IronFleet: Proving Practical Distributed Systems Correct

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch,
Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill

Microsoft Research

Abstract

Distributed systems are notorious for harboring subtle bugs. Verification can, in principle, eliminate these bugs a priori, but verification has historically been difficult to apply at full-program scale, much less distributed-system scale.

We describe a methodology for building practical and provably correct distributed systems based on a unique blend of TLA-style state-machine refinement and Hoare-logic verification. We demonstrate the methodology on a complex implementation of a Paxos-based replicated state machine library and a lease-based sharded key-value store. We prove that each obeys a concise safety specification, as well as desirable liveness requirements. Each implementation achieves performance competitive with a reference system. With our methodology and lessons learned, we aim to raise the standard for distributed systems from "tested" to "correct."

1. Introduction

Distributed systems are notoriously hard to get right. Protocol designers struggle to reason about concurrent execution on multiple machines, which leads to subtle errors. Engineers implementing such protocols face the same subtleties and, worse, must improvise to fill in gaps between abstract protocol descriptions and practical constraints, e.g., that real logs cannot grow without bound. Thorough testing is considered best practice, but its efficacy is limited by distributed systems' combinatorially large state spaces.

In theory, formal verification can categorically eliminate errors from distributed systems. However, due to the complexity of these systems, previous work has primarily focused on formally specifying [4, 13, 27, 41, 48, 64], verifying [3, 52, 53, 60, 61], or at least bug-checking [20, 31, 69] distributed protocols, often in a simplified form, without extending such formal reasoning to the implementations. In principle, one can use model checking to reason about the correctness of both protocols [42, 60] and implementations [46, 47, 69]. In practice, however, model checking is incomplete—the accuracy of the results depends on the accuracy of the model—and does not scale [4].

SOSP'15, October 4–7, 2015, Monterey, CA.

ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815428

This paper presents IronFleet, the first methodology for automated machine-checked verification of the safety and liveness of non-trivial distributed system implementations. The IronFleet methodology is practical: it supports complex, feature-rich implementations with reasonable performance and a tolerable proof burden.

Ultimately, IronFleet guarantees that the implementation of a distributed system meets a high-level, centralized specification. For example, a sharded key-value store acts like a key-value store, and a replicated state machine acts like a state machine. This guarantee categorically rules out race conditions, violations of global invariants, integer overflow, disagreements between packet encoding and decoding, and bugs in rarely exercised code paths such as failure recovery [70]. Moreover, it not only rules out bad behavior, it tells us exactly how the distributed system will behave at all times.

The IronFleet methodology supports proving both *safety* and *liveness* properties of distributed system implementations. A safety property says that the system cannot perform incorrect actions; e.g., replicated-state-machine linearizability says that clients never see inconsistent results. A liveness property says that the system eventually performs a useful action, e.g., that it eventually responds to each client request. In large-scale deployments, ensuring liveness is critical, since a liveness bug may render the entire system unavailable.

IronFleet takes the verification of safety properties further than prior work (§9), mechanically verifying two full-featured systems. The verification applies not just to their protocols but to actual imperative implementations that achieve good performance. Our proofs reason all the way down to the bytes of the UDP packets sent on the network, guaranteeing correctness despite packet drops, reorderings, or duplications.

Regarding liveness, IronFleet breaks new ground: to our knowledge, IronFleet is the first system to mechanically verify liveness properties of a practical protocol, let alone an implementation.

IronFleet achieves comprehensive verification of complex distributed systems via a methodology for structuring and writing proofs about them, as well as a collection of generic verified libraries useful for implementing such systems. Structurally, IronFleet's methodology uses a *concurrency containment* strategy (§3) that blends two distinct verification styles within the same automated theorem-proving framework, preventing any semantic gaps between them. We use TLA-style state-machine refinement [36] to reason about protocol-level concurrency, ignoring implementation complexities, then use Floyd-Hoare-style imperative verification [17, 22] to reason about those complexities while ignoring concurrency. To simplify reasoning about concurrency, we impose a machine-checked *reduction-enabling obligation* on the implementation (§3.6). Finally, we structure our protocols using *always-enabled actions* (§4.2) to greatly simplify liveness proofs.

To facilitate writing proofs about distributed systems, we have developed techniques for writing automation-friendly invariant proofs (§3.3), as well as disciplines and tool improvements for coping with prover limitations (§6). For liveness proofs, we have constructed an embedding of TLA (§4.1) in our automated verification framework that includes heuristics for reliably unleashing the power of automated proving.

To help developers, we have built general-purpose verified libraries for common tasks, such as packet parsing and marshalling, relating concrete data structures to their abstract counterparts, and reasoning about collections. We have also written a verified library of 40 fundamental TLA rules useful for writing liveness proofs.

To illustrate IronFleet's applicability, we have built and proven correct two rather different distributed systems: IronRSL, a Paxos-based [35] replicated-state-machine library, and IronKV, a sharded key-value store. All IronFleet code is publicly available [25].

IronRSL's implementation is complex, including many details often omitted by prior work; e.g., it supports state transfer, log truncation, dynamic view-change timeouts, batching, and a reply cache. We prove complete functional correctness and its key liveness property: if the network is eventually synchronous for a live quorum of replicas, then a client repeatedly submitting a request eventually receives a reply.

Unlike IronRSL, which uses distribution for reliability, IronKV uses it for improved throughput by moving "hot" keys to dedicated machines. For IronKV, we prove complete functional correctness and an important liveness property: if the network is fair then the reliable-transmission component eventually delivers each message.

While verification rules out a host of problems, it is not a panacea (§8). IronFleet's correctness is not absolute; it relies on several assumptions (§2.5). Additionally, verification requires more up-front development effort: the automated tools we use fill in many low-level proof steps automatically (§6.3.1), but still require considerable assistance from the developer (§6.3.2). Finally, we focus on verifying newly written code in a verification-friendly language (§2.2), rather than verifying existing code.

In summary, this paper makes the following contributions:

- We demonstrate the feasibility of mechanically verifying that *practical* distributed implementations, i.e., functionally complete systems with reasonable performance, match simple, logically centralized specifications.
- We describe IronFleet's novel methodology for uniting TLA-style refinement with Floyd-Hoare logic within a single automated verification framework.
- We provide the first machine-verified liveness proofs of non-trivial distributed systems.
- We describe engineering disciplines and lessons for verifying distributed systems.

2. Background and Assumptions

We briefly describe the existing verification techniques that IronFleet draws upon, as well as our assumptions.

2.1 State Machine Refinement

State machine refinement [1, 18, 34] is often used to reason about distributed systems [4, 27, 41, 48, 52, 64]. The developer describes the desired system as a simple abstract state machine with potentially infinitely many states and non-deterministic transition predicates. She then creates a series of increasingly complex (but still declarative) state machines, and proves that each one *refines* the one "above" it (Figure 1). State machine L refines H if each of L's possible *behaviors*, i.e., each (potentially infinite) sequence of states the machine may visit, corresponds to an equivalent behavior of H. To gain the benefits of abstraction this approach provides, the developer must choose the layer abstractions intelligently, a subtle choice needed for each new context.

State machine refinement in a distributed-system context (e.g., TLA-style refinement) typically considers declarative specifications, not imperative code. PlusCal [37] attempts to bridge this gap, but has only been used for tiny programs.

2.2 Floyd-Hoare Verification

Many program verification tools support Floyd-Hoare style [17, 22] first-order predicate logic reasoning about imperative programs. In other words, they allow the programmer to annotate a program with assertions about the program's state, and the verifier checks that the assertions hold true for all possible program inputs. For example, the code in Figure 2 asserts a condition about its input via a *precondition* and asserts a condition about its output via a *postcondition*.

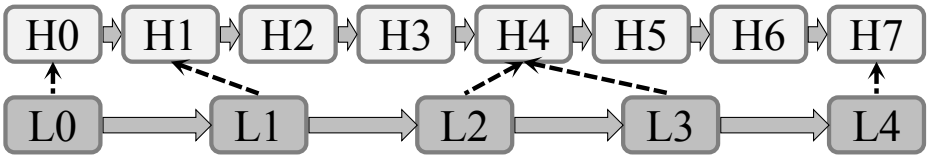

Figure 1. State Machine Refinement. *The low-level state machine behavior L0...L4 refines the high-level one H0...H7 because each low-level state corresponds to a high-level state. For each correspondence, shown as a dashed line, the two states must satisfy the spec's refinement conditions. Low-level step L0→L1, as is typical, maps to one high-level step H0→H1. However, low-level steps can map to zero (L2→L3) or several (L3→L4) high-level steps.*

```
method halve(x:int) returns (y:int)
  requires x > 0;
  ensures  y < x;
{
  y := x / 2;
}
```

Figure 2. *Simple Floyd-Hoare verification example.*

As in our previous work [21], we use Dafny [39], a high-level language that automates verification via the Z3 [11] SMT solver. This enables it to fill in many low-level proofs automatically; for example, it easily verifies the program in Figure 2 for all possible inputs `x` without any assistance.

However, many proposition classes are not decidable in general, so Z3 uses heuristics. For example, propositions involving universal quantifiers (∀) and existential quantifiers (∃) are undecidable. Thus, it is possible to write correct code in Dafny that the solver nevertheless cannot prove automatically. Instead, the developer may insert annotations to guide the verifier's heuristics to a proof. For instance, the developer can write a *trigger* to cue the verifier as to which values to instantiate a quantified variable with [12].

Once a program verifies, Dafny compiles it to C# and has the .NET compiler produce an executable. Other languages (e.g., C++) are currently unsupported, but it would likely be possible to compile Dafny to them. Our previous work [21] shows how to compile Dafny to verifiable assembly to avoid depending on the Dafny compiler, .NET, and Windows.

Like most verification tools, Dafny only considers one single-threaded program, not a collection of concurrently executing hosts. Indeed, some verification experts estimate that the state-of-the-art in concurrent program verification lags that of sequential verification by a decade [51].

