Churn-tolerant Leader Election Protocols

Jiangran Wang
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Champaign, IL, USA
jw22@illinois.edu

Indranil Gupta
Department of Computer Science
University of Illinois at Urbana-Champaign
Champaign, IL, USA
indy@illinois.edu

***Abstract*—Classical leader election protocols typically assume complete and correct knowledge of underlying membership lists at all participating nodes. Yet many edge and IoT settings are dynamic, with nodes joining, leaving, and failing continuously—a phenomenon called *churn*. This implies that in any membership protocol, a given node's membership list may have entries that are missing (e.g., false positive detections, or newly joined nodes whose information has not spread yet) or stale (e.g., failed nodes that are undetected)—these would render classical election protocols incorrect. We present a family of four leader election protocols that are churn-tolerant (or c-tolerant). The key ideas are to: i) involve the minimum number of nodes necessary to achieve safety; ii) use optimism so that decisions are made faster when churn is low; iii) incorporate a preference for electing healthier nodes as leaders. We prove the correctness and safety of our c-tolerant protocols and show their message complexity is optimal. We present experimental results from both a trace-driven simulation as well as our implementation atop Raspberry Pi devices, including a comparison against Zookeeper.**

***Index Terms*—Leader Election, Membership, Edge Computing, Churn**

I. Introduction

Driven by the promise of autonomous operation, self-adaptability, low latency, and bandwidth [1], the global edge computing market size is expected to reach 155.9 billion USD by 2030 [2], with over 29 billion IoT devices by 2030 [3]. Edge computing scenarios range from "stable" ones like smart homes, Industry 4.0, and infrastructure deployments (e.g., smart bridges, roads, etc.), to "dynamic" settings—examples include robots in remote or inhospitable terrains (e.g., emergency rescue and recovery scenarios, battlefields, smart farms) [4], [5], [6], to constellations of satellites (low earth LEO, or medium earth MEO) [7], [8], to fast-moving vehicle platoons [9]. While a bulk of research today deals with the former stable edge settings, the rapid recent growth of the latter dynamic settings brings forth many problems that remain unsolved.

This paper tackles the classical problem of leader election. A leader election protocol can be initiated by any one node, and aims to satisfy both: 1) (Safety) elect a unique leader that has the single "best attribute" (e.g., the lowest hash ID, or the most amount of data, etc.), and 2) (Liveness) inform all non-faulty nodes of this single unique leader's ID. Leader election is a key building block for coordination among nodes in dynamic edge settings, especially when remoteness means no central hub or infrastructure is present nearby to perform coordination actions. For instance, robots in rescue and recovery scenarios, or smart farms [6], need a leader for coordination when no human operator is nearby. A second example—LEO or MEO satellites [7] move very fast (finishing an orbit in under 2 hrs), so the set of satellites above a geographical area (e.g., a state) is a dynamic group and yet needs a leader satellite for coordination actions (since ground stations are sparse and may be absent in that area). A third example is vehicle platoons using a leader to achieve autonomous caravan driving [9].

Unfortunately, classical solutions to leader election [10], [11], [12], [13], [14], [15], [16], [17], [18] assume a "stable" membership (i.e., "strongly-consistent membership"), wherein every node knows all other nodes, and if any node joins, leaves, or fails, the underlying membership protocol *sends an instantaneous update* to all non-faulty nodes. However, inconsistencies in membership lists are unavoidable in asynchronous edge systems since an update (node join, leave, or failure) cannot instantly propagate to all nodes: network latencies are non-zero and vary, while timeouts for failure detection (e.g., via heartbeating [19] or ping-ack protocols [20], [21]) are not synchronized across nodes.

We aim to solve leader election in such *churned* edge environments where nodes continuously join, leave, and fail (by crashing only [1]). Concretely, our leader election protocols can be layered atop an arbitrary *weakly-consistent* membership protocol such as gossip-style heartbeating [19], SWIM [20], or Medley [21]. These weakly-consistent membership protocols detect failures and spread membership updates quickly, yet they only provide *eventual* consistency guarantees, i.e., a membership update (join, leave, or failure) is received only eventually at all non-faulty nodes, and in an arbitrary order. Due to their scalability, weakly-consistent membership protocols are an appropriate choice for edge computing settings.

The problem of churn-tolerant leader election is non-trivial. Given strongly-consistent membership, a group can quickly elect a leader: each node uses a consistent and local hash function (e.g., SHA3, MD5, etc.) on all IDs present in its local membership list and selects as leader node whose ID has the lowest hash [22], [10], [21]. With identical and correct membership lists (strong), everyone elects the same leader, without extra messages. However, if membership lists are

[1]Fail-stop model only. This paper does not consider Byzantine failures.

inconsistent (weak), the would-be leader may be unknown at some nodes, thus multiple leaders may be elected, violating safety. Thus a new churn-tolerant variant of this protocol is needed. (For simplicity the rest of this paper uses the lowest hash ID as the leader selection criteria, but this can be replaced by an arbitrary attribute of choice, and our results still apply.)

The contributions of this paper are:

- We propose a progression of four variants for churn-tolerant (or *c-tolerant*) election that (respectively) involve the fewest messages, have *optimism* by completing quickly in the common (consistent) case, accommodate a "health" *preference* for the leader, and a hybrid optimistic-preference variant.
- We prove Safety and Liveness of all our protocols, and analyze the optimality of their message complexity.
- We present simulation results of our churn-tolerant election protocols, where we inject traces from the Medley membership protocol [21].
- We implemented our churn-tolerant election protocols in Raspberry Pi devices, and we present results from a lab deployment, and we show that our protocols consume far fewer resources than stock solutions like Zookeeper [11].

The rest of the paper is organized as follows. Sec. II presents background and related work. Sec. III presents our system model. Sec. IV presents our c-tolerant election protocols. Sec. V analyzes them formally. Sec. VI presents simulation results, and Sec. VII presents Raspberry Pi deployment results.

II. Background and Related Work

A. Membership Protocols

There are two major kinds of membership protocols in asynchronous distributed systems—strong and weak.

Strongly-Consistent Membership: The strictest variants assume that membership lists at each alive node are identical at all times. This implies that membership updates are delivered to all nodes at the exact same time, which is impossible to achieve in asynchronous systems [23], [24]. A slightly relaxed variant (within strong membership) assumes that membership updates will reach each node in the same order, but possibly at different physical arrival times. Examples include virtual synchrony [25], Raft [13], Zookeeper [11], and others [26], [27]. Yet the overhead of strongly-consistent membership protocols scales poorly with churn, rendering nodes unable to do useful work. For instance, this is why a typical Zookeeper cluster only contains between 3 and 7 servers.

Weakly-Consistent Membership: A weakly-consistent membership propagates membership updates to all nodes eventually, but it does not guarantee either ordering or timing of the updates. Examples include gossip-style heartbeating [19], SWIM [20], and other IoT membership protocols [21]. These protocols scale better than strongly-consistent membership protocols and have been used in systems with thousands of participating nodes [28], [29]. However, membership inconsistencies create a mismatch with the desire to provide a strong safety property above it. Concretely, a node M_i could be missing in other nodes' membership lists if M_i just joined or it is mistakenly detected as failed (false positive) by some other nodes. A node M_i could also falsely exist at another node's membership list but M_i is actually failed, as failure detection is not instantaneous.

Membership in Edge Settings: Edge settings are better suited to weakly-consistent membership protocols. Strongly-consistent membership protocols have difficulty scaling in asynchronous systems as they incur high bandwidth and tend to splinter the node group beyond a few 10s of nodes [30], [25], [23], [31]. Weakly-consistent membership protocols [32], [33], [21] only guarantee eventual delivery of membership changes without order, but they scale well and use low bandwidth. Hence all election protocols presented in this paper are layered over a weakly-consistent membership.

