Theoretical Computer Science 412 (2011) 4247–4259

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

A self-stabilizing transformer for population protocols with covering☆

Joffroy Beauquier [a], Janna Burman [b,*], Shay Kutten [c]

[a] LRI, University Paris-Sud 11, UMR 8623, Orsay, F-91405, France

[b] MASCOTTE Project - INRIA, 2004, route des Lucioles, B.P. 93, F-06902 Sophia-Antipolis Cedex, France

[c] Department of Industrial Engineering & Management, Technion, Haifa 32000, Israel

ARTICLE INFO

Keywords:
Population protocols
Cover time
Self-stabilization
Transformer

ABSTRACT

Developing *self-stabilizing* solutions is considered to be more challenging and complicated than developing classical solutions, where a proper initialization of the variables can be assumed. Hence, to ease the task of the developers, some automatic techniques have been proposed to design self-stabilizing algorithms. In this paper, we propose an *automatic transformer* for algorithms in an extended *population protocol model*. Population protocols is a model that was introduced recently for networks with a large number of resource-limited mobile agents. We use a variant of this model. First, we assume agents having characteristics (e.g., moving speed, communication radius) affecting their intercommunication "speed", which is reflected by their *cover times*. Second, we assume the existence of a special agent with an unbounded memory, the *base station*. The automatic transformer takes as an input an algorithm solving a *static problem* (and meeting some additional rather natural requirements) and outputs a self-stabilizing algorithm for the same problem. The transformer is built using a *re-execution approach* (the technique consisting of executing an algorithm repeatedly in order to obtain its self-stabilizing version). We show that in the model we use, a transformer based on such an approach is impossible without the assumption of an unbounded memory agent.

1. Introduction

Mobile sensor networks have been developed recently in applications ranging from environment monitoring to emergency search-and-rescue operations. For instance, ZebraNet [1] is a habitat monitoring application where sensors are attached to zebras and collect biometric information (e.g., heart rate and body temperature) and information about their behavior and migration patterns (via GPS). All the zebras in the population meet each other and ZebraNet's agents (zebras' attached sensors) send data to peer agents. Each agent stores its own sensor data as well as data of other sensors that were in range in the past. They upload data to a base station whenever it is near by. Another example, where mobile sensors move in a more predictable manner, is the EMMA project [2]—a pollution monitoring network of sensors attached to different kinds of public transport vehicles. In EMMA, agents may share information whenever two vehicles meet and later, forward it to a central server at a major bus or train stop.

As mobile sensor networks have their own unique characteristics, attempts have been made for developing specific models. Angluin et al. [3,4] have proposed the population protocol model to describe networks of tiny mobile agents, where the size of the population is large and possibly unknown. Each agent is represented by a finite state machine. Agents are

☆ A preliminary version of this paper "Making Population Protocols Self-stabilizing" appeared in the proceedings of SSS 2009.

* Corresponding author. Tel.: +33 0 4 92 38 79 29; fax: +33 0 4 89 73 24 00.
E-mail addresses: joffroy.beauquier@lri.fr (J. Beauquier), janna.burman@inria.fr (J. Burman), kutten@ie.technion.ac.il (S. Kutten).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.09.016

anonymous and move in an asynchronous way such that each pair of agents is repeatedly close enough to communicate. It has been emphasized that, due to the very nature of potential applications, the available memory in each sensor has to be small. It has been also noted that such sensors are exposed to failures. In this paper, we address the latter problem too.

It was shown in [5] that the set of applications that can be solved in the original population protocol model of [3] is rather limited. Hence, various extensions were suggested for the model of [3] (e.g., [6–9]). In [8], an oracle for eventual leader detection is assumed. However, even with the help of an oracle, it is shown that constructing uniform self-stabilizing leader election in (communication) rings is impossible when local fairness is assumed (a somewhat weaker fairness than in [3]).

In the present paper, we consider a variant of population protocols that enables us to construct a general automatic transformer (Section 3) that transforms a non-self-stabilizing population protocol (given as an input) into its self-stabilizing version. The input protocol has to satisfy the following conditions. First, it should be correct for a non-simultaneous start (which is a natural requirement in asynchronous systems; see Section 2 for details). Second, it should legally terminate with the same vector of correct output values in every execution starting from the same initial configuration, regardless of the schedule "chosen" by the adversary. Similar conditions on the input protocol are required for the self-stabilizing transformers in [10–12] (we adopt the main idea of their method of transformation, as explained in Section 4). These conditions are not very restrictive and define an important and large sub-class of distributed protocols. In Section 5, we show specific protocols for information gathering and leader election, which are examples of protocols from this sub-class.

The model we use here has been introduced in [13,14]. In this model, every two agents meet sometime (a common assumption in population protocols [5,15]) and each agent v is associated with a *cover time*, $\mathbf{cv}_v$. The cover time of an agent is the (*unknown*) number of *events* (pairwise interactions of agents) that occur in the whole system before an agent meets with each other agent with certainty (see definition in Section 2.2). The cover time is a kind of an abstraction of agent's intercommunication/mobility characteristics such as physical speed, movement pattern, frequencies to visit different places, communication range or reliability of a sensor. This model of population protocols with *covering* suits well networks where the movements of agents are somewhat predictable. For example, recent observational studies and statistical analysis (e.g., [16–20]) of human/animal random walks with "homecoming" tendency (the tendency to return to some specific places, e.g., agents' homes) or whose mobility is restricted to being within a finite area, support this model with finite and bounded cover times (a detailed discussion on the issue appears in [21]). Note that the cover times make it possible to construct fast converging deterministic protocols and evaluate their complexities. It is impossible in the original model [3]. See Section 2.2 for details on the cover times.

In addition to the covering, we assume a distinguishable agent, the *base station* (BS), which has an unbounded memory in contrast with the other agents (the size of the memory available to BS grows with the number of agents in the system). Note that this is a natural assumption, because an agent such as BS is present in many sensor network applications, and even in the original population protocol model [3], BS is assumed to start all the agents simultaneously by the transmission of a global signal. In addition, a distinguishable agent is used in other extensions of population protocols (e.g., [7,9]). In Section 4, we justify the usage of BS in our design.

Self-stabilization [22] deals with transient failures and is related to the self-*techniques. The transient failures can corrupt the states of the agents, but not the code of the algorithm they execute. Note that dynamic events, in which the set of agents changes, can be modeled as transient failures. After an arbitrary number of transient failures, a self-stabilizing system recovers by itself, without any external intervention, and in a bounded time. In the model, it is assumed that all the transient failures happen at the beginning of the execution (this deals with the case that the next fault occurs after a rather long time). This is equivalent to assuming that the system is started in an arbitrary configuration. In the sequel, we use the term *classical* for an algorithm assuming initialization, in contrast with a self-stabilizing algorithm.

Self-stabilizing population protocols were studied in [23,8]. There, variants of population protocols (such as assuming complete or ring communication graphs, local fairness, or assuming an oracle for eventual leader detection [8]) are considered and some self-stabilizing protocols for problems such as leader election and token circulation are given. The main difference between the current paper and these previous ones is that we do not deal with specific self-stabilizing solutions, but rather with a general transformer that converts any algorithm that is correct in a failure-free environment into a self-stabilizing one.

Developing self-stabilizing solutions (and proving them) is considered to be more challenging and complicated than developing classical ones. Thus, it is desirable to ease the task of designers by providing automatic transformers that receive a classical algorithm and output its self-stabilizing version. Several such transformers have been designed and studied for other models (see, for instance, [10,24–27]). However, to the best of our knowledge, up to now, none was suggested for the model of population protocols. Note that the transformer presented in [15] deals with crash faults as well as transient faults, however it assumes the bound on the number of these faults is known, contrary to the common assumption made in self-stabilization.

This extended version draws upon results presented in a preliminary version [28]. The extensions are the following. First, several examples motivating the transformation have been added (Section 5). Second, the assumptions on BS are weakened. Here, BS is required to estimate only an *upper bound* on the cover time value of an agent it meets (see Section 2). This is in contrast to the exact estimate requirement in [28]. Finally, we present an impossibility result, justifying the memory assumption made on BS (Section 4).

2. The model

2.1. Transition system

A *system* $\mathcal{S}$ is given as a set A of agents, where $|A| = \mathbf{n}$ and $\mathbf{n}$ is unknown to agents. As in [7], among the agents, there is a distinguishable one, the *base station* (BS), which is (usually) non-mobile[1] and can have unbounded memory in contrast with the other agents. All the other agents are finite-state, anonymous (no identifiers and uniform codes) and referred to in the paper as *mobile*.

Population protocols can be modeled as transition systems. An agent is modeled as a set of *states* and a set of *transitions* between states. The state of an agent is the sequence of the values of its variables. The transitions are of the form $(s_x, s_y) \rightarrow (s'_x, s'_y)$, where s_x and s'_x are two states of agent x, and s_y and s'_y two states of another agent y. A transition can be interpreted as follows: when x meets y (this is denoted by *event* (x, y)) they communicate and exchange values. As a result, x and y set their states to s'_x and s'_y respectively. A *configuration* is a vector of the states of all the agents. We extend the transitions between states to configurations as follows. First, without loss of generality and as in [3,4], we assume that no two events happen "simultaneously". Then, there is a transition between two configurations C and C', iff there is a transition $(s_x, s_y) \rightarrow (s'_x, s'_y)$ for some two agents x and y. The states of all the other $(\neq x, y)$ agents are identical in C and C'.

An *execution* e of $\mathcal{S}$ is a sequence of couples (configuration, transition):
$(C_0, t_0)(C_1, t_1)(C_2, t_2) \ldots$ such that C_{i+1} is obtained from C_i by the transition t_i. An execution is said to be *finite*, iff from some point on, no applicable transition changes the configuration. In this case, that non-changing configuration is said to be *terminal*. When a terminal configuration is reached, we say that the *termination* has occurred. Each execution corresponds to a unique sequence of events. If an execution e is finite, its *length*, denoted by $|e|$ is the minimum number of events until the termination.

