
3: **when** agent x enters its exit section **do**
4: $\quad \texttt{state}_x \leftarrow \textbf{out}$
5: **when** agent x communicates with BS - event (x, BS)) **do**
6: $\quad$ **if** $(\texttt{clock}_{BS} = \texttt{clock}_x = 0 \mod 3 \wedge \texttt{state}_x = \textbf{request})$ **then** // *Request Phase*
7: $\quad\quad \texttt{state}_x \leftarrow \textbf{registered}$
8: $\quad\quad \texttt{req\_ctr} \leftarrow \texttt{req\_ctr} + 1$
9: $\quad$ **if** $(\texttt{clock}_{BS} = \texttt{clock}_x = 1 \mod 3 \wedge \texttt{req\_ctr} > 0)$ **then** // *Access Phase*
10: $\quad\quad$ **if** $(\texttt{token}_{BS} \wedge \texttt{state}_x = \textbf{registered})$ **then** // *entering CS*
11: $\quad\quad\quad \texttt{token}_{BS} \leftarrow \textbf{false}$
12: $\quad\quad\quad \texttt{token}_x \leftarrow \textbf{true}$
13: $\quad\quad\quad \texttt{state}_x \leftarrow \textbf{in}$
14: $\quad\quad\quad \langle\ x$ enters CS $\rangle$
15: $\quad$ **if** $(\texttt{clock}_{BS} = \texttt{clock}_x = 1 \mod 3 \wedge \texttt{token}_x \wedge \texttt{state}_x = \textbf{out})$ **then** // *exiting CS*
16: $\quad\quad \texttt{token}_{BS} \leftarrow \textbf{true}$
17: $\quad\quad \texttt{token}_x \leftarrow \textbf{false}$
18: $\quad\quad \texttt{state}_x \leftarrow \textbf{neutral}$
19: $\quad\quad$ **if** $(\texttt{req\_ctr} > 0)$ **then**
20: $\quad\quad\quad \texttt{req\_ctr} \leftarrow \texttt{req\_ctr} - 1$
21: $\quad$ **if** $(\texttt{clock}_{BS} = 1 \mod 3 \wedge \texttt{req\_ctr} > 0)$ **then** // *control of requests*
22: $\quad\quad$ **if** $(\texttt{state}_x = \textbf{registered} \vee \texttt{state}_x = \textbf{in})$ **then**
23: $\quad\quad\quad \texttt{no\_req\_evntctr} \leftarrow 0$
24: $\quad\quad$ **else**
25: $\quad\quad\quad \texttt{no\_req\_evntctr} \leftarrow \texttt{no\_req\_evntctr} + 1$
26: $\quad$ **if** $(\texttt{clock}_{BS} = 1 \mod 3)$ **then** // *control of tokens*
27: $\quad\quad$ **if** $(\texttt{token}_{BS} \vee (\texttt{token}_x \wedge (\texttt{state}_x = \textbf{in} \vee \texttt{state}_x = \textbf{out})))$ **then**
28: $\quad\quad\quad \texttt{no\_token\_evntctr} \leftarrow 0$
29: $\quad\quad$ **else**
30: $\quad\quad\quad \texttt{no\_token\_evntctr} \leftarrow \texttt{no\_token\_evntctr} + 1$
31: $\quad$ **if** $(\texttt{clock}_{BS} = 1 \mod 3)$ **then** // *end of access phase*
32: $\quad\quad$ **if** $(\texttt{req\_ctr} \leq 0) \vee (\texttt{no\_req\_evntctr} \geq \textbf{cv}^*_{\texttt{max}}) \vee (\texttt{no\_token\_evntctr} \geq \textbf{cv}^*_{\texttt{max}})$ **then**
33: $\quad\quad\quad \texttt{switch}()$
34: $\quad$ **if** $(\texttt{clock}_{BS} = \texttt{clock}_x = 2 \mod 3)$ **then** // *Sweeping Phase*
35: $\quad\quad \texttt{token}_{BS} \leftarrow \textbf{true}$
36: $\quad\quad \texttt{req\_ctr} \leftarrow 0$
37: $\quad\quad \texttt{no\_req\_evntctr} \leftarrow 0$
38: $\quad\quad \texttt{no\_token\_evntctr} \leftarrow 0$
39: **when** two mobile agents x and y communicate - event (x, y) **do**
40: $\quad$ **if** $(\texttt{clock}_x = \texttt{clock}_y = 2 \mod 3)$ **then** // *Sweeping Phase*
41: $\quad\quad$ **if** $\texttt{state}_x \neq \textbf{request}$ **then**
42: $\quad\quad\quad \texttt{state}_x \leftarrow \textbf{neutral}$
43: $\quad\quad \texttt{token}_x \leftarrow \textbf{false}$