B. Related Work

Broadly, the topic of layering consistent services over inconsistent substrates has received recent attention. [34] implements transactional systems on top of inconsistent replication in distributed systems, and [31] supports virtual synchrony over gossiping. The key idea in [34] and [31] are similar: they both *reactively* try to fix inconsistency issues raised from lower layers, as opposed to our (membership-based) approach which is *proactive*. [35] builds a microservices OS over inconsistent networks by integrating strong consistency properties into the membership layer itself. Neither of the aforementioned papers builds over inconsistent membership lists, nor did they propose fully-distributed election protocols. Some work [10], [36] provides *probabilistic* safety property for distributed protocols atop weak membership, while we provide 100% safety. Ad-hoc routing [29] uses gossip [30], [37] to spread information fast, but requires flooding to maintain safety. To the best of our knowledge, our work is the first to explore the challenges and mismatches of layering strongly-consistent distributed protocols directly over weak membership layers.

Churn is a common phenomenon in peer-to-peer systems and past work has built distributed hash tables and applications that are churn-tolerant [38], [39], yet the notions of safety provided therein are merely probabilistic or best-effort (while we provably guarantee safety). Classical leader election solutions such as the Bully Algorithm [14] or consensus-based election protocols like Zookeeper [11], Paxos [12], and Raft [13], assume full and correct membership. Election in mobile ad-hoc networks that are subject to partitions exists [15], [16], yet they assume FIFO links and consistency within each partition (these partition-tolerant techniques can be applied orthogonally to us as we do not consider partitions). Self-stabilizing leader election [40], [41], [42] aims to recover from transient faults, but they assume that the underlying "overlay" (ring, tree, etc.) stabilizes after failures. We assume no membership convergence, and our membership lists can be inconsistent all the time.

III. System Model and Problem Statement

A. System Model

We assume an asynchronous system where messages have arbitrary delay but are delivered eventually (without a known bound), and nodes may crash by failing. Like classical literature, we assume a known upper-bound f on the number of simultaneously failed nodes (after the protocol quiesces, further f nodes may fail).

We assume a weakly-consistent membership protocol [19], [20], [21] runs in the background at each node and continually updates its local membership. Each node in the system can join, leave, or fail (crash) arbitrarily, and the membership protocol only provides the property that such updates are eventually propagated to non-faulty nodes. We also assume multicast is eventually reliable, e.g., via gossip [43] or R-multicast [44]. To summarize: in the rest of this paper, a *churned setting* means a weakly-consistent membership and eventually reliable multicast running at each node.

Denote the total number of nodes in the system as N. Edge and IoT systems naturally know N. This assumption is backed up by studies—even in high-churn peer-to-peer systems, studies have shown that the number of alive nodes remains stable in spite of churn [45], [46].

Now, let a given node M_i be missing in the membership lists of c_i other nodes. We assume that the maximum of all c_i values is bounded from above by a known value c (analogous to the classical assumption of f failures), i.e.,

$$max(\{c_i | 1 \leq i \leq N\}) \leq c$$

We aim to design an election protocol that provides safety for a given maximum value of c—we call such an election as a *c-tolerant election* protocol (or more precisely *(c,f)-tolerant*). Our notion of *c-tolerance* handles missing entries in membership lists, thus generalizing the traditional *failure (f)-tolerant protocol* that deals with extra entries (nodes that are failed but not yet detected).

In practice, values of c are in fact small. Table I shows 5-minute runs of the Medley failure detector [21] with 3 different topologies, system sizes, and message drop rates. (Experimental settings are in Sec. VI.)

We observe that the calculated $c(= max(\{c_i | 1 \leq i \leq N\}))$ values are small. For message loss rates at or under 5%, c values never exceed 10% of N. Hence we assume the membership graph is strongly connected, not partitioned, and that c is small.

B. Leader Election Problem

An election run must satisfy the following two properties:

Definition 1. ***Liveness:*** *Protocol terminates, i.e., each alive node eventually elects a leader (sets its local leader variable to a non-null value).*

Definition 2. ***Safety:*** *At the end of the leader election, each alive node only sets its local leader variable to that unique non-faulty node which has the lowest hash of all non-faulty nodes currently in the system.*

TABLE I: *c values (from 5-minute Medley [21] runs) stay small across different configurations*

Topology Type	Topology Size	Message Drop Rate	c
Grid	49	0.05	4
Cluster	49	0.05	4
Random	49	0.05	4
Random	49	0.1	7
Random	49	0.15	10
Random	49	0.2	13
Random	32	0.05	2
Random	64	0.05	5
Random	128	0.05	9
Random	256	0.05	17

For ease of exposition, our description of the protocols assumes a single "initiator" node. This node may be the one that detects the failure of the old leader or is the bootstrap node when the system boots up. We do not tackle initiator failure because weak membership protocol failures are eventually detected, so *some* node will eventually detect the old leader's failure and initiate an election. To handle multiple initiators—any initiator that hears of a lower ID initiator does not complete its own protocol and instead participates in the lower ID initiator's protocol, thus only the lowest ID initiator completes. In this way, multiple initiators would elect the same leader.

IV. Protocol Design

In this section, we present our four leader election protocols for churned settings.

A. Base Protocol

We first observe that in order to minimize message complexity, the initiator must communicate with the least number of nodes, to safely make a decision about who the leader is.

Assume node M_i is the *would-be (or presumptive) leader* (i.e., the non-faulty node with the lowest hash in the system, though other nodes may not know it yet). Assume no failed nodes at first. By our assumption, M_i may be missing in up to c other nodes' membership lists. Thus, our protocol has the initiator send a QUERY message to at least $(c+1)$ nodes, selected arbitrarily, each of which then sends back a RESPONSE message with its lowest known hash node (from its local membership lists). By definition, at least one of the $(c+1)$ responses will contain the would-be leader M_i.

Next, because there may be up to f failures, some of these nodes contacted by the initiator may not respond. Thus we increase the number of nodes that the initiator contacts to $(c+f+1)$ nodes (selected arbitrarily) so that the initiator receives at least $(c+1)$ responses.

Once the initiator calculates the leader, it sends the would-be leader a NOTIFYLEADER message. Upon receiving this, the leader knows it is the leader, and it multicasts a LEADER message to the entire group.

Non-responsive would-be leaders may cause election retries but do not violate Safety. We discuss two cases. First, if the would-be leader is alive but suffers temporary message losses that drop election messages to/from it, then the initiator times out and restarts the election. It still elects the same would-be leader since it is present in at least one of the $(c + f + 1)$ RESPONSE messages. If the would-be leader continues being flaky/lossy, then the failure detector will detect it as failed (at sufficient number of alive nodes), and a different leader will be elected in a subsequent election retry. Second, if the would-be leader itself crashes during the election (before it multicasts a LEADER message), and this crash is updated by the failure detector module at sufficient number of alive nodes, then the initiator restarts the election and elects the next would-be leader. In either case, Safety is maintained. Additionally, since failure detectors like Medley [21] are fast in practice, a failed would-be leader does not hinder Liveness for too long.

This *Base Protocol* is depicted formally in Algorithm 1. The TIMEOUT in line 14 is set based on the expected round-trip time so that if the would-be leader failed during the election run, a new election is started.

Algorithm 1 Base Protocol

```
1:  function InitiateElection // At Initiator
2:      id ← ∞, receivedIds ← ∅
3:      Send Query to arbitrary c + f + 1 nodes
4:      while size(receivedIds) < c + 1 do
5:          if no new Response after Timeout then
6:              Send Query to c+f+1−size(receivedIds)
    nodes excluding receivedIds
7:          end if
8:          newId ← Response from node i
9:          receivedIds.add(i)
10:         id ← min(id, newId)
11:     end while
12:     Send NotifyLeader to node id
13:     if No Leader message from id after Timeout then
14:         InitiateElection()
15:     end if
16: end function
17:
18: function RecvMessage(msg from node i) // Any Node
19:     if msg is of type Query then
20:         Send Response (← lowest hash node) to node i
21:     else if msg is of type NotifyLeader then
22:         Multicast Leader
23:     else if msg is of type Leader then
24:         Mark node i as leader
25:     end if
26: end function
```

Fig. 1 depicts an example of the Base Protocol with 4 nodes. We use $c = 2$, $f = 0$, and node 3 is the initiator.