2.3 Reduction

Given a fine-grained behavior from a real concurrent system, we can use *reduction* [40] to convert it to an equivalent behavior of coarse-grained steps, simplifying verification. Crucially, two steps can swap places in the behavior if swapping them has no effect on the execution's outcome.

Reduction is typically used in the context of shared-memory concurrent programs [9, 14, 33] and synchronization primitives [65]. Applying reduction requires identifying all of the steps in the system, proving commutativity relationships among them, and applying these

relationships to create an equivalent behavior with a more useful form. We tackle these challenges in the context of distributed systems in §3.6.

2.4 Temporal Logic of Actions (TLA)

Temporal logic [54] and its extension TLA [34] are standard tools for reasoning about safety and liveness. Temporal logic *formulas* are predicates about the system's current and future states. The simplest type of formula ignores the future; e.g., in a lock system, a formula P could be "host h holds the lock now." Other formulas involve the future; e.g., $\Diamond P$ means P eventually holds, and $\Box P$ means P holds now and forever. For example, the property $\forall h \in \mathsf{Hosts} : \Box\Diamond P$ means that for any host, it is always true that h will eventually hold the lock.

TLA typically considers abstract specifications, not imperative code. Furthermore, a naïve embedding of TLA can often pose problems for automated verification. After all, each $\Box$ involves a universal quantifier and each $\Diamond$ involves an existential quantifier. Since Z3 needs heuristics to decide propositions with quantifiers (§2.2), it can fail due to inadequate developer annotations. We address this in §4.1.

2.5 Assumptions

Our guarantees rely on the following assumptions.

A small amount of our code is assumed, rather than proven, correct. Thus, to trust the system, a user must read this code. Specifically, the spec for each system is trusted, as is the brief main-event loop described in §3.7.

We do not assume reliable delivery of packets, so the network may arbitrarily delay, drop, or duplicate packets. We *do* assume the network does not tamper with packets, and that the addresses in packet headers are trustworthy. These assumptions about message integrity are easy to enforce within, say, a datacenter or VPN, and could be relaxed by modeling the necessary cryptographic primitives to talk about keys instead of addresses [21].

We assume the correctness of Dafny, the .NET compiler and runtime, and the underlying Windows OS. Previous work [21] shows how to compile Dafny code into verifiable assembly code to avoid these dependencies. We also rely on the correctness of the underlying hardware.

Our liveness properties depend on further assumptions. For IronRSL, we assume a quorum of replicas run their respective main loops with a minimum frequency, never running out of memory, and the network eventually delivers messages synchronously among them; more details are in §5.1.4. For IronKV, we assume that each host's main loop executes infinitely often and that the network is fair, i.e., a message sent infinitely often is eventually delivered.

3. The IronFleet Verification Methodology

IronFleet organizes a distributed system's implementation and proof into layers (Figure 3) to avoid the intermingling of subtle distributed protocols with implementation complexity. At the top (§3.1), we write a simple spec for the system's behavior. We then write an abstract distributed protocol layer (§3.2) and use TLA-style techniques to prove that it refines the spec layer (§3.3). Then we write an imperative implementation layer to run on each host (§3.4) and prove that, despite the complexities introduced when writing real systems code, the implementation correctly refines the protocol layer (§3.5).

To avoid complex reasoning about interleaved execution of low-level operations at multiple hosts, we use a *concurrency containment* strategy: the proofs above assume that every implementation step performs an atomic protocol step. Since the real implementation's execution is not atomic, we use a reduction argument (§3.6) to show that a proof assuming

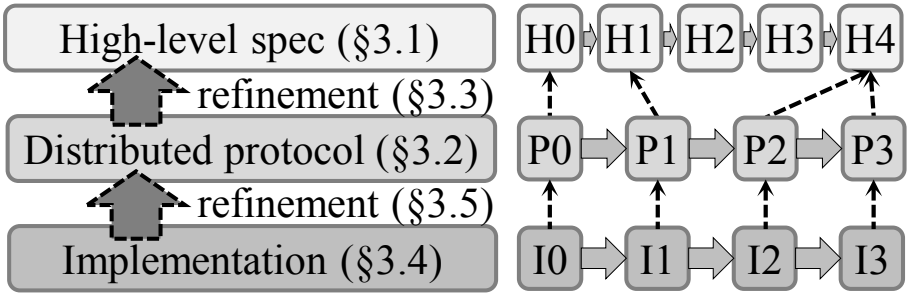

Figure 3. Verification Overview. *IronFleet divides a distributed system into carefully chosen layers. We use TLA style verification to prove that any behavior of the protocol layer (e.g., P0...P3) refines some behavior of the high-level spec (e.g., H0...H4). We then use Floyd-Hoare style to prove that any behavior of the implementation (e.g., I0...I3) refines a behavior of the protocol layer.*

atomicity is equally valid as a proof for the real system. This argument requires a mechanically verified property of the implementation, as well as a small paper-only proof about the implications of the property.

§4 extends this methodology to prove liveness properties.

3.1 The High-Level Spec Layer

What does it mean for a system to be *correct*? One can informally enumerate a set of properties and hope they are sufficient to provide correctness. A more rigorous way is to define a *spec*, a succinct description of every allowable behavior of the system, and prove that an implementation always generates outputs consistent with the spec.

With IronFleet, the developer writes the system's spec as a state machine: starting with some initial state, the spec succinctly describes how that state can be transformed. The spec defines the state machine via three *predicates*, i.e., functions that return true or false. `SpecInit` describes acceptable starting states, `SpecNext` describes acceptable ways to move from an old to a new state, and `SpecRelation` describes the required conditions on the relation between an implementation state and its corresponding abstract state. For instance, in Figure 3, `SpecInit` constrains what H0 can be, `SpecNext` constrains steps like H0→H1 and H1→H2, and `SpecRelation` constrains corresponding state pairs like (I1, H1) and (I2, H4). To avoid unnecessary constraints on implementations of the spec, `SpecRelation` should only talk about the externally visible behavior of the implementation, e.g., the set of messages it has sent so far.

As a toy example, the spec in Figure 4 describes a simple distributed lock service with a single lock that passes amongst the hosts. It defines the system's state as a history: a sequence of host IDs such that the nth host in the sequence held the lock in epoch n. Initially, this history contains one valid host. The system can step from an old to a new state by appending a valid host to the history. An implementation is consistent with the spec if all lock messages for epoch n come from the nth host in the history.

By keeping the spec simple, a skeptic can study the spec to understand the system's properties. In our example, she can easily conclude that the lock is never held by more than one host. Since the spec captures all permitted system behaviors, she can later verify additional properties of the implementation just by verifying they are implied by the spec.

```
datatype SpecState = SpecState(history:seq<HostId>)

predicate SpecInit(ss:SpecState)
{
     |ss.history| == 1
  && ss.history[0] in AllHostIds()
}

predicate SpecNext(ss_old:SpecState, ss_new:SpecState)
{
  exists new_holder :: new_holder in AllHostIds()
                    && ss_new.history == ss_old.history + [new_holder]
}

predicate SpecRelation(is:ImplState, ss:SpecState)
{
  forall p :: p in is.sentPackets && p.msg.lock? ==>
              p.src == ss.history[p.msg.epoch]
}
```

Figure 4. *A toy lock specification.*

3.2 The Distributed-Protocol Layer

At the untrusted distributed-protocol layer, the IronFleet methodology introduces the concept of independent hosts that communicate only via network messages. To manage this new complexity, we keep this layer simple and abstract.

In more detail, we formally specify, in Dafny (§2.2), a distributed system state machine. This state machine consists of N host state machines and a collection of network packets. In each step of the distributed system state machine, one host's state machine takes a step, allowing it to atomically read messages from the network, update its state, and send messages to the network; §3.6 relaxes this atomicity assumption.

The developer must specify each host's state machine: the structure of the host's local state, how that state is initialized (`HostInit`), and how it is updated (`HostNext`). IronFleet reduces the developer's effort in the following three ways.

First, we use a simple, abstract style for the host state and network interface; e.g., the state uses unbounded mathematical integers (ignoring overflow issues), unbounded sequences of values (e.g., tracking all messages ever sent or received), and immutable types (ignoring memory management and heap aliasing). The network allows hosts to send and receive high-level, structured packets, hence excluding the challenges of marshalling and parsing from this layer.

Second, we use a declarative predicate style. In other words, `HostNext` merely describes how host state can change during each step; it gives no details about how to effect those changes, let alone how to do so with good performance.

Third, from the protocol's perspective, each of the steps defined above takes place atomically, greatly simplifying the proof that the protocol refines the spec layer (§3.3). In §3.6, we connect this proof assuming atomicity to a real execution.

Continuing our lock example, the protocol layer might define a host state machine as in Figure 5. During the distributed system's initialization of each host via `HostInit`, exactly one host is given the lock via the `held` parameter. The `HostNext` predicate then says that a host may step from an old to a new state if the new state is the result of one of two

```
datatype Host = Host(held:bool,epoch:int)

predicate HostInit(s:Host, id:HostId, held:bool)
{
     s.held==held
  && s.epoch==0
}

predicate HostGrant(s_old:Host, s_new:Host, spkt:Packet)
{
     s_old.held
  && !s_new.held
  && spkt.msg.transfer?
  && spkt.msg.epoch == s_old.epoch+1
}

predicate HostAccept(s_old:Host, s_new:Host, rpkt:Packet, spkt:Packet)
{
     !s_old.held
  && s_new.held
  && rpkt.msg.transfer?
  && s_new.epoch == rpkt.msg.epoch == spkt.msg.epoch
  && spkt.msg.lock?
}

predicate HostNext(s_old:Host, s_new:Host, rpkt:Packet, spkt:Packet)
{
     HostGrant(s_old, s_new,spkt)
  || HostAccept(s_old, s_new, rpkt, spkt)
}
```

Figure 5. *Simplified host state machine for a lock service.*

actions, each represented by its own predicate. The two actions are giving away the lock (`HostGrant`) and receiving the lock from another host (`HostAccept`). A host may grant the lock if in the old state it holds the lock, and if in the new state it no longer holds it, and if the outbound packet (`spkt`) represents a transfer message to another host. Accepting a lock is analogous.

