DOI:10.1145/3068608

IronFleet: Proving Safety and Liveness of Practical Distributed Systems

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

Abstract

Distributed systems are notorious for harboring subtle bugs. Verification can, in principle, eliminate these bugs, but it 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 temporal logic of actions-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 such as "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,[1, 8, 18] verifying,[20] or at least bug-checking[9] 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[15] and implementations.[17] In practice, however, model checking is incomplete—the accuracy of the results depends on the accuracy of the model—and does not scale.[1]

This paper presents IronFleet, the first methodology for automated machine-checked verification of the safety and liveness of nontrivial 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 as a key-value store, and a replicated state machine acts as 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. Moreover, it not only rules out bad behavior but also 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; for example, replicated-state-machine linearizability says that clients never see inconsistent results. A liveness property says that the system eventually performs a useful action, for example, that it 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 (Section 7), 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 (Section 3) that blends two distinct verification styles within the same automated theorem-proving framework, preventing any semantic gaps between them. We use temporal logic of actions (TLA)-style state-machine refinement[13] to reason about protocol-level concurrency, ignoring implementation complexities, then use

The original version of this paper was published as "IronFleet: Proving Practical Distributed Systems Correct" in the *25th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

Floyd–Hoare-style imperative verification[5,7] to reason about those complexities while ignoring concurrency. To simplify reasoning about concurrency, we impose a machine-checked *reduction-enabling obligation* on the implementation. Finally, we structure our protocols using *always-enabled actions* (Section 4) to greatly simplify liveness proofs.

To illustrate IronFleet's applicability, we have built and proven correct two rather different distributed systems: IronRSL, a Paxos-based[12] replicated-state-machine library, and IronKV, a sharded key-value store. All IronFleet code is publicly available.

IronRSL, our first application, has a complex implementation including many details often omitted by prior work, such as state transfer, log truncation, dynamic view-change timeouts, batching, and a reply cache. We prove full functional correctness and the key liveness property: if the network is eventually synchronous for a live quorum of replicas, then clients that persist in sending requests eventually get replies.

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. IronFleet's correctness relies on several assumptions (Section 2.4). Also, verification requires more up-front development effort: the automated tools we use fill in many low-level proof steps automatically, but still require considerable assistance from the developer. Finally, we focus on verifying newly written code in Dafny, a verification-friendly language (Section 2.2), rather than verifying existing code.

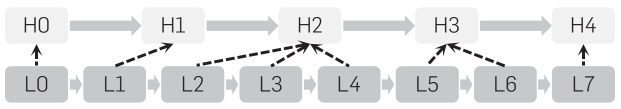
2. BACKGROUND AND ASSUMPTIONS

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

2.1. State machine refinement

State machine refinement[11] is often used to reason about distributed systems.[1,8,18] The developer describes the desired system as a simple abstract state machine with potentially infinitely many states and with nondeterministic 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*, that is, each (potentially infinite) sequence of states the machine may visit, corresponds to an equivalent behavior of H. State machine refinement in a distributed-system context (e.g., TLA-style refinement[13]) typically considers declarative specifications, not imperative code.

Figure 1. State machine refinement. The low-level state machine behavior L0…L7 refines the high-level behavior H0…H4. Each low-level state corresponds to a high-level state; for each such correspondence, shown as a dashed line, the two states must satisfy the spec's refinement conditions. Each low-level step maps to one high-level step (e.g., L0→L1 maps to H0→H1) or no high-level steps (e.g., L2→L3).



2.2. Floyd–Hoare verification

Many program verification tools support Floyd–Hoare style[5,7] first-order predicate logic reasoning about imperative programs. That is, the programmer annotates a program with assertions about the program's state, and the verifier checks that the assertions hold 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*.

We use Dafny,[14] a high-level language that automates verification via the Z3[2] 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. In such cases, the developer may insert annotations to guide the verifier's heuristics to a proof.

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 be possible to compile Dafny to them to, for example, simplify integration with existing code. Our previous work[6] 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.[19]

2.3. Temporal logic of actions (TLA)

Temporal logic and its extension TLA[11] 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; for example, a formula P could be "host h holds the lock now." Other formulas involve the future; for example, $\Diamond P$ means P eventually holds, and $\Box P$ means P holds now and forever. Thus, $\forall h \in \text{Hosts}: \Box\Diamond P$ means that for any host, it is always true that h will eventually hold the lock.

