Camera: Churn-Tolerant Mutual Exclusion for the Edge

Aman Khinvasara
Siebel School of Computing and Data Science
University of Illinois at Urbana-Champaign
Champaign, USA
amantk2@illinois.edu

Indranil Gupta
Siebel School of Computing and Data Science
University of Illinois at Urbana-Champaign
Champaign, USA
indy@illinois.edu

***Abstract*—Emerging edge and IoT networks are characterized by *churn*, wherein nodes join, leave, and fail continuously. Together with flaky network links and delays in edge and IoT systems, this means nodes have differing views of the system membership at any point of time. This paper focuses on the classical mutual exclusion problem—which ensures at most one node executes the "critical section" at any point in time—over such churned edge networks. We first show that two classical algorithms for mutual exclusion (Ricart-Agrawala and Maekawa) violate safety even under very small amounts of churn (up to 4 churned entries in membership lists). We then present Camera, a churn-tolerant variant of Ricart-Agrawala's algorithm. We formally prove Camera satisfies key properties including safety, deadlock-freedom, and starvation-freedom. Our trace-driven simulation results, using synthetic traces, distributions, and churn traces from peer to peer environments, show that Camera has scalable wait times and bandwidth overheads.**

***Index Terms*—churn, mutual exclusion, edge, IoT, distributed system**

I. Introduction

Edge computing systems are rapidly becoming critical infrastructure. Driven by falling hardware costs, data-hungry needs, and burgeoning use cases, the Internet of Things (IoT) market is expected to grow 16.7% YoY, reaching $650.5 billion in 2026 [1]. Use cases in static environments include smart manufacturing, retail [1], energy grids [2], structural health monitoring [3], and healthcare [4]. While static settings have been the focus of most previous work, dynamic environments are becoming increasingly important, including smart agriculture [5], livestock management [6], on-body healthcare [4], defense [7], LEO (Low Earth Orbit) satellite constellations [8], smart transportation [1] (cargo monitoring, autonomous vehicles, etc.), and more.

These deployments often involve *shared resources* that need to be accessed by multiple *writers*. Examples include: (1) sensors attached to grazing cattle that take turns writing to a given file (e.g., log of activities); (2) reconnaissance robots in the battlefield taking turns surveying a particular area; and (3) coordinated actions for energy efficiency, e.g., executing environment sensing at only one IoT device at a time [9]. Such coordination may involve either physical resources (e.g., files) or non-physical resources (e.g., coordinated actions).

This work was supported in part by a gift from Micrsoft.

The above examples are in fact the classical *mutual exclusion* problem, also known as the *Critical Section* problem. A Critical Section (CS) is a portion of the code that accesses the shared resource. Concretely it provides: (i) (*Safety*) at most one device should execute the CS simultaneously, and (ii) (*Liveness*) all requests should eventually be granted. Many classical solutions to mutual exclusion exist, including by Ricart-Agrawala [10], Raymond [11], and Maekawa [12]. All these algorithms assume *strong membership* underneath them. In strong membership each non-faulty node (process) is assumed to always know the up/down status of every other node in the system.

Yet, edge environments are highly dynamic: *churn* arises from devices (nodes) that continuously join, leave, move, and fail. For instance, peer-to-peer (P2P) systems are well-known to suffer hourly churn rates anywhere between 25% to 100% [13], [14]. Other emerging edge scenarios are also plagued by churn—smart farm robot swams experience churn due to battery depletion or getting stuck in mud, and LEO satellites over a region also suffer churn due to fast mobility, e.g., the set of LEO satellites over a state has a 1500% churn rate [8]! Further, the lossy nature of wireless links in edge networks means that any scalable membership protocol will naturally have *false positives*, meaning it may detect healthy nodes as failed [15]. Under such scale and churn, strongly consistent membership protocols [16]–[18]—which aim to ensure all nodes' membership lists are always identical and correct before allowing applications to use them—are known to scale poorly (beyond a few 10s of nodes), use many messages, and continually block the system from making progress [19]; these behaviors arise because strong membership is constrained by its need to maintain correctness of membership lists, all the time. Layering over strong membership is thus a poor design choice for large edge networks.

Recent work on *weakly-consistent* membership protocols for edge settings addresses this, e.g., Medley [20], RaWMS [21], and others [22]. These *eventually* update membership lists in message-efficient and scalable ways. However, the membership lists they maintain at each node may be: (1) *incomplete*, i.e., missing entries pointing to some nodes; (2) *stale*, i.e., contains listings of nodes that have failed or left the system; and (3) *heterogeneous*, i.e., some nodes may have a nearly-complete membership list while others may be missing several entries. This means that classical mutual exclusion algorithms, when run over the (more practical) weakly consistent membership protocols, may violate correctness.

In this paper, we first show how both classical Ricart-Agrawala (RA) [10] and Maekawa [12] algorithms violate safety when subject to churn. Then we focus on RA, and we present a churn-tolerant variant of the RA algorithm (to be run over weak membership) and prove that it satisfies safety and liveness. We evaluate our implementation using synthetic workloads and real churn traces. To the best of our knowledge, this is the first fully-decentralized churn-tolerant mutual exclusion algorithm for dynamic edge environments.

Our new algorithms leverage the key observation that weak membership protocols [20] result in membership graphs wherein even when two nodes *do not* know each other, they actually *both* know a common third node with high probability—we call this the *Mutual Friend Property (or MFP)*. Camera uses this property to nudge the system back towards a correct state. We validate that MFP holds true in practice w.h.p. We also propose an extended design ("slow path") which ensures safety even when MFP is untrue.

The contributions of this paper are:

1) Counter-examples to show how two classical algorithms [10], [12] violate safety when subjected to churn;
2) Description of Camera (<u>C</u>hurn-toler<u>a</u>nt <u>M</u>utual <u>E</u>xclusion by extending <u>R</u>icart-<u>A</u>grawala), an efficient algorithm for mutual exclusion in ad hoc edge networks, layered over weakly-consistent membership;
3) Formal proofs of correctness of safety, starvation-freedom, and deadlock-freedom;
4) A detection mechanism and fall-back algorithm for scenarios when the MFP does not hold; and
5) Experimental results using synthetic workloads and churn traces, showing that Camera scales with system size, spatial and temporal heterogeneity, and churn, all with low overhead compared to Ricart-Agrawala, and scales better than baseline wireless Paxos solutions.

II. Preliminaries

A. Problem Statement

The Mutual Exclusion (or Critical Section (CS)) problem is characterized by a safety clause and two liveness clauses:

Definition 1 (Safety)**.** At any point in time, at most one node is in the CS.

Definition 2 (Deadlock-freedom, or Liveness 1)**.** The system never deadlocks, i.e., a state where no node is in the CS and no node can ever proceed into the CS.

Definition 3 (Starvation-freedom, or Liveness 2)**.** The system never starves any node's request to enter the CS (while others are able to access the CS).

B. System Model

We adopt an asynchronous fail-stop system model (we do not tackle Byzantine failures). There are up to N nodes. Nodes may leave, fail (by crashing), and (re-)join at any time. The communication medium may arbitrarily delay messages but delivers them eventually. Clocks are unsynchronized. Each node sends at most one Critical Section (CS) request at a time, but multiple requesters may simultaneously send CS requests. We assume that if a node A knows (via its membership list) a node B, then A can send messages to B (although Camera's design is agnostic to network internals, our later experiments will use an ad-hoc network among nodes).

§ **Membership Assumptions:** We assume that running underneath the mutual exclusion protocol is a weakly-consistent membership protocol (henceforth called *weak membership*), such as Medley [20]. Arbitrary propagation delays for joins and fails/leaves may occur. Failures are detected eventually but false positives may occur (non-faulty/flaky nodes being mistakenly detected as failed).

Note that the membership graph is a directed graph, i.e., a node p_i may know node p_j, but p_j may or may not also know p_i. We assume the membership graph is strongly connected (i.e., each node can reach any other node indirectly through the membership pointers), because:

Theorem 1. *A strongly connected membership graph is necessary for any mutual exclusion algorithm to satisfy safety and liveness. Proof in IDEALS version [23].*

Intuitively, without strong connectivity, there is no way for information to flow between an arbitrary pair of conflicting requesters (even indirectly), and thus scenarios exist wherein safety or liveness is violated. Next, we define:

Definition 4 (**Mutual Friend Property (MFP)**)**.** Denote p_i's membership list as $Memlist_i$ (contains p_i). For any pair of requesting nodes p_i, p_j: $Memlist_i \cap Memlist_j \neq \emptyset$.

