Mediated Population Protocols: Leader Election and Applications

Shantanu Das*, Giuseppe Antonio Di Luna**, Paola Flocchini**, Nicola Santoro***, and Giovanni Viglietta**

Abstract. Mediated population protocols are an extension of population protocols in which communication links, as well as agents, have internal states. We study the leader election problem and some applications in constant-state mediated population protocols. Depending on the power of the adversarial scheduler, our algorithms are either stabilizing or allow the agents to explicitly reach a terminal state.
We show how to elect a unique leader if the graph of the possible interactions between agents is complete (as in the traditional population protocol model) or a tree. Moreover, we prove that a leader can be elected in a complete bipartite graph if and only if the two sides have coprime size.
We then describe how to take advantage the presence of a leader to solve the tasks of token circulation and construction of a shortest-path spanning tree of the network. Finally, we prove that with a leader we can transform any stabilizing protocol into a terminating one that solves the same task.

1 Introduction

Background. The *population protocol* model, introduced in the seminal paper of Angluin et al. [3] has recently received a lot of interest among researchers in distributed computing. The model consists of a set of simple anonymous finite-state agents that interact pairwise, and each interaction changes the state of both agents. Normally each pair of agents is supposed to interact infinitely often in any infinite execution of the protocol; however, these interactions may occur in any arbitrary order. This models the asynchrony and uncertainty in a distributed system. Moreover, as the agents have constant memory independent of the size of the system, this means that the protocol can be scaled to populations of any size. The population protocol model is useful for modeling large-scale networks consisting of small mobile devices, such as sensor networks or swarms of microrobots.

Since the introduction of this model, several variants of population protocols have been studied. For example, there could be restrictions on which only certain

* LIF, Aix-Marseille University, and CNRS, Marseille, France.
E-mail: shantanu.das@lif.univ-mrs.fr.
** University of Ottawa, Canada.
E-mails: {gdiluna, paola.flocchini, gvigliet}@uottawa.ca.
*** Carleton University, Ottawa, Canada. E-mail: santoro@scs.carleton.ca.

pairs of agents are allowed to interact, giving rise to arbitrary *interaction graphs* instead of the complete graph [2]. This paper considers the interaction graph to be an arbitrary connected graph on the set of agents. Another possibility is to consider restrictions on the schedule of interactions, e.g., allowing a periodic scheduler, or a k-bounded scheduler, or a probabilistic scheduler [1].

The power of population protocols in terms of what kinds of predicates can be computed by them has been studied extensively. Angluin et al. [5] showed that the class of computable predicates is exactly the class of semilinear predicates (or, equivalently, all predicates that can be defined by first-order logical formulas in Presburger arithmetic). Further studies introduced enhancements in the model to increase its computational power and allow the computation of larger classes of predicates: endowing each agent with non-constant memory [1], assuming the presence of a leader [7], allowing a certain amount of information to be stored on the edges of the interaction graph [12,13]. In the present paper we study the latter category of population protocols, which are called *mediated population protocols*. We assume that the amount of memory per node and per edge of the graph is constant, and we study what can be computed in several restricted classes of interaction graphs and with several types of schedulers.

Our Contributions. In this paper we focus on algorithms to elect a unique *leader* in a mediated population protocol, as well as applications of a leader in several common situations. In Section 2, we formally define the mediated population protocol model and related concepts. The types of schedulers we consider are the *recurrent* scheduler, which only implements a bland notion of fairness on interactions, and the *k-bounded* scheduler, which cannot neglect any interaction for too long. We also distinguish between *stabilizing* protocols, which reach a configuration in which no agent changes state any more, and *terminating* protocols, in which the agents "realize" that the configuration is stable (or about to become stable), and explicitly terminate the execution. Typically, when the scheduler is recurrent and the task is not trivial, there exists no terminating protocol to solve it. In these cases, we will give only stabilizing protocols. On the other hand, when the scheduler is k-bounded, we will give terminating protocols.

In Section 3, we study the problem of leader election in several network topologies: complete graphs, complete bipartite graphs, and trees. We prove that a unique leader can always be elected in a complete graph and in a tree, and we give a characterization of the complete bipartite graphs in which a leader can be elected under a 1-bounded scheduler: those are the complete bipartite graphs in which the two sides have coprime sizes.

In Section 4, we assume that the network contains a unique leader, and we show how to use this feature to accomplish some typical tasks in arbitrary networks. First we show how to solve the *token circulation* problem and how to construct a *shortest-path spanning tree*. Then we show how to convert any stabilizing protocol into an "equivalent" but terminating one. As a byproduct, given any protocol for the 2-bounded scheduler, we can make it work also under the k-bounded scheduler, for any $k > 2$. By combining these solutions with the leader election protocols of Section 3, we can solve the same tasks even if a

leader is not given in advance, provided that the network is a complete graph, a complete bipartite graph with coprime sides, or a tree.