Figure 2. Simple Floyd–Hoare verification example.

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

Although Dafny does not directly support the temporal logic □ and ◊ operators, Dafny's logic is powerful enough to encode □ and ◊ using universal and existential quantifiers (∀ and ∃). Section 4 describes our encoding, which is a simple library written in Dafny that does not require any extensions to the Dafny language. Thus, we do not need a separate tool for reasoning about TLA, nor do we modify Dafny; instead, we use the existing Dafny language to reason about both our executable implementation and our high-level TLA-style specifications. Using a single language avoids any semantic gaps between implementation and specification.

2.4. 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 that runs `ImplInit` and `ImplNext` (see Section 3). We do not assume reliable packet delivery, so the network may arbitrarily delay, drop, or duplicate packets. We *do* assume the network does not tamper with packets, and that addresses in packet headers are trustworthy. These integrity assumptions can be enforced within, say, a datacenter or VPN, and could be relaxed by modeling the necessary cryptographic primitives to talk about keys instead of addresses.[6]

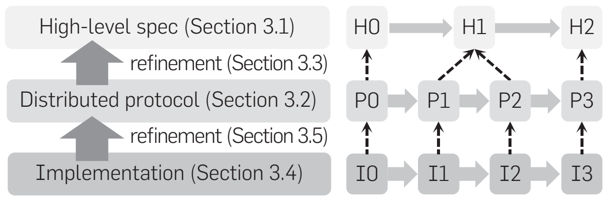
We assume the correctness of Dafny, the .NET compiler and runtime, and the underlying Windows OS. Our previous work[6] shows how to compile Dafny code into verifiable assembly code to avoid these dependencies. We also rely on the correctness of the underlying hardware.

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. For IronKV, we assume that each host's main loop executes infinitely often and that the network is fair, that is, a message sent infinitely often is eventually delivered.

3. VERIFICATION METHODOLOGY

IronFleet organizes a distributed system's implementation and proof into layers (Figure 3), all of which are expressed in Dafny. This layering avoids the intermingling of subtle distributed protocols with implementation complexity. At the top (Section 3.1), we write a simple spec for the system's behavior. We then write an abstract distributed protocol layer (Section 3.2) and use TLA-style techniques to prove that it refines the spec layer (Section 3.3). Then we write an imperative implementation layer to run on each host (Section 3.4) and prove that, despite the complexities introduced when writing real systems code, the implementation correctly refines the protocol layer (Section 3.5). Section 4 extends this methodology to liveness properties.

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…H2). 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.



High-level spec (Section 3.1)
refinement (Section 3.3)
Distributed protocol (Section 3.2)
refinement (Section 3.5)
Implementation (Section 3.4)
H0
H1
H2
P0
P1
P2
P3
I0
I1
I2
I3

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 verified reduction argument to show that a proof assuming atomicity is equally valid as a proof for the real system. This argument imposes a mechanically verified property on the implementation.

3.1. The high-level spec layer

What does it mean for a system to be *correct*? One can informally enumerate properties and hope they suffice to provide correctness. A more rigorous way is to define a *spec*, a succinct description of all allowable behaviors 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 expressed in Dafny (Section 2.2): starting with some initial state, the spec succinctly describes how that state can be transformed. The spec defines the state machine via three *predicates*, that is, 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 spec state. For instance, in Figure 3, `SpecInit` constrains H0, `SpecNext` constrains steps such as H0→H1 and H1→H2, and `SpecRelation` constrains corresponding state pairs such as (I1, H1) and (I3, H2). To avoid unnecessary constraints on implementations of the spec, `SpecRelation` should only talk about the externally visible behavior of the implementation, for example, the set of messages it has sent so far.

As a toy example, the Dafny spec in Figure 4 describes a simple distributed lock service with a single lock that passes among the hosts. It defines the system's state as a history: a sequence of host IDs such that the *n*th host in the sequence held the lock in epoch *n*. Initially, this history contains one valid host. The system can step from an old state 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 *n*th 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.

Figure 4. A toy lock specification.

```
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 5. Simplified host state machine for a lock service.