In case (B), BS has no token and cannot receive one, since the only possibility to receive a token is by executing lines 15-20. In this case, these lines cannot be executed, since for any agent x, the condition $\texttt{token}_x \wedge \texttt{state}_x = \mathbf{out}$ (line 15) is false. Hence, during each meeting with BS, line 30 is executed and `no_token_evntctr` is incremented until the end of the phase condition (in line 32) triggers. ▮

Lem. 1 is the key lemma of the proof of correctness, since it ensures that whatever the initial configuration is, a sweeping phase is eventually reached and causes all the variables to reset. Then, from Remark 3, there is no loss of generality in assuming that at the beginning of an execution, the phase clock is stabilized.

Lemma 2. *In any execution of Alg. 1, at the end of a complete phase 2 (the sweeping phase),* $\texttt{token}_{BS} = \mathbf{true}$ *and, for every mobile agent* x, $\texttt{token}_x = \mathbf{false}$.

Proof: By Remark 2, during phase 2, there is a period of at least $\mathbf{cv}^*_{\texttt{min}}$ events where all the agents have the clock value 2. During $\mathbf{cv}^*_{\texttt{min}}$ events happening in the system, at least $\mathbf{cv_{min}}$ events occur. During $\mathbf{cv_{min}}$ events, at least one agent (a fastest one) meets every other agent, including BS. Hence, during phase 2, for every mobile agent, the condition in line 40 and, for BS, the condition in line 34 become $\mathbf{true}$ at least once. Thus, by the end of phase 2, for every mobile agent x, $\texttt{token}_x = \mathbf{false}$ due to the execution of line 43, and $\texttt{token}_{BS} = \mathbf{true}$ due to the execution of line 35. ▮

Lemma 3. *In Alg. 1, after the end of the first complete phase 2, in each configuration during a (complete) phase 1, there is exactly one agent* x *with* $\texttt{token}_x = \mathbf{true}$*, and at the end of the phase 1,* $\texttt{token}_{BS} = \mathbf{true}$.

Proof: By Lem. 2 and the fact that during a request phase the token variables are not updated (lines 6-8), at the end of a request phase (0) the only token is in BS. Right after, in phase 1, the update of the token variables can be done only by executing lines 9-14. There, BS and some mobile agent x in an event exchange the token. Then, the token can be exchanged again, only between the same agents (x returns the token to BS) in lines 15-20. Later in phase 1, the token can move according to the same unique scenario only (with a mobile agent $y \neq x$), because the token variables are updated in lines 9-14 and 15-20 only. Hence, during phase 1 (that comes after the first complete phase 2), there is only one token in the system.
After the sequence of the complete phases 2 and then, 0 (at the beginning of the phase 1), `req_ctr` equals the number of the registered agents. Hence, the scenario described in phase 1 will repeat to the very end of this phase (till `req_ctr` becomes 0). Thus, at the end of phase 1, $\texttt{token}_{BS} = \mathbf{true}$. ▮

Lemma 4 (*safety*). *In Alg. 1, in each configuration after the end of the first complete phase 2, there is exactly one agent* x *with* $\texttt{token}_x = \mathbf{true}$.

