CAMERA: CHURN-TOLERANT MUTUAL EXCLUSION FOR THE EDGE

BY

AMAN KHINVASARA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of B.S. in Computer Engineering
in the Grainger College of Engineering of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Adviser:

Professor Indranil Gupta

ABSTRACT

Mutual exclusion is an essential primitive in distributed systems to ensure at most one process at a time accesses a shared resource. While classical distributed mutual exclusion algorithms assume full, consistent membership, today's IoT and adhoc networks are characterized by high churn that often leaves membership inconsistent and incomplete. We extend a classical mutual exclusion algorithm by taking advantage of the observation that any two nodes typically have a common friend, even if they don't know each other. The presented algorithm is tolerant to churn, robust to heterogeneity in membership lists, bandwidth-efficient, and degrades gracefully. We accompany this algorithm with formal proofs of safety, starvation-freedom, and deadlock-freedom; a slow path ensuring safety even when common-case assumptions happen to be false; and experimental results validating performance at scale.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Edge computing systems are quickly becoming critical infrastructure in the modern world. Driven by falling hardware costs, ubiquitous, data-hungry AI, and burgeoning use cases, the Internet of Things (IoT) market is expected to grow 16.7% YoY, reaching $650.5 billion in 2026 [1]. The plethora of use cases can be viewed as comprising two categories - static and dynamic environments. Examples of the former include smart manufacturing, smart retail [1], smart 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, with use cases such as smart transportation (cargo monitoring, fleet management, autonomous vehicles) [1], smart agriculture [5], defense [6], on-body healthcare [4], and livestock management [7].

These deployments often contain shared resources which require coordination to access, such as sensors attached to grazing cattle that need to take turns writing into a given file, or reconnaissance robots in the battlefield taking turns surveying a particular area. In other cases, network performance can improve by executing the CS code at only one node; for example, better power efficiency can be achieved by executing environment sensing at only one IoT device [8]. Additionally, mutual exclusion algorithms must satisfy safety and liveness, providing freedom from deadlock and starvation.

Classical solutions to mutual exclusion [9], [10], [11] shared the flawed assumption of strong membership, where each node always knows every other node. However, harsh, dynamic environments are characterized by lossy, bandwidth constrained networks and high churn. They run weak membership protocols that can not guarantee that each node always knows about all others, but only deliver membership changes *eventually* (though often quickly), e.g., Medley [12]. We show that classical mutual exclusion solutions break down under such membership models. While several mutual exclusion algorithms have been proposed for static environments, no mutual exclusion algorithm for ad hoc networks exists, with truly dynamic, inconsistent membership (allowing nodes to leave and join).

Dynamic environments present significant challenges for mutual exclusion. Membership inconsistency is heterogeneous across nodes; for example, one node may be missing a quarter of the network from its membership list while another is missing only one node. Bandwidth scarcity requires efficiency, and the critical use cases require graceful degradation and *provable* safety even when different nodes have inconsistent system views.

Contributions This paper presents the first permission-based mutual exclusion algorithm that is churn-tolerant. Our algorithm and system, named Camera ($\underline{\text{C}}$hurn-toler$\underline{\text{a}}$nt $\underline{\text{M}}$utual $\underline{\text{E}}$xclusion by extending $\underline{\text{R}}$icart-$\underline{\text{A}}$grawala)) extends the classical mutual exclusion algorithm that was originally by Ricart and Agrawala [9], and imbues it with churn-tolerance. We leverage the key observation that, with very high probability, ad hoc membership protocols [12] result in membership graphs where any two nodes either know each other or have a third node they both know—we call this the *Mutual Friend Property (MFP)*. Camera uses this property to nudge the system back into a correct state.

Our key contributions are:

1. Camera, an efficient algorithm that ensures mutual exclusion in ad hoc networks with inconsistent, dynamic membership;

2. Formal proofs of correctness of safety, starvation-freedom, and deadlock-freedom;

3. A detection mechanism and fall-back algorithm for when the MFP does not hold; and

4. Experimental results showing that synchronization delay falls under contention and zipfian-distributed requesters produce wait times independent of request arrival rate

CHAPTER 2

BACKGROUND

2.1 System Model

We adopt an asynchronous system model consisting of up to N nodes at a time, allowing churn. The communication medium can arbitrarily delay messages but will not drop them. Clocks are not synchronized. We adopt a fail-stop model, and exclude Byzantine failures. These nodes must coordinate to execute a 'critical section' one at a time, known as the mutual exclusion problem. Note that multiple requests at a single node should be serialized by the node itself, below the Camera level. This is the same system model as the original Ricart-Agrawala paper, with one key distinction - membership lists.

We assume a weak underlying membership protocol, and use traces from ad hoc network membership algorithm Medley [12] to fuel our simulation.

We also assume the membership protocol includes a **failure detector** that is eventually complete, so all non-faulty nodes eventually learn about all failures. However, arbitrary propagation delays for joins and fails/leaves are acceptable, as are false positives. Additionally, nodes can join or fail at any time.

2.2 Problem Statement

We now formally define the three key properties of mutual exclusion algorithms - safety, deadlock-freedom, and starvation-freedom.

Definition 1 (Safety)**.** Safety guarantees that no more than one node will enter a critical section at any point in time.

Definition 2 (Deadlock-freedom)**.** Freedom from deadlock, the state in which when no node is in its critical section and no node can ever proceed into its critical section.

Definition 3 (Starvation-freedom)**.** Freedom from starvation, the state in which some node is waiting indefinitely to enter its critical section while other nodes continue entering and exiting.

Deadlock-freedom and starvation-freedom together constitute liveness in mutual exclusion algorithms.

We also assume that the membership graph G is strongly connected - which is a minimal assumption to maintain safety and liveness:

Theorem 1. *Let ALG be a mutual exclusion algorithm with the following assumptions:*

- *Any node can run* `enter()` *at any time*
- *Messages can only be sent along edges in the membership graph*
- *'Information' can only be conveyed across processes via messages*

Then, a strongly connected membership graph is necessary for ALG to satisfy safety and liveness.

The proof is in Appendix A.

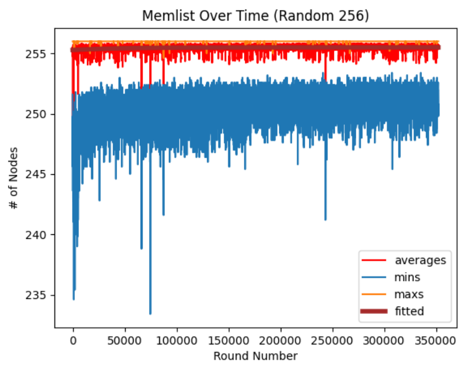
2.3 Mutual Friend Property

Definition 4 (Mutual Friend Property (MFP))**.** Write $Memlist_i$ to denote the membership list at node p_i. For every pair of nodes p_i, p_j requesting access to the critical section, their membership lists intersect in at least one node; formally, $Memlist_i \cap Memlist_j \neq \emptyset$

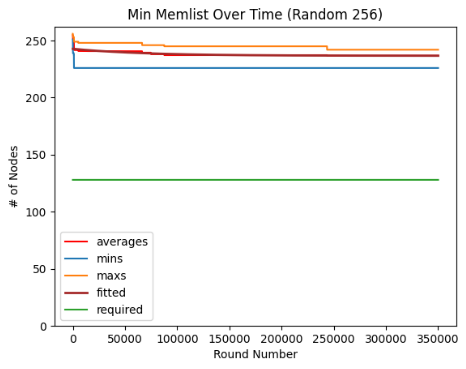
The Mutual Friend Property says any pair of requesting nodes must either know each other, or have at least one other node they both know. In particular, this property still allows nodes p_i, p_j to not be *directly* aware of each other. Camera provides safety under the Mutual Friend Property, and we present an extension that maintains safety under strongly connected membership. We additionally observe that the Mutual Friend Property holds in most practical ad hoc systems, as weak membership protocols typically still provide membership lists containing more than half the system.

Lemma 2. *Let* $|Memlist_i| > \frac{N}{2}$ *at all simultaneously requesting* p_i*, where* N *is the total number of nodes in the system. Then, the Mutual Friend Property is satisfied, because any two quorums intersect in at least one node.*

Simulation results, such as those from Medley [12], show that membership lists are quorums with *very* high probability. Figure 2.1 shows the result of running Medley for 350,000 rounds on a random topology of 256 nodes with a default hop-to-hop drop rate of 0.05. Subfigure 2.1a shows the sizes of the membership lists over time, smoothed with a width-3 rolling average. The best exponential fit for the average is $-0.2157e^{-0.0000223x} + 255.50$, suggesting it rapidly stabilizes to 255 nodes, much higher than the required quorum. In Subfigure 2.1b, the average size of the smallest membership list each node has *ever* seen resembles the exponential $6.1057e^{-0.0000145x} + 236.69$ and stabilizes at 240, far higher than the 129 quorum requirement. Thus, MFP will be satisfied with high probability.



(a) Membership Size



(b) Min Membership Size

Figure 2.1

CHAPTER 3

BACKGROUND: RICART-AGRAWALA

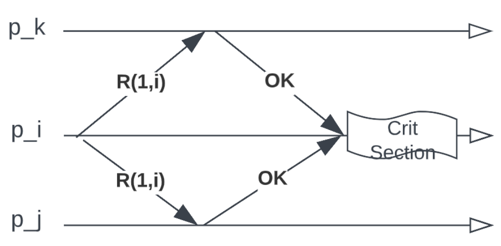
We begin by understanding the original Ricart-Agrawala algorithm and how it fails under inconsistent membership induced by churn.

