Building Reliable Distributed Systems: Bridging Specifications and Implementations

by

Ding Ding

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

August, 2026

Dr. Aurojit Panda

Dr. Jinyang Li

Dedication

To my family.

Acknowledgements

First and foremost, I would like to thank my advisors, Professors Aurojit Panda and Jinyang Li. They are perhaps the most unconditionally supportive advisors any student could hope for. Under their guidance, I experienced the genuine freedom to do the research I like. They always offer insightful advice whenever I needed it given their breadth of knowledge and depth of experience. They approached every project and discussion with unwavering care and commitment. Their office doors were always open, and I could turn to them for help at any time. They consistently supported my choices and offered generous encouragement. I never left a conversation with them feeling diminished or discouraged. I am profoundly grateful to have been their student.

I am deeply grateful to Zhanghan Wang, with whom I completed several projects and raced through more deadlines than I can count. I would also like to thank Jinkun Lin and Qiutong Men; I truly enjoyed every research discussion we had. Beyond research, Zhanghan, Jinkun, and Qiutong became some of the dearest friends I gained during this journey. The four of us formed our little hot-pot group, and I will always cherish the weekends we spent gathering and sharing hot pot together.

I am also grateful to my best friends, Xinyi Zhang and Sijian Wang. Thank you for always being there and for listening to everything happening in my life.

I would like to thank my mentor during my internship at VMware Research, Marcos K. Aguilera, and my hosts during my internship at Google, Ahmed Awad and Professor Brad Karp. You are all exceptionally kind and wise, and your guidance and support have been invaluable. I feel

deeply honored and grateful to have had the opportunity to work with you.

Finally, to my parents: thank you for always supporting every choice I have made, without condition, and for bringing me warmth and reassurance whenever I felt lost. And to my grandmother in heaven: I know you are watching over me. I wish you could have been here to see this moment, and I hope I have made you proud.

Abstract

Distributed systems such as etcd and ZooKeeper implement protocols that have been formally specified and verified. Yet developers use the protocol only as a reference and write the implementation by hand, so the specification that was verified is never directly connected to the code that runs.

This leaves two gaps. The first is *faithfulness*: the manual translation from specification to code often introduces mistakes, especially where developers add optimizations such as concurrency, causing the implementation to diverge from what the specification permits. The second is *coverage*: specifications can omit certain runtime details, such as which in-memory protocol state a crash destroys, and thus fail to specify how the implementation must handle those details to maintain correctness. Because of these two gaps, implementations of verified protocols continue to ship with safety bugs [26, 36, 105].

This dissertation bridges both gaps to involve the specifications into building the systems to enhance the correctness of the implementations. Ellsberg checks at runtime whether a deployed implementation's messages comply with its specification, requiring no access to internal state and no changes to the system being checked; it detects ten protocol implementation bugs across etcd, ZooKeeper, and RedisRaft. PostMark extends the specification to model volatile state lost in crashes, synthesizes a crash-consistency policy from the extended specification, and enforces that policy in existing implementations; in our experiments, it reduces etcd's average read latency by 63%.

Table of Contents

List of Figures

List of Tables

1 Introduction

Distributed protocols, implemented in lock services [11, 22, 104], distributed databases [90, 103], etc., lie at the heart of all fault-tolerant applications. Ensuring that these services, which we refer to as *distributed protocol implementations* (DPIs), correctly implement distributed protocols remains challenging. Nowadays, the development of a production DPI often proceeds in three stages. First, a distributed systems researcher designs a protocol and produces pseudocode and text describing the protocol and its correctness properties. Next, researchers use interactive theorem provers, verification tools, or handwritten proofs to check the protocol's correctness. Finally, developers implement and optimize the protocol. Note, each of these stages is performed by different people, and in different languages. For example, researchers may use TLA$^+$ [10, 98] or Coq [89] to specify and verify a protocol such as Raft [68]. Developers then implement the protocol in a general-purpose language such as Go or Java, as in etcd [22] and ZooKeeper [104].

While this workflow has been successful in producing distributed systems, structurally it makes it challenging to produce *correct* distributed systems. Developers use the algorithm only as a *reference*: they read the pseudocode and write code that they believe corresponds to it. The specification that was so carefully verified is never directly connected to the implementation that actually runs. Verification therefore establishes that the *specification* is correct but does not by itself establish the correctness of the resulting implementation. In reality, popular DPIs have shipped with application visible bugs: *e.g.*, prior versions of etcd [22] would return stale values in some scenarios due to a bug [26] in its implementation of a linearizable read protocol [99].

Similarly, prior versions of ZooKeeper [104] would, in some scenarios, lose updates during leader elections due to a protocol implementation bug [105]. Both systems are widely used, and both bugs can result in application bugs such as data corruption.

1.1 Two Gaps Between Specification and Implementation

This disconnection opens two distinct gaps between the verified specification and the deployed system. They have different causes and require different solutions as we argue below.

1.1.1 Gap 1: Faithfulness

A correct implementation should *refine* its specification: every action it takes should correspond to behavior that the protocol permits. However, manually translating an algorithm into an implementation can introduce mistakes, causing the implementation to exhibit behavior that its specification does not permit. When implementing the protocol, developers have to manually map protocol state and transitions onto concrete data structures while adding mechanisms to support concurrency, batching, and caching that a high performance production system requires. Thus, a single atomic transition in the specification may become several implementation steps, which can interleave in ways the specification never permits. The implementation can therefore diverge from its specification [26, 105].

Static refinement proofs [12, 37, 50, 92] can close this gap by proving the code against the specification. However, applying these frameworks generally requires developers to rewrite the system within a framework that constrains the implementation language, libraries, or program structure. They are therefore difficult to apply to existing, mature, heavily optimized systems. One can also narrow the gap through testing. However, although testing is essential, it cannot exercise every message ordering, failure, and concurrent execution that may arise in deployment. What is missing is a practical way to check the behavior of an *existing* implementation against

the protocol it claims to implement.

1.1.2 Gap 2: Coverage

Specifications often omit important implementation details, such as when state must be persisted to ensure correct recovery after a crash. To remain simple and tractable for verification, they often abstract away certain aspects of the runtime environment, including the distinction between volatile and durable state and the timing of persistence operations. However, a real implementation must decide which state to make durable and when. An incorrect persistence policy can cause the system to lose data [36].

Because the specification does not model durability, it provides no basis for implementation to make correct decisions w.r.t. durability. Instead, developers must reason about persistence and crash recovery outside the specification and encode their persistence mechanisms by hand. Developers may extend the specification with persistence logic and verify that the resulting model preserves its safety properties. However, this remedial approach has three limitations. First, a persistence operation in the implementation may fall within a sequence that the specification represents as a single atomic transition, and thus cannot be expressed without decomposing that transition. Second, each implementation change may require revising the specification and repeating the verification. Third, a proven-correct persistence policy is not necessarily efficient, and the choice of persistence policy can have a substantial impact on performance. The challenge is therefore not merely to check code against an existing specification, but to automatically *derive* a correct and efficient persistence policy that can be enforced in the implementation.

Together, these two gaps explain a persistent and troubling observation: even when a protocol has been specified and verified, the system built from it can still be wrong.

1.2 Bridging the Gaps

This dissertation therefore asks: how can we bridge these two specification-implementation gaps, so that the correctness we establish for a specification carries over to the real system built from it? Bridging them requires bringing the specification directly into the process of building the system, rather than leaving it behind once the protocol has been verified. The two gaps call for different solutions, and this dissertation presents one system for each.

1.2.1 Ellsberg: Checking Refinement at Runtime

We develop Ellsberg, an approach called *runtime protocol refinement checking* (RPRC), to address the faithfulness gap. Ellsberg notifies administrators when an implementation's behavior deviates from its protocol specification, allowing administrators to mitigate the effects and programmers to fix the underlying bug. To do so, Ellsberg observes the messages sent and received by each deployed DPI process and checks whether that process's behavior is consistent with a simulated execution of the protocol according to its specification. A sent message that no compatible protocol execution could have produced reveals that the implementation has diverged from its specification.

Applying this idea to production systems requires handling concurrency without depending on the implementation's internal structure. A DPI may process several messages concurrently, and its network trace does not reveal the order in which those messages affected local protocol state. Ellsberg therefore maintains the set of protocol states induced by every possible processing order, and uses an incremental breadth-first search to prune that state space and keep online checking practical. Refinement can be checked locally because the protocol specification defines each process's behavior in terms of its local state and the messages it sends and receives. Each Ellsberg instance can therefore check its colocated process independently, without coordinating with other instances.

We evaluate Ellsberg on etcd [22], ZooKeeper [104], and RedisRaft [34, 80]. It detects ten protocol implementation bugs, including a previously unreported RedisRaft reconfiguration bug. Ellsberg processes traces faster than the systems can generate them, requires no cross-node coordination, and increases the 99th-percentile response latency by at most 10.7% with no throughput loss. A protocol specification can therefore serve as a practical runtime oracle for detecting translation errors in existing, highly optimized implementations.

1.2.2 PostMark: Synthesizing Policies for Disk-State Persistence

PostMark addresses the coverage gap. It represents the ordering between persistence and message transmission as a per-message predicate `CanSend(m, d_s)` that determines whether an outbound message `m` may be exposed given the latest durable protocol state `d_s`. If the predicate holds, the implementation may send the message immediately; otherwise it delays the message until the state that justifies its transmission has become durable. Casting the decision as a per-message predicate makes explicit the correct ordering between state persistence and the external exposure of protocol progress, and thereby allows persistence to proceed concurrently with protocol execution without sacrificing correctness.

In order to derive a provable correct and efficient persistence policy, PostMark augments the protocol's existing TLA$^+$ specification with a drop-in durability-aware message library. PostMark derives candidate predicates by comparing the protocol state exposed by each message with the latest durable state. The result is a policy that preserves the checked safety properties without imposing unnecessary persistence constraints. To use that policy in a real implementation, we provide a runtime library that separates persistence from protocol execution in a dedicated thread, tracks durable progress, and gates outbound messages according to the synthesized predicate.

We integrate PostMark with etcd, RedisRaft, and ZooKeeper. In etcd, our experiments show the synthesized policy reduces average read latency by 63% by allowing messages unrelated to

pending writes to bypass persistence. PostMark thus turns the specification into a synthesis oracle capable of searching for optimal persistence behavior the original model omitted, helping developers build a crash-consistent implementation without sacrificing the concurrency that performance requires.

Ellsberg and PostMark address different gaps, but they answer the same question: given that we can verify a protocol's specification, how do we ensure that the specification's guarantees carry over to the real system? Ellsberg uses the specification as a runtime oracle for checking implementation behavior, ensuring that every message sent is the result of some compatible protocol execution according to the specification. PostMark extends the specification to cover what it originally omitted, synthesizes the missing policy, and enforces it in the implementation. In both cases, the verified specification ceases to be an isolated artifact that developers leave behind when implementation begins; instead, it continues to improve the correctness of the system built from it.

2 Ellsberg: Runtime Protocol Refinement Checking

This chapter presents Ellsberg, a runtime system for detecting safety bugs in deployed distributed protocol implementations. It closes the *faithfulness* gap of Section 1.1.1: the divergence that arises when developers translate a verified specification into production code by hand.

2.1 Runtime refinement: an overview

RPRC provides a runtime approach for connecting statically verified protocols to an implementation: RPRC systems observe a DPI's runtime behavior, and notify administrators when the implementation's behavior deviates from what is required by the protocol. Analyzing runtime behavior allows RPRC tools to avoid making assumptions about how the protocol is implemented or what libraries are used. Indeed, *Ellsberg*, the RPRC system described in this chapter, treats the DPI as a blackbox, assumes no access to the DPI's internal state, and does not impose any limits on how DPIs are implemented. Note, RPRC systems do not replace static protocol and implementation verification [10, 37, 50, 52, 59, 72, 92, 95, 96] (they cannot statically generate proofs to show that a protocol is correct, or that an implementation refines the protocol) and fuzz testing tools [6, 44–46, 51, 70] (they do not try to maximize coverage and find all bugs before an implementation is deployed), but rather complement them.

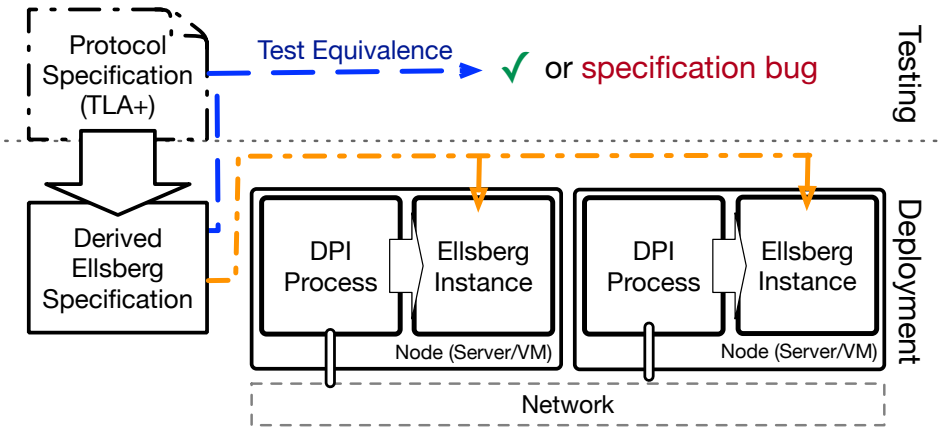


Figure 2.1: An overview of Ellsberg, which runs an instance colocated with each DPI process, and compares the colocated process's behavior to a correct protocol implementation. We assume the TLA$^+$ protocol specification is provided by the protocol designer, and programmers derive a Ellsberg specification from this specification.

This chapter describes an RPRC algorithm that we have implemented in a system called Ellsberg. We designed Ellsberg to work with existing unmodified DPIs, to minimize communication and processing overheads, and to ensure that it had no effect on a DPI's failure guarantees. These design goals allow Ellsberg to be used in deployment and to find bugs that were missed during testing.

Ellsberg (Figure 2.1) consists of a set of co-located instances that run alongside (that is, co-located with) each DPI process. An Ellsberg instance has access to a trace of incoming and outgoing messages sent by its colocated process, but has no access to its internal state. Furthermore, Ellsberg instances are designed to work without communicating with each other: each instance processes the trace produced by its colocated process and can independently generate a notification when it observes an implementation bug. Our design builds on the observation that a DPI correctly implements a distributed protocol if and only if all processes executing the DPI have correctly implemented the protocol, allowing us to check implementation correctness without requiring additional communication or coordination. In cases where the protocol has been proven to maintain global safety properties (*e.g.*, agreement or linearizability), Ellsberg's local checks are sufficient to ensure that administrators are notified when these properties are

violated.

Administrators and programmers use Ellsberg (Figure 2.1) as follows: before deployment, the administrator (or DPI programmer) translates the protocol the DPI purportedly implements into an Ellsberg specification and tests it (Section 2.4) to check correctness. When deploying the DPI, the administrator colocates an Ellsberg instance, configured with protocol specification, with each DPI process, and configures the DPI process so that the colocated instance has access to a trace of all messages received or sent by the process (Section 2.3.1). Each instance uses the specification and trace to simulate a correct implementation of the protocol and reports a bug if the process's behavior diverges from the simulated behavior.

We needed to address two challenges when developing Ellsberg: (a) **Concurrency within DPI processes.** Many DPI implementations are concurrent, and to avoid false notifications, Ellsberg needs to ensure that its simulation state corresponds to the DPI's protocol state. However, the DPI's protocol state depends on the order in which messages are processed, and the trace available to Ellsberg does not reveal this order. Consequently, Ellsberg instances need to consider all simulation states reachable by reordering messages in the trace, which can grow exponentially over time. Therefore, to be feasible Ellsberg needs to efficiently explore the set of reachable states; and (b) **Incremental checking,** by which we mean the ability to check DPI correctness without needing to process the entire trace or requiring access to messages sent or received in the future. Incremental checking prevents us from using existing approaches for efficiently processing multiple simulation states: these approaches assume access to a complete trace (including future messages), or assume that messages are annotated with a vector clock [83, 85] that records processing order.

Ellsberg addresses these challenges using a novel incremental RPRC algorithm (Section 2.3.4): each Ellsberg instance maintains the set of simulation states the associated DPI process can be in, and uses breadth first search to update the instances states and check DPI correctness. We use two optimizations to speed up breadth first search: we prune branches

whose execution produces a simulation state that is guaranteed, based on the observed trace, to differ from the DPI's protocol state; and we identify messages whose effect will not change after being reordered with future messages and apply them immediately.

We have evaluated Ellsberg using three open-source DPIs: etcd and ZooKeeper, which are consistent key-value stores that are used by projects such as Kubernetes for configuration and state management; and RedisRaft, an extension of the Redis key-value store that adds fault-tolerance. Both etcd and RedisRaft implement the Raft [68] consensus protocol, and ZooKeeper implements the Zab [41] atomic broadcast protocol. Our evaluation (Section 2.6) shows that Ellsberg can detect bugs in these systems, that it can do so with minimal impact on response latency (we observed a worse case impact of 11% on the 99th percentile latency), no impact on throughput, and has modest processing and memory requirements.

Although the RPRC approach is general, and can be used with any distributed protocol that assumes fail-stop behavior and the asynchronous (or partially synchronous) model, it has an important limitation: it only suffices to detect protocol bugs that lead to changes in either the content or order of messages sent or received by a DPI process.We cannot detect several important classes of bugs, including ones that lead to livelock or deadlock, corrupt stored data, or degrade performance, and must rely on tools developed by prior work (Section 2.2.1) to detect these bugs. Nevertheless, as our evaluation (Section 2.6) shows, safety bugs of the kind we find do occur in practice.

2.2 Other existing approaches and related work

Our goal is to catch protocol bugs in deployed DPIs that can lead to violations of safety properties. Before describing our approach, RPRC, we first review existing approaches, focusing on those that prevent or detect safety bugs. We categorize these work into five classes:

Trace Validation. A recently developed approach [16, 38] validates totally ordered traces generated during the testing against a TLA$^+$ specification. This approach seeks to check protocol refinement at test time. However, it can only check execution traces generated by tests and, consequently can miss bugs. By contrast, Ellsberg is designed to minimize overheads, allowing it to be used in deployment and uncover bugs that were not found during testing. As a result, both approaches are complimentary: trace validation identifies bugs before deployment, while Ellsberg identifies bugs in deployment.

Trace-validation also makes different assumptions than Ellsberg. It assumes that there is a close correspondence between implementation and specification, assuming that the implementer used the TLA$^+$ specification as a blueprint when writing code ([16, §1]) or that the specification was reverse engineered from the implementation. For DPI implementations, this requirement means that the DPI must log all changes to state variables, and log 'linearization points' that correspond to TLA$^+$ state transitions. Thus, in contrast to our approach, DPIs must be implemented (or updated) for use with trace validation. Howard et al [38, §6.1] report that they needed to add 15 additional log statements to capture linearization points, which requires understanding both the protocol specification and the implementation. Similarly, for specifications, requiring close correspondence between implementation and specification means that specifications must be changed as applications are optimized or changed, and cannot necessarily be reused across applications. Indeed, the CCF effort had to develop a new detailed Raft specification [38, §4, §5] to use trace validation. By contrast, Ellsberg does not require close correspondence between the specification and the traces, and in our evaluation, we reuse the same Raft specification (modulo a few changes to account for the use of different reconfiguration protocols) for both etcd and RedisRaft.

Furthermore, the trace-validation algorithm relies on two assumptions that make it challenging to use this approach in deployment: It assumes that (a) checks are run after the test has terminated; and (b) the test produces a totally ordered output. These assumptions require addi-

tional coordination – *e.g.,* Microsoft CCF [38, §6.1] uses a test driver to serialize node execution and a synchronized global clock to order process logs, and to decide when a test has completed – which affects a DPI's fault tolerance guarantees and performance.

Protocol Verification. Prior work has described several strategies to prove a distributed protocol's safety properties. Verdi [92], Diesel [86] and others [93] provide frameworks that make it easier for users to write proofs that can be mechanically checked by a proof assistant (*e.g.,* Coq [89]). Ivy [72, 73] provides an interactive tool to help users produce inductive invariants that can be used to prove the protocol's safety properties with an SMT solver. Recent works [59, 71, 95, 96] have extended Ivy's approach and automated the process of inferring inductive invariants.TLA$^+$ [10, 66, 98] allows users to model check distributed protocol specification to find safety violations. Each of these strategies offers varying levels of automation and imposes different requirements on how protocols are specified. While these systems can prove that a protocol is correct (i.e. it maintains desired safety properties), they cannot prevent implementation bugs.

Use refinement proofs to prove implementations correct. This approach tries to derive the correctness of DPI implementations from verified protocols using refinement proofs. IronFleet [37] takes as input safety properties (referred to as a high-level specification), a protocol specification together with an implementation, and prove using Dafny [52] that (a) the protocol specification satisfies the specified safety properties, and (b) the implementation refines the protocol and thus also meets the safety properties. Recently, pretend synchrony [33] has shown how to reduce the proof burden for IronFleet by restricting the communication primitives used by the program. Concurrently, Verdi [92, 93] and Diesel [86] implement refinement proof using Coq's Ocaml extraction capabilities. All of these frameworks produce provably correct DPIs, however, to ensure that refinement proof generation and checking is feasible, they need to limit how DPI code is written, what libraries are used, and how the code is optimized [55].