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 $\mathcal{D}$. A scheduler $\mathcal{D}$ is said to be *fair*, iff for every agent x, in any infinite schedule of $\mathcal{D}$, x is chosen by $\mathcal{D}$ infinitely often. This *fairness* is somewhat weaker (and more common in the literature) than the one used in the model of [3,4]. Refer to, e.g., [23,8] for a discussion of fairness.

As in [29], a *specification* $\mathcal{P}$ of a problem is a predicate on the executions. We say that an algorithm $\mathcal{A}$ solves specification $\mathcal{P}$, iff any execution of $\mathcal{A}$ satisfies predicate $\mathcal{P}$. The specifications we consider here asks for termination and also for a *property $\mathcal{O}$ of the terminal configuration* of an execution. This property is given as a predicate on a subset of variables called *output variables*. We call *legal* a terminal configuration satisfying property $\mathcal{O}$. In a legal configuration, output variables are said to be *correct*. We call problems specified that way, *static problems*.

Self-stabilization. We adopt the definitions of [29] related to self-stabilization, in particular, those dealing with the notions of *convergence* and *correctness*. Classical algorithms assume that every execution is started from an initial configuration. This is not the case for self-stabilizing algorithms, whose executions can be started from any configuration. Given a static problem $\mathcal{P}$, we say that algorithm $\mathcal{A}$ *stabilizes* for $\mathcal{P}$ if there exists a subset $\mathcal{L}$ of the set of configurations, called *legitimate configurations*, such that: (i) (convergence) every execution from any possible initial configuration reaches a configuration in $\mathcal{L}$. (ii) (correctness) every execution from a configuration in $\mathcal{L}$ only reaches configurations satisfying the property of terminal configuration of $\mathcal{P}$. In other words, an algorithm $\mathcal{A}$ stabilizes for $\mathcal{P}$, iff it converges towards the subset of legitimate configurations and, once converged, never reaches configurations in which the property of terminal configuration of $\mathcal{P}$ is not satisfied. When this happens, we say that *stabilization* has occurred.

Definition 2.1 (*Local and Global Counting*)**.** Let l be any non-negative integer and x an agent in A.

- Let l *locally counted events at* x, denoted by $[l]^x$, be l consecutive (from x's point of view) events in which agent x participates.
- Let l *globally counted events*, or just l *(global) events*, be l consecutive events in an execution.

Note that during $[l]^x$, at least l globally counted events occur.

Definition 2.2 (*Event Complexity*)**.** The *worst case event complexity* (or just the *event complexity*) of a system $\mathcal{S}$ (or of an algorithm $\mathcal{A}$) is the maximum length (counted by the number of global events) of an execution until termination (in case of a system with initialization) or until stabilization (in case of a self-stabilizing system). In the latter case, we also call it the *stabilization complexity*.

2.2. The cover time property (covering)

The cover time, defined below, is an abstraction of agent's mobility characteristics detailed in the introduction. Informally, it indicates the "time" for a mobile agent to communicate successfully with all the other agents. As the systems we consider are asynchronous, implying no real time, the "time" referenced here is the total number of communications (events) during some interval.

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

Given $\mathbf{n}$ 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 $\mathcal{D}$, we say that $\mathcal{D}$ (as well as each of its schedules) satisfies the *cover time property*, if in any $\mathbf{cv}_i$ ($i \in \{1 \ldots \mathbf{n}\}$) consecutive events of each schedule of $\mathcal{D}$, agent i meets every other agent at least once. In addition, $\mathbf{cv}_i$ is the minimum such number of events.[2] Any *execution* of a system under such a scheduler is one that *satisfies the cover time property*.
For two agents x and y, if $\mathbf{cv}_x < \mathbf{cv}_y$, then we say that x is *faster* than y, and y is *slower* than x. The minimum cover time value is denoted by $\mathbf{cv_{min}}$ and the maximum one by $\mathbf{cv_{max}}$. A *fastest/slowest* agent z has $\mathbf{cv}_z = \mathbf{cv_{min}}/\mathbf{cv}_z = \mathbf{cv_{max}}$. We denote by F the set of fastest agents (whose $\mathbf{cv} = \mathbf{cv_{min}}$) and by $|F|$ the size of F.

Remark 1. According to the definition of fairness in Section 2.1, a scheduler satisfying the cover time property is fair.

Remark 2. Note that there are vectors of integers such that there is no possible schedule satisfying the cover time property implied by the vector (e.g, $\overline{\mathbf{cv}} = (4, 6, 10, 10)$). From now on, we assume $\overline{\mathbf{cv}}$s implying at least one possible schedule. For an additional discussion on the validity of the cover time values, refer to [14].

Agents are not assumed to know cover times. Instead, we do assume that when two agents meet, they are able to detect which of them is faster (unless none of them is). That is, every agent x is given a *primitive* **Faster** such that for any agent y it meets, **Faster**(x, y) returns 1, if x is faster than y, and 0 otherwise.

There may be different ways to implement **Faster**. For example, in a real system, each agent x may be given a *category* number $\mathbf{cat}_x$ (a positive integer) such that for each two agents x and y, $\mathbf{cat}_x < \mathbf{cat}_y \Longleftrightarrow$ **Faster**(x, y). In the EMMA project [2], different kinds of public transport vehicles (moving according to different itineraries) can correspond to different categories. In ZebraNet [1], a measured temperature (or pulse) that is far from the normal can imply an ill animal, that is slower. Hence, this animal is assigned a category number that is bigger than the one of healthy animals. In this example of the implementation of **Faster**, we use a rather natural assumption that the number of different categories, $\mathbf{m}$, is much smaller than the size of the population $\mathbf{n}$ ($\mathbf{m} \ll \mathbf{n}$) and agents do not know the value of $\mathbf{m}$. In addition, note that categories are not identifiers, because there can be an arbitrary number of agents in the same category and because agents in the same category are indistinguishable. For simplicity of presentation, we assume the existence of such categories.

For BS, we need the following stronger requirement. We assume that BS, but not a mobile agent, is able to estimate an *upper bound* on the value of the cover time of an agent *it meets*. Recall that BS has an unbounded memory, what may help it in this task. For simplicity of presentation, we assume that BS maintains a table providing an upper bound on the cover time of each category. For each agent j, its estimated cover time is denoted by $\mathbf{cv}^*_j$. The minimum (maximum) $\mathbf{cv}^*_j$ is denoted by $\mathbf{cv}^*_\mathbf{min}$ ($\mathbf{cv}^*_\mathbf{max}$).

2.3. Start of computation

For a non-self-stabilizing algorithm, there are two alternative assumptions for the start of the computation: *simultaneously* and *non-simultaneously*. In the non-simultaneous case, at least one agent starts the computation spontaneously. Then, each time an already started agent x meets a not yet started agent y, agent y starts too. A simultaneous start can be viewed as a special case of the non-simultaneous one, in which the agents respond simultaneously to some global signal, e.g., from BS, to initiate the computation.

The simultaneous start can be difficult (or even impossible) to realize in practice. For instance, in the example with BS above, it implies that BS has a communication power strong enough to broadcast, instantaneously, to each mobile agent, at whatever distance they may be. The non-simultaneous start seems a more realistic assumption. Moreover, a non-simultaneous start is generally more natural for the algorithms designed to run in an asynchronous model (as in our case).

Assume a non-simultaneous start. Then, in no more than $\mathbf{cv_{min}}$ events, a fastest agent starts the algorithm and then, in additional $\mathbf{cv_{min}}$ events it meets each other agent, causing everybody to start. Hence, in $2 \cdot \mathbf{cv_{min}}$ events, all the agents start the computation. However, if $2 \cdot \mathbf{cv_{min}} > \mathbf{cv_{max}}$, this start happens in at most $\mathbf{cv_{max}}$. Hence, all the agents start the computation in no more than $\min(2 \cdot \mathbf{cv_{min}}, \mathbf{cv_{max}})$. This simple observation can be useful for adapting some algorithms assuming a simultaneous start to be correct for a non-simultaneous one, with an increase of at most additional $\min(2 \cdot \mathbf{cv_{min}}, \mathbf{cv_{max}})$ events in the event complexity.

3. The transformer

Let us now present a transformer (compiler) that takes as an input a classical algorithm $\mathcal{A}$ satisfying the conditions below and solving a static problem $\mathcal{P}$. The transformer outputs the self-stabilizing version of $\mathcal{A}$ solving the self-stabilizing version of $\mathcal{P}$. Roughly speaking, the stabilization time of the output algorithm is of the order of the worst case complexity of the input algorithm $\mathcal{A}$ with a multiplicative factor of $\frac{\mathbf{cv_{max}}}{\mathbf{n}-1}$. Refer to Theorem 3.9.

Conditions on the input algorithm. First, the input algorithm $\mathcal{A}$ solves a static problem $\mathcal{P}$ with a non-simultaneous start (see Section 2).[3] Second, $\mathcal{A}$ legally terminates with the same vector of correct output values (finds the same solution) in every execution starting from the same initial configuration (regardless of the schedule "chosen" by the scheduler).

[2] We emphasize that this definition does *not* imply that an agent knows its cover time.

[3] In fact, this condition can be weakened. It is sufficient that $\mathcal{A}$ solves $\mathcal{P}$ for only a subset of all the possible non-simultaneous starts—the subset where BS is the first agent that starts the computation.

We assume that an upper bound on the worst case event complexity of $\mathcal{A}$ is given as a non-decreasing function of the cover times of the agents $\{\mathbf{cv}_1, \mathbf{cv}_2, \ldots, \mathbf{cv_n}\}$. Hence, we also can express the bound as a non-decreasing function of $\mathbf{cv_{min}}$ and $\mathbf{cv_{max}}$, because any cover time value is at most $\mathbf{cv_{max}}$. We denote this upper bound expressed in that way by $\mathtt{WCC}_{\mathcal{A}}$.