Essentially MFP means any pair of nodes either knows each other or knows at least one common node. In fact, MFP holds even in the presence of plenty of inconsistency across membership lists:

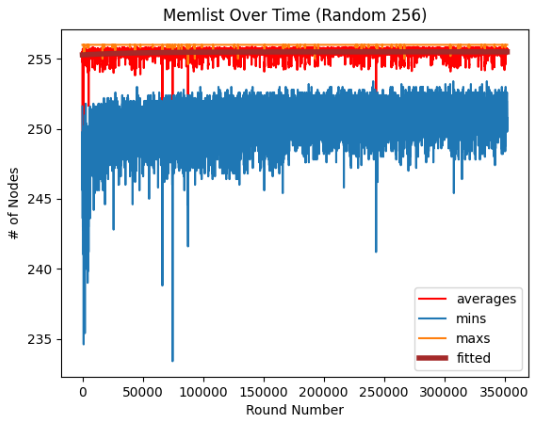
Lemma 2. *Let $|Memlist_i| > \frac{N}{2}$ at all requesting nodes p_i. Then, the Mutual Friend Property is satisfied.*

Proof. In a system with at most N nodes, any pair of (membership) sets of size $> \frac{N}{2}$ intersects in at least one node. So given two nodes p_i, p_j, we have that $Memlist_i \cap Memlist_j \neq \emptyset$, therefore a third process p_k is known by both p_i and p_j. This satisfies MFP. □

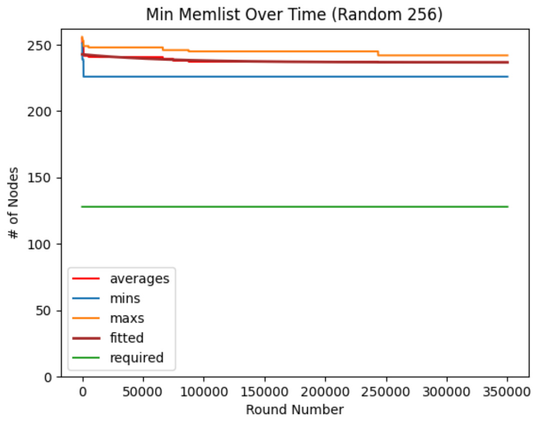
To empirically validate this, we use the openly available code of the Medley membership protocol [20]. Figure 1 shows a 350K-round run of a $N = 256$-node Medley system. While the average membership length stays nearly at N, even the *minimum* membership list length never drops below 90% $\times N$ (in particular, never approaches 50% $\times N$). We conclude that Lemma 2 and thus MFP are true in practice.

III. Churn Breaks Classical Mutual Exclusion

We show scenarios where *just a few (two to four) churned membership entries across the entire system* suffice to vio-

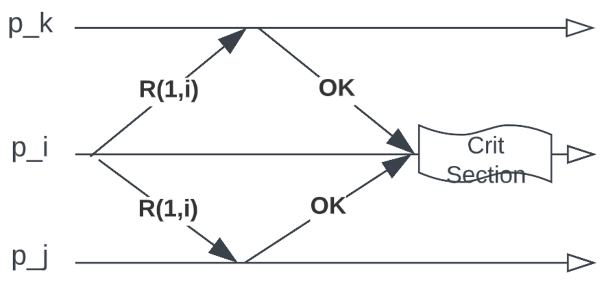


(a) Membership List Size

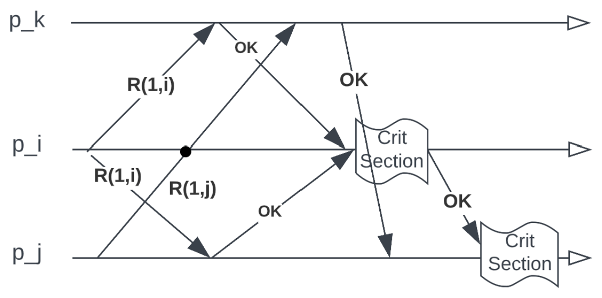


(b) Minimum Membership List Size

Fig. 1: **Weak Membership Lists maintain Quorum and thus MFP: Experiments from Medley [20].**



(a) Single Node Requesting.



(b) Two Nodes Successfully Requesting.

Fig. 2: **Classical Ricart-Agrawala Algorithm [10].**

late safety in classical mutual exclusion algorithms: Ricart-Agrawala [10] and Maekawa [12].

A. Breaking Classical Ricart-Agrawala Algorithm

§ **How Classical Ricart-Agrawala Algorithm Works:** In classical Ricart-Agrawala, a node p_i, wishing to enter the CS, sends a REQUEST message containing an ordered tuple/pair of (Lamport timestamp [24], p_i's ID) to *every* node in p_i's membership list. p_i enters its CS only after receiving OKs back from *all* these contacts. When a node p_j receives a REQUEST from p_i, p_j locally queues (defers) the request only if either: (i) p_j is currently in the CS, or (ii) p_j has an outstanding request with a lower timestamp than p_i's request (ties broken by process ID). Otherwise (if the request is not queued), p_j immediately responds to p_i with an OK. Upon exiting the CS, a node p_i replies OK to *all* its locally queued requests. The timestamps in the original Ricart-Agrawala algorithm are Lamport timestamps [24].

Figure 2a shows a single node's request in Ricart-Agrawala. Figure 2b shows concurrent requests from 2 nodes. Other nodes (p_k in figure) give OKs to both p_i, p_j. However, because p_i's request has a lower timestamp than p_j (by tiebreaker i ¡ j), p_i queues p_j's request. Thus p_i enters the CS first, then exits, and sends an OK to p_j. Finally, p_j enters the CS.

§ **Churn Breaks Ricart-Agrawala:** We prove:

Theorem 3. *Let p_i and p_j be two concurrently requesting nodes, with timestamps $Req_i < Req_j$. There exist runs of Ricart-Agrawala in which safety is violated with just 2 churned nodes (missing entries in membership lists).*

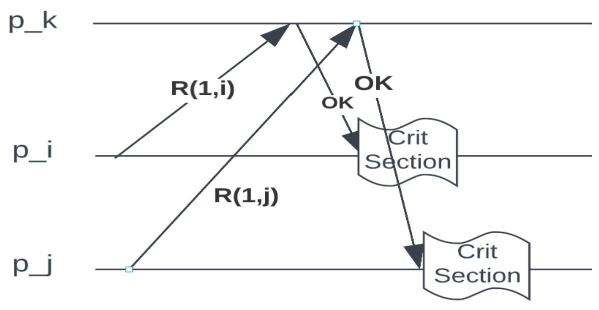
Proof. Figure 3 shows the two cases that can violate safety:

1) Symmetric Ignorance:
   $p_i \notin Memlist_j$ and $p_j \notin Memlist_i$
2) Asymmetric Ignorance:
   $p_i \notin Memlist_j$ and $p_j \in Memlist_i$

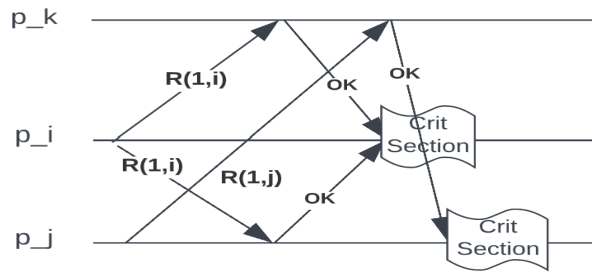
Figure 3a shows an example of failure under Symmetric Ignorance. Two processes p_i, p_j request access to the CS, so any other node p_k responds OK to all received requests. However, since $p_i \notin Memlist_j$ and $p_j \notin Memlist_i$, each of p_i, p_j receives OKs from everyone in its (respective) membership list and each proceeds into the CS, violating safety.

Figure 3b shows an example of failure under Asymmetric Ignorance. p_i and p_j request access to the CS, where p_i knows p_j but p_j does not know p_i. Because the non-requesting nodes immediately respond with OK, and p_j locally evaluates $Req_i < Req_j$ and also sends an OK, p_i receives OKs from every member of its membership list. Simultaneously, p_j receives OKs from all members of its membership list (which was missing p_i) and enters the CS, violating safety.

□



(a) Symmetric Ignorance Failure.



(b) Asymmetric Ignorance Failure.

Fig. 3: **Ricart-Agrawala [10] violates Safety with just two missing entries in membership lists.**

In fact, we experimentally confirmed that Ricart-Agrawala violates safety under churn and incomplete membership. In our experiment (settings in Section VII—using Weibull spatial distribution, Zipfian temporal distribution, a churn forward rate of 60), even with only 10% message drop rate, up to 2.74 nodes (on average) executed the CS simultaneously.