Related Work. The task of electing a leader has been extensively studied in the context of (non-mediated) population protocols. The majority of papers focus on self-stabilizing leader election under the assumption that the scheduler is *globally fair*. This is a more powerful scheduler than the recurrent one, and it may or may not be more powerful than the k-bounded one. It has been shown that leader election requires either as many states as agents [9] or the presence of an *oracle* that informs agents about the presence of a leader in the system [15]. Concerning restricted interaction graphs, in [10] self-stabilizing leader election algorithms for trees are given; the case of the ring is studied in [15]. Both papers assume the presence of an oracle. These results have been extended, under the same set of assumptions, to arbitrary graphs in [6]. In [4], a constant-space algorithm for the ring graph is also given. In the context of mediated population protocols, a non-constant-space algorithm for leader election is shown in [16].

Under the globally fair scheduler, the self-stabilizing construction of a spanning tree has been investigated in [4], where an algorithm requiring $\mathcal{O}(\log D)$ states is given, D being the graph's diameter. In the same paper, a self-stabilizing token circulation algorithm for the ring graph is given. In a model similar to population protocols, a token circulation algorithm for arbitrary graphs is discussed in [11], assuming the presence of an oracle.

Several papers assumed a unique leader as a computational tool to enhance the power of population protocols. Counting algorithms are given in [7]; in [8], a self-stabilizing transformer for general protocols has been studied in a slightly different model and under the additional assumption of unbounded memory. In the context of fault tolerance, [14] uses a leader to make any protocol tolerant to omission failures.

Other papers have discussed the computational power of mediated population protocols in terms of the predicates that can be computed [12,13].

In light of the above results, our paper represents a breakthrough in that we show how to elect a leader in some large classes of networks using only a constant number of states per agent and per edge; moreover, we often make weaker assumptions on the scheduler, and we do not resort to querying an oracle. The same holds for the applications of leader election, which in addition apply to all networks in which a leader is present (regardless of the interaction graph's topology).

2 Model and Definitions

Network and Configuration. A *network* is an unoriented connected finite graph G on a set V of at least two vertices, which are called *agents*. Each agent has an *agent state* belonging to a finite set Q. In turn, Q is partitioned into *input states* Q_I, *work states* Q_W, and *terminal states* Q_T.

Each edge of G has a *port* for each of its two endpoints; the set of all ports is denoted by P. If $\{a, b\}$ is an edge of G, we denote by $p(a, b)$ the port on a's side

of $\{a, b\}$ and by $p(b, a)$ the port on b's side. Each port has a *port state* belonging to a finite set U; there exists a unique *initial state* $u_0 \in U$.

A *configuration* C is a pair of functions (f, g), where $f\colon V \to Q$ and $g\colon P \to U$. C is said to be *initial* if $f(V) \subseteq Q_I$ and $g(P) = \{u_0\}$.

Interaction and Scheduler. An *interaction* is an ordered pair of agents (a_s, a_r), where a_s is called the *sender* and a_r is called the *receiver*, such that $\{a_s, a_r\}$ is an edge of G. The set of possible interactions of G (i.e., two for each edge) is denoted by I. A *schedule* S is an infinite sequence of interactions, i.e., $S\colon \mathbb{N} \to I$. A *scheduler* is a set of schedules. We define the following schedulers:

- the *recurrent scheduler* is the set of schedules in which each interaction of I appears infinitely often;
- the *k-bounded scheduler*, where k is a positive constant, is the set of schedules that belong to the recurrent scheduler and such that, between two consecutive occurrences of the same interaction within a schedule, none of the other interactions appears more than k times.

Let us observe that all the schedules in the 1-bounded scheduler are periodic.

Transition Function, Execution, and Task. A *transition function* (or *protocol*) is a function $\delta\colon Q \times U \times Q \times U \to Q \times U \times Q \times U$ such that, if $\delta(q_s, u_s, q_r, u_r) = (q'_s, u'_s, q'_r, u'_r)$ and $q_s \in Q_T$ (respectively, $q_r \in Q_T$), then $q_s = q'_s$ (respectively, $q_r = q'_r$). That is, δ leaves terminal states unchanged.

Given a configuration $C = (f, g)$, we say that configuration $C' = (f', g')$ *results* from C by the interaction $i = (a_s, a_r)$ according to the transition function δ if f' coincides with f on $V \backslash \{a_s, a_r\}$, g' coincides with g on $P \backslash \{p(a_s, a_r), p(a_r, a_s)\}$, and

$$(f'(a_s), g'(p(a_s, a_r)), f'(a_r), g'(p(a_r, a_s))) = \delta(f(a_s), g(p(a_s, a_r)), f(a_r), g(p(a_r, a_s))).$$

If this is the case, we write $C \xrightarrow{\delta, i} C'$.

