Runtime Protocol Refinement Checking for Distributed Protocol Implementations

Ding Ding, Zhanghan Wang, Jinyang Li, Aurojit Panda
NYU

Abstract

Despite significant progress in verifying protocols, services that implement distributed protocols (we refer to these as DPIs in what follows), e.g., Chubby or Etcd, can exhibit safety bugs in production deployments. These bugs are often introduced by programmers when converting protocol descriptions into code. This paper introduces *Runtime Protocol Refinement Checking* (RPRC), a runtime approach for detecting protocol implementation bugs in DPIs. RPRC systems observe a deployed DPI's runtime behavior and notify operators when this behavior evidences a protocol implementation bug, allowing operators to mitigate the bugs impact and developers to fix the bug. We have developed an algorithm for RPRC and implemented it in a system called Ellsberg that targets DPIs that assume fail-stop failures and the asynchronous (or partially synchronous) model. Our goal when designing Ellsberg was to make no assumptions about how DPIs are implemented, and to avoid additional coordination or communication. Therefore, Ellsberg builds on the observation that in the absence of Byzantine failures, a protocol safety properties are maintained if all live DPI processes correctly implement the protocol. Thus, Ellsberg checks RPRC by comparing messages sent and received by each DPI process to those produced by a simulated execution of the protocol. We apply Ellsberg to three open source DPIs, Etcd, Zookeeper and Redis Raft, and show that we can detect previously reported protocol bugs in these DPIs.

1 Introduction

Distributed protocols, implemented in lock services [11, 21, 100], distributed databases [85, 99], 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. Popular DPIs have shipped with application visible bugs: *e.g.,* prior versions of Etcd [21] would return stale values in some scenarios due to a bug [25] in its implementation of a linearizable read protocol [95]. Similarly, prior versions of Zookeeper [100] would, in some scenarios, lose updates during leader elections due to a protocol implementation bug [101]. Both systems are widely used, and both bugs can result in application bugs such as data corruption.

In this paper we develop an approach, *runtime protocol refinement checking* (RPRC), that notifies administrators when bugs such as the ones above occur, allowing administrators to mitigate the bug's effects and programmers to fix the bug. We started developing RPRC because we observed that despite the widespread use of verification tools [3, 4, 10, 41, 42, 55, 60, 65, 68, 90–92, 94] that prove the safety of distributed protocols, and testing tools [6, 15, 35, 41–43, 48, 66] that check implementation correctness, deployed DPIs continue to exhibit bugs. This is because ensuring that programmer correctly implements a verified protocol is challenging, and detecting (and eliminating) all bugs with testing is infeasible. Recently, IronFleet [34], Verus [47], Goose [12] and others, have proposed using static refinement proofs to connect verified protocols to implementations. However, to ensure feasibility (of refinement proof generation and checking) these frameworks must limit how code is written, making it hard for developer to produce verified, high-performance DPIs.

RPRC provides an alternate 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 paper, 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, 34, 47, 49, 55, 68, 88, 91, 92] (they cannot statically generate proofs to show that a protocol is correct, or that an implementation refines the protocol) and fuzz testing tools [6, 41–43, 48, 66] (they do not try to maximize coverage and find all bugs before an implementation is deployed), but rather complement them.

This paper 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 find bugs that were missed during testing.

Ellsberg (Figure 1) consists of a set of colocated instances that are run alongside (that is colocated 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

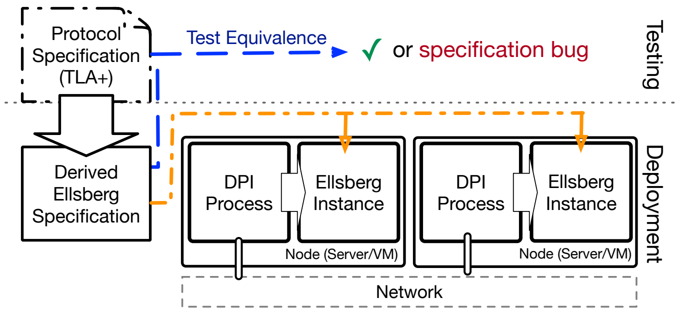


Figure 1: *An overview of* Ellsberg, *which runs an instance colocated with each DPI process, and compares the colocated process's behavior to a correct protocol implementation. We assume the TLA+ protocol specification is provided by the protocol designer, and programmers derive a* Ellsberg *specification from this specification.*

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 1) as follows: before deployment, the administrator (or DPI programmer) translates the protocol the DPI purportedly implements into an Ellsberg specification and tests it (§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 (§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 [78, 80] that records processing order.

