

Leader Election for Autonomous Decision-Making (LEAD) for Coordinating CADRE’s Team of Lunar Rovers

Sriramya Bhamidipati¹, Keenan Albee², Federico Rossi¹, Viet Nguyen¹,
Josh Vander Hook³, and Jean-Pierre de la Croix¹

Abstract—Multi-agent systems are vital for executing complex planetary surface missions, however, their success relies on fault-resilient coordination among agents. A key mechanism for this is leader election—the ability to dynamically assign and reassign leadership roles among agents. Existing literature present a number of shortcomings in the context of practical hardware deployment; furthermore, a non-optimal leader can result in unproductivity for an entire operational cycle. To address this, a dynamic leader election protocol, termed as Leader Election for Autonomous Decision-making (LEAD), is proposed in the context of the Cooperative Autonomous Distributed Robotic Exploration (CADRE) technology demonstration. Built around the Gallager-Humblet-Spira (GHS) algorithm, LEAD accounts for hardware constraints, such as agent dropouts, imprecise time synchronization, and dynamic variations in the agent health, to determine the leader. The algorithm readiness is demonstrated through integration into a spaceflight software stack and successful validation on CADRE hardware, including high-fidelity development models and the actual flight rovers.

I. INTRODUCTION

Multi-agent systems are vital for executing complex tasks in dynamic, communication-constrained settings—specifically for planetary robotic teams. Success in these domains require resilient coordination where the failure of one agent does not jeopardize the entire operation. This creates an urgent need for fault-tolerant solutions.

A key enabler for such robustness is the ability to dynamically assign and reassign leadership roles among agents to facilitate quick centralized decision-making, resource allocation, and task delegation. Leader election enables this capability by allowing multiple agents to autonomously select and maintain the most suitable peer as their leader, based on real-time sensor data, resource availability (such as power and thermal state), and evolving network conditions [1]. This capability is crucial as

relying on a ground operator to determine a leader agent undermines onboard autonomy and can be cumbersome or might be impossible in situations with denied or limited communications with Earth. However, despite the importance of leader election protocols, existing state-of-the-art methods prioritize global consistency over system availability and lack hardware-proven implementations that remain operational during network partitions in resource-constrained space regimes [2], [3]. This necessitates dynamic leader election that can perform recurring selection of the most suitable agent as the leader.

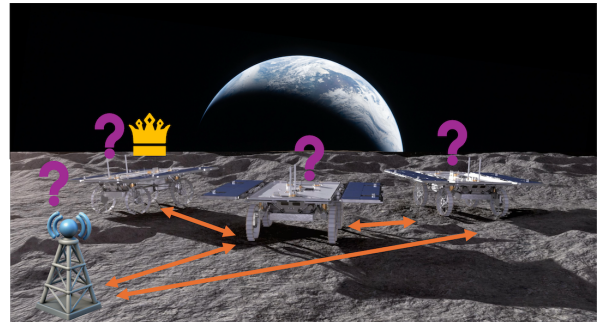


Fig. 1. Multi-agent systems like the CADRE lunar rovers rely on electing a leader to perform decision-making. Practical complications, like agent dropout and clock drift, hinder leader election.

One use case for such a leader election paradigm is NASA’s proposed Cooperative Autonomous Distributed Robotic Exploration (CADRE) lunar technology demonstration of multi-agent autonomy [4], [5], comprising three mobile rovers and a static base station. CADRE’s autonomy is supported by a planning system organized by a leader agent. The leader agent is responsible for receiving sensor data from the other agents and generating and disseminating plans—a role that requires significant computational resources, resulting in severe thermal and power demands. By dynamically appointing a fit leader, the agents autonomously and seamlessly coordinate complex cooperative tasks such as formation driving and multi-static ground-penetrating radar surveys.

We propose an algorithm for leader election for autonomous decision-making (LEAD).¹ The LEAD algorithm, developed for CADRE, showcases resiliency in situations where the various assumptions of common leader election algorithms are violated on hardware or require

^{*}This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004). Copyright ©2026. All rights reserved.

¹Sriramya Bhamidipati, Federico Rossi, and Jean-Pierre de la Croix are with the NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109, USA. {sriramya.bhamidipati, federico.rossi, jean-pierre.de.la.croix}@jpl.nasa.gov

²Keenan Albee is now with the Department of Astronautical Engineering and Department of Aerospace and Mechanical Engineering, University of Southern California, Los Angeles, CA 90089, USA. kalbee@usc.edu

³Josh Vander Hook is now with Flight Sciences, Los Angeles, CA, USA.

¹This manuscript subsumes our non-archival workshop paper [6], expanded to present a complete algorithmic framework alongside new experimental validation on the CADRE flight hardware.

modification for persistent operation. The proposed solution is based on the well-known Gallager-Humblet-Spira (GHS) algorithm [7]. LEAD proceeds by (1) creating spanning trees for the (possibly disconnected) graph of agents using GHS; (2) selecting the unique root of the spanning tree to receive the health information from each node; (3) using the root node (“the appointer”) to select only the leader but also a designated survivor (i.e., a backup leader) based on the received health information; and (4) disseminating the updated leader and designated survivor to all agents in the network (Figure 2).

The key contributions of this proposed work are:

- The implementation of a resilient GHS-based appointer election that performs periodic reelections to accommodate shifting network connectivity (such as loss of agents or bifurcation of the network)—thereby overcoming the downsides of running GHS only once at startup, which leaves the network vulnerable to stale roles as connectivity changes.
- The introduction of robust mechanisms to address hardware-induced concerns including network delays, agent/message dropouts, and clock drift.
- Demonstration of the flight readiness of the LEAD algorithm by integrating it in an end-to-end flight software (FSW) framework [8] and successfully validating performance on CADRE hardware platforms including the actual flight models (FMs) and the high-fidelity development models (DMs).

