Planning for Change in a Formal Verification of the Raft Consensus Protocol

Doug Woos James R. Wilcox Steve Anton
Zachary Tatlock Michael D. Ernst Thomas Anderson
University of Washington, USA
{dwoos, jrw12, santon, ztatlock, mernst, tom}@cs.washington.edu

Abstract

We present the first formal verification of state machine safety for the Raft consensus protocol, a critical component of many distributed systems. We connected our proof to previous work to establish an end-to-end guarantee that our implementation provides linearizable state machine replication. This proof required iteratively discovering and proving 90 system invariants. Our verified implementation is extracted to OCaml and runs on real networks.

The primary challenge we faced during the verification process was proof maintenance, since proving one invariant often required strengthening and updating other parts of our proof. To address this challenge, we propose a methodology of planning for change during verification. Our methodology adapts classical information hiding techniques to the context of proof assistants, factors out common invariant-strengthening patterns into custom induction principles, proves higher-order lemmas that show any property proved about a particular component implies analogous properties about related components, and makes proofs robust to change using structural tactics. We also discuss how our methodology may be applied to systems verification more broadly.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords Formal verification, distributed systems, proof assistants, Coq, Verdi, Raft

1. Introduction

Distributed systems play a critical role in modern computing infrastructure and therefore must be reliable. However, despite billions of dollars invested in extensive testing, major distributed systems still regularly fail in practice. Indeed, on a single day in the summer of 2015, the New York Stock Exchange halted trading, the Wall Street Journal web page went down, and United Airlines was forced to ground all flights, all due to errors in distributed systems [28]. These errors arise even though the developers of these systems are typically highly trained experts with advanced degrees and decades of experience. Without the necessary tools to ensure the correctness of their systems, there is little hope of eliminating errors.

In previous work, we began to address this challenge by building Verdi [39], a framework for implementing and formally verifying distributed systems in the Coq proof assistant [9]. In this paper we describe our primary result to date using Verdi: the first formally verified implementation of the Raft [32] distributed consensus protocol.[1] The original Verdi paper discusses an implementation of Raft as a *verified system transformer*; Raft's correctness, the classic linearizability property, is expressed as correctness of a transformation from an arbitrary state machine to a fault tolerant system [39]. However, our previous proofs were focused only on phrasing linearizability as a VST correctness property; the proofs consisted of about 5000 lines and assumed several nontrivial invariants of the Raft protocol. This paper discusses the verification of Raft as a whole, including all the invariants from the original Raft paper [32]. These new proofs consist of about 45000 additional lines. Combining this with our previous proofs yields a complete proof that our Raft implementation is linearizable. Our effort yielded a verified implementation as well as insights into managing the verification process.

Raft ensures that a cluster of machines presents a consistent view of a state machine to the outside world, even in the presence of machine failures and unreliable message delivery. Broadly speaking, Raft provides similar functionality to the Paxos and Viewstamped Replication protocols [21, 29]. In practice, clusters using such algorithms are often used to store metadata, such as the map from file names to servers in a distributed file system, the locations of components in a service-oriented architecture, or distributed locks in a work queue. Raft is used in this capacity by companies such as CoreOS [31], Facebook [31], and Google [10].

Despite decades of research, distributed consensus protocols remain notoriously difficult to implement [5, 18, 22, 23, 26, 30, 32, 38]. Many of the challenges are inherent to the domain: nodes of an asynchronous distributed system are generally never in global agreement, and these systems are designed to tolerate many types of fault including node crashes and packet drops, duplication, and reordering. Together, these factors make verification challenging because core system invariants tend to be interdependent. For example, electing a leader is conceptually orthogonal to executing state machine commands, but in order to provide a consistent view, the leader election process must take the state machine into account. Thus proving a property about the state machine typically requires proving that leader election correctly preserves the property. This entanglement causes a change in one invariant to require extensive updates to the invariants and proofs for other components. To successfully build a verified implementation of Raft, we developed a

[1] Our implementation and proofs are available at `https://github.com/uwplse/verdi/tree/cpp2015`.

methodology to support the kinds of changes that frequently arise during large system verification efforts.

One might imagine that the formal verification process proceeds by first writing a detailed pen-and-paper proof and then simply translating the proof to Coq to ensure the absence of any small mistakes. However, the reality is that paper proofs are inevitably incomplete, and seemingly small omissions from the paper proof can require large changes to fix, e.g., stating and proving new inductive invariants. Indeed, this is precisely the reason that machine-checked proofs are useful: people make errors in constructing and checking proofs.

More realistically, formal verification is an iterative process:

1. write a pen-and-paper proof which serves as a high-level verification plan;
2. translate each theorem and proof from the plan to Coq;
3. eventually *get stuck*, discovering that the high-level plan fails to account for some important detail;
4. update the high-level plan to address the new challenge, and change system definitions and theorem statements accordingly;
5. go back to step (2), and *rework* all theorems whose proofs are no longer valid.

In practice, phases (3), (4), and (5) dominate the verification effort.

To manage this process for Raft, we developed a methodology consisting of the following recommendations to reduce rework:

- Extend classical information hiding techniques with **interface lemmas** that support reasoning about hidden definitions (Section 4). This allows changing implementation details without perturbing the rest of the system.
- Capture common patterns of strengthening induction hypotheses in **custom induction principles** (Section 5). These principles employ overapproximations (e.g., on the set of reachable states) to simplify proofs and improve modularity (e.g., allowing operations to be reordered while minimizing proof change as discussed in Section 8).
- Exploit the structure of the system to prove higher-order **affinity lemmas**, which show that any property proved about a particular component can be used to guarantee analogous properties about its related components (Section 6). This allows a developer to update proofs for one component after a change and maintain results for related components without additional rework.
- Ensure that proofs tolerate adding, reordering, and renaming hypotheses with **structural tactics** (Section 7). These tactics support strengthening invariants without changing the proofs that use them.

Some of our methodology's recommendations, such as information hiding, mirror well-known recommendations for designing and implementing well-structured software [33]. In these cases, our contribution is to adapt them to formal verification. Other recommendations, such as our notion of structural tactics, are new. We suspect that some of our recommendations are already known to verification experts, but we believe it is valuable to codify them to help other research groups avoid the expensive and painful process of re-discovering these insights.

Adopting this methodology accelerated our proof efforts and provided benefits exceeding its relatively high initial design costs. When we started verifying Raft, we wrote proofs largely without considering their robustness to change. We quickly discovered that most of our verification effort was devoted to reworking proofs in response to small changes during the iterative development process.

For example, while developing our first proof of a Raft invariant, changing the plan (as part of the iterative process discussed above) meant updating nearly every line of the proof constructed so far.

```
network := {
  nwState   : name -> data;
  nwPackets : list (name * name * payload);
  failed    : list name
}
Inductive step : network -> network -> trace -> Prop :=
| step_drop :
  forall st xs p ys f,
    step {st, xs ++ p :: ys, f} {st, xs ++ ys, f} []
| step_input :
  forall ps h inp,
    not (In h failed) ->
    handleInput h inp (st h) = (st', out, ps') ->
    step {st, ps, f} {st[h := st'], ps ++ ps', f}
         [inp, out]
| step_deliver :
  forall xs ys src dst m st f d' out ps',
    not (In dst f) ->
    handleMessage dst src m (st dst) = (d', out, ps') ->
    step {st, xs ++ (src, dst, m) :: ys, f}
         {st[dst := d'], xs ++ ys ++ ps', f} [out]
| ...
```

Figure 1. Verdi network semantics pseudocode supporting network and node failure. A semantics is defined as a ternary relation over pre- and post-network state and trace of externally-visible I/O events. `failed` is a list of failed hosts, which cannot receive messages or inputs. `st[h := d']` denotes an update to the map `st`, setting the value for key `h` to `d'`.

