Lunar Leader: Persistent, Optimal Leader Election for Multi-Agent Exploration Teams

Keenan Albee
Jet Propulsion Laboratory,
California Institute of Technology
Pasadena, United States
keenan.albee@jpl.nasa.gov

Sriramya Bhamidipati
Jet Propulsion Laboratory,
California Institute of Technology
Pasadena, United States
sriramya.bhamidipati@jpl.nasa.gov

Federico Rossi
Jet Propulsion Laboratory,
California Institute of Technology
Pasadena, United States
federico.rossi@jpl.nasa.gov

Joshua Vander Hook
Jet Propulsion Laboratory,
California Institute of Technology
Pasadena, United States
j.vanderhook@gmail.com

Jean-Pierre de la Croix
Jet Propulsion Laboratory,
California Institute of Technology
Pasadena, United States
jean-pierre.de.la.croix@jpl.nasa.gov

ABSTRACT

Multi-agent exploration teams often benefit from election of a single leader agent to simplify autonomous operations. We discuss distributed leader election (DLE) in the context of the Cooperative Autonomous Distributed Robotic Exploration (CADRE) lunar rover mission–the first fully autonomous multi-agent robotic space mission. DLE is used to select a central planning robot whose role is to receive updated sensor information and to disseminate plans. However, existing algorithms present a number of a shortcomings in the context of practical hardware deployment. These embedded systems pose unique challenges such as limited communication bandwidth, possible network partitions, imprecise timing, limited computational resources, and changing conditions for the optimal leader. This combination of challenges necessitates a DLE algorithm that provides additional robustness to many assumptions of the prior literature. An algorithm built around the Gallager-Humblet-Spira (GHS) algorithm is proposed that can perform leader election despite the aforementioned hardware complications in a persistent fashion where agents might exit, reenter, or change their suitability to be leader. The proposed algorithm is demonstrated in CADRE's four-agent setting, using CADRE's flight software implementation of the algorithm. The necessary algorithmic adaptations are detailed and a simulation case study is provided.

KEYWORDS

distributed algorithms, leader election, multi-agent planning

ACM Reference Format:
Keenan Albee, Sriramya Bhamidipati, Federico Rossi, Joshua Vander Hook, and Jean-Pierre de la Croix. 2024. Lunar Leader: Persistent, Optimal Leader Election for Multi-Agent Exploration Teams. In *Proc. of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), Auckland, New Zealand, May 6 – 10, 2024*, IFAAMAS, 8 pages.

Proc. of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), N. Alechina, V. Dignum, M. Dastani, J.S. Sichman (eds.), May 6 – 10, 2024, Auckland, New Zealand. This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004).

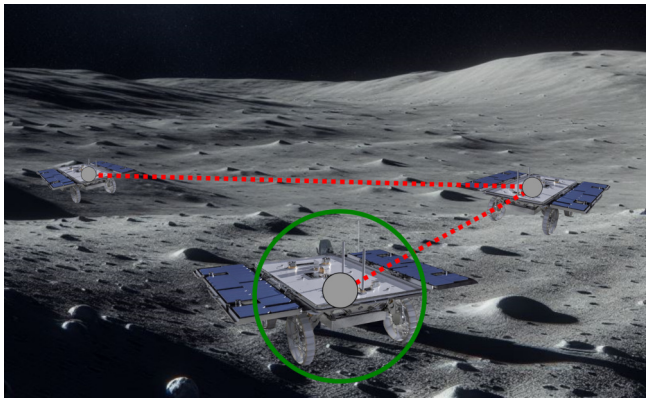

Figure 1: Multi-agent exploration systems like the CADRE lunar rovers depicted above often rely on electing a unique leader to perform decision-making. Practical complications, like dropout and clock drift, hinder leader election.

1 INTRODUCTION

Multi-agent robotic operations frequently require a leader agent to reduce distributed coordination to single-agent decision-making. The Cooperative Autonomous Distributed Robotic Exploration (CADRE) mission [1] is a technology demonstrator that follows a similar approach, relying on a single, unique elected leader for autonomous operation. The benefits of multi-agent systems like CADRE for space exploration are numerous, including increased robustness, fault tolerance, redundancy, and the ability to perform multi-point simultaneous measurements. The CADRE mission consists of a number of small, low-cost, and fully autonomous rovers, each equipped with a suite of scientific instruments. The rovers are intended to explore the lunar surface and drive in formation, under coordination of a unique leader agent.

CADRE's autonomy is supported by a planning system organized by the leader agent. The leader agent is responsible for receiving sensor data from the other agents and generating and disseminating plans—this is a resource-heavy role (e.g., power, heat, and computational resources). An unfit leader can cause all agents to be unproductive for an operational cycle. It is desirable for the leader to shift based on available suitability and possible connectivity changes, such that the current leader is fit for efficient operation for a significant duration. Relying on a ground operator to determine a leader agent can be cumbersome or might be impossible in situations with denied communications. This necessitates an automated, onboard leader election module that performs recurring selection of the most suitable agent as the leader.