Ellsberg addresses these challenges using a novel incremental RPRC algorithm (§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 optimization 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 Redis Raft, an extension of the Redis key-value store that adds fault-tolerance. Etcd and Redis Raft implement the Raft [64] consensus protocol, and ZooKeeper implements the Zab [38] atomic broadcast protocol. Our evaluation (§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.

While 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 (§2.1) to detect these bugs. Nevertheless, as our evaluation (§6) shows, safety bugs of the kind we find do occur in practice.

2 Current 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 [15,35] validates totally ordered traces generated during 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, thus 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 reduces the number of bugs before a DPI is deployed, while Ellsberg can identify bugs that were missed by testing but appear 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 the implementor used the TLA+ specification as a blueprint when writing code ([15, §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 [35, §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 [35, §4, §5] to use trace validation. By contrast, Ellsberg does not require close correspondence between specification and 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 Redis Raft.

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 additional coordination – *e.g.,* Microsoft CCF [35, §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 [88], Diesel [81] and others [89] provide frameworks that make it easier for users to write proofs that can be mechanically checked by a proof assistant (*e.g.,* Coq [84]). Ivy [67, 68] 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 [55, 69, 91, 92] have extended Ivy's approach and automated the process of inferring inductive invariants.TLA+ [10, 62, 94] 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 [34] takes as input safety properties (referred to as a high-level specification), a protocol specification together with an implementation, and prove using Dafny [49] 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 [86] has shown how to reduce the proof burden for IronFleet by restricting the communication primitives used by the program. Concurrently, Verdi [88, 89] and Diesel [81] 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 [51].

Furthermore, these approaches make assumptions about the runtime environment, including assumptions about system call semantics and the network. Prior work [29] 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, 43, 66], and ones that use model checking [3, 4, 6, 41, 42, 48, 60, 87, 90]. Many of these approaches [3, 43, 48, 66] are explicitly designed to be used with existing DPIs. Recent work from Majumdar and Niksic [58] 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.

Runtime verification to find bugs in DPIs. Runtime verification approaches, *e.g.,* Dinv [33], D3S [50], Pip [77], Aragog [93], Oathkeeper [53] and Hydra [76], 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 Franclanza et al's survey [31] for details). The use of DPI state snapshots allows these tools to check richer properties without concerns about decidability or scaling, *e.g.,* Aragog [93] 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 [53, 76, 93], using fork-join parallelism to distribute predicate checking [33, 50, 76, 77, 93]. 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.

2.1 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, 20, 30, 57, 70, 82] 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, 17, 40, 45, 46, 56, 61] 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 [79], lineage driven fault injection [3–5, 59], crash consistency checking [2], and analysis tools for partial and correlated failures [37, 52, 96] 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 [37], metastable failures [36], and bugs due to misconfiguration [97].

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, 52, 80]. As we discussed previously, these include record-replay [8, 78, 80] based debuggers which, similar to RPRC, also need to consider multiple interleavings when reproducing bugs.

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 (§3.1) and describe a ticket-lock service that we use as a running example in this section (§3.2). Then, we discuss the protocol specification a user must provide to Ellsberg (§3.3), and our RPRC algorithm (§3.4).

Overview. Ellsberg (Figure 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 (§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.

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 assumes that the user-provided protocol specification is correct. We provide a testing tool (§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 faithfully 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

```
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 1: *Example ticket-lock server.*

in the trace. Our prototype uses an IPC channel, that connects each Ellsberg instance to its colocated process, to collect the trace (§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 [15, 35], 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.

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 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 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 Redis Raft) for the evaluation (see Appendix C for specifications).

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 [27, 54]. 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 proceesed, 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 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.

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, we use the ticket-lock service's (§3.2) specification to illustrate the parts of a Ellsberg specification.

A Ellsberg protocol specification consists of:

```
1  struct ProtState {
2      highest: int,
3      current: int,
4      held: bool,
5  }
6
7  fn equal(s: ProtState, s': ProtState) -> bool {
8      return s.highest == s'.highest
9        && s.current == s'.current
10       && s.held == s'.held
11 }
12
13 fn infer_inducing(m: Message) -> ProtState {
14     return match m {
15         Acquired(tkt) =>
16             ProtState{highest: None,
17               held: true, current: tkt},
18         // ...
19     }
20 }
21
22 fn apply(s: ProtState, e: Event) -> ProtState {
23   match e {
24     case Message(Acquire(tkt)) => {
25         if s.current ==  tkt{
26             return ProtState{highers
                   : s.highest, current:tkt, held:true}
27         } else {
28             return s
29         }
30     }, // ...
31   }
32 }
33
34 fn reachable(s: ProtState,
35              pending: Set[Set[Message]], e: Event,
36              p: ProtState) -> bool {
37    if state(s).current > p.current {
38     return false
39     } // ...
40   }
41 }
42
43 fn apply_asap?(s: ProtState, m: Message) -> bool {
44     return m.type == Acquire && m.tkt < s.current
45 }
46
47 fn lookahead_type(m: Message) -> Message_type {
48     if Type(m) == Acquired {
49         return Assigned
50     }
51     // ...
52 }
```

Listing 2: *User-provided protocol specification for the ticket-lock server*

(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 22), 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 7) 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 13 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 34).

(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 (§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 in meets these requirements in our example (line 43).

(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 (§3.4.2), this function is used in an optimization to improve Ellsberg's efficiency, but its semantics do not affect correctness. Line 47 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 §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.

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,...,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 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 2). For notational convenience, we use s_{sim}