The rest of the paper is outlined as follows: Section II describes related work; Section III formalizes the problem statement; Section IV discusses the proposed algorithm; Section V demonstrates results on hardware; and Section VI summarizes the paper.

II. RELATED WORK

Literature on leader election in multi-agent systems is divided based on the definition of “leader” and the system assumptions made. Population protocol models assume many agents with extremely limited memory and computation that can only interact with a randomly chosen neighbor at any time. 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 realistic for exploration robots that have non-deterministic interaction patterns with other agents.

Significant control theory literature [9], [10] chooses a leader that optimize graph controllability or minimize network variance under noise, however, planetary robots require prioritizing resilient role maintenance and network connectivity over algebraic network optimization.

The 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 communication, no memory

and no shared clock [11]. The selection of a “leader” in such cases 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.

Another category of literature covers self-stabilizing systems, which recover from arbitrary transient failures to a desired state. Classic results by Dolev et al. [12] show that uniform networks of shared-memory processors can elect a leader with bounded local memory; however, these randomized algorithms can incur high message overhead.

Research in mobile ad-hoc networks and multi-hop radio networks is also highly 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 [13]. Electing a leader in such multi-hop radio environments often introduces additional timing and collision-avoidance constraints [14]. However, for less than a few dozen connected agents, simpler algorithms exhibit faster convergence without the overhead of multi-phase clustering.

More broadly, classical distributed paradigms and consensus frameworks rely on strict operational assumptions that are not realistic on resource-constrained space hardware. Bully-style algorithms [15] depend on rigid timeouts, causing continuous election churn under transient link drops. Alternatively, frameworks like Paxos [16] and Raft [17] maintain replicated transaction ledgers to elect leaders and preserve state. While Raft is simpler than Paxos, it relies on a strict majority quorum and stalls when agents are less than three or when network partitions isolate small sub-swarms. Although distributed ledgers can be adapted for small networks, they incur significant message passing overhead tailored for high-performance data centers rather than exploration robots.

Our system model closely aligns with the classical “distributed system” model where nodes possess unique identifiers, communicate via bidirectional message-passing, and maintain knowledge restricted to immediate local neighbors [18]. As noted in [19], 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 [7] used in this work. Even when edge weights are constructed deterministically from agent IDs, GHS offers fundamental efficiency advantages over the alternative approaches discussed above: it strictly bounds message complexity to $\mathcal{O}(|E| + N \log_2 N)$ on arbitrary graphs while achieving time-optimal execution bounds of $\mathcal{O}(N)$ [19]. Also, GHS operates purely through localized neighborhood exchanges and natively constructs spanning trees within isolated components, enabling lightweight message passing to achieve leader election without requiring global network knowledge.

III. PROBLEM STATEMENT AND ASSUMPTIONS

A distributed, dynamic leader election protocol is desired to designate a unique, agreed-upon leader among a group of distributed agents, $\mathcal{A} := \{n_1, n_2, \dots, n_p\}$, 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, $\mathcal{G} := \{\mathcal{A}, \mathcal{E}\}$, where $\mathcal{E}$ is the set of edges between agents (illustrated in the left side of Fig. 2). It is assumed that edges form a bidirectional communication link; the graph is not necessarily connected (agents may form subgraphs). We assume that each agent has real-time knowledge of its local neighbors (i.e., agent i knows the availability of communication links $e_{i,j}$ and the set of agents $\{j : e_{i,j} \in \mathcal{E}\}$; we do not assume that agents know the entire set of agents $\mathcal{A}$ or the communication graph $\mathcal{E}$. We assume a message-passing communication architecture, where messages arrive atomically and with guaranteed and possibly asynchronous delivery.

As mentioned in Section I, the GHS algorithm is used to create the spanning tree and identifying a unique root agent. 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 a symmetric edge weight between them, $w_{i,j} = w_{j,i}$.
- 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).
- $\mathcal{M}_i$ are delivered *eventually*, meaning finite delay.
- Edge weights are unique, such that $\{w_{i,j}, w_{j,i}\} \neq \{w_{l,m}, w_{m,l}\}, \forall i \neq l, j \neq m$.

In practice, network partitions might occur due to agent failure, wakeup failure, or poor communication. When such partitions exist, instead of declaring failure, a leader is selected within each isolated connected component.

IV. PROPOSED LEAD ALGORITHM

LEAD runs an identical process on each agent to elect a leader based on agent states. For connected network, this yields a single fleet-wide leader, whereas disconnected networks produce a component-local unanimous leader for each isolated partition. During this, an inter-agent message-passing layer provides ordered delivery of messages, $\mathcal{M}_i$, which is placed on an internal queue, $\mathcal{Q}_i$, on each agent. The LEAD algorithm is executed in two stages. First, GHS is run on graph $\mathcal{G}$ to produce a unique root node, called the *appointer*, and a *spanning tree*. Next, the appointer *broadcasts* a request for health metrics, which is later *convergecast*—meaning data is collected from children back to the appointer—via tracing the spanning tree. This second stage is known as *leader selection*, since a leader is chosen with intent

rather than being governed by network state as in the first stage. Upon receipt from all children, the appointer selects the leader and sends out a request to change the leader (if different); upon leader change, each agent relays the leader change to their listening processes.

A. Stage 1: Appointer Election based on GHS

While a detailed analysis of GHS convergence can be found in [1], [7], a summary is provided here. The GHS algorithm starts with a forest in which each agent forms its own disconnected cluster and serves as the root of that cluster. Clusters grow by merging with other clusters through their *minimum-weight outgoing edge* (MWOE), which is identified by collecting the weights of all outgoing edges within a cluster and propagating the candidate information to the root. When two clusters merge, one of the agents connected by the selected edge is designated as the root of the newly formed cluster. Additional mechanisms ensure that cycles cannot be created during merging and that the process remains correct and eventually halts even when message delays cause some nodes to lag behind others.