```
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
  && rpkt.msg.epoch > s_old.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)
}
```

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 the subtle concurrency, we keep this layer simple and abstract.

In more detail, we formally specify, in Dafny, a distributed system state machine. This state machine consists of *N* host state machines and a set 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; our reduction argument relaxes this atomicity assumption (see full paper).

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`). Within the protocol layer, 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; for example, 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 (Section 3.3). In our reduction argument we connect this atomicity-assuming proof to a real execution.

Continuing our lock example, the protocol layer might define a host state machine in Dafny 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. `HostNext` then says that a host may step from an old to a new state, given some incoming and outgoing packets, if the new state is the result of one of two *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 protocol to specification

The first major theorem we prove about each system is that the distributed protocol layer refines the high-level spec layer; that is, given a behavior of IronFleet's distributed system in which *N* hosts take atomic protocol steps defined by `HostNext`, 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 *protocol abstraction function* `PAbs` 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. Second, we prove that `PAbs` 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 either `PAbs(ps_old) = PAbs(ps_new)` or `SpecNext(PAbs(ps_old), PAbs(ps_new))`.

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.

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 by, for example, using mutable arrays instead of immutable sequences and using `uint64s` 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.

The trusted network interface maintains a ghost variable (a variable used only for verification, not execution) that represents a "journal" of every `Send` and `Receive` that the implementation might make, including all of the arguments and return values. We use this journal when connecting the implementation to the protocol.

3.5. Connecting implementation to 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 Section 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 an abstraction function `HAbs` that maps a host's implementation state to a host protocol state. As shown in Figure 6, we prove that the code `ImplInit`, which initializes the implementation state, ensures `HostInit` for the abstraction of that state. Similarly, we prove that the code `ImplNext`, which executes one host step, ensures `HostNext`. Note that `HostNext` refers to the journal of network events, thus connecting the implementation's low-level network actions to the protocol's abstract description of how the host should handle packets it sends and receives.

We then use our proof about one host implementation to prove that a distributed system comprising *N* host implementations, which is what we actually intend to run, refines the distributed protocol of *N* hosts. We use an *implementation abstraction function* `IAbs` 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 abstraction functions from our two major refinement theorems, composing them to form our final abstraction function `PAbs(IAbs(·))`. The key part of this proof is establishing that the specified relation conditions hold, that is, that for all implementation states `is`, `SpecRelation (is, PAbs (IAbs (is)))` holds.

4. VERIFYING LIVENESS

Section 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, we also often want 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 7). Thus, a

Figure 6. Mandatory host event-handler loop.

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

Figure 7. Desired liveness property for the lock service.