```
1
2  struct SimState {
3      state: State,
4      pending: Set[Set[Messages]]
5  }
6
7  // The instance's current simulation state.
8  S: Set[SimState]
9
10 fn prune_asap(s: SimState) -> SimState {
11     for e in pending(s) {
12         if apply_asap?(state(s), e) {
13             return prune_asap(apply(s, e))
14         }
15     }
16     return s
17 }
18
19 // Called
       when m is an incoming message in the trace.
20 fn process_incoming(m: message) {
21     for s in S {
22         // Check if m should be applied immediately.
23         if apply_asap?(state(s), m) {
24             S.replace(s, prune_asap(apply(s, m)))
25         } else {
26             s.add_pending_message(m)
27         }
28     }
29 }
30
31 // Called when
       message m is an outgoing message in the trace.
32 fn process_outgoing(m: message) {
33     // Get m-inducing states.
34     let target = infer_inducing(m)
35     // Check if we need to lookahead.
36     let m_lookahead = if lookahead_type(m) {
37         infer_inducing(
38             find_next_of_type(lookahead_type(m)))
39     } else {
40         None
41     }
42     // Updated simulation state.
43     let S' = new Set[SimState]
44     for s in S {
45         // Find reachable m-inducing states from s.
46         S_s = find_reachable(s,
47                 target, m_lookahead)
48         S' = S'.union(S_s)
49     }
50     if S'.is_empty() {
51         // No reachable m-inducing state indicates
52         // divergence b/w spec and implementation.
53         raise Error
54     } else {
55         S = S'
56     }
57 }
```

Listing 3: *Algorithm for checking a single outgoing message.*

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 §3.4.2) to the pending message set of all simulation states $s_{sim} \in S$ (line 20). 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 46, §3.4.1). The algorithm notifies an error (line 53) if no m-inducing simulation states are reachable, otherwise it updates S to be the set of reachable m-inducing simulation states (line 55).