$\forall n_i \in \mathcal{A}$ 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 the MWOE. Each agent receives $\mathcal{E}_i$, creating (a) of Fig. 2. Given a graph $\mathcal{G}$ among $\mathcal{A}$, each agent initializes its s_i , as in Fig. 2 (b), forming an initial *component*, C_i , marked with $k_i = 0$. An initial SRCH message is sent on each n_i to all $e_{i,j} \in \mathcal{E}_i$, initiating the GHS. The algorithm proceeds in rounds, as in Fig. 2 (c), generating and sending messages in response to those received on the queue $\mathcal{Q}_i$. There are a total of seven possible messages [7], that 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 component C_i with a common level k_i . At this point, the final core edge (u, v) connecting the last two merging fragments is identified, where each endpoint proposes itself as the root l_i . To break symmetry and establish a single persistent appointer, a deterministic tie-breaking rule selects the node with the higher identifier, $l_i = \max(u, v)$, as the root. The root l_i then initiates a directed pass along the spanning tree to establish parent-child edge orientations pointing away from l_i . Finally, a NOOP message indicates algorithm convergence—a rooted spanning tree and unique appointer l_i now exist. The GHS message passing is shown by **GhsRound** of Algorithm 1. The network is assumed to remain stationary throughout the GHS process; otherwise, appointer election may fail. In practice, this GHS phase is typically very fast, making the static assumption reasonable in many settings.

B. Stage 2: Leader Selection

Each agent n_i has a vector of m health metrics, $\mathbf{h}_i = [h_{i,1}, h_{i,2}, \dots, h_{i,m}]^\top$, normalized such that elements are independent. It is desired that the LEAD algorithm

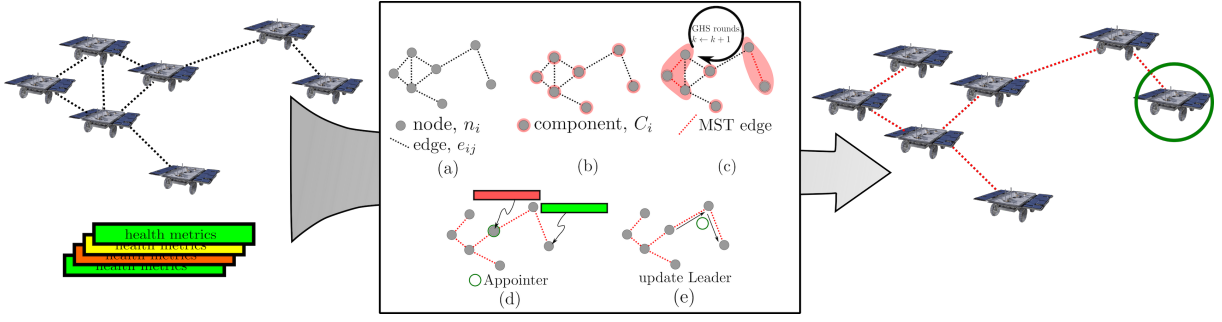


Fig. 2. Appointer election begins with a connectivity graph $\mathcal{G}$, shown at left, for which each agent has a set of queryable health metrics, $\mathbf{h}_i$. Appointer election runs using a delay-robust periodically rerun version of GHS, producing a spanning tree (red lines) with a unique *appointer* (green circle). Leader selection is performed agent-wise, with the appointer initiating the final spanning tree broadcast if a new leader is declared. The result is an *optimal leader* over $\mathcal{G}$, shown at right.

output a leader that is optimal with respect to a weighted combination of these health metrics,

$$J^* = \max_{i \in \mathcal{A}} J_i, \text{ where } J_i = \sum_{j=1}^m \alpha_j f_j(h_{i,j}) \text{ and } \sum_{j=1}^m \alpha_j = 1, \quad (1)$$

where J_i defines overall suitability of agent n_i to serve as leader with $J_i = 1$ representing optimal health and 0 the least desirable state. $f_j(\cdot)$ is a normalization mapping, and $\alpha_j \in [0, 1]$ are metric weighting factors.

With an appointer available, (green circle, Fig. 2), coordinated *selection* of an optimal leader begins. In Fig. 2 (d), a query process is begun for the optimal J^* , detailed in Algorithm 1. The spanning tree is leveraged in this stage to relay requests from appointer to all the agents in the network and vice-versa. Agents receive a **RequestCost** message *broadcast* passed down from the appointer, and then *convergecast* their local health metrics J_i back upstream via **RecvCost** messages. Once the appointer verifies **ReceivedAllCosts** from all agents, the optimal leader L^* and survivor S^* are determined. A final *broadcast*—meaning data is disseminated from appointer to all children using **DeclareLeaderSurvivor** message along the spanning tree, (e), designating a new leader throughout $\mathcal{G}$. In the event that two or more agents report identical health metrics, the appointer selects the agent whose message was received first.

For simplicity, the flight implementation of the *convergecast* shown in line 18 of Algorithm 1 is an $O(N^2)$ operation. This complexity can be readily reduced to $O(N)$ by storing all children's health at each node, and only forwarding the top two health to the parent when all children's reports have been gathered.

C. Robust Hardware Deployment via Reelections, Quiescence, Leader Persistence and Designated Survivor

To ensure robustness on physical hardware such as the CADRE lunar rovers, the proposed LEAD algorithm incorporates several unique adaptations. A few notable modifications are leader persistence, timed reelections, a message-passing pause known as the quiescent period to account for possible clock drift during reelection, and a rate-driven query for stepping forward message processing within a single-threaded environment.