```
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)
}
```

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,[9] but it is better to definitively *prove* their absence. With such a proof, we do not have to reason about, for example, 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 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 (Section 4.2), often causing the prover to time out rather than return a successful verification or a useful error message.

With IronFleet, we address these challenges by writing a library in Dafny that defines standard TLA operators and proves standard TLA rules 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 library

As discussed in Section 2.3, TLA[11] is a standard mathematical formalism for reasoning about liveness. IronFleet encodes TLA in Dafny by expressing a TLA behavior, an infinite sequence of system states, as a Dafny 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, the Dafny code in Figure 7 says that for every host h, there is always a later time when h will hold the lock.

Our encoding 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, such as 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*.[3] 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 $(\Diamond Q) \wedge (\Box P) \Rightarrow \Diamond\,(P \wedge Q)$.

This heuristic is effective enough to automatically prove 40 fundamental TLA proof rules, that is, rules for deriving one formula from other formulas.[11] The heuristic allows us to prove complicated rules efficiently; for example, we state and prove a key rule about invariants in only 27 lines of Dafny, and a key rule about fairness in only 16 lines. Our liveness proofs then use these fundamental proof-rule lemmas to justify temporal formula transformations.

4.2. Always-enabled actions

Liveness properties depend on *fairness assumptions*, that is, assumptions that the underlying environment will enable progress. For instance, in IronRSL our liveness property depends on a quorum of participants continuing to run, and on the network delivering packets among that quorum and the client in a timely fashion. Fairness assumptions let us prove *fairness properties*: properties indicating that our protocol makes progress. An example fairness property is "Each host executes `HostGrant` infinitely often."

Lamport[13] suggests that fairness properties take the form "if action A becomes always *enabled*, that is, always possible to do, the implementation must eventually do it." However, reasoning about such properties is challenging. For instance, it is difficult to verify that an implementation's scheduler really has such a property. Also, to use such a property one must prove that A will always be enabled as long as some condition C holds, that is, that $\forall s.\ C(s) \Rightarrow \exists s' \mid A(s, s')$. Proving statements with alternating universal and existential quantifiers is notoriously challenging for automated theorem provers.

We thus adopt *always-enabled actions*; that is, 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 without 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. This means we can write a method that always does `HostGrant` no matter what state the host is in. Then, the fairness property "Each host executes `HostGrant` infinitely often" can be proven by showing that each host runs the method infinitely often; we accomplish this by invoking `HostGrant` inside a round-robin scheduler that itself sits inside an infinite loop.

Since our approach deviates from Lamport's standard fairness formulas, it can admit specifications that are not machine closed.[13] 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. 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.[11] 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 establish requirement 3, a fairness property, as discussed in Section 4.2. Finally, having established the three preconditions for the WF1 lemma from our verified library, we call that lemma.

5. SYSTEM IMPLEMENTATION

We use the IronFleet methodology to implement two practical distributed systems. All IronFleet code is publicly available.

5.1. IronRSL

IronRSL replicates a deterministic application on multiple machines to make that application fault-tolerant. Such replication is commonly used for critical services, such as Chubby and Zookeeper, on which many other services depend.

IronRSL guarantees safety and liveness while supporting complex implementation features. For instance, it uses batching to amortize consensus costs, log truncation to constrain memory usage, and state transfer to let nodes recover from extended network disconnection. 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 linearizability via the MultiPaxos[12] consensus protocol. It is worth noting that our spec does not enforce *exactly once* semantics, as it is a matter of much debate whether linearizability implies such semantics or not. If required, exactly-once semantics can be implemented—and formally proven—at the application level. We also prove that 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,[4] so we prove liveness of IronRSL under a set of fairness assumptions about the network and nodes.

5.2. IronKV

IronKV 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 map (Figure 8).

In IronKV's distributed-protocol layer, each host's state consists of a map storing a subset of the key space and a "delegation map" mapping each key to the host responsible for it. To gain throughput and to relieve hot spots, IronKV allows an administrator to delegate key ranges 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 spec, since the corresponding key-value pairs vanish. To avoid this, we design a reliable-transmission component that requires each host to acknowledge messages it receives, track its own set of unacknowledged messages, and periodically resend them. We prove desirable safety and liveness properties of this component.