The algorithm proposed herein addresses the DLE problem in such a situation, where various assumptions of common DLE algorithms are distorted on hardware or require modification for persistent operation. The proposed solution begins with the well-known Gallager-Humblet-Spira (GHS) algorithm. We proceed by (1) creating spanning trees (ST) for the (possibly disconnected) graph of agents; (2) selecting the unique root of the ST to receive the resource information from each node; (3) using the root node ("the appointer") to select the leader based on the resource information; and (4) disseminating the updated leader. This work's primary contributions are:

- The addition of periodic reelection to accommodate shifting network connectivity (such as loss of agents or bifurcation of the network), easing some of the limitations of single-run GHS.
- Introduction of various practical improvements for hardware concerns such as clock drift and rate-driven timing.
- Composition of the above approach and verification within a spaceflight robotics software framework.

The rest of the paper is outlined as follows: Section 2 introduces related DLE literature; Section 3 formalizes the problem statement; Section 4 discusses the proposed approach; Section 5 demonstrates a four-agent case study result; and Section 6 concludes.

2 RELATED WORK

Leader election in the multi-agent systems literature divides by the definition of "leader" and the assumptions made about the system. Population protocol models assume many agents with extremely limited memory and computation that can only interact with a randomly chosen neighbor at any time [3]. Remarkably, most classical algorithms from distributed systems can be realized on such models, including leader election [2]. However, algorithms designed for such models rely on assumptions that are not relevant to an embedded exploration robotic system, such as nondeterministic interaction patterns with other agents.

Similarly, literature on "flocking" or "swarms" defines a leader not by identifier or fitness for a purpose but as *taking the lead position in a formation*. It is again common to assume robots have a majority of the following limitations: no unique identification, no common coordinate frame, no means of synchronous communication (or, often, any communication at all), no memory, no shared clock, and no agreement on the "size" of the reference frame [6, 9]. The selection of a "leader" in this setting is done by sensing the motion of neighbors and choosing velocity so that the global collective moves in unison; this is beyond the use case needed for many embedded, distributed multi-agent systems.

A third category of literature is that of "self-stabilizing systems" [17]. These systems are designed to recover from any initial state to a desired state, and are often used in the context of distributed systems. A classic leader election result from Dolev et al. [10, 11] shows that in *uniform* networks (those where agents have no unique identifier) of shared-memory processors (those with no message passing cost), a leader can be elected with a limited amount of internal state used for book-keeping. The undesirable aspect of these algorithms is that they are not deterministic, and may require a large number of messages to be passed, which is a concern for multi-robot systems.

Finally, research in "MANET" (mobile ad-hoc networks) is relevant, with multi-phase algorithms of particular interest. These techniques commonly find clusters of nodes, elect a leader within the cluster, then join leaders in a cleaner topology like rings on which higher-level protocols can be built [7]. However, for less than a few dozen connected agents, simpler algorithms exhibit faster convergence.

Our system model is close to the classical "distributed system" model, in which nodes have identifiers and bidirectional communication graphs, and message-passing is allowed, even if the "distributed" nature means that nodes are only aware of their local neighbors. This is the model of Lynch [15] and the leader election problem is well-studied in this context. As noted in [4], the leader election problem on these systems is reducible to finding a spanning tree, because the "root" of the tree is the leader. This is a common first step, and the basis is often the "GHS" algorithm [12] used in this work.

Related to leader election is *consensus*, in which all nodes must agree on a value. This is a more general problem than leader election. The classic Paxos algorithm [14], and the more recent Raft algorithm [16] are designed to ensure a consistent ledger of transactions among all nodes. One such transaction could be the election of a leader. Paxos is notable for its complexity; it is also designed to have a large number of nodes, which are separated into groups, and each group provides redundancy. For small numbers of agents, the complexity of Paxos is not justifiable given the limited available redundancy. Raft is a more recent algorithm, and is algorithmically simpler than Paxos. However, Raft fails when there are fewer than three nodes. It is desirable for systems like CADRE to be able to operate with potentially small disconnected subsets of agents. This was addressed in [8], but a distributed ledger adds complexity and increases message passing overhead. These algorithms are designed for highly performant datacenters, and therefore are not suited for the leader election problem.

Finally, a rich body of work exists on "security," usually using Byzantine fault-tolerant algorithms [18], often in the context of mobile cellular networks. These are not considered in this work, since all nodes are assumed to be non-adversarial. Erroneous sensor data, for example, is not considered a security threat, and is handled by other systems.