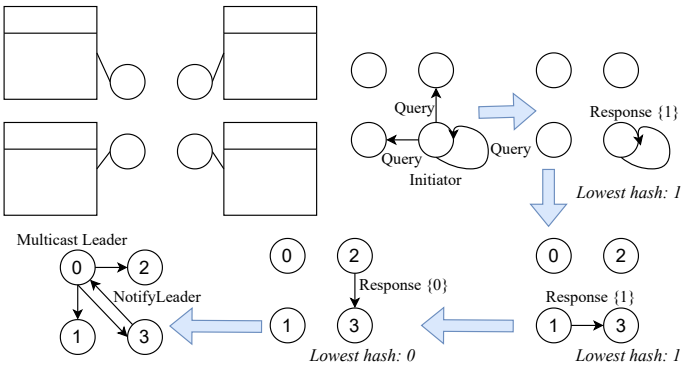


Fig. 1: *Base Protocol Example*

B. Optimistic Protocol

The Base Protocol can be slow in the common case when membership lists are *nearly* consistent. To converge quickly in this optimistic case, we present the Optimistic variant shown in Algorithm 2. The key idea is for the initiator to "stream" the leader calculation—whenever a new RESPONSE is received the leader is re-calculated and if it changes, a new NOTIFYLEADER message is sent to the newly elected leader (which then multicasts LEADER to the group). Nodes may receive multiple LEADER messages and use only the lowest hash value node as the leader.

If membership lists are nearly consistent, the Optimistic Protocol converges quickly, because one of the early RESPONSE messages received at the initiator will contain the would-be leader and further NOTIFYLEADER messages will not be sent.

Algorithm 2 Optimistic Protocol

```
1:  function InitiateElection // At Initiator
2:      id ← ∞, receivedIds ← ∅
3:      Send Query to arbitrary c + f + 1 nodes
4:      while size(receivedIds) < c + 1 do
5:          if no new Response after Timeout then
6:              Send Query to c+f+1−size(receivedIds)
    nodes excluding receivedIds
7:          end if
8:          newId ← Response from node i
9:          receivedIds.add(i)
10:         if newId < id then
11:             id ← newId
12:             Send NotifyLeader to node id
13:         end if
14:     end while
15:     if No Leader message from id after Timeout then
16:         InitiateElection()
17:     end if
18: end function
19:
20: function RecvMessage is identical to the Base Protocol
```

C. Preferred Protocol

Some applications additionally need to elect the lowest hash leader who is also *healthy*, i.e., is being suspected less often by other nodes' failure detectors as a false positive. Our third protocol, called Preferred Protocol (Algorithm 3), has each RESPONSE message contain both the top (parameterized) y unhealthy nodes and the lowest (parameterized) x hash nodes. (In our implementation, typical values are $x = y = 5$.) The health of a node can be calculated in an application-defined way: possibilities include metrics such as false positive count [19] or suspicion counts (in SWIM [20] or Medley [21])—these metrics are readily available from the membership protocol itself. The initiator calculates the set $leaders$: by taking the union of all received lowest hash ID nodes, and removing the union of all received unhealthy nodes. The initiator uses the lowest hash node from $leaders$ as the leader (the rest of the protocol remains unchanged). If $leaders = \emptyset$, the protocol retries by increasing x (GETNEXTX) and reducing y (GETNEXTY).

D. Hybrid Protocol: Combining Optimistic and Preferred

Optimism and Preference are orthogonal and our Hybrid Protocol in Algorithm 4 combines both. Whenever the initiator receives a new RESPONSE, it calculates the lowest hash node with unhealthy nodes excluded. If this results in a leader change, a new NOTIFYLEADER is sent to the newly elected leader. This protocol converges quickly if membership lists happen to be mostly consistent system-wide.

Example—Hybrid Protocol: In Fig. 2 there are 5 nodes (with respective hash values $0, 1, 2, 3, 4$), and the parameters are set as: $x \leftarrow 2, y \leftarrow 2, c \leftarrow 2, f \leftarrow 0,$ and $c + f + 1 \leftarrow 3$. The unhealthiness metric takes integer values. Consider the following snapshot of the system operation. For each line, the first number denotes which node's membership list we are looking at, and the corresponding dictionary of (key : value) pairs indicate (node : unhealthiness metric) pairs:

$$0 : \{1 : 2, 2 : 1, 3 : 4, 4 : 0\}$$

$$1 : \{0 : 5, 2 : 0, 3 : 2, 4 : 1\}$$

$$2 : \{0 : 2, 1 : 2, 3 : 3, 4 : 1\}$$

$$3 : \{0 : 3, 1 : 0, 2 : 2, 4 : 1\}$$

$$4 : \{0 : 0, 1 : 1, 2 : 0, 3 : 2\}$$

Suppose node 4 is the initiator and it sends QUERY to nodes $0, 1, 4$. They reply back with their top $y = 2$ unhealthy nodes and the lowest $x = 2$ hash nodes. Below is one possible execution outcome of the Hybrid Protocol:

- Initiator receives RESPONSE from node 4
  - $excludes \leftarrow \{3 : 2, 1 : 1\}$
  - $candidates \leftarrow \{0, 2\}$
  - $leaders \leftarrow \{0, 2\}$
  - Node 0 becomes the tentative leader and the initiator sends NOTIFYLEADER to node 0
  - Node 0 multicasts LEADER to everyone

Algorithm 3 Preferred Protocol

```
1:  function INITIATEELECTION(x, y) // At Initiator
2:      candidates, excludes, receivedIds ← ∅
3:      Send QUERY{x, y} to arbitrary c + f + 1 nodes
4:      while size(receivedIds) < c + 1 do
5:          if no new RESPONSE after TIMEOUT then
6:              Send QUERY to c+f+1−size(receivedIds)
    nodes excluding receivedIds
7:          end if
8:          {candidate, exclude} ← RESPONSE from node i
9:          receivedIds.add(i)
10:         candidates ← candidates ∪ candidate
11:         excludes ← excludes ∪ exclude
12:     end while
13:     leaders ← candidates − excludes
14:     if leaders = ∅ then
15:         x ←GETNEXTX(x), y ←GETNEXTY(y)
16:         INITIATEELECTION(x,y)
17:         return
18:     end if
19:     id ← min(leaders)
20:     Send NOTIFYLEADER to node id
21:     if No LEADER message from id after TIMEOUT then
22:         INITIATEELECTION(x,y)
23:     end if
24: end function
25:
26: function RECVMESSAGE(msg from node i) // Any Node
27:     if msg is of type QUERY then
28:         {x, y} ← QUERY
29:         exclude ← top y unhealthy nodes
30:         candidate ← x lowest hash (excludes exclude)
31:         Send RESPONSE{candidate, exclude} to node i
32:     else if msg is of type NOTIFYLEADER then
33:         Multicast LEADER
34:     else if msg is of type LEADER then
35:         Mark node i as leader
36:     end if
37: end function
38:
39: function GETNEXTX(x): return min(N, x + 1)
40: function GETNEXTY(y): return max(0, y − 1)
```