B. Breaking Classical Maekawa Algorithm

While our paper focuses on modifying RA, we also show here that another mutual exclusion is broken by churn (although modifying this latter algorithm is beyond our scope).
§ **How Classical Maekawa's Algorithm Works:** Maekawa [12] defines for each node p_i a voting set V_i consisting of $O(\sqrt{N})$ nodes (including p_i). Like Ricart-Agrawala, a requesting node p_i sends REQUESTs to all its voting set members, which respond back with an OK. p_i can enter the CS only after all in V_i have responded. Unlike Ricart-Agrawala, however, in Maekawa a voting set member can send *at most one OK* at a time, queuing all other received requests. When p_i exits the CS it sends a RELEASE message to all in V_i. A voting set member p_j receiving a RELEASE dequeues its next queued request and sends the requester an OK (maintaining the rest of the queue).

When membership lists are complete and correct, Maekawa ensures safety by constructing voting sets such that any pair V_i, V_j intersect in at least one node — this common node can give an OK to at most one requester at a time. Maekawa recommends doing this by arranging nodes in a $\sqrt{N}$ by $\sqrt{N}$ *voting matrix* [25], and defines node p_i's voting set V_i as the union of its row and column in that matrix.

§ **Churn Breaks Maekawa:** With churn and resulting weak membership, Maekawa can simultaneously allow two requesting nodes into the CS if churn causes the (typical) two intersecting nodes (in the matrix described above) to be replaced in requesters' membership lists. Figure 4 depicts a system with 11 nodes where p_1 and p_{11} are each missing 2 nodes, resulting in a safety violation. Thus we can state:

Theorem 4. *Maekawa can violate safety with just four replaced entries in membership lists.*

1	2	3
5	6	7
8	10	11

p_1 voting matrix

1	2	4
5	6	7
9	10	11

p_{11} voting matrix

Fig. 4: **Maekawa violates safety if p_1 and p_{11} request simultaneously because their voting sets are disjoint, having swapped p_3 with p_4 and p_8 with p_9.**

IV. Camera: Churn-tolerant Mutual Exclusion by extending Ricart-Agrawala

This paper focuses on the Ricart-Agrawala algorithm (which Section III showed is broken by churn). We present Camera, a churn-tolerant variant of the Ricart-Agrawala algorithm.

Ricart-Agrawala fails under churn when requesting nodes don't know each other. The key intuition of Camera is to leverage the MFP property (Section II) and use overlapping nodes to nudge the system towards correct behavior.

Below we will refer to the Ricart-Agrawala steps from Section III-A. The key idea in Camera is that whenever a node p_j responds OK to p_i's request, p_j also includes in the OK message a list of all other nodes to whom p_j has *recently* sent an OK. We expect this set to be small, because the tight resource constraints in most dynamic systems mean only a few nodes would request to enter the CS simultaneously. For example, IoT transportation settings find 95% of requests have interarrival times greater than 100*ms* [26].

Given this, the requesting node p_i receiving a recentlyOKed set (inside an OK message) adds any nodes it does not know about, from this set, to its own (p_i's) membership list. p_i then sends its REQUEST to these newly-learned nodes as well, and waits for their OKs (before entering the CS).

Algorithm 1 shows the full pseudocode of Camera—what we describe in this section is the *fast path*. Later we will describe the slow path of Camera (Section VI).
Ensuring Safety (Intuition): Camera's key design idea addresses both safety violations in Theorem 3. First, when requesters p_i, p_j don't know each other (Symmetric Ignorance Case 1 in Theorem 3), the recentlyOKed set provided by mutual friend p_k will inform the second requester (say p_j) about the first requester p_i. This causes p_j to send a REQUEST

Algorithm 1: Camera Pseudo-code

```
Definitions:
enum State = {WAIT,HELD,NONE};
struct Request = {int timestamp, string id};
Initialization:
Set Deferred, recentlyOKed, Pending_OKs = ∅;
string me = ”myId”;
Request myRequest = None;
State myState = State.NONE;
int max_timestamp_seen = 0;

def enter():
    myState = State.WAIT;
    max_timestamp_seen += 1;
    myRequest = (max_timestamp_seen, me);
    Pending_Oks = Memlist;
    Pending_Oks.remove(me);
    for peer in Pending_Oks:
        send_message(myRequest,peer);
    WAITFOR(Pending_Oks.empty());
    myState = State.HELD;
    //Enter critical section!

def exit():
    myState = State.None;
    IP_MULTICAST(RELEASE(myRequest));
    for request in Deferred:
        Deferred.remove(request);
        send_message(OK(recentlyOKed),request.id);
        recentlyOKed.insert(request);
```

Algorithm 1: Camera Pseudo-code (cont.)

```
def receive_request(Request req, string sender):
    if req.id not in Memlist:
        if myState == State.WAIT:
            Pending_OKs.insert(req.id);
            send_message(myRequest,req.id);
        Memlist.insert(req.id);
    max_timestamp_seen =
      max(max_timestamp_seen,req.timestamp);
    if myState = State.HELD or (myState = State.WAIT
            and req > myRequest):
        Deferred.insert(req);
    else:
        send_message(OK(recentlyOKed),req.id);
        recentlyOKed.insert(req);

def receive_release(Request req):
    recentlyOKed.remove(req);

def receive_OKs(sender, senderRecentlyOKed, Request
  req):
    for node in senderRecentlyOKed:
        if node not in Memlist:
            Pending_OKs.insert(node);
            send_message(myRequest, node);
            Memlist.insert(node);
    Pending_OKs.remove(sender);
```

to p_i and wait for its OK. When only one requester knows of the other (Asymmetric Ignorance Case 2 in Theorem 3), this is addressed in Camera by having REQUEST receivers add unknown requesters' IDs to their membership lists, e.g., when node p_j receives a REQUEST from a node p_i whom p_j doesn't know. If the receiver p_j is also WAITing, it sends the requester p_i its own (p_j's) request.

Figures 5 and 6 show the same cases as Figure 3 (where Ricart-Agrawala violated safety), but where Camera ensures safety in spite of churn.

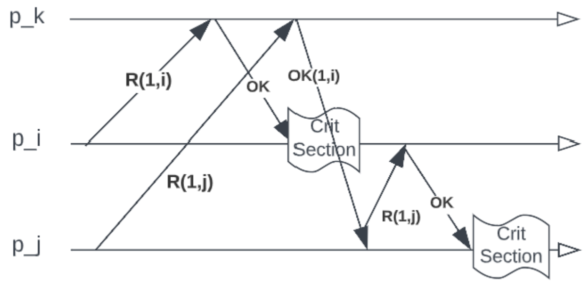


Fig. 5: **Camera Success Despite $p_i \notin Memlist_j$ and $p_j \notin Memlist_i$.**

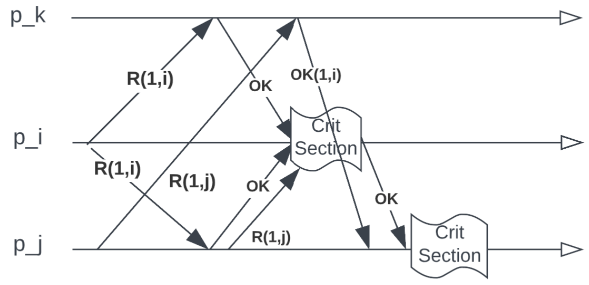


Fig. 6: **Camera Success Despite $p_i \notin Memlist_j$.**

Retention Period for recentlyOKed entries: An entry in the recentlyOKed list can be deleted when the corresponding RELEASE is received—this determines for how long recentlyOKed entries are retained. At any node p_k, while the recentlyOKed cache is bounded by N (since each node has at most one outstanding request), this still consumes large memory resources. Note that a request cannot contribute to safety violations after exiting the CS. Therefore, retention periods can be shortened by sending RELEASE messages eagerly on exiting the CS, instructing recipients to clear the corresponding OKs from their caches. A delayed RELEASE message from p_i to p_k does not violate safety or liveness since p_k will instruct future requesters to check with p_i directly, and p_i will respond with OKs. We later show this method to be effective.

V. Camera Analysis

We formally prove Camera's Safety and Liveness. Then we analyze causality and performance.

A. Safety

Safety must guarantee at most one node is executing the critical section (CS) at any point in time.

Theorem 5. *Camera ensures safety.*

Proof. Assume the contrary - that two nodes p_i and p_j execute the CS simultaneously. Assume without loss of generality that their requests are ordered (using Lamport timestamps) such that $Req_i < Req_j$. This generates four cases:

1) $p_i \in Memlist_j$ AND $p_j \in Memlist_i$
2) $p_i \notin Memlist_j$ AND $p_j \in Memlist_i$
3) $p_i \in Memlist_j$ AND $p_j \notin Memlist_i$
4) $p_i \notin Memlist_j$ AND $p_j \notin Memlist_i$

Case 1 represents normal Ricart-Agrawala operation, while Case 2 and Case 4 are respectively Asymmetric Ignorance and Symmetric Ignorance from Theorem 3 and violate safety (note that Case 3 does not violate safety).