3 PROBLEM STATEMENT

A leader election submodule is desired in order to designate a unique, agreed-upon leader among a group of distributed agents, $\mathcal{A} \coloneqq \{n_1, n_2, \ldots, n_p\}$, where p denotes the total number of agents. The members of $\mathcal{A}$ constitute a set of nodes that may communicate with one another using a list of outgoing edges, $e_{i,j}$ forming a communication graph, $\mathcal{G} \coloneqq \{\mathcal{A}, \mathcal{E}\}$, where $\mathcal{E}$ is the set of edges between agents (illustrated in the left side of Figure 3). It is assumed that edges are bidirectional (if you can communicate with an agent, that agent can also communicate with you); the graph is not necessarily connected (agents may form subgraphs). Each agent also has a set of m health metrics, which are assumed to be independent of each other $\mathbf{h}_i \coloneqq \left[h_{i,1}, h_{i,2}, \ldots, h_{i,m}\right]^\top$. It is desired that the leader election module output a leader that is optimal with respect to a weighted combination of these health metrics,

$$J^* = \min_i J_i, \text{ where } J_i = \alpha_i^\top \mathbf{h}_i \text{ and } \sum_j^m \alpha_{i,j} = 1 \qquad (1)$$

which defines an agent's suitability to be leader.[1]

We assume a message-passing communication architecture, where messages arrive atomically and with guaranteed and possibly asynchronous delivery. Dropping of duplicates and message ordering is assumed to be handled at the network layer, using e.g., `TCP` as the underlying message passing mechanism. A synchronized clock is assumed, with reasonably small[2] time synchronization error between agents, δ_t. It is desired that the system tolerate common complications that occur in practice: communication dropout (whereby an edge $e_{i,j}$ disappears) and loss of exact synchronization between system clocks for timed reelection ($\delta_t \neq 0$).

3.1 Assumptions on $\mathcal{G}$ and the Network

As mentioned in Section 1, the GHS algorithm is used to create the spanning tree and create a unique initial leader. The GHS algorithm has a number of assumptions that are commonly made in asynchronous distributed message-passing systems, which are woven into the problem statement to form a "well-behaved" set of agents:

- $\mathcal{G}$ is undirected.
- Pairs of nodes, if connected, agree on the symmetric edge weight between them, $w_{i,j} = w_{j,i}$.
- Edge weights are unique, $\{w_{i,j}, w_{j,i}\} \neq \{w_{l,m}, w_{m,l}\}, \forall i \neq l, j \neq m$.
- At least one node, n_0, wakes up at the start of the algorithm, and all others nodes queue messages ("listening") after n_0 wakes up.
- Received messages, $\mathcal{M}_i$, are queued (FIFO), using $\mathcal{Q}_i$.
- $\mathcal{M}_i$ are delivered *eventually*, meaning finite delay.

An important assumption is not made; $\mathcal{G}$ may not be connected. This means that disconnected subgraphs (network partitions) might exist. In practice, partitions can occur for a number of reasons, including agent failure, wakeup failure, or poor communication. It is assumed that if a disconnected subgraph exists, instead of declaring failure it is preferable to elect a unique leader of each individual subgraph.

4 METHODS

The leader election submodule fits within a larger autonomy stack as shown in Figure 2 (a), outlining the leader election subsystem's connection to other major processes and the order of execution of the algorithm. Ultimately, the leader election submodule is responsible for creating a spanning tree of the agents and electing a leader for each spanning tree based on the state of the agents, as specified in Section 3. The leader election subsystem runs an identical process on each agent. An inter-agent message-passing layer provides ordered delivery of messages, $\mathcal{M}_i$, which may be placed on an internal queue, $\mathcal{Q}_i$, on each agent. Agents may also query for a list of current neighbors (based on valid edges $\mathcal{E}_i$), and health metrics, $\mathbf{h}_i$. Upon completion, the subsystem provides a broadcast of the current leader to other processes, such as an autonomy manager as in Figure 2 (a).

The solution approach performs optimal leader election in two stages, outlined in Figure 2 (b). First, GHS is run on $\mathcal{G}$ to produce a unique root, called the "appointer," and a ST. Next, the appointer broadcasts a request for health metric information, which is convergecast back to the appointer.[3] Upon receipt of all children, the appointer sends out a request to change the leader; upon leader change, each agent relays the leader change to listening processes. This second stage is referred to as leader *selection*, since a leader is chosen with intent rather than by coincidence as in the first stage.

At each agent the leader change is communicated by the leader election module to an autonomy manager. The autonomy manager at the previously elected leader makes a decision (between *allowing for completion* or *terminating*) for each ongoing task based on its remaining duration and criticality. After the autonomy manager at the previous leader terminates, the autonomy manager at the newly assigned leader starts generating new plans for the multi-agent team of its subgraph.

[1] A simple linear combination of health metrics is demonstrated to aid in interpretability of leader selection. However, note that the leader election module is not limited to this cost function and can minimize across any desired non-linear combination of health metrics to define the optimal leader.

[2] "Reasonably small" is defined as $\delta_t << \Delta_{reelect}$, given in Section 4.

[3] Note that convergecast is the inverse of a broadcast in a message-passing system, i.e., instead of a message propagating down from a single root to all nodes, data is collected from outlying nodes through a direct spanning tree to the root.

Numerous complications arise that bend the assumptions of Section 3, which are detailed in Section 4.3. A few notable modifications are required to ensure algorithm convergence: (1) timed reelections; (2) a message-passing pause known as the quiescent period to account for possible clock drift during reelection; and (3) a tick-driven query for stepping forward message processing within a single-threaded environment. The leader selection algorithm itself is depicted in Figure 3, and is outlined in Sections 4.1 and 4.2.