3.1 Ricart-Agrawala Algorithm

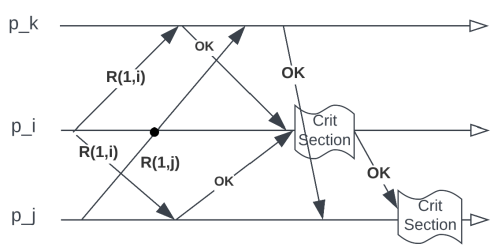
The following is a summary of Ricart-Agrawala to accompany the pseudocode in Algorithm 1. Node p_i, wishing to enter the CS, sends a REQUEST message with (sequence_number,p_i) to every node in its membership list. p_i enters its CS only after receiving OKs back from *all* its peers. When receiving a request from p_j, it buffers the request if it is executing the CS or if its own outstanding request has priority (lower sequence number) over p_j's. Otherwise, it immediately responds with an OK. Upon exiting, it sends an OK in response to all buffered requests.

In Figure 3.1a, p_i is requesting access to the CS. p_i begins by sending a Request(1,p_i) message to every other node. Each independently receives the request, doesn't defer because it is neither in the CS nor waiting to enter, and responds with an OK. p_i thus enters the CS.

Now, p_j also attempts to enter the CS in Figure 3.1b. Under Ricart-Agrawala's assumption of complete membership, p_i and p_j know each other



(a) **Single Node Requesting**



(b) **Two Nodes Successfully Requesting**

Algorithm 1: Ricart-Agrawala Pseudo-code

```
Definitions:
enum State = {WAIT,HELD,NONE};
struct Request = {int seq_num, string id};

Initialization:
Set Deferred, Pending_OKs = ∅;
string me = "myId";
Request myRequest = None;
State myState = State.NONE;
int max_seq_seen = 0;

def enter():
    myState = State.WAIT;
    myRequest = (max_seq_seen + 1, 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 CS!

def exit():
    myState = State.None;
    for request in Deferred:
        Deferred.remove(request);
        send_message(OK,request.id);

def receive_request(Request req, string sender):
    max_seq_seen = max(max_seq_seen,req.seq_num);
    if myState = State.HELD or (myState = State.WAIT and req >
            myRequest): // we have priority over incoming request
        Deferred.insert(req);
    else:
        send_message(OK,req.id);

def receive_OKs(sender, senderRecentlyOKed, Request
  req):
    Pending_OKs.remove(sender);
```

(and p_k). They send their requests to each other and p_k. The latter receives the request, decides not to defer because it is neither in the CS nor waiting to enter, and responds with OK messages. p_j and p_i are each waiting for just one more OK - from each other! Both evaluate p_i's request as having higher priority because $i < j$, so p_j immediately sends an OK (allowing p_i into the CS) while p_i defers p_j's request. When p_i exits the CS, it empties the queue by sending OKs to all deferred requests, giving p_j the final OK it needs to enter its CS. p_i and p_j exemplify the intended operation of Ricart-Agrawala with complete membership lists.

3.2 Ricart-Agrawala Safety Violation

Operating Ricart-Agrawala under the strongly connected membership model can violate safety for a pair of requesting nodes p_i and p_j who don't know about each other. Because Ricart-Agrawala relies on nodes with higher priority to defer conflicting requesters from entering the CS, safety is maintained as long as the node with lower priority sends a request to the node with higher priority. On the other hand, the following theorem enumerates cases that can result in a safety violation under weak membership.

Theorem 3. *Let p_i and p_j be two requesting nodes, such that $Req_i < Req_j$. Under either of the two following conditions, there exists runs of Ricart-Agrawala in which safety is violated even under the Mutual Friend Property:*

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

Proof. It suffices to show examples of safety violation under each of these conditions, as follows:

Figure 3.2 shows an example of failure under Symmetric Ignorance, condition (1) above. Only two processes p_i, p_j request access to the CS. Following the normal operation of the Ricart-Agrawala algorithm, any nodes p_k belonging to either of the two membership lists respond OK to all received requests. However, if $p_i \notin Memlist_j$ and $p_j \notin Memlist_i$, each of them receives OKs from everyone in their respective membership lists and proceeds into the CS, thus violating safety.

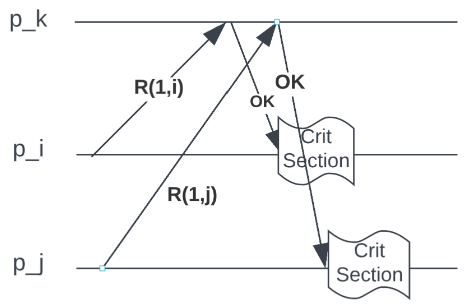


Figure 3.2: **Symmetric Ignorance Failure**

Figure 3.3 shows an example of failure under Asymmetric Ignore, condition (2) above. 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 respond with OK immediately, 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 and enters the CS, violating safety.

$\square$

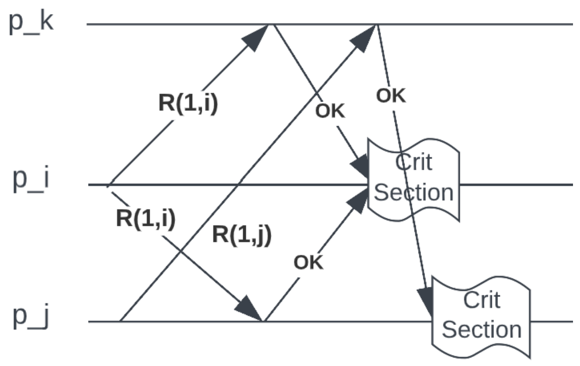


Figure 3.3: **Asymmetric Ignorance Failure** **(**$p_i \notin Memlist_j$**)**

CHAPTER 4

CAMERA

4.1 Camera Design

In this chapter we present our churn-tolerant variant of the Ricart-Agrawala algorithm from section 3.1 [9].

The primary issue with running Ricart-Agrawala under churn is that requesting nodes may not know each other. The key intuition is that Camera leverages the Mutual Friend Property (which we expect to be commonly true), and uses such 'overlapping' nodes to nudge the system back to a correct state.

Camera extends Ricart-Agrawala as follows. Whenever p_j responds OK to a request, it also includes a list of all nodes to whom it has *recently* said OK. We expect this set to be small, because the tight resource constraints in most ad hoc systems mean only a few nodes would request CS access simultaneously. For example, IoT transportation settings find 95% of requests have interarrival times greater than 100ms [13]. When a requesting node receives an OK message with the recently_OKed set, it checks that set for any nodes it does not know about. It adds those new nodes to its own membership list and sends them the REQUEST, waiting for those OKs before entering. Algorithm 2 describes the operation of Camera in further detail.

The safety violation that occurs in Ricart-Agrawala under Asymmetric Ignorance from Case 2 in Section 3.2 (only one node does not know the other) is solved by a few additional lines of code in the process to receive REQUEST messages. When the requester p_i is unknown by the receiving node p_j, p_j adds p_i to its membership list and, if it's in the WAIT state, sends p_i its own REQUEST as well, incrementing the number of OKs it must wait for.

When we have Symmetric Ignorance ($p_i \notin Memlist_j$ and $p_j \notin Memlist_i$),

Algorithm 2: Camera Pseudo-code

```
Definitions:
enum State = {WAIT,HELD,NONE};
enum Path = {SLOW, FAST};
struct Request = {int seq_num, string id, Path path};
Initialization:
Set Deferred, Recently_OKed, Pending_OKs = ∅;
string me = "myId";
Request myRequest = None;
State myState = State.NONE;
int max_seq_seen = 0;
def enter():
    myState = State.WAIT;
    max_seq_seen += 1;
    myRequest = (max_seq_seen, me, Path.FAST);
    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(Recently_OKed),request.id);
        Recently_OKed.insert(request);
```

Algorithm 2: 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_seq_seen = max(max_seq_seen,req.seq_num);
    if myState = State.HELD or (myState = State.WAIT and req >
            myRequest): // we have priority over incoming request
        Deferred.insert(req);
    else:
        send_message(OK(Recently_OKed),req.id);
        Recently_OKed.insert(req);

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

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

the common friend p_k will inform the second requester about the first, nudging the system back into a correct state. Suppose p_k, known by both p_i and p_j, receives p_i's request before p_j's. p_k will respond OK to p_i, and will include p_i in the OK to p_j. p_j will thus forward its request to p_i, informing them of each other. The system will be forced back into a correct informational state, and critical section entry will be delayed at p_j until p_i exits, maintaining safety. The following exposition presents examples of both cases.

Camera in Action

Here are examples of Camera maintaining safety when Ricart-Agrawala would have violated it. p_k is the node that both p_i and p_j know about. Figure 4.1 depicts p_i and p_j ignorant of each other. p_i and p_j simultaneously send REQUEST messages to p_k, who happens to receive p_i's before p_j's. p_k responds to p_i with OK, so p_i enters the critical section. p_k responds to p_j with OK(1,p_i), so p_j knows to relay its own request to p_i. p_i defers responding until after exiting the critical section, preventing p_j from violating safety. After exiting, p_i responds to the deferred message with OK, letting p_j into the critical section. Thus, we see how Camera solves the safety issues resulting from requesters under Symmetric Ignorance.

