Self-stabilising Priority-Based Multi-Leader Election and Network Partitioning

Danilo Pianini
University of Bologna
Cesena, Italy
danilo.pianini@unibo.it

Roberto Casadei
University of Bologna
Cesena, Italy
roby.casadei@unibo.it

Mirko Viroli
University of Bologna
Cesena, Italy
mirko.viroli@unibo.it

***Abstract*—A common task in situated distributed systems is the self-organising election of leaders. These leaders can be devices or software agents appointed, for instance, to coordinate the activities of other agents or processes. In this work, we focus on the multi-leader election problem in networks of asynchronous message-passing devices, which are a common model in self-organisation approaches like aggregate computing. Specifically, we introduce a novel algorithm for space- and priority-based leader election and compare it with the state of the art. We call the algorithm Bounded Election since it leverages bounding (i.e. minimisation or maximisation) of candidacy messages to drop or promote candidate leaders and ensure stabilisation. The proposed algorithm is formally proven to be self-stabilising, allows for leader prioritisation, and performs on-the-fly network partitioning (namely, as a side effect of the leader election process, the areas regulated by the leaders are also established). Also, we experimentally compare its performance together with the state of the art of leader election in aggregate computing in a variety of synthetic scenarios, showing benefits in terms of convergence time and resilience.**

***Index Terms*—leader election, self-stabilisation, self-organisation, network partitioning, aggregate computing**

I. Introduction

In the *autonomic computing* vision [1], the complexity (e.g., in terms of integration [2] and maintenance) of recent and forthcoming distributed systems is urging engineers to design systems capable of autonomously managing themselves, i.e., to endow them with *self-** properties and capabilities [3]. One such property is *self-organisation* [4], [5], by which a system spontaneously seeks and maintains order or structure without external control. The benefits include system-level robustness and adaptiveness to environmental change, and scalability by means of decentralisation. This property can be engineered into computing systems, through self-organisation mechanisms or algorithms [6] which typically leverage loops of local computation and interaction to solve specific problems.

Many problems in distributed systems can be seen as self-organisation problems [7] when considered in dynamic environments with ongoing system operation. Relevant problems include reaching consensus, taking snapshots, partitioning a system into regions, and electing leaders [8]–[10]. A notion connecting distributed algorithms to self-organisation is *self-stabilisation* [11], introduced by Dijkstra [12] to characterise algorithms that converge to correct states in finite time from any arbitrary initial state, hence in spite of transitory changes (i.e., resiliently).

In this work, we consider the *leader election* problem. There are several research contributions to leader election, considering the problem under a wide spectrum formulations, assumptions, and goals. Indeed, it is possible to distinguish between algorithms for single [13] and multi-leader election [14]; algorithms designed for specific network topologies (e.g., rings [15], hierarchies [16], or ad-hoc networks [17]); algorithms for anonymous networks [18]; deterministic vs. probabilistic algorithms [19]; algorithms optimising for time, space, or message complexity; algorithms for specific distributed protocols (e.g., population protocols [20]), etc.

Specifically, our paper focuses on *self-stabilising multi-leader election* in *aggregate computing* systems [21], namely on round-based networks of asynchronous, message passing agents. In this setting, a space-based version of the algorithm is also called ***S**parse-choice* (or simply **S** [22]), since leader election could also be seen as a way to sparsely pick devices to "cover" a system. A good sparse-choice primitive is paramount, as it is frequently used in self-organising applications to break symmetry, assign responsibilities to important nodes, perform sparse sampling of a phenomenon [23], or partition the network. Indeed, network partitioning can be achieved by denoting a partition (or area) through the set of a leader and its followers. Areas can then be used to scope activity.

In this paper, we propose an algorithm that is both self-stabilising and solves the multi-leader election problem, doing so by using solely one propagation step. Moreover, it also considers priorities (in addition to distance) to determine leaders, induces a network partitioning with no additional effort (e.g., without the need of propagating information from the leaders), and is shown to perform better, in terms of convergence time and resilience, than the state of the art.

The content of the paper is organised as follows. Section II provides background. Section III covers motivation and related work. Section IV provides the contribution, presenting the novel, proposed algorithm and proving that it is self-

The authors would like to thank Ferruccio Damiani, who suggested a successful path for proving the self-stabilisation of Bounded Election.

This work has been supported by the MIUR PRIN 2017 Project "Fluidware" (N. 2017KRC7KT) and the EU/MUR FSE REACT-EU PON R&I 2014-2020.

stabilising. Section V provides experimental evaluation in simulation settings. Section VI summarises results and discusses future work.

II. Background

In this paper, we focus on self-stabilising multi-leader election in aggregate computing settings, namely in networks of message-passing agents operating in asynchronous rounds. This section provides corresponding background on leader election (Section II-A) and the aggregate computing model (Section II-B).

A. Leader election

Leader election (LE) is one fundamental problem in distributed as well as self-organising systems [7], [11], [29]. Consider an undirected graph $G = (V, E)$, where V is the set of nodes in the network and E is the set of edges representing communication channels between nodes. The LE problem consists of determining one node in the system as the *leader*, by means of a distributed protocol or algorithm involving multiple rounds of local computations and communication. Notice that this can also be seen as a particular kind of graph labelling and graph colouring; however, LE is typically subject to different constraints (e.g., other than assigning different colours to adjacent edges or vertices) and generally considers dynamic scenarios with partial observability. The LE problem can be formulated in a weak or strong manner [30]. In the *weak formulation*, every node outputs a Boolean value indicating whether it is a leader or not. In the *strong formulation*, every node must know the leader, or be able to locate it—so, the output could be the identifier of the leader (in non-anonymous networks), or a path leading to it (in anonymous networks). A variant of LE is *multi-leader election (MLE)* [14], [31], where the goal is to elect multiple leaders, e.g. for redundancy.

Several LE algorithms have been devised over time [32], with differences related to the problem formulation, the underlying assumptions, the performance trade-offs (in terms of time, space, and message complexity), as well as target properties and guarantees. Common underlying assumptions include the network topology, the distributed protocol defining legitimate actions, and the scheduling assumptions (e.g., in terms of a/synchronism and fairness) [33]. Regarding the target properties, an important one is *self-stabilisation* [11], [12]: informally, an algorithm is self-stabilising if, regardless of the initial configuration, it converges in finite time to a correct configuration. In other words, the property means an algorithm can automatically recover from transient faults—which is a desirable feature, especially in large-scale distributed systems like those found in the Internet of Things (IoT). Therefore, in this paper we focus on the *self-stabilising* (multi-)leader election problem [13], [34], in which the protocol must be able to converge to a correct leader selection in finite time, from any initial system configuration. The other model assumptions (topology, protocol, scheduling) are elicited in the following section.

B. Model and assumptions

We consider a round-based distributed protocol on dynamic networks of asynchronous message-passing agents. This is, indeed, the distributed protocol assumed by *aggregate computing* [35], a paradigm for self-organizing systems programming that has been subject of intense research in the last decade [21]. For simplicity, we would refer to the model described in this section as the *aggregate computing model*.