- Initiator receives RESPONSE from node 0
  - $excludes \leftarrow \{3 : 6, 1 : 3\}$
  - $candidates \leftarrow \{0, 2\}$
  - $leaders \leftarrow \{0, 2\}$
  - Node 0 remains the tentative leader
- Initiator receives RESPONSE from node 1
  - $excludes \leftarrow \{3 : 8, 0 : 5, 1 : 3\}$
  - $candidates \leftarrow \{2\}$
  - $leaders \leftarrow \{2\}$
  - Node 2 becomes the tentative leader and the initiator sends NOTIFYLEADER to node 2
  - Node 2 multicasts LEADER to everyone

Algorithm 4 Optimistic and Preferred (Hybrid) Protocol

1: **function** INITIATEELECTION(x, y) // At Initiator
2: $candidates, excludes, receivedIds \leftarrow \emptyset, id \leftarrow \infty$
3: Send QUERY$\{x, y\}$ to arbitrary $c + f + 1$ nodes
4: **while** $size(receivedIds) < c + 1$ **do**
5: **if** no new RESPONSE after TIMEOUT **then**
6: Send QUERY to $c+f+1-size(receivedIds)$ nodes excluding $receivedIds$
7: **end if**
8: $\{candidate, exclude\} \leftarrow$ RESPONSE from node i
9: $receivedIds.add(i)$
10: $candidates \leftarrow candidates \cup candidate$
11: $excludes \leftarrow excludes \cup exclude$
12: $leaders \leftarrow candidates - excludes$
13: **if** $leaders = \emptyset$ **then**
14: $x \leftarrow$GETNEXTX(x), $y \leftarrow$GETNEXTY(y)
15: INITIATEELECTION(x,y)
16: **return**
17: **end if**
18: $leaderId \leftarrow min(leaders)$
19: **if** $id\ != leaderId$ **then**
20: $id \leftarrow leaderId$
21: Send NOTIFYLEADER to node id
22: **end if**
23: **end while**
24: **if** No LEADER message from id after TIMEOUT **then**
25: INITIATEELECTION(x,y)
26: **end if**
27: **end function**
28:
29: **function** RECVMESSAGE, GETNEXTX, and GETNEXTY are identical to the Preferred Protocol

- At this point the initiator has received all the RESPONSE messages, thus the confirmed leader is node 2

V. FORMAL ANALYSIS

We formally analyze the properties of the four protocols. Readers may skip this section without loss of continuity—for such readers, we provide here a summary of our findings:

1) The Base Protocol and Optimistic Protocol both satisfy Safety and elect the lowest hash ID node as the leader. (Note that the Preferred Protocol and Hybrid Protocol may not elect the lowest hash ID leader if it is unhealthy.)
2) All our four protocols (Base, Optimistic, Preferred, Hybrid) satisfy Liveness: they complete and elect a leader.
3) The Base Protocol is optimal in message complexity.
4) The Preferred Protocol and Hybrid Protocol do not elect top unhealthy nodes as leaders, with high probability.

Theorem 1. *(Safety) Both the Base Protocol and Optimistic Protocol satisfy Safety (as defined in Sec. III-B)).*

Proof. We prove this by contradiction. Assume the would-be leader is the non-faulty node M_i (with the lowest hash) but instead, there exists at least one node that sets another node

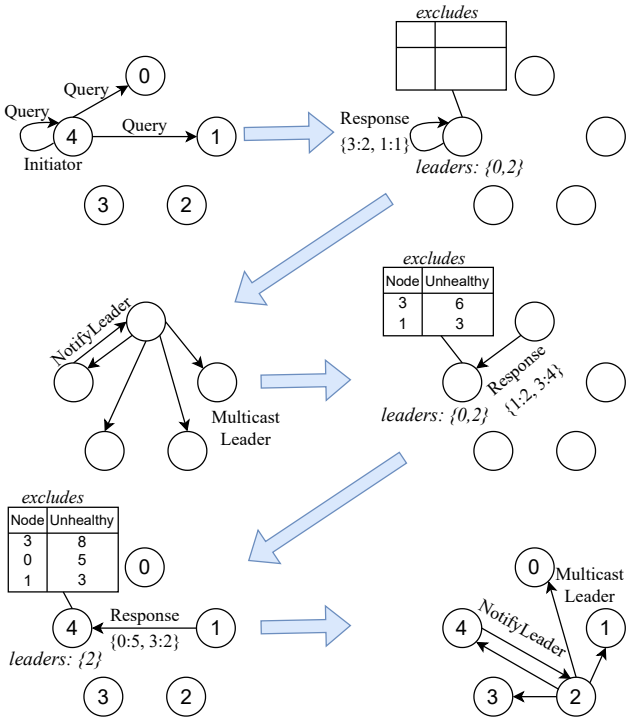


Fig. 2: *Hybrid Protocol Example*

M_j $(j \neq i)$ as its confirmed leader. This can only occur if all the initiator's received $(c + 1)$ responses contain M_j but not M_i. However, M_i is missing in c_i nodes' membership lists, and hence this implies that $c_i > c$, which is a contradiction to the definition of c (Section III-A). □

Theorem 2. *(Liveness) If the initiator and would-be leader remain alive, all our four protocols (Base, Optimistic, Preferred, Hybrid) satisfy Liveness (as defined in Sec. III-B)).*

Proof. We prove the theorem by contradiction. Assume the leader is never elected and the protocols do not finish. This may occur due to three reasons:

(1) If the initiator does not receive at least $(c+1)$ RESPONSE messages: Since the initiator sends QUERY to $(c + f + 1)$ arbitrary nodes and the maximum number of failed nodes is f, at least $(c + 1)$ nodes are alive and will send back a RESPONSE. Because messages are eventually delivered, the initiator receives at least $(c + 1)$ responses.

(2) If the initiator never receives LEADER message from the lowest hash node: Since multicast is reliable, this happens only if:

- The would-be leader is dead: in this case, the election will be restarted after the initiator times out; or
- The would-be leader does not receive NOTIFYLEADER: this cannot happen as messages are delivered eventually.

(3) For the Preferred Protocol and Hybrid Protocol, if $leaders$ set (Algorithm 3 and 4) is empty: Then the protocol will restart with the new x and y values. If this keeps reoccurring then according to GETNEXTX and GETNEXTY, eventually we have $x = N$ and $y = 0$. Then, $candidates$

contains all the N nodes and $excludes$ is empty, which makes $leaders = candidates - excludes = candidates$ non-empty. Thus, there will always be a leader elected in this case.

Hence all four protocols satisfy Liveness. □

Lemma 1. *Assume unicast is reliable and nodes don't fail. Then, the Base Protocol involves $(2 \cdot (c + f + 1) + 1)$ unicasts and 1 multicast.*

Proof. The leader multicasts LEADER only once at the end of the election process. The initiator sends $c + f + 1$ QUERY messages to arbitrary nodes, and these nodes will reply back with $c + f + 1$ RESPONSE messages. The initiator also sends another NOTIFYLEADER to the would-be leader. □

Theorem 3. *(Message Optimality) Assume unicast is reliable and nodes don't fail. Then (among all initiator-based election protocols) the Base Protocol is optimal in message complexity in order to satisfy Safety.*

Proof. For any leader election protocol, we need at least 1 multicast message to let everyone know the new leader. The single NOTIFYLEADER unicast is also necessary since the leader needs to be informed by the initiator that it is the leader. So at least 1 multicast and 1 unicast are required by any election protocol.

Next, suppose there exists a leader election protocol that works with less than $2 \cdot (c + f + 1)$ unicast messages. We prove by contradiction that this would violate Safety. Suppose only $(c + f)$ nodes are contacted by the initiator (and thus $2 \cdot (c+f)$ total unicasts). If f of these nodes fail, the initiator will only receive c replies. In the worst-case scenario, the would-be leader M_i (non-faulty node with the lowest hash) is absent in c_i node's membership lists, and these are the $c = c_i$ nodes that send RESPONSE messages to the initiator. This will result in a different node than M_i being elected as leader, thus violating Safety. Therefore, we conclude that at least $2 \cdot (c + f + 1)$ unicasts are necessary.