Case 1 - $p_i \in Memlist_j$ and $p_j \in Memlist_i$
This is the normal Ricart-Agrawala case from Figure 2b, and so satisfies safety [10].

Case 2 - $p_i \notin Memlist_j$ and $p_j \in Memlist_i$
From Figure 6, for p_i to be in the CS, it must have sent Req_i to p_j and received an OK back. There are 3 sub-cases depending on when p_j enters the CS vs. when it receives Req_i:

1) p_j receives Req_i while in its CS, so p_j defers until exiting, maintaining safety
2) p_j receives Req_i before entering its CS but after sending out Req_j. p_j learns about p_i and sends its own Req_j to p_i, along with an OK because p_j is WAITing and evaluates $Req_i < Req_j$. With both Req_j and OK in flight from p_j to p_i, there are two states p_i might be in when it receives Req_j.
   a) WAIT: Req_j arrives before some OK (either p_j's or another that is pending), so p_i evaluates $Req_i < Req_j$ and defers responding (queues).
   b) HELD: Req_j arrives after p_j's OK and all other pending OKs, and p_i automatically defers (queues).

   In either state a or b above, p_i waits until exiting to send the OK allowing p_j into the CS, contradicting the assumption that safety is violated.
3) p_j receives Req_i before initiating its process, so p_j responds with OK and adds p_i to $Memlist_j$. When p_j initiates the request, we have $p_i \in Memlist_j$, contradicting this case's assumption.

Case 3 - $p_i \in Memlist_j$ and $p_j \notin Memlist_i$
p_j would have sent Req_j to p_i and received an OK back. There are a few sub-cases of when p_i receives Req_j:

1) p_i is in state HELD when it receives Req_j, and thus defers until after exiting the CS
2) p_i receives Req_j before receiving its own needed OKs, but after initiating the process. p_i evaluates $Req_i < Req_j$ so defers sending the OK to p_j until after exiting the CS.
3) p_i receives Req_j before initiating its own process, so $Req_i > Req_j$ because $timestamp_i \geq timestamp_j + 1$, contradicting $Req_i < Req_j$!

Case 4 - $p_i \notin Memlist_j$ AND $p_j \notin Memlist_i$
From Figure 5, let $p_k \in Memlist_i \bigcap Memlist_j \neq \emptyset$, by MFP. For p_i and p_j to simultaneously enter the CS, both would have sent a REQUEST to p_k and received an OK. p_k processes the incoming requests serially. Assume without loss of generality that p_k processes Req_j first, so p_k first responds to p_j with OK and later to p_i with OK(Req_j). When p_i receives that message, it immediately adds p_j to $Memlist_i$. We now have $p_j \in Memlist_i$ and $p_i \notin Memlist_j$. Because we assigned p_j to be the node whose request p_k received first, two sub-cases arise:

1) $Req_i < Req_j$ - proof from Case 2 applies here;
2) $Req_j < Req_i$ - If we simply switch the nodes we've assigned to p_i and p_j, we have $p_i \in Memlist_j$ and $p_j \notin Memlist_i$, with $Req_i < Req_j$, so the proof for Case 3 applies here.

□

Corollary 5.1. *Even with arbitrary failures, as long as the Mutual Friend Property holds, Camera ensures Safety.*

This is because only one mutual friend is needed for safety, so other mutual friends failing does not matter (as long as MFP holds).

For correctness, for each potentially conflicting pair of requesters, we need to additionally assume an extension of the MFP property:

Extended MFP: During the course of any given requests from p_i, p_j, the overlap set between the membership lists $Memlist_i \cap Memlist_j$ needs to have at least one non-faulty node remaining throughout.

However, this is true with high probability:

Theorem 6. *Suppose p_i and p_j issue potentially conflicting requests. With N nodes in a system experiencing churn with inter-departure times exponentially distributed with parameter λ, where $|Memlist_i|, |Memlist_j| \geq \alpha N$, and the ratio of churn rate to average CS duration is c, the probability that there exists some p_k such that $p_k \in Memlist_i \cap Memlist_j$ during the entire time that p_i and p_j's requests may conflict is at least $\sum_{k=0}^{(2\alpha-1)N-1} \frac{e^{-\frac{N}{c}}(\frac{N}{c})^k}{k!}$.*

E.g., using parameters $c = 0.6, N = 189$, and (as suggested by Figure 1) $\alpha = 0.9$ already gives 0.99895018, and the result only increases with N. Proof in IDEALS version [23].

B. Liveness

A deadlock occurs when 1) no node is in the CS, and 2) no node will further be able to enter the CS.

Theorem 7. *Camera prevents deadlocks.*

Proof. Assume there is a deadlock. There must be some cycle of nodes deferring REQUESTs, meaning every node in that cycle has *at least* one OK deferred. Consider any edge $e = p_i \rightarrow p_j$ in this cycle, which means that p_j is waiting for an OK from p_i before entering, so $Req_i < Req_j$. Therefore the cycle $p_1, p_2, \ldots, p_k, p_1$ implies $Req_1 < Req_2 <$

$Req_3 < \ldots < Req_k < Req_1$, producing $Req_1 < Req_1$, an impossibility.

□

Theorem 8. *No request will be starved under Camera.*

Proof. Assume there is a starved node p_i. Define set S_i as all nodes that receive Req_i—note that $|S_i| \leq N$ (total number of nodes). Since messages are delivered eventually, after a finite (but unknown) time, all $p_j \in S_i$ will have eventually received Req_i. It is only in the corner case where an OK is pending from one of those nodes in S_i that p_i will be unable to enter the CS. But after p_j exits the CS, this corner case will dissipate and thereafter all subsequent Req_j's (from p_j) will be such that $Req_j > Req_i$ (due to causality).

Thus, as other nodes continue to enter and exit, p_i will eventually have the minimum timestamp (due to the total order on requests). No other node $p_j \in S_i$ will be able to continue deferring p_i, so after the final OK is received (which happens in finite time), p_i will be next to enter the critical section, contradicting starvation.

□

Note that failures do not affect liveness. Suppose p_i is waiting for an OK from p_j but instead detects that p_j failed. Then p_i treats this as a "silent" OK from p_j. Note this covers the case where p_j failed inside the CS. Further, to avoid false positive detections (of p_j) from causing inconsistency, p_i waits an extra timeout (to process silent OK) after which p_j will be forced to leave the system (e.g,. in [20] this extra timeout is the suspicion timeout, which is O(log(N)).

C. Causality

The classical Ricart-Agrawala ensures that requests are satisfied in causal order. Similarly:

Theorem 9. *If membership is strongly consistent, Camera maintains causality.*

Proof. Under consistent full membership, Camera's actions are indistinguishable from Ricart-Agrawala's (RA) actions. No Camera node would execute branches to handle learning about new nodes, which is the only logic distinguishing Camera from RA. RA already assures causality under strong membership, shown in Section 15.2 of [25]. □

Intuitively, while Camera's *recentlyOKed* sets increase bandwidth, they don't affect correctness under complete membership. However, in some corner cases under incomplete membership, both Ricart-Agrawala and Camera violate causality, e.g., when application messages outpace protocol messages. We reiterate that Theorem 9 implies that Camera's set of causality-violating configurations is a subset of Ricart-Agrawala's set of causality-violating configurations.

D. Performance Analysis

Bandwidth: Camera's number of messages for entering the CS lie in the interval $[2\times(length(Memlist_i)-1), 2\times(N-1)]$, and the number of messages for exiting the CS is O(1). These are identical to Ricart-Agrawala, with the caveat that including recentlyOKed sets may result in Camera OK messages of size O(N) - we later show they remain small.

Client Delay: The time to enter the CS assuming no contention from other requesters is 1 RTT, identical in Camera as Ricart-Agrawala.

Synchronization Delay: The time to handover the CS from one exiting node to the next solitary already-waiting node is 0.5 RTT, just as in Ricart-Agrawala.

VI. Slow Path: Relaxing The MFP Property

Does Camera work even when MFP does not hold? Camera as described in Section III relies on the Mutual Friend Property. We call that the *fast path* version of Camera. We now describe the *slow path*, which ensures safety even when MFP is untrue.