```
 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              // Did we already enque a
                      semantically equivalent sim. state?
28              if to_explore.contains_state(s'') {
29                  // Add pending messages
                          from s'' to the queued copy.
30                  merge_pending(
31                      to_explore.get_state(s''), s'')
32              } else if !explored.contains_state(s'')
                     {
33                  // Add sim. state
                          s'' to the exploration queue.
34                  to_explore.insert_tail(s'')
35              } else {
36                  // Add s'' to the exploration
                          queue only if its pending
37                  // events differ from
                          what was previously explored.
38                  if !pending(explored.get_state(s''))
39                      .contains(pending(s'')) {
40                      to_explore.insert_tail(s'')
41                  }
42              }
43          }
44      }
45      return reachable_inducing
46  }
```

Listing 4: *Ellsberg's breadth-first search algorithm to find reachable m-inducing simulation states (passed in as argument* `target`*) starting from simulation state* `s`*.*

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.

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 3 line 34), 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' where one is a subsequence of the other (WLOG assume 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 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—44) to find reachable target states. Breadth first search ensures that shorter schedules, and thus simulation states

[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.

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 39).

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 `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 39, 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 (Line 22). We prune exploration from $apply(s_{sim}, e)$ if the `reachable` function returns false.

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 (§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 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 10) to replace s_{sim} with the result of applying $apply(s_{sim}, m)$ and all other messages that can be applied immediately (Line 24); and (b) in the `find_reachable` (Listing 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 [28] (POR), our technique is designed for the incremental setting.

In Appendix A 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 3 line 44). 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 4 line 24).

We use the `lookahead_type` function in the user-provided specification (§3.3, Listing 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 3 line 38) 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 `find_reachable` would have been pruned by a future call to `find_reachable` when considering the next outgoing message of type T.

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 3 Line 20) 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) in the trace triggers a check to determine if the DPI is buggy (List-

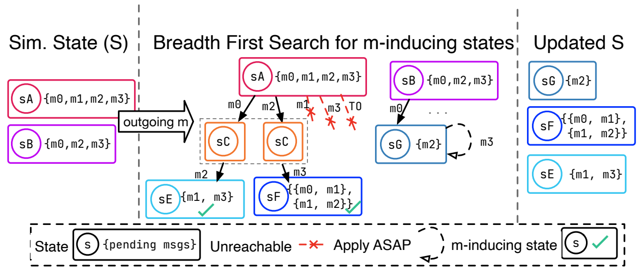


Figure 2: *An overview of how a Ellsberg processes outgoing message m using BFS to find all m-inducing states reachable from the current simulation state.*

System	Ellsberg LOC	Baseline	Baseline LOC
ETCD	952	CCF Raft [35]	1503
RedisRaft	894	CCF Raft	1503
Zookeeper	989	ZooKeeper TLA+ [38]	1615

Table 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 [35] Etcd and RedisRaft, and a recent implementation derived TLA+ specification [38] for ZooKeeper.*

ing 3 Line 32), which involves iterating through the simulation states $s_{sim} \in S$ and using the `find_reachable` (Listing 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? 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 (§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 protocol, including the RSM protocols we evaluate our work using. 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 [83, 98].

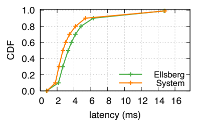
4 Writing Ellsberg Specifications

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 [63] provided by the Raft authors, while for ZooKeeper we used an implementation derived TLA+ specification [38].

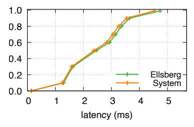
Table 1 shows the length of our specifications ('Ellsberg LOC' column). We use the same Raft specification for Etcd and Redis Raft, the difference in length is because they support different reconfiguration protocols. To put our specifications length into context, we also report the length of specifications 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 Microsft 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 C provides additional details about the Ellsberg specifications, including a detailed breakdown of lengths for each part (Appendix C.1).

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

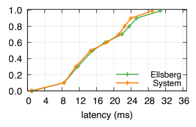
- **`ProtState`**. The `ProtState` structure contains all per-server variables that appear in the TLA+ specification. We derived it by identifying all variable in the TLA+ specification, filtering out those that are global variables (*e.g.,* `messages` in the Raft specification [63]) or ghost variables (*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;



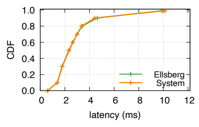
(a) *Etcd (balanced).*



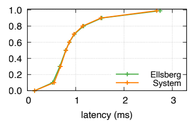
(b) *ZK (balanced).*



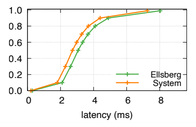
(c) *Redis (balanced).*



(d) *Etcd (read-heavy).*



(e) *ZK (read-heavy).*



(f) *Redis (read-heavy).*

Figure 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.*

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 proved correct (described in Appendix A), and the `lookahead_type` function has no impact on Ellsberg's correctness (§3.4.2).

Testing Ellsberg specifications. We also provide a testing tool, inspired by Mocket [87], to check that the derived Ellsberg specification matches the original TLA+ specification, and does not have translation bugs.

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 [44] 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.

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 [64] and ZAB [38], and used Etcd, ZooKeeper and RedisRaft as DPIs.

The Raft specification we use consists of ∼500 lines of code, and the ZooKeeper specification consists of ∼2000 lines of code. We describe how we developed these specifications from existing machine checked specifications in Appendix C. We used the Ellsberg testing tool (§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 Redis Raft. 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 [19] copy messages to Ellsberg.

6 Evaluation

We evaluated Ellsberg using three widely-used open source DPIs: Etcd, Zookeeper, and Redis Raft. Our evaluation focused on three questions: (a) Can Ellsberg detect bugs at runtime? (§6.2) (b) Can Ellsberg be deployed in production? (§6.3) (c) How does Ellsberg perform: how long does it take Ellsberg to process a trace, and the impact of our design (§3.4) on performance? (§6.4)

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 [18], 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 [16]. 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.

System	Bug	Type
Etcd	741 [25]	Linearizable read