Designated Survivor—To ensure operational redundancy in CADRE lunar rovers, the LEAD enforces a fail-safe requirement to elect a *designated survivor*, alongside the leader. The appointer assigns this role to the agent with the next best health evaluated during the convergecast. In the event of leader failure, the designated survivor preserves system state by maintaining a redundant copy of system logs and distributed plans; this is maintained by listening processes outside LEAD.

Leader Persistence—By separating appointer selection, a sophisticated "persistence" logic can be applied to the leader selection process. To stop the network from constantly switching leaders due to small changes in health data, the LEAD module uses a stability threshold, γ_{min} . When a maximum health candidate ($b = \arg \min_i J_i$) is found, the system calculates its percentage improvement over the current leader (L_{t-1}^*) using

$$\Delta\% = \left(\frac{J_{L_{t-1}^*} - J_b}{J_{L_{t-1}^*}} \right) \times 100. \quad (2)$$

Note that if $J_{L_{t-1}^*} = 0$ (i.e., no agent reports a valid health metric), the leader selection process terminates, marking a failure of LEAD for the current round. The algorithm subsequently re-initializes during the next election cycle.

The past leader is replaced if the maximum health candidate is better by more than a threshold ($\Delta\% > \gamma_{min}$). If it does not meet this threshold, the past leader continues ($L_t^* = L_{t-1}^*$), and the maximum health candidate is named the designated survivor ($S_t^* = b$). If the maximum health candidate beats the threshold and becomes the new leader, the old leader becomes designated survivor ($S_t^* = L_{t-1}^*$), if its health is better than the second-best candidate ($s = \arg \min_{i \neq b} J_i$). This rule ensures the swarm only switches leaders when there is a significant benefit to the team. In degenerate cases where fewer than two connected agents exist ($|\mathcal{V}| < 2$), S^* is set to n/a, and leader operates without a designated survivor.

Unique Edge Weights – A strict requirement of appointer election is that edge weights in GHS across the graph $\mathcal{G}$ are strictly unique. These weights $w_{i,j}$ are constructed by formatting agent identifiers into zero-padded, fixed-width strings and concatenating them with

the higher ID placed first.

$$w_{i,j} = \text{fmt}(\max(i,j)) \parallel \text{fmt}(\min(i,j)), \quad (3)$$

where $\parallel$ denotes string concatenation and $\text{fmt}(\cdot)$ formats the ID into a fixed-width string. Using fixed-width ensures that string boundaries remain unambiguous, guaranteeing an injective mapping across all node pairs. For example, given agents 2, 3 and 2-digit zero-padding, yields a unique edge weight $w_{2,3} = w_{3,2} = "0302"$.

Algorithm 1 One Round of Proposed LEAD Algorithm

```

1:  $\mathcal{G} \leftarrow \{\mathcal{A}_o, \mathcal{E}_o\}$   $\triangleright \mathcal{E}_i$  updated on each agent,  $n_i$ 
2:  $s_i \leftarrow \{i, 0, \infty\}$ ,  $\forall i$  and  $w_i \leftarrow w_{i,0}$ ,  $\forall i$ 
3:  $J_i \leftarrow$  based on Eq. (1)
4:  $\triangleright$  Appointer election, via GHS:
5: while agent  $n_i$  is not converged do
6:    $\text{ReceiveFromNeighbors}(\mathcal{M}_i) \rightarrow \mathcal{Q}_i$ 
7:    $\mathcal{M}_i, s_i \leftarrow \text{GhsRound}(\mathcal{Q}_i, \mathcal{E}_i)$ 
8:    $\text{SendToNeighbors}(\mathcal{M}_i)$ 
9: end while
10:  $\triangleright$  Leader selection:
11: if  $\text{IsAppointer}(n_i)$  then
12:    $\text{Broadcast}(\text{RequestCost})$ 
13: end if
14: while  $\neg \text{DeclareLeaderSurvivor}$  received do
15:   if  $\text{type}(\mathcal{M}_i) = \text{RequestCost}$  then
16:      $\text{Broadcast}(\text{RequestCost})$ 
17:      $\text{Convergecast}(i, J_i)$   $\triangleright$  Forward up local cost
18:   else if  $\text{type}(\mathcal{M}_i) = \text{RecvCost}(j, J_j)$  then
19:     if  $\text{IsAppointer}(n_i)$  then
20:        $\triangleright$  Agent  $n_i$  stores health  $J_j$  for agent  $n_j$ 
21:        $\text{StoreCost}(j, J_j)$ 
22:     if  $\text{ReceivedAllCosts}()$  then
23:        $b \leftarrow \arg \max_{k \in \mathcal{A}} J_k$   $\triangleright$  Best current candidate
24:       if  $\text{quiescence\_enabled}$  then
25:         if  $J_{L_{t-1}^*} == 0$  then
26:           break  $\triangleright$  Leader selection fails
27:         end if
28:       if  $\text{CheckCost}(J_b, J_{L_{t-1}^*}, \gamma_{\min})$  then
29:          $L_t^* \leftarrow b$  and  $S_t^* \leftarrow L_{t-1}^*$ 
30:       else
31:          $\triangleright$  Persistence: retain past leader
32:          $L_t^* \leftarrow L_{t-1}^*$  and  $S_t^* \leftarrow b$ 
33:       end if
34:     else
35:        $\triangleright$  Select new leader with max health
36:        $L_t^* \leftarrow b$  and  $S_t^* \leftarrow \arg \max_{k \in \mathcal{A} \setminus \{b\}} J_k$ 
37:     end if
38:      $\text{Broadcast}(\text{DeclareLeaderSurvivor}(L_t^*, S_t^*))$ 
39:     break  $\triangleright$  Appointer completes selection phase
40:   end if
41: else
42:    $\text{Convergecast}(j, J_j)$   $\triangleright$  Forward cost to parent
43: end if
44: end if
45: end while

```

This construction guarantees that edge weights $w_{i,j} = w_{j,i}$ are strictly unique, symmetric, and lexicographically ordered across any pair of agents (n_i, n_j) . More complex edge-weighting metrics remain an avenue for future work.