We then prove a key invariant—every key is claimed either by exactly one host or in-flight packet—that we use in conjunction with the semantics ensured by the reliable-transmission component to show that the protocol layer refines the high-level spec. Finally, we implement the protocol and prove it refines the protocol layer.

Figure 8. Complete high-level spec for IronKV state machine.

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

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

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

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

5.3. Common libraries

We wrote several libraries when building IronRSL and IronKV.

Marshalling and parsing. All distributed systems need to marshal and parse network packets, a tedious task prone to bugs. 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. The library automatically converts byte arrays to and from a datatype conforming to the grammar.

Collection properties. We have developed a library proving many useful relationships about collections such as sequences, sets, maps, etc. These are common for reasoning about distributed systems, for example, to reason about whether a set of nodes form a quorum.

Generic refinement. We also built a library for reasoning about refinement between collections, for example, to prove the refinement from protocol-layer collections containing abstract node identifiers to implementation-layer collections containing IP addresses.

6. 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?

6.1. 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 integrated development environment feedback. Hence, as the developer writes a given method or proof, she typically sees feedback in 1–10 s indicating whether the verifier is satisfied. To ensure the entire system verifies, our build 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 6 h, in practice, the developer rarely waits more than 6–8 min, which is comparable to a traditional large system integration build and test pass.

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. Table 1 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, preconditions and postconditions, loop invariants, and all lemmas and invocations thereof. Our ratio of proof annotation to implementation is 7.7:1 (5.4:1 if liveness proof annotations are excluded). In total, developing the IronFleet methodology and applying it to build and verify two real systems required approximately 3.7 person-years.

Table 1. Code sizes and verifi cation times

	Spec	Impl	Proof	Time to verify
	Source lines of code			(minutes)
High-level spec	327			
Distributed protocol				
IronRSL	202	–	12,450	145
IronKV	134	–	6817	37
TLA library	–	–	1824	2
Implementation	737	5114	18,162	207
Total	1400	5114	39,253	395

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 view changes, log truncation, etc.) as well as IronKV (including delegation and reliable delivery) worked the first time we ran them.

6.2. Performance

We run IronRSL on three replicas on three separate machines, each equipped with an Intel Xeon 2.13 GHz processor and connected over a 1 Gbps network. Our IronKV experiments use two such machines connected over a 10 Gbps network. In all our experiments the bottleneck was the CPU (not the memory, disk, or 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 implementation from the EPaxos codebase.[16] For both systems, we measure with and without batching, and we use the same application state machine: it maintains a counter and it increments the counter for every client request. Figure 9 summarizes our results. We find that IronRSL's peak throughput is within 2.4× of the baseline.

IronKV. To measure the throughput 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, 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 10 summarizes our results. We find that IronKV's performance is competitive with that of Redis.

While our systems achieve respectable performance, they do not yet match that of the unverified baselines. Since verifying mutable data structures is challenging, we sometimes employ immutable data structures instead; our measurements indicate that these create significant bottlenecks. The baselines we compare against are highly optimized; we have also optimized our code, but each optimization must be proven correct rather than just implemented and tested. Hence, given a fixed time budget, IronFleet may produce fewer optimizations. IronFleet also suffers from compiling to C#, which imposes run-time

Figure 9. IronRSL's performance is competitive with an unverified MultiPaxos system. Results averaged over three trials.

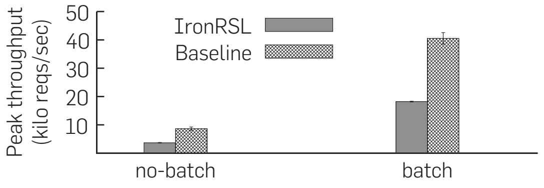
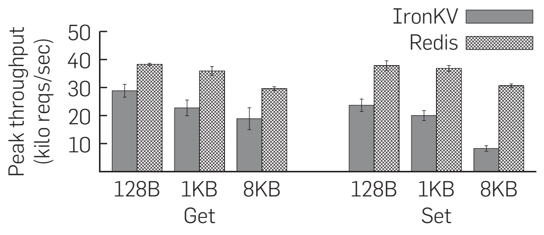


Figure 10. IronKV's performance is competitive with Redis, an unverified key-value store. Results averaged over three trials.



overhead to enforce type safety on code that provably does not need it.

7. RELATED WORK

The recent increase in the power of software verification has emboldened several research groups to use it to prove the correctness of single-machine implementations, for example, the seL4 microkernel.[10] Our Ironclad project[6] shows how to completely verify the security of sensitive services all the way down to the assembly code.

Distributed systems are known to harbor subtle design and implementation errors. Researchers have recently started generating machine-checkable proofs of correctness for their protocols, since paper proofs, no matter how formal, can contain serious errors.[25] In some cases, the proof of correctness encompasses the implementation, as well. In all cases, the systems proven correct have been much smaller and simpler than ours.