By a structural point of view, we still consider an *undirected graph*, to model a network of devices (or nodes). The devices connected to a device through an edge are called its *neighbours*. A device can only directly communicate with its neighbours. The neighbouring relationship is symmetric and reflexive. We assume each node is given a *unique identifier (UID)*. A node can be equipped with *sensors* and *actuators* to interact with the environment, for instance, a node can query a sensor to get its own UID.

The distributed protocol consists of an asynchronous execution of *rounds*. Every node alternates sleeping periods with rounds (in general, not synchronised with other devices) comprising the following steps.

1) *Context acquisition.* The node gets data from sensors and collects messages from neighbours. We assume only the last message from any neighbour is retained, and that messages have a configurable expiration time.
2) *Computation.* The node runs a computation considering its up-to-date context as input. We assume all nodes share the same program, and such program is scheduled in an asynchronous and weakly fair way (where a continuously enabled node will eventually be scheduled) [33]. The output of a computation includes a single *result value* (e.g., the UID of the local leader), which could also be structured data, as well as the message to be sent to neighbours (also called an *export*).
3) *(Inter)Action.* The node sends the same export message to all its neighbours (broadcast) and may run actuations (e.g., a robot may use a part of the result value to command wheels for movement). For message passing, we assume *at-most-once* delivery and *message ordering per sender-receiver pair*.

A comprehensive survey of aggregate computing research and details about its programming model are available in [21]. In this paper, we consider the notion of *self-stabilisation*, formalised for aggregate computing systems in [36].

III. Motivation and Related Work

In this section, we provide motivation for this work in terms of two desirable features of multi-leader election and the connection to network partitioning (Section III-A), as well as related work about leader election in aggregate computing (Section III-B).

A. Spatiality, priority-sensitivity, and network partitioning

In this section, we motivate two additional features or constraints that may apply to multi-leader election, that are desirable characteristics in general and fundamental features

of the novel algorithm proposed in this paper: *spatiality* and *priority-sensitivity*.

Indeed, we focus on *space-based* multi-leader election, where the election of leaders has to generally take into account distances between nodes in metric spaces. This is a relevant problem since it is intimately related to the problem of *partitioning* a situated network (cf. IoT systems) into areas or regions. The creation and adaptation of regions is a significant self-organising mechanism. Indeed, by dynamically partitioning a system into regions it is possible to divide a large problem into smaller problems (cf. *divide-et-impera*) as well as to regulate the desired level of decentralisation in a system. The combination of sparse-choice, dynamic partitioning, and leader-followers feedback loops is at the basis of *Self-organising Coordination Regions* (SCR) [10], a pattern for supporting distributed situated recognition and action in dynamic environments, which is a documented, proved solution to recurrent problems in scenarios like environmental monitoring, situated problem solving, and computational ecosystems.

We also focus on *priority-based* multi-leader election, where a notion of priority reflecting the strength or suitability of a leader candidacy is considered in the election process. Indeed, since leaders usually have to coordinate the work of other nodes, take decisions based on collected data, and so on, they often need to be rather powerful computing nodes (e.g., powered fog servers rather than resource-constrained devices) or be chosen among high-degree nodes (e.g., hubs). So, considering priorities is reasonable, also because it provides a natural way to locally break symmetry or to structure competitive processes.

B. Related work

Several leader election algorithms have been proposed in the literature. However, given our assumed system model, in this paper we do not compare with works having substantially different assumptions and goals, e.g., works tailored to single-leader election [13], specific topologies (e.g., rings [15]), anonymous networks [18], entirely different distributed protocols (e.g., population protocols [20]).

For the considered aggregate computing model, there exist two leader election algorithms proposed in literature, to the best of our knowledge. The first one, recently proposed by Mo et al. [13], is self-stabilising, as it leverages a combination of aggregate computing blocks demonstrated to be self-stabilising in a way that does not disrupt self-stabilisation. However, it relies on network diameter estimations that require the construction of a feedback loop based on collection into and propagation from candidate leaders (de-facto, leveraging a known pattern [10]); and focuses on the problem of a single-leader election. Although its more recent extension [26] can be tuned to produce multiple leaders, the reliance on diameter estimations (and related feedback loop) remains intact. Our focus, instead, is on multi-leader election, to be performed with a single propagation.

The second one is the *Sparse spatial choice (S)* algorithm, described in [22], which performs a multi-leader election process in which leaders are elected at a certain distance between one another. Although frequently used and available in existing libraries [24], it has never been formally proven to be self-stabilising. Unproven self-stabilisation is a limiting factor for widespread usage, as it essentially breaks self-stabilisation guarantees that would otherwise be preserved by composition of self-stabilising algorithms [36]. Additionally, self-stabilisation says little about the actual level of reactivity to change (i.e., time-to-convergence); so, it is important to also consider how quickly stabilisation occurs—aspect that is examined in Section V.

IV. Self-stabilising Leader Election Algorithm

In this section, we present our main contribution: a novel priority-based multi-leader election algorithm for the aggregate computing model. We present a starting-point naive algorithm that will work as baseline (Section IV-A), the proposed algorithm (Section IV-B) and its implementation, and finally prove it is self-stabilising (Section IV-C).

A. Idea and trivial implementation

At the root of the idea behind the proposed algorithm is the observation that, in many systems, although each node is potentially a valid leader candidate, different nodes should have different priorities, as in many circumstances some nodes are better-suited leaders than others: for instance, in an IoT system with edge servers, the latter should be preferentially picked over "thin" devices to sustain data-intensive operations; in non-ad-hoc networks (for instance, scale-free networks [37]), it may be convenient to select leaders located in "central" points of the network (for instance, to minimize the communication lag); in systems where devices are battery-powered, leaders with more battery would improve the overall reliability, and so forth. Thus, devices may compete based on their perception of how good they would be as leader. Assuming that there is a way to generate a unique identifier for each device (to break the symmetry in case multiple devices are exactly as good potential leaders), we would obtain an order of preference. This ordering could be then exploited as follows:

1) find the "best" leader among all devices (e.g., by gossiping the information on the "goodness" of each leader);
2) make it the leader of all devices "close enough" to it;
3) repeat the procedure recursively, excluding all devices that already have a leader.

This algorithm can be expressed quite easily, for instance, with the Protelis aggregate programming language [38][1]:

```
1 def leaderElection(id, priority, radius) {
2   let best = gossip([priority, id], min).get(1)
3   if (distanceTo(best == id) <= radius) { best }
4   else { leaderElection(id, priority, radius) }
5 }
```

For the reader interested in the semantic details of aggregate programs, `leaderElection` works as follows: first, `gossip` selects the `best` leader of the current (sub-)network by

[1]In Protelis sources, language keywords are shown in bold purple and library functions are shown in blue.

minimising over $(priority, id)$ tuples, propagating the result to neighbours, and projecting on the second element (at position 1, assuming 0-indexing of tuples); then, `distanceTo` computes for each node the distance to that leader, e.g. through a gradient (cf. Section III-A) considering as source the node whose `id` equals `best`; finally, the **`if`**/**`else`** branching construct splits the (sub-)network in two parts, one of devices closer to `best` than `radius`, whom will output `best` as their leader (return value of `leaderElection`), and the other part where the same `leaderElection` logic is recursively applied.