After we refactored the proof to make our theorem statements and proofs robust against change in the definitions on which they rely, we found that our codebase more easily incorporated fixes during the development process. As another example, Section 8 discusses a case where our approach allowed us to implement a performance optimization while requiring only minor changes to the proof.

In summary, we make the following contributions:

- We present the first formal verification of an implementation of the Raft consensus protocol.
- We present a methodology to reduce rework in response to changes in definitions and theorems during the iterative system verification process.
- We describe our experience applying these techniques during our implementation and verification of Raft and discuss how they may be applied to systems verification more broadly.

2. Verdi Background

Verdi [39] is a general framework for implementing and formally verifying distributed systems in the Coq proof assistant. In Verdi, a distributed system is implemented as a finite set of processes, which communicate by exchanging messages over a network. A system can also communicate with the outside world via input and output. The behavior of a system is specified by its event handlers, including `handleMessage` and `handleInput`, which can be extracted to OCaml and run on real networks using a small trusted shim. Proofs about systems in Verdi are carried out with respect to a *network semantics*, which encodes the system's assumptions about the behavior of the underlying network, e.g., what kinds of failure may occur. Figure 1 shows pseudocode for part of a network semantics that includes network and node failure.

Verdi introduced *verified system transformers* to separate fault tolerance mechanisms from application logic. As shown in Figure 2, a verified system transformer is a function whose input is a system implementation that is verified with respect to one network semantics and whose output is a new system implementation that is

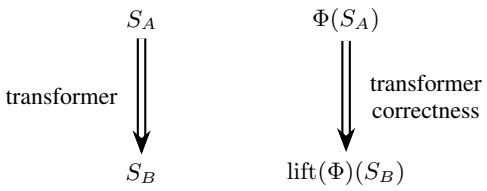

Figure 2. A verified system transformer takes a system (S_A) written against some network semantics and returns a new system (S_B) in another semantics. Its correctness property states that for any property Φ of the original system, a lifted version of that property holds on the transformed system.

verified with respect to a different network semantics. For instance, a verified system transformer could add sequence numbers to messages in order to tolerate message duplication. Each transformer comes with a correctness result that shows that one can reason about the transformed system in terms of the original system.

3. Implementation and Verification of Raft

This section provides background on Raft [32] as well as its implementation and verification in Verdi. Sections 4 to 7 describe the methodology we developed to successfully complete our proof of Raft.

3.1 Raft

Raft is a *state machine replication protocol*. The state machine is a deterministic program that specifies the desired behavior of the cluster as a whole. The state machine processes a sequence of *commands*, which are given by the clients of the cluster. External clients interact with the system as if it were a single node running a single copy of the state machine.

Each node in a Raft cluster simulates a copy of the state machine, and the goal of the protocol is to maintain consistency across the copies. Replication allows the system to continue serving clients whenever a majority of machines are available. However, maintaining consistency among replicas is difficult in the presence of asynchrony, network failures (packet drops, duplications, and reordering) and node failures (crashes and reboots). In particular, the combination of asynchrony and failure means that the nodes in the system are never guaranteed to be in global agreement [11].

Since Raft requires that the state machine it replicates is deterministic, the replicas will be consistent as long as the same client commands are executed on each replica's copy in the same order. Raft's main internal correctness invariant, called state machine safety, captures this property.

Property 1 (State Machine Safety)**.** Each replicated copy of the state machine executes the same commands in the same order.

The list of commands to execute on the state machine is kept in the *log*, and the position of a command in the log is called its *index*. Each node has its own copy of the log, and state machine safety reduces to maintaining agreement between all copies of the log.

Figure 3 shows an example execution of the Raft protocol.[2] Time is logically divided into *terms*, and each term consists of a leader election phase followed by a log replication phase. During leader election, the cluster chooses a *leader*, who coordinates the cluster and handles all communication with clients during the following log replication phase. Nodes are either leaders, *candidates*, or *followers*. Candidates are in the process of trying to become leader.

[2] `https://raft.github.io/` has a visualization of Raft in operation.

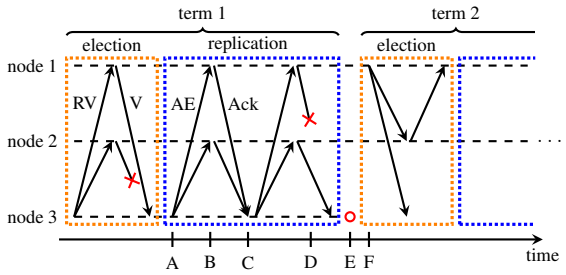

Figure 3. Two terms of the Raft protocol, each consisting of a leader election phase (orange) followed by a log replication phase (blue). Node 3 is the leader of the first term, and node 1 is the leader of the second term. Messages are RequestVote (RV), Vote (V), AppendEntries (AE), or Acknowledgment (Ack). × represents a dropped message and ○ represents a crashed node.

Followers passively obey the leader of the current term and respond to `RequestVote` messages from candidates.

Leader Election If the leader node crashes or is isolated by a network partition (e.g., node 3 at event E in Figure 3), the Raft system elects a new leader. When a node times out waiting to hear from a leader (as node 1 does at event F in Figure 3), it becomes a candidate.[3] A candidate tries to get itself elected as the new leader by sending messages requesting votes from all other nodes. Once a candidate receives votes from a majority of nodes in the system, it becomes the leader. If no candidate successfully wins the election, a new election will take place following a timeout. Requiring a majority ensures that there is only one leader elected per term.

Property 2 (Election Safety)**.** There is at most one leader per term.

Log Replication During normal operation, the cluster is in the log replication phase. In log replication, when a client sends an input to the leader[4] (e.g., at event A in Figure 3), the leader first appends a new *log entry* containing that command to its local log. Then the leader sends an *AppendEntries* message containing the entry to the other nodes in the Raft system. Each other node appends the entries to its log (e.g., at event B in Figure 3), and responds to the leader with an acknowledgment. To ensure that follower logs stay consistent with the log at the leader, AppendEntries messages include the index and term of the *previous* entry in the leader's log; the follower checks that it too has an entry at that index and term before appending the new entries to its log. This consistency check guarantees the following property:

Property 3 (Log Matching)**.** If two logs contain entries at a particular index and term, then the logs are identical up to and including that index.

Once the leader learns that a majority of nodes (including itself) have received the new entry (e.g., at event C in Figure 3), the leader marks the entry as *committed*.[5] Note that the leader need not receive acknowledgments from all nodes before proceeding (e.g., an acknowledgment is dropped at event D in Figure 3, but nodes 2 and 3 constitute a majority). The leader then executes the command

[3] Timeouts are randomized and configured so that candidates rarely compete for leadership. See Ongaro's thesis for more detail on the leader election process [30].

[4] Raft implementations have various mechanisms for clients to locate the leader. In our implementation, clients can send their operations to every node in the cluster until the leader is found.

[5] Committing old entries (those from leaders who failed before completely replicating them) is more complex; see the Raft paper [32] for details.

contained in the committed entry on the state machine and responds to the client with the output of the command. The followers are also informed that they can safely execute the command on their state machines.

Once an entry is committed, it becomes *durable*, in the sense that its effect will never be forgotten by the cluster. To ensure that leader elections do not violate this property, a new leader must have heard of all committed entries created by the previous leader. Therefore, Raft specifies that a node only votes for candidates whose log is at least as advanced as the voter's. Because a newly elected leader was voted in by a majority, it has a log that is at least as advanced as a majority of the cluster. Since any committed entry is present on a majority, every committed entry is present on at least one node that voted for the candidate. The successful candidate's log thus contains every committed entry.

Property 4 (Leader Completeness)**.** A successfully elected candidate's log contains every committed entry.