3.3 Connecting the Protocol Layer to the Spec Layer

The first major theorem we prove about each system is that the distributed protocol layer refines the high-level spec layer. In other words, given a behavior of IronFleet's distributed system in which N hosts take atomic protocol steps defined by the `HostNext` predicate, we provide a corresponding behavior of the high-level state machine spec.

We use the standard approach to proving refinement, as illustrated in Figure 3. First, we define a *refinement function* `PRef` that takes a state of the distributed protocol state machine and returns the corresponding state of the centralized spec. We could use a relation instead of a function, but the proof is easier with a function [1]. Second, we prove that `PRef` of the initial state of the distributed protocol satisfies `SpecInit`. Third, we prove that if a step of the protocol takes the state from `ps_old` to `ps_new`, then there exists a legal sequence of high-level spec steps that goes from `PRef(ps_old)` to `PRef(ps_new)`.

Unlike previous refinement-based work (§2.1), we use a language, Dafny [39], designed for automated theorem proving. This reduces but does not eliminate the human proof effort

```
lemma ReplyToReq(reply:MessageReply, behavior:map<int,HostState>, step:nat)
        returns (req:MessageRequest)
  requires IsValidBehaviorUpTo(behavior, step);
  requires reply in behavior[step].network;
  ensures  req in behavior[step].network;
  ensures  Matches(req, reply);
{
  assert step > 0; // because a packet was sent
  if !(reply in behavior[step-1].network) {
    req := OnlyExecReplies(behavior, step-1);
  } else { // apply induction
    req := ReplyToReq(behavior, step-1, reply);
  }
}
```

Figure 6. *Establishing an invariant with implicit quantifiers.*

required (§6.3). Since we also verify our implementation in Dafny (§3.5), we avoid any semantic gaps between the implementation's view of the protocol and the protocol we actually prove correct.

The challenge of proving the protocol-to-spec theorem comes from reasoning about global properties of the distributed system. One key tool is to establish *invariants*: predicates that should hold throughout the execution of the distributed protocol. In the lock example, we might use the invariant that the lock is either held by exactly one host or granted by one in-flight lock-transfer message. We can prove this invariant inductively by showing that every protocol step preserves it. Showing refinement of the spec is then simple.

Identifying the right invariants for a given protocol requires a deep understanding of the protocol, but it is a skill one develops with experience (§6).

Invariant quantifier hiding. Many useful invariants, like "For every reply message sent, there exists a corresponding request message sent," involve quantifiers. Unfortunately, such quantifiers pose problems for verifiers (§2.2). We have thus adopted a style we call *invariant quantifier hiding*: we prove some invariants involving quantifiers without explicitly exposing those quantifiers to the verifier. The key is to establish the invariant with a proof that explicitly instantiates all bound variables. For each universal quantifier in the invariant that does not succeed an existential quantifier, the quantified variable is an input parameter of the proof. For each existential quantifier in the invariant, the quantified variable is an output parameter. For instance, the invariant from the beginning of this paragraph could be proved with Figure 6.

It is easy to write this proof because we must prove it only for a specific reply message, not all of them. As illustrated above, there are only two cases to consider: (1) the reply message was just generated, in which case we only have to consider the last action taken, or (2) the reply message was already present in the previous step, in which case we can complete the proof by induction on `step`.

It is also easy to *use* this proof because instead of stating a fact about the existence of a request message, it explicitly provides a witness to that existence. Typically, a developer only needs to prove the invariant for a specific reply message; this form lets her establish precisely that fact. If the developer needs the universally-quantified version, she can establish it by invoking the invariant's proof in a loop.

3.4 The Implementation Layer

Unlike in the declarative protocol layer, in the implementation layer the developer writes single-threaded, imperative code to run on each host. This code must cope with all of the ugly practicalities we abstracted away in the protocol layer. For instance, it must handle real-world constraints on how hosts interact: since network packets must be bounded-sized byte arrays, we need to prove the correctness of our routines for marshalling high-level data structures into bytes and for parsing those bytes. We also write the implementation with performance in mind, e.g., using mutable arrays instead of immutable sequences and using `uint64`s instead of infinite-precision integers. The latter requires us to prove the system correct despite the potential for integer overflow.

Dafny does not natively support networking, so we extend the language with a trusted UDP specification that exposes `Init`, `Send`, and `Receive` methods. For example, `Send` expects an IP address and port for the destination and an array of bytes for the message body. When compiled, calls to these Dafny methods invoke the .NET UDP network stack. `Send` also automatically inserts the host's correct IP address, satisfying our assumption about packet headers in §2.5.

The network interface maintains a ghost variable (i.e., a variable used only for verification, not execution) that records a "journal" of every `Send` and `Receive` that takes place, including all of the arguments and return values. We use this journal when proving properties of the implementation (§3.5).

3.5 Connecting the Implementation to the Protocol

The second major theorem we prove about each IronFleet system is that the implementation layer correctly refines the protocol. To do this, we prove that even though the implementation operates on concrete local state, which uses heap-dependent, bounded representations, it is still a refinement of the protocol layer, which operates on abstract types and unbounded representations.

First, we prove that the host implementation refines the host state machine described in the protocol layer. This refinement proof is analogous to the one in §3.3, though simplified by the fact that each step in the implementation corresponds to exactly one step of the host state machine. We define a refinement function `HRef` that maps a host's implementation state to a host protocol state. We prove that the code `ImplInit` to initialize the host's state ensures `HostInit(HRef(hs))`, and that the code `ImplNext` to execute one host step ensures `HostNext(HRef(hs_old),HRef(hs_new))`.

Then, we use this to prove that a distributed system comprising N host implementations, i.e., what we actually intend to run, refines the distributed protocol of N hosts. We use a refinement function `IRef` that maps states of the distributed implementation to states of the distributed protocol. The refinement proof is largely straightforward because each step of the distributed implementation in which a host executes `ImplNext` corresponds to one step of the distributed protocol where a host takes a `HostNext` step. The difficult part is proving that the network state in the distributed system implementation refines the network state in the protocol layer. Specifically, we must prove that every send or receive of a UDP packet corresponds to a send or receive of an abstract packet. This involves proving that when host A marshals a data structure into an array of bytes and sends it to host B, B parses out the identical data structure.

The last major theorem we prove is that the distributed implementation refines the abstract centralized spec. For this, we use the refinement functions from our two major refinement theorems, composing them to form our final refinement function `PRef(IRef(·))`. The

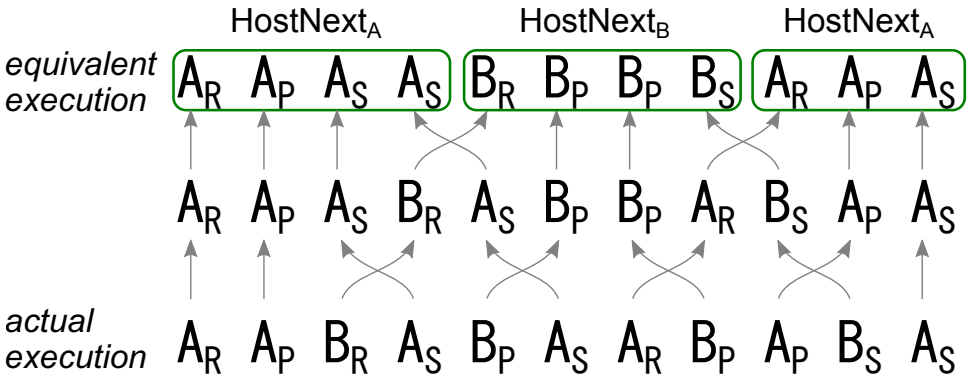

Figure 7. Reduction. *In the real execution behavior, the send (S), receive (R), and local processing (P) steps at hosts A and B are fully interleaved. However, certain steps commute to yield an equivalent behavior. Since we impose constraints on the structure of the implementation's event handlers (Figure 8), we can commute steps until all of the implementation-level steps in a given host's event handler (circled) are contiguous. This* reduced *behavior then admits a direct refinement to the distributed protocol layer.*

key part of this proof is establishing that the specified relation conditions hold, i.e., that for all implementation states `is`, `SpecRelation(is, IRef(PRef(is)))` holds.

3.6 Abstracting Non-Atomicity via Reduction

Sections 3.1–3.5 describe a mechanically verified proof structure that assumes that every implementation step performs an atomic protocol step. However, the implementation's event handler is not atomic: while one host receives packets, computes locally, and sends packets, other hosts do the same concurrently, leading to arbitrary interleavings of these low-level operations. To bridge this gap, we use a "reduction" argument (§2.3). Reduction is typically used to reason about threads and processes sharing a single machine, but we apply it to reasoning about distributed systems. Although Dafny does not provide a *general* mechanism for reasoning about reduction, we are still able to use Dafny to enforce an *obligation* on the implementation that enables reduction. A machine-checked proof that this obligation enables reduction is future work; instead, we sketch an informal argument here.

Hosts are unable to see others' state except indirectly by observing the packets they send. Thus, it is possible to take a behavior of the system, representing the order in which events really occurred, and posit an alternate order in which (1) each host receives the same packets in the same order, (2) packet send ordering is preserved, (3) packets are never received before they are sent, and (4) the ordering of operations on any individual host is preserved. Any proof of correctness assuming such an ordering implies a proof for the original behavior, since only the externalized behavior of the system, the content and ordering messages sent, matters.

Figure 7 shows an example of such reordering. We start with the real behavior at the bottom and reorder until we reach the behavior at the top. For instance, we can reorder A's first send before B's first receive because we know its contents cannot have depended on B's receive. The top behavior has no interleavings between different hosts' `HostNext` steps, and thus is a legal behavior in which we have proved correctness. Thus, the correctness proof also applies to the real behavior.