	7331 [24]	Stale read after election
	12133 [22]	Reconfiguration
	7280 [23]	Reconfiguration
Zookeeper	1154 [101]	Data inconsistency
Redis Raft	17 [71]	Reconfiguration
	19 [72]	Linearizable read
	52 [74]	Lost updates
	256 [73]	Reconfiguration
	Unreported	Reconfiguration

Table 2: *Bugs used to evaluate ELLSBERG's bug detection capability.*

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

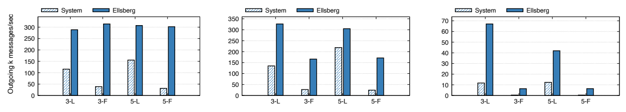
6.2 Can ELLSBERG detect bugs?

We evaluated ELLSBERG's ability to detect protocol implementation bugs by reproducing previously reported bugs in Etcd, Redis Raft 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 D, but summarize them in Table 2. The bugs we tested on including bugs in the linearizable read protocol implemented by Etcd [25] and Redis Raft [72]; a stale-read bug in Etcd [24] because a new leader's commit index might lag behind the previous leader's commit index; a bug in Zookeeper's [101] leader election protocol (leading to data corruption); a bug in Redis Raft's caches that leads to lost updates [73]; and reconfiguration bugs in Etcd [22] and Redis Raft [71]. During testing, we also identified an unreported bug in Redis Raft's reconfiguration protocol implementation.

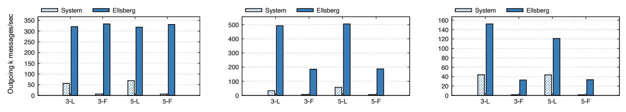
6.3 Can ELLSBERG be used in production?

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 limits each ELLSBERG instance to 2 cores. Furthermore, we found (explained in detail in Appendix E) 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. Additional, ELLSBERG does not use the network by design. We thus conclude that ELLSBERG has minimal resource overheads.

In terms of DPI performance impact, Figure 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 Redis Raft 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 3f) 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.



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



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

Figure 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.*

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 (§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 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 Redis Raft 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.

In Appendix E we evaluate how `apply_asap?` reduces the number of traces explored by ELLSBERG and the amount of memory required by ELLSBERG; and the impact of the `reachable` function on exploration time.

7 Conclusion

This paper proposed *runtime protocol refinement checking*, an approach that combines ideas from refinement proofs and runtime verification to notify administrator of protocol bugs in deployed 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 our implementation of this algorithm Ellsberg, and applied it to three commonly used DPIs: Etcd, ZooKeeper and Raft. We believe that RPRC and Ellsberg provide a practical approach for improving the reliability of deployed DPIs.

References

[1] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 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. *Queue*, 20(4):107–115, 2022.

[3] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. Automating failure testing research at internet scale. In *SoCC*, pages 17–28, 2016.

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

[5] Peter Alvaro and Severine Tymon. Abstracting the geniuses away from failure testing. *ACM Queue*, 15(5):29–53, 2017.

[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/, December 2020.

[7] Peter Bailis, Peter Alvaro, and Sumit Gulwani. Research for practice: Tracing and debugging distributed systems; programming by examples. *Communications of the ACM*, 60(7):46–49, 2017.

[8] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. Debugging distributed systems. *ACM Queue*, 14(2):91–110, 2016.

[9] Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, David A Rosenblueth, and Corentin Travers. Decentralized asynchronous crash-resilient runtime verification. In *CONCUR*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[10] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, 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, Joseph Tassarotti, M Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent Go code in Coq with Goose. In *CoqPL*, volume 2020, 2020.

[13] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[14] Himanshu Chauhan, Vijay K Garg, Aravind Natarajan, and Neeraj Mittal. A distributed abstraction algorithm for online predicate detection. In *International Symposium on Reliable Distributed Systems*, pages 101–110. IEEE, 2013.

[15] Horatiu Cirstea, Markus A Kuppe, Benjamin Loillier, and Stephan Merz. Validating Traces of Distributed Programs Against TLA+ Specifications. *arXiv preprint arXiv:2404.16075*, 2024.

[16] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[17] Biplob Debnath, Mohiuddin Solaimani, Muhammad Ali Gulzar, Nipun Arora, Cristian Lumezanu, Jianwu Xu, Bo Zong, Hui Bin Zhang, Guofei Jiang, and Latifur Khan. LogLens: A Real-Time Log Analysis System. *ICDCS*, pages 1052–1062, 2018.

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of CloudLab. In *USENIX ATC*, 2019.

[19] Envoy Proxy. https://www.envoyproxy.io/.

[20] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–35, 2012.

[21] etcd. https://coreos.com/etcd/.

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

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

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

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

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

[27] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

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

[29] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *EuroSys*, 2017.

[30] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[31] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. Runtime verification for decentralised and distributed systems. In *Lectures on Runtime Verification: Introductory and Advanced Topics*, pages 176–210. Springer, 2018.

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

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

[34] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *SOSP*, pages 1–17, 2015.

[35] Heidi Howard, Markus A Kuppe, Edward Ashton, Amaury Chamayou, and Natacha Crooks. Smart Casual Verification of CCF's Distributed Consensus and Consistency Protocols. *arXiv preprint arXiv:2406.17455*, 2024.

[36] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *OSDI*, 2022.

[37] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *HotOS*, 2017.

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

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

[40] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Vekataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An End-to-End Performance Tracing And Analysis System. In *SOSP*, 2017.

[41] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.

[42] Charles Edwin Killian, James W Anderson, Ryan Braud, Ranjit Jhala, and Amin M Vahdat. Mace: language support for building distributed systems. In *PLDI*, 2007.

[43] Kyle Kingsbury. Jepsen: Distribtued Systems Safety Research. https://jepsen.io/.

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

[45] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *SoCC*, 2018.

[46] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *SoCC*, 2019.

[47] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. In *OOPSLA*, 2023.