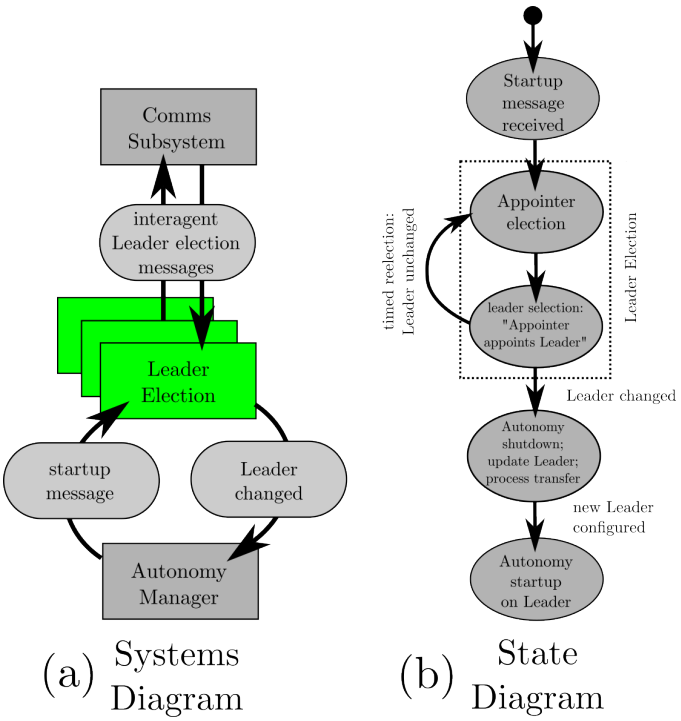

Figure 2: (a) shows a systems diagram of the leader election process on each agent (green) which communicates with a message-passing layer (comms) to pass messages between agents and convey a leader to an autonomy manager. (b) shows state transistions of each leader election process synchronizing an initial election round, followed by persistent reelection cycles following Figure 3. A leader change restarts leader-dependent processes on each agent, such as those of an autonomy manager.

4.1 Stage 1: Appointer Election

The algorithm is built around GHS, which provides: (1) a spanning tree[4], in the form of a list of ST-marked edges for each agent; (2) a unique, agreed-upon root of the final level constructed by GHS. (1) provides a data-efficient method of communicating with the agent graph, while (2) provides the requisite consensus to make single-agent decisions for the entire agent graph.

While a detailed analysis and accompanying proofs of GHS' convergence guarantees can be found in [13] and [15], a cursory summary is also provided here. $\forall n_i \in \mathcal{A}$ a state $s_i := \{l_i, k_i, mw_i\}$ is defined where l_i is a root, k_i is a "level," and mw_i is a "minimum weight outgoing edge," also referred to as a MWOE. Each agent receives $\mathcal{E}_i$, creating (a) of Figure 3. Note that agents do not have global knowledge of connections, only local knowledge of connected neighbors. Given a connectivity graph $\mathcal{G}$ among $\mathcal{A}$, each agent initializes its s_i, as in 3 (b), forming an initial "component," C_i, marked with $l_i = 0$. This is the initialization of GHS. An initial `SRCH` message is sent on each n_i to all $e_{i,j} \in \mathcal{E}_i$, initiating the algorithm. The algorithm proceeds in rounds, as in Figure 3 (c), generating and sending messages in response to those received on the queue $\mathcal{Q}_i$. There are a total of seven possible messages, fully detailed in [13], that mainly consist of probing neighbors for their s_i and determining their status of being within the same C_i. Merge and absorb operations either join C_i along a shared MWOE, or designate the MWOE and root of an absorbing C_i, respectively.

Eventually, all of the graph $\mathcal{G}$ exists as a single C_i with a common k_i and l_i. At this point, a `NOOP` message indicates algorithm convergence—a ST and unique l_i appointer now exist. In greatly simplified form, the message processing of GHS is represented by `GhsRound` of Algorithm 1. Additionally, the assumptions of Section 3 allow the formation of disconnected subgraphs. The powerful benefit of this is that one can use GHS to create multiple spanning trees, and elect a leader within each. As long as communication is preserved, appointer election and leader selection will successfully terminate.

4.2 Stage 2: Leader Selection

With an appointer available, (green circle, Figure 3), coordinated *selection* of a unique, optimal leader can begin. Returning to Figure 3, (d) begins a query process for the optimal J^*, detailed in Algorithm 1. Agents receive a `RecvCost` message broadcast passed down from the appointer, and then convergecast their locally optimal J_i^* back. Once the appointer receives responses from all children the optimal n^* is determined. A final broadcast of a `DeclareLeader` message is made, (e), designating a new leader throughout $\mathcal{G}$. This leader is optimal with respect to the weighting of the queried health metrics, $\mathbf{h}_i$.

The ST is leveraged using an additional leader selection step (Algorithm 1). Note that GHS produces the *minimum* spanning tree, with respect to the given unique, symmetric edge weights. This invites additional useful forms of edge metric, such as communications speed, distance, or other useful quantities. However, since this algorithm only requires *a* spanning tree (for simple broadcasts) and a unique initial leader (to coordinate optimal leader decision-making) a simple metric is used: the agent IDs themselves. Therefore for agents n_i and n_j, edge weights $w_{ij} = w_{ji}$ are simply

[4]Note that GHS provides a *minimum* spannning tree. However, in our case where the weights used are inconsequential, such as agent ID, we simply refer to the tree as *a* spanning tree.