Furthermore, these approaches make assumptions about the runtime environment, including assumptions about system call semantics and the network. Prior work [30] has shown that these assumptions are sometimes wrong, thus provably correct DPIs can still violate safety. Therefore, RPRC systems are useful even when using refinement proofs to produce provably correct implementations.

Use model checking and fuzzing to test existing implementations. Nearly all deployed DPIs use unit and integration tests to catch bugs. However, as with any large software system, developer provided tests cannot provide sufficient coverage. Prior work has developed tools for automatically generating tests. These tools can be classified into ones that use fuzzing or randomized testing [6, 46, 70], and ones that use model checking [3, 5, 6, 44, 45, 51, 64, 91, 94]. Many of these approaches [5, 46, 51, 70] are explicitly designed to be used with existing DPIs. Recent work from Majumdar and Niksic [62] has argued that randomized testing approaches are effective in finding DPI bugs. However, as is common with testing approaches, these tools cannot guarantee that all bugs are found. RPRC complements testing, allowing developers and administrators to discover bugs that occur when DPIs are deployed, but were not found during testing. However, note that as is the case with any runtime verification tool, bugs discovered using RPRC in production can also be found by a fuzzer that runs for long enough and explores enough of the implementation's code paths.

Runtime verification to find bugs in DPIs. Runtime verification approaches, *e.g.,* Dinv [35], D3S [54], Pip [82], Aragog [97], Oathkeeper [57] and Hydra [81], check program properties at runtime. These approaches are designed to check global properties, *e.g.,* agreement or state consistency, which are often expressed as predicates over the state of all (or some subset) of DPI processes. Consequently, these tools check program properties by first collecting a consistent snapshot of the relevant DPI state [13] and then evaluating a predicate on this state (see Francalanza et al's survey [32] for details). The use of DPI state snapshots allows these tools to check

richer properties without concerns about decidability or scaling, *e.g.,* Aragog [97] checks performance and probabilistic staleness properties. However, the need to collect consistent state snapshots adds coordination and communication overheads, and makes verification in the presence of DPI process failures challenging. Recent work has focused on reducing these overheads by minimizing the amount of process state collected [57, 81, 97], using fork-join parallelism to distribute predicate checking [35, 54, 81, 82, 97]. Other work has used multivalued logics [9, 14] to mitigate the effect of failures on runtime verification.

Our approach: Runtime protocol refinement checking (RPRC). RPRC seeks to combine mechanisms from both refinement proofs and runtime verification. Similar to refinement proofs, RPRC aims to ensure that each DPI process correctly implements a protocol, whose safety has verified using other approaches. However, unlike traditional refinement proofs which do so statically, RPRC's refinement checking occur at runtime to compare each DPI process' behavior to that of its protocol specification. Thus, RPRC avoids the feasibility concerns that lead tools like Verdi, Dafny and Verus to impose constraints on how DPIs are implemented. Additionally, runtime refinement checking does not require strong assumptions about system call semantics that are required by existing refinement tools. Unlike runtime verification, RPRC checks that DPI implementation refines a distributed protocol rather than checking that a safety property holds. Refinement can be checked locally without cross-node communication and coordination, and without imposing any limits on fault tolerance. By contrast, checking properties requires gathering that state at all processes, which necessitates coordination and is unavailable under failures.

2.2.1 Other Related Work

The tools discussed above focus on detecting safety bugs in DPIs. Below we discuss tools that look beyond safety to other issues including performance bugs, non-fail-stop failures and how to fix bugs once they are found. We discuss these below:

Distributed tracing tools. Distributed tracing [7, 21, 31, 61, 74, 87] is the most widely deployed approach to finding bugs in deployed DPIs. These tools generate logs that trace requests through the system, and several tools utilize these trace logs [8, 18, 43, 48, 49, 60, 65] to debug performance problems and failed user requests. These tools are valuable for debugging bugs, but cannot detect them.

Checking behavior under failures. Several tools and engineering practices, including chaos engineering [84], lineage driven fault injection [3–5, 63], crash consistency checking [2], and analysis tools for partial and correlated failures [40, 56, 100] have been developed to improve DPI behavior when recovering from crashes. We assume a fail-stop model, and do not currently handle these failures.

Performance and configuration bugs. Recent work has also developed tools and approaches to identifying performance bugs including gray failures [40], metastable failures [39], and bugs due to misconfiguration [101].

Record-replay based debuggers. Bugs, once found, need to be further diagnosed for debugging. Prior work has proposed several approaches for debugging DPIs [1, 8, 56, 85]. As we discussed previously, these include record-replay [8, 83, 85] based debuggers which, similar to RPRC, also need to consider multiple interleavings when reproducing bugs.

2.3 Ellsberg Design

We now describe Ellsberg, a RPRC system that we have implemented. We first state our assumptions about DPIs and the deployment environment (Section 2.3.1) and describe a ticket-lock service that we use as a running example in this section (Section 2.3.2). Then, we discuss the protocol specification a user must provide to Ellsberg (Section 2.3.3), and our RPRC algorithm

(Section 2.3.4).

Overview. Ellsberg (Figure 2.1) performs runtime refinement checks on whether a deployed DPI correctly implements a protocol by comparing each DPI process's behavior against a user-provided protocol specification (Section 2.3.3). To do so, we require administrators to co-locate an Ellsberg instances with each DPI process, and to configure the deployment so that the Ellsberg instance has access to a trace of all messages sent and received by its co-located DPI process. At a high-level, the Ellsberg instance uses the following RPRC algorithm to check its co-located DPI process: it checks refinement by using the incoming messages in the trace to simulate potential protocol executions (using a protocol specification) and decides refinement check has failed if an invalid or out-of-order going message appears in the trace. It notifies administrators when the refinement check fails. Because Ellsberg must account for internal concurrency within a DPI, when simulating the protocol it maintains multiple simulation states, and uses breadth first search to consider multiple event interleavings (or schedules). Ellsberg uses outgoing messages in the trace to prune the set of simulation states that must be maintained, and the event interleavings considered.

2.3.1 Assumptions and Guarantees

Ellsberg assumes that the DPI and protocol work under the asynchronous (or partially synchronous) model and fail-stop failures. We do not consider byzantine failures.

When checking refinement, Ellsberg instances assume that the user-provided protocol specification is correct. We provide a testing tool (Section 2.4), that administrators can use before deploying Ellsberg to check the specification's correctness using the protocol's TLA$^+$ specification.

We assume that the Ellsberg instance and its co-located DPI processes share fate, i.e. if an Ellsberg instance fails, its co-located DPI process also fails. We also require that the trace faith-

fully records all messages sent and received by its co-located DPI process, and is incrementally updated, *i.e.,* messages are added as the DPI process sends or receives them. Messages in the trace need to be annotated with the connection on which they were sent or received. We impose some ordering requirements on the trace. Specifically, we require that messages received over the same connection appear in order in the trace, and that outgoing messages follow program order, *i.e.,* they appear in the trace in the order in which they are sent, and that any incoming messages processed by the DPI before producing an outgoing message m appear before m in the trace. Our prototype uses an IPC channel, that connects each Ellsberg instance to its colocated process, to collect the trace (Section 2.5). Other approaches, *e.g.,* one where a network proxy mirrors messages to the Ellsberg instance, can also be used instead. Beyond the trace, Ellsberg has no access to DPI state: it cannot read the DPI process's memory, nor does it know the order in which received messages are processed.

Guarantees. When our assumptions are met, Ellsberg guarantees that it will generate an alert shortly after a DPI processes sends a message that is inconsistent (either because of its contents, or the order in which it was sent) with the specification, *i.e.,* shortly after a refinement violation is observed. Consequently, Ellsberg does not raise false-alarms because alerts are only generated when a violation is observed. However, similar to trace validation [16, 38], Ellsberg cannot prove that an implementation is correct because (a) it cannot detect refinement violations that do not result in the DPI sending out an inconsistent message; and (b) it is not guaranteed to have observed all possible DPI behaviors.

2.3.2 Running Example: Ticket Lock Service

We use a simple, non-fault tolerant, ticket lock-service as an illustrative example in this section. The service, shown in Listing 2.1, runs on a single server and maintains two integers: *highest* tracking the largest ticket the service has previously granted, *current* tracking what client should

```
1 struct State { /**/
2     highest: int,
3     current: int,
4     held: bool
5 }
6
7 fn assign(state: State) { /**/
8     let ticket = state.highest;
9     state.highest = state.highest + 1;
10    respond(Assigned(ticket))
11 }
12
13 fn acquire(state: State, ticket: int) { /**/
14    if state.current ==  ticket{
15        state.held = true;
16        respond(Acquired(ticket))
17    }
18 }
19 fn release(state: State) { /**/
20    let ticket = state.current;
21    state.current = state.current + 1;
22    state.held = false;
23    respond(Released(ticket))
24 }
```

Listing 2.1: Example ticket-lock server.

go next, and *held* which tracks whether a client holds the lock. The service implements three RPC calls:

1. `assign` (line 7), which assigns the client a ticket, increments *highest* so that clients get unique tickets, and responds with the assigned ticket number.
2. `acquire` (line 13), which takes as input a ticket number (*ticket*, previously acquired using `assign`) and checks if it is the client's turn to acquire the lock ($current == ticket$). If it is the client's turn, the service sets *held* to true and responds with an *Acquired* message.
3. `release` (line 19), which releases the lock by incrementing *current*, setting *held* to false, and responds with a *Released* message.

For ease of exposition, we assume that clients using this service are correct, and that a client does not call `release` unless it holds the lock. A correctly implemented version of this protocol guarantees mutual exclusion, ensuring that no-more than one client holds the lock at a time. We

use this simple protocol to illustrate how to specify a protocol in Ellsberg, but we use real DPIs (etcd, ZooKeeper and RedisRaft) for the evaluation (see Appendix A.3 for specifications).

2.3.3 Definitions and Specification

We model a distributed protocol as a collection of communicating processes, each of which is specified by an I/O automaton [28, 58]. More concretely, Ellsberg requires users to specify the distributed protocol as a state machine (executed by each process), whose execution is driven by a sequence of *events* that are *applied* to it. There are two types of events: the delivery of a message, and timeouts. When an event is applied, a process can update its state and send 0 or more messages.

The terms **pending events**, **reachable states** and **inducing states** are defined as follows:

Pending Events: A pending event is one that can be applied to a process, and consists of previously-received messages that the have not yet been processed, and timeouts. Timeouts in a DPI process are not recorded in the trace, and our model assumes that a timeout is always pending. We use the term *schedule* to refer to a sequence of events.

Reachable state: Given a process p in state s, we say that state s' is reachable if and only if there exists a schedule of events (some of which might not yet be received) that when applied to s result in state s'.

Inducing states: Given a message m, we say that state s is m-inducing (we drop the prefix $m-$ when the message is clear from the context) if and only if there exists state s' and event e, such that applying e to a process in state s' results in the process *sending* the message m and then transitioning to state s. In the ticket-lock example (Listing 2.1), the m-inducing state for an `Acquired` message for ticket 2 is any state s, where $s.current = 2$. Observe that in general, a message m does not have a unique inducing state, and some messages might have an unbounded

number of inducing states: in the previous example, any state s for which $s.current = 2$ is m-inducing, regardless of $s.highest$'s value.

2.3.3.1 Specification

Ellsberg requires users to provide two inputs: a specification for the distributed protocol the DPI purportedly implements and mapping functions that allows Ellsberg to map network messages sent or received by the DPI into messages that appear in the specification. In the remainder of this section, we only refer to messages that appear in the protocol specification. In Listing 2.2, we use the ticket-lock service's (Section 2.3.2) specification to illustrate the parts of a Ellsberg specification.

A Ellsberg protocol specification consists of:

(i) The definition for a *protocol state structure*, `ProtState` (line 1), that Ellsberg uses to track the current simulation's protocol state, and a function to create the simulation's initial protocol state. For notational convenience, all elements of the `ProtState` are nullable and we use this to encode sets of states.

(ii) A transition function, `apply` (line 19), that takes as input a protocol state s, a collection of sets of pending events and an event e, and returns the protocol state produced by applying e to s. We guarantee that all events passed to the transition function are pending for s. For notational convenience, we use $apply(s, \Sigma)$ to represent the output of applying the sequence of events in the schedule Σ to s.

(iii) A function, `equal`, to check equality between two protocol states s and s'. Two states are considered equal if the distributed protocol exhibits the same behavior for both. In other words, we assume that $equal(s, s') = true$ implies that for any schedule Σ, applying the schedule to both states results in final states that are equal and produces the same sequence of outgoing messages. In the ticket-lock example (line 6) the function compares *current*, *highest* and *held*.

(iv) An inference function, `infer_inducing`, that given an *outgoing* message m returns a `ProtState` of all m-inducing states. This function populates only those values that can be inferred from m, leaving the others unknown. Line 11 shows this function for our running example: we need to consider each type of outgoing message sent by the service when writing it.

(v) A `reachable` function that takes protocols state s, the set of pending messages, and a target partial protocol state S, and returns true if there exists s' such that $s' \in S$ and s' is reachable from $apply(s, e)$. We assume that this function over-approximates reachability, *i.e.,* it might return true even if there exists no reachable s' such that $s' \in S$, but never incorrectly returns false. For brevity, we only show a portion of the reachable function for the running example, which returns false if the state's *current* value is larger the target state (line 30).

(vi) A function, `apply_asap?`, that we use to reduce the simulation's memory requirements and runtime by reducing the number of schedules that need to be considered (Section 2.3.4.2). This function takes a protocol state s and an event e, and returns true if and only if we can safely apply the event immediately, *i.e.,* we do not need to consider schedules that reorder e with respect to other events (including ones that occur in the future). We define `apply_asap?`(s,e) to be correct if it meets the following two requirements:

   (a) **Event e can be reordered.** For any schedule Σ where $e \in \Sigma$ and message m, if $apply(s, \Sigma)$ is m-inducing, then so is $apply(apply(s, e), \Sigma - e)$.

   (b) **Event e does not block outgoing messages.** For any schedule Σ where $e \notin \Sigma$ and message m, if $apply(s, \Sigma)$ is m-inducing, then $apply(apply(s, e), \Sigma)$ is also m-inducing.

   These requirements ensure that applying e to protocol state s does not change the set of reachable m-inducing states, and Ellsberg can avoid exploring schedules that reorder e. An acquire message for a ticket smaller than the server's current ticket meets these requirements in our example (line 38).

(vii) An optional function `lookahead_type` that we use to reduce the number of m-inducing

states found by the `infer_inducing` function, thus reducing simulation time. This function takes as input a message m and either returns `None` (indicating there is no such type) or a message type. As we explain later (Section 2.3.4.2), this function is used in an optimization to improve Ellsberg's efficiency, but its semantics do not affect correctness. Line 41 shows this function for the lock service: when processing an outgoing acquire message Ellsberg looks ahead to find the next assigned message. This is because we cannot infer the value of the `highest` field from an acquire message, but can from an assign message, and considering both reduces the number of m-inducing states,and thus schedules that Ellsberg must explore.

In Section 2.4, we describe how a user can derive Ellsberg specifications from existing TLA$^+$ specifications used to prove protocols correct, and test the derived specification. However, note that deriving a Ellsberg specification from a TLA$^+$ specification is simpler than deriving an implementation: unlike implementations, Ellsberg specifications do not need to interact with communication libraries, consider failure handling, or implement application logic. Furthermore, Ellsberg specifications are written assuming sequential execution, and operators do not need to protect against data races or other concerns common to concurrent code. Consequently, it is easier to derive a correct Ellsberg specification than an implementation from a TLA$^+$ specification.

2.3.4 The Ellsberg Algorithm

We now detail the Ellsberg algorithm. Concurrency and incrementally checking DPI correctness (so Ellsberg can report bugs soon after they are observed) were the two main challenges we addressed when developing this algorithm.

Concurrency is challenging because the protocol's behavior depends on the order in which events are applied, but Ellsberg does not know the order in which the messages and timeouts were processed by its associated DPI process. Similar to other work, we resolve this by considering multiple schedules consisting of different interleavings of incoming messages in the trace

and timeouts.

Incremental checking makes constructing multiple schedules challenging because the DPI process might process a received message m, which appears in the trace, after message m' that has not yet been received, and thus does not appear in the trace. We resolve this challenge using the assumed the trace ordering property that guarantees that any received messages processed by the DPI process before it produces an outgoing message m must appear before m in the trace, and that outgoing messages appear in program order. Thus, when checking an outgoing message m, Ellsberg only considers schedules that contain incoming messages appearing before m in the trace. For notational convenience, we refer to the trace before m as the m-prefix. Furthermore, if the trace contains a sequence of outgoing messages $m_0, m_1, \ldots, m_n$, the simulation needs to only consider schedules which use incoming messages in the m_0-prefix before producing outgoing message m_0, messages in the m_1-prefix before producing outgoing message m_1, etc.

Ellsberg uses this observation to implement an incremental checking algorithm (Listing 2.3): Each Ellsberg instance maintains a set of simulation states S, which we define as a protocol state with a set of collections of pending message (line 1). For notational convenience, we use s_{sim} to refer to simulation states, and $state(s_{sim})$ to reference the underlying protocol state. Initially S contains a single element: a simulation state with the initial protocol state and an empty pending message set. The algorithm iterates through the trace, and adds incoming messages (with one exception discussed in Section 2.3.4.2) to the pending message set of all simulation states $s_{sim} \in S$ (line 16). It processes an outgoing message m by finding all m-inducing simulation states that can be reached from any $s_{sim} \in S$ using s_{sim}'s pending messages and zero-or-more timeouts (line 41, Section 2.3.4.1). The algorithm notifies an error (line 48) if no m-inducing simulation states are reachable, otherwise it updates S to be the set of reachable m-inducing simulation states (line 50).

For example, if the ticket-lock service sends a message m indicating that the client with ticket 2 has acquired the lock ($m = Acquired(2)$) message, and S contains a single simulation s_{sim} for which $state(s_{sim}).held = true$ and $state(s_{sim}).current = 1$, then Ellsberg would throw an error

if $pending(s_{sim})$ did not contain a release message.

Observe that because the trace is in program order, and the algorithm processes the trace iteratively, Ellsberg only considers incoming messages (and zero-or-more timeouts) in the trace's m-prefix when processing outgoing message m Furthermore, since outgoing messages are processed iteratively in order, the algorithm checks both the order and contents of outgoing messages sent by the DPI process.

2.3.4.1 Finding reachable states

The algorithm described above needs to find m-inducing simulation states reachable from the current simulation state s_{sim}. To do so, it first uses the `infer_inducing` function to compute *target* (Listing 2.3 line 29), which is the partial protocol state derived from outgoing message m. The algorithm then calls `find_reachable` with s_{sim} and *target* as arguments to find the subset of simulation states that are both consistent with the partial state *target* and reachable from the current simulation state s_{sim}. To make our state exploration efficient, we must design `find_reachable` to work efficiently by returning the smallest set of simulation state. In particular we require that if $s, s' \in find\_reachable(s_{sim}, t)$ then $equal(s, s') = false$. We had to solve two challenges when designing the `find_reachable` algorithm to meet this requirement:

Challenge 1: Many schedules can lead to the same m-inducing state. The schedule used to reach a simulation state s_{sim} dictates its pending message set. But there exist cases where `find_reachable` can reach two (or more) simulation states s_{sim} and s'_{sim} with the same underlying protocol state (*i.e.,* $equal(state(s_{sim}), state(s'_{sim})) = true$) but different pending message sets $pending(s_{sim}) \neq pending(s'_{sim})$. Which of these simulation states is returned can affect correctness. In this case, our handling depends on why multiple schedules reach the same semantically equivalent state:

a. There might exist cases where a protocol state s can be reached through two schedules

Σ and Σ', and Σ is a subsequence of Σ' . This often happens because applying some message m has no effect on the protocol state s. For example, consider the ticket-lock server (Listing 2.1) with simulations state s_{sim} such that $state(s_{sim}).current = 1$ and $pending(s_{sim})$ contains an acquire message m_a for ticket 2 ($m_a = acquire(2)$). Applying m_a to s_{sim} does not change its protocol state because the client cannot acquire the lock, and $equal(state(apply(s_{sim}, m_a)), state(s_{sim})) = true$. But, the pending set of $apply(s_{sim}, m_a)$ does not contain m_a and is a subset of s_{sim}'s pending set.

However, including the simulation state with a smaller pending state in `find_reachable`'s return can lead to false alarms: consider the case where s_{sim}'s pending set is $\{m_a, m_r\}$ where $m_r = release()$, and we are searching for an $Acquired(2)$-inducing state. It is clear that an inducing state is reachable from s_{sim} using the schedule $\Sigma = [m_r, m_a]$, but is not reachable from $apply(s_{sim}, m_a)$, whose pending set does not include m_a.

