Verdi: A Framework for Implementing and Formally Verifying Distributed Systems

James R. Wilcox Doug Woos Pavel Panchekha
Zachary Tatlock Xi Wang Michael D. Ernst Thomas Anderson
University of Washington, USA
{jrw12, dwoos, pavpan, ztatlock, xi, mernst, tom}@cs.washington.edu

Abstract

Distributed systems are difficult to implement correctly because they must handle both concurrency and failures: machines may crash at arbitrary points and networks may reorder, drop, or duplicate packets. Further, their behavior is often too complex to permit exhaustive testing. Bugs in these systems have led to the loss of critical data and unacceptable service outages.

We present Verdi, a framework for implementing and formally verifying distributed systems in Coq. Verdi formalizes various network semantics with different faults, and the developer chooses the most appropriate fault model when verifying their implementation. Furthermore, Verdi eases the verification burden by enabling the developer to first verify their system under an idealized fault model, then transfer the resulting correctness guarantees to a more realistic fault model without any additional proof burden.

To demonstrate Verdi's utility, we present the first mechanically checked proof of linearizability of the Raft state machine replication algorithm, as well as verified implementations of a primary-backup replication system and a key-value store. These verified systems provide similar performance to unverified equivalents.

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

1. Introduction

Distributed systems serve millions of users in important applications, ranging from banking and communications to social networking. These systems are difficult to implement correctly because they must handle both concurrency and failures: machines may crash at arbitrary points and networks may reorder, drop, or duplicate packets. Further, the behavior is often too complex to permit exhaustive testing. Thus, despite decades of research, real-world implementations often go live with critical fault-handling bugs, leading to data loss and service outages [10, 42]. For example, in April 2011 a malfunction of failure recovery in Amazon Elastic Compute Cloud (EC2) caused a major outage and brought down several web sites, including Foursquare, Reddit, Quora, and PBS [1, 14, 28].

Our overarching goal is to ease the burden for programmers to implement correct, high-performance, fault-tolerant distributed systems. This paper focuses on a key aspect of this agenda: we describe Verdi, a framework for implementing practical fault-tolerant distributed systems and then formally verifying that the implementations meet their specifications. Previous work has shown that formal verification can help produce extremely reliable systems, including compilers [41] and operating systems [18, 39]. Verdi enables the construction of reliable, fault-tolerant distributed systems whose behavior has been formally verified. This paper focuses on safety properties for distributed systems; we leave proofs of liveness properties for future work.

Applying formal verification techniques to distributed system implementations is challenging. First, while tools like TLA [19] and Alloy [15] provide techniques for reasoning about abstract distributed algorithms, few practical distributed system *implementations* have been formally verified. For performance reasons, real-world implementations often diverge in important ways from their high-level descriptions [3]. Thus, our goal with Verdi is to verify working code. Second, distributed systems run in a diverse range of environments. For example, some networks may reorder packets, while other networks may also duplicate them. Verdi must support verifying applications against these different fault models. Third, it is difficult to prove that application-level guarantees hold in the presence of faults. Verdi aims to help the programmer separately prove correctness of application-level behavior and correctness of fault-tolerance mechanisms, and to allow these proofs to be easily composed.

Verdi addresses the above challenges with three key ideas. First, Verdi provides a Coq toolchain for writing executable distributed systems and verifying them; this avoids a *formality gap* between the model and the implementation. Second, Verdi provides a flexible mechanism to specify fault models as *network semantics*. This allows programmers to verify their system in the fault model corresponding to their environment. Third, Verdi provides a *compositional* technique for implementing and verifying distributed systems by separating the concerns of application correctness and fault tolerance. This simplifies the task of providing end-to-end guarantees about distributed systems.

To achieve compositionality, we introduce *verified system transformers*. A system transformer is a function whose input is an implementation of a system and whose output is a new system implementation that makes different assumptions about its environment. A verified system transformer includes a proof that the new system satisfies properties analogous to those of the original system. For example, a Verdi programmer can first build and verify a system

PLDI'15 , June 13–17, 2015, Portland, OR, USA
DOI: http://dx.doi.org/10.1145/10.1145/2737924.2737958
Reprinted from PLDI'15, June 13–17, 2015, Portland, OR, USA, pp. 357–368.

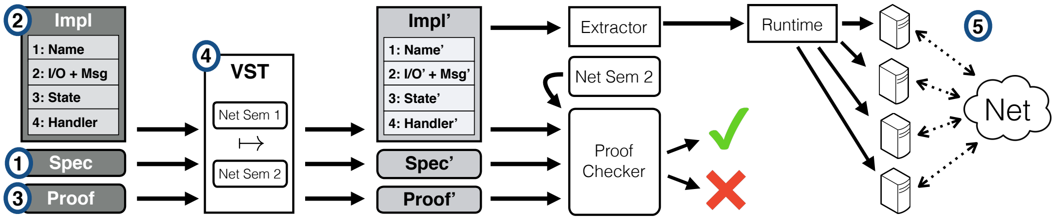

Figure 1. Verdi workflow. Programmers provide the dark gray boxes in the left column: the specification, implementation, and proof of a distributed system. Rounded rectangles correspond to proof-related components. To make the proof burden manageable, the initial proof typically assumes an unrealistically simple network model in which machines never crash and packets are never dropped or duplicated. A verified system transformer (VST) transforms the application into one that handles faults, as shown in the column of light gray boxes in the middle column. Note that the programmer does not write any code for this step. Verdi provides the white boxes, including verified systems transformers (VSTs), network semantics encoding various fault models, and extraction of an implementation to an executable. Programmers deploy the executable over a network for execution.

assuming a reliable network, and then apply a transformer to obtain another version of their system that correctly and provably tolerates faults in an unreliable network (e.g., machine crashes).