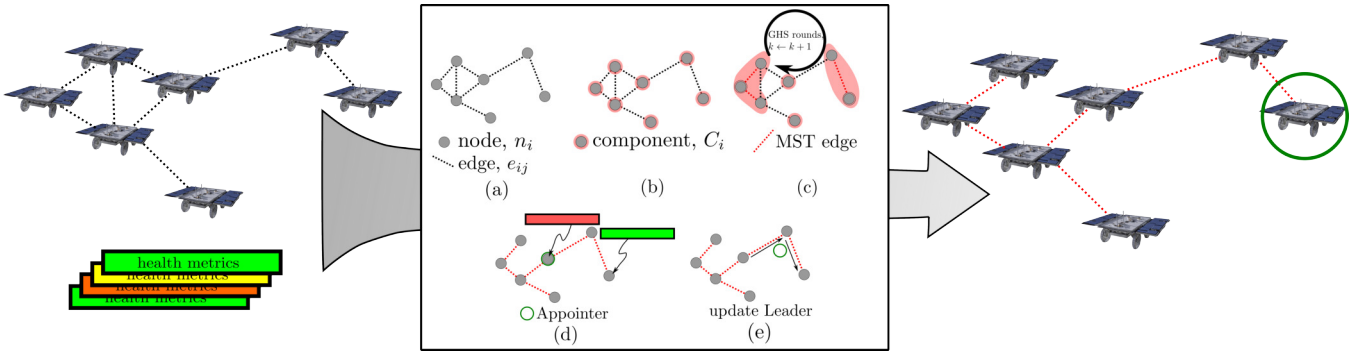

Figure 3: Appointer election begins with a connectivity graph $\mathcal{G}$, shown at left, for which each agent has a set of queryable health metrics, $\mathbf{h}_i$. Leader election runs using a delay-robust periodically rerun version of GHS, producing a ST (red lines) with a unique "appointer" (green circle). Leader selection is performed agent-wise, with the appointer initiating the final ST broadcast if a new leader is declared. The result is an optimal leader over $\mathcal{G}$, shown at right.

Algorithm 1 AppointerElection and LeaderSelection

$\mathcal{G} \leftarrow \{\mathcal{A}_o, \mathcal{E}_o\}$ ▷ $\mathcal{E}_i$ updated on each agent, n_i
$s_i \leftarrow \{i, 0, \infty\}, \ \forall i$
$w_i \leftarrow w_{i,0}, \ \forall i$
$J_i \leftarrow$ based on Eq. (1)

▷ Appointer election, via GHS:
while agent n_i is not converged **do**
  `ReceivefromNeighbors`$(\mathcal{M}_i) \rightarrow \mathcal{Q}_i$
  $\mathcal{M}_i, s_i \leftarrow$ `GhsRound`$(\mathcal{Q}_i, \mathcal{E}_i)$
  `SendToNeighbors`$(\mathcal{M}_i)$
end while

▷ Leader selection:
if `IsAppointer`(n_i) **then**
  `Broadcast`(`RequestCost`)
end if
while ¬ `DeclareLeader` received **do**
  if `type`$(\mathcal{M}_i)$ = `RequestCost` **then**
    `Broadcast`(`RequestCost`)
  else if `type`$(\mathcal{M}_i)$ = `RecvCost` ∧ [`RecvCost` $\forall e_{i,j}$] **then**
    $J^* \leftarrow \min(J_i, \min_{k \in Children(i)} J_k)$
    $n^* \leftarrow$ `getagentID`(J^*)
    if `IsAppointer`(n_i) **then**
      `Broadcast`(`DeclareLeader`(n^*))
    else
      `Convergecast`(n^*, J^*)
    end if
  end if
end while

`strcat`(ij) : $i < j$ (e.g., the edge weight is 25 for agents with IDs 5 and 2).

Accounting for GHS compute based on [13] and adding in the compute from the leader *selection* process, the total time until completion will be $O(N\,log_2 N + 2(N-1))$. Thus, given that the leader election submodule requires only a small number of messages to be exchanged, we can assume that the communication is completed within each reelection cycle. As long as communication is completed, appointer election and leader selection will successfully terminate.

4.3 Ensuring Algorithmic Assumptions: Reelection and Quiescence

Practical deployments may encounter breaches of the assumptions of Section 3. Unique additions make the leader selection algorithm deployable on hardware that distorts these assumptions. The implications of satisfying these assumptions are discussed here for embedded robotics hardware such as the CADRE rovers, where deviations might occur, and solutions are offered via Algorithm 2.

Undirected Graph and Agreed-Upon Edge Weights If using a physical quantity like communications bandwidth/latency, the cost to communicate may not be symmetric; a node may have an easier time hearing than transmitting. Instead, for these metrics we include an "exchange" phase in the GHS algorithm, where the two nodes probe each other to determine the cost of communication. This cost is then exchanged, and a shared cost is used for the edge. Being distributed, the edge weights only need to be agreed upon by the two adjacent nodes, and need not be shared with any other nodes. An alternative solution is to use agent IDs as in Section 4.2.