A. Identifying MFP Violations

Camera detects violations of MFP. We rely on an an estimator for system size, e.g., [27], which uses both active and passive approaches. We use this estimator as a black box. Such an estimator provides an upper bound on system size, which we label as $N.upper$. When a process p_i initiates its request, it switches to the slow-path if $length(Memlist_i) \leq \frac{N.upper}{2}$. False positives are possible but rare because membership lists are *mostly* consistent in practice (Subsection II-B).

In case stale entries (failed nodes whose failure p_i doesn't yet know about) make $Memlist_i$ large enough to use the fast path, p_i will find out about their failures while waiting for the OKs. Upon learning of each failure, p_i will recheck if it needs to switch to the slow path, making slow-path detection robust to failures and stale entries.

B. Slow Path Description

When requester p_i's membership list doesn't satisfy a quorum, we build a Breadth First Search (BFS) spanning tree with the requester as root, in order to propagate its request. We adopt the classical BFS algorithm from [28] (Algorithm 2.11 in Chapter 2), described here in Algorithm 2. By obtaining OKs from every node in the tree, Camera ensures that there is at least one node who both p_i and any other requester have asked for permission, and is robust to any combination of slow or fast paths being run by other nodes.

We make the following additions, none of which change the spanning tree algorithm.

- Piggy-back REQUESTs on top of the "join p + 1" message in line 3
- Copy topology information for f+1 levels atop the echo algorithm at line 6, so any given node knows the topology of the subtree rooted at itself up to depth f+1.

Because our membership graph is strongly connected, the resulting tree T includes every node in the network, and our piggy-backing of REQUEST messages guarantees that every node has received the REQUEST.

Once the tree is constructed, a node p_i sends an OK to its parent only when it has *both* received an OK from all its

Algorithm 2: Base Spanning Tree Construction

```
1 p = 1 do
2     Root sends "start p" through tree
3     Leaves send "join p + 1" to nodes they have not
      talked to yet
4     Upon first receiving "join p + 1", $p_i$ responds
      "ACK" to become leaf of tree
5     $p_i$ replies "NACK" to any additional "join"
      messages
6     Old leaves use echo algorithm to pass responses
      back to root, which then increments p
7 until no new node detected
```

children in T and has met the fast-path conditions (either chose to not defer or is clearing the defer buffer). The root enters when it has received an OK from all its children.

C. Slow-Path Safety

Lemma 10. *A node p_i will send its parent an OK only after all nodes in the subtree rooted at p_i (including itself) have said OK by fast-path conditions. Proof in IDEALS version [23].*

Theorem 11. *The slow path maintains safety. Proof in IDEALS version [23].*

Remark (Liveness)**.** *The liveness properties (deadlock-freedom and starvation-freedom) of the slow path also hold.*

This follows as in the fast path because there must be some highest priority request that thus receives all necessary OKs.

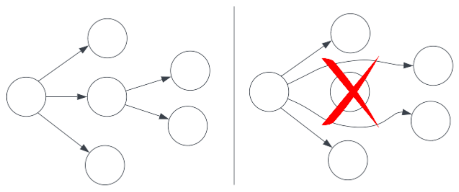


Fig. 7: **Camera Handling of Failures on Slow Path.**

a) Slow Path Fault Tolerance

If p_i fails during execution of the slow path, its parent adopts its children in the BFS tree. Figure 7 shows an example. Since each node knows the topology for the next $f+1$ levels, the slow-path can tolerate up to f failures between request initiation and CS entry.

VII. Experimental Results

We implemented Camera's mutual exclusion mechanism, and layered it over the open-source implementation of the Medley membership protocol [20]. We present simulation results driven by both synthetic workloads and real churn traces. We address 5 key research questions:

1) What is the *Wait Time*? It is defined as the gap from a request being initiated to requester executing the CS.

Term	Default Value
time unit	Simulation Time Unit $\simeq$ 1 ms (1000 time units $\simeq$ 1 second)
Trials per data point	50
(Message) Drop rate (per hop)	0.05
System size (N)	256
Concurrent requesters	30
IA:D (Interarrival:CS Duration)	1.0
Network Topology	Random
Per-Hop Network Delay	5 time units
Medley [20] ping period	2000 time units
Critical Section Duration	200 time units

TABLE I: **Key Experimental Simulation Parameters.**

2) What is the bandwidth incurred by Camera? We count *End-to-End Messages* (including those that are dropped).
3) What if requests arrive very frequently?
4) How does Camera deal with a real churn trace?
5) How does Camera compare to Ricart-Agrawala?

Table I shows default simulation values. Nodes are connected in an ad-hoc network topology, typical of an edge network scenario. Nodes are deployed randomly inside a 15×15 meter area, with transmission radius of 4 meters. We focus on a Random topology so we can investigate Camera-specific aspects; our underlying membership protocol Medley was already shown to be robust to a variety of topologies: Random, Grid, Cluster, and real maps [20]. As long as Medley assures MFP, Camera will be correct, making it agnostic to ad-hoc network topology. This ad-hoc network routes packets via the Optimized Link State Routing (OLSR), akin to the Medley simulator [20].

We perform three stages of simulations: (I) high contention workload (Section VII-A), (II) requester heterogeneity via spatial and temporal distributions (Section VII-B), and (III) churn trace-driven evaluations (Section VII-C).

A. High Contention Workload

We induce high contention by having *all* requesters *concurrently* (i.e., simultaneously) send out their request.

1) Concurrent Requesters

Because more concurrency means proportionally higher wait times, Figure 8 plots the *normalized* wait time, defined as wait time divided by the total number of concurrent requesters. This is essentially the handover time. Camera shows quick convergence of wait times (to 0.3 s) at small concurrency level (20 requesters).

We can intuitively explain the behavior in Figure 8 via a back-of-the-envelope analysis. Take m simultaneous requesters who each spend D time unit in the CS. In the contention-free case, where there is one node in the CS and only one more node waiting, let H time units denote the *hand-off time* (also known as synchronization delay in literature [10], [12]). Then in the m-requester case, the wait time for the first node that enters the CS is 0, for the second requester to enter CS is $D + H$, third requester is $2 \cdot (D + H)$, and so on until $(m - 1) \cdot (D + H)$. Hence the average wait time is $= \frac{\sum_{i=0}^{m-1}(D+H)\cdot i}{m-1} = \frac{(D+H)\cdot(m-1)\cdot m}{2\cdot(m-1)} = m \cdot \frac{D+H}{2}$. Normalizing

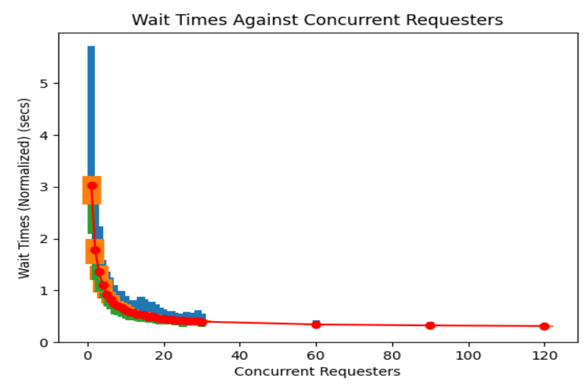


Fig. 8: **High Contention:** *Normalized wait times.*

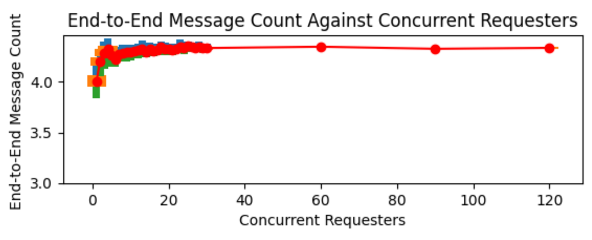
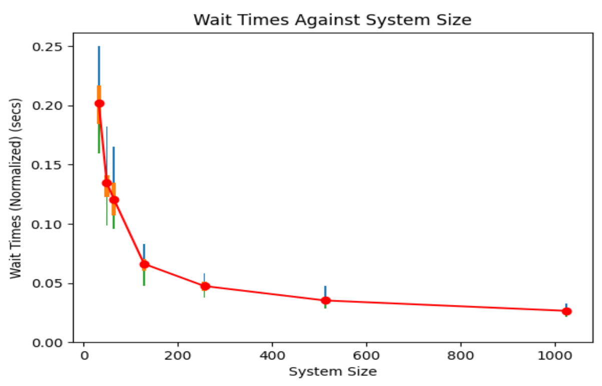
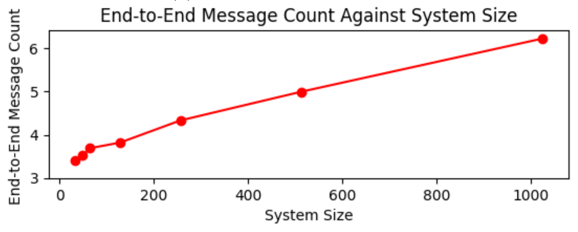


Fig. 9: **High Contention:** *Bandwidth. End-to-End Message Count against concurrent requesters.*



(a) Normalized wait times.