Recall from Section II-B that the program has to be evaluated by all the devices, in asynchronous rounds, and that neighbourhood broadcasts (logically, as optimisations are possible) are sent at the end of each round. The actual data exchanged with neighbours depends on the program and is specified in aggregate computing languages through appropriate constructs (such as **`share`**, see next) and their occurrences within library functions (such as `gossip` and `distanceTo`).

This trivial implementation has two important drawbacks, though. First, gossip algorithms are well-known to be monotonic: they quickly adjust in one direction, but never accept updates in the opposite direction. In our example, if the leader selected first is no longer a good candidate (e.g., because it is running out of battery, or because it moved in a peripheral region of the network, or even because it left the system), it would still be advertised as the globally best leader. In other words, *this solution is not self-stabilising*. This behaviour can be worked around by restarting the gossip or by adopting replication strategies [39], [40], requiring additional complexity in computation and communication. Second, to converge, the algorithm requires a gradient [27], [28] (although "hidden" behind a library function call to `distanceTo`) to stabilise across the network *for each recursive call*. This makes the algorithm potentially slow-converging and fragile, as changes on the first recursive calls (due, e.g., to nodes moving apart) would require a re-computation across all the parts of the network whose final value is computed more in-depth.

B. Algorithm

A better solution to the problem, that overcomes the aforementioned issues, can be devised by rethinking the recursive approach into a parallel solution: every device always competes with its neighbours and, upon loss, supports the best leader known to it. Informally, the algorithm can be explained as follows.

1) Every device produces its opinion on the best leader in form of a triple including the following information:
   a) the leader's priority;
   b) the distance between this device and the best leader it knows of;
   c) the id of such leader.

   Of course, at the beginning, every device will know no candidate leader but itself.
2) Every device observes the best candidates in its neighbourhood, discarding those that propose the device itself as leader and those that are too far away.
3) The new best leader is the best between itself and the best (if any) of those acquired from the neighbourhood.
4) The best leader can be communicated to the neighbours.

If this procedure is executed asynchronously by every device in the network, the emergent result is a partitioning of the network, with every device assigned to the best leader available within the allowed range from its location.

We present in Algorithm 1 a reference implementation in form of pseudocode. The algorithm can be expressed succinctly in the framework of aggregate computing, hence, we also provide actionable implementations of Bounded Election (Bounded Election) written in Protelis [38]; and in Scala through the ScaFi [41] internal domain-specific language at the companion artefact [42].

Algorithm 1 Bounded Election (single round)

Require: id, $radius$, $strength$, $metric$
Ensure: $leader$ is the local leader

```
local ← (val : −strength, dist : 0, lead : id)
others ← receive()   // set of neighboring leader triplets
valid ← ∅   // will collect valid neighboring candidacies
for each proposal in others do
  d ← distanceBetween(id, proposal.lead, metric)
  distance ← proposal.dist + d        // actual distance
  if distance ≤ radius ∧ id ≠ proposal.lead then
    proposal.dist ← distance     // update the distance
    valid ← valid ∪ {proposal}        // add to valid set
  end if
end for
result ← min(local, min(valid))          // select the best
send(result)          // propagate the choice to neighbors
leader ← result.lead                 // update the local leader
```

We notice that the algorithm is agnostic with respect to the distance $metric$. Also, since the maximum $radius$ of each area is provided as parameter, this algorithm supports single-leader election by passing a finite (over)estimate of the network diameter (measured according to the selected metric, and not as hop-count), and thus guaranteeing that the "highest priority" leader is known to any device in the network. Depending on the implementation of the diameter calculation, this strategy makes the algorithm very similar to [26] with $K > 1$, where the "C block" is responsible for collecting diameter estimations from the network.

C. Self-stabilisation of Bounded Election

1) Strategy: To prove that Bounded Election is self-stabilising, we show that it is an instance of the *minimising* `share` pattern, which is proved in [43], following the framework of [36], that it is self-stabilising. We do so by rewriting the *minimising* `share`, preserving self-stabilisation, until we obtain an algorithm that is functionally equivalent to to Algorithm 1.

2) The minimising `share` *pattern:* In [43], it is proved that an expression of the following form is self-stabilising:

```
share(x <- e) { fR(minHoodLoc(fMP(x, s̄), e), x)}
```

where the meaning of the terms is summarised in the remainder of this section (please, see Section 5.2 in [36] and, in particular, Section 4.7 and Figure 7 in [43]).

a) share block: `share(x <- e){ body }` is a construct meant to model evolution in time (through computation of new information locally) and space (through information exchange with other devices) [43]. The declared variable `x`, assigned on the first round to the result of the evaluation of expression `e`, collects the evaluations of the overall expression in neighbour devices (including the device itself), working initially as the `receive` function of Algorithm 1. The expression `body` is then evaluated, and its result is used both as next local value (for the round to come) and as content of the message to be sent to neighbours, thus working as the `send` function of Algorithm 1.

b) raising function: `fR(x, prev)` is a raising function, with respect to partial orders, of `x` and `prev`, where `prev` is the value of `x` at the previous round. In other words, the function always returns a value greater or equal than the arguments: `fR(x, prev)` $\geq max($`x, prev`$)$.

c) monotonic progressive function: `fMP` is a monotonic non-decreasing progressive function of `x`, with the following implications:

1) $\forall$ `x1` $\leq$ `x2` $\Rightarrow$ `fMP(x1)` $\leq$ `fMP(x2)`, and
2) $\forall$ `x` $\Rightarrow$ `fMP(x,...)` $\geq$ `x`.

`fMP` can take additional arguments `s̄`, as far as they are self-stabilising expressions that do not contain the **share**-bounded variable `x`.

d) selection of the minimum: `minHoodLoc(e, loc)` selects the minimum among the neighbours' values of expression `e` and the current device's local value `loc`.

Theorem 1 (Bounded Election is self-stabilising)**.**

Proof. We note that our initial expression is the self-candidacy, and that the identity function on the first parameter is a trivially valid raising function (see Example 5.5 in [36]), we can thus rewrite:

```
loc ← (val : −strength, dist : 0, lead : id)
share(x <- loc) { minHoodLoc(fMP(x, s̄), loc)}
```

To improve readability, we then extract `fMP` and store its value in a local variable v:

```
loc ← (val : −strength, dist : 0, lead : id)
share(x <- loc) {
  v ←fMP(x, s̄)
  minHoodLoc(v, loc)
}
```

Then, the increment in distance and filtering operation (the for-each cycle in Algorithm 1) is a valid replacement for `fMP`, as any non-discarded element (hence including the minimum in the set) gets larger, as its `dist` increases while other values remain unchanged. In order to preserve the purely functional structure of the proof, we rewrite the for-each operation as a functional `flatMap` (we also apply the De Morgan's law to the boolean condition for compactness):

```
loc ← (val : −strength, dist : 0, lead : id)
share(x <- loc) {
  v ← x.flatMap { c ->
    d ← c.dist + distanceBetween(id, c.lead, metric)
    if (d > radius ∨ id = c.lead) ∅
    else {(val : c.val, dist : d, lead : c.lead)}
  }
  minHoodLoc(v, loc)
}
```