As a result, we can always reduce a real execution behavior to a sequence of atomic steps via such reorderings if we constrain the implementation to, in any given step, perform all

```
method Main() {
  var s := ImplInit();
  while (true)
    invariant ImplInvariant(s);
  {
    ghost var journal_old := get_event_journal();
    ghost var ios_performed:seq<IoEvent>;
    s, ios_performed := ImplNext(s);
    assert get_event_journal() == journal_old + ios_performed;
    assert ReductionObligation(ios_performed);
  }
}
```

Figure 8. *Mandatory host event-handler loop.*

of its receives before all its sends. We call this a *reduction-enabling obligation*, which we use Dafny to enforce (§3.7). With this obligation, we ensure that our proof of correctness assuming atomicity is equally valid as a proof of correctness for the real system.

One complication is that when a host performs a time-dependent operation like reading its clock, it creates a causal constraint even without communication with other hosts. This is because the clock represents an imperfect sample from a globally shared reality. Thus, the reduction-enabling obligation is extended as follows: A step may perform at most one time-dependent operation, i.e., at most one clock read, blocking receive, or non-blocking receive that returns no packets. The step must perform all receives before this time-dependent operation, and all sends after it.

3.7 Trusted Code

Nearly all IronFleet code is verified using the above methodology, so there are only a few lines of code and proof assertions that a user must read to gain confidence in the system. First, she must read the high-level centralized spec to understand what is being guaranteed. Second, she must read the assertion, but not the proof of the assertion, that if each host in a distributed system runs `ImplInit` followed by a loop of `ImplNext`, then there exists a corresponding abstract behavior of the centralized spec. Third, she must read the top-level main host routine (Figure 8) to convince herself that each host runs `ImplInit` and `ImplNext`. This code also ensures that each host step meets its reduction-enabling constraint by using the journal of externally visible events from §3.4.

4. Verifying Liveness

§3 describes the high-level spec as a state machine. Such a spec says what the implementation *must not* do: it must never deviate from the state machine's behavior. However, it is also useful to specify what the implementation *must* do; properties of this form are called *liveness* properties. For example, we might specify that the lock implementation eventually grants the lock to each host (Figure 9). Thus, a spec will typically include not just a state machine but also liveness properties.

Some researchers have proposed heuristics for detecting and quashing likely sources of liveness violations [31, 66], but it is better to definitively *prove* their absence. With such a proof, we do not have to reason about, e.g., deadlock or livelock; such conditions and any others that can prevent the system from making progress are provably ruled out.

Liveness properties are much harder to verify than safety properties. Safety proofs need only reason about two system states at a time: if each step between two states preserves

```
predicate LockBehaviorFair(b:map<int,SpecState>)
{
  forall h:Host, i:int :: h in AllHostIds() && i >= 0
    ==> exists j :: j >= i && h == last(b[j].history)
}
```

Figure 9. *Desired liveness property for the lock service.*

the system's safety invariants, then we can inductively conclude that all behaviors are safe. Liveness, in contrast, requires reasoning about infinite series of system states. Such reasoning creates challenges for automated theorem provers (§2.4), often causing the prover to time out rather than return a successful verification or a useful error message.

With IronFleet, we address these challenges via a custom TLA embedding in Dafny that focuses the prover's efforts in fruitful directions. We then use our TLA embedding to build a library of fundamental TLA proof rules verified from first principles. This library is a useful artifact for proving liveness properties of arbitrary distributed systems: its rules allow both the human developer and Dafny to operate at a high level by taking large proof steps with a single call to a lemma from the library. Finally, by structuring our protocols with *always-enabled actions*, we significantly simplify the task of proving liveness properties.

4.1 TLA Embedding and Library

As discussed in §2.4, TLA [34] is a standard tool for reasoning about liveness. IronFleet embeds TLA in Dafny by modeling a TLA behavior, an infinite sequence of system states, as a mapping B from integers to states, where $B[0]$ is the initial state and $B[i]$ is the ith subsequent state. A liveness property is a constraint on the behavior of the state machine. For example, Figure 9 says that for every host h, there is always a later time when h will hold the lock.

Our embedding hides key definitions from the prover except where truly needed, and instead provides verified lemmas that relate them to one another. For example, we represent temporal logic formulas as opaque objects (i.e., objects Dafny knows nothing about) of type `temporal`, and TLA transformations like $\Box$ as functions that convert `temporal` objects to `temporal` objects.

Of course, in some contexts we actually do need to reason about the internal meaning of $\Box$ and $\Diamond$. State-of-the-art SMT solvers like Z3 do not yet provide decision procedures for temporal operators like $\Box$ and $\Diamond$ directly. However, we can encode these operators using explicit quantification over steps ($\Box$ universally quantifies over all future steps, while $\Diamond$ existentially quantifies over some future step). We can then provide the SMT solver with heuristics to control these quantifiers using the solver's support for triggers [12], as discussed in §2.2. One simple heuristic proved effective in many situations: when the solver is considering a future step j for one formula, such as $\Diamond Q$, the heuristic requests that the solver also consider j as a candidate step for other formulas starting with $\Box$ or $\Diamond$, such as $\Box P$ and $\Diamond(P \wedge Q)$. This allows the solver to automatically prove formulas like $(\Diamond Q) \wedge (\Box P) \Longrightarrow \Diamond(P \wedge Q)$.

This heuristic is effective enough to automatically prove 40 fundamental TLA proof rules, i.e., rules for deriving one formula from other formulas [34]. The heuristic allows us to prove complicated rules efficiently; e.g., we stated and proved Lamport's INV1 rule about invariants in only 27 lines of Dafny, and his WF1 rule about fairness in only 16 lines.

Our liveness proofs use these fundamental proof-rule lemmas to justify temporal formula transformations. For instance, as we discuss in §4.4, a liveness proof can usually prove most of its steps by repeatedly invoking the WF1 rule.

4.2 Always-Enabled Actions

To achieve liveness, our protocol must satisfy *fairness properties*. That is, it must ensure that each action, e.g., `HostGrant` or `HostAccept`, occurs in a timely fashion.

Lamport [36] suggests that such properties take the form "if action A becomes always *enabled*, i.e., always possible to do, the implementation must eventually do it." However, having terms of this form in verified code is problematic. If the fairness property is a complex formula, it can be difficult to characterize the set of states from which the action is possible. This difficulty complicates both proving that the fairness property is sufficient to ensure liveness properties, and proving that the protocol has the fairness property.

Thus, we instead adopt *always-enabled actions*; i.e., we only use actions that are always possible to do. For instance, we would not use `HostGrant` from Figure 5 since it is impossible to perform if you do not hold the lock. Instead, we might use "if you hold the lock, grant it to the next host; otherwise, do nothing", which can always be done.

Our approach deviates from Lamport's standard fairness formulas, which means it can admit specifications that are not machine closed [36]. Machine closure ensures that liveness conditions do not combine with safety conditions to create an unimplementable spec, such as that the implementation must both grant a lock (to be fair) and not grant a lock (to be safe, because it does not hold the lock). Fortunately, machine closure is no concern in IronFleet: the existence of an implementation that meets a fairness property is itself proof that the property does not prevent implementation!

4.3 Proving Fairness Properties

Following IronFleet's general philosophy of having the implementation layer deal only with implementation complexities, we put the burden of satisfying fairness properties on the protocol layer. The implementation satisfies the properties automatically since its main method implements `HostNext`.

The mandatory structure from Figure 8 ensures that `HostNext` runs infinitely often. So, all we must prove is that if `HostNext` runs infinitely often, then each action occurs infinitely often. We do this by having `HostNext` be a scheduler that guarantees each action occurs regularly.

One way to do this is to use a simple round-robin scheduler. We currently have proofs in our library that if `HostNext` is a round-robin scheduler that runs infinitely often, then each action runs infinitely often. Furthermore, if the main host method runs with frequency F (expressed, e.g., in times per second), then each of its n actions occurs with frequency F/n.

4.4 Liveness Proof Strategies

Most of a liveness proof involves demonstrating that if some condition C_i holds then eventually another condition C_{i+1} holds. By chaining such proofs together, we can prove that if some assumed initial condition C_0 holds then eventually some useful condition C_n holds. For instance, in IronRSL, we prove that if a replica receives a client's request, it eventually suspects its current view; if it suspects its current view, it eventually sends a message to the potential leader of a succeeding view; and, if the potential leader receives a quorum of suspicions, it eventually starts the next view.

Most steps in this chain require an application of a variant of Lamport's WF1 rule [34]. This variant involves a starting condition C_i, an ending condition C_{i+1}, and an always-enabled

action predicate Action. It states that C_i leads to C_{i+1} if the following three requirements are met:

1. If C_i holds, it continues to hold as long as C_{i+1} does not.
2. If a transition satisfying Action occurs when C_i holds, it causes C_{i+1} to hold.
3. Transitions satisfying Action occur infinitely often.

We use this in Dafny as follows. Suppose we need a lemma that shows C_i leads to C_{i+1}. We first find the action transition Action intended to cause this. We then establish each of requirements 1 and 2 with an invariant proof that considers only pairs of adjacent steps. We then invoke the proof from §4.3 that each of the action transitions occurs infinitely often to establish requirement 3. Finally, having established the three preconditions for the WF1 lemma from our verified library, we call that lemma.

In some cases, we need lemmas from our library that prove other variants of the WF1 proof rule sound. For instance, often we must prove that C_i leads to C_{i+1} not just eventually but within a bounded time. For this, we have a variant of WF1 that proves C_{i+1} holds within the inverse of Action's frequency. It uses a modified requirement 3: that Action occurs with a minimum frequency.