(b) End-to-End Message Count.

Fig. 10: **High Contention:** *Effect of System Size.*

this by dividing by m, gives $\frac{D+H}{2}$, a converged value that is visible in Figure 8. Because the default critical section duration is $D = 0.2$ s, we can estimate that the converged handoff time in Camera is 0.4 s. Finally, Figure 9 shows that Camera scales well on bandwidth, converging quickly to a constant number of messages per request per node.

2) System Size

Figure 10a shows that increasing the number of nodes causes the normalized wait time (normalized by dividing by N) to converge quickly beyond about 200 nodes to a value of 0.025 s. Wait time's linear increase with N is because an individual node's wait time is determined by the *maximum* OK message delay, which increases with system size but quickly converges (note that this is similar to the well-studied straggler problem [29]).

Figure 10b shows the bandwidth increases linearly with system size, which is also expected as underlying Ricart-Agrawala requires OKs from everyone. Specifically, Figure 10b's linear scaling implies that Camera's churn-tolerance techniques do not add any *additional* message overhead to Ricart-Agrawala. Further churn experiments are in Section VII-C.

3) Effect of Churn: Message Drop Rate

Membership lists can be churned due to false positives, which in turn largely arise due to dropped messages (because a dropped message from a node may cause the underlying membership protocol to mistakenly mark that node as failed). We measure how such churn adversely affects Camera's behavior.

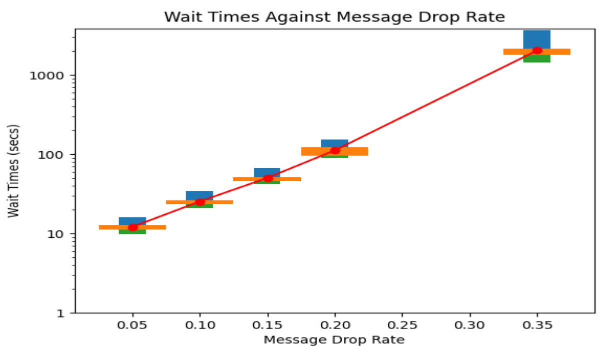
Figure 11 shows that both bandwidth and wait times (raw, not normalized) grow exponentially with drop rate. This is expected, since handover time at high concurrent requesters is dominated by the message delivery time from the CS-exiting node to the next-incoming node, and this expected message delivery time grows exponentially with hop-to-hop (H2H) drop rate. If H2H drop rate is p_{H2H}, then a path with m links has an end-to-end (E2E) drop rate $p_{E2E} = 1 - (1 - p_{H2H})^m$, i.e., $(1 - p_{E2E}) = (1 - p_{H2H})^m$. Hence E2E latency rises exponentially with H2H drop rate. Nevertheless, even at high drop rates of 20%, increase in wait times and bandwidth are less than an order of magnitude compared to a 5% drop rate.

B. Requester Heterogeneity via Distributions

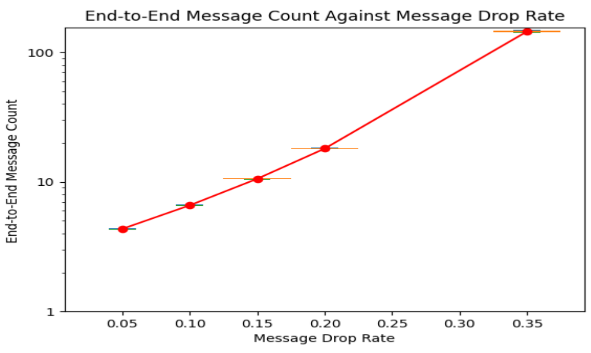
Some nodes may request the CS more often than others. Further, requests may arrive more burstily at times. We injected workloads with both spatial and temporal distributions, wherein: (i) (Spatial) the choice of requester was either Zipf or uniform (where each node is equally likely to be chosen), and (ii) (Temporal) system-wide CS request interarrival times were based on a distribution, either Weibull or exponential. Weibull distributions are known to represent bursty arrivals [30], while exponential distributions are memoryless.

1) Interarrival Time v.s. CS Duration

We vary *IA:D*, the ratio of average interarrival time (IA) to the CS execution duration (D), with the latter fixed at $D = 0.2$ s. A higher ratio means slower arrivals, so we expect lower wait times. Figure 12a shows that under uniform spatial distribution Camera's wait times decrease with IA:D. The Zipf distribution has consistently lower wait times, independent of both IA:D ratio and the interarrival distribution. Zipf's

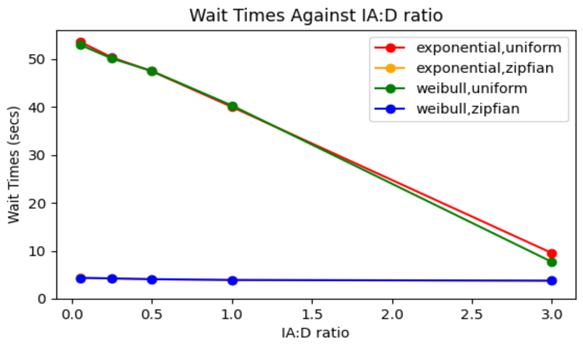


(a) Wait times.

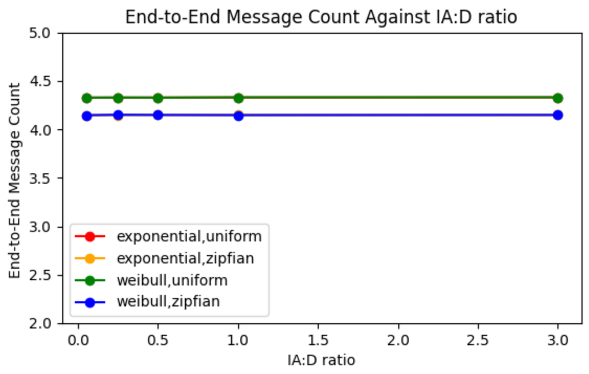


(b) End-to-End Message Count.

Fig. 11: **High Contention:** *Effect of Drop Rate.*



(a) Wait times.



(b) E2E Messages.

Fig. 12: **Requester Heterogeneity:** *Effect of IA:D.*

concentration of requesters results in: 1) fewer instances of finding out about potentially conflicting requesters and needing corrective action via Camera's techniques, and 2) lower contention since a given node sends only one request at a time (locally serializing its own requests). That advantage becomes less prominent when requests arrive slower (high IA:D), so the performance of the two distributions converges. Figure 12b shows that total end-to-end messages are independent of IA:D, showing Camera's bandwidth overhead scales.

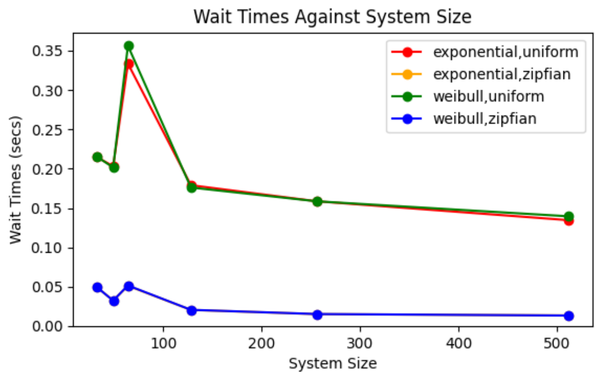
2) *System Size*

Figure 13a reveals that Camera handles the more challenging Zipf distribution rather well: average wait times (normalized) are lower under Zipf than under uniform. The advantage of concentrated requesters under the Zipf distribution is enhanced in large systems, with fewer lingering OKs (which are the primary cause of the upward trend). Nevertheless, the wait time scales well with N. Figure 13b shows that the number of messages (normalized with system size) sent by each node to service each request converges quickly, and thus is also scalable with system size.