Unique Edge Weights – While it is unlikely that two pairs of nodes will have the exact same cost to communicate, it is a strict requirement that edge weights are unique for the correctness of the GHS algorithm. To accommodate this, we set the edge weights (after probing) to be,

$$w_{i,j} = \texttt{shared\_weight}(a_i, a_j) << 16 + \max(i,j) << 8 + \min(i,j), \tag{2}$$

where << is the bit shift operator and 16 and 8 clear space for two bytes of memory to store ordered agent ID. This is a simple way to ensure that the edge weights are distinct but

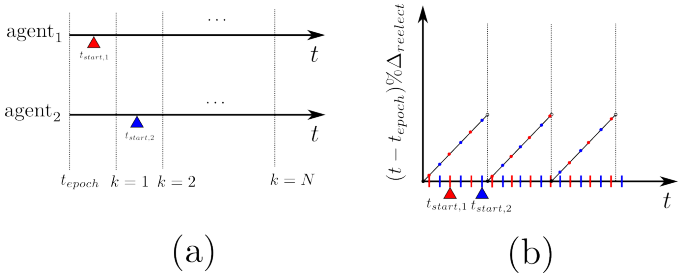

Figure 4: Agent queries are performed assuming a hardware-driven periodic timer. Agents desire reelection at every k-th reelection interval (a), but agents might start ahead of or behind the ground truth clock. Regardless, queries can be made at the timer frequency f, providing a trigger if the k-th reelection period has been passed, (b).

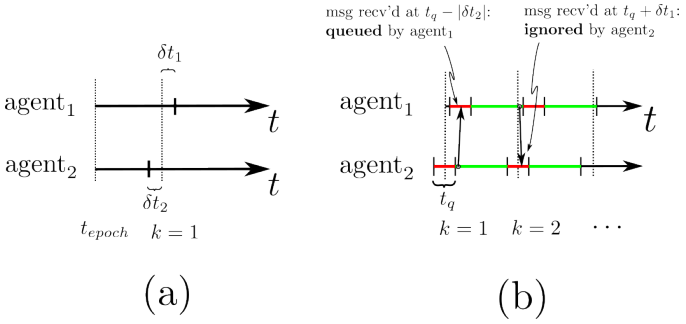

Figure 5: Here, agent$_1$ is delayed relative to the ground truth clock (δt_1); agent$_2$ is ahead of time (δt_2), (a). In (b), time-bounded out-of-sync messages are dealt with by adding a quiescent period t_q (red), during which messages may be received, but are not sent. Messages with old timestamps are ignored within the quiescent period; successive rounds are indicated by k.

consistent for pairs of nodes. Again, using agent IDs as in Section 4.2 is an alternative simple solution.

Wake Up and Listening – Ideally, the wake up sequence would be coordinated by a synchronized clock, triggered by a high rate system query. In practice, this is accommodated by a hardware rate-driven query that periodically checks against a system clock that might have drift, δt. This process triggers the first leader election process, and all subsequent reelection cycles. The problem is illustrated in Figure 4. We desire reelections at $(t - t_{epoch})\%\Delta_{reelect} = 0$ where $\Delta_{reelect}$ is the desired reelection period, and t_{epoch} is a synchronized epoch time; that is, $t_{k,reelect} = t_{epoch} + k\Delta_{reelect},\ \forall k$.

t can be sampled at some sample frequency, f, provided by what is often known as a rate group in embedded software frameworks. This leads to an accuracy of, at worst, $\frac{1}{f}$ away from the ground truth reelection period or $\frac{2}{f}$ between any set of two agents. For wakeup, this means that agents may not be perfectly synchronized, Figure 4 (a); however, periodic queries will eventually get all agents to an initialized, listening state Figure 4 (b).

Message Delivery: FIFO – This is a requirement imposed on the network layer. We assume that the network layer is implemented using e.g., `TCP`, which provides this guarantee.

Message Delivery: Eventual – Rather than requiring that messages be delivered in a finite amount of time, we instead impose a *timeout* on message delivery, in the form of reelection, described in the following subsection. When a timeout is triggered on the k_ith round, all agents stay idle until the next reelection cycle is triggered, i.e., the k_{i+1}th round. At every reelection, the agent graph $\mathcal{G}$ is reinitialized based on available neighbors $\mathcal{E}_i$ provided by a communications subsystem.

No Agents Drop During Execution – Typically, a static communications graph $\mathcal{G}$ is assumed for the duration of GHS' execution. However, as noted in Section 1, various practical reasons can lead to an edge dropping. In some cases, this can cause GHS to deadlock, such as when awaiting a returned `SRCH` message. This is dealt with through periodic reelections, at reelection rate $\Delta_{reelect}$. A strong assumption that t_{epoch} is synchronized $\forall i$ between agents is required; as in Figure 5 offsets may occur.