Finally, the selection of the minimum value between the local candidacy and the minimum found in the neighbourhood reproduces the behaviour of `minHoodLoc`:

```
loc ← (val : −strength, dist : 0, lead : id)
share(x <- loc) {
  v ← x.flatMap { c ->
    d ← c.dist + distanceBetween(id, c.lead, metric)
    if (d > radius ∨ id = c.lead) ∅
    else {(val : c.val, dist : d, lead : c.lead)}
  }
  min(min(v), loc)
}
```

Since the **share** operator replaces the reception and sending of messages, this algorithm is equivalent to Algorithm 1, and since it is adhering to the minimising share pattern, Bounded Election is in turn self-stabilising [36], [43]. □

Summarising, our proof consisted of the following steps:

1) observation that the minimising `share` pattern is self-stabilising;
2) observation that the **share**`(x <- e){ body }` reproduces the behaviour of a `receive`-compute-`send` block;
3) observation that the identity function is a valid raising function;
4) substitution of `fR` with the identity;
5) observation that the for-each block in Algorithm 1, when rewritten in functional form using a `flatMap`, produces a monotonic progressive function;
6) substitution of `fMP` with the `flatMap` block
7) observation that minimising over the results of the flatMap and the local candidacy has the same behaviour of `minHoodLoc`;
8) substitution of `minHoodLoc` with $min(min(v), loc)$;
9) observation that the produced algorithm preserves self-stabilisation and is functionally equivalent to Algorithm 1; which is in turn self-stabilising.

D. Resource consumption estimation

1) Runtime per node: The algorithm requires, for each round, the evaluation of one message per neighbour, as the operations of distance increase, self-candidacy filtering, and minimisation can be compacted in a single iteration. Thus, the algorithm complexity in time grows linearly with the number of neighbours, the denser the network, the higher the cost.

2) Payload size: The payload is composed of two parts: the triple representing a local candidacy and the information required to estimate the distance from each neighbour (if any information is needed). Since the algorithm abstracts away the $metric$ as an input parameter, several methods can be used to estimate it, generating different metric spaces. In case of GPS-equipped devices, the local position could be used to

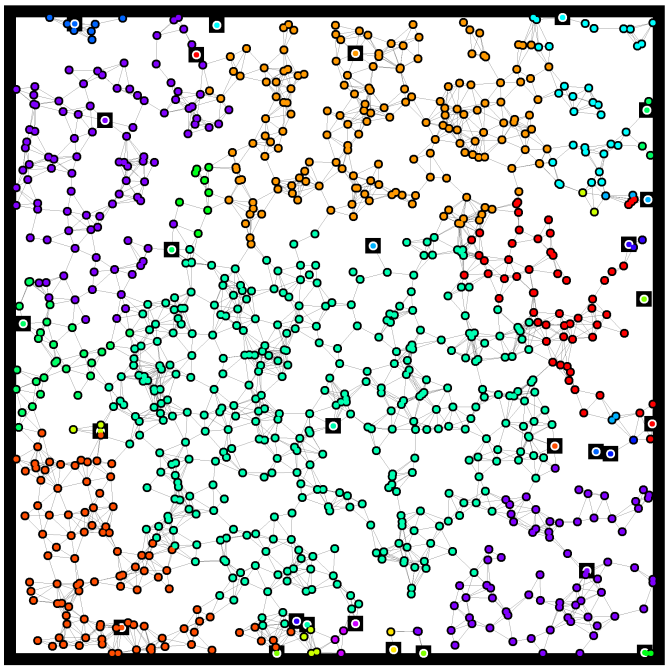

Fig. 1. Snapshot of the randomly moving scenario. Devices are depicted as coloured dots. Devices with the same colour belong to the same partition (hence, share the same leader). Leaders are identified, within a partition, by a thicker border. One thousand devices are deployed in a square arena, where they are free to move. Devices sufficiently close-by are considered connected (ad-hoc network), but the network is sparse and may part.

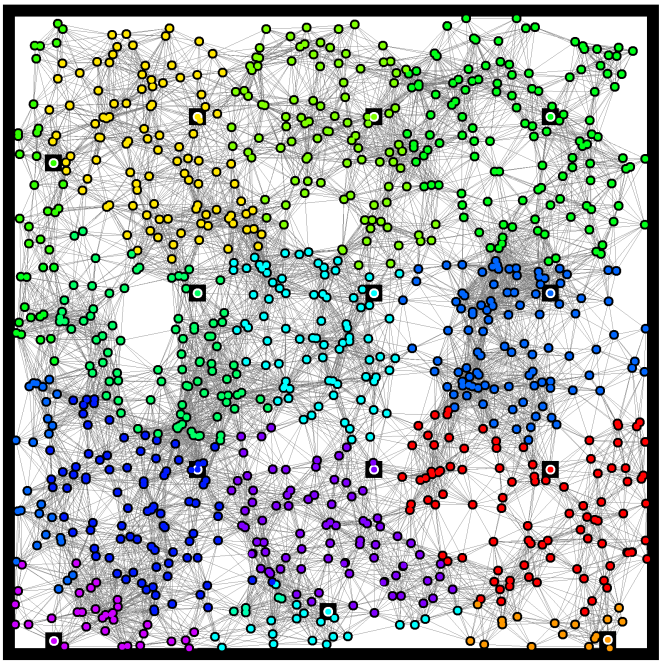

Fig. 2. Snapshot of the edge scenario. Devices are depicted as coloured dots. Devices with the same colour belong to the same partition (hence, share the same leader). Leaders are identified, within a partition, by a thicker border. 975 devices are deployed in a square arena, where they are free to move. Inside the arena are also 25 "thick" wall-powered edge devices, that do not move and should be preferentially selected as leaders. Devices sufficiently close-by are considered connected (ad-hoc network); the network is dense enough that segmentation occurs rarely.

compute the distance between devices, and thus need to be communicated to neighbours. In case of an estimation coming from a local source (e.g., wireless signal attenuation, round-trip-time at the previous round, or other information that can be used as a proxy metric provided by the underlying network protocol) the cost of this part of the payload goes to zero. In case of custom metric spaces, the size depends on the specific instance of the metric function; for instance, if nodes are considered close if the value of some sensor reading is similar, a reasonable metric could be the variance between the local and neighbour value, which requires the sensor reading to be included in the payload.

V. Evaluation

The self-stabilisation proof of Section IV-C provides strong guarantees on the eventual behaviour of Bounded Election, but it does not provide information on its dynamics (namely, on the transition between stable states); that, however, in dynamic systems may be as (and even more) important than the eventual guarantee. In fact, some systems may even never reach a "steady state" that lasts enough for the self-stabilising algorithm to be allowed to reach its final configuration: in these cases, the transitional behaviour rises to prominence, and "good" algorithms are expected to continuously drive the system towards the would-be stable configuration.

A. Scenarios

To gather information on the dynamic behaviour of the proposed algorithm, we resort to the simulation of three paradigmatic scenarios.