Therefore, in this case `find_reachable` should add the simulation state with the *largest pending set*[1] to the set of returned state. We ensure that this is the case by having `find_reachable` use *breadth-first search* (Lines 12—45) to find reachable target states. Breadth first search ensures that shorter schedules, and thus simulation states with larger pending message sets, are explored before simulation states with smaller pending message sets.

b. Semantically equivalent states might also be reached through two schedules Σ and Σ' neither of which is a subsequence of the other. We resolve this by having simulation states in Ellsberg track a set of pending message sets, allowing us to use a single simulation state to track the pending message sets in this case (Lines 31 and 40).

Challenge 2: Terminating the search. Our algorithm searches through schedules that consist of pending incoming messages and zero-or-more timeout. These schedules can have unbounded length, because timeouts are always enabled. This poses a challenge, since

[1] Σ is a subsequence of Σ', and thus the pending set of the simulation state reached from Σ must be a superset of the other.

`find_reachable` might explore unnecessarily long schedules and might not terminate. We address this problem in two ways: (a) We avoid redundant explorations by tracking simulation states that have already been explored by `find_reachable` (line 32). During breadth-first search `find_reachable` checks if a simulation state s_{sim}'s protocol state ($state(s_{sim})$) is the same as that of a previously explored simulation state (line 40, the `contains_state` function finds simulation states with equal protocol states): if not, it enqueues that simulation state for exploration, and otherwise terminates exploration for that simulation state. (b) Before scheduling exploration for simulation state $apply(s_{sim}, e)$, we use the `reachable` function in the specification to first check if a target m-inducing state can be reached from s_{sim} (Line 22). We prune exploration from $apply(s_{sim}, e)$ if the `reachable` function returns false.

2.3.4.2 Optimizing State Exploration

We use two performance optimizations:

Avoiding redundant exploration. The `find_reachable` algorithm explores different schedules which reorder incoming messages and zero-or-more timeout events in its search process. However, if the current simulation state s and a pending message m meet the `apply_asap?` requirements (Section 2.3.3) then when m is applied does not affect the search.

Therefore, we use the `apply_asap?` function to improve exploration efficiency in two places: (a) When processing an incoming message m from the trace (Listing 2.3), Ellsberg checks whether it can be applied immediately to a simulation state $s_{sim} \in S$, and if so it uses the recursive function `prune_asap` (Line 7) to replace s_{sim} with the result of applying $apply(s_{sim}, m)$ and all other messages that can be applied immediately (Line 20); and (b) in the `find_reachable` (Listing 2.4) algorithm when adding a new simulation state s_{sim} to the set of states that need to be explored (Line 25) we again use `prune_asap`. Note, that while this optimization is similar in principle to partial-order reduction [29] (POR), our technique is designed for the incremental

setting.

In Appendix A.1 we describe how we can mechanically check that `apply_asap?` is correct, and thus the use of this approach does not come at the cost of correctness.

LOOKING AHEAD TO THE NEXT OUTGOING MESSAGE. The number of m-inducing states found by `find_reachable` is determined by the outgoing message m being considered: an outgoing message m that reveals little about the DPI's protocol state, *e.g.,* client response messages, may allow for a large set of reachable m-inducing states. Consequently, when processing m, `find_reachable` might return a large set of simulation states that the ELLSBERG instance would use when it next processes an outgoing message. However, the time taken to process an outgoing message depends on the size of the simulation set S because it determines the number of `find_reachable` calls made (Listing 2.3 line 39). Smaller simulation sets are thus preferable. ELLSBERG implements an optimization that can reduce the set of states returned by `find_reachable` in some cases. Specifically, the optimization "looks ahead" to the next message m_T of type T that occurs after m in the trace and passes to `find_reachable` an optional argument (`ahead`) containing the m_T-inducing state, allowing `find_reachable` to prune any simulation states $s_{sim} \in S$ from which an m_T-inducing state is not reachable (Listing 2.4 line 24).

We use the `lookahead_type` function in the user-provided specification (Section 2.3.3, Listing 2.2) to decide when to use this optimization. As a reminder, the `lookahead_type` takes as input the outgoing message m that ELLSBERG is processed, and either returns `None` (indicating lookahead should not be used) or a message type T. If the function returns a type T, ELLSBERG scans the trace, starting at m to find the next outgoing message m_T of type T (Listing 2.3 line 33) and passes the m_T-inducing state to `find_reachable`. The `find_reachable` function then only returns simulation states that are both m-inducing and m_T-inducing, a smaller set than the set of all m-inducing sets. Note that our use of `lookahead_type` ensures that its output has no impact on correctness: if the function returns type T, then any simulation state s_{sim} pruned by

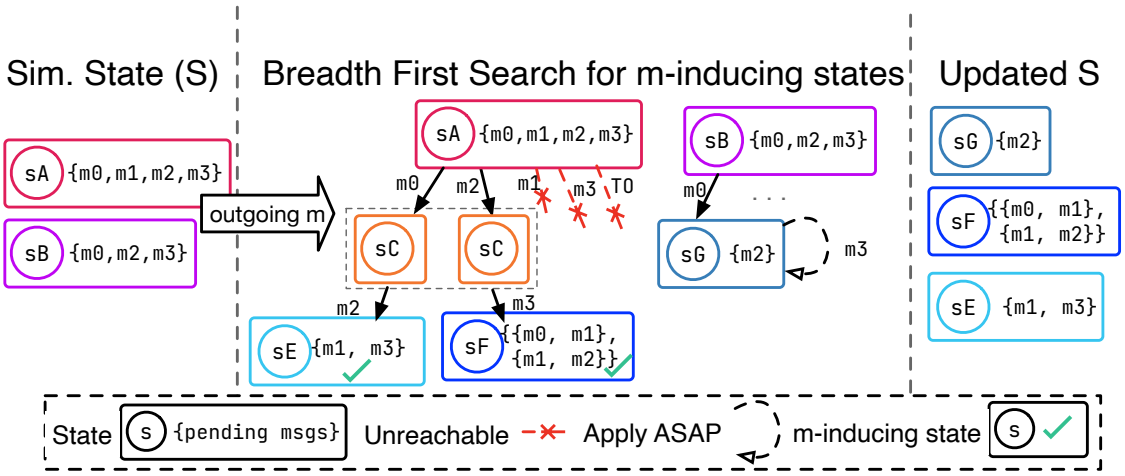


Figure 2.2: An overview of how a ELLSBERG processes outgoing message m using BFS to find all m-inducing states reachable from the current simulation state.

`find_reachable` would have been pruned by a future call to `find_reachable` when considering the next outgoing message of type T.

2.3.5 Algorithm Summary and Generality

In summary, each ELLSBERG instance works as follows: it maintains a set of simulation states S and iterates through the trace. An incoming message m in the trace is processed (Listing 2.3 Line 16) by iterating through S, and checking for each simulation state $s_{sim} \in S$ if the message should be immediately applied: if so, s_{sim} is replaced by $apply(s_{sim}, m)$, if not m is added to s's pending set. An outgoing message m (Figure 2.2) in the trace triggers a check to determine if the DPI is buggy (Listing 2.3 Line 27), which involves iterating through the simulation states $s_{sim} \in S$ and using the `find_reachable` (Listing 2.4) algorithm to determine if an m-inducing state is reachable from s_{sim}. ELLSBERG reports a bug if no m-inducing states are reachable from any simulation state $s_{sim} \in S$, otherwise S is replaced by the union of all reachable m-inducing simulation states.

When does ELLSBERG detect a bug? ELLSBERG only has visibility into what messages have been sent or received by DPI thus far, and hence, at any point in time, it can only detect bugs that

are apparent from the trace prefix provided to it thus far. Furthermore, even for this prefix, our approach checks if there exists any schedule (*i.e.,* a total order) of incoming messages that when applied to the specified distributed protocol produces the sequence of output messages observed in the trace prefix. So, as we noted earlier (Section 2.3.1) Ellsberg can only detect a bug once it observes an output message that is inconsistent with the specification.

Generality & Applicability. Our algorithm assumes that the protocols being checked can be specified as a collection of processes, each of which is an I/O automaton. Concretely, this assumption means that events (received messages or timeouts) must be applied atomically, and a messages behavior cannot be influenced by a message being applied concurrently. This condition holds for many but not all distributed protocols, *e.g.,* it does not hold for a distributed concurrency control protocol where multiple transactions can be executed simultaneously and have their results validated after the fact.

An additional protocol assumption is required to make incremental checking feasible: protocols must include one or more messages that (perhaps together) allow Ellsberg to infer the entire state of a DPI process. Absent such messages, Ellsberg must maintain an ever increasing set of partial states and consider a growing number of schedules when checking outgoing messages, making the use of Ellsberg infeasible in practice. These messages exist for most protocols, including the RSM protocols we use in our evaluation. However, we have encountered one case where this assumption does not hold: databases that use multi-version concurrency control, and whose messages do not include information about what version a transaction read from or updated. In this case applying Ellsberg requires us to consider all possible orderings for concurrent transactions, incurring similar complexity as prior transaction verification approaches [88, 102].

System	Ellsberg LOC	Baseline	Baseline LOC
etcd	952	CCF Raft [38]	1503
RedisRaft	894	CCF Raft	1503
ZooKeeper	989	ZooKeeper TLA$^+$ [41]	1615

Table 2.1: Lines of code in Ellsberg specification when compared to those used by existing tools. For baselines, we use the Raft specification from Microsoft CCF [38] etcd and RedisRaft, and a recent implementation derived TLA$^+$ specification [41] for ZooKeeper.

2.4 Writing Ellsberg Specifications

To use Ellsberg, users or protocol authors need to translate existing specifications, *e.g.,* ones used to verify safety, into Ellsberg specifications. Our evaluation uses specifications derived from TLA$^+$ protocol specifications: for Raft we used the TLA$^+$ specification [67] provided by the Raft authors, while for ZooKeeper we used an implementation derived TLA$^+$ specification [41].

Table 2.1 shows the length of our specifications ('Ellsberg LOC' column). We use the same Raft specification for etcd and RedisRaft, the difference in length is because they support different reconfiguration protocols. To put our specifications length into context, we also report the length of specifications used by other tools: for the two Raft DPIs we compare to the Raft specification used by Microsoft CCF (written specifically for use with that tool), and for ZooKeeper we compare to the implementation derived TLA$^+$ specification we used. Note, that the Microsoft CCF Raft implementation supports a different (and simpler) reconfiguration protocol than etcd, but the protocols are comparable. Our specification is comparable in length to these baseline specifications, and indeed a few lines shorter. Appendix A.3 provides additional details about the Ellsberg specifications, including a detailed breakdown of lengths for each part (Appendix A.3.1).

We adopted the following approach to derive the `ProtState` structure, and the `equal`, `infer_inducing`, and `apply` functions from TLA$^+$:

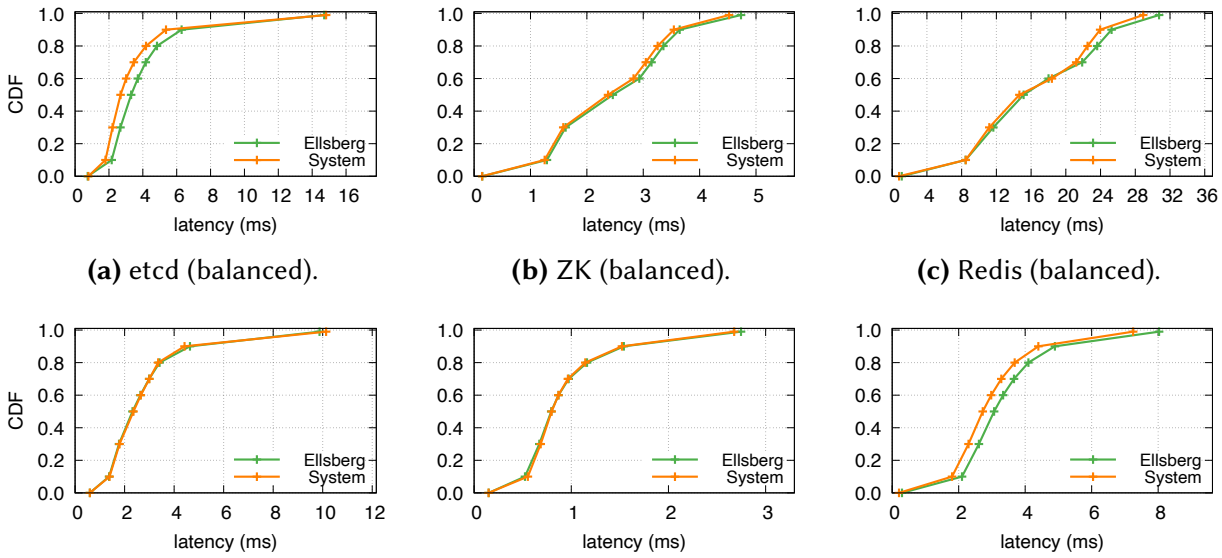
- **`ProtState`**. The `ProtState` structure contains all per-server variables that appear in the TLA$^+$ specification. We derived it by identifying all variables in the TLA$^+$ specification,

filtering out those that are global variables (*e.g.,* `messages` in the Raft specification [67]) or ghost variables (*e.g.,* `voteResponded` in the Raft specification), and adding the remaining variables to the structure.

- **`equal`**. The equal function merely checks that all state variables have the same value, and the language (or runtime) provided equality function suffices.
- **`apply`**. The `apply` function matches over all possible state machine transitions, and updates a provided `ProtState` structure. We derived it by walking through the TLA$^+$ specification to find transitions (*e.g.,* `ClientRequest(i, v)` in Raft) and copying over any state updates that appear in the transition.
- **`inference`**. The `inference` function takes an outgoing message and tries to infer the process' DPI state. To derive this function, we first identify transitions that send messages (*e.g.,* by finding transitions that call `Send` in the Raft specification).Given these transitions, we use three techniques to infer state variables values: (i) If the outgoing message directly uses a state variable, *e.g.,* term, we can infer the state value directly from the message; (ii) If the outgoing message contains a value computed from a state variable, *e.g.,* last log index, we use the inverse computation to infer the state value; and (iii) If the outgoing message is sent only when a particular state variable has a given value, then we can infer the value of the state variable when we observe the message: *e.g.,* we can infer that Raft a node sending a `RequestVoteRequest` message is a candidate.

We had to write de novo `apply_asap?`, `reachable`, and `lookahead_type` functions. Of these, the `apply_asap?` function can be *mechanically proven correct* (described in Appendix A.1), and the `lookahead_type` function has no impact on Ellsberg's correctness (Section 2.3.4.2).

Testing Ellsberg specifications. We also provide a testing tool, inspired by Mocket [91], to check that the derived Ellsberg specification matches the original TLA$^+$ specification, and does not have translation bugs.



(a) etcd (balanced). **(b)** ZK (balanced). **(c)** Redis (balanced).

(d) etcd (read-heavy). **(e)** ZK (read-heavy). **(f)** Redis (read-heavy).

Figure 2.3: Response latency CDFs from a 5-node cluster when just the DPI is deployed (System) and when a colocated Ellsberg instance is deployed (Ellsberg). 'ZK' refers to ZooKeeper.

Our testing tool takes as input a TLA$^+$ specification, and generates valid message traces (that is message traces that do not lead to safety violations) using this specification. Message traces are generated from the TLA$^+$ specification as follows: first, the tool uses TLC's [47] bounded model checking functionality to generate a transition graph for a bounded number of steps; next, it performs depth first traversal of the transition graph and keeps track of the sequence of states along each traversed path; finally, it translates the sequence of states into message traces. The last step builds on the observation that TLA$^+$ specifications model messages sends and receives as modifications to a message vector, allowing our tool to translate state sequences into sequences of message sends (or timeouts).

The tool then uses the generated traces to exercise Ellsberg specification, and reports a bug if the Ellsberg specification would have (falsely) notified an error for the trace.

2.5 Implementation

We implemented Ellsberg in Go, and our current prototype requires protocol specifications to be written in Go and compiled with Ellsberg. We evaluated our prototype using specifications for Raft [68] and ZAB [41], and used etcd, ZooKeeper and RedisRaft as DPIs.

The Raft specification we use consists of ~900 lines of code, and the ZooKeeper specification consists of ~1000 lines of code. We describe how we developed these specifications from existing machine checked specifications in Appendix A.3. We used the Ellsberg testing tool (Section 2.4) to test both specifications. For Raft, our test ran 15 steps of bounded model checking (this was sufficient to cover all protocol states), and validated the specification using 8,750,468 message traces; while for ZooKeeper, our test ran 20 steps of bounded model checking (again, sufficient to cover all protocol states) and validated the specification using 1,904,456 message traces.

Overall, the Ellsberg implementation (including both specifications) consists of ~4500 lines of code. Of this total, the testing tool consists of about 500 lines of Python code, and the rest is Go code for the Ellsberg instance and both specifications. In addition to the protocol specification, users must also provide Ellsberg with DPI specific code to deserialize and map implementation messages to specification messages. The mapping function for etcd is 356 lines of Go code, for ZooKeeper it is 370 lines of Go code, and for RedisRaft it is 380 lines of Go code.

Our prototype uses an IPC channel to get the trace, and we modified the network APIs for both DPIs to update the trace. Doing so required adding logic (to forward messages) to 11 functions in etcd, 31 functions in ZooKeeper, and 10 functions in RedisRaft. We were careful to ensure that no additional timing or ordering information was sent over this IPC channel. We used IPC for simplicity, but our implementation can adopt alternate approaches, *e.g.,* having a service proxy [20] copy messages to Ellsberg.

2.6 Evaluation

We evaluated Ellsberg using three widely-used open source DPIs: etcd, ZooKeeper, and RedisRaft. Our evaluation focused on three questions: (a) Can Ellsberg detect bugs at runtime? (Section 2.6.2) (b) Can Ellsberg be deployed in production? (Section 2.6.3) (c) How does Ellsberg perform: how long does it take Ellsberg to process a trace, and the impact of our design (Section 2.3.4) on performance? (Section 2.6.4)

2.6.1 Setup and Workload

Our evaluation deployed Ellsberg on 3- and 5-node DPI clusters. Each cluster node was a C6525-25G instance in CloudLab [19], with a 16-core 3GHz AMD EPYC 7302P processor, 128 GB of RAM, two 25Gbps NICs, and ran Ubuntu 20.04. We deployed a DPI process and Ellsberg instance on each node, and used `taskset` to limit the Ellsberg instance to two cores. Ellsberg instances were configured to process trace messages every second.

We exercised the DPIs using workloads derived from YCSB [17]. We use two benchmarks: a *balanced* workload with 50% reads and 50% writes, and a *read-heavy* workload with 95% reads and 5% writes. For both workloads, keys are drawn uniformly at random from among 1 million possible keys. We used 120 concurrent DPI clients as load generators, we found that this maximized observed throughput for all DPIs. To minimize performance variance, we disabled checkpointing for all DPIs.

In the interest of space, we use *ZK* to refer to ZooKeeper and *Redis* to refer to RedisRaft in figure captions in this section.

System	Bug	Type
etcd	741[26]	Linearizable read
	7331[25]	Stale read after election
	12133[23]	Reconfiguration
	7280[24]	Reconfiguration
ZooKeeper	1154[105]	Data inconsistency
RedisRaft	17[76]	Reconfiguration
	19[77]	Linearizable read
	52[79]	Lost updates
	265[78]	Reconfiguration
	Unreported	Reconfiguration

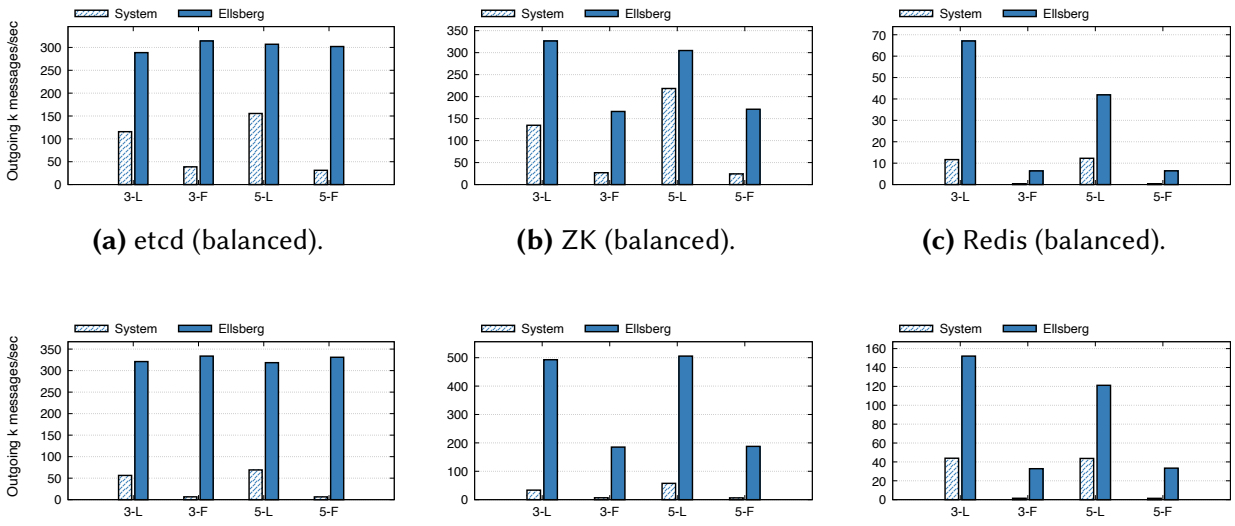
Table 2.2: Bugs used to evaluate Ellsberg's bug detection capability.