[48] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *OSDI*, 2014.

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

[50] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI*, 2008.

[51] Jacob R Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R Wilcox, and Xueyuan Zhao. Armada: low-effort verification of high-performance concurrent programs. In *PLDI*, 2020.

[52] Chang Lou, Peng Huang, and Scott Smith. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *NSDI*, Feburary 2020.

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

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

[55] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *SOSP*, 2019.

[56] Jonathan Mace and Rodrigo Fonseca. Universal context propagation for distributed system instrumentation. In *EuroSys*, 2018.

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

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

[59] Christopher S Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. Service-level fault injection testing. In *SoCC*, pages 388–402, 2021.

[60] Ellis Michael, Doug Woos, Thomas Anderson, Michael D Ernst, and Zachary Tatlock. Teaching rigorous distributed systems with efficient model checking. In *EuroSys*, pages 1–15, 2019.

[61] Francisco Neves, Nuno Machado, and José Pereira. Falcon: A Practical Log-Based Analysis Tool for Distributed Systems. *DSN*, pages 534–541, 2018.

[62] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. Use of formal methods at Amazon Web Services. See http://research.microsoft.com/en-us/um/people/lamport/tla/formal-methods-amazon.pdf, 2014.

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

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

[65] Lingzhi Ouyang, Yu Huang, Binyu Huang, Hengfeng Wei, and Xiaoxing Ma. Leveraging TLA+ Specifications to Improve the Reliability of the ZooKeeper Coordination Service. *arXiv preprint arXiv:2302.02703*, 2023.

[66] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *Proceedings of the ACM on Programming Languages*, 2:1–28, 2018.

[67] Oded Padon, Giuliano Losa, Shmuel Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. In *OOPSLA*, 2017.

[68] Oded Padon, Kenneth McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Interactive Verification of Parametrized Systems via Effectively Propositional Reasoning. In *PLDI*, 2016.

[69] Oded Padon, James R Wilcox, Jason R Koenig, Kenneth L McMillan, and Alex Aiken. Induction duality: primal-dual search for invariants. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022.

[70] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, 2020.

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

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

[73] Test failure : test_transfer_unexpected. https://github.com/RedisLabs/redisraft/issues/256, 2022.

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

[75] Github: RedisLabs/redisraft. https://github.com/RedisLabs/redisraft.

[76] Sundararajan Renganathan, Benny Rubin, Hyojoon Kim, Pier Luigi Ventre, Carmelo Cascone, Daniele Moro, Charles Chan, Nick McKeown, and Nate Foster. Hydra: Effective Runtime Network Verification. In *SIGCOMM*, 2023.

[77] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, volume 6, 2006.

[78] Michiel Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133–152, 1999.

[79] Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones, and Ali Basiri. *Chaos engineering*. O'Reilly Media, Incorporated, 2020.

[80] Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George C. Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing Faulty Executions of Distributed Systems. In *NSDI*, 2016.

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

[82] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[83] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *OSDI*, 2020.

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

[85] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD*, 2012.

[86] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[87] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model checking guided testing for distributed systems. In *EuroSys*, 2023.

[88] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, 2015.

[89] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. Planning for change in a formal verification of the raft consensus protocol. In *CPP 2016*, 2016.

[90] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *NSDI*, 2009.

[91] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *OSDI*, 2022.

[92] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *OSDI*, 2021.

[93] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable Runtime Verification of Shardable Networked Systems. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

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

[95] Pierre Zemb. Diving into ETCD's linearizable reads. https://pierrezemb.fr/posts/diving-into-etcd-linearizable/, Sep 2020.

[96] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. Check before you change: Preventing correlated failures in service updates. In *NSDI*, 2020.

[97] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. Static detection of silent misconfigurations with deep interaction analysis. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA), 2021.

[98] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. Viper: A Fast Snapshot Isolation Checker. In *EuroSys*, 2023.

[99] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. Foundationdb: A distributed key value store. In *SIGMOD*, 2021.

[100] Apache Zookeeper. https://zookeeper.apache.org/.

[101] 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.

A Proving apply ASAP correct

Protocol specifications in Ellsberg include an `apply_asap?` function (§3.3), whose output we use to avoid redundant exploration (§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 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 $e \notin \Sigma$, $apply(apply(s,e),\Sigma) \sqsupseteq apply(s,\Sigma)$.*

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

Condition A.1 ***Order Preserving.*** *The protocol should be such tha 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.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.3 ***Expansion.*** *For any event e and any state s that satisfies $C(s,e)$, $apply(s,e) \sqsupseteq s$*

Condition A.4 ***Reorder Safety.*** *For any event e,e', 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 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 $\Sigma' = e_1, e_2, \ldots, e_k$. To simplify the expressions, we denote $apply(s,e)$ by $e(s)$. We first prove that:

Lemma 1

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

Let $s' = e_i \circ \ldots \circ e_1(s)$, 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**:

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

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

B Mechanically Checking `apply_asap?`

We used Z3 to check the `apply_asap?` functions in our specification satisfy the four conditions A.2–A.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 ||
            (e.term == s.term &&
                (e.type == AppendEntriesResponse ||
                e.type == RequestVoteResponse &&
                    s.state != CANDIDATE) ||
                e.type == AppendEntryRequest
                    && s.state == FOLLOWER
                    && subseq(e.entries, s.log) ||
                e.type == RequestVoteRequest
                    && s.votedFor != None)
}
```