An important additional complication arises for clock drift δt_1 and δt_2 between any two sets of agents, (a) of Figure 5. Delays due to query rate, clock drift, and message-relaying time can cause relative drift between when repeated leader election cycles are triggered. This is problematic because, as shown in (b), messages from a prior round k_i might enter the k_{i+1}th round on another agent, breaking GHS and leader selection assumptions.

This is dealt with using a quiescent period, t_q, highlighted in red, during which messages are not sent and are only queued if a message's timestamp is in the future. The quiescent period, along with the reelection solution and rate-driven queries are summarized in Algorithm 2. Significantly,

$$t_q > \sum_m t_{delay,m} \tag{3}$$

that is, the longest possible relative delay is contained within the quiescent period. This ensures that no matter how out-of-sync agents become within $\delta t_1 + \delta t_2 < t_q$ FIFO, eventual delivery is assured.

5 RESULTS

The resulting algorithm is demonstrated in a four-agent case study to demonstrate some of its most important features: (1) robustness to agent dropout; and (2) optimal leader changes upon changes to health metrics and network dropout/reconnection. Subsections 5.1 and 5.2 operate on a fully-connected agent graph of four agents,

$$\mathcal{E} := \{e_{1,2}, e_{1,3}, e_{1,4}, e_{2,1}, e_{2,3}, e_{2,4}, e_{3,1}, e_{3,2}, e_{3,4}, e_{4,1}, e_{4,2}, e_{4,3}\}.$$

Two equally weighted, normalized health metrics are used: `state of charge` $\in [0, 100]$ and `CPU temperature` $\in [0, 150]$,

Algorithm 2 RobustReelection

```
Start(Q_i)
t_elapsed ← 0
t_elapsed,last ← Δ_reelect
for ClockTick do
    t_elapsed ← (t − t_epoch)%Δ_reelect
    if t_elapsed < t_elapsed,last then
        if t_elapsed > t_q then
            ▷ From Algorithm 1:
            AppointerElection and LeaderSelection
        else
            if t_M_i > t then  ▷ Sender agent's clock ahead
                Q_i ← M_i                          ▷ Queue
            else               ▷ Sender agent's clock behind
                ∅ ← M_i                            ▷ Drop
            end if
        end if
    end if
    t_elapsed,last ← t_elapsed
end for
```

in units of percent and Celsius, respectively. The algorithm is implemented in simulation, running four independent processes that are integrated into CADRE's flight software.[5] The implementation used for these results will also be used to perform persistent leader election on the lunar surface for the four CADRE lunar rovers.

5.1 Reelection: Robustness to Agent Dropout

In this demonstration, agents are dropped one-by-one, starting with n_4 and progressing through n_3 and n_2. Total message counts exchanged are recorded and shown in Figure 6. The average number of messages exchanged decreases as $|\mathcal{G}|$ shrinks, ultimately reaching zero when agent n_1 becomes the *de facto* leader. Initially, `state of charge` is set to 100% on n_4 and equal on other agents, appointing n_4 as the initial optimal leader. Leader election, performed persistently after each agent drop, progresses from $n_4 \to n_1 \to n_1 \to n_1$. Note that $n_{1:3}$ have equal health metrics, in which case the appointer (in this case n_1) is selected as a tiebreaker. Message counts are inclusive of both messages received during a round of GHS (stage 1), and during leader selection (stage 2). In CADRE's flight software implementation, the leader election submodule also assigns additional weight to the leader selected in the previous reelection cycle, reducing excessive leader changes.

5.2 Reelection: Optimal Leader Change

A simple experiment is performed for the fully connected case. The health metrics in Table 1 are specified for four fully-connected agents. Health scores, J_i, are determined after normalizing and weighting metrics. Multiple agent dropouts

[5]CADRE's flight software is based on F-prime, a free and open-source flight software framework that is tailored to small-scale systems such as CubeSats, SmallSats, and instruments, and has been demonstrated on the Ingenuity helicopter technology demonstration on Mars [5].

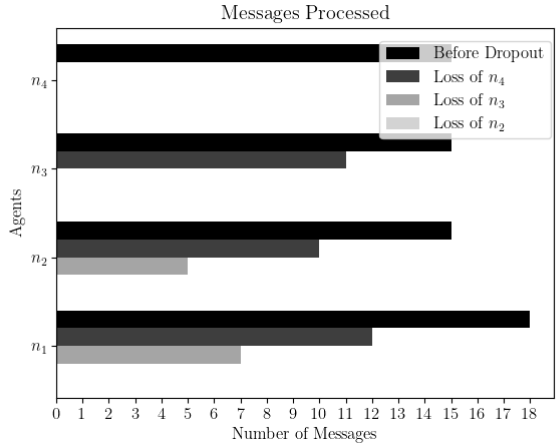

Figure 6: Messages processed by each agent after successive reelection cycles following the loss of an agent in $\mathcal{G}$. Note that this demonstration includes additional message counts for a `Convergecast` of each child in LeaderSelection.

and reconnections are performed. The reelection sequence (without leader persistence) proceeds as anticipated, selecting the most desirable leader at each reelection, updating $\mathcal{E}_i$ appropriately at each cycle. Optimal temperature was set to `CPU temperature`* := 27; optimal `state of charge`* := 100. n_3 was set to near-optimal values, while the other agents had middling weighted health scores, J_i. Leaders were elected in the expected order between reelection cycles, with dropouts and reconnections noted in Table 2.