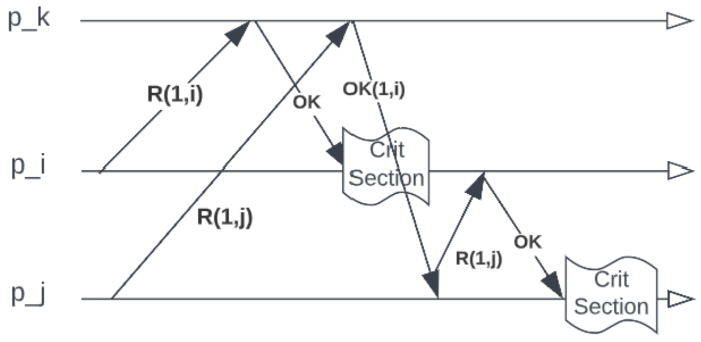


Figure 4.1: **Camera Success Despite** $p_i \notin Memlist_j$ **and** $p_j \notin Memlist_i$

Figure 4.2 shows the simpler solution for the Asymmetric Ignorance case (2). When p_i's request arrives at p_j, p_j adds p_i to its membership list and relays its request to p_i. p_i has higher priority and thus waits until exiting before sending the final OK allowing p_j into the CS. Thus, Camera also solves the safety issue resulting in Ricart-Agrawala from Asymmetric Ignorance.

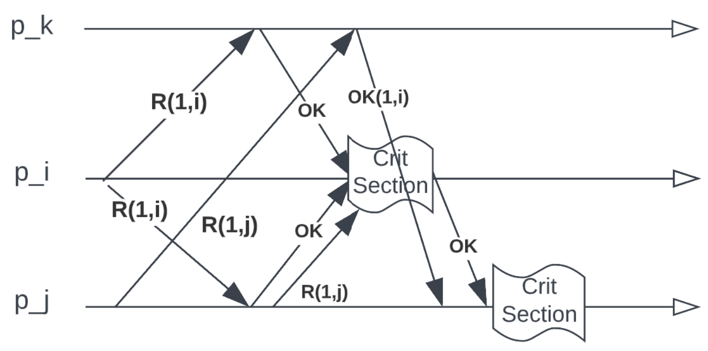


Figure 4.2: **Camera Success Despite** $p_i \notin Memlist_j$

Retention Period

The cached OK responses in the algorithm (line 27 of Algorithm 2 and line 13 of Algorithm 2 (cont.)) need to be retained long enough to not violate safety. While the size of this list is bounded by N (since each node has at most one outstanding request), this still consumes large memory resources. We discuss here how retention periods can be shortened. The key idea is to send RELEASE messages eagerly on exiting the critical section, allowing the recipients to clear the corresponding OKs.

Safety only requires retaining a given node in the OK cache before and while it is executing its critical section. As soon as it has exited, it sends a RELEASE message to everyone who sent it an OK, so they can prune its request from their OK cache. A RELEASE message arriving after the next REQUEST from the same node will simply be discarded. We use raw IP multicast to optimize bandwidth, due to the following result:

Lemma 4. *Dropped RELEASE messages do not cause safety or liveness violations.*

Proof. Let p_i exit its critical section and send a RELEASE message that drops. Nodes in $Memlist_i$ maintain the cached OK, so future requesters send their REQUEST to p_i as well. If p_i is in NONE state (so it can't contribute to safety violations), it will simply send an OK (maintaining liveness). Otherwise, the algorithm will arbitrate requests as intended, maintaining safety and liveness. □

4.2 Camera Analysis

Safety

Recall that safety is the guarantee that at most one node is executing a critical section at any point in time.

Theorem 5. *Camera ensures safety.*

Proof. Assume the contrary - that two nodes p_i and p_j execute the critical section simultaneously. Assume without loss of generality that their requests are ordered such that $Req_i < Req_j$, generating four unique 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 $p_j \notin Memlist_i$

The first case represents the Ricart-Agrawala assumption of complete membership, while Case 2 and Case 4 are safety-violating Asymmetric Ignorance and Symmetric Ignorance from Section 3.2.

Case 1 - $p_i \in Memlist_j$ and $p_j \in Memlist_i$
Note the original Ricart-Agrawala algorithm also maintained safety in this case, as shown in Figure 3.1b, so the proof for Camera is straightforward. To enter, p_i and p_j would have each sent REQUEST and OK to the other. Recall $Req_i < Req_j$ means Req_j must have arrived at p_i after p_i initiated its own process (otherwise, p_i's sequence_number $\geq$ p_j's sequence_number + 1). Thus, there are only two sub-cases for when p_i receives Req_j.

1. p_i receives Req_j before receiving all pending OKs $\rightarrow$ p_i is WAITing, evaluates $Req_i < Req_j$, and defers
2. p_i receives Req_j after receiving all pending OKs $\rightarrow$ p_i is in HELD state and defers by default

In either sub-case, p_i sends an OK allowing p_j into the critical section only *after* p_i exits its own critical section, contradicting our assumption of a safety violation.

Case 2 - $p_i \notin Memlist_j$ and $p_j \in Memlist_i$
Figure 4.2 shows a visual example of Camera maintaining safety in this case. For p_i to be in the critical section, it must have sent Req_i to p_j and received an OK in response. There are three sub-cases for when p_j receives Req_i in relation to when p_j entered its critical section.

1. p_j receives Req_i while in its critical section $\rightarrow$ p_j defers responding until after exiting the critical section $\rightarrow$ i receives its OK and enters the critical section after p_j exits $\rightarrow$ contradiction!

2. p_j receives Req_i before entering its critical section but after sending out Req_j $\rightarrow$ 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) $\rightarrow$ i evaluates $Req_i < Req_j$ and defers responding

   (b) HELD: Req_j arrives after p_j's OK and all other pending OKs, and p_i automatically defers

   In either state, p_i defers the OK allowing p_j into the critical section until after p_i exits, 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 executes `enter()`, we have $p_i \in Memlist_j$, contradicting the case assumption.

In all sub-cases, safety is maintained.

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 receives Req_j after getting the OKs it needed, so is in HELD state, and thus defers the OK until after it exits its critical section. Contradiction!

2. p_i receives Req_j before receiving its own needed OKs, but after initiating the process. p_i would take the following steps:

   (a) Add p_j to $Memlist_i$

   (b) send Req_i to p_j

   (c) evaluate $Req_i < Req_j$ so defer responding to p_j

   (d) wait for OK from p_j (and the rest of the pending OKs)

   p_j receives Req_i while in WAIT state, evaluates $Req_i < Req_j$, and immediately sends OK. p_i receives the OK, executes its critical section, and only sends the final OK allowing p_j into the critical section after p_i has exited, contradicting the assumption!

3. p_i receives Req_j before even initiating its own process. This would cause $Req_i > Req_j$ because $seq\_num_i \geq seq\_num_j + 1$, contradicting the assumption that $Req_i < Req_j$!

Case 4 - $p_i \notin Memlist_j$ AND $p_j \notin Memlist_i$
Refer to Figure 4.1 for a visual example. Let $p_k \in Memlist_i \bigcap Memlist_j \neq \emptyset$, by the Mutual Friend Property. For p_i and p_j to simultaneously enter the critical section, 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, there are two possible sub-cases for the comparison of the two requests:

1. $Req_i < Req_j$ - proof from Case 2 holds

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 $Req_i < Req_j$. This is the same setup to the proof for Case 3, which holds here as well

Thus, we show that mutual exclusion is safely achieved. □

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

Because each common neighbor acts independently to inform p_i and p_j of each other, only one common neighbor is needed for correctness. Thus, the failure of other common neighbors don't affect correctness, as long as the Mutual Friend Property holds.

Deadlock-freedom

Recall that one critical property of mutual exclusion algorithms is deadlock-freedom. We say a deadlock occurs when 1) no node is in its critical section and 2) no node will be able to enter the critical section.

Theorem 6. *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. There are only two reasons for a node p_i to defer an incoming REQUEST Req_j:

1. p_i is in HELD state - but this equates to executing the critical section, so, by definition, the system can not be deadlocked.

2. p_i is WAITing and evaluates $Req_i < Req_j$. Because deadlock requires every node in the cycle to have its REQUEST deferred by at least one node, $\forall Req_j$ that is pending, $\exists Req_i$ such that $Req_i < Req_j$. In other words, every request in the cycle must have at least one other request with lower priority than it. This means there can not be any minimum request (with higher priority than all other ongoing requests). Note that evaluating requests in their lexicographic order means every pair is comparable, and this comparison is transitive, providing a total order on requests. Therefore, there must be some minimum request in the set of ongoing requests that comprise the cycle. This is a contradiction.

□

Starvation-freedom

Recall the third critical property of mutual exclusion algorithms is starvation-freedom, the guarantee that any given request will, in finite time, be allowed access to the critical section if other nodes are entering and exiting.

Theorem 7. *No request will be starved under Camera.*

Proof. Assume there is a starved node p_i. p_i can only be prevented from entering the critical section because it has a pending OK from some node $p_j \rightarrow p_i$ previously sent a REQUEST to p_j. At all such nodes p_j that receive Req_i, all subsequent Req_j will evaluate $Req_j > Req_i$ because the sequence number (first dimension) of Req_j is greater than Req_i's sequence number by line 13 of Algorithm 2. Thus, as other nodes continue to enter and exist, p_i will eventually have the minimum sequence number (recall the total order on requests). Due to our result from Theorem 6 that deadlock is impossible, some node will have to enter the critical section. Req_i will have the highest priority, so no node will be able to defer Req_i (while p_i will defer all incoming requests). Thus, p_i will enter the critical section, contradicting the assumption of starvation.

□

Causality

Camera maintains safety even incomplete membership at the expense of causal ordering of critical section execution.

Definition 5 (Causality)**.** Request B is causally dependent on Request A if there is a path of direct causal links from A to B. There are two types of direct causal links: 1) from an event to a subsequent event at the same node and 2) from a send event of a message to the receive event of the same message (on a different node).

Using Lamport timestamps (which track causality) as the sequence numbers in Ricart-Agrawala under complete membership ensures that if p_j's request is causally dependent on p_i's request, p_i will be allowed into the critical section before p_j.