Proof: By Lem. 2 and the fact that during a request phase the token variables are not updated (lines 6-8), during the whole phase 0 (following a complete phase 2) the only token is in BS. Then, in phase 1, the lemma is correct by Lem. 3. By the same lemma, at the end of this phase 1, the only token is in BS. Then, during the whole phase 2, the token stays in BS (lines 35 and 43). Then, the whole scenario of the 3 complete phases repeats and hence, the lemma follows. ▮

Lemma 5 (*bounded waiting*). *Assume that in Alg. 1, following a complete phase 2, a mobile agent x asks to enter CS (by line 2). Then, x enters CS (in line 14) during the next (or after the next) phase 1.*

Proof: Starting from the event when x asks to enter CS, x meets BS during phase 0. If the closest phase 0 is incomplete and x does not meet BS, then it meets BS during the next complete phase 0. This is because the duration of the complete phase 0 is $\mathbf{cv}^*_{\mathtt{max}}$ events (during which every mobile agent meets BS). During the event of x with BS in phase 0, x becomes registered and `req_ctr` is incremented (lines 6-8). Moreover, there are exactly `req_ctr` registered agents at the end of this phase. During the following phase 1 (this is case (A)-(a) of Lem. 1 proof), there are `req_ctr` entries in CS (in line 14) by the `req_ctr` different agents, including x. In the worst case, x has asked to enter CS during phase 0 in which it was not registered. Even in this case, it has not waited more than $2 \cdot (\mathbf{n} - 1)$ entries of the other agents before entering CS. ∎

Lemmas 1-5 yield the following theorem.

Theorem 1. *Alg. 1 (composed with the extended phase clock from Sec. 3) is a self-stabilizing solution to the mutual exclusion problem.*

Proof: Define the legitimate configurations as those reached in an execution after a complete phase 2. Lem. 1 involves that starting from an arbitrary configuration, any execution reaches a legitimate configuration. Lem. 2-5 involve that any execution, starting from a legitimate configuration, satisfies the mutual exclusion specification - Def. 2. ∎

5 From Mutual Exclusion to Group Mutual Exclusion

Our solution to group mutual exclusion is given in Alg. 2 and follows a similar scheme as the solution for mutual exclusion (Alg. 1). It also iterates three phases: sweeping phase (phase 2 of the phase clock), request (phase 0) and access phase (phase 1). The difference here is that each such iteration represents a session and has a session number $j \in \{1, 2, \ldots, \mathbf{r}\}$ during the execution. BS executes sessions successively for resources $\{1, 2, \ldots, \mathbf{r}\}$ in a repetitive way. Session j starts with the first event of phase 2, where the session number is incremented to value j (line 33) and it ends with the last event of the next phase 1.

During session j, in the access phase, agents that have requested an access to session j (by going into the state $\mathbf{request}_j$), receive a token from BS and access CS as in Alg. 1. The difference here is that several tokens can be given at the same time. This allows several agents requesting the same resource to enter CS, what (partially) satisfies the *concurrent entering* condition of GME. For each session, the request and the sweeping phases operate similarly as in Alg. 1 with the following differences. In the request phase, BS registers only agents having requested the current session j (those which are in state $\mathbf{request}_j$; see line 6). In the sweeping phase, BS also advances to the next session by incrementing a session number j by 1 ($\mod \mathbf{r}$), in line 33.

Again, as in Alg. 1, to deal with bad initialization and to achieve self-stabilization, every execution should reach a complete sweeping phase. This is ensured by exactly the same mechanism as in Alg. 1 (lines 19-31). The *mutual exclusion* condition of

GME is satisfied because two successive access phases are totally disjoint in time (by the correctness of the phase clock). All the other conditions of GME are mainly or partially satisfied by ensuring that each phase in the algorithm is finite. The proof of the algorithm follows almost the same lines as the proof of Alg. 1. Due to the lack of space, it is provided in [7].