The *execution* of a schedule $S = (i_0, i_1, \ldots)$ from an initial configuration C_0 according to a transition function δ is the sequence of configurations $(C_0, C_1, \ldots)$ such that, for every $j \in \mathbb{N}$, $C_j \xrightarrow{\delta, i_j} C_{j+1}$. Let an execution $E = (C_j)_{j \geq 0}$ be given, with $C_j = (f_j, g_j)$. We say that E is *stable* if there is a $j^* \in \mathbb{N}$ such that, for every $j' > j^*$, $f_{j'} = f_{j'+1}$. We say that E *terminates* if there is a $j^* \in \mathbb{N}$ such that $f_{j^*}(V) \subseteq Q_T$. Note that an execution that terminates is also stable.

A *task* or *problem* is a set of executions. We say that a protocol δ on a set of agent states Q and a set of port states U *solves* a task $\mathcal{T}$ under scheduler $\mathcal{S}$ in a given network if the execution according to δ of every schedule in $\mathcal{S}$ from any initial configuration is in $\mathcal{T}$. If such executions are all stable, the protocol is said to be *stabilizing*. If such executions all terminate, the protocol is said to be *terminating*.

Algorithmic notation. When describing transition functions, we will sometimes use an "algorithmic style" (cf. Figures 1 and 2). When the interaction (a, b) occurs, the function Transition function is applied to the tuple $(a, p(a, b), b, p(b, a))$; note that, with a little abuse of notation, we identify agents and ports with their respective states. In our formalism, a state is seen as a tuple of variables. To refer to variable x of the state of agent a, we use the expression $a.x$.

3 Leader Election

In this section we study the task of electing a leader in several types of networks. Formally, the set of agent states includes some *leader states*, and *leader election* is the task consisting of the executions in which eventually there is a unique agent in the network with a leader state. Note that a protocol solving the leader election problem need not be stabilizing.

3.1 Complete Graphs

If the network is a complete graph, there is a simple leader election protocol that works under the recurrent scheduler.

Theorem 1. *There exists a stabilizing protocol that solves the leader election problem in K_n, for all $n > 1$, under the recurrent scheduler.*

Proof. We use only two agent states: an input state, which is also a leader state, and a work state, which is a non-leader state. There are no terminal states and only one port state. Whenever two agents with the leader state interact, the sender retains the leader state and the receiver takes the non-leader state. In all other cases, the agents retain their states.

As a result, in every execution all agents will initially have the leader state (because it is the only input state) and, whenever two leaders meet, one will be "eliminated". Since all ordered pairs of agents are going to interact infinitely many times (because the network is the complete graph and the scheduler is recurrent), it is obvious that eventually only one agent with the leader state will remain, and its state will never change. Hence the protocol is stabilizing. □

3.2 Complete Bipartite Graphs

Next we give a characterization of the complete bipartite graphs in which the leader election problem is solvable under the 1-bounded scheduler. When the problem is solvable, we can also give a terminating protocol.

Theorem 2. *There exists a (terminating) protocol that solves the leader election problem in $K_{m,n}$ under the 1-bounded scheduler if and only if m and n are coprime.*

Proof. Suppose that m and n are coprime. Without loss of generality, let $m < n$. The idea of our protocol is to make the m agents in the smaller side of the graph "eliminate" m agents in the larger side. What is left is a smaller complete bipartite graph, on which the same procedure is repeated until only one agent remains: this agent will be the leader.

The protocol uses the fact that the schedule has period $2mn$, and hence the concept of *round* can be defined as a set of $2mn$ consecutive interactions. Whenever an agent a is involved in an interaction for the first time (which is easy to detect, since a still has an input state), and its partner is some agent b, the port $p(a, b)$ is "marked" with a special state that also encodes the role of a in the interaction (i.e., sender or receiver). So, when a sees the marked port again, and its role is the same as the one encoded by the mark, it knows that a new round has started. With this technique, agents can implicitly coordinate their actions and do different things at different rounds.

In the first round, a maximal matching is constructed. Initially all agents are unmatched; whenever two unmatched agents interact, they become matched and change their state accordingly. By the end of the first round, all agents in the smaller side of the graph have been matched. During the second round, all agents discover if they belong to the smaller side or the larger side: the ones in the smaller side will see some unmatched agents, and the ones in the larger side will only see matched agents. Note that each agent can perform this check by having a "flag" in its state that is cleared at the beginning of the round and is set whenever an unmatched agent is encountered. During the third round, the agents in the smaller side revert their state to unmatched, and the matched agents in the larger side become eliminated.

This three-round cycle is then repeated, ignoring the eliminated agents, until only one agent remains unmatched (note that this happens if and only if m and n are coprime). Finally, this situation has to be detected by all agents, so that the protocol can terminate. This is done by adding another flag, which is used in the third round, to check if the encountered unmatched agents are more than one. So, when only one unmatched agent is left, the agents in the opposite side detect it and get a terminal (non-leader) state. As soon as the other agents see some terminated agents, they also get a terminal state (which will be a leader or non-leader state, depending on whether they are unmatched or matched).