3.1. The main idea and structure

Basically, the transformer is a composition of three modules, *TClient* (Section 3.4), *TServerMin* (Section 3.3), and *TServerMax* (Section 3.2). *TServerMax* and *TServerMin* perform independently and provide inputs to *TClient* using a *fair composition* [30,29]. Note that the fair composition requires that every execution contains infinitely many steps of each module in the composition (or, for any module in the composition, contains an infinite suffix in which no step of the module is applicable). To fulfil this requirement here, at each event, an applicable step of each module in the composition is chosen to be executed.

The main module *TClient* is aimed at initializing and executing $\mathcal{A}$ repeatedly. At the end of each such repetition, *TClient* starts outputting a correct output of $\mathcal{A}$ (if no faults occur during the current repetition; otherwise, a correct output will start being output at the end of the next repetition). To achieve this, *TClient* uses repeatedly three non-overlapping rounds synchronized by BS. In the first round, all the agents are initialized according to the input classical algorithm $\mathcal{A}$. In the second round, agents are informed that the previous round has terminated. This is to ensure that no initializing transition is performed during the next, third, round (note that this is not automatically ensured in the *asynchronous* model as ours). In the third round, assuming that a proper initialization of $\mathcal{A}$ is performed, a "simulation" of an execution of $\mathcal{A}$ is performed (with a non-simultaneous start, where the first agent that starts is BS).

Each new round is started by BS and this information is propagated (together with executing the appropriate transitions) to the other agents. To know when to switch to the next round, BS locally counts an appropriate number of events. Below, we show that in order to accomplish the tasks of the first two rounds, BS has to count at least $2 \cdot \mathbf{cv_{min}}$ events for each of those rounds. To accomplish the tasks of the third round, BS has to count at least $\max(2 \cdot \mathbf{cv_{min}}, \mathtt{WCC}_{\mathcal{A}})$ events. To enable such counting, BS has to estimate the upper bounds on $\mathbf{cv_{min}}$ and (possibly) $\mathbf{cv_{max}}$ (and then, on $\mathtt{WCC}_{\mathcal{A}}$). Algorithm *TServerMin* provides such an estimation of $\mathbf{cv_{min}}$ as an input to *TClient* (in variable `mincv`). Algorithm *TServerMax* below provides an estimation of $\mathbf{cv_{max}}$ as an input to *TClient* (in variable `maxcv`). The evaluated $\mathtt{WCC}_{\mathcal{A}}$ (by the `mincv` and `maxcv` values) in *TClient* is denoted by $\mathtt{WCC}^*_{\mathcal{A}}$ (see Fig. 2). Note that if $\mathtt{WCC}_{\mathcal{A}}$ only depends on $\mathbf{cv_{min}}$, *TClient* makes no use of the `maxcv` value provided by *TServerMax*.

We note that in the following analysis we assume that the system is started in an arbitrary configuration, but then, no faults or population changes occur until stabilization. As we already noted in the introduction, this is a common assumption in self-stabilization.

Observation 1. *If* $\mathtt{WCC}_{\mathcal{A}}$ *only depends on* $\mathbf{cv_{min}}$*, the transformer is correct even if* `maxcv` *is incorrect.*

Proof. The observation follows from Lemma 3.8 (that is presented and proven later, in Section 3.4). □

We take advantage of this observation for constructing *TServerMax*.

3.2. Algorithm TServerMax

First, we construct (below) a non-self-stabilizing algorithm *NSSmaxcv* that estimates an upper bound of $\mathbf{cv_{max}}$ (from a non-simultaneous start), in $3 \cdot \mathbf{cv_{min}}$ events. Hence, $\mathtt{WCC}_{NSSmaxcv} = 3 \cdot \mathbf{cv_{min}}$. By Observation 1, we can use the transformer (presented and proven later in Section 3.4) to transform *NSSmaxcv* to be self stabilizing. This may look cyclical, since *TServerMax* (that we construct now) is also a part of the transformer. However, by Observation 1, in this case (where $\mathtt{WCC}_{NSSmaxcv}$ only depends on $\mathbf{cv_{min}}$), the transformer makes no use of *TServerMax*. Hence, the transformation of *NSSmaxcv* results in a self-stabilizing algorithm *TServerMax* estimating $\mathbf{cv_{max}}$. The output of *TServerMax* is provided in variable `maxcv` that resides in BS.

Algorithm NSSmaxcv. In this algorithm, every agent x (including BS) has a variable $\mathtt{maxcat}_x$ which is initialized to $\mathbf{cat}_x$. When a started agent x meets agent y (y becomes started, if not started already), x assigns $\mathtt{maxcat}_x := \max(\mathtt{maxcat}_x, \mathtt{maxcat}_y)$. In $\mathbf{cv_{min}}$ events, all fastest agents meet some started agent and start the computation. Then, in additional $\mathbf{cv_{min}}$ events, every fastest agent f meets a slowest agent, and f assigns a maximum category number to $\mathtt{maxcat}_f$. Finally, in additional $\mathbf{cv_{min}}$ events (after $3 \cdot \mathbf{cv_{min}}$ in overall), some fastest agent meets BS, and BS assigns a maximum category number to its $\mathtt{maxcat}_{BS}$. Now, by the assumption on BS (see Section 2.2), BS estimates the upper bound of $\mathbf{cv_{max}}$ using the maximum category number, and saves the estimation (whose value denoted by $\mathbf{cv}^*_{\mathbf{max}}$) in `maxcv`.

Lemma 3.1. *TServerMax stabilizes to* $\mathbf{cv}^*_{\mathbf{max}}$ *in* `maxcv` *in* $O([2 \cdot \mathbf{cv}^*_{\mathbf{min}}]^{BS})$ *events. The space complexity of TServerMax for every mobile agent is* $O(\mathbf{m})$*, where* $\mathbf{m}$ *is the number of different categories (see Section 2.2).*

Proof. In Section 3.3, we show that *TServerMin* stabilizes in $[2 \cdot \mathbf{cv}^*_{\mathbf{min}} + 1]^{BS}$ events. Hence and by Lemma 3.8, *TServerMax* stabilizes to a correct output in $O([2 \cdot \mathbf{cv}^*_{\mathbf{min}}]^{BS})$ events. Let us now analyze the memory requirements for *TServerMax*. *NSSmaxcv* requires space for variable `maxcat` in every mobile agent. *TServerMin* is not executed in mobile agents. *TClient* requires additional constant memory (a number of bits) for every mobile agent (see Fig. 2). Hence, the lemma follows. □

Memory in a mobile agent $j \neq$ BS:
$\mathbf{cat}_j$: positive integer /* the category number of j */

Memory in BS:
$\texttt{counter}^{\texttt{srv}}$: integer /* counter of the local events at BS */
$\texttt{mincv}, \texttt{mincv}'$: positive integer /* mincv is an output of *TServerMin* */
$\texttt{mincat}, \texttt{mincat}'$: positive integer /* mincat is an output of *TServerMin* */
$\mathbf{cv}^*_j$: positive integer /* the **estimated** (by BS) cover time value of j */

Whenever an agent j communicates with BS:
1 **if** $\texttt{counter}^{\texttt{srv}} \leq 0$ **then**
2 $\texttt{mincv} := \texttt{mincv}'$; $\texttt{mincat} := \texttt{mincat}'$ /* end of a round - output update */
3 $\texttt{mincv}' := \mathbf{cv}^*_j$; $\texttt{counter}^{\texttt{srv}} := \mathbf{cv}^*_j$ /* start of a round - initialization */
4 $\texttt{mincat}' := \mathbf{cat}_j$
else
5 $\texttt{counter}^{\texttt{srv}} := \min(\texttt{counter}^{\texttt{srv}}, \texttt{mincv}') - \max(1, \texttt{mincv}' - \mathbf{cv}^*_j + 1)$
6 $\texttt{mincv}' := \min(\mathbf{cv}^*_j, \texttt{mincv}')$
7 $\texttt{mincat}' := \min(\mathbf{cat}_j, \texttt{mincat}')$

Fig. 1. *TServerMin*.

3.3. Algorithm TServerMin (Fig. 1)

The purpose of this algorithm running at BS is to estimate $\mathbf{cv_{min}}$ (to compute $\mathbf{cv^*_{min}}$) and to find out the agents' minimum category number in a self-stabilizing manner. The output of *TServerMin* is saved in variables mincv and mincat that are used as inputs in *TClient*.

TServerMin executes rounds repeatedly. These are different and separate rounds from those of *TClient*. *TServerMin* uses an event counter (variable $\texttt{counter}^{\texttt{srv}}$) to start a new round when the counter becomes 0 or smaller (line 1). The output (mincv and mincat) is updated once in a round (line 2). A *round* is a segment of an execution of *TServerMin* which ends at line 2; and a *complete round* is a round that had *also* been started at line 3. *Incomplete* rounds arise from a bad (faulty) initialization.

By Lemma 3.2 below, each round lasts at most $[\mathbf{cv^*_{min}} + 1]^{BS}$ events. The output is updated to the *correct* value after each *complete* round (at line 2). Hence, convergence and correctness are ensured after $[2 \cdot \mathbf{cv^*_{min}} + 1]^{BS}$ events.

Informally, in each round, *TServerMin* counts local events (a lower bound on the number of the global events occurring during the counting) to estimate the "time" when at least one fastest agent (with $\mathbf{cv} = \mathbf{cv_{min}}$) has met BS. An adversary can initialize the counter to be very big and cause a very long round. To avoid that and to self-stabilize fast, each time BS meets an agent j with $\mathbf{cv}^*_j$ smaller than the value to which the counter was "initialized" the last time, it acts as follows (line 5): (1) it "initializes" the event counter to $\mathbf{cv}^*_j$ (because it is enough to count $\mathbf{cv^*_{min}}$ to meet at least one fastest agent); and then, (2) it subtracts from the new counter value (resulting from (1)) the number of events that have been already counted at BS since the last such "initialization".