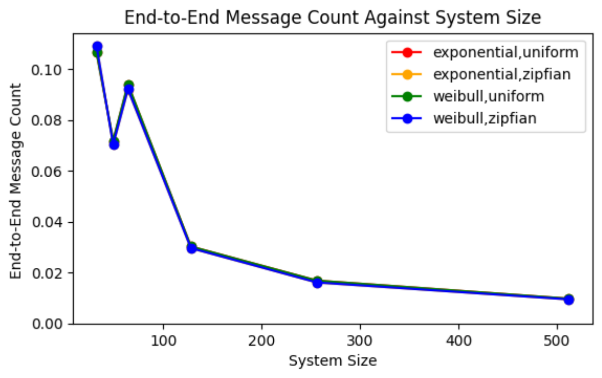
C. *Injecting a Real Churn Trace*

Since churn is well-studied in peer-to-peer (P2P) systems, we use a real P2P system churn trace. P2P systems have high rates of churn, up to 25-100% *per hour* [13], [14]. We inject the prober traces collected from the Overnet P2P system by Bhagwan, Savage, and Voelker [13], representing about 1400 clients sampled at 20 minute intervals. Camera's system size was 256 nodes, so we picked a random 256 of Overnet's 1400 nodes, and injected their arrival+departure behavior into our system. While our analysis assumed failures were independent of each other, churn traces naturally contain *correlated* failures. We replay the churn trace faster than the original—the *churn forward rate* indicates the ratio of time in the original (Overnet) trace in seconds to simulation time units. Higher churn forward rates mean Camera is subjected to more frequent churn. For example, a churn forward rate of 1.0 means a 20 min (1200 sec) segment of the original Overnet trace trace is replayed in Camera in 1200 time units (1.2 sec), while a higher churn forward rate of 60.0 means the same 20 min Overnet trace is replayed much quicker in just 20 Camera time units (0.02 sec).

Figure 14a shows with increasing churn rates, Camera's wait times converge quickly and thereafter remain flat. Intuitively, churn affects wait times in Camera only when a requester finds out about a new node right before entering the CS, and consequently needs to wait an extra RTT. This occurs fairly rarely in Camera, even under high churn. We observe the message count remains flat–requesters may ask the same nodes multiple times, but also may need to ask fewer nodes (some have churned out). We conclude that Camera behaves gracefully under high churn.

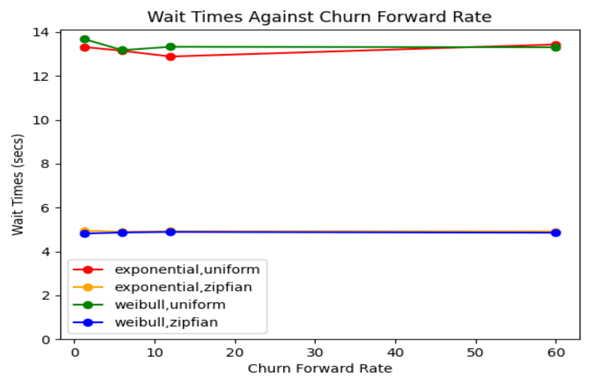


(a) Wait times.

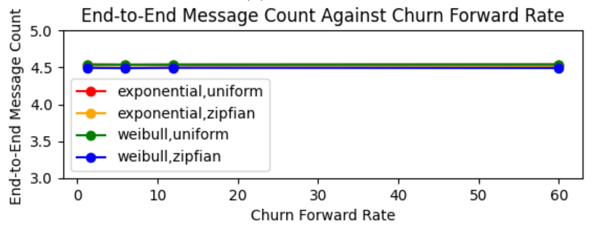


(b) E2E Messages per node.

Fig. 13: **Requester Heterogeneity:** *Effect of System Size.*



(a) Wait times.



(b) E2E Messages per node per request.

Fig. 14: **Real Churn Trace Injected into Camera.**

Request Distribution	RA BW	Cam. BW	Overhead
exponential, uniform	11.996	22.321	86.073%
exponential, zipfian	12.114	22.210	83.344%
weibull, uniform	12.030	22.248	84.938%
weibull, zipfian	12.110	22.183	83.185%

TABLE II: **Camera vs. Baseline RA: Bandwidth (MB)**

D. Comparison to Ricart-Agrawala

By extending RA [10] with recentlyOKed sets, Camera requires extra bandwidth, memory, and compute. Concretely: (i) (Bandwidth) for each O(1) sized message that RA sends, Camera sends a O(—recentlyOKed—)-sized message; (ii) (Memory) only additional memory in Camera is the recentlyOKed set (the other memory is the membership list, needed by both RA and Camera); and (iii) (Compute) when any OK message is received, RA checks O(1) membership entries, while Camera checks O(—recentlyOKed—) membership entries.

Given that each overhead is proportional to the recentlyOKed set size, we capture all three by comparing Camera's bandwidth overhead vs. RA's. For a fair comparison, we only consider scenarios where membership stays consistent (so baseline RA maintains safety). Table II shows that Camera's bandwidth is only 85% over RA's, validating that recentlyOKed sets are small in practice. We believe this overhead is acceptable in return for churn-tolerance.

E. Comparison to Wireless Paxos

We compare against the well-known Wireless Paxos [31], using the Cooja simulator [32]. Figure 15 shows normalized wait time (over 100 rounds) vs. number of nodes for Wireless Paxos, along with Zipf workloads on Camera. First we note that Wireless Paxos had difficulty scaling to even 32 nodes, and did not run beyond 32 nodes. Second, Camera scales well to much higher system sizes. Essentially Camera "continues" where Wireless Paxos "left off" ($N = 32$), serving as a scalable alternative to Wireless Paxos.

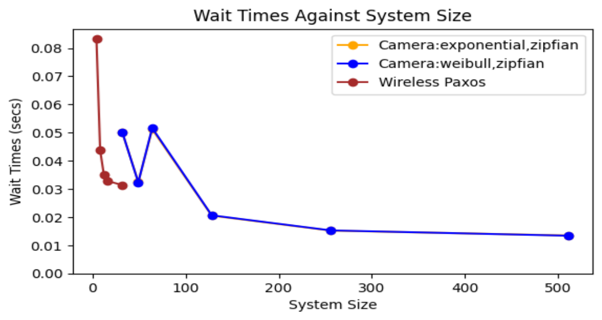


Fig. 15: **Wireless Paxos vs. Camera (2 distributions).** *Wireless Paxos did not run beyond 32 nodes.*

VIII. Discussion and Other Related Work

Why not Central Solutions? Centralized solutions to mutual exclusion are difficult in edge scenarios due to remoteness (e.g., smart farms: no servers among corn) and high churn (e.g., satellites: too fast for ground station coordination). Hence these devices must coordinate using local links (Wifi, LoRa, Inter-satellite links for LEO satellites) that create wireless ad-hoc networks. Further, if the critical section resource is physical (e.g., a file), using it to arbitrate requests would overwhelm the central node.

Why Not Consensus Solutions? Apart from our empirical comparison to Wireless Paxos we comment more generally on consensus solutions. Protocols like Zookeeper [33], Chubby [34], Raft [35], Mencius [36], etc., are used in datacenters to solve mutual exclusion at small scales, typically in a group of 3 to 5 processes that arbitrate all requests. Beyond these group sizes, it is known that the throughput of consensus solutions drops [33]–[36]. Using them to solve mutual exclusion in ad-hoc edge settings would require either: (1) designing scalable consensus protocols (a known hard problem), or (2) entrusting a small consensus group with arbitrating all requests. The latter (i.e., (2)) entails an overloaded "central" group, untenable in edge settings. A major difficulty in scaling consensus to churned edges is that consensus relies on membership lists being repaired immediately after each failure—strong membership cannot do this at high churn rates. Solving consensus for settings that are large-scale *and* churn-tolerant remains an open problem (and one that we do not tackle in this paper).

Partitions: Partitions are hard in all agreement-related problems, from consensus to mutual exclusion [37]. Nevertheless, once a partition is repaired, underlying weak consistent membership repairs membership lists quickly [20]–[22], re-establishing MFP, and consequently Camera's properties.

Other Related Work: Classical solutions to the mutual exclusion problem can be divided into *permission-based* and *token-based* approaches. The former includes Ricart-Agrawala [10], and Maekawa's $\sqrt{N}$ Algorithm [12]. Token-based approaches include Raymond's algorithm [11], overlaying a logical tree (with token holder as root) on the network.

Some mutual exclusion algorithms in ad-hoc networks [38], [39], [40] are token-based. Some [41] are efficient under high mobility, but underperform when mobility is limited. Even permission-based approaches that modify Ricart-Agrawala assume stable mobile support stations [42].

In broader edge systems there has been work on distributed hash tables [43], bounded degree topologies [44], grid-based virtual filesystems [45], and consensus [46]. Existing mutual exclusion in P2P DHTs [47] only provide probabilistic safety, but not provable 100% safety (under the stated assumptions) like Camera.

IX. Summary

We presented the first churn-tolerant mutual exclusion algorithm intended for ad-hoc edge and IoT networks. Camera is a modified variant of the classical Ricart-Agrawala algorithm, runs over over weakly-consistent membership, and ensures safety and liveness under churn. Trace-driven simulations—under high workload, spatial and temporal request distributions, and real P2P churn traces—all indicate that Camera scalably incurs low bandwidth overhead (compared to the baseline) while resolving mutual exclusion requests quickly, scalably, and under high churn.