Message Delivery: FIFO—This is a requirement imposed on the network layer. It is assumed 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 wall clock time, a *timeout* is imposed on message delivery, in the form of a convergence threshold. When a timeout is triggered on the k th round, all agents stay idle until the next reelection is triggered, i.e., the $(k+1)$ th round. At every reelection, the agent graph $\mathcal{G}$ is reinitialized based on available neighbors $\mathcal{E}_i$ provided by a communications subsystem. Messages are labeled with the round ID k ; messages from previous rounds are ignored if received.

This timeout also applies to leader selection. Upon receiving the **DeclareLeaderSurvivor** broadcast, agents immediately adopt the selected leader for round k . Nodes failing to receive the declaration before the timeout do not commit, and eventually transition to the $(k+1)$ th round. Periodic reelections ensure that such agents safely synchronize and adopt the leader in the next round.

Wake Up and Listening—Ideally, the wake up sequence that triggers the appointer election 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. However, the system clocks on different agents might experience drift, δt as illustrated in Fig. 3. We desire reelections at $(t - t_{\text{epoch}}) \% \Delta_{\text{reelect}} = 0$ where Δ_{reelect} is the desired reelection period, and t_{epoch} is a synchronized epoch time with $t_{k,\text{reelect}} = t_{\text{epoch}} + k\Delta_{\text{reelect}}$, $\forall k$. 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 or $\frac{2}{f}$ between any two agents. For wakeup, this means that agents may not be perfectly synchronized, Fig. 3 (a); however, periodic queries eventually get all agents to an initialized, listening state Fig. 3 (b).

Complexity Analysis — The algorithm's overhead is measured across both message and time complexity:

- **Message Complexity:** The GHS phase requires $O(|\mathcal{E}| + N \log_2 N)$ messages for a graph $\mathcal{G}$, where $N = |\mathcal{A}|$ [7]. During leader selection, **RequestCost**, **RecvCost**, and **DeclareLeaderSurvivor** each traverse $N - 1$ edges of the spanning tree, contributing $3(N - 1) = O(N)$ messages. Thus, the message complexity per round is $O(|\mathcal{E}| + N \log_2 N)$.
- **Logical Time Complexity:** Evaluated as asynchronous rounds (where a round spans the maximum message delay in a causal chain) without requiring bounded physical delivery times, the GHS phase completes in $O(N \log_2 N)$ rounds². The broadcast and convergecast of leader selection require bounded delay proportional to the depth of spanning tree

²For ease of implementation and verification, we omit the GHS modification proposed in [19] that results in linear time complexity; the performance impact on a system of CADRE's size is negligible.

$D \leq N - 1$, yielding $O(D)$ execution time. Thus, the time complexity is $O(N \log_2 N + D) = O(N \log_2 N)$.

The reelection period is to be chosen with these complexities in mind to ensure that a LEAD round completes comfortably before the next round gets triggered.

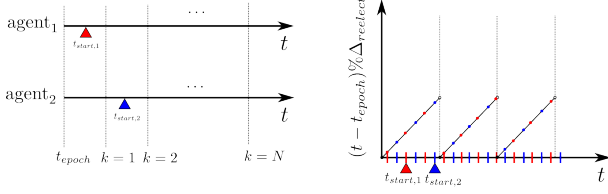


Fig. 3. Agent wake up queries are performed assuming a hardware-driven periodic timer. Agents desire reelection at every k -th reelection interval (left), but agents might start ahead of or behind the ground truth clock due to the drift in the system clock. Regardless, queries can be made at the timer frequency f , providing a trigger if the k -th reelection period has been passed (right).

No Agents Drop During Execution — A static graph $\mathcal{G}$ is assumed during any LEAD round. However, 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 via periodic reelections, at reelection period $\Delta_{reelect}$.

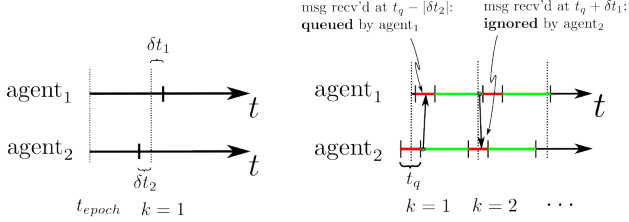


Fig. 4. Here, agent₁ is delayed relative to the ground truth clock (δt_1); agent₂ is ahead of time (δt_2), (left). On the right, time-bounded out-of-sync messages are dealt with by adding a quiescent period t_q (red), during which messages may be received, but are not sent. Messages with old timestamps are ignored within the quiescent period; successive rounds are indicated by k .

Another complication arises due to clock drift δt_1 and δt_2 between any two agents, Fig. 4 (a). Delays due to query rate, clock drift, and message-relaying time can cause relative drift between agents when repeated leader election cycles are triggered. This is problematic because, as shown in Fig. 4 (b), messages from a prior round k might enter the $(k+1)$ th round on another agent. 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, that is, tagged as being from the $(k+1)$ th round. The quiescent period, along with the reelection and rate-driven queries are summarized in Algorithm 2. To ensure that prior-round messages clear the network before the next round begins, t_q must satisfy the additive bound combining maximum propagation delay and worst-case relative clock skew:

$$t_q \geq \sum_m t_{\text{delay},m} + (\delta t_1 + \delta t_2) \quad (4)$$

Assuming known, finite delay bounds and a functioning transport layer, this condition ensures complete delivery and isolation between consecutive rounds.