Client-facing correctness Clients expect to interact with Raft nodes as if the nodes were collectively a single state machine. More formally, clients see a *linearizable* view of the replicated state [17], i.e., if any node responds to a client command c, all subsequently requested commands will execute on a state machine that reflects the execution of c. Section 3.3 gives a precise definition of linearizability.

Property 5. Raft implements a linearizable state machine.

Raft also provides a liveness guarantee: if there are sufficiently few failures, then the system will eventually process and respond to all client commands. To date, we have only verified Raft's safety properties, leaving liveness for future work.

3.2 Raft Implementation using the Verdi Framework

We implemented Raft as a verified system transformer from a single node semantics with no faults to a multi-node semantics with network and machine faults. To use the transformer, a programmer first implements an algorithm as a (non-distributed) state machine in which a single process responds to input from the outside world. Then, Raft transforms this into a system where the original state machine is consistently replicated across a number of nodes. As a result, the programmer can prove properties about the replicated system by reasoning only about the underlying state machine. The Raft transformer produces a system that is proven correct in an environment in which all messages can be arbitrarily reordered, duplicated, delayed, or dropped, and in which nodes can crash and reboot. These faults correspond to the real world failure scenarios that Raft is designed to tolerate.

Figure 4 shows signatures for key parts of Raft as implemented in Verdi.[6] There are two classes of messages: *external* messages (inputs from clients) and *internal* messages exchanged between nodes in the system.

Raft has three kinds of external messages. Nodes running Raft receive `ClientRequest` messages from external clients; each such message contains a command of type `cmd`, which is a parameter of the system. These are delivered in the `step_input` step in Figure 1. Nodes respond with `NotLeader` to indicate that the client should find the current leader or `ClientResponse`, containing the result of the command, once the system has successfully processed a client command. To ensure that network failures do not cause a single client command to be executed multiple times, each `ClientRequest` includes a unique identifier, shown as `uid` in Figure 4. Raft guarantees that a request with a given identifier will only be executed once. Clients can thus repeatedly retry a request; when a client receives a `ClientResponse` with the same `uid`, it knows the command has executed exactly once on the state machine.

Raft has four kinds of internal messages: `AppendEntries` and `Acknowledgment`, used in log replication, and `RequestVote` and `Vote`, used in leader election. These messages correspond directly to the behavior described in Section 3.1.

Our Raft implementation consists of event handlers for external messages, internal messages, and timeouts. Each of these handlers begins with some event-specific code and then calls two bookkeeping functions, `leaderHeartbeat` and `executeEntries`.

```
input := ClientRequest (c : cmd) (uid : nat)...

output := ClientResponse (uid : nat) (r : result) ...
        | NotLeader

handleInput (i : input) :=
  match i with
  | ClientRequest ...
  end;
  leaderHeartbeat();
  executeEntries()

(* internal Raft messages *)
msg := RequestVote ...
     | Vote ...
     | AppendEntries ...
     | Acknowledgment ...

handleMessage (m : msg) :=
  match m with
  | AppendEntries ...
  | Acknowledgment ...
  | RequestVote ...
  | Vote ...
  end;
  leaderHeartbeat();
  executeEntries()

handleTimeout :=
   ...; leaderHeartbeat(); executeEntries()

leaderHeartbeat :=
  (* send AppendEntries to followers *)
  (* mark entries as committed *)
  ...

executeEntries :=
  (* execute entries on the local state machine *)
  (* respond to clients if necessary *)

logEntry := { c     : cmd;
              index : nat;
              term  : nat; ... }

nodeType := Leader | Candidate | Follower

data := { log : list logEntry;
          commitIndex : nat;
          term : nat;
          type : nodeType;
          sm : stateMachine; ... }

init : data := { log := [];
                 commitIndex := 0;
                 term := 0;
                 type := Follower;
                 sm := initialStateMachine; ... }
```

Figure 4. Signatures of key parts of our Raft implementation.

[6] For more detail, see `raft/Raft.v` at `https://github.com/uwplse/verdi/tree/cpp2015`.

`leaderHeartbeat` performs leader-specific tasks, such as sending `AppendEntries` messages to followers and marking entries as committed. `executeEntries` performs tasks that should be done by every server, such as executing committed entries on the state machine.

The local state of each Raft node is given in Figure 4 by the type `data` and includes the log, the index of the most recently committed entry, the node's current term, the node's type (`Leader`, `Candidate`, or `Follower`), and its copy of the state machine. The log is a list of entries, each of which contains a command to be executed on the state machine, its index (position in the log), and the term in which the entry was initially received by the cluster.

The initial state of each node is given by the value `init`. The log is initially empty, no entries are committed, the current term is 0, every node is a follower (nodes will time out and start an election in term 1 to determine the first leader), and the state machine is in its initial state, having not yet processed any commands.

Our verified implementation of Raft in Coq consists 530 lines of code and 50,000 lines of proof, excluding code from the core Verdi framework. It does not support extensions to Raft which are useful in practice, such as dynamic reconfiguration and log compaction. It also includes more data on `Acknowledgment` messages than is necessary. These limitations are not fundamental, but addressing them would increase the proof burden.

3.3 Raft Proof

The behavior of a Verdi system is described by *traces*, which record the interaction between the system and its clients. Internal messages sent between nodes of the system are *not* included in the trace, as they are not observable by clients of the cluster. For example, if Raft is used to replicate a simple key-value store, a valid trace of the resulting system might be:

```
[ClientRequest (Put "x" "hello") 1;
 ClientResponse 1 "";
 ClientRequest (Get "x") 2;
 ClientResponse 2 "hello"].
```

In this execution, a client first sends a `ClientRequest` containing a command to set the key `"x"` to the value `"hello"`; this request is assigned the unique identifier 1. The system then sends a response containing the empty string as its result, which serves as an acknowledgment that the `Put` has taken place. The client then sends a request to read the value of the key `"x"`; the request is assigned the unique identifier 2. Finally, the system responds with the value `"hello"`.

The correctness of a system *transformer* such as Raft is a *relation* that must hold between the traces generated by the transformed system and those generated by the original system. In Figure 2, this relation is called *lift*.

The relational specification of the Raft transformer is that the traces it generates *linearize* (see below) to traces generated by the single-node state machine. Intuitively, linearizability means that once the Raft cluster sends a `ClientResponse` for a command c, the execution of all subsequently issued commands will reflect the execution of c. More precisely, a trace of a replicated system linearizes to a trace of the underlying system if its operations can be reordered to match the underlying trace without moving an incoming command before a previously acknowledged command. For example, in Raft, the system can reorder concurrently issued client requests, but if a request is received after a previous request is acknowledged, then the system must respect that ordering.

We formalize the linearizes-to relation as follows.[7]

Definition 1 (Linearizes-to)**.** Let τ be a trace of inputs and outputs, where each input-output pair is given a unique key. Then τ *linearizes to* a sequence of state machine commands σ if the events of τ can be reordered into a trace τ' such that

1. τ' is sequential, i.e., it consists of alternating inputs and outputs with matching keys;
2. τ' agrees with σ, i.e., they consist of the same sequence of commands, and each output in τ' equals the result given by the corresponding command in σ; and
3. if an output o appears in τ before an input i, then o also appears before i in τ'.

Note that this definition requires τ and σ to contain the same set of commands. Thus, we can define linearizability:

Definition 2 (Linearizability)**.** A trace τ is linearizable if there exists a sequence σ of state machine commands such that τ linearizes to σ.

This definition captures the notion of linearizability, but establishing it directly for Raft would be difficult because it would require strengthening it to be an inductive invariant of the system. Instead, we proved Raft linearizable by relating the system's trace to the local state of each node and the set of packets in the network.