References

[1] Marketsandmarkets, "Iot market by component (hardware, software solutions and services), organization size, focus area (smart manufacturing, smart energy and utilities, and smart retail) and region - global forecasts to 2026," Marketsandmarkets, Tech. Rep., 2022.

[2] L. Tightiz and H. Yang, "A comprehensive review on iot protocols' features in smart grid communication," in *Energies*, vol. 13, 2020.

[3] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon., "Wireless sensor networks for structural health monitoring," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, October 2006, pp. 427–428.

[4] M. R. Yuce and C. K. Ho, "Implementation of body area networks based on mics/wmts medical bands for healthcare systems," in *2008 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. IEEE, August 2008, pp. 3417–3421.

[5] A. N. Sivakumar, S. Modi, M. V. Gasparino, C. Ellis, A. E. Velasquez, G. Chowdhary, and S. Gupta, "Learned visual navigation for under-canopy agricultural robots," *arXiv e-prints*, 2021.

[6] Y. Guo, P. Corke, G. Poulton, T. Wark, G. Bishop-Hurley, and D. Swain, "Animal behaviour understanding using wireless sensor networks," in *2006 31st IEEE Conference on Local Computer Networks*. IEEE, 2006, pp. 607–614.

[7] T. Bokareva, W. Hu, S. Kanhere, B. Ristic, N. Gordon, T. Bessell, M. Rutten, and S. Jha, "Wireless sensor networks for battlefield surveillance," in *Proceedings of the Land Warfare Conference*, October 2006, pp. 1–8.

[8] B. Denby and B. Lucia, "Orbital edge computing: Nanosatellite constellations as a new class of computer system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[9] D. Cho, "A redundant sensing elimination technique for improving energy efficiency of iot sensor networks," in *Journal of Physics: Conference Series*, vol. 1927. IOP Publishing, May 2021, p. 012001.

[10] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," in *Communications of the ACM*, vol. 24. IEEE, Jan. 1981, pp. 9–17.

[11] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," in *ACM Transactions on Computer Systems (TOCS)*, 1989, pp. 61–77.

[12] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," in *ACM Transactions on Computer Systems (TOCS)*, 1985, pp. 145–159.

[13] R. Bhagwan, S. Savage, and G. M. Voelker, "Understanding availability," in *Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003*. Springer Berlin Heidelberg, 2003, pp. 256–267.

[14] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a dht," in *Proceedings of the USENIX annual technical conference*, vol. 6, 2004, pp. 127–140.

[15] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the impossibility of group membership," in *Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, 1996, pp. 322–330.

[16] K. P. Birman, "Replication and fault-tolerance in the isis system," in *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1985, pp. 79–86.

[17] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, "The totem single-ring ordering and membership protocol," in *ACM Transactions on Computer Systems (TOCS) 13*, 1995, pp. 311–342.

[18] K. Birman, B. Constable, M. Hayden, J. Hickey, C. Kreitz, R. V. Renesse, O. Rodeh, and W. Vogels, "The horus and ensemble projects: Accomplishments and limitations," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, 2000, pp. 149–161.

[19] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," in *ACM Transactions on Computer Systems (TOCS)*, 1999, pp. 41–88.

[20] R. Yang, J. Wang, J. Hu, S. Zhu, Y. Li, and I. Gupta, “Medley: A membership service for iot networks,” in *IEEE Transactions on Network and Service Management 19*, vol. 3, 2022, pp. 2492–2505.

[21] Z. Bar-Yossef, R. Friedman, and G. Kliot, “Rawms-random walk based lightweight membership service for wireless ad hoc networks,” in *ACM Transactions on Computer Systems (TOCS)*, vol. 26, 2008, pp. 1–66.

[22] G.-C. Roman, Q. Huang, and A. Hazem, “Consistent group membership in ad hoc networks,” in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE, 2001, pp. 381–388.

[23] A. Khinvasara and I. Gupta, “Camera: Churn-tolerant mutual exclusion for the edge,” Oct 2024. [Online]. Available: https://hdl.handle.net/2142/124923

[24] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[25] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems*. Addison Wesley, 2011.

[26] U. Tadakamalla and D. A. Menascé, “Characterization of iot workloads,” in *Edge Computing–EDGE 2019: Third International Conference*, S. 2019, Ed., vol. 3. Springer International Publishing, 2019, June 2019, pp. 1–15.

[27] L. Massoulié, E. L. Merrer, A.-M. Kermarrec, and A. Ganesh, “Peer counting and sampling in overlay networks: random walk methods,” in *PODC ’06: Proceedings of the twenty-fifth annual ACM Symposium on Principles of Distributed Computing*, 2006, pp. 123–132.

[28] R. Wattenhofer, “Principles of distributed computing (fs 2023),” 2023, [Course Website from ETH Zurich. Chapter 2 Lecture Notes - Algorithm 2.11]. [Online]. Available: https://disco.ethz.ch/courses/podc/

[29] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, “Effective straggler mitigation: Attack of the clones,” in *NSDI*, vol. 13, 2013, pp. 185–198.

[30] A. Arfeen, K. Pawlikowski, D. McNickle, and A. Willig, “The role of the weibull distribution in modelling traffic in internet access and backbone core networks,” in *Journal of Network and Computer Applications*, vol. 141, 2019, pp. 1–22.

[31] V. Poirot, B. A. Nahas, and O. Landsiedel, “Paxos made wireless: Consensus in the air.” in *International Conference on Embedded Wireless Systems and Networks*, 2019, pp. 1–12.

[32] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. IEEE, 2004, pp. 455–462.

[33] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: wait-free coordination for internet-scale systems,” in *USENIX Annual Technical Conference*, vol. 8, June 2010.

[34] M. Burrows, “The chubby lock service for loosely-coupled distributed systems,” in *7th Symposium on Operating Systems Design and Implementation*, November 2006, pp. 335–350.

[35] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *USENIX Annual Technical Conference*, 2014, pp. 305–319.

[36] Y. Mao, F. P. Junqueria, and K. Marzullo, “Mencius: Building efficient replicated state machines for wans,” in *8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[37] D. Barbara and H. Garcia-Molina, “Mutual exclusion in partitioned distributed systems,” in *Distributed Computing 1*, 1986, pp. 119–132.

[38] R. Baldoni, A. Virgillito, and R. Petrassi, “A distributed mutual exclusion algorithm for mobile ad-hoc networks,” in *Proc. of ISCC’02, IEEE Computer Society*, 2002, pp. 539–544.

[39] N. Malpani, Y. Chen, N. H. Vaidya, and J. L. Welch, “Distributed token circulation on mobile ad hoc networks,” in *Proc. of ICNP’01, IEEE Computer Society*, 2001, pp. 4–13.

[40] J. E. Walter and S. Kini, “Mutual exclusion on multihop, mobile wireless networks,” in *Technical Report, Dept. of Computer Science, Texas AM Univ.*, 1997.

[41] J. E. Walter, J. L. Welch, and N. H. Vaidya, “A mutual exclusion algorithm for ad hoc mobile networks,” in *Wireless Networks 7*. Kluwer Academic Publishers, 2001, pp. 585–600.

[42] M. Singhal and D. Manivannan, “A distributed mutual exclusion algorithm for mobile computing environments,” in *Proc. of ICIIS’97, IEEE Computer Society*, 1997, pp. 557–561.

[43] S. Legtchenko, S. Monnet, P. Sens, and G. Muller, “Relaxdht: A churn-resilient replication strategy for peer-to-peer distributed hash-tables,” in *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2012, pp. 1–18.

[44] J. Augustine, G. Pandurangan, P. Robinson, S. Roche, and E. Upfal, “Enabling robust and efficient distributed computation in dynamic peer-to-peer networks,” in *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, 2015, pp. 350–369.

[45] L. Lindbäck, V. Vlassov, S. Mokarizadeh, and G. Violino, “Churn tolerant virtual organization file system for grids,” in *Parallel Processing and Applied Mathematics: 8th International Conference*, vol. 2, 2009, pp. 194–203.

[46] J. Augustine, G. Pandurangan, P. Robinson, and E. Upfal, “Towards robust and efficient computation in dynamic peer-to-peer networks,” in *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, 2012, pp. 551–569.

[47] S.-D. Lin, Q. Lian, M. Chen, and Z. Zhang, “A practical distributed mutual exclusion protocol in dynamic peer-to-peer systems,” in *Peer-to-Peer Systems III: Third International Workshop, IPTPS 2004*. Springer Berlin Heidelberg, Feb. 2004, pp. 11–21.