Another useful variant of WF1 is *delayed, bounded-time WF1*. It applies when Action only induces C_{i+1} after a certain time t; this is common in systems that rate-limit certain actions for performance reasons. For instance, to amortize the cost of agreement, the IronRSL action for proposing a batch of requests has a timer preventing it from sending an incomplete batch too soon after the last batch. Delayed, bounded-time WF1 uses a modified requirement 2: "If Action occurs when C_i holds and the time is $\geq t$, it causes C_{i+1} to hold." This variant proves that C_{i+1} eventually holds after t plus the inverse of the action's frequency.

Sometimes, a liveness proof needs more than a chain of conditions: it must prove that multiple conditions eventually hold simultaneously. For instance, in IronRSL we must prove that a potential leader eventually knows suspicions from every replica in the quorum at once. For this, we use our temporal heuristics to prove sound the proof rule: "If every condition in a set of conditions eventually holds forever, then eventually all the conditions in the set hold simultaneously forever." We also have and use a bounded-time variant of this rule.

5. System Implementation

We use the IronFleet methodology to implement two practical distributed systems and prove them correct: a Paxos-based replicated state machine library and a lease-based sharded key-value store. All IronFleet code is publicly available [25].

5.1 IronRSL: A Replicated State Machine Library

IronRSL replicates a deterministic application on multiple machines to make that application fault-tolerant. Such replication is commonly used for services, like Chubby and Zookeeper [5, 24], on which many other services depend. Due to these dependencies, correctness bugs in replication can lead to cascading problems, and liveness bugs can lead to widespread outages of all dependent services.

IronRSL guarantees safety and liveness without sacrificing complex implementation features necessary to run real workloads. For instance, it uses batching to amortize the cost of consensus across multiple requests, log truncation to constrain memory usage, responsive view-change timeouts to avoid hard-coded assumptions about timing, state transfer to let nodes recover from extended network disconnection, and a reply cache to avoid unnecessary work.

```
predicate ExistsProposal(m_set:set<Msg1b>, op:Op)
{
  exists p :: p in m_set && op in p.msg.votes
}

predicate ProposeBatch(s:Proposer,s':Proposer)
{
  if |s.1bMsgs| < quorumSize then no_op()
  else if ExistsProposal(s.1bMsgs,s.nextOp) then
    var new_batches := s.proposedBatches[s.nextOp :=
                          BatchFromHighestBallot(s.1bMsgs, s.nextOp)];
    s' == s[nextOp := s.nextOp + 1]
           [proposedBatches := new_batches]
  else ...
}
```

Figure 10. *A step predicate example from IronRSL (simplified).*

5.1.1 The High-Level Specification

The spec for IronRSL is simply *linearizability*: it must generate the same outputs as a system that runs the application sequentially on a single node. Our implementation achieves this in the same way typical replicated state machine libraries do: it runs the application on multiple nodes, and uses the MultiPaxos [35] consensus protocol to feed the same requests in the same order to each replica.

5.1.2 The Distributed-Protocol Layer

Protocol. In the protocol layer, each host's state consists of four components, based on Lamport's description of Paxos [35]: a proposer, an acceptor, a learner, and an executor. The host's action predicates include, for instance, proposing a batch of requests (Figure 10) or sending the local application state to a host that has fallen behind.

Protocol invariants. The protocol's key invariant, known as *agreement*, is that two learners never decide on different request batches for the same slot. Establishing this invariant requires establishing several more invariants about earlier protocol actions. For instance, we prove that `ProposeValue` (Figure 10) cannot propose a batch if a different one may have already been learned. The action's predicate states that batches can only be proposed when the host has received a 1b message from at least $f+1$ acceptors. We use this to prove that this quorum of acceptors intersects with any other quorum that might have accepted a batch in a previous ballot.

Protocol refinement. After establishing the agreement invariant, we prove that executing the sequence of decided request batches is equivalent to taking steps in the high-level state machine. One challenge is that multiple replicas execute the same request batches, but the corresponding high-level steps must be taken only once. We address this by refining the distributed system to an abstract state machine that advances not when a replica executes a request batch but when a quorum of replicas has voted for the next request batch.

5.1.3 The Implementation Layer

Often, the most difficult part of writing a method to implement a protocol action is proving that the method has the appropriate effect on the refined state. For this, IronRSL relies on our generic refinement library (§5.3), which lightens the programmer's burden by proving useful properties about the refinement of common data structures.

Another difficulty is that the protocol sometimes describes the relationship between the host's pre-action and post-action state in a non-constructive way. For instance, it says that the log truncation point should be set to the *n*th highest number in a certain set. It describes how to *test* whether a number is the *n*th highest number in a set, but not how to actually *compute* such a quantity. Thus, the implementer must write a method to do this and prove it correct.

Writing and maintaining invariants is also useful in the implementation. Most IronRSL methods need some constraints on the concrete state they start with. For instance, without some constraint on the size of the log, we cannot prove that the method that serializes it can fit the result into a UDP packet. We incorporate this constraint (and many others) into an invariant over the concrete state. Each method learns these properties on entry and must prove them before returning.

Invariants are also a crucial part of performance optimization. Consider, for example, the `ExistsProposal` method in `ProposeBatch`. A naïve implementation would always iterate through all votes in all 1b messages, a costly process. Instead, we augment the host state with an additional variable, `maxOpn`, and prove an invariant that no 1b message exceeds it. Thus, in the common case that `s.nextOp` $\geq$ `maxOpn`, the implementation need not scan any 1b messages.

5.1.4 IronRSL Liveness

We also prove our implementation is live: if a client repeatedly sends a request to all replicas, it eventually receives a reply. No consensus protocol can be live under arbitrary conditions [16], so this property must be qualified by assumptions. We assume there exists a quorum of replicas Q, a minimum scheduler frequency F, a maximum network delay Δ, a maximum burst size B, and a maximum clock error E, all possibly unknown to the implementation, such that (1) eventually, the scheduler on each replica in Q runs with frequency at least F, never exhausting memory; (2) eventually, any message sent between replicas in Q and/or the client arrive within Δ; (3) eventually, no replica in Q receives packets at an overwhelming rate, i.e., each receives no more than B packets per $\frac{10B}{F} + 1$ time units; (4) whenever a replica in Q reads its clock, the reading differs from true global time by at most E; and (5) no replica in Q ever stops making progress due to reaching an overflow-prevention limit.

Our proof strategy is as follows. First, we use our library's round-robin scheduler proofs to prove that our protocol fairly schedules each action (§4.3). Next, we prove that eventually no replica in Q has a backlog of packets in its queue, so thereafter sending a message among replicas in Q leads to the receiver acting on that message within a certain bound. Next, using WF1 (§4.4), we prove that if the client's request is never executed, then for any time period T, eventually a replica in Q becomes the undisputed leader for that period. Finally, using bounded-time WF1 variants (§4.4), we prove there exists a T such that an undisputed leader can ensure the request gets executed and responded to within T.

5.2 IronKV: A Sharded Key-Value Store

We also apply the IronFleet methodology to build IronKV, a system that uses distribution for a completely different purpose: to scale its throughput by dynamically sharding a key-value store across a set of nodes.

The high-level spec of IronKV's state machine is concise: it is simply a hash table, as shown in Figure 11.

```
type Hashtable = map<Key,Value>
type OptValue = ValuePresent(v:Value) | ValueAbsent

predicate SpecInit(h:Hashtable)
{
  h == map []
}

predicate Set(h:Hashtable, h':Hashtable, k:Key, ov:OptValue)
{
  h' == if ov.ValuePresent? then h[k := ov.v]
        else map ki | ki in h && ki!=k :: h[ki]
}

predicate Get(h:Hashtable, h':Hashtable, k:Key, ov:OptValue)
{
     h' == h
  && ov == if k in h then ValuePresent(h[k])
           else ValueAbsent()
}

predicate SpecNext(h:Hashtable,h':Hashtable)
{
  exists k, ov :: Set(h,h',k,ov) || Get(h,h',k,ov)
}
```

Figure 11. *Complete high-level spec for IronKV state machine*

5.2.1 The Distributed-Protocol Layer

Each host's state consists of a hash table storing a subset of the key space and a "delegation map" mapping each key to the host responsible for it. On protocol initialization, one designated host is responsible for the entire key space; thus, each host's delegation map maps every key to that host.

To gain throughput and to relieve hot spots, IronKV allows an administrator to delegate sequential key ranges (shards) to other hosts. When a host receives such an order, it sends the corresponding key-value pairs to the intended recipient and updates its delegation map to reflect the new owner.

If such a message is lost, the protocol layer cannot be shown to refine the high-level specification, since the corresponding key-value pairs vanish. To avoid this, we design a sequence-number-based reliable-transmission component that requires each host to acknowledge messages it receives, track its own set of unacknowledged messages, and periodically resend them. The liveness property we prove is that if the network is fair (i.e., any packet sent infinitely often is eventually delivered), then any packet submitted to the reliable-transmission component is eventually received.

The most important invariant for IronKV's proof is that every key is claimed either by exactly one host or in-flight packet. Using this invariant and the exactly-once delivery semantics we prove about our reliable-transmission component, we show that the protocol layer refines the high-level spec.

5.2.2 The Implementation Layer

As in IronRSL, we prove that modifications to a host's concrete state refine changes to the protocol-layer state. The delegation map, however, poses a challenge unique to IronKV.

The protocol layer uses an infinite map with an entry for every possible key. However, the implementation layer must use concrete data types with bounded size and reasonable performance. Thus, we implement and prove correct an efficient data structure in which each host keeps only a compact list of key ranges, along with the identity of the host responsible for each range. This complexity we introduce for the sake of performance creates opportunities for bugs. However, by establishing invariants about the data structure (e.g., the ranges are kept in sorted order), we prove that it refines the abstract infinite map used by the protocol layer. This lets us introduce this complex data structure without risk of data loss or any other error.

5.3 Common Libraries

In developing IronRSL and IronKV, we have written and verified several generic libraries useful for distributed systems.

Generic refinement. A common task is proving that an operation on concrete implementation-layer objects refines the corresponding operation on protocol-layer objects. For example, IronRSL's implementation uses a map from `uint64`s to IP addresses where the protocol uses a map from mathematical integers to abstract node identifiers. In the proof, we must show that removing an element from the concrete map has the same effect on the abstract version.