In the first scenario, depicted in Figure 3, we set up a random *scale-free network* of one thousand devices through preferential attachment [37]. We let the devices compute their partition, but at fixed time intervals we modify their priority, cyclically switching from the node degree (the number of neighbours) to the device's unique identifier to a randomly generated value. The random value is generated once and kept constant until the workmode is switched. This scenario is meant to investigate how the algorithms adapt to sudden changes in the way the network symmetry is broken.

In the second scenario, one thousand devices are located within a square arena and allowed to move following Lévy Walks [44][2]. They communicate with neighbours located in their proximity (within a predefined radius); the communication range has been selected to make sure that the network segments frequently in sub-networks, forcing a continuous re-adaptation of the election and partitioning process: the goal of this scenario is to investigate a situation in which there

[2]We selected Lévy Walks as movement pattern as they reasonably approximate the walking pattern of human beings [45].

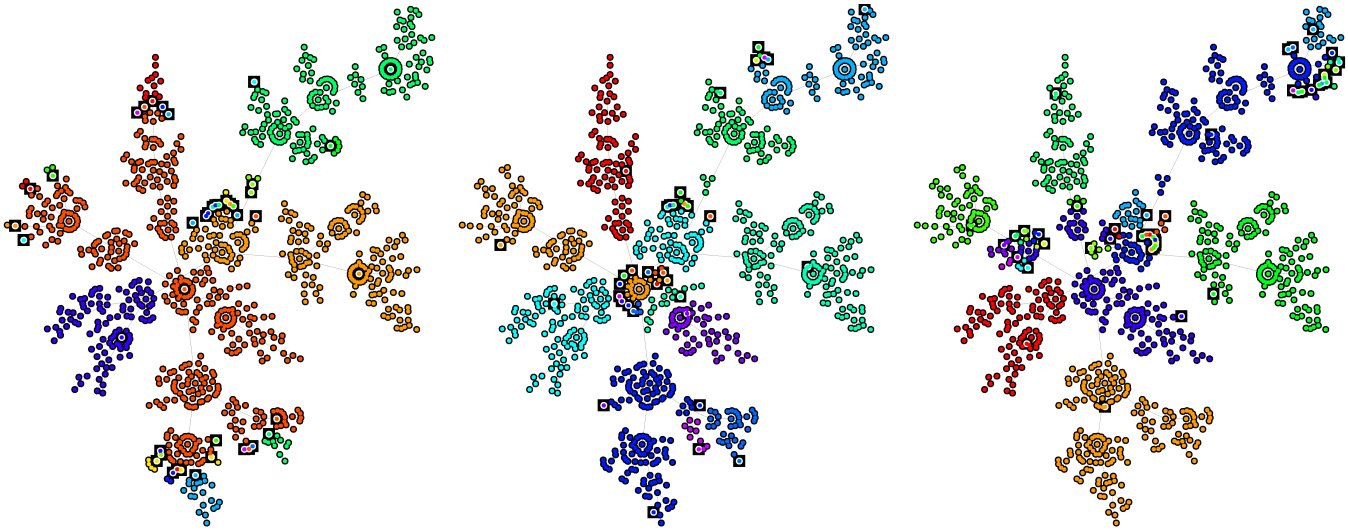

Fig. 3. Subsequent snapshots of the simulation of the scale-free network scenario. Devices are depicted as coloured dots. Devices with the same colour belong to the same partition (hence, share the same leader). Leaders are identified, within a partition, by a thicker border. Devices' priorities switch from using the node degree, favouring more "central" devices as leaders (left), to using their device unique identifier (centre), to using a selection based on a pseudo-randomly generated number (right). (Full resolution images are provided at the accompanying repository.)

is a continuous perturbation, and stability is never reached. Figure 1 shows a simulation snapshot of the scenario.

The last scenario, whose snapshot is presented in Figure 2, simulates a mixed IoT/edge situation. 975 devices are deployed inside the same arena of the previous scenario and are free to move within its boundaries with the same movement rule; however, this time 25 devices are marked as edge servers, they are deployed in a 5x5 grid inside the arena, they cannot move, and they are the preferred leaders for their area. This scenario is designed to investigate the behaviour of the network when there are heterogeneous devices, and some are better suited to work as leaders. In each scenario, devices compute rounds asynchronously with a frequency of 1Hz. We do not model network errors and packet losses in these experiments.

B. Algorithms

We compare three algorithms:

1) Bounded Election;
2) the trivial recursive version of the leader election presented in Section IV-A; and
3) the current state of the art, represented by the `S` block as implemented in the Protelis-lang library [24].

For the trivial recursive version, in the scale-free scenario we adopt an arguably unfair way to detect changes: we do reset the gossip process right at the time the leader strength changes. In a nominal situation, that would be hardly possible: the algorithm would have needed to run overlapping replicates of the gossip process [39], with a much higher resource consumption and a slower time to adjust to non-monotonic changes.

C. Metrics

We are interested in stabilisation times as, in cases when the execution of other algorithms depend on the network partitioning (as in the case of the SCR pattern [10]), then even a small improvement on the stability or convergence time can lead to much faster adaptation. This potentially translates to fewer rounds required to stabilise and lower power consumption, especially if advanced scheduling strategies (reactive or partially reactive, as in [46]) are used. One simple metric would have been the time required for the whole network to stabilise, but such a metric can not tell apart "almost stabilised" scenarios from completely off ones, while we want an indication of the quality of the transient. Moreover, such metric would have never converged in the non-static scenarios, where adaptation must be executed continuously. To have insights on the dynamic behaviour, we concocted a metric that, informally: looks behind $R{+}1$ rounds; counts how many times each device changed its leader; sums these changes for every device; and finally normalises on the overall number of rounds considered. In other words, we measure the mean probability that there was a leader change in a round, in any device, in the last $R+1$ rounds. More formally:

- let I be the set of all possible device UIDs;
- let $l_t^d \in I$ be the leader selected in device d at round t;
- let T be the current round;
- for each device d, we consider the set $L_{TR}^d = \{l_T^d, l_{T-1}^d, \ldots, l_{T-R}^d\}$ comprising the leaders perceived in a mobile window spanning the last $R+1$ rounds;
- we then consider the R couples of subsequent leaders $S^d = \{(l_i^d,\ l_j^d) \mid \forall j = i+1;\ l_i^d, l_j^d \in R\}$;
- we define a function over 2-ples $c : I^2 \to \{0, 1\}$

$$c((k_1, k_2)) = \begin{cases} 1, & k_1 = k_2 \\ 0, & \text{otherwise} \end{cases}$$

that outputs 1 iff the elements of the tuple are equal;

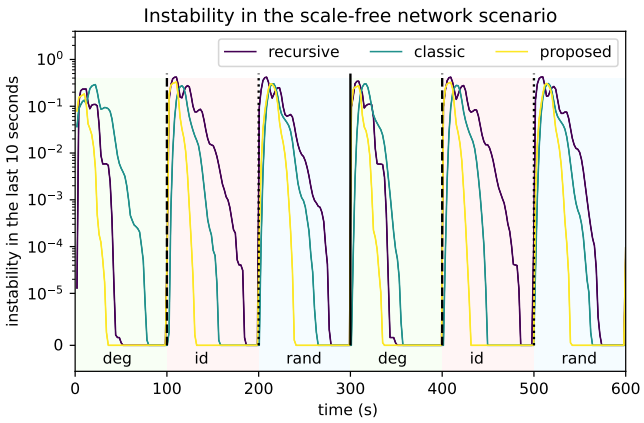

Fig. 4. Algorithms compared in the scale-free network scenario; the instability metric is detailed in Section V-C. Every 100 seconds, the leader strength is changed as indicated in the labels at the bottom of the chart: **deg** (degree of the node), **id** (id of the node), and **rand** (randomly generated value). Despite the (unfairly) timely reset of the gossip, the recursive algorithm shows the worst performance overall. Although the self-stabilisation of the classic algorithm has never been formally proven, it always stabilised in our experiments. The proposed algorithm is the best across the chart: it reaches stability first in all cases.