We prove Lemma 3.2 in two parts. First, we show that if the counter at BS is initialized by the adversary or by *TServerMin* (in case of a complete round) to be bigger than or equal to $\mathbf{cv^*_{min}}$, then in at most $[\mathbf{cv^*_{min}} + 1]^{BS}$ events, BS obtains a correct output. Otherwise, if the counter at BS is initialized to be smaller than $\mathbf{cv^*_{min}}$, then the ongoing round cannot be a complete one (since a complete round starts at line 3, where the counter is set to be at least $\mathbf{cv^*_{min}}$) and it ends in less than $[\mathbf{cv^*_{min}} + 1]^{BS}$ events.

Lemma 3.2. *Each round of TServerMin lasts at most* $[\mathbf{cv^*_{min}} + 1]^{BS}$ *events. In addition, at the end of a complete round, the output of TServerMin is correct, i.e.,* $\texttt{mincv} = \mathbf{cv^*_{min}}$ *and* mincat *is set to a minimum category number.*

Proof. If at the first local event of the round at BS, the condition at line 1 is true, the round (an incomplete one) is ended and the lemma follows trivially. Otherwise, there are two cases: (1) at the beginning of a round (just after the faults or at the beginning of a *complete* round, at line 3), $\texttt{mincv}' \geq \mathbf{cv^*_{min}}$ and $\texttt{counter}^{\texttt{srv}} \geq \mathbf{cv^*_{min}}$; or (2) the initial value of $\texttt{mincv}' < \mathbf{cv^*_{min}}$ or/and the initial value of $\texttt{counter}^{\texttt{srv}} < \mathbf{cv^*_{min}}$ (cannot be the case of a complete round; see line 3).
First, note that in both cases ((1) and (2)), $\texttt{mincv}'$ is assigned $\mathbf{cv}^*_j$ (line 3) or $\min(\mathbf{cv}^*_j, \texttt{mincv}')$ (line 6), at least once in the round (at least at the first local event at BS in the round). After that, during the round, we have $\texttt{mincv}' \leq \mathbf{cv^*_{max}}$.
We treat case (1) first. Let e_x be an event (BS, j) and also the $[x]^{BS}$*th* event of the round such that e_x is the first meeting of BS with some fastest agent (j, in this case), in the round. Just before e_x, $\texttt{mincv}' \geq \mathbf{cv}^*_j$ $(= \mathbf{cv^*_{min}})$. In e_x, at line 5, the counter is updated such that $\texttt{counter}^{\texttt{srv}} \leq (\mathbf{cv^*_{max}} - (x-1) - (\mathbf{cv^*_{max}} - \mathbf{cv^*_{min}} + 1)) = \mathbf{cv^*_{min}} - x$. At the next lines, lines 6–7, $\texttt{mincv}'$ is assigned with $\mathbf{cv^*_{min}}$ and $\texttt{mincat}'$ with the minimum category. In at most additional $\mathbf{cv^*_{min}} - x + 1$ local events, the condition at line 1 becomes true and mincv/mincat is assigned with $\texttt{mincv}'/\texttt{mincat}'$, which is equal to $\mathbf{cv^*_{min}}$/minimum category ($\texttt{mincv}'$ / $\texttt{mincat}'$ stayed unchanged since e_x and till now). Thus, in total, in case (1), in at most $\mathbf{cv^*_{min}} + 1$ local events, the round ends (and the new one starts) and at line 2, mincv and mincat are assigned correctly.
In case (2), it is easy to see by the code (line 5) that the round ends in less than $\mathbf{cv^*_{min}} + 1$ local events. In this case, at the first local event of the round at BS, line 5 has to be executed (since line 3 cannot be executed during the round in this case).

Memory in a mobile agent $j \neq$ BS:
$\texttt{round}_j \in \{0, 1, 2\}$ /* the round indicator of j */
$\texttt{fastest}_j \in \{0, 1\}$ /* a bit to mark a fastest agent */
$\texttt{output}_j$ /* set of the output variables of *TClient* */
$\textbf{cat}_j$: positive integer /* the category number of j */

Memory in BS:
$\texttt{counter}^{cln}$: integer /* counter of the local events at BS */
$\texttt{round} \in \{0, 1, 2\}$ /* the round indicator of BS */
mincv : positive integer /* output of *TServerMin*; used here as an input */
mincat : positive integer /* output of *TServerMin*; used here as an input */
maxcv : positive integer /* output of *TServerMax*; used here as an input */
$\texttt{WCC}^*_{\mathcal{A}}$: positive integer /* evaluated $\texttt{WCC}_{\mathcal{A}}$ */
$\texttt{output}_{BS}$ /* set of the output variables of *TClient* */

Whenever agent j communicates with BS:
1 $\texttt{WCC}^*_{\mathcal{A}} :=$ ⟨ $\texttt{WCC}_{\mathcal{A}}$ evaluated by the values of mincv and (possibly of) maxcv ⟩
2 $\texttt{counter}^{cln} := \min(\texttt{counter}^{cln}, \max(2 \cdot \texttt{mincv}, \texttt{WCC}^*_{\mathcal{A}})) - 1$
3 **if** $\texttt{round} = 0$ **then**
4 **if** $\texttt{round}_j = 2$ **then** ⟨ update $\texttt{output}_j$ by the corresponding output variables of $\mathcal{A}$ ⟩
5 ⟨ initialize variables of $\mathcal{A}$ at j and BS ⟩
6 **if** ($\textbf{cat}_j \leq$ mincat) **then** $\texttt{fastest}_j := 1$ /* j is a faster agent */
7 **else** $\texttt{fastest}_j := 0$
8 **if** $\texttt{counter}^{cln} \leq 0$ **then**
9 $\texttt{counter}^{cln} := 2 \cdot \texttt{mincv}$; $\texttt{round} := 1$ /* start of 1-round */
10 **else if** $\texttt{round} = 1$ **then**
11 **if** $\texttt{counter}^{cln} \leq 0$ **then**
12 $\texttt{counter}^{cln} := \max(2 \cdot \texttt{mincv}, \texttt{WCC}^*_{\mathcal{A}})$; $\texttt{round} := 2$ /* start of 2-round */
13 **else** /* $\texttt{round} = 2$ */
14 ⟨ perform a transition of $\mathcal{A}$ for event (BS, j) ⟩
15 **if** $\texttt{counter}^{cln} \leq 0$ **then**
16 $\texttt{counter}^{cln} := 2 \cdot \texttt{mincv}$; $\texttt{round} := 0$ /* start of 0-round */
17 ⟨ update $\texttt{output}_{BS}$ and $\texttt{output}_j$ by the corresponding output variables of $\mathcal{A}$ ⟩
18 $\texttt{round}_j := \texttt{round}$

Whenever agent j communicates with an agent $i \neq$ BS:
19 **if Faster**(i, j) **then** $\texttt{fastest}_j := 0$ /* j is not a faster agent */
20 **if** $(\texttt{round}_i = 0 \wedge \texttt{fastest}_i \wedge \neg\texttt{fastest}_j)$ **then**
21 **if** $\texttt{round}_j = 2$ **then** ⟨ update $\texttt{output}_j$ by the corresponding output of $\mathcal{A}$ ⟩
22 $\texttt{round}_j := 0$; ⟨ initialize variables of $\mathcal{A}$ at j ⟩
23 **else if** $(\texttt{round}_i = 1 \wedge \texttt{fastest}_i \wedge \texttt{round}_j = 0 \wedge \neg\texttt{fastest}_j)$ **then**
24 $\texttt{round}_j := 1$
25 **else if** $(\texttt{round}_i = 2 \wedge \texttt{round}_j \neq 0)$ **then**
26 $\texttt{round}_j := 2$; ⟨ perform a transition of $\mathcal{A}$ for event (i, j) ⟩

Fig. 2. *TClient*.

In this event, at line 5, the counter is updated such that $\texttt{counter}^{\texttt{srv}} < \textbf{cv}^*_{\textbf{min}} - 1$. Then, in less than $\textbf{cv}^*_{\textbf{min}}$ local events, the condition at line 1 becomes true and the round ends at line 2. □

3.4. Algorithm TClient (Fig. 2)

TClient executes at BS and at the mobile agents. It uses as an input, a non-self-stabilizing algorithm $\mathcal{A}$, the output (mincv and mincat) of *TServerMin*. In the case where $\texttt{WCC}_{\mathcal{A}}$ depends also on $\textbf{cv}_{\textbf{max}}$, *TClient* also uses the output (maxcv) of *TServerMax*. The mincv, mincat and maxcv variables are the only variables shared between the various modules of the transformer. *TClient* is the only module who uses them as inputs. It reads these variables, but does not write to them. Hence, the conditions for the fair composition are satisfied (see [29,30]). As *TServerMin* and *TServerMax* are self-stabilizing, they will provide the correct output values eventually. Below, we prove that *TClient* is itself self-stabilizing given that those values are correct.

As we already mentioned, *TClient* executes three rounds repeatedly. The different rounds are numbered from 0 to 2 and denoted 0-round, 1-round and 2-round. Each new round is started by BS and the round number is propagated from agent to agent via their round indicators. We explain the details of this propagation later.

Each of the three rounds has a task to perform. BS counts local events to determine when the task terminates and then, switches to the next round. Each 0-round is used to "reset" (initialize) the states of all the agents to start the upcoming

execution of $\mathcal{A}$ with properly initialized variables. In this round, just before each agent performs the initialization action, it saves the output values from the previous execution of $\mathcal{A}$ (see details below). Each 1-round is used to inform that the previous "reset" round has terminated. This ensures (if no faults occurred recently) that no "reset" action is performed during the next round, 2-round, in which $\mathcal{A}$ is executed.