Suppose now that m and n are not coprime, and let $d > 1$ be their greatest common divisor. Partition one side of the graph into m/d groups of size d and the other side in n/d groups of size d. Let $A = \{a_0, \ldots, a_{d-1}\}$ and $B = \{b_0, \ldots, b_{d-1}\}$ be two groups of agents on opposite sides, and let $S_{A,B}$ be the following sequence of d^2 interactions: first all the interactions of the form (a_j, b_j), with $0 \leq j < d$, then all the interactions of the form $(a_j, b_{j+1 \bmod d})$, then all the interactions of the form $(a_j, b_{j+2 \bmod d})$, etc. We then construct a sequence S by concatenating all the sequences $S_{A,B}$ for every ordered pair (A, B) of groups of agents located on opposite sides of the graph. The resulting sequence has length $2mn$ and involves all possible interactions in the network. Finally, we construct a schedule S^* by concatenating infinitely many copies of S.

Suppose that in the initial configuration all agents have the same input state (which is a valid initial configuration, regardless of the protocol), and suppose that the above scheduler S^* is executed. Then, every d interactions, all agents in a same group will have the same state, and in particular there will not be a unique agent with a leader state. This means that no protocol solves the leader election problem under the 1-bounded scheduler (since S^* is 1-bounded). □

3.3 Tree Graphs

If the network is a tree and the scheduler is recurrent, we can always elect a leader with a stabilizing protocol. Moreover, if the scheduler is k-bounded, we have a terminating protocol.

Theorem 3. *For every $k \geq 1$, under the k-bounded scheduler (respectively, under the recurrent scheduler), there exists a terminating (respectively, stabilizing) protocol that solves the leader election problem in every tree.*

Proof. First we describe how to elect a stable leader if the scheduler is recurrent: the protocol is summarized in Figure 1. Then we will show how to make the same protocol terminate under the k-bounded scheduler. The idea of the protocol is to establish a parent-child relation between adjacent vertices of the tree in such a way that eventually the tree becomes rooted: the root will then be the leader. Assuming that $\{a, b\}$ is an edge, the way we represent the fact that a is a parent of b is by setting a *parent flag* in the state of port $p(a, b)$. In the initial configuration, no parent flag is set. Each agent has a *parents variable* too, which counts how many parents the agent has (initially 0, and ranging from 0 to 2). Both agents and ports also have a *busy flag*, initially not set.

Whenever a pair of non-busy agents (a, b) is activated and none of $p(a, b)$ and $p(b, a)$ have their parent flag set, then the parent flag of $p(a, b)$ is set, encoding the fact that b has become a child of a. Then the parents variable of b is incremented; if b has now two parents, both b and $p(b, a)$ become "busy" by setting their respective busy flag. b will then look for its "old parent" c. Note that, while b has its busy flag set, it will accept no more parents or children. When b interacts again with c (which is recognizable because the parent flag of $p(c, b)$ is set and the busy flag of $p(b, c)$ is not set) and c is not busy, b becomes a parent of c and c becomes a child of b (i.e, the parent flags of $p(b, c)$ and $p(c, b)$ are switched). Also, the parents variable of c is incremented; if c has two parents, then both c and $p(c, b)$ become busy. At the same time, the parents variable of b is decremented, meaning that b has a unique parent again. So, the next time b interacts with a (which is recognizable because the parent flag of $p(a, b)$ is set), b will clear its own busy flag, as well as the busy flag of $p(b, a)$. In the meantime, if the busy flag of c is set, c looks for its old parent d and does the same operations that b just did; then d will do the same, etc.

Let us see how an execution of this algorithm works globally. Initially, all edges of the network are "unoriented"; as soon as some edge is activated, it gets an "orientation", telling which of the two endpoints is the parent. As the

execution continues, a forest of oriented subtrees is constructed, and each tree in this forest has a unique root. When two subtrees meet because an interaction (a, b) occurs, they merge, and the root of a's subtree becomes the root of the new tree. So, the orientations of all the edges in the path from b to the root of b's subtree have to reverse. While edges are being reversed, the agents involved become temporarily busy, so that no other subtrees can merge at those points and interfere with the process. Note that deadlocks are impossible because the network is cycleless. Also, progress will always be made, because the scheduler is recurrent, and therefore all possible interactions will eventually occur. When all the subtrees have finally merged and all edges stop reversing, the entire tree is oriented and has a unique root. The root is also the only agent in the tree whose parent flag is not set. If we define this as a leader state, we have a stabilizing leader election protocol.

```
1:  Agent variables
2:  parents := 0
3:  busy := false
4:
5:  Port variables
6:  parent := false
7:  busy := false
8:
9:  Transition function δ(a, p(a, b), b, p(b, a))
10: if ¬a.busy ∧ ¬b.busy ∧ ¬p(a, b).parent ∧ ¬p(b, a).parent then
11:     p(a, b).parent := true
12:     b.parents := b.parents + 1
13:     if b.parents = 2 then
14:         b.busy := true
15:         p(b, a).busy := true
16: else if a.busy ∧ ¬b.busy ∧ ¬p(a, b).busy ∧ p(b, a).parent then
17:     p(a, b).parent := true
18:     p(b, a).parent := false
19:     a.parents := a.parents − 1
20:     b.parents := b.parents + 1
21:     if b.parents = 2 then
22:         b.busy := true
23:         p(b, a).busy := true
24: else if a.busy ∧ a.parents = 1 ∧ p(b, a).parent then
25:     a.busy := false
26:     p(a, b).busy := false
```