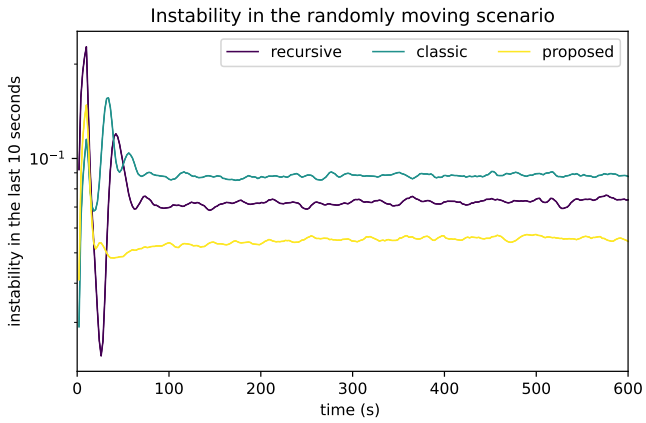

Fig. 5. Algorithms compared in the random movement scenario; the instability metric is detailed in Section V-C. In this setting, we expected the classic algorithm to perform better than the alternatives. Instead, data shows that the opposite happens: both the recursive version and the Bounded Election outperform the baseline, with the proposed algorithm consistently showing better stability throughout the experiment.

- we then count how many times in the last $R+1$ rounds the leader has not changed, normalised on the considered number of rounds: $C^d = \sum_{x \in S^d} c(x) / |S^d|$, we consider this a measure of *local instability*;
- finally, we obtain our global metric of *global instability* $Y = \sum_{d \in I} C^d / |I|$ by normalising the sum of the local contributions.

In our experiment, we consider the instability in the last 11 rounds ($R = 10$) as metric.

D. Implementation and reproducibility

Each scenario has been simulated 200 times with a different random seed; the results discussed in Section V-E are averaged over all the repetitions. The seed controls the shape of the network in the scale-free network scenario, the initial position of devices in the scenarios with movement, and the asynchronous round scheduling. The simulations have been realised in Alchemist [47], the implementation of the recursive and self-stabilising Bounded Election has been written in Protelis [38]. Data generated by the simulator has been analysed using xarray [48]; visual reports of the data have been created via matplotlib [49]. For the sake of reproducibility, the experiment has been open-sourced with MIT Licence, archived on Zenodo, assigned a DOI [42], and made available on GitHub[3].

E. Results

Data shows that Bounded Election outperforms the other algorithms across the board, converging first in the scale-free scenario and showing lower instability in the scenarios with mobile devices.

[3]https://github.com/DanySK/experiment-2022-self-stab-leader-election

The behaviour of the three algorithms under testing for the scale-free scenario is depicted in Figure 4. The behaviour of the recursive version is interesting, as it performs pretty well for some specific choices of the leaders: when leaders are selected based on their degree (hence, centrality in the network), it outperforms the classic algorithm. The situation reverses in all other cases, though: the recursive version suffers from much longer stabilisation times when the best leaders are on the peripheral areas of the network. Although the self-stabilisation of the classic algorithm has never been formally proven, it always stabilised in our experiments.

With all nodes moving randomly (Figure 5), we expected the classic algorithm to perform better than the alternatives, as the selection of the leaders could better adapt. Instead, data shows that the opposite happens: both the recursive version and the Bounded Election outperform the baseline, with the proposed algorithm consistently showing better stability throughout the experiment. The most likely cause of the behaviour is the fact that, on average, the best leader remains quite central in this network, and, as a consequence, in most cases it "projects" a large "area of stability".

Finally, in the edge-inspired scenario (Figure 6), data shows that the recursive version is competitive with the classic one, with Bounded Election by far outperforming the other algorithms. This is indeed the case where the algorithm was expected to shine, as there is a clear preference over which leader should be picked, and picking them has an advantage, as they remain static.

As discussed in Section V-C, obtaining consistently shorter convergence time and higher stability can lead to cascading benefits when (as it usually is the case) other algorithms run on top of network partitioning. Faster adaptation allows the partition-based algorithm to operate with a correct view of their partition, and better stability means that adaptation

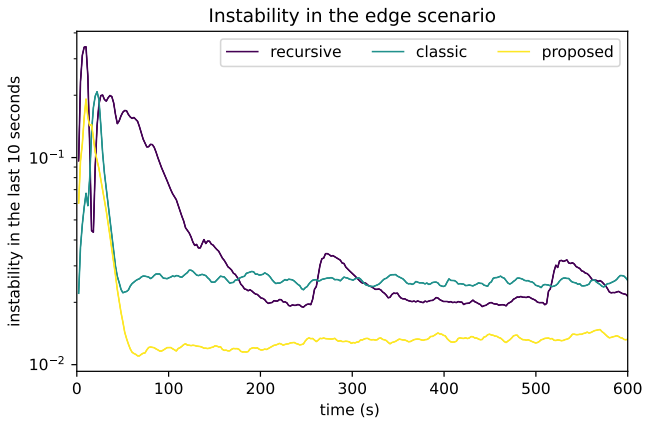

Fig. 6. Algorithms compared in the edge scenario; the instability metric is detailed in Section V-C. In these condition, we expected the algorithms capable to select edge devices over the other devices to perform better, as their areas can remain more stable. Data shows that, while Bounded Election consistently outperform the alternatives, the recursive version, although better than the baseline for most of the time, shows a more erratic behaviour.

mechanisms that sit on top of the partitions will get triggered less frequently. Depending on the specific scenario at hand, this could in turn translate to better performance, improved user experience, and/or reduced energy requirements.

VI. Conclusion

In this work, we have proposed a novel leader election algorithm, Bounded Election, for the aggregate computing model, which is space-based, priority-based, induces network partitioning, and, crucially, is proved to be self-stabilising. Compared to other state-of-the-art solutions, besides its proven self-stability, Bounded Election also shows better dynamic performance, both in terms of shorter stabilisation time and average resilience to small continuous disruption.