The output variables of $\mathcal{A}$ that were computed during a 2-round, are saved in the corresponding variables of *TClient*, during the following 0-round. In the code (Fig. 2), we represent these *TClient*'s variables by generic type variables $\texttt{output}_i$, for every agent *i*. These variables are the output variables of *TClient* and of the output (transformed) self-stabilizing algorithm. Note lines 4, 17, and 21 in the code of *TClient* where these variables are updated and saved. This is necessary, because the variables of $\mathcal{A}$ (and possibly its output variables) are re-initialized just after, during the 0-round (while the output variables $\texttt{output}_i$ of *TClient* are not modified until the very end of the *next* 2-round). As $\mathcal{A}$ is assumed to terminate with the same vector of correct output values from a given initial configuration, these output values are re-computed identically in each ("complete") repetition of an execution of $\mathcal{A}$. This provides a stabilization of the output algorithm.

The propagation of a round number is performed as follows. Each agent *i* (including BS) has a round variable—an indicator of the ongoing round in the system. Any agent communicating with BS sets its round indicator according to the round indicator of BS. In addition, any agent eventually knows whether it is a fastest agent or not, via the fastest bit value. These bits are computed in *TClient* in a self-stabilizing way, using primitive **Faster** (in the mobile agents; line 19) and the minimum category number (in BS; lines 6–7) provided by *TServerMin*. In 0- and 1- rounds, non-fastest mobile agents adopt the round value of the fastest ones (with the fastest bit equals 1) only. However, only BS can change the round indicator of a fastest agent. Such an acyclic way of propagating the round number prevents an undesirable instability of the round indicators.

Definition 3.1 (*Complete and Incomplete i*-round)**.** Each *i*-round is a segment of an execution of *TClient* during which the round indicator at BS equals *i*. A *complete* 0-round is a 0-round which starts at line 16 (Fig. 2) and ends at line 9. A *complete* 1-round is a 1-round which starts at line 9 and ends at line 12. A *complete* 2-round is a 2-round which starts at line 12 and ends at line 16.
An *incomplete i*-round is an i-round which is not a complete one. Incomplete rounds arise from a bad (faulty) initialization.

Lemma 3.3. *Assume that the output of TServerMin is correct. Then, each* complete 0-round *lasts* $[2 \cdot \mathbf{cv}^*_{\mathbf{min}}]^{BS}$ *events. In addition, at the end of the* 0-round *(just before line* 9*), the* round *indicators of all the agents are set to* 0 *and all the agents are initialized according to the initialization of the given non-self-stabilizing algorithm* $\mathcal{A}$*. The* fastest *bit of every fastest agent is set to* 1*, and to* 0 *for others. These bits stay unchanged thereafter.*

Proof. Since, the lemma considers a complete 0-round (see Definition 3.1), the round has started at line 16. In this line, $\texttt{counter}^{cln}$ is set to $2 \cdot \texttt{mincv}$. Hence and by line 2, the 0-round lasts $[2 \cdot \mathbf{cv}^*_{\mathbf{min}}]^{BS}$ (during which at least $2 \cdot \mathbf{cv}^*_{\mathbf{min}}$ global events occur in the system). After the round has started at line 16, in $\mathbf{cv}^*_{\mathbf{min}}$ events, every fastest agent f meets BS and sets $\texttt{round}_f := 0$ (line 18) and $\texttt{fastest}_f := 1$ (line 6). From this point on, no line of the code can change $\texttt{fastest}_f$ of any f (see lines 6, 7 and 19). Hence, line 24 cannot be executed by any fastest agent. Line 26 cannot be executed by f with $\texttt{round}_f := 0$. Hence, the value of $\texttt{round}_f$ cannot change during the remaining events of the corresponding 0-round. Hence, since $\texttt{round}_f = 0$ and $\texttt{fastest}_f = 1$ in this round, lines 14 and 26 (the transitions of $\mathcal{A}$) cannot be executed by any fastest f until the end of the round. Note that f is initialized at line 5, at the same meeting with BS, when it assigns $\texttt{round}_f = 0$ and $\texttt{fastest}_f = 1$. Hence, after the first $\mathbf{cv}^*_{\mathbf{min}}$ events of the round, every fastest agent and BS initialize $\mathcal{A}$ internal variables at line 5 and these variables stay unchanged at least until the end of the round.
Starting from the first event of the round, in $\mathbf{cv}^*_{\mathbf{min}}$ events, every fastest agent meets every other non-fastest agent *s*. Thus, the $\texttt{fastest}_s$ bit of *s* is set to 0 (at line 19). After the first $\mathbf{cv}^*_{\mathbf{min}}$ events of a complete 0-round, the fastest bits of non-fastest agents stay unchanged, because the condition in line 6 is correct only for fastest agents. Any other line of the code cannot change a fastest bit to 1. Hence, after the first $\mathbf{cv}^*_{\mathbf{min}}$ events of a complete 0-round, all the fastest bits stay unchanged. After additional $\mathbf{cv}^*_{\mathbf{min}}$ events, in $2 \cdot \mathbf{cv}^*_{\mathbf{min}}$ events in total, every non-fastest agent meets a fastest one, sets its round indicator to 0 and initializes its $\mathcal{A}$ variables, at line 22. From this point on and until the end of the round, lines 14, 26 (the only lines that can change the variables of $\mathcal{A}$) cannot be executed. Line 24 cannot be executed either and thus, the round indicators of all the agents stay unchanged until the end of the round. The lemma follows. □

Lemma 3.4. *Assume that the output of TServerMin is correct. Let e be a TClient execution sequence composed of* 2 *consecutive* complete *i*-rounds *such that* $e \equiv$ *[*0-round 1-round*]. Then, at the end of e (just before line* 12*), the* round *indicator of every agent equals* 1 *and all the agents are initialized according to (the non-self-stabilizing) algorithm* $\mathcal{A}$*. In addition, the corresponding* 1-round *lasts* $[2 \cdot \mathbf{cv}^*_{\mathbf{min}}]^{BS}$ *events.*

Proof. By Lemma 3.3, at the end of a complete 0-round, all the agents are initialized (according to $\mathcal{A}$), all the round indicators are equal to 0, and the fastest bits of all the fastest agents are equal to 1 and to 0 for the others. In addition, at BS, the counter $\texttt{counter}^{cln}$ is set to $2 \cdot \mathbf{cv}^*_{\mathbf{min}}$ and the round indicator to 1 (due to the start of the 1-round; line 9). Then, during the next $[2 \cdot \mathbf{cv}^*_{\mathbf{min}}]^{BS}$ events, the round indicator at BS is 1 (line 2, 9, 10–12). Hence, line 14 (transition of $\mathcal{A}$) cannot be executed. Thus, BS stays initialized during all the 1-round. Hence, the lemma follows for BS.
Let us now prove by induction on the number of events that line 26 cannot be executed (also), during the 1-round. First of all, note that during the 1-round, line 26 can be executed only if one of the agents in the event has its round indicator set to 2. By Lemma 3.3, at the beginning of the 1-round, no round indicator equals 2. Hence, the basis of the induction is proven.

Now, assume that during the first k events (of the 1-round), line 26 cannot be executed. Hence, by the end of the kth event, no round indicator equals 2 (because in the 1-round, only in line 26, round indicator may be set to 2). Hence, the induction is correct also for event $k+1$.
Thus, during all the 1-round, round indicators can be set to 1 or to 0 by a fastest agent (lines 22, 24) and to 1 by BS (line 1). During the first $\mathbf{cv}^*_{\mathbf{min}}$ events in a 1-round, every fastest agent f meets BS and sets its $\mathtt{round}_f$ indicator to 1 (line 18). Hence, after the first $\mathbf{cv}^*_{\mathbf{min}}$ events in the 1-round, round indicators can be set to 1 only (by the fastest agents, line 24). Hence, in $2 \cdot \mathbf{cv}^*_{\mathbf{min}}$ events, all the round indicators are set to 1 and stay unchanged until the end of the 1-round. In addition, by Lemma 3.3 and since we showed that lines 14 and 26 cannot be executed during the 1-round, all the agents are initialized (according to $\mathcal{A}$) at the end of the round. □

Lemma 3.5. *Assume that the output variables of TServerMin and of TServerMax are correct. Let e be a TClient execution sequence composed of* 3 *consecutive* complete *i*-rounds *such that* $e \equiv$ *[*0-round 1-round 2-round*]. Then, at the end of e (just before line* 16*), the* round *indicator of all agents equals* 2 *and the output variables of* $\mathcal{A}$ *are correct. In addition, the corresponding* 2-round *lasts* $[\max(2 \cdot \mathbf{cv}^*_{\mathbf{min}}, \mathtt{WCC}^*_{\mathcal{A}})]^{BS}$ *events.*
If $\mathtt{WCC}_{\mathcal{A}}$ *is a function of* $\mathbf{cv}_{\mathbf{min}}$ *only, the statements in the lemma hold even if* maxcv *is incorrect.*