2.6.2 Can Ellsberg detect bugs?

We evaluated Ellsberg's ability to detect protocol implementation bugs by reproducing previously reported bugs in etcd, RedisRaft and ZooKeeper, and checking whether Ellsberg notified when the bugs occurred. Due to space constraints we provide the bugs and their root causes in Appendix A.4, but summarize them in Table 2.2. The bugs we tested on include bugs in the linearizable read protocol implemented by etcd [26] and RedisRaft [77]; a stale-read bug in etcd [25] because a new leader's commit index might lag behind the previous leader's commit index; a bug in ZooKeeper's [105] leader election protocol (leading to data corruption); a bug in RedisRaft's caches that leads to lost updates [79]; and reconfiguration bugs in etcd [23] and RedisRaft [76]. During testing, we also identified an unreported bug in RedisRaft's reconfiguration protocol implementation.

2.6.3 Can Ellsberg be used in production?

We demonstrate that deploying Ellsberg in production is feasible by showing that it has low resource requirements, and does not impact DPI performance. In terms of resources: our evaluation uses `taskset` to limit each Ellsberg instance to 2 cores. Furthermore, we found



(a) etcd (balanced). **(b)** ZK (balanced). **(c)** Redis (balanced).

(d) etcd (read-heavy). **(e)** ZK (read-heavy). **(f)** Redis (read-heavy).

Figure 2.4: Ellsberg's throughput compared to DPI throughput (System) for different workloads and DPIs. In these graphs, 3-L and 3-F respectively refer to the leader and follower in a 3-node cluster, and 5-L and 5-F refer to the leader and follower in a 5-node cluster.

(explained in detail in Appendix A.5) that across all our evaluations, each Ellsberg instance tracks one simulation state on average (*i.e.,* $|S| = 1$) and has between 0 and 5 pending messages, and thus has minimal memory requirements. Additionally, Ellsberg does not use the network by design. We thus conclude that Ellsberg has minimal resource overheads.

In terms of DPI performance impact, Figure 2.3 shows the performance of the DPIs with ('Ellsberg') and without ('System') a collocated Ellsberg process. We show results for a 5-node etcd, ZooKeeper, and RedisRaft cluster when running a read-heavy and balanced workload. We omit results for 3-node clusters, but they were similar. We observe that collocating Ellsberg increases tail-latencies slightly: the largest increase we observed was for Redis when running the read-heavy workload(Figure 2.3(f)) where 99th percentile latency increases by 10.7% (from 7.25ms to 8.03ms). These results show that using Ellsberg has minimal impact on response latency, and thus production deployment is feasible.

2.6.4 End-to-End Performance

Ellsberg's detection latency is determined by how often it checks the trace, in our evaluation we use a configuration (Section 2.6.1) where it checks the trace every second (allowing us to batch checking and reduce overheads). Therefore, we evaluate Ellsberg's performance by measuring its throughput, and report results in Figure 2.4. We also report the DPI's throughput when it is run without a colocated Ellsberg process: the DPI throughput both serves as a comparison point to evaluate Ellsberg's algorithmic efficiency; and addresses whether Ellsberg can keep up with the colocated DPI process (if Ellsberg had lower throughput it might not keep up with a loaded DPI). We report results from both 3- and 5-node clusters and both workloads, and measure throughput in outgoing messages per second. The results were collected by measuring throughput in a 10-second interval over 10 experiments. We omit error bars because we observed nearly no variance across experiments.

Our results show that across DPIs and workloads, Ellsberg achieves higher throughput than the DPI: for etcd Ellsberg has 2.0—51.7× higher throughput, for ZooKeeper Ellsberg has 1.4—29.7× higher throughput, and for RedisRaft Ellsberg has 3.1—25.5× higher throughput. In practice, we found that processing a second of trace events took Ellsberg between 30 – 700ms when the co-located DPI node was a leader, 20 – 180ms otherwise. In sum, this means that in our evaluation Ellsberg notifies administrators within 1.7 second of a bug occurring, though this could likely be reduced with a different configuration. Appendix A.5 further evaluates the impact of Ellsberg's optimizations.

2.7 Summary

This chapter proposed *runtime protocol refinement checking*, an approach that combines ideas from refinement proofs and runtime verification to notify administrators of protocol bugs in de-

ployed DPIs. Unlike static refinement proofs, RPRC can be used with existing unmodified DPIs, and unlike runtime verification, RPRC does not require coordination or communication. We also described an algorithm for RPRC and its implementation in Ellsberg, and applied it to three commonly used DPIs: etcd, ZooKeeper, and RedisRaft. We believe that RPRC and Ellsberg provide a practical approach for improving the reliability of deployed DPIs.

```
struct ProtState { /**/
    highest: int,
    current: int,
    held: bool,
}
fn equal(s: ProtState, s': ProtState) -> bool { /**/
    return s.highest == s'.highest
      && s.current == s'.current
      && s.held == s'.held
}
fn infer_inducing(m: Message) -> ProtState { /**/
    return match m {
        Acquired(tkt) =>
            ProtState{highest: None,
              held: true, current: tkt},
        // ...
    }
}
fn apply(s: ProtState, e: Event) -> ProtState { /**/
  match e {
    case Message(Acquire(tkt)) => {
        if s.current ==  tkt{
            return ProtState{highers: s.highest, current:tkt, held:true}
        } else {
            return s
        }
    }, // ...
  }
}
fn reachable(s: ProtState, /**/
             pending: Set[Set[Message]], e: Event,
             p: ProtState) -> bool {
   if state(s).current > p.current {
    return false
    } // ...
  }
}
fn apply_asap?(s: ProtState, m: Message) -> bool { /**/
    return m.type == Acquire && m.tkt < s.current
}
fn lookahead_type(m: Message) -> Message_type { /**/
    if Type(m) == Acquired {
        return Assigned
    }
    // ...
}
```

Listing 2.2: User-provided protocol specification for the ticket-lock server

```
struct SimState { /**/
    state: State,
    pending: Set[Set[Messages]]
}
// The instance's current simulation state.
S: Set[SimState] /**/
fn prune_asap(s: SimState) -> SimState { /**/
    for e in pending(s) {
        if apply_asap?(state(s), e) {
            return prune_asap(apply(s, e))
        }
    }
    return s
}
// Process incoming message m in the trace.
fn process_incoming(m: message) { /**/
    for s in S {
        // Check if m should be applied immediately.
        if apply_asap?(state(s), m) {
            S.replace(s, prune_asap(apply(s, m)))  /**/
        } else {
            s.add_pending_message(m)
        }
    }
}
// Process outgoing message m in the trace.
fn process_outgoing(m: message) { /**/
    // Get m-inducing states.
    let target = infer_inducing(m) /**/
    // Check if we need to lookahead.
    let m_lookahead = if lookahead_type(m) {
        infer_inducing(
            find_next_of_type(lookahead_type(m))) /**/
    } else {
        None
    }
    // Updated simulation state.
    let S' = new Set[SimState]
    for s in S { /**/
        // Find reachable m-inducing states from s.
        S_s = find_reachable(s, /**/
                  target, m_lookahead)
        S' = S'.union(S_s)
    }
    if S'.is_empty() {
        // No reachable m-inducing state indicates
        // divergence b/w spec and implementation.
        raise Error  /**/
    } else {
        S = S' /**/
    }
}
```

Listing 2.3: Algorithm for checking a single outgoing message.

```
1
2 fn find_reachable(s: SimState,
3        target: ProtState,
4        ahead: Option[ProtState]) -> Set[SimState] {
5     // The queue of SimState's to explore.
6     let to_explore = new Queue[SimState]
7     // Previously explored SimState's
8     let explored = new Set[SimState]
9     // The set of reachable m-inducing SimStates.
10    let reachable_inducing = new Set[SimState]
11    to_explore.insert_tail(s)
12    while !to_explore.is_empty() {/**/
13        s' = to_explore.remove_head()
14        explored.add(s')
15        // Is s' m-inducing?
16        if state(s') in t {
17            reachable_inducing.add(s') /**/
18            continue
19        }
20        for e in pending(s') {
21            // Is target reachable from apply(s', e)
22            if reachable(state(s'), pending(s'), e, target) && /**/
23               (ahead.is_none() ||
24                reachable(state(s'), pending(s'), e, ahead)) { /**/
25                s'' = prune_asap(apply(s', e)) /**/
26            }
27            // Equivalent to enqueued state?
28            if to_explore.contains_state(s'') {
29                // Add pending messages in s'' to existing state.
30                merge_pending(
31                   to_explore.get_state(s''), s'') /**/
32            } else if !explored.contains_state(s'') { /**/
33                // Add s'' to the exploration queue.
34                to_explore.insert_tail(s'')
35            } else {
36                // Add s'' to the exploration queue
37                // if pending events differ from
38                // previously explored.
39                if !pending(explored.get_state(s''))
40                    .contains(pending(s'')) { /**/
41                    to_explore.insert_tail(s'')
42                }
43            }
44        }
45    }/**/
46    return reachable_inducing
47 }
```

Listing 2.4: Ellsberg's breadth-first search algorithm to find reachable m-inducing simulation states (passed in as argument `target`) starting from simulation state `s`.

3 Enabling Crash Consistency in Real-World Distributed Systems through Synthesis

This chapter studies how a distributed protocol implementation should order its actions that persist state to durable storage relative to those that transmit externally visible messages. It presents a specification-guided method that synthesizes a message-specific persistence policy and a runtime that enforces the policy in existing implementations. It closes the *coverage* gap of Section 1.1.2: the decisions a real implementation must make in order to correctly recover from node crashes, which the original protocol specification often fail to model.

3.1 The Tradeoff in When to Persist State

Distributed protocols rely on persistent state to preserve their safety guarantees across crashes. For example, a consensus node must not forget log entries, terms, or votes that other nodes or clients have already observed. If such state is lost after a restart, the recovered node may behave inconsistently with the protocol history and violate safety.

Existing protocol specifications treat persistence as instantaneous and free. Figure 3.1 shows two excerpts from Raft's specification. `Restart` specifies which state survives a crash: a restarting

```
\* Server i restarts from stable storage.
\* It loses everything but currentTerm, votedFor, and log.
Restart(i) ==
    /\ state' = [state EXCEPT ![i] = Follower]
    /\ ...
    /\ UNCHANGED <<currentTerm, votedFor, log, ...>>

\* Leader i receives a client request to add v to the log.
ClientRequest(i, v) ==
    /\ state[i] = Leader
    /\ LET entry == [term |-> currentTerm[i], value |-> v]
       IN  log' = [log EXCEPT ![i] = Append(log[i], entry)]
    /\ UNCHANGED <<messages, serverVars, ...>>
```

Figure 3.1: Excerpts from Raft's TLA⁺ specification [67]. `Restart` fixes which state survives a crash, but no action states when that state reaches the disk: `ClientRequest` appends an entry to `log` and ends there.

server loses everything except its `currentTerm`, `votedFor`, and `log`. In `ClientRequest`, a leader appends a new entry to its `log`, and because `Restart` preserves the log unconditionally, the entry is durable the moment the action executes. Doing so would require an immediate, synchronous disk write after every change to the state before protocol execution can continue. For example, upon receiving a client request, the leader would have to `fsync` the new log entry immediately after appending it to the log, imposing an unacceptable performance cost.

To achieve practical performance, implementations instead overlap protocol execution with disk persistence. Developers must then decide manually when persistence must complete: before committing a log entry, applying it to application state, responding to another node, or replying to a client. Each decision must allow enough concurrency for good performance while still preserving the protocol's safety properties.

In general, completing persistence too early forces unnecessary disk synchronization onto critical paths and sacrifices performance. Completing it too late allows a node to expose state that cannot be recovered after a crash, violating the protocol's safety. The challenge is therefore to derive a persistence policy that preserves safety while allowing protocol execution to overlap with persistence whenever possible.

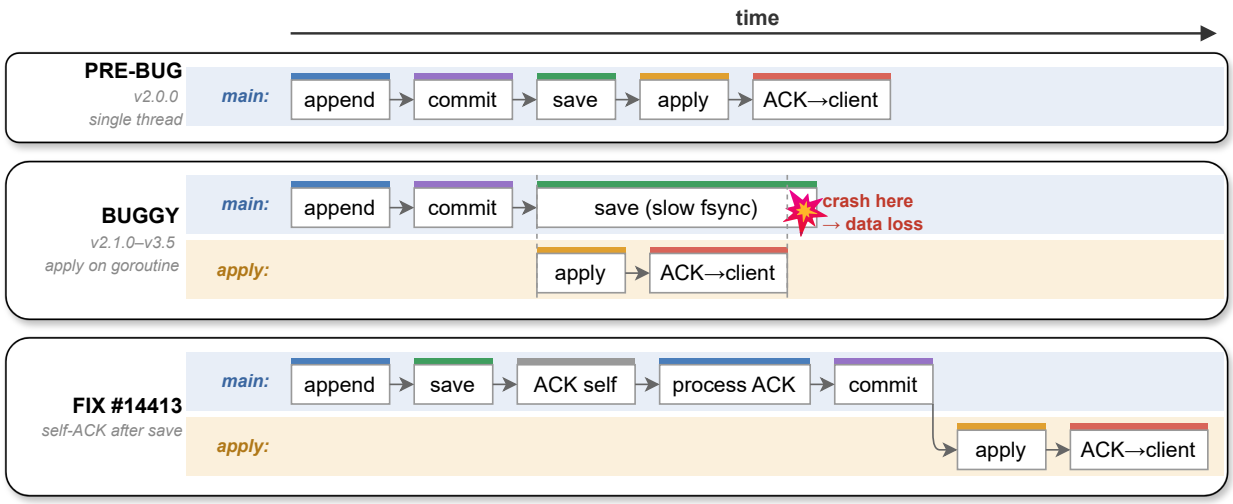


Figure 3.2: Etcd's persistence optimization moved protocol execution concurrently with disk persistence and crossed the safety boundary. In v2.0.0, the single-threaded Ready loop persisted an entry before applying it and replying to the client. Starting in v2.1.0, apply and client reply could proceed concurrently with the ongoing save. In a single-voter cluster, a crash after the client reply but before the save completed could therefore lose an acknowledged entry. PR #14413 restored the required behavior by processing the leader's self-acknowledgment only after the entry became durable.

3.1.1 Case Study I: Persisting Too Late

Figure 3.2 shows how a seemingly reasonable disk persistence performance optimization can violate crash consistency.

To reduce request latency, etcd delayed persistence so that the leader can apply and acknowledge a newly replicated log entry without waiting for the entry to be durable. This optimization removed a disk synchronization from the critical path, improving the common-case performance.

However, it also introduced a crash window. In a single-voter cluster, where the leader's own vote is sufficient to commit an entry, the leader could reply to the client before persisting the corresponding log entry. If the leader crashed after replying but before the disk write completed, the entry would be lost after recovery even though the client had already observed a successful operation, violating linearizability.

This optimization remained in production for more than seven years, until it was fixed in etcd v3.5 [36]. The eventual fix restored the required behavior by ensuring that the leader processes

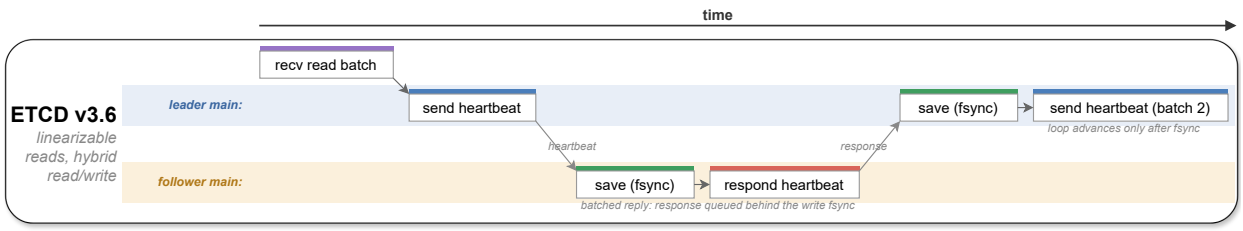


Figure 3.3: Conservative persistence causes disk-free linearizable reads to wait behind fsyncs under a hybrid read/write workload. Etcd processes Raft events in batches. At the follower, a batch containing both write updates and a heartbeat must persist the writes before responding to the heartbeat, delaying the leader's read quorum. At the leader, the Raft loop also waits for its pending fsync before advancing and releasing the waiting reads. Consequently, a linearizable read that performs no durable write can wait behind fsyncs at both the follower and the leader.

its own acknowledgment only after the corresponding log entry becomes durable.

The challenge is that neither ordering is obviously correct from the implementation alone. Determining whether the implementation is correct requires reasoning about protocol semantics and subtle implementation details.

3.1.2 Case Study II: Persisting Too Early

Figure 3.3 shows the opposite problem: conservative persistence can place unnecessary disk synchronization on the critical path.

In etcd, linearizable reads do not modify durable state. To serve such a read, the leader only needs to confirm that it still holds leadership, which it does by sending heartbeat messages to followers. However, under a hybrid read/write workload, heartbeat and regular AppendEntry responses are processed together in the same Raft loop. A follower that receives a batch containing both write-related Raft updates and heartbeat messages must first persist the write updates before responding. Thus, even though the read itself requires no disk write, its heartbeat response is delayed behind the follower's fsync.

The leader then experiences the same effect. Its Raft loop also advances in batches and waits for its own pending disk synchronization before processing the heartbeat responses and releasing

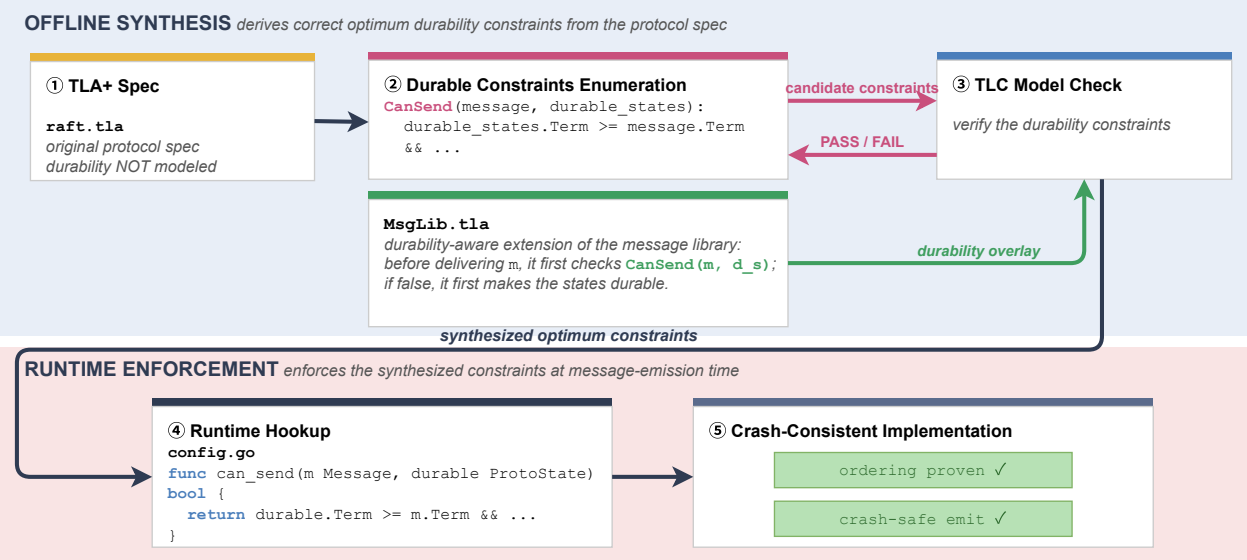


Figure 3.4: PostMark's two-phase workflow. *Durability synthesis* (top) takes the protocol's existing TLA+ specification (e.g. `raft.tla`, which treats durability as instantaneous and free), enumerates candidate *CanSend* predicates that order outbound messages against the durable state, and uses TLC to verify each candidate. To do this, we need to modify the original specification to model the memory states crash and durability operations. This is achieved through `MsgLib.tla`, a durability-aware extension of the message library that checks each message delivery by *CanSend* and persists state when the predicate fails. The PASS/FAIL feedback loop converges to a local-minimum sufficient predicate. *Runtime enforcement* (bottom) plugs the synthesized predicate into the implementation's outbound-message path, yielding a crash-consistent system without the developer writing any persistence-ordering code.

the waiting reads. As a result, a disk-free linearizable read can wait behind fsyncs on both sides: once at the follower before the heartbeat response is sent, and again at the leader before the read is completed.