Fig. 1: Stabilizing leader election in a tree

Suppose now that the scheduler is k-bounded, and let us show how to make the above protocol terminate. Observe that a technique similar to the one used in Theorem 2 allows any agent to determine when it has interacted with all of its neighbors in the tree. The first time an agent is involved in an interaction as the sender, it marks the corresponding port. Then it counts how many times that same interaction occurs; at the $k + 1$th occurrence, the agent knows that all the possible interactions have occurred at least once, and therefore it has interacted

with all of its neighbors. Since k is fixed, a constant number of agent states is sufficient to implement this counter. Furthermore, an agent can determine whether it is a leaf or an internal vertex of the network: if the agent sees a marked port every time it is activated as the sender of an interaction for $k+1$ times consecutively, then it is a leaf. Now, if a leaf agent has a parent, it knows that it will never become leader, and therefore it can get a terminal non-leader state. More generally, if an agent with a parent realizes that all its neighboring agents except its parent are in a terminal state, then it can get a terminal non-leader state, as well. Once again, this check can be performed with a flag and a finite counter. It is easy to prove by induction that eventually all agents except the final root of the tree will get a terminal non-leader state. When this happens, the root easily realizes and gets a terminal leader state. □

4 Applications of a Unique Leader

In this section we will show how the presence of a unique leader can help us solve several different tasks. Formally, these are tasks consisting of executions in which the initial configuration has a unique agent in a leader state.

4.1 Token Circulation

Here we provide a stabilizing solution to the *token circulation* task that works in every network under the recurrent scheduler. Formally, the set of agent states includes some *token states*, and token circulation is the task consisting of the executions in which, if in the initial configuration there is a unique agent with token state, then in every configuration there is a unique agent with token state, and each agent has token state in at least one configuration.

Protocol Variables. Each agent's state consists of three flags: *token*, *tree*, and *summoning*. Each port's state consists of the single flag *parent*. The token states coincide with the leader states, and are those in which $token = true$. All flags of all agents and ports are initially set to $false$, with the exception of the *token* flag of the leader, which is set to *true*.

Protocol Description. Our protocol is given in Figure 2. The token circulates along the edges of a spanning tree of the network, which is constructed incrementally as the algorithm is executed. Each agent remembers if it has already obtained the token: this is done by setting the flag *tree*. With this flag, the agent also remembers that it belongs to the "partial" spanning tree. The flag *summoning* is used by an agent to remember that the token has to be sent to a new agent that recently joined the spanning tree. The ports of each edge have a *parent* flag that we use to encode a parent-child relationship between the endpoint agents or an orientation of the edge, in the same way as we did in Theorem 3. The resulting oriented edges can point either in the direction of

```
1: Agent variables
2: token                                   ▷ true for the leader, false for non-leaders
3: tree := false
4: summoning := false
5:
6: Port variables
7: parent := false
8:
9: Transition function δ(a, p(a, b), b, p(b, a))
10: if a.token ∧ ¬a.tree then
11:     a.tree := true
12: if ¬a.tree ∧ ¬a.summoning ∧ b.tree then
13:     a.summoning := true
14:     p(b, a).parent := true
15: if a.summoning ∧ ¬b.summoning ∧ p(b, a).parent then
16:     b.summoning := true
17:     p(a, b).parent := true
18:     p(b, a).parent := false
19: if a.summoning ∧ b.token ∧ p(a, b).parent then
20:     a.token := true
21:     if ¬a.tree then
22:         a.tree := true
23:         a.summoning := false
24:     b.token := false
25:     b.summoning := false
```

Fig. 2: Token circulation protocol

the token (along the spanning tree) or toward an agent that is summoning the token.

The details of the algorithm are as follows. If an agent has the token and is not in the partial spanning tree (i.e., $a.token$ and $\neg a.tree$), it sets its own *tree* flag, thus becoming part of the spanning tree. This is an initialization operation that is performed only once in every execution.

If a sender a not in the spanning tree and not summoning (i.e., $\neg a.tree$ and $\neg a.summoning$) interacts with a receiver b in the spanning tree (i.e., $b.tree$), then it sets it own *summoning* flag and orients the edge $\{a, b\}$ toward b, setting the $p(b, a).parent$ flag.

If a summoning sender a (i.e., $a.summoning$) interacts with a non-summoning receiver b along an edge of the spanning tree that is oriented toward b (i.e., $\neg b.summoning$ and $p(b, a).parent$), then b becomes a summoner as well, and the orientation of the edge $\{a, b\}$ is reversed.