Proof. By Lemma 3.4, at the end of the 1-round in e, the round indicators of all the agents are equal to 1 and each agent is in the initial state according to $\mathcal{A}$. Then, the next round, starts at line 12, where BS sets its round indicator to 2. By Definition 3.1 at this moment, a complete 2-round, starts. During this round, in line 18, a round indicator can be set only to 2. By a simple induction on the number of events in this 2-round, it can be shown that during the 2-round, no round indicator equals 0. Thus, lines 5 or 22 cannot be executed. This ensures that no initialization action is executed during the 2-round.
First, by lines 2, 12, and 13–16, the 2-round lasts at least $2 \cdot \mathbf{cv}^*_{\mathbf{min}}$ events. We show that there are enough events to set all the round indicators to 2 by the end of the 2-round. During the first $\mathbf{cv}^*_{\mathbf{min}}$ events in this 2-round, every fastest agent meets BS and sets its round indicator to 2. Then and till the end of the round, this indicator stays unchanged, because the conditions at lines 20 and 23 are false (for any meeting (i, j)). Then, in additional $\mathbf{cv}^*_{\mathbf{min}}$ events, a fastest agent meets all the others and they set their indicators to 2, at line 26. These indicators stay unchanged too, until the end of e, by the same lines as for the fastest agents.
BS is the first agent that starts a complete 2-round by setting its round indicator to 2 at line 12. Then, any agent communicating with BS during this 2-round, sets its round indicator to 2, and both agents perform a transition of $\mathcal{A}$ (line 14). Then, each time an agent i with a round indicator equal to 2 meets another agent j with $\mathtt{round}_j = 1$, agent j sets its round indicator to 2, and both agents perform a transition of $\mathcal{A}$ (line 26). Whenever any two agents, both with round indicators equal to 2, meet, they perform a transition of $\mathcal{A}$ too (line 26). Note that (by the above) during this 2-round, a round indicator cannot be changed after it has been set to 2.
Such a behavior simulates an execution of $\mathcal{A}$ exactly (this execution starts non-simultaneously at BS). Note that by the arguments earlier in the proof, no initialization actions of $\mathcal{A}$ are performed during this simulation, but only the actual transitions of $\mathcal{A}$. In addition, due to the correctness of mincv and maxcv, $\mathtt{WCC}^*_{\mathcal{A}}$ is evaluated at line 1 correctly. Hence and because $\mathtt{WCC}_{\mathcal{A}}$ is a non-decreasing function of $\mathbf{cv}_{\mathbf{min}}$ and $\mathbf{cv}_{\mathbf{max}}$, the 2-round in e lasts at least $[\max(2 \cdot \mathbf{cv}^*_{\mathbf{min}}, \mathtt{WCC}_{\mathcal{A}})]^{BS}$ events (line 12), which are at least $\mathtt{WCC}_{\mathcal{A}}$ global events. Hence, at the end of the 2-round in e, the output variables of $\mathcal{A}$ are correct. Note that if $\mathtt{WCC}_{\mathcal{A}}$ is a function of $\mathbf{cv}^*_{\mathbf{min}}$ only, the proof above does not need maxcv. □

Lemma 3.6. *Assume that the output variables of TServerMin and of TServerMax are correct. Let e be a TClient execution sequence composed of* 4 *consecutive* complete *i*-rounds *such that* $e \equiv$ *[*0-round 1-round 2-round 0-round*]. Then, at the end of e (line* 9*), the output variables of TClient (*output*) are correct (satisfy the property of terminal configuration of* $\mathcal{P}$ *that* $\mathcal{A}$ *solves) and stay correct and unchanged thereafter.*
If $\mathtt{WCC}_{\mathcal{A}}$ *is a function of* $\mathbf{cv}_{\mathbf{min}}$ *only, the statements in the lemma hold even if* maxcv *is incorrect.*

Proof. By Lemma 3.5, at the end of the 2-round in e, all the round indicators are set to 2. Now, consider the next 0-round in e. During this round, line 26 cannot be executed for an agent that has already set its round indicator to 0. Line 24 cannot be executed too, because the round indicators of the fastest agents can be set (only by BS) only to 0, during a 0-round. Thus, during this 0-round, the round indicators stay unchanged, after they have been set to 0. Hence, during the last 0-round in e, the update of the output variables of *TClient* by those of $\mathcal{A}$ (at lines 4, 17 and 21) is performed *at most once for any agent*. In addition, when it happens, the output of $\mathcal{A}$ is copied to the corresponding output of the *TClient* before the initialization actions of $\mathcal{A}$ are performed at the same event (at lines 5 and 22). Hence, these actions cannot change the output of $\mathcal{A}$ (from the last execution of $\mathcal{A}$) before it is saved at lines 4, 17 and 21. Hence and by Lemma 3.5, the *correct* output of $\mathcal{A}$ is saved in these lines.
Let us now show that in no more than $2 \cdot \mathbf{cv}^*_{\mathbf{min}}$ events during the last 0-round in e, every agent saves the correct output of $\mathcal{A}$ in the corresponding output variables of *TClient*. Note that by Lemma 3.3, any complete 0-round lasts at least $2 \cdot \mathbf{cv}^*_{\mathbf{min}}$ (global) events. In the first event (BS, j) of the last 0-round in e, at line 17, BS and the agent j update the output variables of *TClient* to the correct ones of $\mathcal{A}$. Then, every (except perhaps j) fastest agent meets BS in $\mathbf{cv}^*_{\mathbf{min}}$ events and updates the output variables of *TClient* (to the correct ones) at line 4 (j does that at line 17, as noted above). Then, in additional $\mathbf{cv}^*_{\mathbf{min}}$ events, all the other agents update these variables at line 21 or at line 4 (if they did not make it already in this round).
From this moment, it is easy to see that the output variables of *TClient* stay untouched until the *end* of the next 2-round (the one that starts *after* e). In the following 0-round, these variables are updated again to the correct values (by the same

arguments as above). These values are identical to those that were saved before, because $\mathcal{A}$ is assumed to terminate with the same vector of correct output values from the same initial configuration.
The lemma follows. □

Lemma 3.7. *Assume that the output variables of TServerMin and of TServerMax are correct. Then, each* i-round *lasts at most* $[\max(2 \cdot \mathbf{cv}^*_{\mathbf{min}}, \mathtt{WCC}^*_{\mathcal{A}})]^{BS}$ *events.*
If $\mathtt{WCC}_{\mathcal{A}}$ *is a function of* $\mathbf{cv}_{\mathbf{min}}$ *only, the statement holds even if* maxcv *is incorrect.*

Proof. The lemma follows from Lemmas 3.3–3.5. □

Lemma 3.8. *Assume that the output variables of TServerMin and of TServerMax are correct. Then, in* $[7 \cdot \max(2 \cdot \mathbf{cv}^*_{\mathbf{min}}, \mathtt{WCC}^*_{\mathcal{A}})]^{BS}$ *events, TClient stabilizes for* $\mathcal{P}$ *(the problem that* $\mathcal{A}$ *solves).*
If $\mathtt{WCC}_{\mathcal{A}}$ *is a function of* $\mathbf{cv}_{\mathbf{min}}$ *only, the statement holds even if* maxcv *is incorrect.*

Proof. Consider an execution sequence e composed of 4 consecutive complete i-rounds $e \equiv$ [0-round 1-round 2-round 0-round]. Let us define the legitimate configurations for *TClient* as the configurations reached after e. Then, by Lemma 3.6, *TClient* stabilizes for $\mathcal{P}$ at the end of e. Consider an incomplete 0-round. By Lemma 3.7 and Definition 3.1, the next complete 0-round starts in at most $[3 \cdot \max(2 \cdot \mathbf{cv}^*_{\mathbf{min}}, \mathtt{WCC}^*_{\mathcal{A}})]^{BS}$. Hence and by Lemma 3.7, *TClient* stabilizes for $\mathcal{P}$ in no more than $[7 \cdot \max(2 \cdot \mathbf{cv}^*_{\mathbf{min}}, \mathtt{WCC}^*_{\mathcal{A}})]^{BS}$. □

Theorem 3.9. *Let the input of the presented transformer be a classical (assuming initialization) algorithm* $\mathcal{A}$ *that solves a static problem* $\mathcal{P}$ *from a non-simultaneous start and legally terminates with the same vector of correct output values in every execution starting from the same initial configuration. In addition, the upper bound* $\mathtt{WCC}_{\mathcal{A}}$ *on the worst case complexity of* $\mathcal{A}$ *is given as an non-decreasing function of* $\mathbf{cv}_{\mathbf{min}}$ *and* $\mathbf{cv}_{\mathbf{max}}$*. Then, the output of the transformer is an algorithm that stabilizes for* $\mathcal{P}$ *in* $O([\max(2 \cdot \mathbf{cv}^*_{\mathbf{min}}, \mathtt{WCC}^*_{\mathcal{A}})]^{BS})$ local *events, which are* $O(\frac{\mathbf{cv}_{\mathbf{max}}}{\mathbf{n}-1} \cdot \max(2 \cdot \mathbf{cv}^*_{\mathbf{min}}, \mathtt{WCC}^*_{\mathcal{A}}))$ global *events.* $\mathtt{WCC}^*_{\mathcal{A}}$ *and* $\mathbf{cv}^*_{\mathbf{min}}$ *are upper bounds on* $\mathtt{WCC}_{\mathcal{A}}$ *and* $\mathbf{cv}_{\mathbf{min}}$*, respectively (these upper bounds are provided by BS).*
The additional memory requirement for the transformation (on top of the memory requirement for $\mathcal{A}$*) is* $O(\mathbf{m})$ *for every mobile agent, where* $\mathbf{m}$ *is the number of different categories (see Section 2.2).*

Proof. The analysis of *TServerMin* and *TServerMax* in this section (Lemmas 3.2 and 3.1) shows that each of these algorithms stabilizes to the correct output in $O([\mathbf{cv}^*_{\mathbf{min}}]^{BS})$ events. In addition, it is clear that *TServerMin* and *TServerMax* make no use of any of the *TClient* variables, so that the variable condition for the fair composition is satisfied. Hence and by Lemma 3.8, *TClient* stabilizes for $\mathcal{P}$ in $O([\max(2 \cdot \mathbf{cv}^*_{\mathbf{min}}, \mathtt{WCC}^*_{\mathcal{A}})]^{BS})$ events.
Now, let us express this complexity by the number of *global* events instead of the local ones at BS. By the cover time property (see Section 2.2), in any $\mathbf{cv}_{BS}$ global events, BS participates in at least one event with every other agent out of $\mathbf{n} - 1$. Hence, in any $\mathbf{cv}_{BS}$ global events, BS counts locally at least $\mathbf{n} - 1$ events. Thus, in $O(\frac{\mathbf{cv}_{\mathbf{max}}}{\mathbf{n}-1} \cdot \max(2 \cdot \mathbf{cv}_{\mathbf{min}}, \mathtt{WCC}^*_{\mathcal{A}}))$ global events ($\mathbf{cv}_{BS} \leq \mathbf{cv}_{\mathbf{max}}$), the convergence and correctness of the output (transformed) algorithm are ensured.
Now, let us show the memory requirement stated in the theorem. The transformer is composed of three modules. By Lemma 3.1, *TServerMax* requires $O(\mathbf{m})$ memory for every mobile agent. *TServerMin* is not executed in mobile agents. *TClient* requires additional constant memory (a constant number of bits) for every mobile agent (see Fig. 2). Hence, the theorem follows. □

4. Impossibility without BS

The technique consisting of executing an algorithm repeatedly in order to obtain its self-stabilizing version (under some conditions on the algorithm) is well known since [10]. In [10,11], an algorithm implementing this technique is called a "re-synchronizer compiler". In [12], the same technique is called a "re-computation of floating output". Let us denote this technique (or approach) by the term the *re-execution approach* . In this approach, a non-stabilizing algorithm with a stable output is re-executed indefinitely. That is, when one execution ends, a new execution starts. If the different re-executions are correctly synchronized, the same stable output is re-computed infinitely often. The transformer we present adopts this technique and adapts it to the model we use here, with the assumption of a powerful base station. In this section, we raise the question of whether or not such an assumption is necessary. We give this question a positive answer.