Together with Lemma 1, this proves the theorem. □

Since the Preferred Protocol and Hybrid Protocol may not elect the lowest hash ID leader if it is unhealthy, we define:

Definition 3. ***Preference:*** *The elected leader does not belong to the system-wide top y unhealthy nodes.*

Theorem 4. *(Preference w.h.p.) With only a logarithmic number of messages, the Preferred Protocol and Hybrid Protocol satisfy Preference with high probability (w.h.p.).*

Proof. Denote r as the number of RESPONSE messages the initiator receives. Denote p as the average percentage of nodes that belongs to system-wide top-y unhealthy nodes but do not exist in an arbitrary RESPONSE message (top-y unhealthy nodes in local membership lists). Then, the probability that a given top-y system-wide unhealthy node will be in at least one of the r RESPONSE is $1 - p^r$. To achieve this with high probability $> (1 - \frac{1}{N})$, we need $r > \log_{1/p}(N)$. This implies the Preferred Protocol and Hybrid Protocol only require a logarithmic number of RESPONSE messages to satisfy Preference w.h.p. □

Theorem 4 in Practice: Practically, we observed that the value of p stays small. Via multiple runs of Medley [21], we find that p only depends on the topology size N (and is independent of topology and message drop rate). In fact, Table II shows that p converges to 0.5 as N increases. Further, the r values never exceed $(c+1)$. Since the initiator receives at least $(c+1)$ RESPONSE messages, this suffices to satisfy *Preference* w.h.p.

TABLE II: *r and $(c + 1)$ values with different topology sizes*

Topology Size N	32	49	64	128	256
p	0.34	0.43	0.48	0.5	0.5
r	3	5	6	7	8
$c + 1$	3	5	6	10	18

VI. TRACE-DRIVEN SIMULATION

The research questions addressed by our trace-driven simulation in this section are:

1) What is the bandwidth and leader election completion time for our four c-tolerant election protocols?
2) How well do the Base Protocol and Optimistic Protocol satisfy Safety?
3) Do the Optimistic Protocol and Hybrid Protocol shorten the completion/convergence time?
4) How healthy are the leaders elected by the Preferred Protocol and Hybrid Protocol?

We wrote and tested a custom simulator incorporating our four election algorithms. Our custom simulator allows us more agility to vary parameters than stock simulators (like NS-3), and also allows us to scale simulations better. In our simulator, messages can be dropped or delayed. Because we are dealing with ad-hoc routing topologies, packets are routed via Dijkstra's shortest path protocol.

Simulations are driven by real membership traces that we collected from running a weakly-consistent membership protocol, Medley (code obtained and run from the authors of [21]). We configured the membership protocol to run with the identical network configuration (e.g., message drop rate, topology, etc.) as our leader election protocols. We collect membership traces only after its warm-up phase has finished. These traces are injected into our election protocols—our simulator continually reads the latest membership list at each node (along with information such as suspicion counts as a health metric) and updates it, and our election protocol implementation has to automatically cope with any changes.

We evaluate using three topologies: Grid topology (default 7x7 grid), Random, and Cluster (total of 5 clusters with 7, 7, 9, 10, and 16 nodes respectively). Topologies with 49 nodes are simulated in a 15m x 15m space. Topologies with other sizes have a fixed node density of around 0.22 nodes/m^2. These are

the settings that we used to generate Table I. We set the value of c accordingly based on that table.

Each plotted data point is from 100 simulation runs using different seeds and different initiators. Message delay on each hop is randomly chosen in $(0, 50]$ (time units), and the communication range is 4 meters. Default parameter values in Section IV's pseudocode are: TIMEOUT $= 500$ time units, $x = 5$, and $y = 5$.

In order to show the benefit of the Preferred Protocol and Hybrid Protocol, our network simulator has higher drop rates for messages sent and received by nodes with lower hash values. The consequence is that nodes with higher hashes will be healthier than nodes with lower hash. This forces the protocols to avoid electing the lowest hash node as the leader.

We show and discuss data for bandwidth, completion/convergence time, and leader health.

A. Bandwidth

Fig. 3, 4, and 5, show the bandwidth for our four protocols, measured as end-to-end bytes (summed across all hops for each packet's route).

Among our four protocols, we observe that while the bandwidth of the Base Protocol, Optimistic Protocol, and Preferred Protocol are similar, the Hybrid Protocol incurs relatively more bandwidth. This is because the unhealthiness information varies across nodes, and thus "top unhealthy" lists may look different at different nodes. Consequently (in the Hybrid Protocol), received RESPONSE messages at the initiator may frequently update the leader, leading to more LEADER multicast messages, and thus higher network bandwidth.

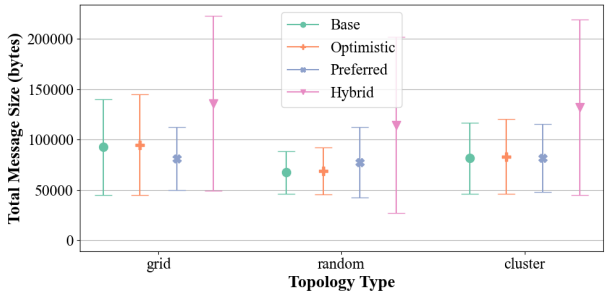


Fig. 3: *Bandwidth with different topology types*

Bandwidth rises linearly with message drop rate and system size, as expected (Fig. 4 and 5). Larger system sizes or higher message drop rates naturally require higher values for c, thus increasing bandwidth via the $(c+f+1)$ QUERY and RESPONSE messages. Across different topology types, the comparative trends do not vary much (Fig. 3).

B. Completion Time (or Convergence Time)

The completion time of leader election is defined as the difference between when the initiator initiates the election and when the last non-faulty node knows the correct leader (i.e., receives LEADER). Fig. 6, 7, and 8 show the completion time across topology types, message drop rates, and system sizes.

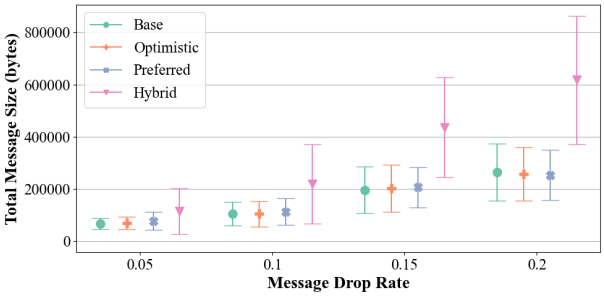


Fig. 4: *Bandwidth with different drop rates*

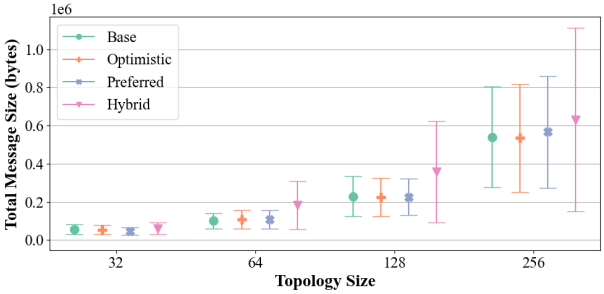


Fig. 5: *Bandwidth with different topology sizes*

We observe that the completion time of the Base Protocol and Preferred Protocol are large since both of them wait for all $(c + 1)$ RESPONSE messages before notifying the leader. On the other hand, the completion time of the Optimistic Protocol is 42.6% less than the Base Protocol on average, and the completion time of the Hybrid Protocol is 33.3% less than the Preferred Protocol on average. These small completion times of the Optimistic Protocol and Hybrid Protocol indicate they are able to naturally leverage the existing consistency across membership lists and converge faster.