Algorithm 2 Periodic Re-election and Quiescence

```

1: StartRound( $Q_i$ ),  $t_{\text{elapsed}} \leftarrow 0$ ,  $t_{\text{elapsed,last}} \leftarrow \Delta_{\text{reelect}}$ 
2: for ClockTick do
3:    $t_{\text{elapsed}} \leftarrow (t - t_{\text{epoch}}) \% \Delta_{\text{reelect}}$ 
4:   if  $t_{\text{elapsed}} < t_{\text{elapsed,last}}$  then
5:     if  $t_{\text{elapsed}} > t_q$  then
6:       LEAD round ▷ Run Algorithm 1
7:       else if  $t_{\mathcal{M}_i} > t$  then
8:          $Q_i \leftarrow \mathcal{M}_i$  ▷ Sender clock ahead: Add
9:       else  $\emptyset \leftarrow \mathcal{M}_i$  ▷ Sender clock behind: drop
10:      end if
11:    end if
12:     $t_{\text{elapsed,last}} \leftarrow t_{\text{elapsed}}$ 
13: end for

```

V. RESULTS AND ANALYSIS

The LEAD algorithm was evaluated to show: (1) robustness to agent dropouts; (2) leader transitions due to variations in agent health, leader persistence or network (sleep and wakeup cycles). Validation was performed in the context of CADRE via experiments on three-agent DMs and four-agent FMs. Note that a bidirectional communication, and a fully-connected topology among active agents is observed at all times. A megabit-class communication is enabled among CADRE rovers and base station via mesh network radios, utilizing single- or multi-hop routing as required. The clocks onboard CADRE rovers are not synchronized to avoid time jumps in onboard pose estimation while driving. LEAD is integrated into the CADRE's FSW and executed as independent processes on each agent's Snapdragon 821 board (running Linux for ARM64). The FSW is based on F' (F-Prime), a free and open-source framework that is tailored to small-scale systems such as CubeSats, and has been implemented for the Ingenuity helicopter demonstration on Mars [8].

Two health metrics are utilized: State of Charge ($\text{SoC} \in [0\%, 100\%]$) and CPU temperature ($T_{\text{CPU}} \in [-25^\circ\text{C}, 80^\circ\text{C}]$). Within these bounds, an SoC of 100% and a CPU temperature of -25°C represents best health. As per Eq. (1), these raw metrics are normalized to $h_{i,\text{SoC}}, h_{i,T} \in [0, 1]$ relative to their optimal baselines are

$$h_{i,\text{SoC}} = \frac{\text{SoC} - 0}{100 - 0}, \quad h_{i,T} = 1 - \frac{T_{\text{CPU}} - (-25)}{80 - (-25)}. \quad (5)$$

Also, they are equally weighted with $\alpha_i = 0.5$ to compute the cumulative health metric. For example, given an $\text{SoC} = 10\%$ and $T_{\text{CPU}} = 25^\circ\text{C}$, the health metric $J_i = 0.574$. Also, $\gamma_{\min}$ is set to 20% for triggering a leader change.

A. Robustness to Agent Dropouts

Given a four-agent network and with leader persistence off, agents are dropped one-by-one, starting with n_4 and progressing through n_3 and n_2 . Total message counts exchanged are shown in Fig. 5. The average number of messages exchanged decreases as $|\mathcal{G}|$ shrinks, ultimately reaching zero when agent n_1 becomes the *de facto* leader.

TABLE I

REGRESSION TESTING FOR PROPOSED LEAD ALGORITHM ON THREE-AGENT DMS AND FOUR-AGENT FMS. OPTIMAL CONVERGENCE AND NOMINAL PERFORMANCE DURING LEADER CHANGES WERE SUCCESSFULLY VERIFIED ACROSS 1200 REELECTION CYCLES.

Configuration Details				Results		
Test	Event Sequence	Health (ordered by ID) (SOC %, temp °C)	Leader persist	(Leader, Survivor)	Approx No. Cycles	Performance Analysis
DM-T1	Three agents active the entire time; only health metrics varied	{100, 25}, {50, -25}, {50, -25} {50, -25}, {100, 25}, {50, -25} {50, -25}, {40, 25}, {50, -25} {48, -25}, {40, 25}, {50, -25} {53, -25}, {40, 25}, {52, -25}	No No No No No	{A1,A2} {A2,A1} {A1,A3} {A3,A1} {A1,A3}	200	With persistence off, leader and survivor are always the best health and second best even when differences are minor; resulting in frequent leadership changes
DM-T2	Three agent active at start; A2 toggles to sleep for ≈ 5 mins and is later back to active	{100, 40}, {70, -25}, {40, 70} {10, 40}, {100, -25}, {40, 70} {100, 40}, {40, 70}, {70, -25}	Yes Yes Yes	{A1,A2} {A2,A3} {A3,A1}	250	When A2 is significantly better, leader switches from A1 even with persist ON; When A2 goes to sleep, leader switches to next best A3
DM-T3	Three agents active, all at same CPU temperature but minor differences in SOC	{49, -25}, {50, -25}, {51 - 25} {51, -25}, {50, -25}, {51 - 25} {55, -25}, {50, -25}, {51 - 25}	Yes Yes Yes	{A2,A3} {A2,A1} {A1,A2}	250	Though A1—51% > A2—50%, it's $\Delta\% < \gamma_{min}$, hence, leader persists to minimize overhead. When A1 SOC further increases, then leader switches. Since no persistence for survivor, always the next best.
DM-T4	Three agents active at start; sequential sleep trigger A1→A2→A3; then all brought back to active	{60, -25}, {70, -25}, {51 - 25}	No	{A2,A1}	100	Given asynchronous sleep, each agent sequentially loses a peer; last leader and survivor elected before A1 sleeps is {A2,A1}, before A2 sleeps is {A2,A3} and lastly, before A3 sleeps is {A3,n/a}
DM-T5	Three agents active with reelection and quiescent periods varied {5, 1}, {1, 1}, {3, 0.1}, {10, 0.1}, {10, 0.01}	{80, -25}, {90, -25}, {51 - 25}	No	{A2,A1}	100	As expected, no convergence with warning when reelection period < 3s; insufficient time for GHS convergence; Quiescent period < 1s converges but not advised; unstable if clocks poorly synchronized.
FM-T1	Four agents active the entire time; health varied; persistence toggled; asynchronous sleep triggered	{90, 40}, {90, 50}, {90, 50}, {90, 50} {50, 40}, {80, 0}, {80, 50}, {80, 50} {50, 40}, {50, 0}, {70, -10}, {70, 50} {50, 40}, {50, 0}, {50, -10}, {60, -25}	No No No Yes	{A1,A2} {A2,A3} {A3,A2} {A3,A4}	100	When persist OFF, leader switches A1→A2→A3; when persist ON, A3 continues as leader while the one with best health A4 becomes survivor; when asynchronous sleep triggered as A1→A4→A3 then leader switches as A4, A3, A2, respectively
FM-T2	Four agents active with default settings	{90, 40}, {90, 50}, {90, 50}, {90, 50}	Yes	{A1,A2}	200	Nominal operations were executed to validate baseline system performance on CADRE rovers.