To simplify such tasks, we have built a generic library for reasoning about refinement between common data structures, such as sequences and maps. Given basic properties about the relationship between the concrete types and the abstract types, e.g., that the function mapping concrete map keys to abstract maps keys is injective, the library shows that various concrete map operations, such as element lookup, addition, and removal, refine the corresponding abstract operations.

Marshalling and parsing. All distributed systems need to marshal and parse network packets, a tedious task prone to bugs. Both tasks necessarily involve significant interaction with the heap, since packets are ultimately represented as arrays of bytes. Unfortunately, even state-of-the-art verification tools struggle to verify heap operations (§6.2). Hence, we have written and verified a generic grammar-based parser and marshaller to hide this pain from developers. For each distributed system, the developer specifies a high-level grammar for her messages. To marshal or unmarshal, the developer simply maps between her high-level structure and a generic data structure that matches her grammar. The library handles the conversion to and from a byte array.

As evidence for the library's utility, we initially wrote an IronRSL-specific library. This took a person-month, and relatively little of this code would have been useful in other contexts. Dissatisfied, we built the generic library. This required several more weeks, but given the generic library, adding the IronRSL-specific portions only required two hours; the IronKV-specific portions required even less.

Collection Properties. Another common task for distributed systems is reasoning about properties of sequences, sets, maps, etc. For instance, many IronRSL operations require reasoning about whether a set of nodes form a quorum. Thus, we have developed a library proving many useful relationships about such collections. For example, one lemma proves that if two sets are related by an injective function, then their sizes are the same.

6. Lessons Learned

We summarize additional lessons we learned, beyond using invariant quantifier hiding (§3.3) and always-enabled actions (§4.2), useful for future developers of verified systems.

6.1 Use the Set of Sent Messages in Invariants

The IronFleet network model is monotonic: once a message is sent, it is kept in a ghost state variable forever. This is necessary to prove that the system behaves correctly even if the network delivers messages arbitrarily late. Since the set of messages can only grow, it is often easy to prove invariants about it. In contrast, an invariant that reasons over mutable host state is harder to prove. Thus, where possible, it is useful to have invariants be properties only of the set of messages sent so far, as is often done in proofs of security for cryptographic protocols [8]. Essentially, the system's network model provides this set as a free "history variable" [1].

6.2 Model Imperative Code Functionally

Verifying imperative code is challenging compared with verifying purely functional code, even when using a state-of-the-art tool like Dafny that is designed for imperative programs (§2.2). Thus, we found it profitable to implement the system in two stages. First, we develop an implementation using immutable value (functional) types and show that it refines the protocol layer. Avoiding heap reasoning simplifies the refinement proof, but, it produces a slow implementation, since it cannot exploit the performance of heap references. In the second stage, we replace the value types with mutable heap types, improving performance while solving only a narrow verification problem.

We apply this pattern in building IronRSL and IronKV; e.g., the functional implementation manipulates IP addresses as value types and the performant one uses references to OS handles. This strategy takes advantage of Dafny's support for mixing functional programming and imperative programming styles: we can first run the functional code and measure its performance, then optimize the performance-critical sections into imperative heap-based code as needed. Using a language without good functional programming support (such as C) would have made it harder to pursue this strategy.

6.3 Use Automation Judiciously

Automated verification tools reduce the human effort needed to complete a proof, but they often require additional guidance from the developer in order to find a proof, or, equally importantly, to find a proof in a reasonable amount of time.

6.3.1 Automation Successes

In many cases, Dafny's automated reasoning allows the developer to write little or no proof annotation. For instance, Dafny excels at automatically proving statements about linear arithmetic. Also, its heuristics for dealing with quantifiers, while imperfect, often produce proofs automatically.

Dafny can also prove more complex statements automatically. For instance, the lemma proving that IronRSL's `ImplNext` always meets the reduction-enabling obligation consists of only two lines: one for the precondition and one for the postcondition. Dafny automatically enumerates all ten possible actions and all of their subcases, and observes that all of them produce I/O sequences satisfying the property.

Similarly, automated reasoning allows many invariant proofs to be quite brief, by reasoning as follows: If the invariant about a host's state holds in step i but not $i+1$, the host must have taken some action. However, none of the actions can cause the invariant to stop holding. Typically, this last part requires no proof annotation as the verifier can internally enumerate all cases, even for IronRSL with its many complicated actions. Sometimes the verifier cannot handle a tricky case automatically, in which case the developer must insert

proof annotations. However, even then, the developer need not mention, let alone enumerate, the other cases.

6.3.2 Automation Challenges

Even the fastest automated verification tools can take a long time to explore a huge search space. By default, Dafny reveals all predicate definitions to its SMT solver Z3, potentially giving Z3 a large search space. For example, each distributed protocol's `HostNext` transitively includes almost every other definition in the protocol. Similarly, message-parsing code refers to a large tree of possible message types. Having such big trees in scope exposes the SMT solver to a bounded but still large search space, e.g., any mention of a state invokes every predicate about states.

To keep verification time manageable and avoid verifier timeouts, we use Dafny's `opaque` attribute and `reveal` directive to selectively hide irrelevant definitions from the SMT solver, and reveal them only when needed to complete a proof [21]. This leads to a more modular style of verification. In addition to hiding large definitions, we also use `opaque` to hide logic features that are hard to automate. For example, we mark recursive predicate definitions opaque to prevent the solver from blindly unrolling the definitions too many times.

To provide greater flexibility, we modify Dafny to also support a `fuel` attribute for functions. Fuel controls how many times the SMT solver may expand a function's definition. Giving a function zero fuel is equivalent to marking the function opaque, while giving a fuel of five allows the solver to unroll a recursive function up to five times. By allowing the programmer to specify a function's fuel at the scope of a statement, method, class, module, or program, we allow different portions of the code to be more or less aggressive about revealing function definitions.

Formulas that make heavy use of quantifiers (forall and exists) may also lead to timeouts because the SMT solver can instantiate the quantifiers more than it needs to, depending on which triggers the solver chooses to control instantiation. In many places, we adopt coding styles that avoid quantifiers (§3.3). In other places, when we find the default triggers in Dafny overly liberal, leading to too many instantiations, we modify Dafny to use more cautious triggers. In some cases, we also annotate our Dafny code with manual triggers to reduce instantiations. In particularly problematic formulas, such as chains of alternating quantifiers (e.g., for all X there exists a Y such that for all Z...) and set comprehensions, we mark the containing predicate opaque. Temporal logic formulas can easily lead to alternating quantifiers, so we define $\Box$ and $\Diamond$ to be opaque by default.

7. Evaluation

IronFleet's premise is that automated verification is a viable engineering approach, ready for developing real distributed systems. We evaluate that hypothesis by answering the following questions: (1) How does verification affect the development of distributed systems? (2) How does the performance of a verified system compare with an unverified one?

7.1 Developer Experience

To assess practicality, we evaluate the developer experience as well as the effort required to produce verified systems.

The experience of producing verified software shares some similarities with that of unverified software. Dafny provides near-real-time IDE-integrated feedback. Hence, as the developer writes a given method or proof, she typically sees feedback in 1–10 seconds indicating whether the verifier is satisfied. To ensure the entire system verifies, our build

	Spec	Impl	Proof	Time to Verify
	(source lines of code)			(minutes)
High-Level Spec:				
IronRSL	85	–	–	–
IronKV	34	–	–	–
Temporal Logic	208	–	–	–
Distributed Protocol:				
IronRSL Protocol	–	–	1202	4
Refinement	35	–	3379	26
Liveness	167	–	7869	115
IronKV Protocol	–	–	726	2
Refinement	36	–	3998	12
Liveness	98	–	2093	23
TLA Library	–	–	1824	2
Implementation:				
IO/Native Interface	591	–	–	–
Common Libraries	134	833	7690	13
IronRSL	6	2941	7535	152
IronKV	6	1340	2937	42
Total	1400	5114	39253	395

Figure 12. *Code sizes and verification times.*

system tracks dependencies across files and outsources, in parallel, each file's verification to a cloud virtual machine. Thus, while a full integration build done serially requires approximately six hours, in practice, the developer rarely waits more than 6–8 minutes, which is comparable to any other large system integration build.

An IronFleet developer must write a formal trusted spec, a distributed protocol layer, and proof annotations to help the verifier see the refinements between them. Figure 12 quantifies this effort by reporting the amount of proof annotation required for each layer of the system. We count all non-spec, non-executable code as proof annotation; this includes, for example, requires and ensures clauses, loop invariants, and all lemmas and invocations thereof. Note that the high-level trusted specification for IronRSL is only 85 SLOC, and for IronKV it is only 34, making them easy to inspect for correctness. At the implementation layer, our ratio of proof annotation to executable code is 3.6 to 1. We attribute this relatively low ratio to our proof-writing techniques (§3.3, §4.1, §6) and our automated tools (§6.3.1).

In total, developing the IronFleet methodology and applying it to build and verify two real systems required approximately 3.7 person-years.

In exchange for this effort, IronFleet produces a provably correct implementation with desirable liveness properties. Indeed, except for unverified components like our C# client, both IronRSL (including replication, view changes, log truncation, batching, etc.) as well as IronKV (including delegation and reliable delivery) worked the first time we ran them.

7.2 Performance of Verified Distributed Systems

A reasonable criticism of any new toolchain focused on verification is that its structure might impair runtime efficiency. While we focus most of our energy on overcoming verification burdens, we also try to produce viable implementations.

Our IronRSL experiments run three replicas on three separate machines, each equipped with an Intel Xeon L5630 2.13 GHz processor and 12 GB RAM, connected over a 1 Gbps network. Our IronKV experiments use two such machines connected over a 10 Gbps network.