The following observations mitigate the loss of causality under Camera:

- If application messages are not considered (such as by using independent sequence numbers rather than Lamport timestamps), then Camera maintains causality, providing application developers with a useful approximation of 'fairness'.

- More importantly, this is no worse than the original Ricart-Agrawala approach. In all situations where Ricart-Agrawala worked correctly, and thus preserved causality, Camera continues to preserve causality. Causality can only be violated when membership is not full and consistent, where the original Ricart-Agrawala would have violated safety.

Figure 4.3 depicts how application messages can outpace protocol messages to violate causality under Camera.

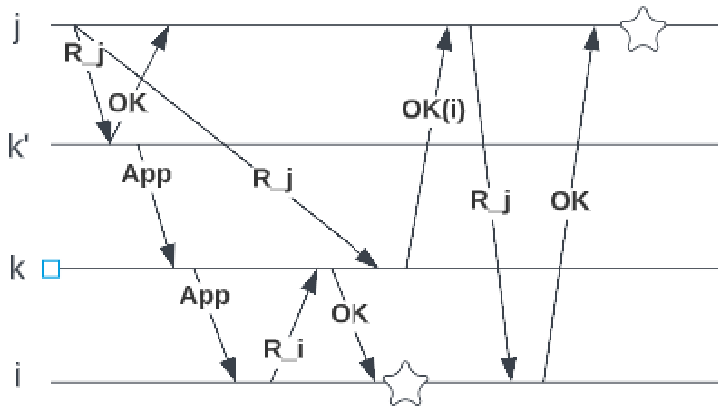


Figure 4.3: **Causality Violation under Camera**

4.3 Performance Analysis

Bandwidth

The number of messages required for p_i to enter the critical section is lower-bounded by 2*($len(Memlist_i)$-1) and upper-bounded by 2*(N-1). While the number of messages is the same as the original Ricart-Agrawala and similar to other distributed mutual exclusion strategies, such as a ring approach, half of the messages are OK messages which might include cached OKs and thus be of size O(N). The rapid RELEASE messages help keep this low in practice, but there is no guarantee. Exit bandwidth is much better - using IP multicast for the RELEASE messages brings exit bandwidth down to effectively O(1).

Client Delay

If nodes stop executing or requesting the critical section long enough for the system to stabilize (clear OK caches), and then node p_i seeks to enter it, Ricart-Agrawala and Camera both require 2*($len(Memlist_i)$-1) messages (outgoing REQUESTs and incoming OKs) for p_i to enter. This takes 1 round-trip time (RTT). Exiting never has any message delay.

Synchronization Delay

When one node p_j finishes executing the critical section with exactly one other node p_i waiting (and having waited long enough to find out about p_j and send it a REQUEST), it will take a single message (1 deferred OK from p_j to p_i) for p_i to enter its critical section. This takes 0.5 RTTs, and the analysis is the same for both Ricart-Agrawala and Camera. Now, loosen the assumption that p_i has been waiting long enough to find out about p_j, and say that 1) $p_j \notin Memlist_i$; and 2) p_i initiates its critical section entry instantaneously before p_j exits. The Mutual Friend Property requires there to be some p_k that is known by both p_i and p_j, which has cached that it sent p_j an OK. In this scenario, we need 1 RTT to reach everyone in p_i's membership list, including p_k, and then 1 more to check with p_j itself. p_j will have released by the time it receives Req_i, so there will be no additional delay there, for a total of 2 RTTs. Note this assumes that p_i already knows about everyone in p_j's OK cache.

Worst Case Synchronization Delay: Loosen that final assumption about p_i knowing everyone in p_j's cache to get worst case behavior of N-1 RTTs, with a line graph of cached OKs. Take p_{i_1} with $Memlist_{i_1} = \{p_{i_2}\}$. Imagine p_{i_2} has only p_{i_3} in its OK cache, p_{i_3} has only p_{i_4} in its OK cache, ..., and $p_{i_{N-1}}$ has only p_{i_N} in its OK cache. p_{i_1} sends its REQUEST to p_{i_2}, which responds with OK(p_{i_3}) - 1 RTT. p_{i_1} sends its REQUEST to p_{i_3}, which responds with OK(i_4) - 2 RTTs. And so on until p_{i_1} sends its REQUEST to

p_{i_N}, which responds with OK - N-1 RTTs (despite the total message count remaining unchanged).

To make matters worse, observe that this is all wasted effort.

Remark. *Assuming the Mutual Friend Property holds, safety doesn't require p_{i_1} to check with p_{i_4} through p_{i_N}, because they all have exited!*

Proof. Assume one of them p_k hadn't exited. The assumed Mutual Friend Property would require $p_{i_2} \in Memlist_k$ to ensure $Memlist_{i_1} \cap Memlist_k \neq \emptyset$. Also, because p_k is cached by some node ($p_{i_{k-1}}$), we know that p_k had to have entered the critical section at some point. So if $p_{i_2} \in Memlist_k$, p_{i_2} should have also received Req_k, responded with OK, and cached p_k at some point. In our setup, we said that p_{i_2} has only p_{i_3} in its cache. The only possible reason for why p_k is no longer in p_{i_2}'s cache is because p_k exited and sent a RELEASE, thus contradicting our assumption that p_k has not exited. Therefore, at least N-3 RTTs out of the N-1 constitute truly unnecessary delay. □

However, note that this line graph was artificially constructed by assuming absolute worst case delivery of several RELEASE messages, and cache relationship graphs of height $N-1$ are highly unlikely in practice due to the practical mechanisms suggested in Subsection 23 for RELEASE messages.

Memory Usage

Another important resource to consider is local node memory. The only significant change in memory usage under Camera is the retention of the list of recently OKed requests. Because nodes must serialize their requests, receiving a second request B for any given node means that its previous request A has already completed, so A can be discarded from the retained list if it is still present. Essentially, we need to retain at most one OKed request for each node, so the additional memory is bounded by O(N), and in practice often less due to the rapid IP-multicast RELEASEs. In modern memory systems, even sensor networks, this is acceptable overhead.

Effect of Drop Rate

Let's examine the effect of drop rate on the expected end-to-end delivery time of a message. Define the following variables:

d	end-to-end drop rate
r	hop-to-hop drop rate
h	1-hop delay
c	average # hops/message (6)
λ	$\mathbb{E}$(delay from E2E drop)
T	timeout

Clearly, $d = 1 - (1 - r)^c$, so the end-to-end drop rate grows rapidly with the hop-to-hop drop rate as seen in Figure 4.4. λ depends on which hop the message drops, and will incur delay from the timeout required to realize the message dropped plus the hop-to-hop latency for all successful hops.

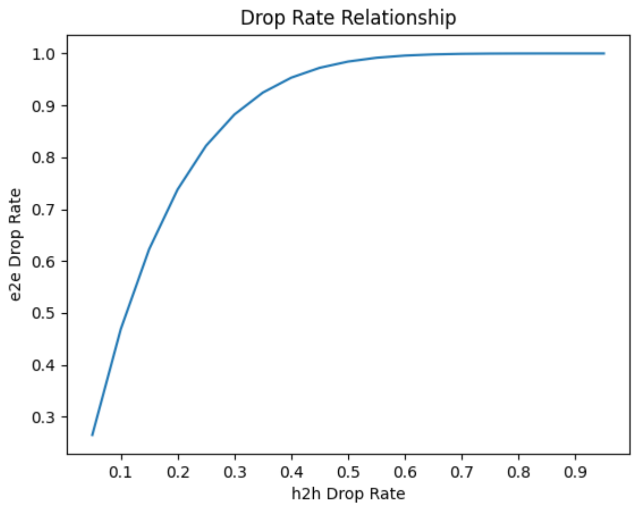


Figure 4.4: **Drop Rate Relationship**

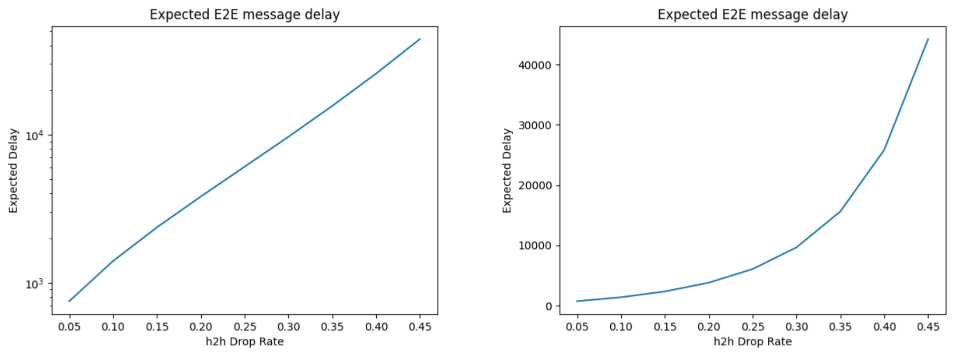
$$\begin{aligned}
\lambda &= \mathbb{E}(\text{delay from E2E drop}) \\
&= T + \sum_{i=1}^{c} \mathbb{P}(\text{drop on ith hop}) + h(i-1) \\
&= T + \sum_{i=1}^{c} (1-r)^{i-1} r + h(i-1) \\
&= T + \frac{r}{1-r} \sum_{i=1}^{c} (1-r)^{i} + h \sum_{i=1}^{c} i - 1 \\
&= T + \frac{r}{1-r} \left(\frac{1-(1-r)^{c+1}}{r} - 1 \right) + h \frac{c(c-1)}{2}
\end{aligned}$$

We note that if a message experiences exactly n E2E drops, the time for delivery will be n times the expected delay from a single drop, plus the ch time required to successfully transmit c hops. Also note that $0 < d < 1$, allowing closed-form solutions to the series below.

$$\begin{aligned}
&\mathbb{E}(\text{E2E delivery time}) \\
&= \sum_{n=0}^{\infty} \mathbb{P}(\text{exactly n drops})(n\lambda + ch) \\
&= \sum_{n=0}^{\infty} d^n (1-d)(n\lambda + ch) \\
&= (1-d)(ch \sum_{n=0}^{\infty} d^n + \lambda \sum_{n=0}^{\infty} n d^n) \\
&= (1-d)\left(\frac{ch}{1-d} + \frac{\lambda d}{(1-d)^2} \right) \\
&= ch + \frac{\lambda d}{1-d} \\
&= ch + \lambda \frac{1-(1-r)^c}{(1-r)^c}
\end{aligned}$$

.