6 Impossibility Result

We prove that if local transition systems of all agents are bounded in memory, the mutual exclusion problem has no self-stabilizing solution (which applies to an infinite family of systems) in the model of population protocols with covering. This impossibility result can be easily extended to GME. It justifies the strong assumption of the existence of BS in the paper.

Similar impossibility results are already known in the classical distributed models and also in the population protocol models without cover times (see [3, 12]). However, since the cover time property of the scheduler has similarities with *partial synchrony* [16] (cover times impose restrictions on scheduling of agents), previous results may appear to be different. Moreover, 'bad execution" demonstrating the impossibility should be proved to satisfy the cover time property. Hence, for completeness, we provide the impossibility proof for the mutual exclusion problem in the considered model. We begin by giving some definitions required for proving and for stating the result.

Since the mobile agents are anonymous and their codes (the set of transitions) are uniform, their local transition systems are identical. Thus, in the model we use, a (global transition) system is entirely characterized by a vector of cover times and the local transition systems of a mobile agent and of BS, if BS exists. Two systems $\mathcal{S}_i$ and $\mathcal{S}_j$, with possibly different number of agents and/or cover time vectors, are said to be *similar*, if and only if they have identical local transition systems.

A *local transition system* of an agent x is said *bounded* if and only if the number of the states (and the transitions) of x is upper bounded by some (predefined) integer constant. A *(global transition) system is bounded*, if and only if the local transition systems of a mobile agent and of BS (if BS exist) are bounded.

A *generic solution* to a problem P is a relation that associates to any positive integer $\mathbf{n}$ and to any acceptable vector of cover times $\overline{\mathbf{cv}}$ (of size $\mathbf{n}$) a system $\mathcal{S}^P_{\mathbf{n},\overline{\mathbf{cv}}}$, with $\mathbf{n}$ agents and a vector of cover times $\overline{\mathbf{cv}}$ (the system scheduler satisfies the cover time property according to $\overline{\mathbf{cv}}$), such that $\mathcal{S}^P_{\mathbf{n},\overline{\mathbf{cv}}}$ solves P. The set of all such systems is called an *image of the generic solution*. A *generic solution* is said *bounded* if and only if every element of its image is bounded. A *self-stabilizing generic solution to the mutual exclusion problem* is a generic solution, such that every system in the image is self-stabilizing for the specification of mutual exclusion in Def. 2.

The first lemma is a simple application of a well known combinatorial property - the pigeonhole principle.

Lemma 6. *Let A be a bounded generic solution to a problem P and let I_A be its image. Then, there exists an infinite subset I_A^{sim} of I_A of similar systems such that for any integer k and for any vector of cover times $\overline{\mathbf{cv}}$, there exists in I_A^{sim} a system of*

Algorithm 2 Self-stabilizing Group Mutual Exclusion

Memory in a mobile agent $x \neq$ BS
$\texttt{token}_x$: *boolean*
$\texttt{state}_x \in \{\textbf{neutral}, \{\textbf{request}_{j'} | j' = 1, 2, \ldots, \textbf{r}\}, \textbf{registered}, \textbf{in}, \textbf{out}\}$
Memory in BS
$\texttt{req\_ctr}$: *integer*
$\texttt{no\_req\_evntctr}$: *integer*
$\texttt{no\_token\_evntctr}$: *integer*
$j \in \{1, 2, \ldots, \textbf{r}\}$