Finally, if a summoning sender a interacts with a receiver b possessing the token and the edge $\{a, b\}$ is oriented toward a, then a gets the token, while b loses it and ceases to be a summoner (in case it was a summoner). Additionally, if a is not in the spanning tree yet, it sets its own *tree* flag and stops being a summoner.

Theorem 4. *The protocol in Figure 2 solves the token circulation task in any network under the recurrent scheduler, provided that there is a unique leader. Moreover, the protocol is stabilizing and the edges with the parent flag set (on*

either port) eventually define a spanning tree of the network with edges oriented toward the token.

Proof. First we shall prove that the edges with the *parent* flag set always form a tree. Indeed, initially no *parent* flag is set. Then, the only line of the algorithm that creates an edge orientation is line 14 (note that lines 16 and 17 only flip an edge that is already oriented). In turn, line 14 is triggered only when a is not in the spanning tree and is not summoning. But as line 14 is triggered, a becomes summoning. Moreover, when a ceases to be summoning, it also becomes part of the spanning tree (lines 22 and 23). And once a is part of the spanning tree, it never leaves it (because the *tree* flag is never cleared in the algorithm). It follows that a is involved in the execution of line 14 at most once. This shows that the edges with the *parent* flag set never form cycles. Showing that they form a connected sub-network (containing the token) is also easy, because line 14 is executed only when a is a neighbor of an agent b with flag *tree* set, and the *tree* flag is set only by the agent that initially has the token (line 11) and by agents that have incident edges with the *parent* flag set (line 22). This proves that the edges with the *parent* flag set form a tree throughout the execution.

Also note that the agents with the flag *tree* set are vertices of this partial spanning tree: indeed, the flag is first set by the agent with the token (when the tree has no edges yet), and then only by agents that have incident edges with the *parent* flag set. Actually, the only agents that do not have the *tree* flag set and are incident to edges with the *parent* flag set are leaves of the partial spanning tree that have the *summoning* flag set.

Let us now prove the correctness of the protocol. We have to show that eventually all agents in the network set their *tree* flag. Since this only happens when they have the token, this would prove that the token reaches all agents. We will prove by induction that the number of agents with the *tree* flag set is bound to increase. Initially no agents have the *tree* flag set, and nothing happens until the agent with the token is involved in an interaction and sets its own *tree* flag. Note that this must happen sooner or later because the network is connected, there are at least two agents, and the scheduler is recurrent. Then, some agent a whose *tree* flag is not set will interact with an agent b whose *tree* flag is set, triggering lines 12–14. So a will become a summoner and the edge $\{a, b\}$ will be oriented toward b. Of course, this may happen to several different agents, not only to a. What will happen next is that lines 15–18 will be triggered and some edges of the partial spanning tree will start reversing. The idea is that each summoner tries to reach the agent with the token by reversing the edges along the path connecting to it. This path can be easily identified because all the non-summoners that are in the partial spanning tree point toward the token. When an edge is reversed, its non-summoner endpoint becomes a summoner, as well. This prevents lines 15–18 from being triggered more then once on the same agent, and therefore prevents different summoners from interfering with each other. Eventually, an edge reversal will reach the token. When this happens, there is a unique path in the partial spanning tree that is oriented from the token to a summoning leaf. Then lines 19–25 will be triggered, and the token

will follow the edges of such an oriented path, until it reaches the summoning leaf. The agents that lose the token will clear their *summoning* flags, but the one with the token will remain a summoner, to avoid triggering lines 15–18 and avoid creating forks in the path. When the summoning leaf obtains the token (the leaf is recognizable because its *tree* flag is not set), it sets its own *tree* flag and clears its own *summoning* flag.

This ends the proof of correctness. Note that the protocol is stabilizing because, as soon as all agents have set their *tree* flag, no new summoners appear and all edges stop reversing. □

4.2 Construction of Shortest-Path Spanning Trees

Next we show how to solve the task of constructing a spanning tree of the network under a k-bounded scheduler, again assuming that there is a unique leader. As a bonus, the distance of the leader from any agent along the spanning tree coincides with the distance over the whole network. Equivalently, this spanning tree is generated by a breadth-first traversal of the network starting at the leader.

Theorem 5. *For every $k \geq 1$, under the k-bounded scheduler there exists a terminating protocol that constructs a shortest-path spanning tree of any network, provided that there is a unique leader.*

Proof. The tree is created level by level, and the leader coordinates the construction: when a new level of the tree is completed, the leader will be notified and will broadcast a message on the partial spanning tree, ordering the construction of a new level. When a leaf receives a "new level" message, it expands identifying its children among the agents that have not been included in the tree, yet. Each of these agents will be part of the new level. Since the scheduler is k-bounded, a leaf is able to detect when it has seen all its neighbors, as in Theorem 3. Then, the leaf sends a "job done" message to its parent; in the message it also communicates if there is a new level or not. Each upper level collects all termination messages and forwards them, until they reach the leader. Initially, the leader is the only leaf of the tree and it will bootstrap the procedure creating the first level. Note that the leader knows that the task has been completed when it detects that no leaf has been able to add a new level to the spanning tree. At that point it broadcasts a "terminate" message along the spanning tree. □

4.3 Detection of Stability

Under the k-bounded scheduler, a unique leader can be used to convert any stabilizing protocol into a terminating one, in any network. A similar technique has been used in [17] in the context of detecting stability in message-passing systems. As byproduct, any protocol for the 2-bounded scheduler can be simulated in all k-bounded schedulers, for $k > 2$. First we give some crucial definitions.