Plugging in λ as calculated above gives the function depicted in Figure 4.5.



(a) Log scale (b) Linear Scale

Figure 4.5: Expected E2E delay against H2H drop rate

4.4 Relaxing The Mutual Friend Property

We refer to the algorithm described so far as the "fast path", which maintains safety under the Mutual Friend Property. We now present an extension of Camera that maintains safety even when the Mutual Friend Property is violated, referring to it as the "slow path".

Identification

The Mutual Friend Property is a safe assumption, because all violations will be detected. We assume an estimator for N (total network size), such as that described in [14], that provides an upper bound, $N.upper$. At the top of `enter()`, p_i switches to the slow-path if $len(Memlist_i) \leq \frac{N.upper}{2}$. While false positives are possible, they will be very rare because membership lists are *mostly* consistent in practice, as shown by the graphs in Section 2.3.

It is possible that p_i's membership list is larger than half the network only due to stale entries (failed nodes whose failure information has not reached p_i yet). p_i will wait indefinitely for the OKs from these nodes until the eventually complete failure detector informs it of their failure, and will resort to the slow path if the membership list ever becomes too small. The fast path is thus fault-tolerant.

Description

When requester p_i's membership list doesn't satisfy a quorum, we build a BFS spanning tree to propogate its request. 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. A node initiates the slow path for a request independently when it locally detects its membership list contains less than half the network, and is robust to any combination of slow or fast paths being run by other nodes. We adopt Algorithm 9 in Chapter 2 of [15] to create a BFS spanning tree as described in Algorithm 3, with the requester as root.

Algorithm 3: Constructing a Spanning Tree

```
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
```

We make the following additions, which clearly don't change the operation of 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 from 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 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.

Slow-Path Safety

Lemma 8. *A node p_i will only send its parent an OK when all nodes in the subtree rooted at p_i (including itself) have said OK by fast-path conditions.*

Proof. Induct on the height h of that subtree. Take the base case of $h = 0$. This leaf has no children, so will respond with OK once when its fast-path conditions allow it to. Now assume the above statement is true for all nodes that root subtrees of height $h - 1$ or less. By our specification of the slow-path, p_i (the root of the subtree of height h) will only respond OK when 1) it has received an OK from each of its children **and** 2) it has met its own fast-path conditions. Each child is the root of a subtree of height at most $h - 1$, so the inductive hypothesis holds and the child's response means all nodes in its subtree have said OK by fast-path conditions. But the subtree rooted at p_i is merely the union of the subtrees rooted at its children and p_i, so p_i's OK to its parent means all nodes in the subtree rooted at p_i have said OK by fast-path conditions. □

Theorem 9. *The slow path maintains safety.*

Proof. Now, take p_i to be the requester. It was the root of the BFS tree, so the subtree rooted at it is the entire network - this means the requester will only be able to enter its critical section once every node in the system has said OK under the fast-path conditions.

Now, assume two nodes p_i and p_j violate safety, both running the slow path. Each is the root of its own BFS tree (recall the slow-path runs independently at the request level), and by Lemma 8, each must have received an OK from every other node. In particular, they must have said OK in response to REQUEST messages from each other. There are the following cases for the ordering of the requests.

1. One received the other's request before sending its own. Assume without loss of generality that Req_i arrived at p_j before Req_j was sent out. $Req_j.seq\_num \geq Req_i.seq\_num + 1$. So p_i would receive Req_j in either the HELD or WAIT state, and would have deferred the OK to p_j until after p_i exited. Safety would have been maintained.

2. Each sent its own request before receiving the other's request, but also before receiving the other's OK. So each was in the WAIT state when it

received the other's request. We asserted that p_i must have received an OK from p_j. So p_j must have evaluated $Req_i < Req_j$. However, with that evaluation, p_i would have deferred p_j's request until after exiting, again contradicting our assumption of safety.

3. p_i received all OKs (including p_j's) before receiving Req_j. p_i was thus in the HELD state and deferred the OK to p_j until p_i exited, contradicting the assumption of safety being violated.

Let P_v denote the nodes who receive Req_v and must say OK to allow that request in. Simply, as soon as one of the conflicting nodes p_i runs the slow path (setting $P_i = N$), and the other p_j has a non-empty membership list, we are guaranteed $P_i \cap P_j \neq \emptyset$, and the proof described for the fast path in Section 4.2 applies.

$\square$

Remark (Liveness)**.** *The liveness properties (deadlock-freedom and starvation-freedom) of the slow path follows in the same way as in the fast path. There must be some minimum request that evaluates to less than all other requests, and thus receives all necessary OKs.*

Slow Path Fault Tolerance

Recall that the fast-path is unaffected by failures while the Mutual Friend Property is satisfied, and defaults to the slow-path otherwise. Once in the slow path, we assume a fixed f failures between a request being initiated and the critical section being executed. If a node p_i fails as in Figure 4.6, then p_i's parent p_j in the BFS spanning tree also becomes the parent for all of p_i's children (waits for OKs from them before responding OK to its own parent). If p_i and p_j fail simultaneously, p_j's parent k will take over as the parent for all of p_i's children as well as all of p_j's children. Because we piggy-backed subtree topology for the next $f+1$ levels on the echo algorithm will constructing the tree, this method can tolerate up to f simultaneous failures. Noting that the echo algorithm could easily carry back all topology information, providing a simple extension to tolerate an arbitrary number of simultaneous failures, as long as the membership graph remains strongly connected.

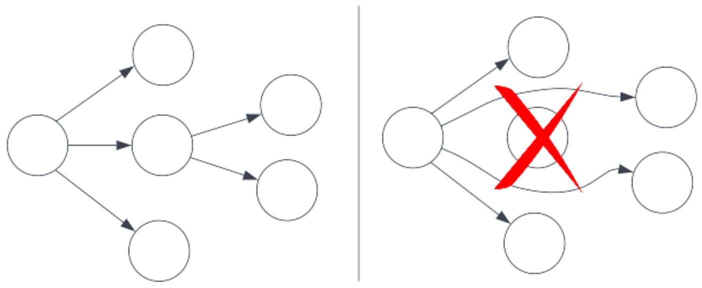


Figure 4.6: **Handling Failures on the Slow-Path**

CHAPTER 5

EXPERIMENTAL RESULTS

We evaluated the performance of Camera with varying numbers of concurrent requesters, system sizes, per-hop message drop rates, and network topologies. The key metrics were request waiting times and bandwidth (end-to-end message count, end-to-end message size, hop-to-hop message count, and hop-to-hop message size), and all parameter tuples were run for $n = 100$ runs. The simulation periodically imported membership traces from Medley [12], a membership protocol for dynamic ad hoc networks. While the duration of each CS execution was held constant, we also varied the spatial and temporal distribution of requests in our realistic workloads. All graphs show the mean as a red dot, the IQR as the orange bar, and the min and max whiskers in green and blue, respectively.

Our simulation sought to address the following questions:

1. How much synchronization delay does Camera add? How much bandwidth does Camera require?

2. How does Camera scale (delay, bandwidth) against the number of concurrent requesters, system size, and drop rate?

3. How does Camera perform against realistic workloads with various temporal (request arrival times) and spatial (which nodes are requesting) distributions?

We tested under high-contention workloads, where all requesters initiated their requester simultaneously. In addition, we simulated realistic workloads under four configurations of spatial and temporal request distributions.

Throughout, we refer to the *waiting time* for an individual request as the time elapsed between the requester executing `enter()` and the requester executing its critical section, and analyze this as a proxy for the synchronization delay. We expect that client delay should be similar to the original Ricart-Agrawala algorithm, because a solo requester should not have to take any corrective action. Under high contention, we expect average synchronization delay to behave similarly to Ricart-Agrawala, because requesters will take corrective action *while* other nodes are executing the critical section.

Table 5 shows default values for parameters, except where stated otherwise. The number of concurrent requesters applies only to the high-contention workloads, while IA:D applies only to the realistic workloads.

trials per data point	100
drop rate	0.05
system size (N)	256
concurrent requesters	10
IA:D	1.0
topology	random

5.1 High Contention Workload

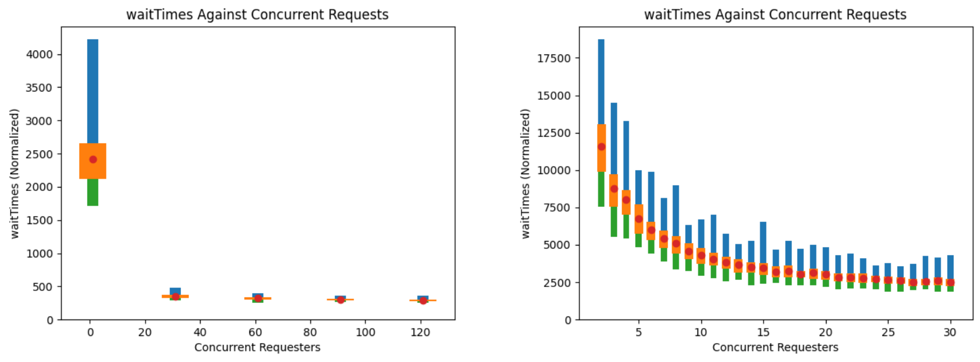
The first set of workloads were high contention, where all requesters initiated their requests concurrently. We isolate the synchronization delay by normalizing wait times by the number of concurrent requesters.