Initially, SOC = 100% on n_4 and equal CPU temperature on other agents, electing n_4 as the initial leader. Reelections continue to perform, during which after each agent drops, the selected leader progresses as: $n_4 \rightarrow n_1 \rightarrow n_1 \rightarrow n_1$. Note that n_1, n_2, n_3 have equal health metrics, hence, the appointer (in this case n_1) is selected as a tiebreaker. Message counts are inclusive of both messages received during GHS (stage 1) and leader selection (stage 2).

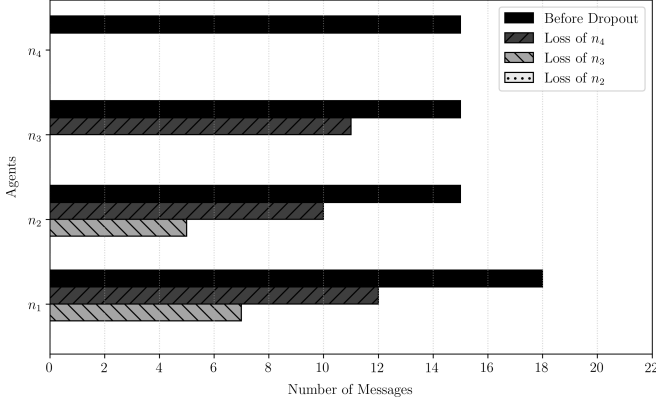


Fig. 5. Messages processed by each agent after successive reelection cycles following the loss of an agent in $\mathcal{G}$. This includes messages from both appointer election and leader selection.

B. Regression Testing: Leader Changes

Table I summarized the regression testing results executed across 1200 reelection cycles. Consistent performance is observed across both three-agent DMS and four-agent FMs. Results validated that the leadership transitions occur dynamically on minor health variations when persistence is disabled, whereas enabling persistence minimizes system overhead until minimum change in health thresholds ($\Delta\% \geq \gamma_{min}$) are exceeded. Opera-

tional boundary tests confirm that network dropouts are handled robustly via localized sequential elections.

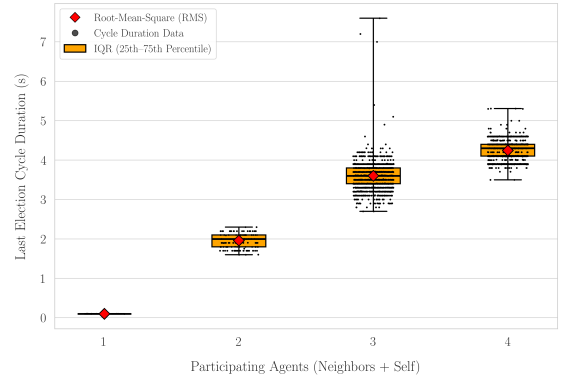


Fig. 6. Election cycle durations across participating agents (from DMS and FMs) with a 10 Hz rate query. Red diamonds denote Root-Mean-Square (RMS) cycle duration, that goes from 0.1 s (247 cycles) for one agent, to 1.9 s (138 cycles) for two agents, 3.6 s (2413 cycles) for three agents and 4.2 s (1920 cycles) for four agents.

A few notable special cases are explained as follows: In DM-T4, the sequential sleep process is explained where A3 cannot select a designated survivor as A1 and A2 gone to sleep by then. Interestingly, during subsequent wakeup, post-startup telemetry issues caused A2 and A3 to report invalid SOC, triggering a fail-safe that zeroed their health ($J_{A2} = 0$ and $J_{A3} = 0$), thus, making these agents less desirable. Consequently, A1 gets elected as the leader until valid health data was restored and leadership reverted to A2 (as was before asynchronous sleep). In DM-T2, a temporary communication glitch caused a message dropout from A3 to A1, resulting in convergence failure for one cycle. The system safely retained the previously elected leader until convergence was achieved

in the next reelection cycle 10 s later. In test case FM-T2, although A_4 is the maximum health candidate ($J_4 = 0.8$), the past leader A_3 ($J_3 = 0.679$) was retained as the leader. This is because the margin of improvement for re-electing A_4 is 17.89%, which falls below the $\gamma_{min} = 20\%$.