Definitions. Let G be a network, and let δ be a transition function for G with agent states Q and port states U. Now let $Q' = Q_S \times Q$ and $U' = U_S \times U$, and let δ' be a transition function for the same network G with agent states Q' and port states U'. Let us refer to the sets Q_S and U_S as the *simulator work states* for agents and ports, while Q and U are the *simulated states*.

A *simulated transition* for δ' is a state transition in which some agents or ports change their simulated states as a result of an interaction according to δ (if, instead, only the simulator work states change, the transition is not considered a simulated one). Given an execution E of δ', its *simulated execution* is the execution of δ that is obtained from E by removing the non-simulated transitions and projecting the agents' and ports' states on Q and U.

We say that δ' under scheduler $\mathcal{S}'$ *simulates* δ under scheduler $\mathcal{S}$ if, for every execution of δ' corresponding to a schedule in $\mathcal{S}'$, its simulated execution is an execution of δ corresponding to a schedule in $\mathcal{S}$. If, additionally, δ is stabilizing under $\mathcal{S}$ and δ' is terminating under $\mathcal{S}'$, we say that δ' *detects the stability* of δ.

Theorem 6. *For every $k > 2$, given a stabilizing protocol δ, there is a protocol δ' that, under the k-bounded scheduler, detects the stability of δ under the* 2*-bounded scheduler, from any initial configuration with a unique leader.*

Proof. The protocol δ' has an *initialization phase* in which the leader builds a shortest-path spanning tree of the network, as in Theorem 5. Recall that the spanning tree construction is terminating, hence the agents can perform other tasks when they are finished. After the initialization phase, the protocol is structured in two alternating phases: a *reset phase* and a *simulation phase*. When the stability is detected, the leader starts the *termination phase*.

In the reset phase, all the flags used in the simulation by agents and ports are reset. This phase is performed level by level and is coordinated by the leader as in Theorem 5. Once again, note that the agents are able to tell when they have reset all their incident ports, because the scheduler is k-bounded.

Once all flags have been reset, the leader starts the simulation phase. In this phase, one simulated interaction between each ordered pair of neighboring agents is performed, starting from the leader and proceeding to the leaves, following the levels of the spanning tree. Each edge port has a *simulation flag*, which tells if the edge has already been part of a simulated interaction in the direction corresponding to the port. This flag is reset during each reset phase. The simulation at level ℓ proceeds as follows: an agent receiving the order to simulate starts scanning each incident port (whenever the scheduler generates the corresponding interaction) and, if its simulation flag is not set, both endpoints of that edge perform a simulated interaction according to δ and set the port's simulation flag. When all its incident ports have the simulation flag set (which can be verified because the scheduler is k-bounded), the agent sets its own *complete flag* and notifies the leader. The leader waits until it detects that each agent at level ℓ has set the complete flag: this can be done with a convergecast. Then the leader issues another order to simulate, which reaches level $\ell + 1$. The simulation phase ends once the lowest level of the tree has finished simulating. Thanks to the complete flag, this phase can be performed in constant space.

During the simulation phase, the agents also perform a "local stability" check on each edge. An edge $\{a, b\}$ is *locally stable* if no (infinite) schedule consisting only of the interactions (a, b) and (b, a) ever causes the simulated state of a or b to change according to δ. Note that the stability of an edge can be verified by its endpoints in a single interaction executing δ'. Each agent has an *unstable flag* that is cleared during the reset phase and is set whenever the agent either changes its simulated state or detects that an incident edge is not locally stable. Then, during the convergecast, agents also communicate the state of their unstable flag to the leader. When the simulation phase is over, the leader knows if the whole network is locally stable. If it is not, it starts the next reset phase; otherwise, it proceeds with the termination phase. The termination phase is simply a broadcast over the spanning tree that orders all agents to get a terminal state.

The correctness of the simulation follows from the fact that, at every phase, each simulation flag is first cleared and then set. This means that all possible simulated interactions occur in some order at each simulation phase. So, the resulting simulated schedule is a sequence of permutations of all the possible interactions in the network. Each permutation contains each interaction exactly once. Therefore, between two occurrences of the same interaction within two consecutive permutations, no other interaction occurs more than twice. In other words, the simulated schedule is 2-bounded.