Take an example of n requesters who run `enter()` simultaneously, each spending 1 time unit in the CS with synchronization delay of H time units. The waiting times are $1 + H$ for the second requester, $2 + 2H$ for the third, and $(H + 1)i$ for the i^{th}. The average waiting time then becomes $\frac{\sum_{i=1}^{n-1}(H+1)i}{n-1} = \frac{(H+1)(n-1)n}{n-1} = (H+1)n$. Dividing again by n for the normalization step approximates isolating the synchronization delay H, because the CS execution time of 1 is constant.

Concurrent Requesters

We begin by examining how Camera scales with the number of concurrent requesters.

Figure 5.1a shows normalized waiting times flatten as a function of the number of concurrent requesters, because requesters who enter later are typically waiting on very few OKs, shortening the synchronization delay and thus bringing down the average. This point is reached quickly, as shown by isolating the leftmost region of the graph in Figure 5.1b.



(a) Normalized wait times (b) Few requesters

Figure 5.1: Normalized wait times against concurrent requesters

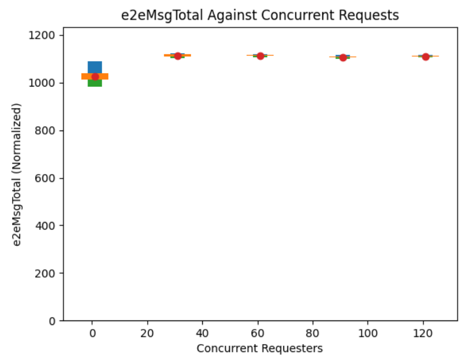
Figure 5.2 shows that the incremental bandwidth per requester is constant, meaning that total bandwidth scales linearly with the number of concurrent requesters. This is expected and ideal behavior because each request must be approved by almost everyone else in the system.

System Size

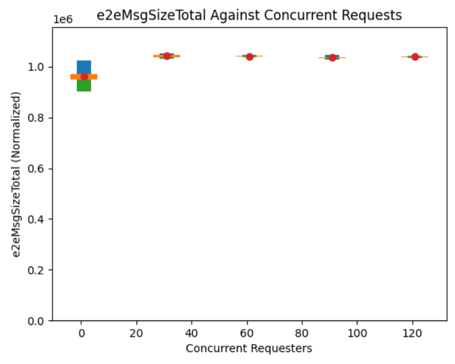
We examined how Camera performs in systems of various sizes.

Figure 5.3 (normalized by N) show that average waiting time increases with system size, rapidly at first and leveling off by N=256. This is because an individual node's wait time at low concurrent requesters (here 10) is determined by the *maximum* round-trip time to other nodes, which increases with the number of nodes.

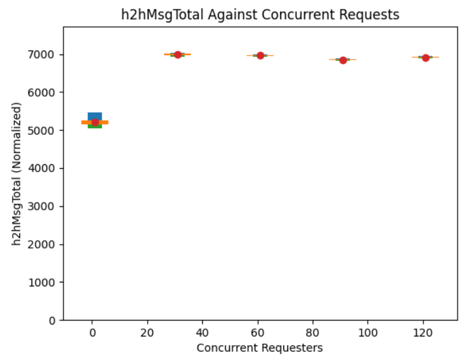
Figure 5.4 shows the total number of messages *per node* increases sublinearly with the number of nodes in the system. The number of messages per *non-requesting* node is proportional to the fixed number of concurrent requesters, while the number of messages per *requesting* node increases linearly (requests must be approved by every node in the system). Messages per node is a weighted average of message per requesting and non-requesting node, so grows sublinearly and slower at larger N where requesters constitute less of the network. Camera thus scales well to large systems.



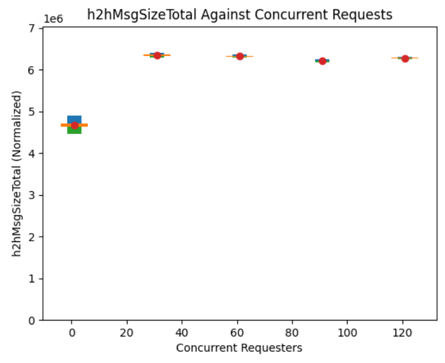
(a) End-to-End Message Count



(b) E2E Message Size



(c) H2H Message Count



(d) H2H Message Size

Figure 5.2: Normalized Network Metrics against Concurrent Requesters

Drop Rate

Message drops play a critical role in dynamic edge networks. We verify Camera performs optimally under a range of realistic hop-to-hop drop rates, fixing the number of concurrent requests at 64 (a quarter of the system) rather than 10 to augment the effect of message drops.

Figure 5.5 and Figure 5.6 show that waiting times and bandwidth both grow exponentially with drop rate, because synchronization delay at high concurrent requesters is dominated by the message delivery time from the exiting node to the incoming node. The growth rate is slightly faster than suggested by our analysis of expected message delay in Subsection 4.3, due to the long initial waiting times. Yet, note that the increase is at most an order of magnitude in the practical range between 0.05 and 0.2, reaffirming the practicality of Camera.

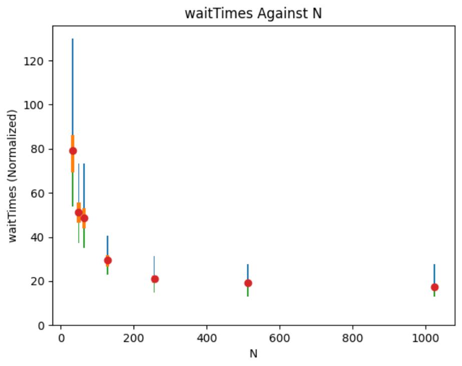
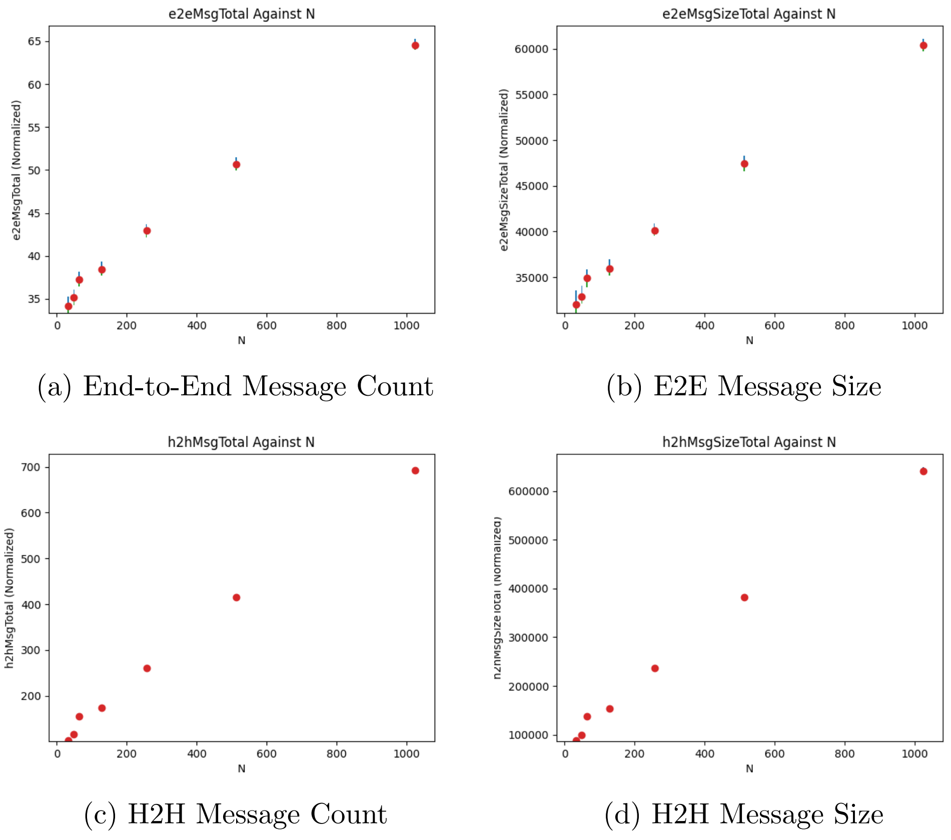


Figure 5.3: **Normalized wait times against system size**



(a) End-to-End Message Count

(b) E2E Message Size

(c) H2H Message Count

(d) H2H Message Size

Figure 5.4: Normalized network metrics against system size. *Normalization is by system size and concurrent requesters = 10.*

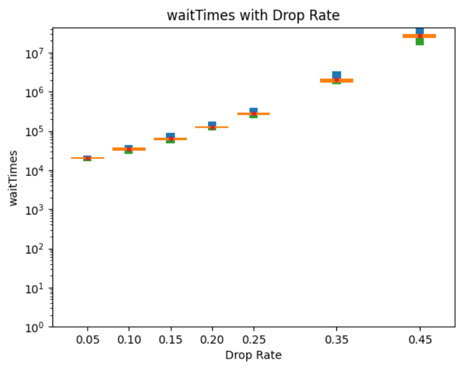
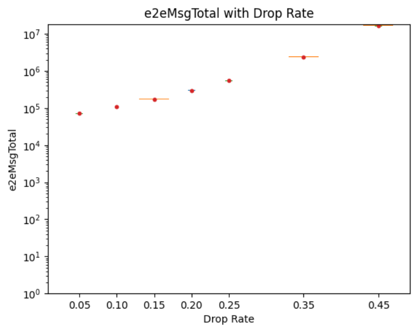
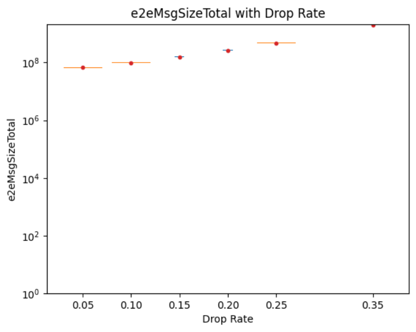