In our experiments on a five-node cluster running a 50/50 YCSB-A workload, linearizable reads average 17 ms even though the writes they wait behind average only 11 ms. The protocol permits these reads to complete without waiting for persistence, but the implementation's conservative batching and synchronization policy forces them to pay the cost of unrelated persistence.

The difficulty mirrors the first case. The implementation cannot tell which messages actually depend on the pending write, so it conservatively delays all of them. Recovering the lost performance again requires reasoning about which protocol messages a crash could invalidate.

3.1.3 Overview of PostMark 's Approach

The challenge. Finding a safe and efficient policy between these extremes is difficult. Consensus implementations contain many operations: commit, apply, client replies, append responses, heartbeats, and more. Changing when any of them executes relative to persistence can have protocol-wide consequences, and whether a particular change is safe depends on protocol invariants and message semantics that the implementation itself never states. Testing does not reliably expose a wrong choice either: the etcd bug above survived seven years of production use because it requires a crash inside a narrow window on a single-voter cluster. Today, developers reason about these choices manually, one optimization at a time.

Our approach. PostMark replaces this manual search with automatic synthesis. Our key idea is to express the persistence policy as a per-message predicate, $CanSend(m, d_s)$, which determines whether protocol execution may emit message m given the current durable state d_s. Offline, PostMark derives and verifies this predicate from the protocol's TLA$^+$ specification under arbitrary crash timings. At runtime, PostMark enforces the synthesized predicate at message emission, allowing protocol execution to proceed concurrently with persistence whenever safety permits and waiting for persistence only when required.

Figure 3.4 summarizes this workflow. Section 3.2 describes how PostMark models, verifies, and synthesizes *CanSend*; section 3.3 describes how the runtime enforces the synthesized predicate in a concrete implementation.

3.2 Durability Synthesis

This section describes how PostMark derives correct and efficient orderings of state persistence and normal protocol execution. Section 3.2.1 defines the ordering model, which reduces the complex decision of when to complete persistance to a single per-message predicate, *CanSend*.

Section 3.2.2 builds a verifier that decides whether a given *CanSend* preserves the protocol's safety under crashes. Section 3.2.3 then uses this verifier to synthesize the minimal *CanSend* from the protocol's specification.

3.2.1 The Ordering Model: The Node Boundary and CanSend

The first challenge is to represent the persistence-timing decisions of section 3.1 in a form that PostMark can systematically search. A naive representation would order every in-memory state transition against every persistence operation. This produces a large and implementation-specific search space.

PostMark's central observation is that only one class of events must be strictly ordered against disk writes: the moment a piece of state crosses the node boundary and becomes visible to other nodes or clients. State changes that remain private to a node do not immediately need to be persisted. If the node crashes before those changes affect any message or client response, it can simply recover from an earlier durable state: no other node or client has observed the lost changes. Durability becomes necessary when the node is about to expose information to the outside world.

We therefore place the durability decision at message emission. Consider a node about to emit message m. Let s denote its current in-memory protocol state and let d_s denote the state captured by the latest completed persistence operation. Because persistence may execute concurrently with protocol processing, d_s may lag behind s.

The question is whether d_s already contains enough protocol progress for message m to be emitted safely. We represent this decision with

$$CanSend(m, d_s).$$

If $CanSend(m, d_s)$ is true, the node may emit m without waiting for the current in-memory

state to become durable. Otherwise, the node must persist its state before emitting m.

This predicate gives a concrete representation of a persistence policy. The maximally conservative policy sets *CanSend* to false for every message and therefore waits for persistence before every emission. At the opposite extreme, setting *CanSend* to true for every message allows protocol execution to proceed without ever waiting for persistence. Predicates between these extremes selectively place durability barriers only before messages that require them.

Raft serves as an example to illustrate why this decision is inherently message-specific. In Raft, a leader may send an `AppendEntriesRequest` containing a newly appended entry before persisting the entry locally. In contrast, a follower's `AppendEntriesResponse` acknowledges that it has accepted entries through `matchIndex`; allowing this response to contribute to a quorum before those entries are durable can violate crash safety. The same protocol state therefore imposes different durability requirements on different messages.

Our model assumes that each protocol transition first updates in-memory state and then emits its messages. Saves complete in transition order, and recovery restores the latest durable state. Section 3.3 describes how the runtime provides these execution properties.

The remaining problem is to determine which *CanSend* predicates are correct and how far a correct predicate can be relaxed without violating safety.

```
\* === STATE ===
VARIABLE log, ...                \* volatile protocol state
VARIABLE durable_log, ...   \* durable shadow added by PostMark
\* === STATE TRANSITIONS ===
HandleAppendEntriesRequest(i, j, m) ==
    /\ log[i]' = ...
    /\ LET resp ==
            [type       |-> AppendEntriesResponse,
            matchIndex |-> ...]
       IN  Lib!Send(i, resp)
```

Figure 3.5: Raft's AppendEntriesRequest handler with the durability overlay. PostMark adds the highlighted durable shadow and replaces Send(resp) with the gated Lib!Send(i, resp).

```
\* Phase 1: park an outbound message.
Send(i, m) ==
    /\ pendingSend = {}
    /\ pendingSend' = {[sender |-> i, msg |-> m]}
\* Phase 2: gate and deliver the parked message.
ProcessPending ==
    /\ pendingSend /= {}
    /\ LET pkt  == CHOOSE x \in pendingSend : TRUE
       IN  Emit(pkt.sender, pkt.msg)
    /\ pendingSend' = pendingSend \ {pkt}
Emit(i, m) ==
    /\ LET durable_state == [log |-> durable_log[i], ...]
       IN IF CanSend(m, durable_state)
          THEN UNCHANGED <<durable_log, ...>>
          ELSE Save(i)
    /\ messages' = messages \cup {m}
Save(i) ==
    /\ durable_log[i]' = log[i]
    /\ ...
\* Synthesized ordering constraint for Raft responses.
CanSend(message, durable_state) ==
    /\ message.type = AppendEntriesResponse
    /\ Len(durable_state.log) >= message.matchIndex
```

Figure 3.6: MsgLib's two-phase gated send. Send parks an envelope; ProcessPending evaluates *CanSend* against the current durable state and persists first only when the predicate requires it.

3.2.2 A Correctness Oracle for CanSend

Synthesizing *CanSend* requires a correctness oracle: given a candidate predicate, PostMark must decide whether enforcing it preserves the protocol's safety properties under crashes. We construct this oracle by adding a durability overlay to the protocol's existing TLA$^+$ specification and checking the resulting model with TLC.

Modeling volatile and durable state. The original protocol specification abstracts away persistence timing by treating durability as instantaneous and free. As shown in Figure 3.5, PostMark keeps the original protocol variables as in-memory state and introduces a durable shadow for each persistent variable. For example, Raft's `log` is paired with `durable_log`.

A `Save` transition copies the current in-memory state to its durable shadow. On a crash and restart, the in-memory state is restored from the durable shadow. Thus, a crash discards all protocol progress made after the latest completed `Save`.

Importantly, the original protocol transitions remain unchanged. The only modification to the message path is that an immediate `Send` is replaced with `Lib!Send`, which routes the message through PostMark's durability-aware message library.

Gating message emission. Figure 3.6 shows the relevant MsgLib transitions. `Send` first parks an outbound message in `pendingSend`. Before the protocol can continue, `ProcessPending` passes the message to `Emit`, which evaluates the candidate $CanSend(m, d_s)$ against the sender's current durable state.

If $CanSend(m, d_s)$ holds, `Emit` delivers the message without changing the durable state. Otherwise, the model executes `Save` before delivering the message. The candidate predicate therefore specifies exactly which message emissions are ordered after persistence.

When *CanSend* holds, our model deliberately leaves d_s unchanged. A real implementation may complete a background save before the node actually sends the message, but safety cannot

```
fn explore_conditions(spec, Newer, ProtoState) {
    let types = infer_types(spec, ProtoState);
    for m_f in msg_fields(types) {
        for s_f in state_fields(Newer) {
            if same_set(types, m_f, s_f) {
                cand[m_f] &= substitute(Newer, s_f -> m_f);
            }
        }
    }
    for m_f in msg_fields(types) {
        if pass(spec, cand - (drop + cand[m_f])) {
            drop = drop + cand[m_f];
        }
    }
    return cand - drop;
}
```

Algorithm 1: Synthesizing *CanSend* by matching message fields to protocol state, instantiating *Newer*, and using TLC to remove unnecessary field constraints.

rely on this opportunistic timing. Modeling the durable state as maximally stale forces TLC to explore the worst case in which asynchronous persistence never catches up unless the candidate predicate requires it.

Checking a candidate. We run TLC on the augmented specification using the protocol's original safety invariants. If TLC finds a counterexample, the candidate permits some message to escape with insufficient durable state and is rejected. If the model satisfies the invariants, the candidate is sufficient under our crash and persistence model.

The resulting PASS/FAIL procedure serves as the correctness oracle used by the synthesis algorithm described next.

3.2.3 Synthesizing CanSend

The correctness oracle can test a candidate *CanSend*, but the space of arbitrary predicates is unbounded. The synthesis problem is therefore to construct a small family of meaningful durability constraints and search that family using the oracle from Section 3.2.2.

PostMark starts from a conservative candidate derived from $Newer(s_1, s_2)$, a user-provided

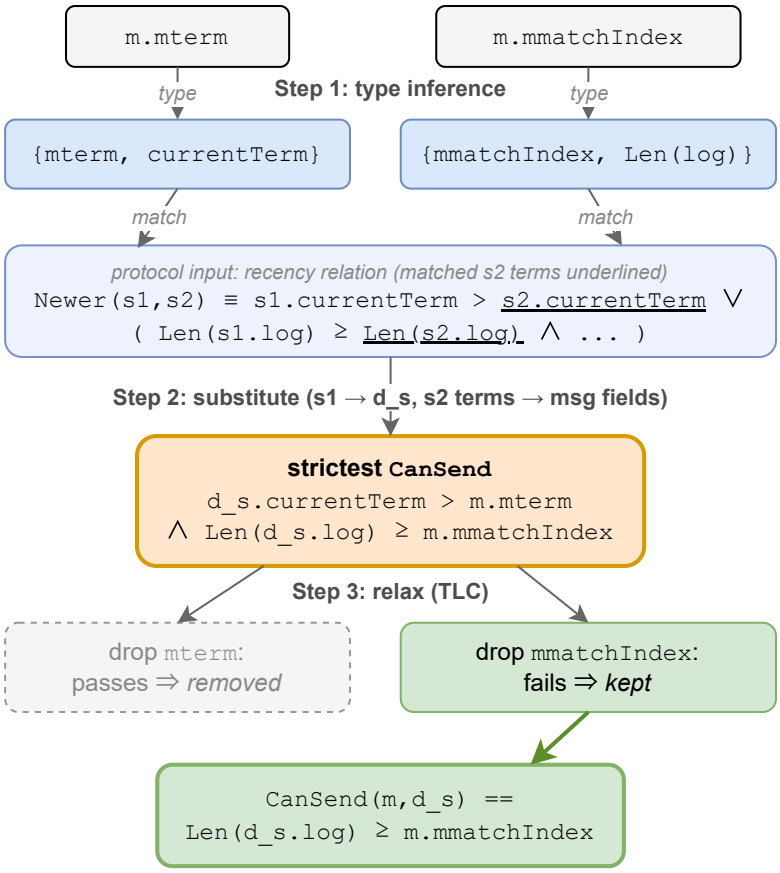


Figure 3.7: Synthesizing *CanSend* for Raft's `AppendEntriesResponse`. Type inference matches message fields to protocol state, and substitution constructs a conservative predicate. If the specification passes TLC after the `mterm` constraint is removed, synthesis removes that constraint; if removing the `mmatchIndex` constraint produces a counterexample, synthesis retains it.

relation that describes when state s_1 is at least as advanced as state s_2, and then removes constraints that TLC proves unnecessary. Algorithm 1 summarizes the procedure. Figure 3.7 walks through the three steps for Raft's `AppendEntriesResponse`.

Step 1: Match message fields to protocol state. A durability constraint must relate information carried by a message to the corresponding protocol state on disk. PostMark uses `infer_types(spec, ProtoState)` to group message fields with compatible protocol-state expressions.

For Raft's `AppendEntriesResponse`, this analysis matches the message field `mterm` with

`currentTerm` and `mmatchIndex` with $Len(log)$. These correspondences identify the state expressions from which candidate constraints can be constructed.

Step 2: Construct a conservative candidate. PostMark next instantiates the protocol's recency relation, $Newer(s_1, s_2)$, which describes when one protocol state is at least as advanced as another. For each message–state correspondence from Step 1, PostMark substitutes the matched message field for the corresponding term of s_2 and interprets s_1 as the current durable state d_s.

Conjoining the resulting conditions yields the most conservative predicate. For `AppendEntriesResponse`, the initial candidate is

$$d_s.currentTerm > m.mterm \ \wedge \ Len(d_s.log) \geqslant m.mmatchIndex.$$

This candidate requires the durable state to cover every message field identified in Step 1. It is intentionally conservative; some of these conditions may not be necessary for the protocol's safety invariants.

Step 3: Remove unnecessary constraints. PostMark uses the verifier from Section 3.2.2 to relax the candidate at message-field granularity. For each message field, the synthesis algorithm temporarily removes the conditions derived from that field and invokes TLC. If the resulting predicate still preserves the protocol's safety invariants, the conditions are permanently removed. Otherwise, they are restored.

For Raft, if removing the condition associated with `mterm` passes TLC but removing the condition associated with `mmatchIndex` produces a counterexample, the synthesized predicate is

$$CanSend(m, d_s) \equiv Len(d_s.log) \geqslant m.mmatchIndex.$$

The result is locally minimal with respect to the generated message-field constraints: removing any remaining field's conditions causes verification to fail. For `AppendEntriesResponse`, the

synthesized predicate captures the asymmetry discussed in Section 3.2.1. A leader may transmit a new entry before saving its local copy, whereas a follower must make the acknowledged log prefix durable before its response may contribute to protocol progress.

Validating the initial candidates. The simplified algorithm above assumes that the initial predicates derived from *Newer* are safe. This assumption need not always hold: for some message type, even the conjunction of all generated field conditions may still violate a safety invariant. The implementation therefore initializes *CanSend* to false for every message type and walks a protocol-specific priority queue of message-field constraints. When the queue first reaches a message type, PostMark temporarily replaces that type's false branch with the conjunction of all its *Newer*-derived field conditions and invokes TLC. We call this replacement *materializing* the message type. Predicates already accepted for earlier types remain active, while types not yet reached remain false. If TLC rejects the replacement, the type remains false and its remaining field trials are skipped. If TLC accepts it, PostMark retains the full predicate and performs the field-removal trials described above, carrying each accepted removal into subsequent trials. Thus, the implementation validates and admits strict candidates one at a time rather than materializing all message types together. The explicit queue also makes the order of this greedy search deterministic and reviewable.

3.2.4 Bounding the Verification State Space

Adding durability to a protocol model compounds its state-space explosion. In addition to the original protocol states and message schedules, each server has a durable copy of its in-memory state, persistence may lag behind protocol execution, and a crash may roll the server back at many points. Because synthesis invokes TLC repeatedly, once for each candidate predicate, exhaustive exploration quickly exceeds a practical time budget.

To reveal such bugs in distributed protocols, TLC usually needs to explore multiple rounds

of communication between servers. We therefore constrain the search to prioritize exploring the execution at greater depths to expose crash–persistence violations within a fixed time, in three ways. First, we bound protocol behavior itself, limiting terms, workloads, and failures. Second, we reduce and direct the search, exploiting server symmetry and seeding exploration from several starting states. Third, we bound the *network state* by limiting both the messages in flight and repeated emissions.

3.2.4.1 Bounding protocol behavior

Bounding protocol execution. We check one-server and three-server configurations. The one-server model covers behavior that requires no replica communication, while the three-server model covers quorum behavior.

Each configuration allows at most three client requests in total and at most two requests per server. Requests carry unique values so TLC can detect lost or reordered entries without exploring repeated values. We also bound the number of terms. These settings still allow replication, commitment, leader changes, crashes, and incomplete saves to interact.

Bounding failures. A single well-timed crash is enough to expose the violations that PostMark targets. We therefore allow each server to crash and restart at most once. We allow at most two election timeouts per server and three in total, so the model can replace a failed leader and continue. A crash only threatens durability once a server has sent state it has not yet persisted, so we only enable crashes immediately after a server has sent state that has not yet been persisted.

3.2.4.2 Reducing and directing the search

Exploiting server symmetry. Renaming servers does not change protocol behavior. We therefore declare the servers as a TLC symmetry set, which merges states that differ only in server identities. We express our bounds over server-indexed counters rather than over particular servers,

so they remain compatible with this reduction.

Starting from different states. Exploration from the initial state of the protocol spends its first levels on a prefix that every execution shares, such as electing the first leader. We therefore also start exploration from canonical mid-execution states, for example a cluster that has already elected a leader, so that the search reaches deeper protocol behavior within the same budget. We quantify these states over the servers, so the set of starting states stays compatible with symmetry reduction.

3.2.4.3 Bounding network state

Buffered messages remain a major source of states. To reduce the state space while continuing to explore different executions, we bound the following aspects of the network state while preserving (1) messages between different servers and (2) different types of messages between servers.

Bounding in-flight messages. We allow at most one in-flight message per sender–receiver pair. Messages to different receivers may coexist, so a leader may still send one message to each follower. This preserves the important send–crash–deliver ordering while removing backlogs to one receiver. It cannot find a violation that requires several messages to be queued for the same receiver.

Bounding message multiplicity. Retransmissions also create many similar states. `MsgLib` records, for each sender–receiver pair, the messages that pair has already emitted, and applies two budgets. A pair may emit at most two *distinct* messages of each type, which bounds how many different versions of a message a sender may produce. Each individual message may be emitted at most twice, an original and one retransmission. Because a new send may not start

while an earlier one is still in flight, the two copies never coexist, and duplication is modeled as sequential retransmission by the sender.

Scope of the bounds. The bounds above are heuristics for finding counterexamples, not completeness-preserving reductions. In our experiments we check each candidate under a fixed time window. We reject a candidate when TLC reports a counterexample, and accept it when TLC either exhausts the bounded state space or consumes the whole window without finding one. The bounds therefore matter because they let TLC explore more steps within the same window. An accepted candidate is safe only within the bounded model, and only up to the steps TLC explored, so a deeper execution or one outside the bounds could still violate the property. Section 3.4.3 measures the additional depth enabled by the network bounds.

3.3 Runtime Enforcement

Section 3.2 derives a $CanSend(m, d_s)$ predicate that specifies when an outbound message may be emitted given the current durable state. This section describes how PostMark enforces the synthesized predicate in a running protocol implementation.

PostMark exposes two operations to the protocol: `EnqueueTask` and `GateMessage`. Figure 3.8 shows the canonical flow. After a protocol transition updates its in-memory state, it submits the resulting state change through `EnqueueTask`. Each outbound message then passes through `GateMessage` before being sent. Persistence can therefore proceed concurrently with protocol execution, while message emission waits only when the synthesized ordering requires it.

```
/* ---- user-defined (no ordering logic) ---- */
struct ProtoState;   // durable state (d_s)
struct ProtoUpdate;  // one step's delta to it
void       save (ProtoUpdate u);   // crash-atomic
ProtoUpdate batch(ProtoUpdate u1, ProtoUpdate u2);
ProtoState  apply(ProtoState d_s, ProtoUpdate u);

/* ---- library API ---- */
void EnqueueTask(ProtoUpdate u); // non-blocking
bool GateMessage(Message m, bool blocking);

/* ---- sync thread (provided by PostMark) ---- */
while (true) {
    // enqueued updates, folded via batch()
    u = next_pending_update();
    save(u);               // one fsync per batch
    d_s = apply(d_s, u);   // new durable state
}

/* ---- canonical RPC handler ---- */
void on_rpc(Request req) {
    u, msgs = apply_in_memory(req); // memory only
    EnqueueTask(u);
    for (Message m : msgs) {
        GateMessage(m, /*blocking=*/true);
        send(m);
    }
}
```

Figure 3.8: PostMark's library interface. The user supplies two types and three functions (save, batch, apply); none carries ordering semantics. The library owns the sync thread and gates every outbound message on the synthesized can_send.

Asynchronous persistence. `EnqueueTask` accepts a `ProtoUpdate` and returns without waiting for disk persistence. A library-owned sync thread processes the enqueued updates and maintains the latest durable protocol state d_s.

The developer defines two state representations, `ProtoState` and `ProtoUpdate`. `ProtoState` represents the durable protocol state needed to evaluate *CanSend*, whereas `ProtoUpdate` represents the changes produced by one protocol transition. We introduce a separate update representation because the persistence interfaces of the systems we study are naturally incremental. Their replicated logs are append based: a protocol transition typically passes newly appended entries and changed metadata to the storage layer rather than the entire protocol state. For example, a Raft implementation persists newly appended log entries instead of rewriting the complete log. `ProtoUpdate` allows PostMark to preserve this existing persistence interface.