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

```
fn dominates(s1: State, s2: State) {
    //all other states are the
          same, except last_committed, last_processed
         , ackid of each log entry, and ack_ld_recv
    //s1's last_committed
          and last_processed are not smaller than s2
    //each entry's ack set of s2 is the subset of s1
    //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.
    return s1.accepted_epoch
          == s2.accepted_epoch && ... &&
           s1.last_committed >= s2.last_committed &&
           s1.last_processed >= s2.last_processed &&
           [for i in range(ENTRIES):
            subset(s2.Log[i].ack, s1.Log[i].ack)] &&
           (s1.zab_state == BROADCAST?
            subset(s2.ack_ld_recv, s1.ack_ld_recv):
            equal(s1.ack_ld_recv, s2.ack_ld_recv))
}

fn C(s: State, e: Event) {
    //we can apply e asap if this is an ACK
    //or it is an ACKLD in outdated epoch
         compared with current state's accepted_epoch
    //or ACKLD in current accepted_epoch but current
          state already has received quorum ACKLD
    return e.Type == ACK ||
           (e.Type == ACKLD &&
                (Epoch(e.Zxid) < s.accepted_epoch
                    || has_quorum(s.ack_ld_recv)))
}
```

C 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.

C.1 Specification Length

Table 3 shows lines of code for each portion of the Ellsberg specifications used in our evaluation. Similar to §4 we also report the length of specification used by Microsoft's CCF project for validating Raft [35] and an implementation derived ZooKeeper specification [38].

Component	Etcd	Redis Raft	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 3: *Lines of code in Ellsberg specification when compared to those used by existing tools. For baselines, we use the Raft specification from Microsoft CCF [35] Etcd and RedisRaft, and a recent implementation derived TLA+ specification [38] for ZooKeeper.*

C.2 Raft

We evaluated our approach and Ellsberg using two Raft implementations: Etcd [21], a distributed key-value store that is widely deployed, often as a core component of Kubernetes [39] and other orchestrators; and Redis Raft [32, 75], a module that adds consistent replication to the Redis key-value store. In this section, we describe the specification we used.

Protocol Specification. Our initial approach for producing a specification was to derive one from Ongaro's TLA+ Raft specification [63], since this closely mirrored the protocol described in the Raft paper [64], 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 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 [95]. 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.[2]

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 [26] 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 Redis Raft 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: Redis Raft 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. Etcd's 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 (§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 differ in when reconfiguration commands take effect.

C.3 Zookeeper Atomic Broadcast

Obtaining a specification for Zookeeper atomic broadcast (ZAB) [38] as implement by ZooKeeper [100] was a bit easier. While the TLA+ specification included in the Junqueira et al's [38] paper no longer reflects the implementation, a recent preprint [65] 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.

D Bug Descriptions

Below, we describe the bugs used in our evaluation (§6.2), their root-cause and how we detected them.

etcd-741 [25] is a bug that predates Etcd incorporating the linearizable read protocol we described in §C.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 acknowledgements from a quorum.

etcd-7280 [23] and **etcd-12133** [22] are bugs in Etcd's reconfiguration protocol. As we noted previously (Appendix C.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). Etcd 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 asynchronous, 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 [24] 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

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

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 [64, §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.