First, we related the trace of the system to each node's local copy of the state machine via state machine safety (Property 1 from Section 3.1). Proving linearizability from state machine safety required proving each of the conditions in Definition 1 by reducing each to an internal property of Raft. We discuss our mechanism for proving such relations in more detail in Section 5.2.

Theorem 1. State machine safety implies linearizability.[8]

Proof. Given an execution trace τ of Raft, we must find σ such that τ linearizes to σ. There is an obvious choice for σ: it is just the sequence of commands executed by the nodes on their local state machines. State machine safety guarantees that the nodes agree on this sequence, so our choice is well defined.

It remains to show that τ linearizes to σ. In other words, we must find τ' such that the conditions of Definition 1 are satisfied. Let τ' be the sequential input–output trace corresponding to σ, i.e., for each command of σ, τ' contains an input immediately followed by the corresponding output for that command. Then τ' is sequential and agrees with σ by construction, and it remains to show that τ' is a permutation of τ that respects the ordering condition (item 3) of Definition 1. Each of these is established as a separate invariant by induction on the execution. □

This result was formalized and proved as part of our work on verified system transformers [39]. The remainder (and vast majority) of our Raft verification effort establishes state machine safety. Since each node executes commands on its state machine as entries become committed in the node's log, state machine safety requires that nodes never disagree about committed entries. The proof of State Machine Safety requires the use of *ghost variables*. Ghost variables are components of system state that are tracked for the purposes of verification but not needed at run time. This state is therefore not tracked in the extracted implementation. For more information, see the Verdi paper [39].

Theorem 2 (State Machine Safety)**.** State machine safety holds for every reachable state of the system.[9]

[7] The relevant Coq development is `raft/Linearizability.v` at `https://github.com/uwplse/verdi/tree/cpp2015`.

[8] This argument is formalized in `raft/RaftLinearizableProofs.v`, along with the lemmas imported by that file.

[9] The top-level proof is in `raft-proofs/StateMachineSafetyProof.v`. The ghost variables required are specified in `raft/RaftRefinement Interface.v` and `raft/RaftMsgRefinementInterface.v`.

```
ghostData := {
  (* list of term, candidate this node voted for,
     log at time of vote *)
  votes : list (nat * name * list logEntry);

  (* term -> list of nodes who voted for
     this node in that term *)
  cronies : nat -> list name;

  (* term, log when this node became leader *)
  leaderLogs : list (nat * list logEntry);

  (* list of term, entry:
     all entries ever present in log at this node*)
  allEntries : list (nat * logEntry)
}

ghostHandleMessage (m : msg) :=
  match m with
  | AppendEntries ... =>
    (* If entries added to log, add to allEntries
       and tag with current term *)
  | RequestVote ...
    (* If voting, add the current term,
       the candidate's name,
       and the current log to votes *)
  | Vote ...
    (* Add sender to cronies at current term *)
    (* If node becomes leader, add current term
       and log to leaderLogs *)
  end
```

Figure 5. Ghost variables used in the verification of Raft

Proof Sketch. First strengthen the induction hypothesis to quantify over ghost state and appropriately constrain each node's history. Next proceed by induction on the step relation, and in each case show that the strengthened hypothesis is preserved. □

The proof of State Machine Safety requires several ghost variables on local data, as well as one on messages. Figure 5 shows pseudocode for the local data ghost state, including the ways in which it is updated in response to incoming messages. Intuitively, each ghost variable stores part of the system's *history*, which is not tracked in the actual implementation but which is necessary for proofs. For example, a node in the system does not actually need to keep a record of every vote that it has every cast; it is sufficient to track only the vote for its current term. However, in order to prove that only one leader is elected per term, the proof uses the `votes` ghost variable. We use the ghost state to establish the Election Safety and Leader Completeness properties, from which we then prove State Machine Safety. As an example, we show how Election Safety follows using these ghost variables.[10]

Theorem 3 (Election Safety)**.** Election safety is true in every reachable state of the system.[11]

Proof Sketch. If a node is a leader, then it has a majority of nodes in its `cronies` for that term. A node h does not appear in `cronies` at a node h' unless h' is in `votes` at h for the same term. A node only votes for one leader for each term. If there are two leaders for one term, at least one node h must be in `cronies` at both leaders since they each have a majority. That node must have voted for both of them at that term, so they must be the same node. Therefore, Election Safety holds. □

[10] The proof of Leader Completeness is available in `raft-proofs/LeaderCompletenessProof.v`.

[11] See `raft-proofs/OneLeaderPerTermProof.v`.

4. Information Hiding

Sections 4 to 7 describe our recommendations and detail how we applied them to complete the Raft proof. This section shows how to extend classical information hiding techniques with **interface lemmas** that support formal reasoning about hidden definitions.

In software development, hiding information from clients (e.g., callers of functions) prevents them from depending on implementation details and thus enables code to be updated without requiring any change to clients. Applying these software engineering principles in the context of proof assistants can significantly reduce the cost of rework in response to the inevitable changes that arise during the iterative system verification process. However, to apply traditional information hiding techniques in the context of Coq requires enriching the traditional notion of interface to support formally reasoning about hidden definitions, rather than just using them.

The core challenge to applying information hiding techniques in Coq arises due to the nature of equality as it is defined in Coq. Coq relies on *definitional equality*, which captures the computational behavior of terms. Since evaluation depends on a term's definition, techniques that hide the details of how a term is defined typically also hide that term's definitional equalites, and thus make equality proofs over that term much more difficult or impossible.

Typical systems developments in Coq try to ease the proof effort by exposing all details of their functions and type definitions.[12] The result is predictable: once the system has been verified, it tends to "freeze", since making any change to one part of the system impacts many other parts of the system and requires reworking numerous proofs. To address this difficulty, we recommend hiding the definition of functions and datatypes, but enriching their interfaces with all the necessary lemmas to reason about them throughout the rest of the system.

Recommendation 1. Hide the definitions of functions and types behind interfaces, and expose only the facts needed through lemmas in the interface.

Following this recommendation, definitions are only unfolded in proofs of their interface lemmas. Then all other proofs in the rest of the system must be completed in terms of the interface lemmas. This allows details of the implementation to be changed without affecting any clients, as long as all the interface lemmas can still be proven for the updated implementation. It is difficult to determine a *complete* set of interface lemmas in advance. When one discovers that the interface is not sufficient to prove a particular theorem, one can add additional lemmas to the interface without forcing any other clients to change. Less often, the interface must be changed in response to an implementation change. This may require updating clients, but only those that use the particular parts of the interface that have been updated.

Throughout our Raft proof, we specified and proved interface lemmas characterizing the behavior of all message handlers and helper functions; two examples are shown in Figure 6. Many of these interface lemmas "overlap," in that they are more general versions of another interface lemma which we found to be too specialized to its original use. However, instead of changing the interface and updating numerous proofs, our approach allowed us to simply extend the interface and continue making progress. Also, these lemmas often specify the behavior of functions (especially message handlers) at various levels of detail depending on the needs of the client (i.e., the invariant proof that uses the specification

[12] There are a few notable exceptions to this rule [3, 12, 25]. For example, the Coq standard library includes implementations of some data structures that use module signatures to hide implementation details. This is a step in the right direction, but such techniques should be more widely used, even when a clean interface is not available in advance, and so some churn is expected.

```
(* Simple specification lemma *)
Lemma handleTimeout_log :
  forall h st out st' l,
    handleTimeout h st = (out, st', l) ->
    log st' = log st.

(* More complex specification lemma *)
Lemma handleAppendEntries_type :
  forall h st st' ps,
    handleAppendEntries h st ... = (st', ps) ->
    type st' = Follower \/ (type st' = type st /\
      currentTerm st' = currentTerm st).
```

Figure 6. Two specification lemmas. The first states that the timeout handler never modifies the Raft log. The second states that the `AppendEntries` handler either maintains the node's type and term or changes the node's type to `Follower`.

lemma). For example, for some invariants, the only fact needed about Raft's timeout handler is that it does not modify the log. When proving an invariant about Raft, we never unfold the definition of a handler function, and instead always apply the relevant specification lemmas. This makes invariant proofs more maintainable, as they will continue working even if the definition of the handler function changes, as long as the new definition leaves the log unchanged.