1: **when** agent x enters its entry section for resource j' **do**
2: $\quad \texttt{state}_x \leftarrow \textbf{request}_{j'}$
3: **when** agent x enters its exit section **do**
4: $\quad \texttt{state}_x \leftarrow \textbf{out}$
5: **when** agent x communicates with BS - event (x, BS) **do**
6: $\quad$ **if** $(\texttt{clock}_{BS} = \texttt{clock}_x = 0 \mod 3 \wedge \texttt{state}_x = \textbf{request}_j)$ **then** // *Request Phase*
7: $\quad\quad \texttt{state}_x \leftarrow \textbf{registered}$
8: $\quad\quad \texttt{req\_ctr} \leftarrow \texttt{req\_ctr} + 1$
9: $\quad$ **if** $(\texttt{clock}_{BS} = \texttt{clock}_x = 1 \mod 3 \wedge \texttt{req\_ctr} > 0)$ **then** // *Access Phase*
10: $\quad\quad$ **if** $(\texttt{state}_x = \textbf{registered})$ **then** // *entering CS*
11: $\quad\quad\quad \texttt{token}_x \leftarrow \textbf{true}$
12: $\quad\quad\quad \texttt{state}_x \leftarrow \textbf{in}$
13: $\quad\quad\quad \langle\ x$ accesses resource $j\ \rangle$
14: $\quad$ **if** $(\texttt{clock}_{BS} = \texttt{clock}_x = 1 \mod 3 \wedge \texttt{token}_x \wedge \texttt{state}_x = \textbf{out})$ **then** // *exiting CS*
15: $\quad\quad \texttt{token}_x \leftarrow \textbf{false}$
16: $\quad\quad \texttt{state}_x \leftarrow \textbf{neutral}$
17: $\quad\quad$ **if** $(\texttt{req\_ctr} > 0)$ **then**
18: $\quad\quad\quad \texttt{req\_ctr} \leftarrow \texttt{req\_ctr} - 1$
19: $\quad$ **if** $(\texttt{clock}_{BS} = 1 \mod 3 \wedge \texttt{req\_ctr} > 0)$ **then** // *control of requests*
20: $\quad\quad$ **if** $(\texttt{state}_x = \textbf{registered} \vee \texttt{state}_x = \textbf{in})$ **then**
21: $\quad\quad\quad \texttt{no\_req\_evntctr} \leftarrow 0$
22: $\quad\quad$ **else**
23: $\quad\quad\quad \texttt{no\_req\_evntctr} \leftarrow \texttt{no\_req\_evntctr} + 1$
24: $\quad$ **if** $(\texttt{clock}_{BS} = 1 \mod 3)$ **then** // *control of tokens*
25: $\quad\quad$ **if** $(\texttt{token}_x \wedge (\texttt{state}_x = \textbf{in} \vee \texttt{state}_x = \textbf{out}))$ **then**
26: $\quad\quad\quad \texttt{no\_token\_evntctr} \leftarrow 0$
27: $\quad\quad$ **else**
28: $\quad\quad\quad \texttt{no\_token\_evntctr} \leftarrow \texttt{no\_token\_evntctr} + 1$
29: $\quad$ **if** $(\texttt{clock}_{BS} = 1 \mod 3)$ **then** // *end of access phase*
30: $\quad\quad$ **if** $(\texttt{req\_ctr} \leq 0) \vee (\texttt{no\_req\_evntctr} \geq \textbf{cv}^*_{\texttt{max}}) \vee (\texttt{no\_token\_evntctr} \geq \textbf{cv}^*_{\texttt{max}})$ **then**
31: $\quad\quad\quad \texttt{switch}()$
32: $\quad$ **if** $(\texttt{clock}_{BS} = \texttt{clock}_x = 2 \mod 3)$ **then** // *Sweeping Phase*
33: $\quad\quad j \leftarrow (j + 1) \mod \textbf{r}$ // *advance to the next session*
34: $\quad\quad \texttt{req\_ctr} \leftarrow 0$
35: $\quad\quad \texttt{no\_req\_evntctr} \leftarrow 0$
36: $\quad\quad \texttt{no\_token\_evntctr} \leftarrow 0$
37: **when** two mobile agents x and y communicate - event (x, y) **do**
38: $\quad$ **if** $(\texttt{clock}_x = \texttt{clock}_y = 2 \mod 3)$ **then** // *Sweeping Phase*
39: $\quad\quad$ **if** $\texttt{state}_x \notin \{\textbf{request}_{j'}\}$ **then** // *for any* j'
40: $\quad\quad\quad \texttt{state}_x \leftarrow \textbf{neutral}$
41: $\quad\quad \texttt{token}_x \leftarrow \textbf{false}$

size strictly greater than k with a vector of cover times strictly greater (component by component) than $\overline{\mathbf{cv}}$.