Fig. 7 and 8 show that, as expected, higher message drop rates and system sizes prolong completion times. Higher drop rates mean some messages (e.g., QUERY, RESPONSE) may be resent. Large system sizes mean higher values of c and thus longer wait time for the initiator to get $(c + 1)$ responses.

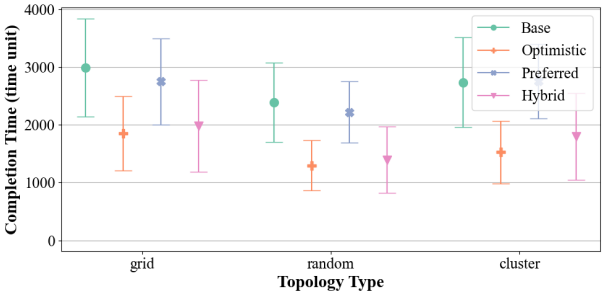


Fig. 6: *Completion time with different topology types*

C. Leader Health

Recall that the Preferred Protocol and Hybrid Protocol (Section IV) are aimed at optimizing the health of the leader, i.e., attempt to elect a leader that is not among the top

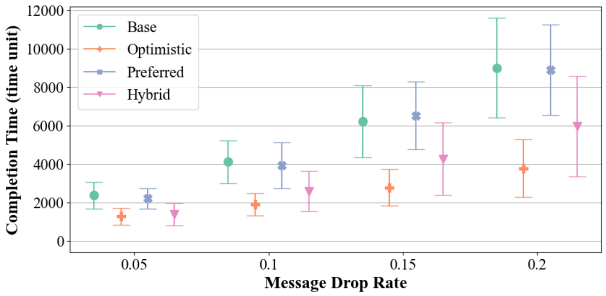


Fig. 7: *Completion time with different drop rates*

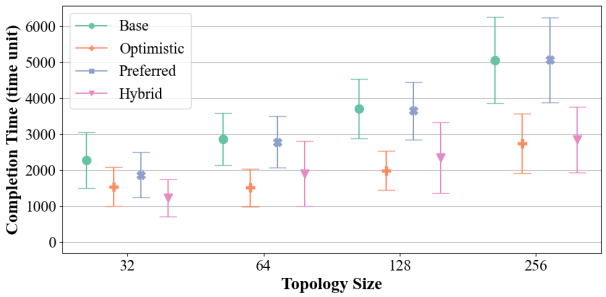


Fig. 8: *Completion time with different topology sizes*

unhealthy nodes in the system, while the Optimistic Protocol and Hybrid Protocol are aimed at finishing quickly.

We measure the implications of these optimizations by measuring three different metrics: the number of times the leader is changed within the same election run, the unhealthiness of the elected leader, and the hash rank of the elected leader.

1) Leader Change Count: This is the *number of times that a different leader multicasts* LEADER *message within the same election run.* This metric measures the disruption to the application (running at nodes) which may need to take actions (e.g., contacting the leader) right after it recognizes a new leader. Fig. 9 shows that, as expected, the leader change count for both the Base Protocol and Preferred Protocol are small ($= 1$, since they only send one leader notification per run). The Optimistic Protocol has a small leader change count because membership lists are largely consistent. The Hybrid Protocol has the highest leader change count, increasing with message drop rate—28% worse than the Base Protocol at 0.05 message drop rate, and 157% worse at 0.2 drop rate. Hence the tradeoff achieved by the Hybrid Protocol is frequent leader changes (during the election) vs. a higher quality leader (next paragraph).

2) Unhealthy Rank: This is the *system-wide unhealthiness rank of the elected leader*, where a higher value means the leader is healthier (0 represents the most unhealthy node and N represents the healthiest node in the system). Fig. 10 shows that: (a) the Base Protocol and Optimistic Protocol may be lucky and elect healthier leaders at low message drop rates (0.05) since many nodes are healthy, but as the drop rate rises to 0.1 and beyond, these protocols start electing largely unhealthy leaders, and (b) the Preferred Protocol and Hybrid Protocol elect healthier leaders with $1.5\times$ better healthiness ranks than the Base Protocol at 0.05 message drop rate and $28.5\times$ at 0.2 message drop rate (the Optimistic Protocol is similar).

3) Hash Rank: This metric shows how close our protocol comes to electing the "best attribute" (lowest hash) leader. Concretely, it is the *rank of the leader node hash*, with 0 representing the lowest hash node (and $N-1$ the highest hash). Fig. 11 shows that both the Base Protocol and Optimistic Protocol only elect node 0 (lowest hash ID), as expected. While the Preferred Protocol and Hybrid Protocol elect higher hash ranks, the hash rank values stay low and range from around 0.5 to 2.0 (out of large N)—this indicates that the Preferred Protocol and Hybrid Protocol are able to elect healthy leaders (as we saw in Fig. 10) while still being able to minimize the "best attribute" (hash ID) of the elected leader.

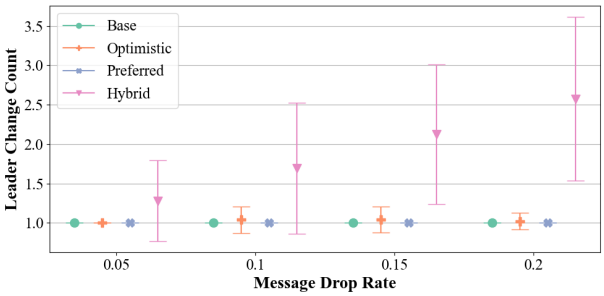


Fig. 9: *Leader change count with different drop rates*

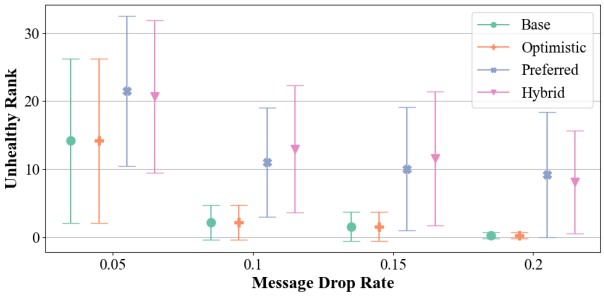


Fig. 10: *Leader unhealthy rank with different drop rates*

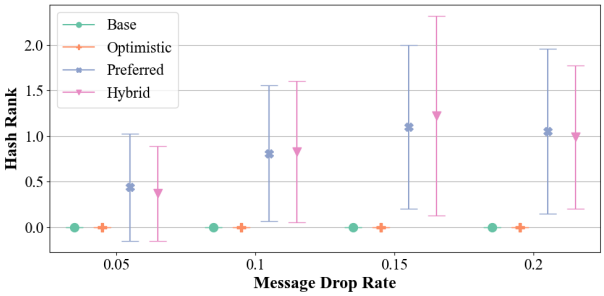


Fig. 11: *Leader hash rank with different drop rates*

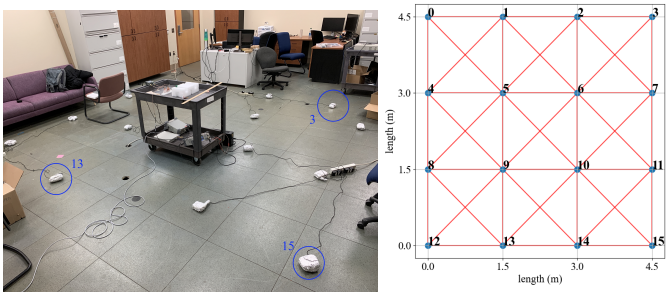
VII. DEPLOYMENT WITH RASPBERRY PIS

We implement our four c-tolerant leader election protocols for Raspberry Pi 4 devices, using about 2500 lines of

Java code. We reuse code from the authors of the Medley system [21] as the membership layer (with their default configuration parameters) and layer our election protocols atop it. Separately, we also deployed Zookeeper [11] on the same Raspberry Pis, and compare it against our protocols.