5. Custom Induction Principles

The majority of theorems in our Raft verification establish properties about the system by induction over the step relation that defines potential network behaviors. In building modular and maintainable proofs of these theorems, we employed common patterns of strengthening induction hypotheses. We captured these patterns in **custom induction principles**. These principles typically use overapproximations to simplify proofs and improve modularity.

Recommendation 2. Factor out common inductive arguments into custom induction principles.

Developing these custom induction principles allows a common reasoning pattern to be proved once and then used throughout the development. It also helps ease proving by handling various common bookkeeping details. Additionally, these custom induction principles improve proof maintainability: some system changes (e.g., as discussed in Section 8) maintain the validity of induction patterns, meaning that the induction principle must be reproved but proofs using the principle continue to be valid without any modification. These principles can be implemented by either creating a new `Inductive` type in order to use its automatically generated induction principles (see Section 5.1) or by proving a theorem that provides a new induction principle for an existing type (see Section 5.2).

5.1 Intermediate Reachability

As discussed in Section 3.2, the implementation of Raft in Verdi consists of event handlers which process client requests, timeouts, and internal messages exchanged between nodes of the system. All of these handlers call two helper functions, `leaderHeartbeat` and `executeEntries`, which run on every event and perform common tasks such as committing log entries and executing commands.

To show that a property is an invariant of a Verdi system, one typically proceeds by induction on the execution of the system. The base case requires showing that the property is true for the initial state of all nodes in the empty network, before any events are handled. The induction case requires showing that the property is preserved by all event and message handlers.

In Raft, since each top-level handler consists of a branch on the kind of event followed by specialized handling code for that

```
Inductive raft_intermediate_reachable : state -> Prop :=
| RIR_init : raft_intermediate_reachable init
| RIR_step :
  forall net net',
    raft_intermediate_reachable net ->
    step net net' ->
    raft_intermediate_reachable net'
| RIR_leaderHeartbeat :
  forall net net',
    raft_intermediate_reachable net ->
    one_node_leader_heartbeat net net' ->
    raft_intermediate_reachable net'
| ...

Lemma RIR_properties_inductive :
  forall (P : network -> Prop),
    (forall net,
      raft_intermediate_reachable net -> P net) ->
    (forall net, step_star init net -> P net)
```

Figure 7. Intermediate reachability is defined inductively as a predicate on states. States reachable via the network semantics' step relation are reachable. States reachable via Raft-specific actions such as a leader heartbeat are also reachable.

event, a direct proof would proceed by case analysis on the kind of event and then reason about each branch independently. The three top-level handlers make calls to `leaderHeartbeat` and `executeEntries`, whose tasks are independent of the event being handled. Direct proof by case analysis would thus require reasoning about `leaderHeartbeat` and `executeEntries` multiple times. Instead, these cases should each be proved to preserve the invariant once. To do so requires constructing intermediate states of the system that occur before and after the calls to these helper functions. These intermediate states may not themselves be reachable under the network semantics. Instead, we introduce the notion of *intermediate reachability* (shown in Figure 7), which rephrases the network semantics into event specific handlers, helper functions, and the network and machine faults. Note that network states which are reachable via this relation are not necessarily reachable in the running system; the set of intermediate-reachable states is a strict superset of the set of actually reachable states. Thus, if a property holds for all intermediate-reachable states, it is guaranteed to hold for all actually reachable states. To use intermediate reachability to prove an invariant of Raft, we prove that event- and message-specific handling code, as well as `executeEntries` and `leaderHeartbeat` all preserve the invariant.

5.2 Trace Relations

Proving linearizability requires relating the external trace of events generated by Raft to its internal state and previous input events. In general, one may wish to show that a trace property implies an internal state property, or that a state property implies a trace property. For example, a client response should only be present in the trace when the relevant command has been executed on the state machine. As an example in the other direction, Raft should never execute a command on the state machine that was not requested by some client. Verdi provides a general method of showing relationships of both types, which we refer to as *trace relations* and *inverse trace relations*.[13]

A trace relation shows that if a trace property `T` holds on a particular execution, then a state property `R` is guaranteed to hold on the final state of that execution, under appropriate assumptions about `T` and `R`. In particular, a trace relation requires that `R` is stable,

[13] See `core/TraceRelations.v`, `core/InverseTraceRelations.v`

T is initially false, T is decidable,[14] and in any step where T becomes true, R is guaranteed to also be true.

```
Variable (T : trace -> Prop) (R : network -> Prop).
Class TraceRelation := {
  R_stable : forall net net' ev,
    R net -> step net net' ev -> R net';
  T_false_init : ~ T [];
  T_dec : forall t, decidable (T t);
  T_implies_R : forall net net' t,
    step_star init net t ->
    step net net' ev ->
    ~ T t -> T (t ++ [ev]) -> R net' }
```

In the example above, T is "command c has a response in the trace," and R is "command c has been executed by at least one state machine." In this example, R is stable because nodes never undo the execution of a command; T is initially false, since the empty trace contains no responses; T is decidable, assuming decidable equality of commands; and T implies R since a node outputs a response only after executing the command.

One can then show that trace relations are valid, in the sense that any execution satisfying T also satisfies R.

```
Theorem trace_relation_valid :
  forall (TR : TraceRelation) net t,
    step_star init net t ->
    T t -> R net.
Proof.
(* by induction on step_star.
   in the base case, T_false_init contradicts
     the hypothesis T t.
   in the inductive case,
     decide whether T holds before the step.
   if so, then induction hypothesis shows that
     R is true in the pre state, and R_stable
     implies that R is true in the post state.
   if not, T_implies_R applies to show that R
     is true in the post state. *)
Qed.
```

Inverse trace relations are proved similarly, except with the roles of T and R flipped. Thus R should be false initially, T should be stable, and if R is false, but a step causes it to become true, then T must be true on the resulting trace.

6. Affinity Lemmas

In many systems, proving a property about one component immediately implies analogous properties about related components. We recommend taking advantage of such relationships by proving higher-order *affinity lemmas*, which show that a property established for a particular component immediately guarantees an analogous property for its related components.

Recommendation 3. Exploit relationships between system components to show that properties established for a particular component imply analogous properties for related components.

In our proof of Raft, we used two instances of this technique, both related to the Raft log. The first shows that any invariant of the log is also an invariant of log data structures elsewhere in the system, including on ghost variables. The second shows that any invariant of the data on an `AppendEntries` message is also an invariant of the data on an `Acknowledgment`.

[14] Coq is based on constructive logic, which allows case analysis only on propositions proved to be *decidable*, i.e., those for which there is a computable function that decides whether the proposition is true or not.

```
Lemma AE_message_symmetry' :
  forall net p,
    raft_intermediate_reachable net ->
    In p (nwPackets net) ->
    body p = Acknowledgment ... ->
    exists net' q,
      raft_intermediate_reachable net' /\
      src q = dst p /\ dst q = src p /\
      body q = AppendEntries ... /\
      nwPackets net' = q :: nwPackets net.
 Proof.
   (* by induction on raft_intermediate_reachable.
      an Acknowledgment is sent only in response
      to an AppendEntries message, which can be
      duplicated before delivery to provide q. *)
 Qed.

Lemma AE_message_symmetry :
  forall P,
    (forall net p,
      raft_intermediate_reachable net ->
      In p (nwPackets net) ->
      body p = AppendEntries data ->
      P (nwState net) data) ->
    forall net p,
      raft_intermediate_reachable net ->
      In p (nwPackets net) ->
      body p = Acknowledgment data ->
      (lift_to_entries P) (nwState net) data.
```

Figure 8. Symmetry lemma for `AppendEntries` messages. This lemma relates the data `AppendEntries` and `Acknowledgment` messages, showing that any property true of one is true of the other.