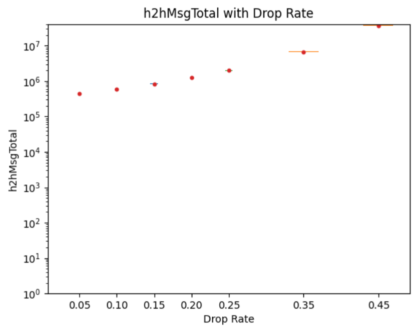
Figure 5.5: **Wait times against drop rate**



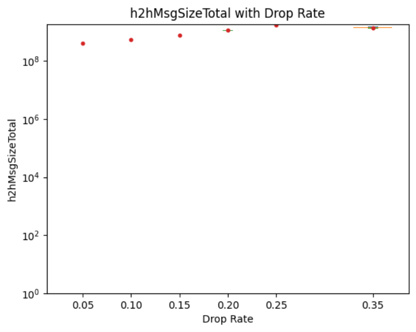
(a) End-to-End Message Count



(b) E2E Message Size



(c) H2H Message Count



(d) H2H Message Size

Figure 5.6: Network Metrics against Drop Rate

5.2 Realistic Workload

In order to understand how Camera behaves in the wild, we injected workloads where the requester interarrival times were distributed as either exponential and Weibull [16] distributions, the latter of which is known to represent network arrival rates [17]. The choice of requester from the N nodes followed either a uniform or Zipfian distribution.

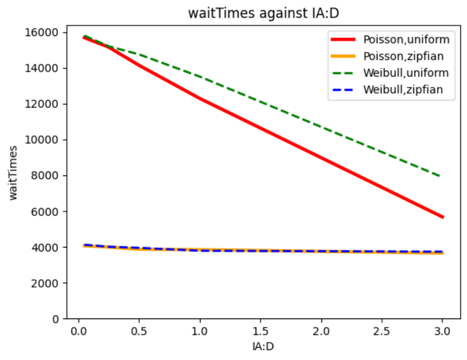
Interarrival Time v.s. CS Duration

These distributions introduced an additional key parameter of inquiry: the ratio between the average interarrival time and the duration of critical section execution (IA:D), which was held constant. In theory, a higher ratio would imply that, on average, requests are coming in slower than they're being executed, and wait times should be driven down.

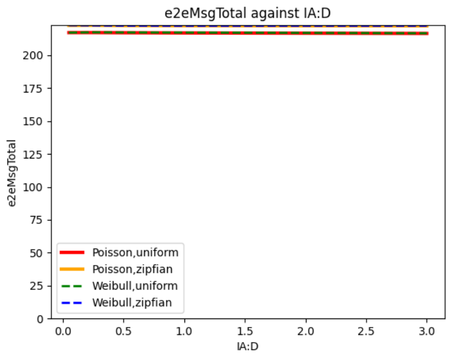
Figure 5.7a shows that this behavior is highly dependent on the spatial distribution. When requesters are uniformly distributed, waitTimes decrease linearly with IA:D as expected, and faster under the Poisson workload. However, the zipfian spatial distribution has consistently lower wait times, independent of both the average and distribution of interarrival rates. The concentration of requesters results in 1) fewer instances of finding out about potentially conflicting requesters and needing corrective action and 2) less effective contention because nodes locally serialize their own requests. That advantage becomes less important when requests are arriving slowly (high IA:D), so the performance of the two distributions converge. Note that the zipfian distribution tends to be seen in the wild [18] , making Camera highly practical.

Figure 5.7b shows total end-to-end messages (and the remaining bandwidth metrics, omitted) are independent of IA:D, because each requests still needs approval from the rest of the system.

In summary, Camera provides low waiting times even at high request arrival rates under a zipfian spatial distribution (likely in practice), and bandwidth is roughly distribution-independent.



(a) Wait times

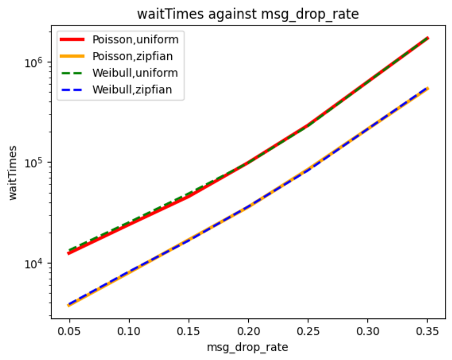


(b) E2E Messages

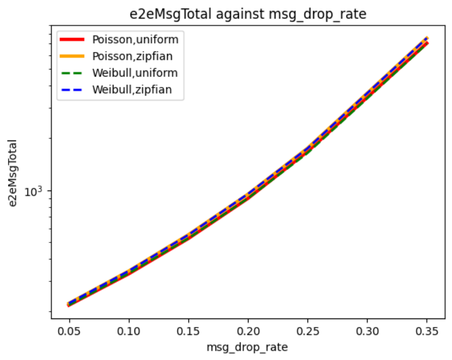
Figure 5.7: Effect of IA:D

Drop Rates

We examined the effect of drop rate under these realistic workload distributions. Figure 5.8a and Figure 5.8b show that all four distributions saw both wait times and bandwidth metrics respond at the same exponential rate to changes in the message drop rate. This is because changes in wait time under increasing drop rates are primarily due to additional message resends, and our analysis in Subsection 4.3 applies equally to all distributions. For the same reasons as discussed with the IA:D analysis above, the zipfian wait times are consistently lower than the Poisson lines.



(a) Wait times



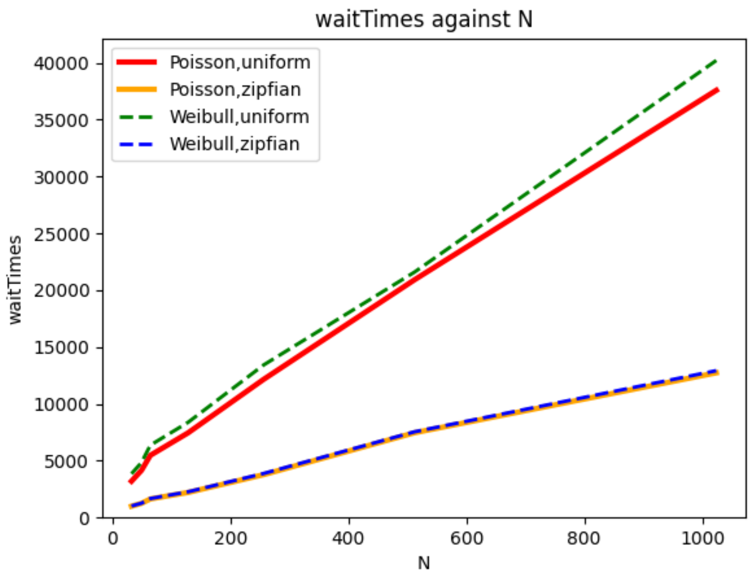
(b) E2E Messages per node

Figure 5.8: Effect of Drop Rate

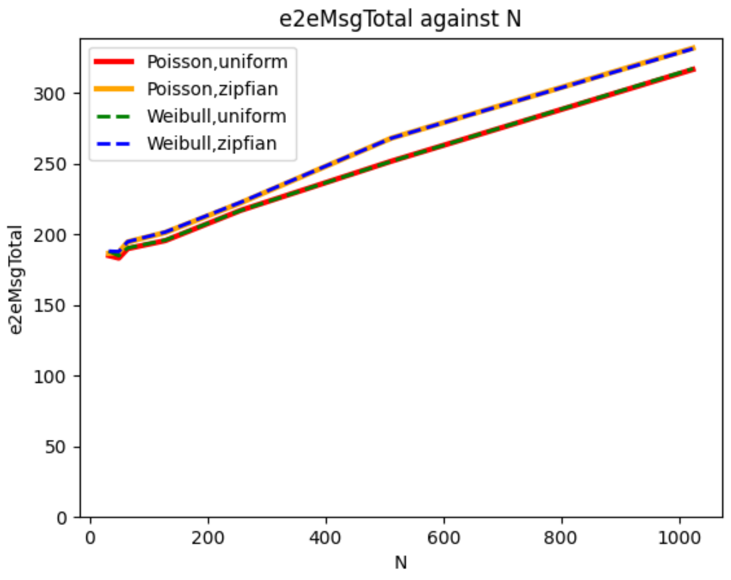
System Size

We evaluated the effect of increasing system size on Camera's performance. Figure 5.9a reveals that wait times grow with system size at a faster rate under uniform distributions than under zipfian distributions, and the latter is concave down (sublinear). Recall that the higher concentration of requesters under a zipfian distribution provide an advantage from local node serialization. This advantage becomes more impactful with more nodes in the system, as fewer nodes who are holding back OK messages results in a smaller maximum values for the delivery time of those lingering OKs. Note that in the uniform case, the Weibull temporal distribution has slightly higher wait times than exponential interarrival times; having more nodes results in larger maximum OK delays, and the effect compounds under Weibull's burstiness.