A. Raspberry Pi Deployment Results

Deployments involve a default of 16 Raspberry Pi 4 devices connected in an ad-hoc mesh network. The topology is a 4x4 grid (in a 4.5m x 4.5m area) shown in Fig. 12b and the actual lab placement is shown in Fig. 12a. Each device is a Raspberry Pi 4 model B, with 2GB LPDDR4 RAM and Broadcom BCM2711, 1.5 GHz quad-core Cortex-A72 CPU. Based on recommendations [21], we attenuate the transmit power of each device to about 15 dBm. We use OLSRD for packet routing due to its easy configurability and popularity.



(a) Lab placement (b) Deployment topology

Fig. 12: *Topology of Raspberry Pi deployment*

The parameters and the network configurations are similar to those used in the simulation results of Sec. VI. The c values are set to $2, 3, 5, 7$ for message drop rates $0.05, 0.1, 0.15, 0.2$ respectively, based on our measurements from 5-minute runs of the failure detector [21].

Fig. 13 shows the completion time vs. message drop rates. Overall, the comparative performance among the four protocols is similar to the simulation results (Fig. 7), thus validating that our simulator of Sec. VI successfully models practically observed behavior. We do observe that the completion time (across protocols) varies a bit less in the Pi deployment than in the simulation. This is because the deployment topology is smaller (16 nodes) compared to the simulation (49 nodes)—lower system size results in lower c values and thus fewer messages.

Fig. 14 shows the number of leader change counts (during a given run) vs. message drop rates. Again the comparative trends parallel simulation results (Fig. 9), with a couple of exceptions. We do observe that the leader change count for the Optimistic Protocol is relatively higher than in the simulation results. This is because the effective message drop rates in deployment are higher than the drop rates we set, e.g., Raspberry Pis themselves can drop packets even with message drop rate set to 0. This higher drop rate leads to more missing entries in membership lists and thus more leader changes. On the contrary, leader change count for the Hybrid Protocol is less than in simulation at high drop rates—this is because the initiator receives fewer RESPONSE (lower c) messages thus limiting the number of leader changes.

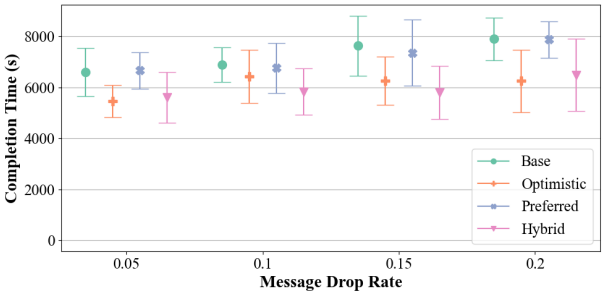


Fig. 13: *Completion time with different drop rates on Raspberry Pis*

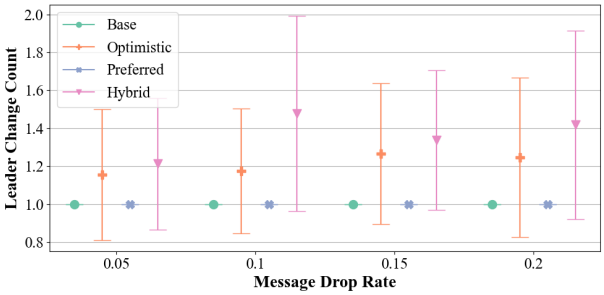


Fig. 14: *Leader change count with different drop rates on Raspberry Pis*

B. Resource Utilization Comparison with Zookeeper

Carefully designed protocols for edge settings can offer significant benefits over "stock" open-source software. To illustrate this, we compare against a state-of-the-art coordination and election system. Specifically, we enabled Zookeeper [11] to run on the Raspberry Pi 4 devices, and we compare against our c-tolerant leader election protocol (our Hybrid Protocol) on the same settings as Sec. VII-A (16 Raspberry Pis connected in a 4x4 grid mesh). Both Zookeeper and our c-tolerant election run continuously for around 10 minutes and we record their memory usage, CPU usage, and network traffic (via tcpdump [47]). For Zookeeper, we set parameter values as suggested by the official documentation [48] ($tickTime = 2000, initLimit = 5, syncLmit = 2$).

Fig. 15 shows the memory utilization and Fig. 16 shows the CPU utilization (averaged across all nodes). In the stable state (after time $t = 100s$), our c-tolerant election utilizes 26.9% less CPU and 5.9% less memory than Zookeeper. During the warm-up phase ($t < 100s$), there is a spike in memory used by c-tolerant election when it runs the election, however even the spike uses less than 4% memory and it's less than 13% above the stable memory usage.

Fig. 17 shows the total network bandwidth across all nodes during the 10-min run. The average bandwidth consumed by our c-tolerant election protocol is 20 Kbps, which is 82.9% less than Zookeeper's average bandwidth (117 Kbps). Further,

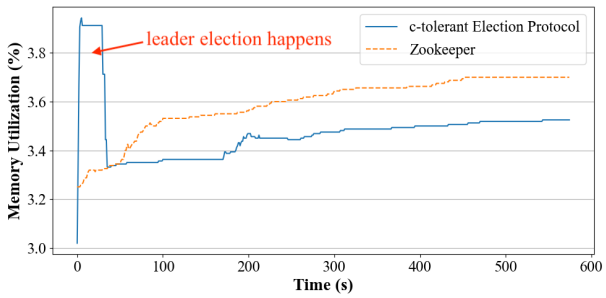


Fig. 15: *Memory Utilization Comparison*

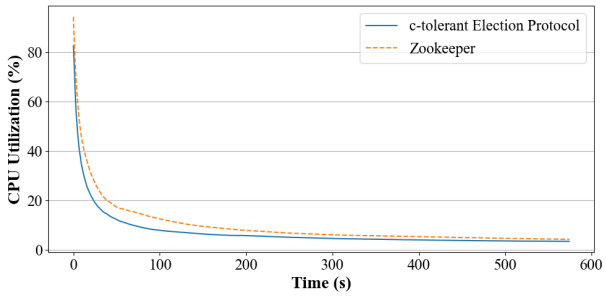


Fig. 16: *CPU Utilization Comparison*

our c-tolerant election's network usage is more stable than Zookeeper, with a standard deviation over $10\times$ lower: the respective standard deviations are 2.1 Kbps and 25.1 Kbps.

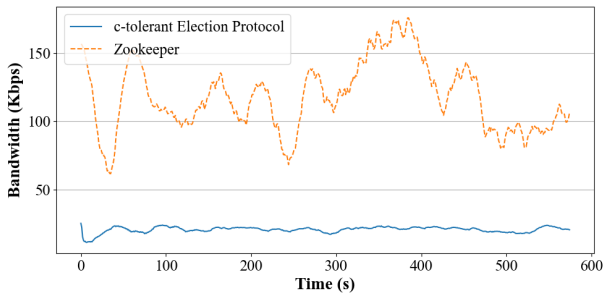


Fig. 17: *Network Bandwidth Comparison*

VIII. Summary

We presented a family of four churn-tolerant (c-tolerant) leader election protocols intended for edge environments that face churn. Our protocols only require a weakly-consistent membership protocol running underneath it. Our four protocols satisfy Safety and Liveness, use provably minimal messages, and elect healthy leaders with high probability. Our experiments with both trace-driven simulations, as well as a Raspberry Pi deployment, showed that: (i) our Optimistic Protocol reduces leader election completion time by 42.6% (vs. Base Protocol), (ii) our Preferred Protocol elects healthier leaders, and (iii) compared to "stock" Zookeeper, our c-tolerant election's bandwidth usage is 82.9% less and $10\times$ more stable, and its memory and CPU usage are respectively 5.9% and 26.9% lower.