6.1 Representation Invariants on Logs

There are several representation invariants Raft maintains on logs. For example, our implementation of Raft represents logs as lists of entries, sorted in decreasing order of index. Logs never contain two entries with the same index, and all indices are greater than 0. To complete our proof of safety, we needed these properties on all the logs in Raft, including the log at each host as well as logs in ghost variables. Instead of proving these properties for each occurrence of a log in Raft we proved an affinity lemma: if any property holds of every entry in a node's log, then it is also true of every entry in the logs in ghost variables.[15]

```
Lemma votes_affinity :
  forall (P : logEntry -> Prop),
    (forall net h e,
      raft_intermediate_reachable net ->
      In e (log (nwState net h)) -> P e) ->
    forall net h e n,
      raft_intermediate_reachable net ->
      In (n,e) (allEntries (nwState net h)) -> P e.
Proof.
  (* by induction on raft_intermediate_reachable,
     since the only entries present in allEntries
     were at one time also present in the log *)
Qed.
```

6.2 Message Symmetry

Raft nodes exchange messages to (1) hold leader elections and (2) replicate state machine operations. Each of these two operations has its own request and reply messages. Since nodes only send replies in response to requests, the existence of a reply guarantees the past existence of a corresponding request. Furthermore, since packets can

[15] See `raft/GhostLogsLogPropertiesInterface.v`.

be arbitrarily duplicated and reordered, any packet that existed in the past could have been duplicated and then not delivered, leaving a copy of it in the current network. Thus in any reachable network where there exists a reply packet p, there exists another reachable network with identical state at each host and the same network but with an additional request packet whose metadata corresponds to that of p. As shown in Figure 8, we use this fact to show that any invariant relating the metadata on each `AppendEntries` message to the global node state holds on the reply messages as well.[16]

Note that both affinity lemmas described in this section are *higher-order*: they hold for all *properties*. This kind of quantification over properties is very comfortable in Coq, and improves the usability of affinity lemmas.

7. Proof Engineering

This section describes some concrete "proof engineering" techniques that we found significantly improved our experience verifying Raft. These techniques address lower-level concerns than the techniques described in the preceding sections. Section 7.1 describes a design principle for *structural tactics* which are robust to changes in the proof context in which they run. Section 7.2 describes a code structuring approach to separate proofs from the statement of their theorems which drastically improves build times and thus accelerates the iterative edit-recompile process.

7.1 Tactics for Robust Development

There are several competing proof styles in the Coq community, ranging from using only the built-in tactics stitched together in a "tactic soup", to small-scale reflection [13], to full tactic automation in the style of Chlipala's CPDT textbook [7], and each provides different development and maintenance tradeoffs. We ultimately settled on a middle ground between full automation in Chlipala's style and the more traditional "tactic soup" approach. In particular, we carefully avoid using any automatically generated names or relying on the order of hypotheses. An initial version of this approach was advocated to us by Greg Morrisett [15].

Recommendation 4. Make proof scripts robust against renaming and reordering hypotheses by not relying on automatically generated hypothesis names and hypothesis ordering.

As an example, given a hypothesis that is an equality, instead of rewriting by using its name, one instead invokes a custom `find_rewrite` tactic, which searches the proof context for an equality, and rewrites using it. This allows the rewrite to continue working even if the surrounding definitions, hypothesis names, or lemma names change in the future. If instead the call to the rewrite tactic explicitly used an automatically generated name, then changing any definition that caused a different set of hypotheses to be present when the rewrite tactic is run will cause the step to fail or rewrite the wrong subterm.

Consider the following example lemma.

```
Variable (A B : Type) (f g : A -> B) (P : B -> Prop).
Definition eg : Prop := (forall x, P (g x)) ->
  (forall x, f x = g x) -> forall x, P (f x).
Lemma eg_soup : eg.
Proof. unfold eg. intros. rewrite H0. auto. Qed.
```

This lemma proves that for any functions `f` and `g` of type `A -> B`, if a predicate `P` holds on all outputs of `g`, and if `f` and `g` are extensionally equal, then `P` also holds on all outputs of `f`. While obvious and simplistic, this lemma and its proof are similar to those that are developed in practice. The "tactic soup" proof above unfolds the statement of the lemma and then uses the `intros` tactic to move the hypotheses into the context, assigning them automatically generated names. Then, the goal is rewritten using the extensional equality between `f` and `g`, at which point the tactic `auto` can finish the proof using the fact that `P` holds on all outputs of `g`.

To illustrate why this proof is not robust in the face of changes to the underlying definitions, consider adding a new consequence, `Q`, to the conclusion of the lemma.

```
Variable Q : B -> Prop.
Definition eg : Prop := (forall x, P (g x)) ->
  (forall x, P (g x) -> Q (g x)) ->
  (forall x, f x = g x) ->
  forall x, P (f x) /\ Q (f x).
```

The new statement adds a hypothesis stating that if `P` is true on an output of `g`, then `Q` is also true on that output of `g`. Under this hypothesis, it follows that `P` and `Q` are true of all outputs of `f`.

Now consider how to update the old proof to this new context.

```
Lemma eg_soup : eg.
Proof. unfold eg. intros. rewrite H1. auto. Qed.
```

The proof is almost identical to the previous one, except that the automatically generated name of the extensional equality hypothesis has changed, and so the proof had to be updated to use the new name.

A structural proof of the lemma achieves context independence by using the `find_rewrite` tactic to search for a hypothesis and rewrite by it in the goal.

```
Lemma eg_structural : eg.
Proof. unfold eg. intros. find_rewrite. auto. Qed.
```

When the new hypothesis and conjunct are added, the proof script can remain entirely unchanged.

This illustrates a well-known downside to using automatically generated names, and many users of Coq advocate explicitly assigning names to hypotheses on introduction. Explicit names certainly improve maintainability and readability of proofs, but they do not achieve the full benefits of structural tactics, since the name assignment must be manually updated whenever the context is changed. These changes seem trivial in a small-scale example, but become a major pain point in real-world proof developments.

More generally, instead of using hypothesis names, we use Coq's tactic language, Ltac, to declaratively specify which hypothesis should be used. These Ltac snippets can either be designed on a case-by-case basis, or packaged into generally useful *structural* tactics, such as `find_rewrite`, which finds an equality anywhere in the contexts and rewrites by it somewhere else in the context or goal. We have found that this development style leads to structural properties for our tactic scripts. Adding additional hypotheses, removing redundant hypotheses, and reordering hypotheses should not cause proof scripts to break. These properties correspond to the well-known weakening and exchange properties enjoyed by standard type systems [34].[17]

Weakening A type system satisfies weakening if the typing judgment is invariant under adding irrelevant variables to the typing context. The analogous property for tactic scripts is that proofs should work when hypotheses are added. In practice, this change arises when a lemma is discovered to be unprovable, and an additional hypothesis must be added to make the lemma true. If the partially developed proof has the weakening property, then it will still work when the additional hypothesis is added. The developer can then proceed to leverage the new hypothesis to complete the proof.

[16] See `raft/AppendEntriesRequestReplyCorrespondenceInterface.v`.

[17] The third standard structural property, contraction, corresponds to removing redundant hypotheses from the context. Although avoiding hypothesis names does make our proofs robust against this change, we have not encountered a need to support it in practice.