We start with an informal explanation of what we are going to prove. Later, we define that formally. Consider the re-execution approach that re-executes an algorithm $\mathcal{A}$ indefinitely. *Let us number the re-executions starting from the first one.* According to the re-execution approach in [10–12], each agent uses the same set of variables in all the executions. The motivation for that in our model is even stronger, since the memory of each agent is small. Hence, at any point in time, an agent can participate in only one re-execution i. Indeed, in [10–12], the consecutive re-executions of $\mathcal{A}$ are designed such that they do not overlap in time. That is, *an agent starts performing the transitions of re-execution* $i + 1$ *only after* all *the agents are done performing transitions of re-execution i.*

Now, envision an *imaginary* unbounded *counter* attached to every agent. That is, the counter has no effect on the execution and is just used for the sake of the argument. It counts the number of successive re-executions performed by an agent. The imaginary counter is incremented by one each time an agent performs its first transition in the new re-execution. For simplicity, assume that no faults occur and that all the counters are initialized to the same value simultaneously at the beginning of the first execution of $\mathcal{A}$. Then, according to the previous paragraph, the following *non-overlapping re-execution property* holds for any algorithm using the re-execution approach of [10–12].

Non-overlapping re-execution property: *all the imaginary counters have to differ by no more than one in every configuration*. However, we formalize this property below and prove that it cannot be satisfied without a memory resourceful agent in the model of population protocols with covering (even in the special case of the classical, non-faulty model). We note that we do not prove that a self-stabilizing transformer is impossible when all agents are resource limited (although we conjecture that). We just prove that a self-stabilizing transformer based on the re-execution approach of [10–12] (formally, the transformer satisfying the non-overlapping re-execution property we define below) is impossible without the assumption of a memory resourceful agent.

Definition 4.1 (*Non-overlapping Re-execution Property*)**.** Let $T_{\mathcal{A}}$ be a transition system that performs re-executions of an algorithm $\mathcal{A}$ indefinitely, such that at any point in time, an agent can participate in only one specific re-execution i. Let every agent have an imaginary unbounded counter which is not a part of $T_{\mathcal{A}}$.[4] This imaginary counter is automatically incremented by one, if and only if an agent performs its first transition of $\mathcal{A}$ in the new re-execution. *$T_{\mathcal{A}}$ is said to satisfy the non-overlapping re-execution property* if and only if:

(a) In every configuration, the counters of two agents, differ by no more than 1.
(b) The counter of every agent is incremented infinitely often and is never decremented.

Definition 4.2. Assume the model of population protocols with covering.

- Let a *generic solution* be an algorithm that outputs a transition system for every possible population of (every) size $\mathbf{n}$ and for (every) vector of cover times $\overline{\mathbf{cv}}$.[5]
- Let a *generic solution for the non-overlapping re-execution* be a generic solution providing only transition systems satisfying the non-overlapping re-execution property (Definition 4.1).
- A *local transition system of an agent x* is a projection of the (global) transition system (defined in Section 2.1) on x. That is, it is the set of all the states and the transitions of x. Since the codes of the mobile agents are uniform, their local transition systems are identical. A *(global) transition system is bounded*, if and only if the two local transition systems of a mobile agent and of BS are bounded in size, independently of $\mathbf{n}$.
- A *generic solution is bounded*, if and only if every (global) transition system provided by this solution is bounded.

Theorem 4.1. *Assume the model of population protocols with covering. Even if there is no fault of any kind and even if the counters of Definition 4.2 are initialized to the same value at once, no bounded generic solution for the non-overlapping re-execution exists.*

Proof. Assume by contradiction that there exists a bounded generic solution G for the non-overlapping re-execution. By Definition 4.2, for an infinite set of populations of agents (for every $\mathbf{n}$ and for every vector of cover times) G provides an infinite set of bounded transition systems satisfying the non-overlapping re-execution property. From this set of systems, we extract an infinite sequence $\bar{S} \equiv S_1, S_2, \ldots$ with the following properties: (1) for any system S_i from $\bar{S}$, $\mathbf{cv_{min}}(S_i) \geq \frac{\mathbf{n}_{S_i}\cdot(\mathbf{n}_{S_i}-1)}{2} + \mathbf{n}_{S_i}$, where $\mathbf{n}_{S_i}$ is the number of agents in S_i and $\mathbf{cv_{min}}(S_i)$ is the minimum cover time in system S_i; (2) for any two systems S_i and S_j (from $\bar{S}$), such that $i < j$, $\mathbf{cv_{min}}$ in S_j is greater than $\mathbf{cv_{max}}$ in S_i.

By the pigeonhole principle and because the systems in $\bar{S}$ are bounded, we can extract from $\bar{S}$ an infinite sub-sequence $\bar{S^*} \equiv S_1^*, S_2^*, \ldots$ such that for any two systems S_i^* and S_j^*, when $i < j$, the transitions of the agents in S_i^* and S_j^* are the same. Note that since every system in $\bar{S^*}$ is bounded, it can neither use $\mathbf{n}$, nor the values of the cover times.

Let us consider an execution e in a system S_i^* from $\bar{S^*}$. By the non-overlapping re-execution property (b) (Definition 4.1), there is an agent x whose counter is incremented infinitely often in e. Then, assume $e = e_0t_0e_1t_1e_2t_2e'$, where the transitions t_0, t_1, t_2 cause consecutive increments of the counter of x. Then, by the non-overlapping re-execution property (a), in $t_0e_1t_1e_2t_2$, the counters of all the other agents (of S_i^*) are incremented at least once (since the counter of x has been incremented three times).

Let $e_{pref} = e_0t_0e_1t_1e_2t_2$. Let us choose from $\bar{S^*}$ a system S_j^*, such that $i < j$, $\mathbf{n}_{S_j^*} \geq |e_{pref}|$ and $\mathbf{n}_{S_j^*} > \mathbf{n}_{S_i^*}$. First, note that, because S_j^* is also in $\bar{S}$, e_{pref} satisfies the cover time property of S_j^* (since $\mathbf{cv_{min}}$ in S_j^* is greater than $\mathbf{cv_{max}}$ in S_i^*, implying that all the cover times in S_j^* are greater than the cover times in S_i^*). Second, recall that $\mathbf{cv_{min}}(S_j^*) \geq \frac{\mathbf{n}_{S_j^*}\cdot(\mathbf{n}_{S_j^*}-1)}{2} + \mathbf{n}_{S_j^*}$ (because S_j^* is also in $\bar{S}$). Hence, $\mathbf{cv_{min}}(S_j^*) - |e_{pref}| \geq \frac{\mathbf{n}_{S_j^*}\cdot(\mathbf{n}_{S_j^*}-1)}{2}$. Because during $\frac{\mathbf{n}_{S_j^*}\cdot(\mathbf{n}_{S_j^*}-1)}{2}$ events every agent can meet every other agent, there are enough events ("time") after e_{pref} (in S_j^*) to satisfy $\mathbf{cv_{min}}(S_j^*)$ and every other $\mathbf{cv}$ of S_j^*. Thus, there exists an infinite schedule in S_j^* where e_{pref} is a prefix and the cover time property of S_j^* is satisfied in this schedule. For example, one can get such a schedule by completing e_{pref} to get a prefix p of length $\mathbf{cv_{min}}(S_j^*)$ such that in p, every agent meets every other agent at least once. Then, repeat p indefinitely. Hence and because the transitions of agents in S_i^* and S_j^* are the same, e_{pref} is a possible prefix of an execution in S_j^*. However, because the population size in S_j^* is strictly greater than in S_i^*, there

[4] We emphasize that these imaginary unbounded counters do not reside in the memory of agents. Hence, this does not violate the assumption that the mobile agents are finite-state.

[5] Recall that we assume only $\overline{\mathbf{cv}}$s implying at least one possible schedule (see Remark 2).

exists an agent y of S_j^* that does not participate in any event in e_{pref} and, in particular, in $t_0e_1t_1e_2t_2$. Thus, the counter of y has not been incremented there. Hence, the non-overlapping re-execution property (a) is not satisfied in the configuration at the end of e_{pref}, in S_j^*. This is a contradiction to the assumption that S_j^* is provided by a bounded generic solution for the non-overlapping re-execution. □

5. Examples of transformation

In this section, we present several examples of classical (non-self-stabilizing) algorithms that can be used as inputs to the transformer presented in Section 3 to become self-stabilizing.

The first example is the (non-self-stabilizing) algorithm *NSSmaxcv*, presented in Section 3, which outputs the estimated value of $\mathbf{cv_{max}}$ at BS. The corresponding transformed algorithm is called there *TServerMax*. Two another examples appear below.