Persisting every enqueued `ProtoUpdate` independently would require a separate write and disk synchronization for each protocol transition, which is impractical for high-throughput systems. Existing implementations instead batch multiple updates before persistence. Following this design, PostMark exposes a user-defined `batch(u_1,u_2)` function that combines pending updates.

The `sync` thread repeatedly fetches a pending `ProtoUpdate` and writes it to disk with `save`. While the sync thread is busy persisting an update, `EnqueueTask` continues to accept new updates and combines them into a single pending update using `batch`. Once the current save completes, the sync thread fetches the batched pending update, clears it, and persists it next. Thus, each persistence operation can drain all updates accumulated during the previous save.

After `save(u)` completes, PostMark invokes `apply(d_s,u)` to incorporate the persisted update into the previous durable state. The resulting `ProtoState` becomes the new d_s. PostMark atomically publishes this state and wakes messages waiting for persistence to advance.

Because `ProtoState` and `apply` are user-defined, an integration may compact or garbage-collect state while advancing d_s. For example, a Raft integration may discard obsolete log entries

while retaining the durable progress needed by *CanSend*. Thus, PostMark need not keep the complete protocol log or the history of persisted updates in memory.

Crash-atomic saves and recovery. Our model treats each `Save` as atomic: after a crash, either the complete save is visible or none of it is. We provide this property by adapting each system's native storage and recovery path. For etcd, RedisRaft, and ZooKeeper, we add a save-completion record to the original persistence path. Each system appends this record after the records in an update and synchronizes the log once. Recovery exposes only updates terminated by this record and discards an incomplete tail. These mechanisms implement the atomic `Save` operation assumed in Section 3.2.2; Section 3.4.2 describes the required implementation changes.

Gating message emission. `GateMessage` enforces the synthesized *CanSend* predicate at the outbound-message boundary. Before sending message m, PostMark evaluates $\mathit{CanSend}(m, d_s)$ against the latest published durable state.

If the predicate holds, the message can be emitted immediately. Otherwise, the message waits for persistence to advance. The sync thread continues persisting pending updates. After a save completes and d_s is updated, PostMark checks the gated message again and releases it once $\mathit{CanSend}(m, d_s)$ holds.

A predicate may be stricter than the progress of d_s and never become true; the conservative case $\mathit{CanSend} \equiv \text{false}$ is one example. In the TLA$^+$ model, such a message is sent after executing `Save`. PostMark mimics this behavior by assigning each save a monotonically increasing sequence number. When a message is gated, PostMark records the save that will cover its corresponding `ProtoUpdate`. Once that save, or a later one, completes, the message is released regardless of *CanSend*. This preserves the save-before-send ordering modeled in Section 3.2.2 and prevents a gated message from waiting indefinitely.

`GateMessage` provides both blocking and non-blocking interfaces. The blocking form parks the caller until the message can proceed. The non-blocking form returns immediately when the

message is gated, allowing the protocol to buffer it and continue execution. Our etcd integration uses the non-blocking form because its Raft loop runs in a single goroutine that must continue processing protocol events rather than block on one outbound message. Rejected messages are buffered and reconsidered after the sync thread publishes a new durable state.

Together, these mechanisms provide the execution properties assumed in Section 3.2.1. The handler pattern in Figure 3.8 submits a state update before gating the messages produced by that transition. The single sync thread preserves the order of enqueued protocol updates, even when adjacent updates are batched, and the system-specific atomic-save mechanism ensures that recovery restores state through the latest completed save.

Integrating PostMark. Integrating PostMark requires changes at two points in the protocol implementation. First, the protocol submits persistence updates through `EnqueueTask` instead of waiting for them to become durable on the protocol path. The protocol continues to update its in-memory state in its original order, while the sync thread batches and persists the resulting updates asynchronously.

Second, the implementation inserts `GateMessage` at the outbound-message boundary. Every protocol message passes through the gate before leaving the node. The synthesized *CanSend* predicate therefore determines which messages must wait for persistence without scattering durability-ordering decisions throughout the protocol implementation.

Synthesis produces *CanSend* as a TLA$^+$ predicate; PostMark does not generate target-language code. The developer directly transcribes this predicate over the implementation's message and durable-state types. This step maps the synthesized state and message fields into Go, C, or Java, but does not require the developer to design a new durability policy.

These two hooks directly enforce the persistence policy synthesized in Section 3.2. If a message requires newer durable state, `GateMessage` holds it until persistence catches up. If its predicate already holds, the message proceeds even while unrelated persistence is in progress. The

former prevents the unsafe message emission illustrated in Figure 3.2; the latter allows messages on the linearizable-read path in Figure 3.3 to avoid waiting for unrelated write persistence.

Section 3.4 quantifies the effort required to integrate PostMark and the performance of the resulting durability orderings.

User inputs and correctness scope. PostMark requires two classes of user input. For synthesis, the user provides the protocol's TLA$^+$ specification and a state progress relation $Newer(s_1, s_2)$ stating when s_1 is at least as advanced as s_2, together with a priority queue that orders the message-field constraints that PostMark attempts to remove. For runtime integration, the user defines `ProtoState` and `ProtoUpdate` and implements `save`, `batch`, and `apply`.

PostMark trusts but does not verify the runtime obligations: `save` must be crash-atomic, and `batch` and `apply` must compose updates consistently with the protocol's transitions (Section 3.3). A mis-specified *Newer* cannot yield an unsafe result: if even the strictest instantiated candidate fails model checking, synthesis rejects it and falls back to the always-safe $CanSend \equiv false$.

Correctness scope. Synthesis runs TLC under the protocol and network bounds in Section 3.2.4. These bounds make deeper exploration practical, but may exclude an execution that exposes an unsafe predicate. A candidate that passes is therefore verified only within the checked model, not proven safe for all executions. The model retains replication, commitment, leader changes, and crashes during an incomplete save; Section 3.4.3 isolates the effect of network bounding on TLC's exploration depth.

3.4 Evaluation

We evaluate PostMark on **etcd** (Raft, Go), **RedisRaft** (Raft, C), and **ZooKeeper** (Zab, Java). We ask three questions:

- **RQ1:** How does PostMark affect throughput and latency? (Section 3.4.1)

System	Config	Tput (kops/s)	All ops (ms)	Read ops (ms)			Write ops (ms)		
			avg	avg	p95	p99	avg	p95	p99
etcd	Baseline	35.2	14.2	17.1	23.0	26.8	11.3	16.9	20.8
	Conservative	33.3	15.0	15.7	22.5	26.7	14.3	21.8	26.9
	PostMark	56.8	8.8	6.3	14.9	21.4	11.3	18.3	24.5
RedisRaft	Baseline	48.1	10.4	10.3	11.8	12.6	10.4	11.9	12.6
	Conservative	19.6	25.5	24.3	52.9	80.0	26.6	54.9	82.0
	PostMark	48.9	10.2	10.2	12.3	15.9	10.3	12.4	16.2
ZooKeeper	Baseline	33.5	14.9	9.7	11.1	32.6	20.0	23.2	49.2
	Conservative	5.5	90.3	59.1	72.6	82.7	121.4	139.9	159.8
	PostMark	33.2	15.0	9.5	12.3	32.7	20.5	24.4	49.1

Table 3.1: Throughput and latency of the three configurations. Latency is reported as average over all operations, and as average, p95, and p99 separately for read and write operations.

- **RQ2:** How much developer effort is required to integrate PostMark? (Section 3.4.2)
- **RQ3:** How effectively does network bounding improve TLA$^+$ exploration? (Section 3.4.3)

Experimental setup. We use five AWS m5.4xlarge servers and one client, each with an EBS gp3 volume. We preload one million records and run YCSB Workload A (50% reads and 50% updates) with 500 closed-loop clients, a uniform key distribution, and all operations, including reads, directed to the leader. *Baseline* is the upstream system, and *PostMark* uses the synthesized *CanSend* predicate. *Conservative* is otherwise identical to PostMark, but sets *CanSend* ≡ false so every message waits for its update to be saved.

3.4.1 End-to-End Performance (RQ1)

Table 3.1 reports the results. For etcd, PostMark reduces average read latency by 63%. Read p99 decreases from 26.8 to 21.4 ms, while write p99 increases slightly from 20.8 to 24.5 ms. This latency reduction comes from addressing the read-path problem in Figure 3.3: read-quorum messages bypass unrelated writes, eliminating the unnecessary persistence wait. For RedisRaft, Post-

Mark matches the baseline, while read and write p99 rise from 12.6 to 15.9 and 16.2 ms, respectively. For etcd and RedisRaft, Conservative shows the cost of waiting for persistence before every message: throughput drops by 5% and 59%, while average latency increases by 6% and 145%, respectively. For ZooKeeper, PostMark also matches the baseline. Read p99 changes from 32.6 to 32.7 ms, while write p99 changes from 49.2 to 49.1 ms.

Overall, PostMark can reduce read latency when an existing implementation makes reads wait unnecessarily for persistence. When the existing implementation already uses substantial manual effort to tune concurrency, PostMark remains competitive while deriving the required durability ordering systematically.

System	`Newer`	`save`	`batch`	`apply`	*CanSend*	Total
etcd	7	77	0	18	29	131
RedisRaft	7	117	61	147	35	367
ZooKeeper	7	124	6	7	52	196

Table 3.2: Source lines for the PostMark library API and *CanSend*, plus the `Newer` relation (TLA$^+$) each protocol supplies to the synthesizer. The two Raft variants share one `Newer`.

3.4.2 Integration Effort (RQ2)

Table 3.2 reports the code required to connect each system to the PostMark library: the `Newer` relation supplied to the synthesizer, then `save`, `batch`, `apply`, and *CanSend*. Although PostMark automatically explores *CanSend* in TLA$^+$, we require developers to translate the resulting predicate into the target language.

The library integration requires 131 SLOC for etcd, 367 for RedisRaft, and 196 for ZooKeeper. We count every function we add or modify along each path, including the changes to the upstream baseline that make a save crash-atomic. The `Newer` relation is 7 lines of TLA$^+$ per protocol, the only specification input the developer writes for synthesis. The etcd integration needs no `batch` code because `Ready` already includes the accumulated updates. RedisRaft requires more code to merge incremental log operations and manage their memory in C, while ZooKeeper must handle transaction and epoch state and more Zab message types.

Beyond this code, we augment the original save and recovery logic in all three systems to make each save atomic. RedisRaft requires an additional storage change. Its original runtime storage serves log reads from a cache and falls back to disk when necessary. We replace it with an etcd-like design that maintains an in-memory log for all runtime reads, while the durable storage layer performs writes asynchronously. However, we find no cache misses in the original implementation during our performance experiments. Thus, both the original and our implementations serve all runtime reads from memory, and this storage change does not account for their performance difference.

	With network bounds		Without network bounds	
Protocol	Depth	States	Depth	States
etcd-raft	41	371M	30	836M
RedisRaft	40	331M	30	802M
ZooKeeper	131*	23.9M	89	100M

Table 3.3: TLC exploration within one hour for the strictest candidate (*CanSend* = false). Depth is the greatest level TLC explored completely. Network bounds add 10–11 steps for the two Raft models while enumerating fewer states; for ZooKeeper (*) the bounded space is exhausted in 19 minutes, so all 131 steps are complete.

3.4.3 Effectiveness of Network Bounding (RQ3)

To isolate the effect of network bounding, we compare runs with and without the network bounds. All runs use the protocol bounds of Section 3.2.4.1 and the search reductions of Section 3.2.4.2; the bounded runs additionally use the network bounds of Section 3.2.4.3. TLC checks the strictest candidate, which sets *CanSend* to false. We give each run the same one-hour budget and report the greatest logical depth for which TLC fully explores the reachable state space.

Table 3.3 shows that network bounding increases the completely explored depth from 30 to 41 for etcd-Raft and from 30 to 40 for RedisRaft. For ZooKeeper the effect is qualitative: with the bounds, TLC *exhausts* the entire bounded state space, 23.9M distinct states across all 131 depth steps, in 19 minutes, turning every passing trial into a complete verification result rather than a timeout. Without the bounds, TLC explores 4.2× more states in the full hour yet completes only 89 steps, with millions of states still queued. The two message models fail differently without the bounds. In the Raft models, whose network is a message bag, TLC enumerates 2.2–2.4× *more* states yet reaches executions ten to eleven steps shorter: unbounded senders may keep several messages in flight to the same peer, and the budget is consumed by permuting their deliveries rather than by protocol progress. In ZooKeeper, whose network is per-channel FIFO queues, unbounded channels grow longer and make every successor computation more expensive, so exploration also slows per state. In both cases, the bounds let TLC spend its budget on deeper

protocol behaviors.

3.5 Summary

The project reduces persistence timing to the ordering between durable state and outbound messages. It represents this ordering with $CanSend(m, d_s)$, checks candidate predicates by adding durability and crash behavior to a protocol's TLA$^+$ specification, and removes constraints that TLC shows are unnecessary within the checked model. At runtime, asynchronous saves advance d_s, while message gating delays only the messages whose predicates require newer durable state.

The evaluation integrates this design with etcd, RedisRaft, and ZooKeeper. The resulting policies avoid both failure modes: exposing protocol progress before it is recoverable and forcing every message to wait for persistence. In etcd, our experiments show that allowing read-quorum messages to bypass unrelated writes reduces average read latency by 63%.

4 Conclusion

4.1 Summary

This dissertation has argued that a verified specification has more potential than current practice draws from it. In a conventional development workflow, the specification serves only as a reference: developers read it, write code that they believe corresponds to it, and then set it aside. This dissertation has shown that the specification can do considerably more. By putting it to work during and after implementation, verification need not end before implementation begins, and the correctness established for the specification can be carried forward to the system built from it.

It has demonstrated two ways to put a specification to work. Ellsberg uses the specification online: by observing only the messages a process sends and receives, it reports behavior that no execution of the protocol can explain. PostMark uses the specification offline: it extends the model to account for the loss of volatile state during crashes, searches for a durability policy that preserves safety without imposing unnecessary persistence, and enforces the resulting policy in the running system.

4.2 Revisiting Specifications and Implementations in the Era of Large Language Models

LLM-based coding agents can now perform repository-level software-engineering tasks and author changes in real projects. They have already seen wide adoption: The AIDev dataset identifies 932,791 agent-authored pull requests across 116,211 repositories [53]; and in April 2026, Google reported that 75% of all new code at the company was AI-generated and approved by engineers, up from 50% the previous fall [75].

The widespread adoption of AI changes software engineering in two important ways: First, developers express what the system should do, and coding agents produce candidate implementations. Second, developers and automated tools review those candidates to determine whether they realize the intended behavior. Code generation does not eliminate human responsibility; instead, it shifts more of that responsibility toward (1) articulating intent precisely and (2) validating the generated code. This shift is especially important when reviewers must assess code that they did not write.

We argue that specifications will play a more critical role in both activities. During generation, a specification can provide a precise, machine-checkable statement of intent that guides the coding agent. During review, the same specification can serve as an oracle against which the generated implementation is checked. The specification is thereby shared by developers, coding agents, and verification tools throughout the development process.

The two systems presented in this dissertation already instantiate these roles, although neither was designed with coding agents in mind. PostMark uses a specification to guide the construction of an implementation policy, while Ellsberg uses a specification to review an implementation's behavior at runtime. Neither approach depends on whether the implementation was written by a human or generated by an agent. Their specification-guided mechanisms therefore

remain applicable as coding agents assume a larger role in producing code.

4.2.1 Specifications as Intent

Natural language remains indispensable for describing goals, rationale, and context, but it does not have a fixed operational semantics. Durability provides a concrete example. A request such as "do not lose an acknowledged write" does not specify when updated state must become durable, and a coding agent must choose an ordering on its own. To review the code, developers must then reconstruct both the durability policy and its correctness argument from an implementation they did not write. The resulting code is therefore harder to reason about and validate.

A formal specification makes these choices explicit. It defines system state, the transitions that may change it, the assumptions made about the environment, and the properties that executions must preserve. The coverage gap is therefore even more consequential in an agentic workflow. The more precisely a specification captures relevant runtime scenarios, including crash recovery and durability, the more completely it constrains the implementation that a coding agent may generate. Conversely, each scenario omitted by the specification will leave the agent to make another correctness-critical implementation decision without a formal oracle, and thus make the resulting code harder to validate.

PostMark provides a complementary example from this dissertation. Rather than generating an entire implementation, it uses the verified protocol specification to derive a correct durability policy that avoids unnecessary persistence constraints. In an agentic workflow, this policy allows the specification to constrain or synthesize implementation decisions that an agent would otherwise have to infer from prose.

4.2.2 Specifications as Review

Agent-generated code changes the developer's position during implementation. When developers write code themselves, they gain familiarity with its invariants and design choices as they work. When an agent supplies a large change, reviewers begin with the finished code and must reconstruct that reasoning after the fact. Line-by-line review is still necessary, but it asks reviewers to infer global protocol behavior from local control flow, and its cost grows with the amount of generated code. This is a poor match for agents that can produce and revise code much faster than humans can inspect it.

A specification provides a complementary form of review. Rather than asking only whether each line appears reasonable, specification-guided review asks whether every relevant implementation behavior is allowed by the protocol and whether the required properties continue to hold. The check can range over concurrent executions and failure cases that are difficult to reason about in large volumes of generated code. Moreover, the oracle is independent of the particular program text: an agent may change data structures or reorganize control flow without changing the behavior that the specification permits.

Ellsberg illustrates the runtime endpoint of this design space. It checks observed messages against compatible specification executions and reports any implementation action that the protocol cannot explain. Because this judgment depends on observable behavior rather than on the source code's authorship or internal structure, Ellsberg can check agent-generated and human-written implementations in the same way. Its oracle continues to apply after deployment and remains useful across later agent-generated revisions.

4.2.3 Discussion: LLMs as Specification-Writing Assistants

If specifications become more important, their construction and maintenance may become the next bottleneck. Writing a useful specification requires both knowledge of a formal language

and judgment about which implementation details to abstract away. LLMs can reduce the mechanical part of this cost by drafting formal specifications from design documents, pseudocode, tests, and relevant source code. Developers must still review the generated specification to ensure that it captures their intent, but that review can focus on algorithmic logic rather than on implementation detail. A protocol specification can expose the algorithm's essential state and transitions while abstracting away concrete data structures, thread management, low-level synchronization, and performance optimizations. Developers can therefore reason about correctness at the algorithmic level instead of reconstructing invariants from optimized concurrent code. Formal languages such as TLA$^+$ provide an additional advantage: model checkers can mechanically explore the specification and determine whether it satisfies the explicitly stated safety properties. Human review establishes that the abstraction and properties express the intended contract; formal checking then verifies their consequences.

An LLM can also refine a specification using feedback from parsers, model checkers, tests, and implementation traces. One recent system, Specula, explores this approach at repository scale: it uses coding agents to construct TLA$^+$ models and invariants from system code and then iteratively refines them using model-checking and conformance feedback [15]. Such tools can detect syntax errors, property violations, and mismatches between a model and its implementation, but they cannot determine whether the generated specification captures what developers actually intend. Developers must therefore still review the specification's scope, assumptions, and critical properties.

4.2.4 Concluding Remarks

Taken together, these observations suggest a reciprocal relationship between formal methods and coding agents. Specifications can ground code generation and hold its output accountable to an explicit behavioral constraint; coding agents can in turn lower the cost of creating and maintaining that constraint. As agents take on more of the work of writing code, the central claim

of this dissertation becomes more, rather than less, important: a verified specification should not be left behind after design, but should remain an active part of building, reviewing, and operating the system.

A | Supplementary Material for Ellsberg

A.1 Proving apply ASAP correct

Protocol specifications in Ellsberg include an `apply_asap?` function (Section 2.3.3), whose output we use to avoid redundant exploration (Section 2.3.4.2). In this appendix, we look at proving the correctness of this function.

We start by redefining the `apply_asap?` requirements in terms of *dominating states*.

Dominating state: We say a state s dominates state s' (or $s \sqsupseteq s'$) if and only if for any message m, s' being m-inducing implies s is m-inducing.

The `apply_asap?` function takes as input state s and an event e, and returns true if the following conditions are met:

- For all schedules Σ where $e \in \Sigma$, $apply(apply(s, e), \Sigma - e) \sqsupseteq apply(s, \Sigma)$.
- For all schedules Σ where $e \notin \Sigma$, $apply(apply(s, e), \Sigma) \sqsupseteq apply(s, \Sigma)$.

We now turn to describing our approach for proving a supplied `apply_asap?` function meets this requirement.

In our specifications (and indeed in all specifications), an $apply\_asap?(s, e)$ consists of a set of constraints $C = \{c_0, c_1, \ldots, c_n\}$ over s and e, and returns true if any constraint c_i holds. For example, in Raft's *apply_asap*? function one of the constraints deals with append entry responses, and returns true if e is an incoming *AppendEntryResponse* message with term t and the simulation state s's term is larger than t ($s.term > t$).