Future Directions: Zookeper's suboptimal performance may arise from either a mismatch between its consensus protocol (ZAB) and churn, or due to extra functionalities that make Zookeeper too "heavy" for election. It remains an open question whether Zookeeper can be pared down or adapted to be efficient in churn-tolerant scenarios.

Code: Open-source implementation of the churn-tolerant election protocol in this paper is available at: http://dprg.cs.uiuc.edu/downloads.php.

Acknowledgments

This work was support in part by the following grants: NSF CNS-1908888, NSF IIS-1909577, and gifts from Capital One and Microsoft.

References

[1] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.
[2] (2022) Global edge computing market to reach $156 billion by 2030. [Online]. Available: https://www.techrepublic.com/article/global-edge-computing-market/
[3] (2022) Internet of things: Key stats for 2022. [Online]. Available: https://techinformed.com/internet-of-things-key-stats-for-2022/
[4] (2022) Earthsense terrasentia robots. [Online]. Available: https://www.earthsense.co
[5] (2022) Earthsense cover crop robots. [Online]. Available: https://www.earthsense.co/regen
[6] A. N. Sivakumar, S. Modi, M. V. Gasparino, C. Ellis, A. E. B. Velasquez, G. Chowdhary, and S. Gupta, "Learned visual navigation for under-canopy agricultural robots," *arXiv preprint arXiv:2107.02792*, 2021.
[7] (2022) Planet inc. dove satellite constellation. [Online]. Available: https://www.planet.com/our-constellations/
[8] (2022) Starlink satellite constellation. [Online]. Available: https://www.starlink.com/
[9] M. A. Khan, H. E. Sayed, S. Malik, T. Zia, J. Khan, N. Alkaabi, and H. Ignatious, "Level-5 autonomous driving—are we there yet? a review of research literature," *ACM Computing Surveys (CSUR)*, vol. 55, no. 2, pp. 1–38, 2022.
[10] I. Gupta, R. Van Renesse, and K. P. Birman, "A probabilistically correct leader election protocol for large groups," in *International symposium on distributed computing*. Springer, 2000, pp. 89–103.
[11] (2022) Apache zookeeper. [Online]. Available: https://zookeeper.apache.org/
[12] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, p. 133–169, may 1998. [Online]. Available: https://doi.org/10.1145/279227.279229
[13] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX ATC (Usenix ATC 14)*, 2014, pp. 305–319.
[14] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE transactions on Computers*, vol. 31, no. 01, pp. 48–59, 1982.
[15] N. Malpani, J. L. Welch, and N. Vaidya, "Leader election algorithms for mobile ad hoc networks," in *Proc. of the 4th international workshop on DIALM*, 2000, pp. 96–103.
[16] S. Vasudevan, J. Kurose, and D. Towsley, "Design and analysis of a leader election algorithm for mobile ad hoc networks," in *Proc. of the 12th IEEE ICNP 2004*. IEEE, 2004, pp. 350–360.
[17] D. Mazieres, "The stellar consensus protocol: A federated model for internet-level consensus," *Stellar Development Foundation*, vol. 32, pp. 1–45, 2015.
[18] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proc. of the 13th EuroSys conference*, 2018, pp. 1–15.
[19] R. Van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Middleware'98*. Springer, 1998, pp. 55–70.
[20] A. Das, I. Gupta, and A. Motivala, "SWIM: Scalable weakly-consistent infection-style process group membership protocol," in *Proc. of IEEE DSN*. IEEE, 2002, pp. 303–312.

[21] R. Yang, J. Wang, J. Hu, S. Zhu, Y. Li, and I. Gupta, "Medley: A membership service for iot networks," *IEEE Transactions on Network and Service Management*, vol. 19, no. 3, pp. 2492–2505, 2022.

[22] D. Boneh, S. Eskandarian, L. Hanzlik, and N. Greco, "Single secret leader election," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 12–24.

[23] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[24] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[25] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proceedings of the eleventh ACM Symposium on Operating systems principles*, 1987, pp. 123–138.

[26] Y. Amir, D. Dolev, S. Kramer, and D. Malki, *Transis: A communication sub-system for high availability*. Hebrew University of Jerusalem. Leibniz Center for Research in Computer Science, 1991.

[27] O. Babaoglu, R. Davoli, L.-A. Giachini, and M. G. Baker, "RELACS: A communications infrastructure for constructing reliable applications in large-scale distributed systems," in *Proc. of 28th HICSS*, vol. 2. IEEE, 1995, pp. 612–621.

[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[29] Z. J. Haas, J. Y. Halpern, and L. Li, "Gossip-based ad hoc routing," *IEEE/ACM Transactions on networking*, vol. 14, no. 3, pp. 479–491, 2006.

[30] K. Birman, "The promise, and limitations, of gossip protocols," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 8–13, 2007.

[31] I. Gupta, K. P. Birman, and R. Van Renesse, "Fighting fire with fire: using randomized gossip to combat stochastic scalability limits," *Quality and Reliability Engineering International*, vol. 18, no. 3, pp. 165–184, 2002.

[32] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz *et al.*, "Handling churn in a dht," in *Proceedings of the USENIX ATC*, vol. 6. Boston, MA, USA, 2004, pp. 127–140.

[33] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, 2006, pp. 189–202.

[34] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports, "Building consistent transactions with inconsistent replication," *ACM TOCS*, vol. 35, no. 4, pp. 1–37, 2018.

[35] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi *et al.*, "Service fabric: a distributed platform for building microservices in the cloud," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[36] I. Gupta, R. Van Renesse, and K. P. Birman, "Scalable fault-tolerant aggregation in large process groups," in *Proc. of IEEE DSN*. IEEE, 2001, pp. 433–442.

[37] C. Sengul, M. J. Miller, and I. Gupta, "Adaptive probability-based broadcast forwarding in energy-saving sensor networks," *ACM Transactions on Sensor Networks (TOSN)*, vol. 4, no. 2, pp. 1–32, 2008.

[38] F. Kuhn, S. Schmid, and R. Wattenhofer, "A self-repairing peer-to-peer system resilient to dynamic adversarial churn," in *International Workshop on Peer-to-Peer Systems*. Springer, 2005, pp. 13–23.

[39] S. Legtchenko, S. Monnet, P. Sens, and G. Muller, "Relaxdht: A churn-resilient replication strategy for peer-to-peer distributed hash-tables," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 2, pp. 1–18, 2012.

[40] 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.

[41] M. Fischer and H. Jiang, "Self-stabilizing leader election in networks of finite-state anonymous agents," in *International Conference on Principles of Distributed Systems*. Springer, 2006, pp. 395–409.

[42] S.-T. Huang, "Leader election in uniform rings," *ACM TOPLAS*, vol. 15, no. 3, pp. 563–573, 1993.

[43] R. Chandra, V. Ramasubramanian, and K. Birman, "Anonymous gossip: Improving multicast reliability in mobile ad-hoc networks," in *Proc. of 21st ICDCS*. IEEE, 2001, pp. 275–283.

[44] J. C. Lin and S. Paul, "Rmtp: A reliable multicast transport protocol," in *Proceedings of IEEE INFOCOM'96. Conference on Computer Communications*, vol. 3. IEEE, 1996, pp. 1414–1424.

[45] R. Bhagwan, S. Savage, and G. M. Voelker, "Understanding Availability," in *Peer-to-Peer Systems II*. Springer, 2003, pp. 256–267.

[46] D. Kostoulas, D. Psaltoulis, I. Gupta, K. P. Birman, and A. J. Demers, "Active and passive techniques for group size estimation in large-scale and dynamic distributed systems," *Journal of Systems and Software*, vol. 80, no. 10, pp. 1639–1658, 2007.

[47] (2022) Tcpdump. [Online]. Available: https://www.tcpdump.org

[48] (2022) Apache zookeeper guide. [Online]. Available: https://zookeeper.apache.org/doc/r3.3.3/zookeeperStarted.html