IronRSL. Workload is offered by 1–256 parallel client threads, each making a serial request stream and measuring latency. As an unverified baseline, we use the MultiPaxos Go-based

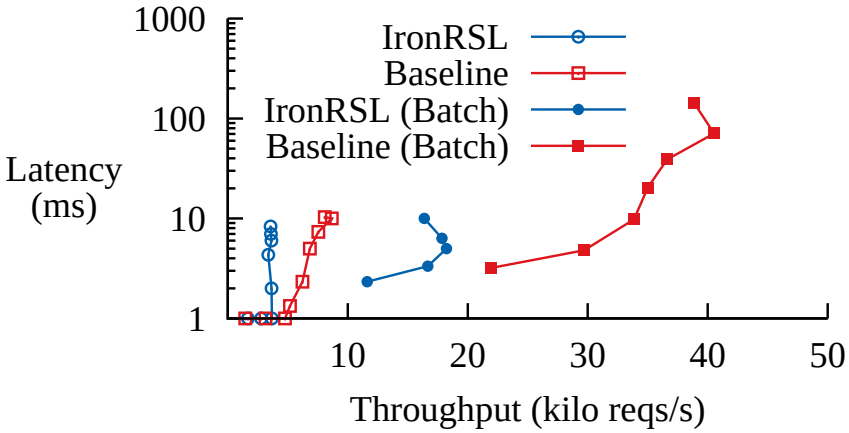

Figure 13. *IronRSL's performance is competitive with an unverified baseline. Results averaged over 3 trials.*

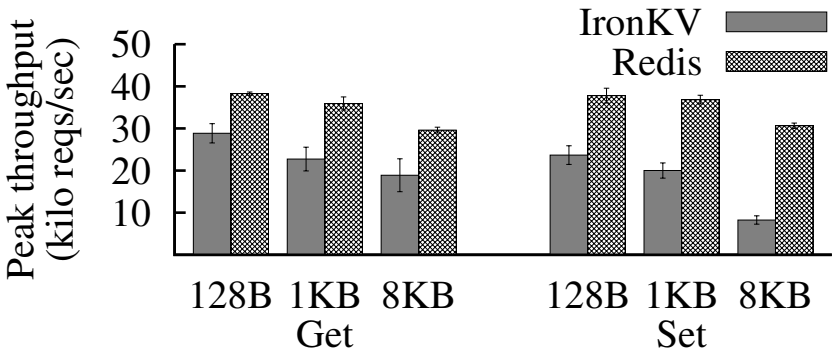

Figure 14. *IronKV's performance is competitive with Redis, an unverified key-value store. Results averaged over 3 trials.*

implementation from the EPaxos codebase [15, 45] For both systems, we use the same application state machine: it maintains a counter and it increments the counter for every client request. Figure 13 summarizes our results. We find that IronRSL's peak throughput is within 2.4× of the baseline.

IronKV. To measure the throughput and latency of IronKV, we preload the server with 1000 keys, then run a client with 1–256 parallel threads; each thread generates a stream of Get (or Set) requests in a closed loop. As an unverified baseline, we use Redis [58], a popular key/value store written in C and C++, with the client-side write buffer disabled. For both systems, we use 64-bit unsigned integers as keys and byte arrays of varying sizes as values. Figure 14 summarizes our results. We find that IronKV's performance is competitive with that of Redis.

As a final note, in all our experiments the bottleneck was the CPU (not the memory, disk, or network).

8. Discussion and Future Work

§7.1 shows that in exchange for strong guarantees (which depend on several assumptions, per §2.5), IronFleet requires considerably more developer effort. Furthermore, in our experience, there is a distinct learning curve when bringing aboard developers unfamiliar with writing verified code. Most developers would prefer to use a language like C++, so enabling that is an important topic of future work.

§7.2 shows that while our systems achieve respectable performance, they do not yet match that of the unverified baselines. Some of that gap stems directly from our use of verification. Verifying mutable data structures is challenging (§6.2), and our measurements indicate that this is a significant bottleneck for our code. The baselines we compare against have been highly optimized; we have also optimized our code, but each optimization must be proven correct. Hence, given a fixed time budget, IronFleet will likely produce fewer optimizations. IronFleet also pays a penalty for compiling to C#, which imposes run-time overhead to enforce type safety on code that provably does not need it.

More fundamentally, aiming for full verification makes it challenging to reuse existing libraries, e.g., for optimized packet serialization. Before our previous [21] and current work (§5.3), Dafny had no standard libraries, necessitating significant work to build them; more such work lies ahead.

While our systems are more full-featured than previous work (§9), they still lack many standard features offered by the unverified baselines. Some features, such as reconfiguration in IronRSL, only require additional developer time. Other features require additional verification techniques; e.g., post-crash recovery requires reasoning about the effects of machine crashes that wipe memory but not disk.

In future work, we aim to mechanically verify our reduction argument and prove that our implementation run its main loop in bounded time [2], never exhausts memory, and never reaches its overflow-prevention limit under reasonable conditions, e.g., if it never performs more than 2^{64} operations.

9. Related Work

9.1 Protocol Verification

Distributed system protocols are known to be difficult to design correctly. Thus, a systems design is often accompanied by a formal English proof of correctness, typically relegated to a technical report or thesis. Examples include Paxos [55], the BFT protocol for Byzantine fault tolerance [6, 7], the reconfiguration algorithm in SMART [23, 41], Raft [49, 50], Zookeeper's consistent broadcast protocol Zab [28, 29], Egalitarian Paxos [44, 45], and the Chord DHT [62, 63].

However, paper proofs, no matter how formal, can contain errors. Zane showed that the "provably correct" Chord protocol, when subjected to Alloy abstract model checking, maintains none of its published invariants [71]. Thus, some researchers have gone further and generated machine-checkable proofs. Kellomäki created a proof of the Paxos consensus protocol checked in PVS [30]. Lamport's TLAPS proof system has been used to prove safety, but not liveness, properties of the BFT protocol [38]. In all such cases, the protocols proven correct have been much smaller and simpler than ours. For instance, Kellomäki's and Lamport's proofs concerned single-instance Paxos and BFT, which make only one decision total.

9.2 Model Checking

Model checking exhaustively explores a system's state space, testing whether a safety property holds in every reachable state. This combinatorial exploration requires that the system be instantiated with finite, typically tiny, parameters. As a result, a positive result provides only confidence, not proof of safety; furthermore, that confidence depends on the modeler's wisdom in parameter selection. Model checking has been applied to myriad systems including a Python implementation of Paxos [26]; Mace implementations of a variety of distributed systems [31]; and, via MODIST [69], unmodified binaries of Berkeley DB, MPS Paxos, and the PacificA primary-backup replication system.

Model checking scales poorly to complex distributed specs [4]. Abstract interpretation can help with such scaling but does not fundamentally eliminate model checking's limitations. For instance, Zave's correction to Chord uses the Alloy model checker but only to partially automate the proof of a single necessary invariant [72].

9.3 System Verification

The recent increase in the power of software verification has emboldened several research groups to use it to prove the correctness of entire systems implementations. seL4 is a microkernel written in C [32], with full functional correctness proven using the Isabelle/HOL theorem prover. mCertiKOS-hyp [19] is a small verified hypervisor, whose verification in the Coq interactive proof assistant places a strong emphasis on modularity and abstraction. ExpressOS [43] uses Dafny to sanity-check a policy manager for a microkernel. Our Ironclad project [21] shows how to completely verify the security of sensitive services all the way down to the assembly. IronFleet differs by verifying a distributed implementation rather than code running on a single machine, and by verifying liveness, as well as safety, properties.

Researchers have also begun to apply software verification to distributed systems. Ridge [59] proves the correctness of a persistent message queue written in OCaml; however, his system is substantially smaller in scale than ours and has no proven liveness properties.

Rahli et al. [57] verify the correctness of a Paxos implementation by building it in EventML [56] and proving correctness, but not liveness, with the NuPRL prover [10]. However, they do not verify the state machine replication layer of this Paxos implementation, only the consensus algorithm, ignoring complexities such as state transfer. They also make unclear assumptions about network behavior. In contrast to our methodology, which exploits multiple levels of abstraction and refinement, the EventML approach posits a language below which all code generation is automatic, and above which a human can produce a one-to-one refinement. It is unclear if this approach will scale up to more complex and diverse distributed systems.

In concurrent work, Wilcox et al. [67, 68] propose Verdi, a compiler-inspired approach to building verified distributed system implementations. With Verdi, the developer writes and proves her system correct in Coq using a simplified environment (e.g., a single-machine system with a perfectly reliable network). Verdi's verified system transformers then convert the developer's implementation into an equivalent implementation that is robust in a more hostile environment; their largest system transformer is an implementation of Raft that adds fault tolerance. Compared with IronFleet, Verdi offers a cleaner approach to composition. Unlike IronRSL, at present Verdi's Raft implementation does not support verified marshalling and parsing, state transfer, log truncation, dynamic view-change timeouts, a reply cache, or batching. Also, Verdi does not prove any liveness properties.

10. Conclusion

The IronFleet methodology slices a system into specific layers to make verification of practical distributed system implementations feasible. The high-level spec gives the simplest description of the system's behavior. The protocol layer deals solely with distributed protocol design; we connect it to the spec using TLA+ [36] style verification. At the implementation layer, the programmer reasons about a single-host program without worrying about concurrency. Reduction and refinement tie these individually-feasible components into a methodology that scales to practically-sized concrete implementations. This methodology admits conventionally-structured implementations capable of processing up to 18,200 requests/second (IronRSL) and 28,800 requests/second (IronKV), performance competitive with unverified reference implementations.

Acknowledgments

We thank Rustan Leino for not just building Dafny but also cheerfully providing ongoing guidance and support in improving it. We thank Leslie Lamport for useful discussions about refinement and formal proofs, particularly proofs of liveness. We thank Shaz Qadeer for introducing us to the power of reduction. We thank Andrew Baumann, Ernie Cohen, Galen Hunt, Lidong Zhou, and the anonymous reviewers for useful feedback. Finally, we thank our shepherd Jim Larus for his interactive feedback that significantly improved the paper.