5.1. Minimum finding and leader election

Consider a system in which every agent has some characterizing value and there is a need to find out and mark agents having a minimum (or maximum) characterizing value. Note that provided that the characterizing value is unique for every agent, a solution to this problem also provides a common way for electing a leader. One may note that in our model, electing a leader may be easy. BS could always be the leader. However, it may be undesirable that the leader is known in advance, or it may be that BS is inappropriate for playing a leader role. For example, consider an application where agents are animals with attached sensors and given BS is not powerful enough to be a leader. For example, BS is a non-mobile station that has a very limited communication range. Hence, it may be preferable to find out (in a self-stabilizing way) an animal having a winning set of characteristics for being a leader.

The specification of the minimum finding problem states that only the agents having the minimum characterizing value have to be marked as "winners". More precisely, each agent has a status variable with values in $\{winner, nonwinner\}$. Each execution terminates and at the termination (the legal configuration is that) only the agents with the minimum characterizing value have their variable status set to $winner$.

We describe and analyze, informally, a simple non-self-stabilizing algorithm *MinFinding* that solves the minimum finding problem above. First, a status variable of every agent is initialized to $winner$. Each time, two agents that have started the computation meet, the one with the larger characterizing value sets its status variable to $nonwinner$. We allow "non-winners" being marked as "winners" before the termination, since, anyhow, a future self-stabilizing solution cannot guarantee that before stabilization.

Consider an agent i that is the first to start in an execution of *MinFinding*. In at most $\mathbf{cv_{max}}$ events, agent i meets all the other agents, so they start too. In at most additional $\mathbf{cv_{max}}$ events, an agent with the minimum characterizing value meets every other agent, so they set their status variables to $nonwinner$. The status variables of the agents with the minimum characterizing value, stay unchanged. Since they are initialized to $winner$, algorithm *MinFinding* terminates in at most $2 \cdot \mathbf{cv_{max}}$ events, and the legal configuration is reached. Hence, there is an upper bound $\mathtt{WCC}_{MinFinding} = 2 \cdot \mathbf{cv_{max}}$. In addition, note that *MinFinding* satisfies the requirement that the correct vector of output variables (the status variables) is unique for a given initial configuration. Also note that *MinFinding* is indeed not self-stabilizing, because, if started from a configuration in which the status variables of all agents are equal to $nonwinner$, these variables will not change during any execution.

Proposition 5.1. *The transformer in Section 3 transforms algorithm MinFinding, with* $\mathtt{WCC}_{MinFinding} = 2 \cdot \mathbf{cv_{max}}$, *into a self-stabilizing solution for the minimum finding problem specified above.*

5.2. Gathering of information

A detailed presentation of the study of the Gathering Problem (GP) in the model of population protocols with covering appears in [14]. In GP, each agent has an initial input value. The aim is, for BS, to output the multi-set of these values. Note that this means we should avoid replication (except in the case where the same value is the input of multiple agents). A legal configuration is a configuration in which BS has gathered all the inputs (and each, exactly once). In the self-stabilizing version, BS has to keep forever the correct multi-set from some point of the execution.

In [14], an algorithm *TTFM* (Transfer to The Faster Marked) is presented for solving GP. This is a one shot, non-stabilizing algorithm with initialization, providing the same solution in all executions starting with the same set of agents and inputs. In *TTFM*, each agent i can store some $\mathbf{M}$ values on top of its own initial value. These can be input values of other agents transferred to i. Every agent transfers its own input value only once. Whenever an agent i transfers a non-own value to another agent j, the value is copied to j and then, deleted from i's memory. For the example here, we proceed to describe just a simplified version of *TTFM* where $\mathbf{M} \geq \mathbf{n}$. The algorithm in [14] can handle any size of $\mathbf{M}$ and it is also suitable for the transformation in this paper.

Thus, in *TTFM*, each mobile agent, in addition to the memory for the transferred values, has a "mark" bit initialized to 1. Each time two agents (f and s) meet, the one (s) with a strictly larger cover time sets its own "mark" bit to 0. Then, (only) if f's mark bit is 1 (or f is BS), s transfers to f all the values it holds. This rule ensures that values can be transferred to a faster agent, only if the latter did not meet (before) a third agent that is yet faster.

According to [14], the worst case event complexity of the presented algorithm *TTFM* that assumes a simultaneous start is $2 \cdot \mathbf{cv_{min}} - |F|$, where F is the set of the fastest agents. Then, as is shown in Section 2.3 (and in [14]), when assuming a non-simultaneous start, the complexity increases by at most additional $\min(2 \cdot \mathbf{cv_{min}}, \mathbf{cv_{max}})$ events.

Proposition 5.2. *The transformer in Section 3 transforms algorithm TTFM (for* $\mathbf{M} \geq \mathbf{n}$*), with* $\mathtt{WCC}_{TTFM} = 2 \cdot \mathbf{cv_{min}} + \min(2 \cdot \mathbf{cv_{min}}, \mathbf{cv_{max}})$*, into a self-stabilizing solution for GP.*

Acknowledgements

The work of the first author was partially supported by grants from Grand Large project, INRIA Saclay. The work of the second author was partially supported by a grant from the Israel Council for Higher Education. The work of the second and third authors was partially supported by a grant from the Israel Science Foundation from the B. and G. Greenberg Research Fund (Ottawa) and from the Technion Gordon Center. The work of the third author was partially supported by grants from the Technion TASP Center.

References

[1] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, D. Rubenstein, Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with Zebranet, in: ASPLOS, 2002, pp. 96–107.
[2] S. Lahde, M. Doering, W. Pöttner, G. Lammert, L.C. Wolf, A practical analysis of communication characteristics for mobile and distributed pollution measurements on the road, Wirel. Comm. Mob. Comput. 7 (10) (2007) 1209–1218.
[3] D. Angluin, J. Aspnes, Z. Diamadi, M.J. Fischer, R. Peralta, Computation in networks of passively mobile finite-state sensors, in: PODC, 2004, pp. 290–299.
[4] D. Angluin, J. Aspnes, Z. Diamadi, M.J. Fischer, R. Peralta, Computation in networks of passively mobile finite-state sensors, Distrib. Comput. 18 (4) (2006) 235–253.
[5] D. Angluin, J. Aspnes, D. Eisenstat, E. Ruppert, The computational power of population protocols, Distrib. Comput. 20 (4) (2007) 279–304.
[6] R. Guerraoui, E. Ruppert, Even small birds are unique: population protocols with identifiers, Technical Report CSE-2007-04, York University, 2007.
[7] J. Beauquier, J. Clement, S. Messika, L. Rosaz, B. Rozoy, Self-stabilizing counting in mobile sensor networks with a base station, in: DISC, 2007, pp. 63–76.
[8] M. Fischer, H. Jiang, Self-stabilizing leader election in networks of finite-state anonymous agents, in: OPODIS, 2006, pp. 395–409.
[9] D. Angluin, J. Aspnes, D. Eisenstat, Fast computation by population protocols with a leader, Distrib. Comput. 21 (3) (2008) 183–199.
[10] B. Awerbuch, G. Varghese, Distributed program checking: a paradigm for building self-stabilizing distributed protocols (extended abstract), in: FOCS, 1991, pp. 258–267.
[11] G. Varghese, Self-stabilization by local checking and correction, Ph.D. Thesis, Cambridge, MA, USA, 1993.
[12] S. Dolev, Self-Stabilization, The MIT Press, 2000.
[13] J. Beauquier, J. Burman, J. Clement, S. Kutten, Brief announcement: non-self-stabilizing and self-stabilizing gathering in networks of mobile agents—the notion of speed, in: PODC, 2009, pp. 286–287 (a brief announcement).
[14] J. Beauquier, J. Burman, J. Clement, S. Kutten, On utilizing speed in networks of mobile agents, in: PODC, 2010, pp. 305–314.
[15] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, E. Ruppert, When birds die: making population protocols fault-tolerant, in: DCOSS, 2006, pp. 51–66.
[16] A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, J. Scott, Impact of human mobility on the design of opportunistic forwarding algorithms, in: INFOCOM, 2006, pp. 1–13.
[17] I. Rhee, M. Shin, S. Hong, K. Lee, S. Chong, On the levy-walk nature of human mobility, in: INFOCOM, 2008, pp. 924–932.
[18] H. Cai, D.Y. Eun, Crossing over the bounded domain: from exponential to power-law inter-meeting time in MANET, in: MOBICOM, 2007, pp. 159–170.
[19] T. Karagiannis, J.L. Boudec, M. Vojnovic, Power law and exponential decay of inter contact times between mobile devices, in: MOBICOM, 2007, pp. 183–194.
[20] S. Hong, I. Rhee, S.J. Kim, K. Lee, S. Chong, Routing performance analysis of human-driven delay tolerant networks using the truncated levy walk model, in: MobilityModels, 2008, pp. 25–32.
[21] J. Burman, Overcoming the effect of asynchrony in distributed algorithms, Ph.D. Thesis, 2010.
[22] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.
[23] D. Angluin, J. Aspnes, M.J. Fischer, H. Jiang, Self-stabilizing population protocols, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 3 (4) (2008).
[24] S. Katz, K.J. Perry, Self-stabilizing extensions for message-passing systems, Distrib. Comput. 7 (1) (1993) 17–26.
[25] Y. Afek, S. Kutten, M. Yung, The local detection paradigm and its application to self-stabilization, Theoret. Comput. Sci. 186 (1–2) (1997) 199–229.
[26] B. Awerbuch, B. Patt-Shamir, G. Varghese, S. Dolev, Self-stabilization by local checking and global reset (extended abstract), in: WDAG, 1994.
[27] S. Ghosh, A. Gupta, T. Herman, S.V. Pemmaraju, Fault-containing self-stabilizing algorithms, in: PODC, 1996, pp. 45–54.
[28] J. Beauquier, J. Burman, S. Kutten, Making population protocols self-stabilizing, in: SSS, 2009, pp. 90–104.
[29] G. Tel, Introduction to Distributed Algorithms, 2nd ed., Cambridge University Press, 2000.
[30] T. Herman, Adaptivity through Distributed Convergence, Ph.D. Thesis, University of Texas at Austin, 1991.