Figure 5.9b shows that the *per node* bandwidth increases sub-linearly with the size of the system, as the weighted average of linearly-increasing *per requesting node* bandwidth and roughly-constant *per non-requesting node* bandwidth. The effect of a constant hop-to-hop drop rate is more evident in large systems where the average end-to-end journey length is larger, partially contributing to the increase. The same reasoning explains why zipfian requester distribution requires more bandwidth than uniform, especially in larger systems; because requesters are more concentrated topologically, the average number of hops between requesters and non-requesters is larger, producing more end-to-end drops. This marginal effect is visible in all bandwidth graphs in this section, and is due to the implementation quirks that nodes are clustered in the topology by their id, which is used to sample from the zipfian distribution.



(a) Wait times



(b) E2E Messages per node

Figure 5.9: Effect of System Size

Against varying IA:D ratios, drop rates, and system sizes, Camera operates close to theoretical values. Wait times are consistently lower under realistic zipfian requester distributions than under uniform distributions, although at the cost of marginal bandwidth increases. Interarrival time distributions have little effect, although Camera performs slightly better when arrivals are a Poisson process than it does under Weibull interarrival times. Camera is thus highly scalable, robust to various distributions, and practical.

CHAPTER 6

RELATED WORKS

Classical solutions to the mutual exclusion problem can be divided into permission-based and token-based approaches. In addition to Ricart-Agrawala [9], the former category includes Maekawa's $\sqrt{N}$ Algorithm [11], a voting-based algorithm that arranges the N nodes into a square matrix to intelligently select a subset of nodes to grant permission. The token-based approaches include Raymond's tree-based algorithm [10], which imposes a logical tree structure upon the network, with the token holder as the root. As discussed in Chapter 1, these classical algorithms all assume fully consistent membership.

General consensus and coordination systems for distributed systems can also be used to achieve mutual exclusion. For example, Paxos [19] and Raft [20] can be used to reach consensus about which node should be currently executing the critical section. Similarly, Google's Chubby Lock System [21] can provide locking for filesystems, a special case of the mutual exclusion problem, but is engineered for high-bandwidth networks. The open-source coordination system ZooKeeper suffers from the same limitation, despite providing coordination and synchronization primitives. The common thread is that existing systems are 1) more general and thus sacrifice performance; and 2) designed for reliable, static networks.

On the other hand, adhoc and edge networks are often characterized by incomplete membership. There has been much recent work in building other critical primitives for such settings, such as distributed hash tables [22], bounded degree topologies [23], grid-based virtual filesystems [24], and consensus [25].

The mutual exclusion problem in this setting has also received attention, primarily focused on token-based approaches. One such example is the direct implementation of mutual exclusion inside a P2P DHT [26], which only provides probabilistic, rather than the provable, safety and is only useful for DHT applications. Another algorithm [27] for mobile adhoc networks is highly efficient under high mobility, but underperforms when mobility is limited. The majority of algorithms focused on MANETs [28], [29], [30] are token-based with the accompanying limitations. Even permission-based approaches that modify Ricart-Agrawala for MANETs make assumptions (such as stable mobile support stations [31] or a fixed set of possible peers [32]) that don't apply to many modern ad hoc networks, such our motivating examples of sensor networks.

CHAPTER 7
CONCLUSION

We presented Camera, a permission-based churn-tolerant mutual exclusion protocol for dynamic edge networks. We leveraged the key insight that edge membership protocols typically satisfy the Mutual Friend Property, and accompanied our primary algorithm with a fall-back for when MFP is not satisfied. We formally proved that Camera guarantees safety and freedom from both deadlock and starvation. Experimental results demonstrated that per-requester waiting time drops exponentially with additional concurrent requesters, grows only linearly with large system sizes, and favors practical spatial and temporal request distributions.

Future Work Future avenues of exploration include but are not limited to the following:

1. Implementation and deployment in order to empirically validate performance in real-world networks and under real workloads. While our simulation results are promising for scalability, implementing Camera in a small-scale sensor network will further corroborate correctness. Deploying in a live large-scale system will provide data about the real-world efficacy of Camera.

2. Optimization of the slow-path algorithm. Our simulation results regarding the Mutual Friend Property as presented in Section 4 suggest that the slow-path will be invoked extremely rarely. We therefore favored simplicity and choose a fairly inefficient spanning tree broadcast method. However, for systems using membership protocols that are less likely to satisfy the Mutual Friend Property, the slow-path can be optimized, such as by using a more efficient broadcast method, allowing OKs multiple routes to reach the requester, or building the spanning tree with topology awareness.

3. Empowering Camera with topology-awareness. There's a vast design space in assigning priority, obtaining permission, or even using algorithmic variants based on node proximity. This would be especially useful in deployments with high hop-to-hop message drop rates, because reducing the average number of hops a message must travel drastically decreases the end-to-end drop rate, and can therefore reduce both bandwidth and waiting times.

4. Generalization to other primitives focused on high-churn edge networks. Sensor and similar networks require many of the same core-layer algorithms as traditional data-center systems, and can benefit from a single integrated solution, the adhoc equivalent of ZooKeeper [33]. Such algorithms can include leader election, consensus, hash tables, RPC frameworks, distributed shared memory, and more.

APPENDIX A

APPENDIX

A permission-based mutual exclusion algorithm might specify that a requesting node must 1) receive some number of OK messages before entering the CS; 2) simply enter the CS if it doesn't receive STAY-OUT message(s) within some fixed timeout t; or 3) some combination of the former two.

Lemma 10. *Let ALG be a permission-based mutual exclusion algorithm that satisfies safety in an asynchronous system. Then ALG must exclusively require a requesting process to receive OK message(s) before entering the CS, rather than entering if it doesn't receive STAY-OUT messages within some fixed timeout t.*

Proof. Suppose that a requesting process p_i entered the CS t time-units after sending its requests, unless receiving a STAY-OUT message. While another node p_j was executing its critical section, maintaining safety would require some node p_k to send p_i a STAY-OUT message. Our asynchronous system allows arbitrary delays, and delaying that message $t + 1$ time units would cause p_i to execute its critical section while p_j was still executing it, violating safety. □

Recall Theorem 1:

Theorem 1. *Let ALG be a mutual exclusion algorithm with the following assumptions:*

- *Any node can run* `enter()` *at any time*
- *Messages can only be sent along edges in the membership graph*
- *'Information' can only be conveyed across processes via messages*

Then, a strongly connected membership graph is necessary for ALG to satisfy safety and liveness.

Proof. We will proceed by first taking permission-oriented ALG and examine the membership edges required to maintain safety and liveness, and then do the same for non-permission-oriented ALG. Intuitively, maintaining both safety and liveness requires passing information between any competing nodes, which can only happen on edges of the membership graph.

Fix ALG to be permission-oriented, and let G be a membership graph on which ALG maintains safety and liveness. Take any pair of nodes p_i, p_j that can request simultaneously, supposing p_i has priority. By Lemma 10, p_j must be waiting for an OK message m from some node p_k. To maintain starvation-freedom, p_k should only withhold m from p_j because p_k 1) is informed that p_i is executing its critical section and 2) will eventually learn when p_i exits. Because information can only be transmitted over messages and thus along edges in G, there must exist some path p in G from p_i to p_k. Maintaining liveness also requires an edge $e = k \rightarrow p_j$ for p_k to send p_j that final OK message. Thus, there exists a path $p' = pe$ from p_i to p_j. Now suppose p_j was already executing its critical section when p_i initiated its request, and apply the symmetric argument, so $p_i \leftrightsquigarrow p_j$. Because we assumed any node can run `enter()` whenever it wishes, any pair of nodes in the membership graph could request simultaneously, so the above holds for all (p_i,p_j) in G. By definition, G must be strongly connected.

Now, assume the algorithm is not permission-oriented. This means there must be some 'token' circulating to permit critical section execution. Because we assumed every node p_j can make a request, starvation freedom means that the token must have a path from its current location to p_j. Because the same assumption means that the current location can be at any node p_i, there must be a path from every node p_i to every other node p_j, making G strongly connected by definition.

We thus see that any mutual exclusion algorithm maintaining both safety and liveness requires a strongly connected underlying membership graph.

□

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. Glaserand, 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.B. Velasquez, G. Chowdhary and S. Gupta. “Learned visual navigation for under-canopy agricultural robots,” *arXiv preprint arXiv:2107.02792*, 2021.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] U. Tadakamalla and D.A. Menascé.. "Characterization of iot workloads," in *Edge Computing–EDGE 2019: Third International Conference, Held as Part of the Services Conference Federation*, S. 2019, Ed., vol. 3. Springer International Publishing, 2019, June 2019, pp. 1–15.

[14] L. Massouli´e, E. Le Merrer, A.-M. Kermarrec and A.J. 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.

[15] R. Wattenhofer. *Principles of Distributed Computing*, 2014.

[16] Weibull Distribution. "Weibull distribution — Wikipedia, the free encyclopedia," 2021, [Online; accessed 20-April-2023]. [Online]. Available: https://en.wikipedia.org/wiki/Weibull_distribution

[17] 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.

[18] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking cloud serving systems with ycsb," in *1st ACM symposium on Cloud computing*, vol. 141, June 2010, pp. 143–154.

[19] L. Lamport. "Paxos made simple," in *ACM SIGACT News (Distributed Computing Column)*, vol. 3, December 2001, pp. 51–58.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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, PPAM 2009, Wroclaw, Poland, September 13-16, 2009, Revised Selected Papers, Part II 8*, 2010, pp. 194–203.

[25] 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.

[26] 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, La Jolla, CA, USA, February 26-27, 2004, Revised Selected Papers 3.* Springer Berlin Heidelberg, 2004, pp. 11–21.

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

[28] 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.

[29] N. Malpani, 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.

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

[31] 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.

[32] W. Wu, J. Cao, and J. Yang. “A fault tolerant mutual exclusion algorithm for mobile ad hoc networks,” in *Pervasive and Mobile Computing*, vol. 4, February 2008, pp. 139–160.

[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.