Future work should be devoted to a more extensive exploration of scenarios, to find cases in which Bounded Election does not perform well enough. These, when identified and properly addressed, could lead to a new family of self-stabilising algorithms for collective leader election and network partitioning, possibly tailored to general use and corner cases. We also notice that, in principle, the algorithm retains self-stabilisation regardless of the way the leader strength is computed, as far as this computation is self-stabilising, too. Thus, work should be devoted into better understand the consequences of different views on the strength of a leader, and whether this mechanism can be leveraged to achieve useful behaviours.

References

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003. [Online]. Available: https://doi.org/10.1109/MC.2003.1160055

[2] K. L. Bellman, J. Botev, A. Diaconescu, L. Esterle, C. Gruhl, C. Landauer, P. R. Lewis, P. R. Nelson, E. Pournaras, A. Stein, and S. Tomforde, "Self-improving system integration: Mastering continuous change," *Future Gener. Comput. Syst.*, vol. 117, pp. 29–46, 2021. [Online]. Available: https://doi.org/10.1016/j.future.2020.11.019

[3] Ö. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. P. A. van Moorsel, and M. van Steen, Eds., *Self-star Properties in Complex Information Systems, Conceptual and Practical Foundations [the book is a result from a workshop at Bertinoro, Italy, Summer 2004]*, ser. Lecture Notes in Computer Science, vol. 3460. Springer, 2005. [Online]. Available: https://doi.org/10.1007/b136551

[4] H. Haken, *Information and self-organization: A macroscopic approach to complex systems*. Springer Science & Business Media, 2006.

[5] T. D. Wolf and T. Holvoet, "Emergence versus self-organisation: Different concepts but promising when combined," in *Engineering Self-Organising Systems, Methodologies and Applications [revised versions of papers presented at the Engineering Selforganising Applications (ESOA 2004) workshop, held during the Autonomous Agents and Multi-agent Systems conference (AAMAS 2004) in New York in July 2004, and selected invited papers]*, ser. Lecture Notes in Computer Science, S. Brueckner, G. D. M. Serugendo, A. Karageorgos, and R. Nagpal, Eds., vol. 3464. Springer, 2004, pp. 1–15. [Online]. Available: https://doi.org/10.1007/11494676_1

[6] F. Dressler, "A study of self-organization mechanisms in ad hoc and sensor networks," *Comput. Commun.*, vol. 31, no. 13, pp. 3018–3029, 2008. [Online]. Available: https://doi.org/10.1016/j.comcom.2008.02.001

[7] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[8] D. Cavin, Y. Sasson, and A. Schiper, "Consensus with unknown participants or fundamental self-organization," in *Ad-Hoc, Mobile, and Wireless Networks: Third International Conference, ADHOC-NOW 2004, Vancouver, Canada, July 22-24, 2004. Proceedings*, ser. Lecture Notes in Computer Science, I. Nikolaidis, M. Barbeau, and E. Kranakis, Eds., vol. 3158. Springer, 2004, pp. 135–148. [Online]. Available: https://doi.org/10.1007/978-3-540-28634-9_11

[9] S. Dolev and N. Tzachar, "Empire of colonies: Self-stabilizing and self-organizing distributed algorithm," *Theor. Comput. Sci.*, vol. 410, no. 6-7, pp. 514–532, 2009. [Online]. Available: https://doi.org/10.1016/j.tcs.2008.10.006

[10] D. Pianini, R. Casadei, M. Viroli, and A. Natali, "Partitioned integration and coordination via the self-organising coordination regions pattern," *Future Generation Computer Systems*, vol. 114, pp. 44 – 68, 2021. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X20304775

[11] S. Dolev, *Self-Stabilization*. MIT Press, 2000.

[12] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974. [Online]. Available: https://doi.org/10.1145/361179.361202

[13] Y. Mo, J. Beal, and S. Dasgupta, "An aggregate computing approach to self-stabilizing leader election," in *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, September 3-7, 2018*. IEEE, 2018, pp. 112–117. [Online]. Available: https://doi.org/10.1109/FAS-W.2018.00034

[14] N. Mohammed, H. Otrok, L. Wang, M. Debbabi, and P. Bhattacharya, "A mechanism design-based multi-leader election scheme for intrusion detection in MANET," in *WCNC 2008, IEEE Wireless Communications & Networking Conference, March 31 2008 - April 3 2008, Las Vegas, Nevada, USA, Conference Proceedings*. IEEE, 2008, pp. 2816–2821. [Online]. Available: https://doi.org/10.1109/WCNC.2008.493

[15] S. Huang, "Leader election in uniform rings," *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 3, pp. 563–573, 1993. [Online]. Available: https://doi.org/10.1145/169683.174161

[16] W. Shi and P. K. Srimani, "Leader election in hierarchical star network," *J. Parallel Distributed Comput.*, vol. 65, no. 11, pp. 1435–1442, 2005. [Online]. Available: https://doi.org/10.1016/j.jpdc.2005.05.005

[17] N. Malpani, J. L. Welch, and N. H. Vaidya, "Leader election algorithms for mobile ad hoc networks," in *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M 2000), Boston, Massachusetts, USA, August 11, 2000*. ACM, 2000, pp. 96–103. [Online]. Available: https://doi.org/10.1145/345848.345871

[18] C. Glacet, A. Miller, and A. Pelc, "Time vs. information tradeoffs for leader election in anonymous trees," *ACM Trans. Algorithms*, vol. 13, no. 3, pp. 31:1–31:41, 2017. [Online]. Available: https://doi.org/10.1145/3039870

[19] R. Bakhshi, W. J. Fokkink, J. Pang, and J. van de Pol, "Leader election in anonymous rings: Franklin goes probabilistic," in *Fifth IFIP International Conference On Theoretical Computer Science - TCS 2008, IFIP 20th World Computer Congress, TC 1, Foundations*

of Computer Science, September 7-10, 2008, Milano, Italy, ser. IFIP, G. Ausiello, J. Karhumäki, G. Mauri, and C. L. Ong, Eds., vol. 273. Springer, 2008, pp. 57–72. [Online]. Available: https://doi.org/10.1007/978-0-387-09680-3_4

[20] J. Burman, H. Chen, H. Chen, D. Doty, T. Nowak, E. E. Severson, and C. Xu, "Time-optimal self-stabilizing leader election in population protocols," in *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, A. Miller, K. Censor-Hillel, and J. H. Korhonen, Eds. ACM, 2021, pp. 33–44. [Online]. Available: https://doi.org/10.1145/3465084.3467898

[21] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, "From distributed coordination to field calculus and aggregate computing," *J. Log. Algebraic Methods Program.*, vol. 109, 2019.

[22] J. Beal and M. Viroli, "Building blocks for aggregate programming of self-organising applications," in *2nd FoCAS Workshop on Fundamentals of Collective Systems*. IEEE CS, 2014, pp. 8–13.