Etcd 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.

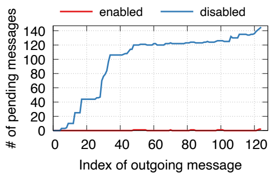
Zookeeper-1154 [101] is data inconsistency bug, where after leader election, nodes disagree on the log. In the leader election protocol implemented by ZooKeeper, a node needs to synchronize its log with all other nodes. To do so, the leader collects the largest entry ID (*ZXIDs*) from each follower, 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 [71] is also a reconfiguration bug: Redis Raft 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.

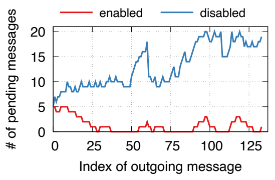
RedisRaft-19 [72] is a bug in Redis Raft's linearizable read implementation. Unlike Etcd-741, this bug occurs despite nodes asserting leadership before responding to clients, and is caused by the node being unaware of the current commit index. In Raft, a leader may not know what log entries were committed 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 [64, §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 Redis Raft's reconfiguration protocol: when adding a node, Redis Raft first issues a reconfiguration command to add a *non-voting* node that cannot participate in quorums. After this command has been committed, Redis Raft 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, Redis Raft acknowledges the client reconfiguration request once the non-voting reconfiguration command has been successfully replicated. Consequently, Redis Raft 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.

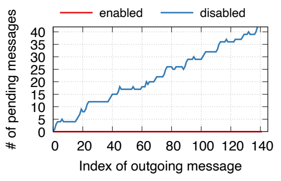
RedisRaft-256 [73] is a reconfiguration bug related to the unreported bug described above. This bug resulted in leader acknowledging a client reconfiguration command that adds a node before replicating it to a quorum, and can result in situation where the client reconfiguration command is never updated. 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.



(a) *Etcd*

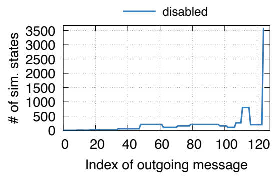


(b) *Zookeeper.*

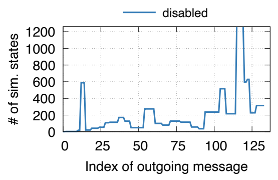


(c) *Redis Raft*

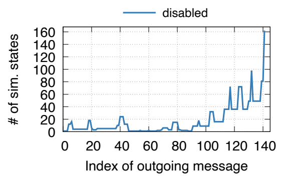
Figure 5: apply_asap? *avoids redundant exploration: graphs show pending messages in a 5-node cluster after checking the n^{th} outgoing message* apply_asap? enabled *and* disabled.



(a) *Etcd.*



(b) *Zookeeper.*



(c) *Redis Raft.*

Figure 6: 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.)*

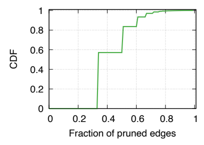
RedisRaft-52 [74] is a bug that leads to data-loss, and is due to a performance optimization implemented by Redis Raft where the last few log entries are stored in a cache. When a node appends an entry it adds it to both the cache and the underlying log. When reading an entry, the node first checks the cache, and only refers to the underlying log if it is not in the cache. A bug in Redis Raft 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.

E Effect of Ellsberg's Design Choices

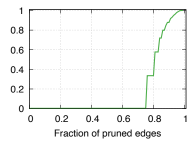
Our evaluation (§6) presented end-to-end performance results for Ellsberg. In this appendix, we evaluate the impact of `apply_asap?` and the `reachable` function on Ellsberg's performance.

E.1 Benefits from `apply_asap?`

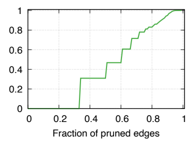
Next we evaluate the effect of using `apply_asap?` to avoid redundant exploration. As we discussed in §3.4.2,



(a) *Etcd.*



(b) *Zookeeper.*



(c) *Redis.*

Figure 7: *CDF of fraction of edges pruned by the* reachable *check (Listing 4 line 22). All experiments are run on a 5-node cluster using the mixed workload.*

`apply_asap?` is used when processing incoming messages (Listing 3 line 24) and during breadth first search (Listing 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 5, 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 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 5. 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 ∼444 states for Etcd, ∼478 states for Zookeeper, and ∼43 for Redis Raft.

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

E.2 The `reachable` function's impact

Ellsberg uses the specification's `reachable` function to to avoid exploring schedules that cannot lead to m-inducing simulation states (§3.4.1, Listing 4 line 22). As we explained in §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 7, 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 Redis Raft, 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.