We prove `apply_asap?` correct by analyzing each constraint $c \in C$. More formally

Theorem A.1. *An* `apply_asap?` *function specified as the set of constraints C is correct if and only if for all $c \in C, s \in S, e \in E$, $c(s, e) = true$ implies that the following two properties hold for s, e:*

- *For all schedules Σ where $e \in \Sigma$, $apply(apply(s, e), \Sigma - e) \sqsupseteq apply(s, \Sigma)$.*
- *For all schedules Σ where $e \notin \Sigma$, $apply(apply(s, e), \Sigma) \sqsupseteq apply(s, \Sigma)$.*

Proving these two conditions mechanically is challenging, as it requires reasoning about the set of all schedules Σ. Therefore, we instead prove that if four simpler properties (that can be mechanically checked) hold for any $c \in C$, then c satisfies Theorem A.1:

Condition A.1.1. ***Order Preserving.*** *The protocol should be such that for any event e, states s_1, s_2, if $s_1 \sqsupseteq s_2$, then $apply(s_1, e) \sqsupseteq apply(s_2, e)$.*

Condition A.1.2. ***Constraints Preserving.*** *For any event e, e', and any state s, if $c(s, e) = true$ then $c(apply(s, e'), e) = true$.*

Condition A.1.3. ***Expansion.*** *For any event e and any state s that satisfies $C(s, e)$, $apply(s, e) \sqsupseteq s$*

Condition A.1.4. ***Reorder Safety.*** *For any events e and e' and any state s that satisfies $C(s, e)$,*

$$apply(apply(s, e), e') \sqsupseteq apply(apply(s, e'), e).$$

Assume c satisfies the four conditions above, we use case splitting to prove that Theorem A.1 must hold:

Case 1 $e \notin \Sigma$. Given **Statement 3**, we have $apply(s, e) \sqsupseteq s$. Given **Statement 1**, we have $apply(apply(s, e), \Sigma) \sqsupseteq apply(s, \Sigma)$. ■

Case 2 $e \in \Sigma$. Suppose $\Sigma = \Sigma' + e + \Sigma''$ where $\Sigma' = e_1, e_2, ..., e_k$. To simplify the expressions, we denote $apply(s, e)$ by $e(s)$. We first prove that:

Lemma A.2.

$$e_k \circ ... \circ \underline{e_{i+1} \circ e} \circ e_i \circ ... \circ e_1(s) \sqsupseteq e_k \circ ... \circ \underline{e \circ e_{i+1}} \circ e_i \circ ... \circ e_1(s)$$

Let $s' = e_i \circ ... \circ e_1(s)$, according to **Statement 2**, given that $C(s, e)$ is satisfied, we have that $C(s', e)$ is also satisfied. Then, according to **Statement 4**, we have $e_{i+1}(e(s')) \sqsupseteq e(e_{i+1}(s'))$. Then, given **Statement 1**, we can prove **Lemma 1**.
Therefore, we can further prove this by inductively applying **Lemma 1**:

$$e_k \circ ... \circ e_2 \circ e_1 \circ e(s) \sqsupseteq e \circ e_k \circ ...e_2 \circ e_1(s)$$

Then, let $s_1 = e_k \circ ... \circ e_2 \circ e_1 \circ e(s)$ and $s_2 = e \circ e_k \circ ...e_2 \circ e_1(s)$, given **Statement 1**, we have $\Sigma''(s_1) \sqsupseteq \Sigma''(s_2)$, where $\Sigma''(s_1) = apply(apply(s, e), \Sigma - e), \Sigma''(s_2) = apply(s, \Sigma)$. ■

A.2 Mechanically Checking `apply_asap?`

We used Z3 to check the `apply_asap?` functions in our specification satisfy the four conditions A.1.2–A.1.4. To do so, we used the following Z3 specifications:

1) $apply(s, e)$ of each event.

2) $dominates(s_1, s_2)$. It returns whether $s_1 \sqsupseteq s_2$

3) $C(s, e)$. It specifies the constraints between e and s. When $C(s, e) = True$, it means we can apply e on s asap.

Among these, we specify $apply(s, e)$ using each protocol's TLA$^+$ specification. We show the *dominates* and C specification for Raft below:

```
fn dominates(s1: State, s2: State) {
    //all other states are the same, except commit_index, match_index and
        vote_granted
    //s1's commit_index is not smaller than s2
    //each slot of s1's match_index is not smaller than s2
    //if s1'state is candidate, then s1's vote_granted is the same as s2.
        Otherwise, s1's vote_granted set is the superset of s2.
    return s1.term == s2.term && ... &&
           s1.commit_index >= s2.commit_index &&
           [for i in range(SERVERS):
            s1.match_index[i]>=s2.match_index[i]] &&
           (s1.state == CANDIDATE ?
                equal(s1.vote_granted, s2.vote_granted) :
                    subset(s2.vote_granted, s1.vote_granted))
}

fn C(s: State, e: Event) {
    //we can apply e asap if this is an event from previous term
    //or it is an AppendEntriesResponse in current state's term
    //or this is a RequestVoteResponse in current state's term, and the
       state now is not CANDIDATE
    //or this is a AppendEntryRequest in current state's term, but it
       broadcasts the entires that are a subsequence of the local log.
    //or this is a RequestVoteRequest in current state's term, but server
        has already voted for someone.
    return e.term < s.term ||
```

```
22             (e.term == s.term &&
23                 (e.type == AppendEntriesResponse ||
24                 e.type == RequestVoteResponse &&
25                     s.state != CANDIDATE) ||
26                 e.type == AppendEntryRequest
27                     && s.state == FOLLOWER
28                     && subseq(e.entries, s.log) ||
29                 e.type == RequestVoteRequest
30                     && s.votedFor != None)
31 }
```

The *dominates* and *C* specification for ZAB are below:

```
1 fn dominates(s1: State, s2: State) {
2     //all other states are the same, except last_committed,
          last_processed, ackid of each log entry, and ack_ld_recv
3     //s1's last_committed and last_processed are not smaller than s2
4     //each entry's ack set of s2 is the subset of s1
5     //if in BROADCAST state, s2's ack leader set is the subset of s2.
          Otherwise, s2's ack leader set is the same as s1.
6     return s1.accepted_epoch == s2.accepted_epoch && ... &&
7            s1.last_committed >= s2.last_committed &&
8            s1.last_processed >= s2.last_processed &&
9            [for i in range(ENTRIES):
10            subset(s2.Log[i].ack, s1.Log[i].ack)] &&
11           (s1.zab_state == BROADCAST?
12            subset(s2.ack_ld_recv, s1.ack_ld_recv):
13            equal(s1.ack_ld_recv, s2.ack_ld_recv))
14 }
15
16 fn C(s: State, e: Event) {
17     //we can apply e asap if this is an ACK
```

```
18     //or it is an ACKLD in outdated epoch compared with current state's
           accepted_epoch
19     //or ACKLD in current accepted_epoch but current state already has
           received quorum ACKLD
20     return e.Type == ACK ||
21             (e.Type == ACKLD &&
22                  (Epoch(e.Zxid) < s.accepted_epoch
23                      || has_quorum(s.ack_ld_recv)))
24 }
```

A.3 Protocol Specifications

Below, we provide additional information about the protocol specification used in our evaluation, including detailed breakdown of their lengths and additional information about how they were derived.

A.3.1 Specification Length

Table A.1 shows lines of code for each portion of the Ellsberg specifications used in our evaluation. Similar to Section 2.4 we also report the length of specification used by Microsoft's CCF project for validating Raft [38] and an implementation derived ZooKeeper specification [41].

A.3.2 Raft

We evaluated our approach and Ellsberg using two Raft implementations: etcd [22], a distributed key-value store that is widely deployed, often as a core component of Kubernetes [42] and other orchestrators; and RedisRaft [34, 80], a module that adds consistent replication to the Redis key-value store. In this section, we describe the specification we used.

Component	etcd	RedisRaft	ZooKeeper
State	45	47	102
`apply`	345	312	433
`equal`	64	64	73
`infer_inducing`	298	271	236
`reachable`	37	37	63
`apply_asap?`	39	39	36
`lookahead_type`	9	9	6
Other code	115	115	40
Total (Ellsberg)	952	894	989
Baseline	1503	1503	1615

Table A.1: Lines of code in Ellsberg specification when compared to those used by existing tools. For baselines, we use the Raft specification from Microsoft CCF [38] etcd and RedisRaft, and a recent implementation derived TLA$^+$ specification [41] for ZooKeeper.

Protocol Specification. Our initial approach for producing a specification was to derive one from Ongaro's TLA$^+$ Raft specification [67], since this closely mirrored the protocol described in the Raft paper [68], and we assumed was the distributed protocol implemented in practice. However, we found this was not the case, both implementations we evaluated implement a protocol which makes several changes to the original protocol, and the specification we use includes these changes. The most significant changes from the protocol in the Raft paper affect read-only operations and reconfiguration, and we describe both below.

Read-only operations: Both of the implementations that we evaluated provide linearizable reads, but neither logs read-only operations. Both require the leader to process all read-only requests, and the leader computes the return value using its current state. To ensure linearizability, a node that is processing a read-only request must first assert that it is still the leader, *i.e.,* it needs to check that its term is the same as a quorum's term [99]. Both implementations use the same approach to assert leadership: when a node (that believes it is currently the leader) receives a read-only request, it associates a *request ID* with it and includes this request ID in a heartbeat request (as a reminder, Raft uses empty append entry messages for heartbeats) to other nodes in

the cluster. The node then waits until a quorum (majority) of nodes have positively acknowledged the heartbeat message (showing that a quorum agrees on the term, and that the node was the leader when the request was received), before responding to the read-only request.[1]

Reconfiguration protocol: The other area where both implementations differ is in how they implement cluster reconfiguration: etcd uses different protocols to handle reconfiguration that adds or removes a single node (the etcd documentation [27] seems to prefer this protocol) and ones that add or remove multiple nodes (where etcd largely uses the original joint quorum protocol with minor changes); while RedisRaft only allows the addition or removal of a single node.

Both implementations use a similar (but not identical) single node reconfiguration protocol that avoids the original protocol's use of joint quorums, because in this case quorums are guaranteed to have intersection before and after configuration changes are applied. Therefore, the reconfiguration protocol in this case merely requires that the leader replicate a reconfiguration command to all followers. The two implementation differ on when they apply the reconfiguration command: RedisRaft nodes apply the command as soon as it is added to their log, while etcd treats reconfiguration commands like any other command and applies them asynchronously after they have been committed. This approach can pose a safety problem, and recent versions of etcd change the request vote protocol to ensure that nodes with replicated but unapplied reconfiguration entries do not send request vote messages (and thus do not try to become leaders). This restriction also applies to multi-node reconfiguration in etcd, and one of the bugs we reproduce and detect with Ellsberg (Section 2.6) is due to errors implementing this change.

Beyond differences due to changes in Raft, our specification differs in another crucial way: we model the behavior of messages sent by clients (*e.g.,* `get` and `put`) while Ongaro's specification doesn't.

Our evaluation uses similar protocol specifications for both implementations, and they only

[1] What we have described corresponds to etcd's `ReadIndex` method. It also implements a lease-based mechanism for linearizable reads, but we neither modeled this mechanism nor use it in our evaluation.

differ in when reconfiguration commands take effect.

A.3.3 ZooKeeper Atomic Broadcast

Obtaining a specification for ZooKeeper atomic broadcast (ZAB) [41] as implement by ZooKeeper [104] was a bit easier. While the TLA$^+$ specification included in the Junqueira et al's [41] paper no longer reflects the implementation, a recent pre-print [69] has produced an updated TLA$^+$ specification from the implementation, and we used this specification with minor changes.

ZooKeeper is also easier to model, since it does not support linearizable reads, and we did not need to consider modifications for this.

A.4 Bug Descriptions

Below, we describe the bugs used in our evaluation (Section 2.6.2), their root-cause and how we detected them.

etcd-741 [26] is a bug that predates etcd incorporating the linearizable read protocol we described in Section A.3.2. In older versions, an etcd leader responded to read-only requests with its current value, without asserting that it was the leader, thus violating linearizability. We reproduced this bug by removing the linearizable read logic from etcd. Ellsberg detected the bug when it observed the leader responding to a client's request before receiving positive heartbeat acknowledgments from a quorum.

etcd-7280 [24] and **etcd-12133** [23] are bugs in etcd's reconfiguration protocol. As we noted previously (Appendix A.3.2), etcd supports two different reconfiguration protocols (a single-node reconfiguration that does not require joint-quorums, and a more general version that uses joint-quorums). It also has two different command types for reconfiguration, one which only allows

single node reconfiguration (V1), and one that can be used to trigger both single-node reconfiguration and joint-quorum based reconfiguration (V2). Regardless of command or protocol, reconfiguration takes effect when the command is *applied* by a node. Furthermore, etcd does not allow a node with an unapplied reconfiguration command in its log to send a request vote message (*i.e.,* to transition to being a candidate).

Both of these bugs are caused by cases where etcd incorrectly allowed a node with an unapplied reconfiguration command to become a candidate. In the case of etcd-7280 this was caused because etcd switched to applying reconfiguration requests asynchronously, but assumed that a single reconfiguration request could be in progress at a given time. If more than one reconfiguration request was in progress, then etcd would allow a node to become candidate after applying the first change, resulting in two leaders being elected simultaneously. Similarly, etcd-12133 was caused by a bug in how etcd counted the number of pending reconfiguration messages: the bug meant that etcd did not consider unapplied V2 reconfiguration commands, resulting in a split-brain problem where two leaders are elected for the same term.

In both cases, etcd's safety invariants say that any inducing states for a request vote message has no unapplied reconfiguration commands. Thus, one might expect that we detect these bugs when a node first sends out a request vote message. However, this is not the case: Ellsberg does not know whether a node has applied a reconfiguration command, since commands are applied asynchronously by a background thread. Instead, when Ellsberg see the request vote message, it merely infers that the reconfiguration command has been applied. However, Ellsberg reports a bug when it later observes that the node has transitioned to being a leader (*e.g.,* by observing outgoing append entry messages) without having received votes from a quorum active in the latest configuration.

etcd-7331 [25] is a bug caused by interaction between etcd's linearizable read protocol and Raft's leader election protocol that can result in a newly elected leader returning a stale value in

response to a read. When an etcd leader receives a read-only request, it records the current commit index, and checks that it is still the leader (by sending a heartbeat or append entry message, and waiting for responses from a quorum). However, a newly elected leader's commit index might be smaller than the previous leader's commit index. This is because while Raft's leader election protocol requires that the new leader's log contain all committed entries (and in fact be the most up-to-date log in a quorum), it cannot guarantee that the commit index (which leader's update asynchronously) is up-to-date.

Furthermore, for safety, Raft does not allow a leader to update its commit index until it has replicated an entry in the current term [68, §5.4.2] to a quorum, *i.e.,* it can only update the commit index to point to entries in the current term. Consequently, a read-request handled by a newly-elected leader might return a previous value, violating linearizability.

The etcd developers fixed this protocol bug in its linearizable read protocol by requiring that leaders commit an entry in the current term before initiating a quorum check for a read-only request. Ellsberg uses the same approach to detect this bug: it raises an alarm if it sees the leader is performing a quorum check for a read-only request (which it can detect because heartbeat or append entry messages used for quorum checks include a request ID for the read request) before committing an entry.

ZooKeeper-1154 [105] is data inconsistency bug, where after leader election, nodes disagree on the log. In the leader election protocol implemented by ZooKeeper, a node needs to synchronize its log with all other nodes. To do so, the leader collects the largest entry ID (*ZXID*s) from each follower, and compares the follower's latest ZXID to the latest ZXID in its own log. If the follower's latest ZXID differs, then the leader needs to first have the follower truncate its own log (to the last ZXID on which both agree) and then send missing entries to the follower. Similar to other protocols, ZXIDs in ZooKeeper are monotonically increasing. When implementing the leader election protocol, the ZooKeeper developers introduced a bug and assumed that if the

ZXID sent by the follower was *smaller* than the latest ZXID in the leader's log, then the follower must contain a subset of the leader's log entries. The leader would then avoid truncating the follower's log, and immediately send new entries. However, this is unsafe since the follower's log might contain an entry that has a lower ZXID but is not present in the leader's log, resulting in an inconsistent log. Our simulation detects this when the leader sends out the DIFF message since the follower's latest ZXID is not contained in the leader's log, and thus no inducing state for the DIFF message can be reached.

RedisRaft-17 [76] is also a reconfiguration bug: RedisRaft does not allow concurrent reconfiguration requests, and will accept at most one reconfiguration request at a time (rejecting the other). It considers reconfiguration to be complete once the command has been committed to quorum. Due to a bug, it was not checking this correctly. Detecting this bug with Ellsberg was relatively simple: our inference function says that the inducing state for a successful reconfiguration request is one where the reconfiguration command was added to the log. However, our transition function does not allow transitioning into a state with two uncommitted reconfiguration requests, and thus Ellsberg cannot find a reachable inducing state for the second successful reconfiguration response.

RedisRaft-19 [77] is a bug in RedisRaft's linearizable read implementation. Unlike etcd-741, this bug occurs despite nodes asserting leadership before responding to clients, and is caused by the node being unaware of the current commit index. In Raft, a leader may not know what log entries were committed by the previous leader (in the previous term). Furthermore, Raft requires that leader's only commit entries from previous terms after they have added an entry from the current term to the log [68, §5.4.2]. Therefore, linearizability might be violated if a Raft leader respond to a read-only request before committing an entry in the current term. When writing our protocol specification, we modeled the linearizable read-only protocol by associating a linearization index with each read-only request: a request's linearization index is the log's

commit index if the last entry committed in the current term, and is the last log index otherwise. Furthermore, our inference function requires that the m-inducing state for a read-only response have committed entries up to the request's linearization index. Therefore, Ellsberg flags a bug in this case, because no valid m-inducing states exists for a non-linearizable return value.

RedisRaft Unreported. This previously unreported bug was discovered by Ellsberg when we ran fuzzing loops to find bugs. The bug is due to an additional complication in RedisRaft's reconfiguration protocol: when adding a node, RedisRaft first issues a reconfiguration command to add a *non-voting* node that cannot participate in quorums. After this command has been committed, RedisRaft next issues a second reconfiguration command to switch the non-voting node to a voting node, allowing it to participate in quorums. However, due to a bug, RedisRaft acknowledges the client reconfiguration request once the non-voting reconfiguration command has been successfully replicated. Consequently, RedisRaft reports that reconfiguration is complete before the quorum size has grown, and thus before the cluster can tolerate additional failures. In this case, an ill-timed failure that occurs between the two commands can render the cluster unavailable, despite the administrator believing that this should not occur. Our specification models both steps of the reconfiguration process, and flagged the bug because it observed the leader acknowledging the client's reconfiguration request before the second step had completed.

RedisRaft-265 [78] is a reconfiguration bug related to the unreported bug described above. This bug resulted in leader acknowledging a client reconfiguration command that adds a node before replicating it to a quorum, and can result in situation where the client reconfiguration command is never updated. Ellsberg alerts on this bug when it observes the leader send a client response without having received append entry responses from a quorum, *i.e.,* before it has been committed.

RedisRaft-52 [79] is a bug that leads to data-loss, and is due to a performance optimization implemented by RedisRaft where the last few log entries are stored in a cache. When a node appends an entry it adds it to both the cache and the underlying log. When reading an entry, the node first checks the cache, and only refers to the underlying log if it is not in the cache. A bug in RedisRaft meant that it did not update the cache when deleting entries from the log. However, Raft requires to delete log entries in some scenarios, *e.g.,* when a new leader has to overwrite uncommitted entries. This bug can therefore result in an inconsistent state machine replica, which can lead to unexpected node behavior, *e.g.,* rejecting append entry or vote requests that it should have accepted. We reproduced a case where an append entry request was incorrectly rejected due to an old conflicting entry in the cache. The Ellsberg specification detected a divergence when checking the append entry response and raised an alert.

A.5 Effect of Ellsberg's Design Choices

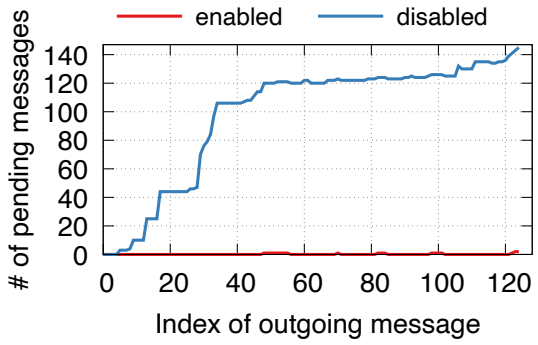
Our evaluation (Section 2.6) presented end-to-end performance results for Ellsberg. In this appendix, we evaluate the impact of `apply_asap?` and the `reachable` function on Ellsberg's performance.