[23] R. Casadei, S. Mariani, D. Pianini, M. Viroli, and F. Zambonelli, "Space-fluid adaptive sampling: A field-based, self-organising approach," in *Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, ser. Lecture Notes in Computer Science, M. H. ter Beek and M. Sirjani, Eds., vol. 13271. Springer, 2022, pp. 99–117. [Online]. Available: https://doi.org/10.1007/978-3-031-08143-9_7

[24] M. Francia, D. Pianini, J. Beal, and M. Viroli, "Towards a foundational API for resilient distributed systems design," in *2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, September 18-22, 2017*. IEEE Computer Society, 2017, pp. 27–32. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/FAS-W.2017.116

[25] R. Casadei and M. Viroli, "Programming actor-based collective adaptive systems," in *Programming with Actors - State-of-the-Art and Research Perspectives*, ser. Lecture Notes in Computer Science, A. Ricci and P. Haller, Eds. Springer, 2018, vol. 10789, pp. 94–122. [Online]. Available: https://doi.org/10.1007/978-3-030-00302-9_4

[26] Y. Mo, G. Audrito, S. Dasgupta, and J. Beal, "A resilient leader election algorithm using aggregate computing blocks," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 3336–3341, 2020. [Online]. Available: https://doi.org/10.1016/j.ifacol.2020.12.1497

[27] G. Audrito, R. Casadei, F. Damiani, and M. Viroli, "Compositional blocks for optimal self-healing gradients," in *Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2017, pp. 91–100.

[28] J. Beal, J. Bachrach, D. Vickery, and M. M. Tobenkin, "Fast self-healing gradients," in *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, R. L. Wainwright and H. Haddad, Eds. ACM, 2008, pp. 1969–1975. [Online]. Available: https://doi.org/10.1145/1363686.1364163

[29] J. J. Daymude, R. Gmyr, A. W. Richa, C. Scheideler, and T. Strothmann, "Improved leader election for self-organizing programmable matter," in *Algorithms for Sensor Systems - 13th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2017, Vienna, Austria, September 7-8, 2017, Revised Selected Papers*, ser. Lecture Notes in Computer Science, A. F. Anta, T. Jurdzinski, M. A. Mosteiro, and Y. Zhang, Eds., vol. 10718. Springer, 2017, pp. 127–140. [Online]. Available: https://doi.org/10.1007/978-3-319-72751-6_10

[30] B. Gorain, A. Miller, and A. Pelc, "Four shades of deterministic leader election in anonymous networks," in *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, K. Agrawal and Y. Azar, Eds. ACM, 2021, pp. 265–274. [Online]. Available: https://doi.org/10.1145/3409964.3461794

[31] K. Yu, M. Gao, H. Jiang, and G. Li, "Multi-leader election in dynamic sensor networks," *EURASIP J. Wirel. Commun. Netw.*, vol. 2017, p. 187, 2017. [Online]. Available: https://doi.org/10.1186/s13638-017-0961-9

[32] K. Nakano and S. Olariu, "A survey on leader election protocols for radio networks," in *International Symposium on Parallel Architectures, Algorithms and Networks, ISPAN 2002, May 22-24, 2002, Makati City, Metro Manila, Philippines*. IEEE Computer Society, 2002, p. 71. [Online]. Available: https://doi.org/10.1109/ISPAN.2002.1004263

[33] S. Dubois and S. Tixeuil, "A taxonomy of daemons in self-stabilization," *CoRR*, vol. abs/1110.0334, 2011. [Online]. Available: http://arxiv.org/abs/1110.0334

[34] L. Blin and S. Tixeuil, "Compact self-stabilizing leader election for general networks," *J. Parallel Distributed Comput.*, vol. 144, pp. 278–294, 2020. [Online]. Available: https://doi.org/10.1016/j.jpdc.2020.05.019

[35] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the internet of things," *IEEE Computer*, vol. 48, no. 9, pp. 22–30, 2015.

[36] M. Viroli, G. Audrito, J. Beal, F. Damiani, and D. Pianini, "Engineering resilient collective adaptive systems by self-stabilisation," *ACM Trans. Model. Comput. Simul.*, vol. 28, no. 2, pp. 16:1–16:28, 2018. [Online]. Available: https://doi.org/10.1145/3177774

[37] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, Oct. 1999. [Online]. Available: https://doi.org/10.1126/science.286.5439.509

[38] D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical aggregate programming," in *ACM Symposium on Applied Computing (SAC)*, 2015, pp. 1846–1853.

[39] D. Pianini, J. Beal, and M. Viroli, "Improving gossip dynamics through overlapping replicates," in *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, ser. Lecture Notes in Computer Science, A. Lluch-Lafuente and J. Proença, Eds., vol. 9686. Springer, 2016, pp. 192–207. [Online]. Available: https://doi.org/10.1007/978-3-319-39519-7_12

[40] R. Casadei, M. Viroli, G. Audrito, D. Pianini, and F. Damiani, "Aggregate processes in field calculus," in *Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, ser. Lecture Notes in Computer Science, H. R. Nielson and E. Tuosto, Eds., vol. 11533. Springer, 2019, pp. 200–217. [Online]. Available: https://doi.org/10.1007/978-3-030-22397-7_12

[41] R. Casadei, M. Viroli, G. Audrito, and F. Damiani, "FScaFi: A core calculus for collective adaptive systems programming," in *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 12477. Springer, 2020, pp. 344–360. [Online]. Available: https://doi.org/10.1007/978-3-030-61470-6_21

[42] D. Pianini, WhiteSource Renovate, and R. Casadei, "Danysk/experiment-2022-self-stab-leader-election: 1.0.0-dev06+d118d26," 2022. [Online]. Available: https://zenodo.org/record/6566627

[43] G. Audrito, J. Beal, F. Damiani, D. Pianini, and M. Viroli, "Field-based coordination with the share operator," *Log. Methods Comput. Sci.*, vol. 16, no. 4, 2020. [Online]. Available: https://lmcs.episciences.org/6816

[44] V. Zaburdaev, S. Denisov, and J. Klafter, "Lévy walks," *Reviews of Modern Physics*, vol. 87, no. 2, p. 483–530, Jun 2015. [Online]. Available: http://dx.doi.org/10.1103/RevModPhys.87.483

[45] I. Rhee, M. Shin, S. Hong, K. Lee, S. J. Kim, and S. Chong, "On the levy-walk nature of human mobility," *IEEE/ACM Transactions on Networking*, vol. 19, no. 3, pp. 630–643, Jun. 2011. [Online]. Available: https://doi.org/10.1109/tnet.2011.2120618

[46] D. Pianini, R. Casadei, M. Viroli, S. Mariani, and F. Zambonelli, "Time-fluid field-based coordination through programmable distributed schedulers," *Log. Methods Comput. Sci.*, vol. 17, no. 4, 2021. [Online]. Available: https://doi.org/10.46298/lmcs-17(4:13)2021

[47] D. Pianini, S. Montagna, and M. Viroli, "Chemical-oriented simulation of computational systems with ALCHEMIST," *J. Simulation*, vol. 7, no. 3, pp. 202–215, 2013.

[48] S. Hoyer and J. Hamman, "xarray: N-D labeled arrays and datasets in Python," *Journal of Open Research Software*, vol. 5, no. 1, 2017. [Online]. Available: http://doi.org/10.5334/jors.148

[49] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science Engineering*, vol. 9, no. 3, pp. 90–95, May 2007.