The intuitive justification for the lemma is the following. Since every system in the image of A is bounded, there are only a finite number of possible choices for the local transition systems. Thus, these choices have necessarily to be made infinitely many times, both for $\mathbf{n}$ and for $\overline{\mathbf{cv}}$.

Theorem 2. *There exists no self-stabilizing bounded generic solution to the mutual exclusion problem (in the model of population protocols with covering).*

Proof: Assume by contradiction that there exists a self-stabilizing bounded generic solution to mutual exclusion. Lem. 6 shows that there exists an infinite subset I^{sim} of similar systems self-stabilizing to the mutual exclusion specification (in Def. 2). Without loss of generality, let us assume that the circulating token is used in these systems. Consider some system $\mathcal{S}_1 \in I^{sim}$ that applies to a population of size $\mathbf{n}_1$ (> 1) with a vector of cover times $\overline{\mathbf{cv}}_1$. Let T be the stabilization time of $\mathcal{S}_1$.

By Lem. 6, in I^{sim} , there is another system $\mathcal{S}_2$ self-stabilizing to the mutual exclusion and similar to $\mathcal{S}_1$, with $\mathbf{n}_2$ agents and a cover time vector $\overline{\mathbf{cv}}_2$ such that $\mathbf{n}_2 > \mathbf{n}_1$ and $\mathbf{cv_{min}}(S_2) > T + \frac{\mathbf{n}_2 \cdot (\mathbf{n}_2 - 1)}{2}$ (where $\mathbf{cv_{min}}(S_2)$ is $\mathbf{cv_{min}}$ in $\mathcal{S}_2$).

Consider a legitimate configuration C' of $\mathcal{S}_2$ in which there is (exactly) one token in some mobile agent x, and let C be the projection of C' on $\mathbf{n}_1$ arbitrary agents not holding a token. We consider C as the initial configuration of an execution e in $\mathcal{S}_1$. Note that C is effectively a configuration of $\mathcal{S}_1$, since $\mathcal{S}_1$ and $\mathcal{S}_2$ are similar. Moreover, execution e does exist, because $\overline{\mathbf{cv}}_1$ is acceptable. Since $\mathcal{S}_1$ is assumed to be a self-stabilizing solution, in the execution e from C, after at most T events, e reaches a configuration in which one token has been created. Let us denote by $< token\_creation >$ the step during which this token is created. Then $e = e_1 < token\_creation > e_2$ and the length of e_1 is at most T. We claim that $e_1 < token\_creation >$ is the prefix of a possible (infinite) execution of $\mathcal{S}_2$, from configuration C'. First, all the transitions used in this prefix are transitions of $\mathcal{S}_2$, since $\mathcal{S}_2$ and $\mathcal{S}_1$ are similar. Second, since $\mathbf{cv_{min}}(S_2) > |e1 < token\_creation > | + \frac{\mathbf{n}_2 \cdot (\mathbf{n}_2 - 1)}{2}$, the fact that the $n_2 - n_1$ agents in $\mathcal{S}_2$, but not in $\mathcal{S}_1$, that have no events in $e_1 < token\_creation >$, does not violate the cover time property. Third, $e_1 < token\_creation >$ can be completed to the segment e' of length $\mathbf{cv_{min}}(S_2)$ events by adding all the missing (according to the cover time property) meetings in at least $\frac{\mathbf{n}_2 \cdot (\mathbf{n}_2 - 1)}{2}$ next events. Now, if we repeat indefinitely many times the schedule of e', we can get an infinite execution e'_∞ of $\mathcal{S}_2$ that satisfies the cover time property (see Remark 1). However, in the configuration reached by this execution after the prefix $e_1 < token\_creation >$, there are two tokens - the token that is just created and the token at agent x. That contradicts the assumption that C' is legitimate and that $\mathcal{S}_2$ is a self-stabilizing solution. That proves the theorem. ∎