Exchange A type system satisfies exchange if the typing judgment is invariant under permutations of variables bound in the typing context. The analogous property for tactic scripts is that proofs should work when hypotheses are reordered. In practice, this change arises when an Ltac match statement that used to select one hypothesis now selects a different one. This occurs when the match pattern is ambiguous and the hypotheses have been reordered, since ambiguity is resolved by Ltac using hypothesis ordering. To combat this problem, we strive to make our match patterns specific enough that they match only a single hypothesis in the context. While this rule of thumb is not always sufficient (e.g., adding a new hypothesis that matches a previously unambiguous pattern), we have found it to work well in practice.

7.2 Separating Theorems from Proofs

Our proof of Raft's safety consists of 90 invariants, whose proofs may depend on other invariants. In Coq, modifying a proof P causes all other proofs that depend on P to be rechecked, and in a large development, this has a significant cost in terms of developer time. We address this by separating theorems from their proofs, analogous to the way interfaces are separated from implementations in software engineering. Theorem statements are placed in an interface, and their proofs are expressed as an implementation of the interface. When the proof of one theorem depends on another, the downstream proof imports the interface of the upstream theorem. Note this is different from other developments which may use modules to separate major system components into distinct namespaces, but typically include the proofs in the interface. Thus if one component changes, all dependent components must be rechecked even if they did not require any changes for their proofs to continue to work.

Recommendation 5. Separate theorem statements from their proofs using interfaces.

This approach cuts all dependencies between proofs, which allows proof checking to proceed completely in parallel after the interfaces have been typechecked. Since the proofs themselves take much longer to execute and check than the interfaces, this leads to radically faster build times (in our Raft development, this made rechecking proofs after edits over 100× faster). Furthermore, editing a proof no longer requires re-checking other proofs that depend on it, as long as the interface is not modified. With this approach, one has to rebuild less frequently, and rebuilds take less time.[18]

For example, consider a lemma `P` that is used to prove a top-level theorem `Q`. To support editing the proof of `P` without rechecking the proof of `Q`, create an interface (using Coq's type class mechanism) containing the statement of `P`.

```
(* File: PInterface.v *)
Class P_interface := { P_is_true : P }.
```

To export a proof of `P`, provide an implementation of the interface (i.e., an instance of the type class).

```
(* File: PProof.v *)
Lemma P_proof : P.
Proof. (* ... *) Qed.

Instance P_implementation : P_interface.
Proof. constructor. apply P_proof. Qed.
```

To use `P` to prove the top-level theorem `Q`, assume an arbitrary instance of `P_interface`.

```
(* File: Q.v *)
Context {Pi : P_interface}.
Lemma Q_proof : Q.
Proof. assert P by apply P_is_true. (* ... *) Qed.
```

Finally, to check `Q` end-to-end, plug the concrete instance of the proof of `P` into the proof of `Q`.

```
(* QEndToEnd.v *)
Theorem Q_end_to_end : Q.
Proof. apply (@Q_proof P_implementation). Qed.
About Q_end_to_end. (* No dependence on P_interface. *)
```

During proof development, the proof of `P` can be edited independently of the proof of `Q`. In particular, a change to the proof of `P` does not force a developer working on the proof of `Q` to rebuild the entire codebase. The final end-to-end check ensures that no circular dependencies exist among the interfaces. This check is important, but rarely fails, and so developers need not check it during normal proof development. We have found it useful to set up a continuous integration server to check the end-to-end condition whenever a change is committed.

In our proof of Raft, each invariant is stated in a separate interface, which is implemented by the proof of the invariant. To resolve dependencies between proofs, we have a single end-to-end file that imports all theorems and all proofs and connects them appropriately. Resolving all the dependencies with a single call to the `auto` tactic takes around two minutes. Using interfaces allowed us to build the full end-to-end proof only infrequently while doing development, relying on the continuous integration server to catch any circular reasoning.

8. Experience

We developed the methodology described in the preceding sections over a period of roughly 18 months. Before we applied our recommendations, we found that managing the complexity of the verification process led to slow progress, as we spent the majority of our time reworking proofs in response to changes. In contrast, once all of the techniques in our methodology were developed and applied throughout our codebase, we found that rework in response to common changes was significantly reduced. This allowed us to successfully complete our verification of Raft.

We now describe an example of a change we made to Raft after our initial verification effort was complete, and discuss how our methodology insulated most parts of the system from rework in response to this change. In an early version of our Raft implementation, client responses had unnecessarily high latency because the leader replied to clients before updating the relevant metadata. Thus responses that could be sent now were instead sent when the next event was processed, leading to what clients would observe as a performance bug. After obtaining a complete proof of linearizability, we changed our implementation to send client responses as soon as possible. More concretely, we changed Raft's top-level event handler to call the function that is responsible for marking a request complete before calling the function that sends client responses; this change is illustrated in pseudocode in Figure 9. After making this change, we proceeded to fix the proof of linearizability.

```
(* before *)                (* after *)
handleMessage m :=          handleMessage m :=
  match m with ...            match m with ...
  executeEntries();           leaderHeartbeat();
  leaderHeartbeat()           executeEntries()
```

Figure 9. Pseudocode for the simple change made to our Raft implementation

[18] Coq 8.5 also has a feature that separates theorem statements from their proofs, using `.vio` interface files [2]. We expect to use this feature when it becomes stable.

The decomposition lemma discussed in Section 5 insulates invariants from this reordering. To prove an invariant using the decomposition, one proves that every low-level event and each of the above helper functions preserves the invariant. Crucially, each of these proofs is independent and does not depend on the exact ordering of functions in the top-level event handler. Thus only the proof that the decomposition is itself sound needs to be updated, and this change is relatively straightforward and localized to a single file.

The decomposition only applies to internal invariants, i.e., invariants on Raft's internal state. Invariants about the trace of externally visible events are instead proved using the trace relation technique discussed in Section 5.2. Unfortunately, the requirements of trace relations do not force the proof to be order independent, so each proof needed to be updated individually. There are around 85 internal invariants and 5 external invariants in our development. Updating the decomposition (and thus fixing the 85 internal invariants all at once) required about 3 hours. Fixing the 5 external invariants required about the same amount of time, despite being about an order of magnitude fewer lines of code. A variant of trace relations that incorporates the order-independence of the decomposition would have made these external invariants just as easy to update.

Our experience fixing this performance bug gives anecdotal evidence that our approach pays off in the long run. This example is admittedly somewhat of a best case, since the change fits so nicely into the strengths of the decomposition. On the other hand, we were eager to make this change in part because we knew the proofs would be relatively easy to update.

9. Related Work

Our verification of Raft builds on work in distributed systems verification (Section 9.1), systems verification in general (Section 9.2), and proof engineering (Section 9.3). We briefly survey the most closely related work below.

9.1 Distributed Systems Verification

We discuss several previous projects verifying distributed systems in previous work [39]. Here we briefly describe two of the most closely related projects: EventML/ShadowDB and IronFleet.

EventML [35, 36] is a language for implementing distributed systems. EventML programs can be verified in NuPRL [8] using the Logic of Events [4]. EventML and the Logic of Events have been used to verify an implementation of Multi-Paxos, a total-order broadcast service, used as part of a distributed database [37].

IronFleet [16] is a concurrent effort to build and verify distributed systems using Dafny (an SMT-based program verification toolchain) and a TLA-style proof strategy. This approach enables building practical distributed systems and proving both safety and, unlike our Raft proof in Verdi, *liveness* properties. Also unlike our Raft implementation in Verdi, IronFleet's implementation of Paxos supports many important practical features including verified marshaling and parsing, state transfer, and log truncation. Compared to IronFleet, Verdi's verified system transformers provide a more compositional approach to building fault tolerant systems.