Let us now show that the stability of δ is correctly detected and that δ' correctly terminates. Of course, when the simulated execution of δ stabilizes, all edges are locally stable and no agents change simulated states, and this is detected by the leader, which then correctly executes the termination phase. We have to prove that the leader cannot start the termination phase "by accident" before the execution of δ has actually stabilized. Equivalently, we have to prove that, if all edges are locally stable at some point during the simulation phase, then the simulated execution of δ has indeed reached a stable simulated configuration. Here the key observation is that, by the way the simulator works, the simulated states of agents and ports change only according to δ. So, if all edges pass the local stability test at some point in the simulation phase (and no agents change their simulated states), it does not matter in what order they are checked, and when. Indeed, in the next simulation phase, the simulated states of the agents will still be the same, and therefore all edges will still be locally stable. □

If δ' is executed under the 1-bounded scheduler, the simulated execution obtained in Theorem 6 corresponds to a 1-bounded schedule, as well.

Corollary 1. *Given a stabilizing protocol δ, there is a protocol δ' that, under the 1-bounded scheduler, detects the stability of δ under the 1-bounded scheduler, from any initial configuration with a unique leader.*

Proof. Recall from Theorem 6 that the simulated schedule generated by δ' consists of a sequence of permutations of all the possible interactions. Now, if δ' is executed under a 1-bounded scheduler, the schedule will actually be periodic, and so will be the resulting simulated schedule. Therefore, the simulated schedule is a repetition of the same permutation of interactions, which implies that it is 1-bounded, as well. □

References

1. D. Alistarh, R. Gelashvili, and M. Vojnovic. "Fast and exact majority in population protocols", *34th Annual ACM Symposium on Principles of Distributed Computing, PODC*, pp. 47–56, 2015.
2. D. Angluin, J. Aspnes, M. Chan, M. J. Fischer, H. Jiang, R. Peralta. "Stably computable properties of network graphs", *1st IEEE International Conference on Distributed Computing in Sensor Systems, DCOSS*, pp. 63–74, 2005.
3. D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. "Computation in networks of passively mobile finite-state sensors", *Distributed Computing*, vol. 18(4), pp. 235–253, 2006.
4. D. Angluin, J. Aspnes, M. J. Fischer, H. Jiang. "Self-stabilizing population protocols", *ACM Transactions on Autonomous and Adaptive Systems*, vol. 3(4), pp. 13:1–13:28, 2008.
5. D. Angluin, J. Aspnes, and D. Eisenstat. "Stably computable predicates are semilinear", *25th Annual ACM Symposium on Principles of Distributed Computing, PODC*, pp. 292–299, 2006.
6. J. Beauquier, P. Blanchard, and J. Burman. "Self-stabilizing leader election in population protocols over arbitrary communication graphs", *13th International Conference on Principles of Distributed Systems, OPODIS*, pp. 38–52, 2013.
7. J. Beauquier, J. Burman, S. Clavière, and D. Sohier. "Space-optimal counting in population protocols", *29th International Symposium on Distributed Computing, DISC*, pp. 631–649, 2015.
8. J. Beauquier, J. Burman, and S. Kutten. "A self-stabilizing transformer for population protocols with covering", *Theoretical Computer Science*, vol. 412(33), pp. 4247–4259, 2011.
9. S. Cai, T. Izumi, and K. Wada. "How to prove impossibility under global fairness: on space complexity of self-stabilizing leader election on a population protocol model", *Theory of Computing Systems*, vol. 50(3), pp. 433–445, 2012.
10. D. Canepa, and M. Gradinariu Potop-Butucaru. "Stabilizing leader election in population protocols", *Research Report*, inria-00166632, 2007.
11. D. Canepa, and M. Gradinariu Potop-Butucaru. "Self-stabilizing tiny interaction protocols", *3rd International Workshop on Reliability, Availability, and Security, WRAS*, pp. 1–6, 2010.
12. I. Chatzigiannakis, O. Michail, and P. G. Spirakis. "Stably decidable graph languages by mediated population protocols", *12th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS*, pp. 252–266, 2010.
13. I. Chatzigiannakis, O. Michail, and P. G. Spirakis. "Mediated population protocols", *Theoretical Computer Science*, vol. 412(22), pp. 2434–2450, 2011.
14. G. A. Di Luna, P. Flocchini, T. Izumi, T. Izumi, N. Santoro, and G. Viglietta. "Population protocols with faulty interactions: the impact of a leader", *arXiv:1611.06864 [cs.DC]*, 2016.
15. M. Fischer and H. Jiang. "Self-stabilizing leader election in networks of finite-state anonymous agents", *10th International Conference on Principles of Distributed Systems, OPODIS*, pp. 395–409, 2006.
16. R. Mizoguchi, O. Hirotaka, S. Kijima, M. Yamashita. "On space complexity of self-stabilizing leader election in mediated population protocol", *Distributed Computing*, vol. 25(6), pp. 451–460, 2012.
17. N. Shavit and N. Francez. "A new approach to detection of locally indicative stability", *13th International Colloquium on Automata, Languages and Programming*, pp. 344–358, 1986.