Some GHS non-convergence warnings were logged during tests; however, these are benign and show expected behavior. They are caused during an initialization, expected timing mismatch where communication links are active but the agents have not activated “LEAD” mode yet, i.e., they are awake but not participating in leader election. During this, both appointer election and leader selection are in temporarily disabled state.

C. Analysis of Election Cycle Durations

In Fig. 6, LEAD demonstrates a bounded election duration across small number of hardware, maintaining stable consensus as local network links increase. From a system-level verification, these empirical benchmarks satisfy operational constraints identified for CADRE lunar rovers. Specifically, the maximum observed execution time is governed by wall-clock latency, where transmission and queueing delays sum across the tree depth D . For CADRE rover swarms ($N \leq 4$), empirical execution remains well below the system’s hard convergence timeout threshold (default = 9 s), which in turn is strictly bounded by the nominal leader reelection period (default = 10 s). While worst-case retransmission delays produced two outlier peaks of 7 s during $N = 3$ testing, all tests successfully converged within the timeout, preserving an acceptable operational margin. This ensures that cyclic scheduling overruns are prevented and sufficient margins exist to absorb unexpected network message delays during operations, thereby confirming that physical latency constraints are satisfied alongside theoretical bounds.

VI. CONCLUSION

A dynamic, distributed, persistent leader election algorithm “LEAD” has been proposed for dynamic agent networks to select a unique leader for connected networks (or distinct leaders for isolated partitions during network disconnections). While retaining foundational GHS assumptions within individual rounds, LEAD employs an operational wrapper to abort and retry at a later round when failures occur. LEAD has been designed for resilience against bounded clock drift, frequent leadership changes with marginal benefit, and agent dropouts. By utilizing a rate-driven query process, periodic re-elections are executed to adapt to network changes and varying agent health metrics, such as battery and temperature, while permitting localized elections within disconnected subgraphs. The algorithm was validated through regression testing across 1200 re-election cycles on various CADRE multi-agent hardware platforms—demonstrating reasonable message overhead and robust adaptation to changing network topologies and health. While validated at the scale of CADRE lunar rovers ($N \leq$

4), future work includes evaluating larger-scale agent networks, disconnected networks, and benchmarking performance across heterogeneous computing architectures.

REFERENCES

- [1] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.
- [2] D. Alistarh and R. Gelashvili, “Recent algorithmic advances in population protocols,” *ACM SIGACT News*, vol. 49, no. 3, pp. 63–73, 2018.
- [3] S. Dolev, A. Israeli, and Moran, S., “Uniform dynamic self-stabilizing leader election,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 4, p. 424–440, Apr. 1997.
- [4] NASA, “Cooperative Autonomous Distributed Robotic Exploration (CADRE),” <https://www.nasa.gov/cooperative-autonomous-distributed-robotic-exploration-cadre/>, 2022, Accessed: 2026-09-24.
- [5] J.-P. de la Croix, F. Rossi, R. Brockers, D. Aguilar, K. Albee, E. Boroson, A. Cauligi, J. Delaune, R. Hewitt, D. Kogan, G. Lim, B. Morrell, Y. Nakka, V. Nguyen, P. Proença, G. Rabideau, J. Russino, M. S. da Silva, G. Zohar, and S. Comandur, “Multi-agent autonomy for space exploration on the cadre lunar technology demonstration,” in *2024 IEEE Aerospace Conference*, Mar. 2024, p. 1–14.
- [6] K. Albee, S. Bhamidipati, F. Rossi, J. Vander Hook, and J.-P. de la Croix, “Lunar leader: Persistent, optimal leader election for multi-agent exploration teams,” in *International workshop on autonomous agents and multi-agent systems for space applications (MASSpace)*, 2024.
- [7] R. G. Gallager, P. A. Humblet, and P. M. Spira, “A distributed algorithm for minimum-weight spanning trees,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 5, no. 1, pp. 66–77, 1983.
- [8] R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison, “F prime: an open-source framework for small-scale flight software systems,” 2018.
- [9] A. Clark, L. Bushnell, and R. Poovendran, “On leader selection for performance and controllability in multi-agent systems,” in *2012 IEEE 51st IEEE conference on decision and control (CDC)*. IEEE, 2012, pp. 86–93.
- [10] F. Lin, M. Fardad, and M. R. Jovanović, “Algorithms for leader selection in stochastically forced consensus networks,” *IEEE Transactions on Automatic Control*, vol. 59, no. 7, pp. 1789–1802, 2014.
- [11] Y. Dieudonné, F. Petit, and V. Villain, “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, 2010, pp. 267–281.
- [12] S. Dolev, A. Israeli, and S. Moran, “Uniform dynamic self-stabilizing leader election,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 4, pp. 424–440, 1997.
- [13] O. Dagdeviren and K. Erciyes, “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, 2008, pp. 509–518.
- [14] B. S. Chlebus, D. R. Kowalski, and A. Pelc, “Electing a leader in multi-hop radio networks,” in *International Conference On Principles Of Distributed Systems*. Springer, 2012, pp. 106–120.
- [15] H. Garcia-Molina, “Elections in a distributed computing system,” *IEEE Trans. Computers*, vol. 31, no. 1, pp. 48–59, 1982.
- [16] L. Lamport, “The part-time parliament,” in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317.
- [17] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 305–319.
- [18] N. A. Lynch, *Distributed Algorithms*, ser. The Morgan Kaufmann Series in Data Management Systems. San Francisco, Calif: Morgan Kaufmann, 1997.
- [19] B. Awerbuch, “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*, 1987, pp. 230–240.