Ridge[21] proves the correctness of a persistent message queue; however, his system is substantially smaller in scale than ours and has no proven liveness properties. Schiper et al.[22] verify the correctness, but no liveness properties, of a Paxos implementation. However, they do not verify the state machine replication layer of this Paxos implementation, only the consensus algorithm, ignoring complexities such as state transfer. In contrast to IronFleet, which exploits multiple levels of abstraction and refinement, their 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 complex distributed systems.

Verdi[23,24] implements verified distributed systems. Its verified system transformers convert a developer's implementation in a simplified environment into an equivalent implementation that is robust in a more hostile environment, offering a clean approach to composition. Unlike IronRSL, Verdi does not prove any liveness properties and its current implementation of Raft does not support verified marshalling and parsing, state transfer, log truncation, dynamic view-change timeouts, a reply cache, or batching.

8. SUMMARY AND FUTURE WORK

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+[13] 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/s (IronRSL) and 28,800 requests/s (IronKV), performance competitive with unverified reference implementations.

In the future, we plan to address two of IronFleet's limitations. First, the performance of even state-of-the-art verification tools limits the scale of the systems we can easily verify. For instance, for every system invariant, we must prove that no action can invalidate that invariant. Automated reasoning handles this with little developer burden when there are tens of actions, but likely not when there are thousands. To fix this, we will require stronger modularity, for example, to enable efficient verification that one component's actions do not interfere with another component's invariants. Another limitation of IronFleet is that it allows concurrency only among processes, not among threads that share memory. The software verification community provides a variety of approaches, such as ownership and separation logic, to address this problem. We plan to make such approaches practical in the context of automated verification of large-scale systems. C

References

1. Bolosky, W.J., Douceur, J.R., Howell, J. The Farsite project: a retrospective. *ACM SIGOPS Oper. Syst. Rev. 41*, 2 (Apr. 2007), 17–26.
2. de Moura, L.M., Bjørner, N. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).
3. Detlefs, D., Nelson, G., Saxe, J.B. Simplify: A theorem prover for program checking. *J. ACM 52* (2003), 365–473.
4. Fischer, M.J., Lynch, N.A., Paterson, M.S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (Apr. 1985), 374–382.
5. Floyd, R. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics* (1967). American Mathematical Society, 19–32.
6. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of USENIX OSDI* (Oct. 2014).
7. Hoare, T. An axiomatic basis for computer programming. *Commun. ACM 12* (1969), 576–580.
8. Joshi, R., Lamport, L., Matthews, J., Tasiran, S., Tuttle, M., Yu, Y. Checking cache coherence protocols with TLA+. *J. Formal Methods Syst. Des. 22*, 2 (2003), 125–131.
9. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.M. Mace: Language support for building

distributed systems. In *Proceedings of ACM PLDI* (2007).
10. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst. 32*, 1 (2014), 1–70.
11. Lamport, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 872–923.
12. Lamport, L.. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.
13. Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, Boston, MA, 2002.
14. Leino, K.R.M. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the LPAR Conference* (2010).
15. Lu, T., Merz, S., Weidenbach, C., Bendisposto, J., Leuschel, M., Roggenbach, M., Margaria, T., Padberg, J., Taentzer, G., Lu, T., Merz, S., Weidenbach, C. Model checking the Pastry routing protocol. In *10th International Workshop Automated Verification of Critical Systems* (Düsseldorf, Germany, Sep. 2010).
16. Moraru, I., Andersen, D.G., Kaminsky, M. There is more consensus in egalitarian parliaments. In *Proceedings of the ACM SOSP* (2013).
17. Musuvathi, M., Park, D., Chou, A., Engler, D., Dill, D.L. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation* (2002).
18. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M. How Amazon Web Services uses formal methods. *Commun. ACM 58*, 4 (Apr. 2015), 66–73.
19. Parkinson, M. The next 700 separation logics. In *Proceedings of IFIP VSTTE* (Aug. 2010).
20. Pek, E., Bogunovic, N. Formal verification of communication protocols in distributed systems. In *Proceedings of the Joint Conferences on Computers in Technical Systems and Intelligent Systems, MIPRO* (2003).
21. Ridge, T. Verifying distributed systems: The operational approach. In *Proceedings of the ACM POPL* (Jan. 2009).
22. Schiper, N., Rahli, V., van Renesse, R., Bickford, M., Constable, R. Developing correctly replicated databases using formal tools. In *Proceedings of IEEE/IFIP DSN* (June 2014).
23. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of ACM PLDI* (June 2015).
24. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T. Planning for change in a formal verification of the Raft consensus protocol. In *ACM Conference on Certified Programs and Proofs (CPP)* (Jan. 2016).
25. Zave, P. Using lightweight modeling to understand Chord. *ACM SIGCOMM Comput. Comm. Rev. 42*, 2 (Apr. 2012), 49–57.

Chris Hawblitzel, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill ({chrishaw, emkaprit, lorch, parno, mirobert, srinath, bzill}@microsoft.com), Microsoft Research.

Jon Howell (jonh@jonh.net), Google.

Watch the authors discuss their work in this exclusive *Communications* video. https://cacm.acm.org/videos/ironfleet