References

[1] ABADI, M., AND LAMPORT, L. The existence of refinement mappings. *Theoretical Computer Science 82*, 2 (May 1991).

[2] BLACKHAM, B., SHI, Y., CHATTOPADHYAY, S., ROYCHOUDHURY, A., AND HEISER, G. Timing analysis of a protected operating system kernel. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)* (2011).

[3] BOKOR, P., KINDER, J., SERAFINI, M., AND SURI, N. Efficient model checking of fault-tolerant distributed protocols. In *Proceedings of the Conference on Dependable Systems and Networks (DSN)* (2011).

[4] BOLOSKY, W. J., DOUCEUR, J. R., AND HOWELL, J. The Farsite project: a retrospective. *ACM SIGOPS Operating Systems Review 41 (2)* (April 2007).

[5] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[6] CASTRO, M., AND LISKOV, B. A correctness proof for a practical Byzantine-fault-tolerant replication algorithm. Tech. Rep. MIT/LCS/TM-590, MIT Laboratory for Computer Science, June 1999.

[7] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS) 20*, 4 (Nov. 2002).

[8] COHEN, E. First-order verification of cryptographic protocols. *Journal of Computer Security 11*, 2 (2003).

[9] COHEN, E., AND LAMPORT, L. Reduction in TLA. In *Concurrency Theory (CONCUR)* (1998).

[10] CONSTABLE, R. L., ALLEN, S. F., BROMLEY, H. M., CLEAVELAND, W. R., CREMER, J. F., HARPER, R. W., HOWE, D. J., KNOBLOCK, T. B., MENDLER, N. P., PANANGADEN, P., SASAKI, J. T., AND SMITH, S. F. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., 1986.

[11] DE MOURA, L. M., AND BJØRNER, N. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[12] DETLEFS, D., NELSON, G., AND SAXE, J. B. Simplify: A theorem prover for program checking. In *J. ACM* (2003).

[13] DOUCEUR, J. R., AND HOWELL, J. Distributed directory service in the Farsite file system. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)* (November 2006).

[14] ELMAS, T., QADEER, S., AND TASIRAN, S. A calculus of atomic actions. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (Jan. 2009).

[15] EPaxos code. https://github.com/efficient/epaxos/, 2013.

[16] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM) 32*, 2 (April 1985).

[17] FLOYD, R. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics* (1967).

[18] GARLAND, S. J., AND LYNCH, N. A. Using I/O automata for developing distributed systems. *Foundations of Component-Based Systems 13* (2000).

[19] GU, R., KOENIG, J., RAMANANANDRO, T., SHAO, Z., WU, X. N., WENG, S.-C., ZHANG, H., AND GUO, Y. Deep specifications and certified abstraction layers. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (2015).

[20] GUO, H., WU, M., ZHOU, L., HU, G., YANG, J., AND ZHANG, L. Practical software model checking via dynamic interface reduction. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2011), ACM.

[21] HAWBLITZEL, C., HOWELL, J., LORCH, J. R., NARAYAN, A., PARNO, B., ZHANG, D., AND ZILL, B. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (October 2014).

[22] HOARE, T. An axiomatic basis for computer programming. *Communications of the ACM 12* (1969).

[23] HOWELL, J., LORCH, J. R., AND DOUCEUR, J. R. Correctness of Paxos with replica-set-specific views. Tech. Rep. MSR-TR-2004-45, Microsoft Research, 2004.

[24] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2010).

[25] IronFleet code. https://research.microsoft.com/projects/ironclad/, 2015.

[26] JONES, E. Model checking a Paxos implementation. http://www.evanjones.ca/model-checking-paxos.html, 2009.

[27] JOSHI, R., LAMPORT, L., MATTHEWS, J., TASIRAN, S., TUTTLE, M., AND YU, Y. Checking cache coherence protocols with TLA+. *Journal of Formal Methods in System Design 22*, 2 (March 2003).

[28] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Dissecting Zab. Tech. Rep. YL-2010-007, Yahoo! Research, December 2010.

[29] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the IEEE/IFIP Conference on Dependable Systems & Networks (DSN)* (2011).

[30] KELLOMÄKI, P. An annotated specification of the consensus protocol of Paxos using superposition in PVS. Tech. Rep. 36, Tampere University of Technology, 2004.

[31] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: Language support for building distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2007).

[32] KLEIN, G., ANDRONICK, J., ELPHINSTONE, K., MURRAY, T., SEWELL, T., KOLANSKI, R., AND HEISER, G. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems 32*, 1 (2014).

[33] LAMPORT, L. A theorem on atomicity in distributed algorithms. Tech. Rep. SRC-28, DEC Systems Research Center, May 1988.

[34] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994).

[35] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (May 1998).

[36] LAMPORT, L. *Specifying Systems: The TLA+ Languange and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[37] LAMPORT, L. The PlusCal algorithm language. In *Proceedings of the International Colloquium on Theoretical Aspects of Computing (ICTAC)* (Aug. 2009).

[38] LAMPORT, L. Byzantizing Paxos by refinement. In *Proceedings of the International Conference on Distributed Computing (DISC)* (2011).

[39] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (2010).

[40] LIPTON, R. J. Reduction: A method of proving properties of parallel programs. *Communications of the ACM, 18*, 12 (1975).

[41] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., AND HOWELL, J. The SMART way to migrate replicated stateful services. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)* (2006).

[42] LU, T., MERZ, S., WEIDENBACH, C., BENDISPOSTO, J., LEUSCHEL, M., ROGGENBACH, M., MARGARIA, T., PADBERG, J., TAENTZER, G., LU, T., MERZ, S., AND WEIDENBACH, C. Model checking the Pastry routing protocol. In *Workshop on Automated Verification of Critical Systems* (2010).

[43] MAI, H., PEK, E., XUE, H., KING, S. T., AND MADHUSUDAN, P. Verifying security invariants in ExpressOS. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2013).

[44] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. A proof of correctness of Egalitarian Paxos. Tech. Rep. CMU-PDL-13-111, Carnegie Mellon University Parallel Data Laboratory, August 2013.

[45] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)* (2013).

[46] MUSUVATHI, M., PARK, D., CHOU, A., ENGLER, D., AND DILL, D. L. CMC: A pragmatic approach to model checking real code. In *Proceedings of the USENIX Symposium Operating Systems Design and Implementation (OSDI)* (2002).

[47] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[48] NEWCOMBE, C., RATH, T., ZHANG, F., MUNTEANU, B., BROOKER, M., AND DEARDEUFF, M. How Amazon Web Services uses formal methods. *Communications of the ACM 58*, 4 (Apr. 2015).

[49] ONGARO, D. Consensus: Bridging theory and practice. Tech. Rep. Ph.D. thesis, Stanford University, August 2014.

[50] ONGARO, D., AND OUSTERHOUR, J. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (June 2014).

[51] PARKINSON, M. The next 700 separation logics. In *Proceedings of the IFIP Conference on Verified Software: Theories, Tools, Experiments (VSTTE)* (Aug. 2010).

[52] PARNO, B., LORCH, J. R., DOUCEUR, J. R., MICKENS, J., AND MCCUNE, J. M. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2011).

[53] PEK, E., AND BOGUNOVIC, N. Formal verification of communication protocols in distributed systems. In *Proceedings of the Joint Conferences on Computers in Technical Systems and Intelligent Systems* (2003).

[54] PRIOR, A. N. *Papers on Time and Tense*. Oxford University Press, 1968.

[55] PRISCO, R. D., AND LAMPSON, B. Revisiting the Paxos algorithm. In *Proceedings of the International Workshop on Distributed Algorithms (WDAG)* (1997).

[56] RAHLI, V. Interfacing with proof assistants for domain specific programming using EventML. In *Proceedings of the International Workshop on User Interfaces for Theorem Provers (UITP)* (July 2012).

[57] RAHLI, V., SCHIPER, N., BICKFORD, M., CONSTABLE, R., AND VAN RENESSE, R. Developing correctly replicated databases using formal tools. In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks (DSN)* (June 2014).

[58] Redis. http://redis.io/. Implementation used: version 2.8.2101 of the MSOpenTech distribution https://github.com/MSOpenTech/redis, 2015.

[59] RIDGE, T. Verifying distributed systems: The operational approach. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (January 2009).

[60] SAISSI, H., BOKOR, P., MUFTUOGLU, C., SURI, N., AND SERAFINI, M. Efficient verification of distributed protocols using stateful model checking. In *Proceedings of the Symposium on Reliable Distributed Systems SRDS* (Sept 2013).

[61] SCIASCIO, E., DONINI, F., MONGIELLO, M., AND PISCITELLI, G. Automatic support for verification of secure transactions in distributed environment using symbolic model checking. In *Conference on Information Technology Interfaces* (June 2001), vol. 1.

[62] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (August 2001).

[63] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. Tech. Rep. MIT/LCS/TR-819, MIT Laboratory for Computer Science, March 2001.

[64] TASIRAN, S., YU, Y., BATSON, B., AND KREIDER, S. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *International Workshop on Microprocessor Test and Verification* (June 2002), IEEE.

[65] WANG, L., AND STOLLER, S. D. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering 32* (Feb. 2006).

[66] WANG, Y., KELLY, T., KUDLUR, M., LAFORTUNE, S., AND MAHLKE, S. A. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (December 2008).

[67] WILCOX, J., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M., AND ANDERSON, T. UW CSE News: UW CSE's Verdi team completes first full formal verification of Raft consensus protocol. https://news.cs.washington.edu/2015/08/07/, August 2015.

[68] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (June 2015).

[69] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (April 2009).

[70] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G. R., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (October 2014).

[71] ZAVE, P. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review 42*, 2 (April 2012).

[72] ZAVE, P. How to make Chord correct (using a stable base). Tech. Rep. 1502.06461 [cs.DC], arXiv, February 2015.