A.5.1 Benefits from `apply_asap?`

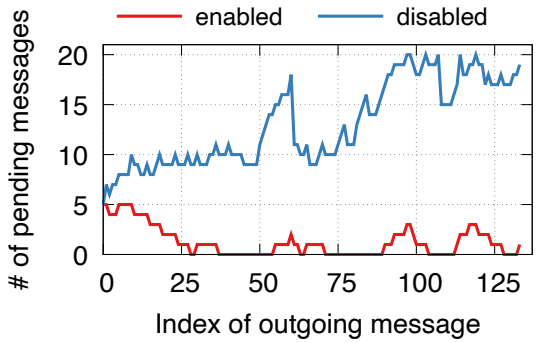
Next we evaluate the effect of using `apply_asap?` to avoid redundant exploration. As we discussed in Section 2.3.4.2, `apply_asap?` is used when processing incoming messages (Listing 2.3 line 20) and during breadth first search (Listing 2.4 line 25) to reduce the number of pending messages, and the size of the simulation state set S. This improves `find_reachable`'s runtime and reduces Ellsberg's memory requirement.

We evaluate improvements in `find_reachable`'s runtime by comparing the number of pending messages when `apply_asap?` is enabled to when it is disabled. In Figure A.1, we show the

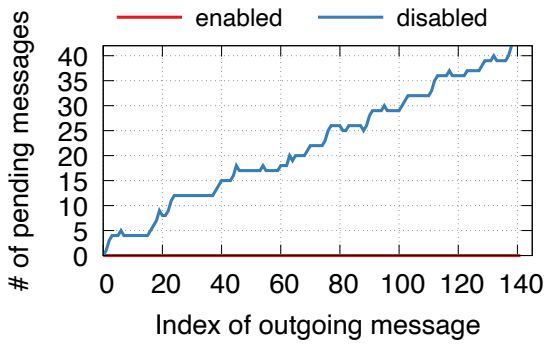
number of pending messages as a function of the number of outgoing messages processed. We report results from a 5-node cluster when running the balanced workload, the results were similar for other workloads. We observe that when `apply_asap?` is disabled, the number of pending



(a) etcd

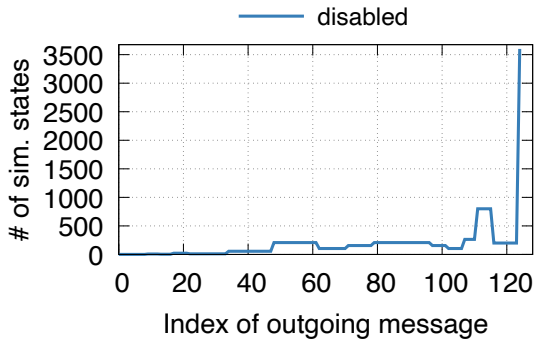


(b) ZooKeeper.

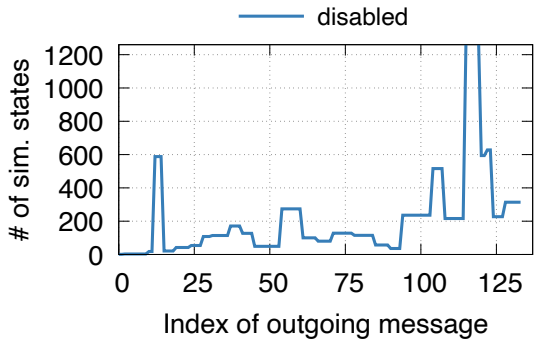


(c) RedisRaft

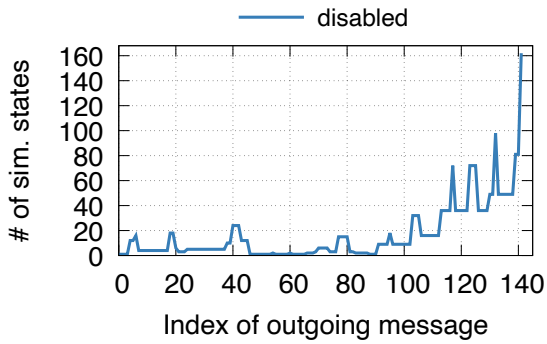
Figure A.1: `apply_asap?` avoids redundant exploration: graphs show pending messages in a 5-node cluster after checking the n^{th} outgoing message `apply_asap?` *enabled* and *disabled*.



(a) etcd.

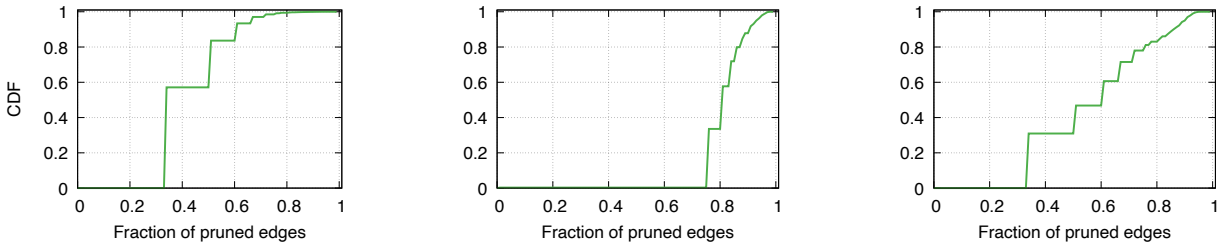


(b) ZooKeeper.



(c) RedisRaft.

Figure A.2: `apply_asap?` reduces memory requirements: graphs show number of simulation states $|S|$ in a 5-node cluster with `apply_asap?` disabled. ($|S| = 1$ when `apply_asap?` is enabled.)



(a) etcd. **(b)** ZooKeeper. **(c)** Redis.

Figure A.3: CDF of fraction of edges pruned by the `reachable` check (Listing 2.4 line 22). All experiments are run on a 5-node cluster using the mixed workload.

messages grows as Ellsberg processes more of the trace. By contrast, enabling `apply_asap?` limits the number of pending messages: in our workloads we observe between 0 and 5 pending messages.

Using `apply_asap?` also reduces the number of simulation states in S, improving both performance (results in fewer calls to `find_reachable`) and memory requirements. We evaluated this improvement by comparing the size of S as a function of number of output messages processed with `apply_asap?` enabled and disabled. Results from running a balanced workload on a 5-node cluster, when `apply_asap?` is disabled, are shown in Figure A.2. When `apply_asap?` is enabled, we found that S contained a single simulation state ($|S|$=1). By contrast, when `apply_asap?` is disabled the number of states can grow large, with an average of $\sim$ 444 states for etcd, $\sim$ 478 states for ZooKeeper, and $\sim$ 43 for RedisRaft.

In sum, these results show the importance of `apply_asap?` for limiting Ellsberg's memory usage and in achieving our performance goals.

A.5.2 The `reachable` function's impact

Ellsberg uses the specification's `reachable` function to avoid exploring schedules that cannot lead to m-inducing simulation states (Section 2.3.4.1, Listing 2.4 line 22). As we explained in Section 2.3.4.1, we need to use this function to ensure that checking an outgoing message

terminates, and we cannot turn it off.

Therefore, we evaluate the effectiveness of reachability checks by drawing a CDF of the fraction of `reachable` calls that return false (and thus prune exploration). In Figure A.3, we show results for the three DPIs running on a 5-node cluster evaluated using the balanced workload. As one would expect, the results depend on the protocol and implementation: pruning is most efficient for ZooKeeper because it assumes in-order deliver. The in-order delivery assumption ensures that the number of pending messages is bound by the number of nodes and the protocol structure allows the `reachable` function to prune more than 75% of possible paths. While pruning is less effective for etcd and RedisRaft, we observe that even in this case, between 40–80% of calls to the `reachable` function return false. These results show that while Ellsberg can work with specification that over-approximate reachability, this approximation can come at a performance cost.

Bibliography

[1] Marcos K Aguilera et al. “Performance debugging for distributed systems of black boxes”. In: *SOSP*. 2003.

[2] Ram Alagappan. “Crash Consistency: Keeping data safe in the presence of crashes is a fundamental problem.” In: *Queue* 20.4 (2022), pp. 107–115.

[3] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. “Lineage-driven Fault Injection”. In: *SIGMOD*. 2015.

[4] Peter Alvaro and Severine Tymon. “Abstracting the geniuses away from failure testing”. In: *ACM Queue* 15.5 (2017), pp. 29–53.

[5] Peter Alvaro et al. “Automating failure testing research at internet scale”. In: *SoCC*. 2016, pp. 17–28.

[6] AWS. *Amazon S3 now delivers strong read-after-write consistency automatically for all applications*. https://aws.amazon.com/about-aws/whats-new/2020/12/amazon-s3-now-delivers-strong-read-after-write-consistency-automatically-for-all-applications/. Dec. 2020.

[7] Peter Bailis, Peter Alvaro, and Sumit Gulwani. “Research for practice: Tracing and debugging distributed systems; programming by examples”. In: *Communications of the ACM* 60.7 (2017), pp. 46–49.

[8] Ivan Beschastnikh et al. “Debugging distributed systems”. In: *ACM Queue* 14.2 (2016), pp. 91–110.

[9] Borzoo Bonakdarpour et al. “Decentralized asynchronous crash-resilient runtime verification”. In: *CONCUR*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[10] James Bornholt et al. “Using lightweight formal methods to validate a key-value storage node in Amazon S3”. In: *SOSP*. 2021.

[11] Mike Burrows. “The Chubby lock service for loosely-coupled distributed systems”. In: *OSDI*. 2006.

[12] Tej Chajed et al. “Verifying concurrent Go code in Coq with Goose”. In: *CoqPL*. Vol. 2020. 2020.

[13] K. Mani Chandy and Leslie Lamport. “Distributed Snapshots: Determining Global States of Distributed Systems”. In: *ACM Trans. Comput. Syst.* 3.1 (Feb. 1985), pp. 63–75.

[14] Himanshu Chauhan et al. “A distributed abstraction algorithm for online predicate detection”. In: *International Symposium on Reliable Distributed Systems*. IEEE. 2013, pp. 101–110.

[15] Qian Cheng et al. *Specula: Scaling Formal Specifications for Autonomous Model Checking of System Code*. 2026.

[16] Horatiu Cirstea et al. “Validating Traces of Distributed Programs Against TLA+ Specifications”. In: *arXiv preprint arXiv:2404.16075* (2024).

[17] Brian F Cooper et al. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154.

[18] Biplob Debnath et al. “LogLens: A Real-Time Log Analysis System”. In: *ICDCS* (2018), pp. 1052–1062.

[19] Dmitry Duplyakin et al. “The design and operation of CloudLab”. In: *USENIX ATC*. 2019.

[20] *Envoy Proxy*. https://www.envoyproxy.io/.

[21] Úlfar Erlingsson et al. "Fay: Extensible distributed tracing from kernels to clusters". In: *ACM Transactions on Computer Systems (TOCS)* 30.4 (2012), pp. 1–35.

[22] *etcd*. https://coreos.com/etcd/.

[23] *Pending conf change is not handled correctly during campaign*. https://github.com/etcd-io/etcd/issues/12133. 2020.

[24] *Raft: async apply ConfChange is not safe*. https://github.com/etcd-io/etcd/issues/7280. 2017.

[25] *ReadIndex may read stale value*. https://github.com/etcd-io/etcd/issues/7331. 2017.

[26] *Consistent reads are not consistent*. https://github.com/etcd-io/etcd/issues/741. 2014.

[27] *etcd v3.5: Runtime reconfiguration*. https://etcd.io/docs/v3.5/op-guide/runtime-configuration/.

[28] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[29] Cormac Flanagan and Patrice Godefroid. "Dynamic partial-order reduction for model checking software". In: *POPL*. 2005.

[30] Pedro Fonseca et al. "An empirical study on the correctness of formally verified distributed systems". In: *EuroSys*. 2017.

[31] Rodrigo Fonseca et al. "X-Trace: A Pervasive Network Tracing Framework". In: *NSDI*. 2007.

[32] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. “Runtime Verification for Decentralised and Distributed Systems”. In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Springer, 2018, pp. 176–210.

[33] Klaus v. Gleissenthall et al. “Pretend synchrony: synchronous verification of asynchronous distributed programs”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–30.

[34] Yossi Gottlieb. *Introducing RedisRaft, a New Strong-Consistency Deployment Option*. https://redis.com/blog/redisraft-new-strong-consistency-deployment-option/. June 2020.

[35] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. “Inferring and Asserting Distributed System Invariants”. In: *ICSE*. 2018.

[36] Tobias Grieger. *raft: don’t emit unstable CommittedEntries*. https://github.com/etcd-io/etcd/pull/14413. etcd pull request #14413; merged September 22, 2022; fixes issue #14370. Sept. 2022.

[37] Chris Hawblitzel et al. “IronFleet: proving practical distributed systems correct”. In: *SOSP*. 2015, pp. 1–17.

[38] Heidi Howard et al. “Smart Casual Verification of the Confidential Consortium Framework”. In: *NSDI*. 2025, pp. 259–276.

[39] Lexiang Huang et al. “Metastable Failures in the Wild”. In: *OSDI*. 2022.

[40] Peng Huang et al. “Gray failure: The achilles’ heel of cloud-scale systems”. In: *HotOS*. 2017.

[41] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. “Zab: High-performance broadcast for primary-backup systems”. In: *DSN*. 2011.

[42] *Operating etcd clusters for Kubernetes*. https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/, retrieved in April 2023.

[43] Jonathan Kaldor et al. “Canopy: An End-to-End Performance Tracing And Analysis System”. In: *SOSP*. 2017.

[44] Charles Killian et al. “Life, death, and the critical transition: Finding liveness bugs in systems code”. In: *NSDI*. 2007.

[45] Charles Edwin Killian et al. “Mace: language support for building distributed systems”. In: *PLDI*. 2007.

[46] Kyle Kingsbury. *Jepsen: Distributed Systems Safety Research*. https://jepsen.io/.

[47] Leslie Lamport. *TLA+ Tools*. http://lamport.azurewebsites.net/tla/tools.html. 2022.

[48] Pedro Las-Casas et al. “Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering”. In: *SoCC*. 2019.

[49] Pedro Las-Casas et al. “Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay”. In: *SoCC*. 2018.

[50] Andrea Lattuada et al. “Verus: Verifying rust programs using linear ghost types”. In: *OOPSLA*. 2023.

[51] Tanakorn Leesatapornwongsa et al. “SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems”. In: *OSDI*. 2014.

[52] Rustan Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. 2010.

[53] Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. “AIDev: Studying AI Coding Agents on GitHub”. In: *Proceedings of the 23rd International Conference on Mining Software Repositories*. 2026, pp. 1029–1033. DOI: 10.1145/3793302.3797249.

[54] Xuezheng Liu et al. “D3S: Debugging Deployed Distributed Systems”. In: *NSDI*. 2008.

[55] Jacob R Lorch et al. “Armada: low-effort verification of high-performance concurrent programs”. In: *PLDI*. 2020.

[56] Chang Lou, Peng Huang, and Scott Smith. “Understanding, Detecting and Localizing Partial Failures in Large System Software”. In: *NSDI*. Feb. 2020.

[57] Chang Lou, Yuzhuo Jing, and Peng Huang. “Demystifying and Checking Silent Semantic Violations in Large Distributed Systems”. In: *OSDI*. 2022.

[58] Nancy A Lynch and Mark R Tuttle. *An introduction to input/output automata*. Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[59] Haojun Ma et al. “I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols”. In: *SOSP*. Huntsville, Ontario, Canada, 2019.

[60] Jonathan Mace and Rodrigo Fonseca. “Universal context propagation for distributed system instrumentation”. In: *EuroSys*. 2018.

[61] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. “Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems”. In: *SOSP*. 2015.

[62] Rupak Majumdar and Filip Niksic. “Why is random testing effective for partition tolerance bugs?” In: *Proc. ACM Program. Lang.* 2 (2017), 46:1–46:24.

[63] Christopher S Meiklejohn et al. “Service-level fault injection testing”. In: *SoCC*. 2021, pp. 388–402.

[64] Ellis Michael et al. “Teaching rigorous distributed systems with efficient model checking”. In: *EuroSys*. 2019, pp. 1–15.

[65] Francisco Neves, Nuno Machado, and José Pereira. “Falcon: A Practical Log-Based Analysis Tool for Distributed Systems”. In: *DSN* (2018), pp. 534–541.

[66] Chris Newcombe et al. *Use of formal methods at Amazon Web Services*. See https://lamport.azurewebsites.net/tla/formal-methods-amazon.pdf. 2014.

[67] Diego Ongaro. *Github: ongardie/raft.tla*. https://github.com/ongardie/raft.tla.

[68] Diego Ongaro and John K. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *USENIX ATC*. 2014.

[69] Lingzhi Ouyang et al. "Leveraging TLA+ Specifications to Improve the Reliability of the ZooKeeper Coordination Service". In: *arXiv preprint arXiv:2302.02703* (2023).

[70] Burcu Kulahcioglu Ozkan et al. "Randomized testing of distributed systems with probabilistic guarantees". In: *Proceedings of the ACM on Programming Languages* 2 (2018), pp. 1–28.

[71] Oded Padon et al. "Induction duality: primal-dual search for invariants". In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022), pp. 1–29.

[72] Oded Padon et al. "Ivy: Safety Verification by Interactive Generalization". In: *PLDI*. 2016.

[73] Oded Padon et al. "Paxos made EPR: decidable reasoning about distributed protocols". In: *OOPSLA*. 2017.

[74] Austin Parker et al. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, 2020.

[75] Sundar Pichai. *Cloud Next '26: Momentum and Innovation at Google Scale*. Google. Apr. 22, 2026. URL: https://blog.google/innovation-and-ai/infrastructure-and-cloud/google-cloud/cloud-next-2026-sundar-pichai/ (visited on 08/13/2026).

[76] *Possible lost updates with partitions and membership changes*. https://github.com/RedisLabs/redisraft/issues/17. 2020.

[77] *Stale reads in normal operation*. https://github.com/RedisLabs/redisraft/issues/19. 2020.

[78] *RAFT.NODE ADD should be a blocking operation*. https://github.com/RedisLabs/redisraft/issues/265. 2022.

[79] *Loss of committed writes, duplicate elements, dueling histories*. https://github.com/RedisLabs/redisraft/issues/52. 2020.

[80] *Github: RedisLabs/redisraft*. https://github.com/RedisLabs/redisraft.

[81] Sundararajan Renganathan et al. "Hydra: Effective Runtime Network Verification". In: *SIGCOMM*. 2023.

[82] Patrick Reynolds et al. "Pip: Detecting the Unexpected in Distributed Systems." In: *NSDI*. Vol. 6. 2006.

[83] Michiel Ronsse and Koen De Bosschere. "RecPlay: a fully integrated practical record/replay system". In: *ACM Transactions on Computer Systems (TOCS)* 17.2 (1999), pp. 133–152.

[84] Casey Rosenthal et al. *Chaos engineering*. O'Reilly Media, Incorporated, 2020.

[85] Colin Scott et al. "Minimizing Faulty Executions of Distributed Systems". In: *NSDI*. 2016.

[86] Ilya Sergey, James R Wilcox, and Zachary Tatlock. "Programming and proving with distributed protocols". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–30.

[87] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010.

[88] Cheng Tan et al. "Cobra: Making Transactional Key-Value Stores Verifiably Serializable". In: *OSDI*. 2020.

[89] The Coq Development Team. *The Coq Proof Assistant*. https://coq.inria.fr/.

[90] Alexander Thomson et al. "Calvin: Fast Distributed Transactions for Partitioned Database Systems". In: *SIGMOD*. 2012.

[91] Dong Wang et al. "Model checking guided testing for distributed systems". In: *EuroSys*. 2023.

[92] James R Wilcox et al. “Verdi: a framework for implementing and formally verifying distributed systems”. In: *PLDI*. 2015.

[93] Doug Woos et al. “Planning for change in a formal verification of the raft consensus protocol”. In: *CPP 2016*. 2016.

[94] Junfeng Yang et al. “MODIST: Transparent model checking of unmodified distributed systems”. In: *NSDI*. 2009.

[95] Jianan Yao et al. “DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols”. In: *OSDI*. 2021.

[96] Jianan Yao et al. “DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols”. In: *OSDI*. 2022.

[97] Nofel Yaseen et al. “Aragog: Scalable Runtime Verification of Shardable Networked Systems”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020.

[98] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. “Model Checking TLA+ Specifications”. In: *CHARME*. 1999.

[99] Pierre Zemb. *Diving into ETCD’s linearizable reads*. https://pierrezemb.fr/posts/diving-into-etcd-linearizable/. Sept. 2020.

[100] Ennan Zhai et al. “Check before You Change: Preventing Correlated Failures in Service Updates.” In: *NSDI*. 2020.

[101] Jialu Zhang et al. “Static detection of silent misconfigurations with deep interaction analysis”. In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021).

[102] Jian Zhang et al. “Viper: A Fast Snapshot Isolation Checker”. In: *EuroSys*. 2023.

[103] Jingyu Zhou et al. “FoundationDB: A Distributed Unbundled Transactional Key Value Store”. In: *SIGMOD*. 2021.

[104] *Apache Zookeeper*. https://zookeeper.apache.org/.

[105] *Data inconsistency when the node(s) with the highest zxid is not present at the time of leader election*. https://issues.apache.org/jira/browse/ZOOKEEPER-1154. 2011.