Table 1: Agent health metric parameters and weighted, normalized health score used for the reelection demonstration.

	n_1	n_2	n_3	n_4
SoC [%]	10	50	100	80
Temp $[C]$	25	5	25	100
Health $[J]$	0.538	0.643	0.988	0.554

Table 2: Leader changes following dropout and reconnection of agents to $\mathcal{G}$ over multiple reelection cycles. Note that the unique leader moves to the best available agent, using J.

Agent Change	$\varnothing$	$\not\|_4$	$\not\|_3$	$\not\|_2$	n_4	n_3
Leader	3	3	2	1	4	3

6 CONCLUSION

A leader election algorithm that runs persistently and provides optimal leader selection has been demonstrated. Easing some of the strong assumptions of GHS, the proposed

approach is a useful addition for embedded multi-agent hardware systems that might suffer from: (1) bounded interagent clock drift; (2) rate-driven system queries; and (3) communications dropout. Moreover, the selection process after an appointer has been declared ensures that periodic reelection also accounts for shifting health metrics that may modify the optimal leader. The algorithm also permits election within disconnected subgraphs, so that divided multi-agent teams may continue with separate leaders.

Results demonstrated the approach for a simple fully connected four-agent system, which shows reasonable message traffic for optimal leader election and reelection of a shifting optimal leader when health metrics and the set of connected agents changes. The algorithm will be deployed to the CADRE lunar rovers launching to the Moon, providing persistent election that is robust to anticipated hardware implementation challenges while shifting leaders in response to on-the-fly conditions like battery state of charge.

While the proposed algorithm has been designed with the CADRE mission in mind, the algorithm is scalable to large numbers of agents and is agnostic to its hardware platform, provided the assumptions of Sections 3 (relaxed somewhat in Section 4.3) are met. Other multi-agent teams such as distributed environmental monitoring systems or search and rescue robots could surely benefit from persistent election of the most suitable leader under realistic hardware assumptions.

REFERENCES

[1] National Aeronautics and Space Administration. 2022. Cooperative Autonomous Distributed Robotic Exploration. (2022). https://www.nasa.gov/cooperative-autonomous-distributed-robotic-exploration-cadre/

[2] Dan Alistarh and Rati Gelashvili. 2018. Recent algorithmic advances in population protocols. *ACM SIGACT News* 49, 3 (2018), 63–73.

[3] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J Fischer, and René Peralta. 2004. Computation in networks of passively mobile finite-state sensors. In *Proceedings of the twenty-third annual ACM Symposium on Principles of Distributed Computing*. 290–299.

[4] Baruch Awerbuch. 1987. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the nineteenth annual ACM Symposium on Theory of Computing*. 230–240.

[5] Robert Bocchino, Timothy Canham, Garth Watney, Leonard Reder, and Jeffrey Levison. 2018. F Prime: an open-source framework for small-scale flight software systems. (2018).

[6] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. 2013. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence* 7 (2013), 1–41.

[7] Orhan Dagdeviren and Kayhan Erciyes. 2008. A hierarchical leader election protocol for mobile ad hoc networks. In *Computational Science–ICCS 2008: 8th International Conference, Kraków, Poland, June 23-25, 2008, Proceedings, Part I 8*. Springer, 509–518.

[8] Ricardo Dias, Bernardo Cunha, José Luis Azevedo, Artur Pereira, and Nuno Lau. 2019. Multi-robot fast-paced coordination with leader election. In *RoboCup 2018: Robot World Cup XXII 22*. Springer, 19–31.

[9] Yoann Dieudonné, Franck Petit, and Vincent Villain. 2010. Leader election problem versus pattern formation problem. In *Distributed Computing: 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings 24*. Springer, 267–281.

[10] Shlomi Dolev, Amos Israeli, and Shlomo Moran. 1993. *Uniform self-stabilizing leader election: part 1: complete graph protocols*. Citeseer.

[11] Shlomi Dolev, Amos Israeli, and Shlomo Moran. 1994. *Uniform self-stabilizing leader election part 2: general graph protocol*. Citeseer.

[12] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. 1983. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)* 5, 1 (1983), 66–77.

[13] R. G. Gallager, P. A. Humblet, and P. M. Spira. 1983. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Transactions on Programming Languages and Systems* 5, 1 (Jan. 1983), 66–77. https://doi.org/10.1145/357195.357200

[14] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*. 277–317.

[15] Nancy A. Lynch. 1997. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, Calif.

[16] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 305–319.

[17] Marco Schneider. 1993. Self-stabilization. *ACM Computing Surveys (CSUR)* 25, 1 (1993), 45–67.

[18] Sudarshan Vasudevan, Brian DeCleene, Neil Immerman, Jim Kurose, and Don Towsley. 2003. Leader election algorithms for wireless ad hoc networks. In *Proceedings DARPA Information Survivability Conference and Exposition*, Vol. 1. IEEE, 261–272.