Corollary 1. *If all agents have a bounded state (bounded by some predefined integer constant, independent of the population size $\mathbf{n}$), there is no generic self-stabilizing solution (and, in particular, no self-stabilizing algorithm) to the mutual exclusion problem.*

References

1. The Dartmouth wireless trace archive - http://crawdad.cs.dartmouth.edu/. Dartmouth College, 2007.
2. D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *DC*, 18(4):235–253, 2006.
3. D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing population protocols. *TAAS*, 3(4), 2008.
4. B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *IEEE TDSC*, 4(3):180–190, 2007.
5. B. Awerbuch and G. Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols (extended abstract). In *FOCS*, pages 258–267, 1991.
6. J. Beauquier and J. Burman. Self-stabilizing synchronization in mobile sensor networks with covering. In *DCOSS*, pages 362–378, 2010.
7. J. Beauquier and J. Burman. Self-stabilizing mutual exclusion and group mutual exclusion for population protocols with covering (extended version). Technical Report inria-00625838, INRIA, 2011. http://hal.inria.fr/inria-00625838/en/.
8. J. Beauquier, J. Burman, J. Clement, and S. Kutten. On utilizing speed in networks of mobile agents. In *PODC*, pages 305–314, 2010.
9. J. Beauquier, J. Burman, and S. Kutten. A self-stabilizing transformer for population protocols with covering. *Theor. Comput. Sci.*, 412(33):4247–4259, 2011.
10. J. Beauquier, J. Clement, S. Messika, L. Rosaz, and B. Rozoy. Self-stabilizing counting in mobile sensor networks with a base station. In *DISC*, pages 63–76, 2007.
11. H. Cai and D. Y. Eun. Crossing over the bounded domain: from exponential to power-law inter-meeting time in MANET. In *MOBICOM*, pages 159–170, 2007.
12. D. Canepa and M. Gradinariu Potop-Butucaru. Self-stabilizing tiny interaction protocols. In *WRAS*, pages 10:1–10:6, 2010.
13. J.-M. Couvreur, N. Francez, and M. G. Gouda. Asynchronous unison (extended abstract). In *ICDCS*, pages 486–493, 1992.
14. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. of the ACM*, 17(11):643–644, Nov. 1974.
15. S. Dolev. *Self-Stabilization*. The MIT Press, 2000.
16. C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
17. M. Fischer and H. Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. In *OPODIS*, pages 395–409, 2006.
18. V. Hadzilacos. A note on group mutual exclusion. In *PODC*, pages 100–106, 2001.
19. T. Herman. *Adaptivity through Distributed Convergence. (Ph.D. Thesis)*. University of Texas at Austin, 1991.
20. S. Hong, I. Rhee, S. Joon Kim, K. Lee, and S. Chong. Routing performance analysis of human-driven delay tolerant networks using the truncated levy walk model. In *Mobility-Models*, pages 25–32, 2008.
21. Peterson J. L and A. Silberschatz. *Operating system concepts*. Addison-Wesley, 1985.
22. Y.-J. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.
23. T. Karagiannis, J. Le Boudec, and M. Vojnovic. Power law and exponential decay of inter contact times between mobile devices. In *MOBICOM*, pages 183–194, 2007.
24. M. McNett and G. M. Voelker. Access and mobility of wireless PDA users. volume 9, pages 40–55, 2005.
25. I. Rhee, M. Shin, S. Hong, K. Lee, and S. Chong. On the levy-walk nature of human mobility. In *INFOCOM*, pages 924–932, 2008.
26. G. Tel. *Introduction to Distributed Algorithms (2nd ed.)*. Cambridge University Press, 2000.