9.2 Systems Verification

The research community has applied proof assistants to verify implementations of several major systems. CompCert is a verified C compiler written in Coq [24]. To establish equivalence between an input C program and the corresponding output assembly, Compcert proves a bisimulation between the two programs. However, instead of proving a forward simulation and backward simulation for every transformation, CompCert instead uses deterministic intermediate languages, proves only a forward simulation for each transformation, and applies to a general, higher-order result which shows that any forward simulation into a deterministic semantics implies a bisimulation. This decomposition inspired some of our design choices which eventually developed into our recommendations for adapting to change.

Bedrock [6], Ynot [27], and the Verified Software Toolchain [1] are verification frameworks based on separation logic and are useful for verifying imperative programs in Coq. The Verified Software Toolchain uses opaque definitions to hide information in a way similar to our recommendations, but does so for reasons of automation rather than proof maintainability. By exposing only a small set of axioms (for instance, those of separation logic), the Verified Software Toolchain enables automated proofs for a large set of proof obligations, without the proof search getting bogged down reasoning about every definition in the system.

seL4 is an OS kernel, verified using the Isabelle/HOL proof assistant [19]. As in our verification of Raft, the bulk of the effort in verifying seL4 was in proving invariants about the internal state of the system. These invariants are then used to prove that the C code implements the abstract specification.

9.3 Proof Engineering

There is a small body of work on improving the development of machine-checked proofs. We believe that this branch of software engineering is becoming increasingly important, and look forward to more research in this area. In particular, how best to develop proofs which are robust in the face of changes to related definitions is an open question.

Our interface lemmas, described in Section 5 are analogous to the use of modules and Abstract Data Types (ADTs) in classical software engineering [33]. Just as an interface to an ADT hides implementation details from the client so too does this unfolding methodology hide implementation details from other proofs (verified clients). In the classic ADT setting, operations are exposed as part of the interface. In the verified setting, the interface also needs to make explicit the specifications of its operations. The definitions of the operations are then only unfolded in the proofs showing that the operations satisfy their specifications.

In their verification of seL4, Klein et al. identify this core challenge and describe four categories of changes, ordered by their relative verification cost: (C1) local, low-level code changes; (C2) adding new, independent features; (C3) adding new, large, cross-cutting features; (C4) fundamental changes to existing features [19, 20]. In particular, they note that categories C3 and C4 have a disproportionately high cost of re-verification. In this terminology, the goal of our methodology is to structure the development such that common changes are in categories C1 or C2, rather than C3 or C4, thus significantly reducing re-verification costs.

The proof engineering techniques presented in this paper represent a particular point in the design space: extensive higher-order reasoning combined with a modular proof architecture and tactics focused on lightweight automation and robustness to change. Chlipala advocates for heavy proof automation, using powerful, purpose-built tactics to dispatch proof obligations [7]. Gonthier and others have advocated for using *canonical structures* to extend Coq's built-in typechecker to support automation [14]. We see our approach as a compromise between the "tactic soup" style of many proofs, which pays no attention to maintainability or automation, and Chlipala's heavy automation, which requires heavy up front investment in tactics.

The Ssreflect library [13] provides an alternative to Coq's default tactic language. Many of Ssreflect's tactics involve "bookkeeping", that is, managing the hypotheses which appear in the goal and in the context. Ssreflect disallows the use of automatically generated hypothesis names, requiring users to explicitly assign names to hypotheses. In contrast, using our methodology, one avoids referring to specific hypotheses at all by using tactics which find the correct hypothesis wherever it is in the context.

10. Conclusion

We presented the first formally verified implementation of the Raft consensus protocol. Our proof establishes Raft's primary safety property, namely that it is a linearizable replicated state machine. Based on our experience, we developed a proof engineering methodology of *planning for change*. Our methodology adapts classical information hiding techniques to the context of proof assistants, factors out common invariant strengthening patterns into custom induction principles, proves higher-order lemmas that show any property proved about a particular component imply analogous properties about related components, and makes proofs robust to change using structural tactics. We believe these techniques are generally applicable, and we hope to see more discussion of proof methodology and engineering techniques in the community.

Acknowledgments

The authors thank Adam Chlipala, Nate Foster, Gregory Malecha, Diego Ongaro, Karl Palmskog, Bryan Parno, Benjamin Pierce, Vincent Rahli, Daniel T. Ricketts, Ilya Sergey, Xi Wang, Daryl Zuniga, and the anonymous reviewers for helpful feedback on earlier versions of this work.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0963754 and the Graduate Research Fellowship Program under Grant No. DGE-1256082. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This material is based on research sponsored by the United States Air Force under Contract No. FA8750-12-C-0174 and by DARPA under agreement number FA8750-12-2-0107. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[2] B. Barras, C. Tankink, and E. Tassi. Asynchronous processing of Coq documents: from the kernel up to the user interface. In *ITP*, 2015.

[3] J. Bengtson, J. B. Jensen, and L. Birkedal. Charge! - A framework for higher-order separation logic in coq. In *ITP*, 2012.

[4] M. Bickford, R. L. Constable, and V. Rahli. Logic of events, a framework to reason about distributed systems. In *LADA*, 2012.

[5] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC*, Aug. 2007.

[6] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, June 2011.

[7] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, Dec. 2013.

[8] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with The Nuprl Proof Development System*. Prentice Hall, 1986.

[9] Coq Development Team. *The Coq Reference Manual, version 8.4*, Aug. 2012. `http://coq.inria.fr/doc`.

[10] Data Center Knowledge. etcd: the Not-so-Secret Sauce in Google's Kubernetes and Pivotal's Cloud Foundry, 2014. `http://www.datacenterknowledge.com/archives/2014/07/16/etcd-secret-sauce-googles-kubernetes-pivotals-cloud-foundry/`.

[11] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2), 1985.

[12] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *TPHOLs*, 2009.

[13] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq System. Technical Report 645, Microsoft Research - Inria Joint Centre, 2009.

[14] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP*, 2011.

[15] Greg Morrisett. Certified Programming and State, June 2014. `https://www.cs.uoregon.edu/research/summerschool/summer14/curriculum.html`.

[16] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: Proving practical distributed systems correct. In *SOSP*, Oct. 2015.

[17] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.

[18] J. Kirsch and Y. Amir. Paxos for system builders. *Dept. of CS, Johns Hopkins University, Tech. Rep*, 2008.

[19] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, Oct. 2009.

[20] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1), 2014.

[21] L. Lamport. The part-time parliament. *TOCS*, 16(2), 1998.

[22] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4), 2001.

[23] B. W. Lampson. The ABCD's of Paxos. In *PODC*, Aug. 2001.

[24] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), July 2009.

[25] G. Malecha. coq-ext-lib. `https://github.com/coq-ext-lib/coq-ext-lib`.

[26] D. Mazieres. Paxos made practical, 2007. `http://www.scs.stanford.edu/~dm/home/papers`.

[27] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, Sept. 2008.

[28] NYTimes. The Stock Market Bell Rings, Computers Fail, Wall Street Cringes, July 2015. `http://www.nytimes.com/2015/07/09/business/dealbook/new-york-stock-exchange-suspends-trading.html`.

[29] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *PODC*, Aug. 1988.

[30] D. Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, Aug. 2014.

[31] D. Ongaro. The Raft consensus website, Nov. 2014. `http://raft.github.io/`.

[32] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, June 2014.

[33] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12), Dec. 1972.

[34] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[35] V. Rahli. Interfacing with proof assistants for domain specific programming using EventML. In *UITP*, July 2012.

[36] V. Rahli, D. Guaspari, M. Bickford, and R. L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using EventML. In *AVOCS*, Sept. 2015.

[37] N. Schiper, V. Rahli, R. van Renesse, M. Bickford, and R. L. Constable. Developing correctly replicated databases using formal tools. In *DSN*, June 2014.

[38] R. Van Renesse. Paxos made moderately complex. *ACM Computing Surveys*, 47(3), 2011.

[39] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and verifying distributed systems. In *PLDI 2015*, June 2015.