Contributions. This paper makes the following contributions: (1) Verdi, a publicly available [37] toolchain for building provably correct distributed systems, (2) a set of formal network semantics with different fault models, (3) a compositional verification technique using verified system transformers, (4) case studies of implementing, and proving correct, practical distributed systems including a key-value store, a primary-backup replication transformer, and the first formally verified proof of linearizability for the Raft consensus protocol [30], and (5) an evaluation showing that these implementations can provide reasonable performance. Our key conceptual contribution is the use of verified systems transformers to enable modular implementation and end-to-end verification of systems.

The rest of the paper is organized as follows. Section 2 overviews the Verdi system. Section 3 details the small-step operational semantics that specify distributed system behavior in different fault models. Section 4 describes how systems in Verdi can be constructed from modular components. Sections 5–7 describe case studies of using Verdi to implement and verify distributed systems. Section 8 evaluates the performance of systems implemented in Verdi. Section 9 discusses related work, and Section 10 concludes.

2. Overview

Figure 1 illustrates the Verdi workflow. The programmer ① specifies a distributed system and ② implements it by providing four definitions: the names of nodes in the system, the external input and output and internal network messages that these nodes respond to, the state each node maintains, and the message handling code that each node runs. ③ The programmer proves the system correct assuming a specific baseline network semantics. In the examples in this paper, the programmer chooses an idealized reliable model for this proof: all packets are delivered exactly once, and there are no node failures. ④ The programmer then selects a target network semantics that reflects their environment's fault model, and applies a verified system transformer (VST) to transform their implementation into one that is correct in that fault model. This transformation also produces updated versions of the specification and proof. ⑤ The verified system is extracted to OCaml, compiled to an executable, and deployed across the network.

The rest of this section describes each of these five steps, using a simple lock service as a running example. The lock service manages a single shared lock. Conceptually, clients communicate with the lock service using the following API: a client requests and releases a lock via the `Lock` and `Unlock` input messages, and the lock service grants a lock by responding with a `Grant` output message.

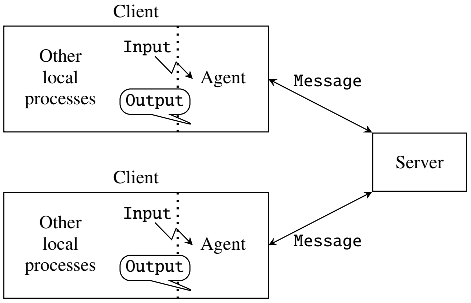

Figure 2. Architecture of a lock service application. Boxes represent separate physical nodes, while dotted lines separate processes running on the same node. Each client node runs an Agent process that exchanges input and output with other local processes. The Agent also exchanges network messages with the Server.

To provide this API, the lock service consists of a central lock Server node, and a lock Agent that runs on every client node, as illustrated in Figure 2. That is, each client node runs a lock Agent along with other client processes that access the API through the Agent. Each lock Agent communicates over the network with the central lock server. The Agent requests and releases the lock with the `LockMsg` and `UnlockMsg` network messages, and the server sends a `GrantMsg` network message to notify an Agent when it has received the lock.

2.1 Specification

A Verdi programmer specifies the correct behavior of their system in terms of *traces*, the sequences of external input and output generated by nodes in the system. For the lock service application, correctness requires mutual exclusion: no two distinct nodes should ever simultaneously hold the lock. This mutual exclusion property can be expressed as a predicate over traces:

$$\begin{aligned} &\texttt{mutex}(\tau) := \\ &\quad \tau \;=\; \tau_1 \mathbin{++} \langle n_1,\ \texttt{Grant}\rangle \mathbin{++} \tau_2 \mathbin{++} \langle n_2,\ \texttt{Grant}\rangle \mathbin{++} \tau_3 \\ &\quad \to \langle n_1,\ \texttt{Unlock}\rangle \in \tau_2 \end{aligned}$$

To hold on trace τ, the `mutex` predicate requires that whenever `Grant` is output on node n_1 and then later `Grant` is output on node n_2, there must first be an intervening `Unlock` input from n_1 releasing the lock.

A system implementation satisfies specification Φ in a particular network semantics if for all traces τ the system can produce under that semantics, Φ holds on τ. For the example lock service applica-

```
(* 1 - node identifiers *)
Name := Server | Agent(int)

(* 2 - API, also known as external IO *)
Inp := Lock | Unlock
Out := Grant
(* 2 - network messages *)
Msg := LockMsg | UnlockMsg | GrantMsg

(* 3 - state *)
State (n: Name) :=
  match n with
    | Server => list Name (* head = agent holding lock      *)
                          (* tail = agents waiting for lock *)
    | Agent n => bool     (* true iff this agent holds lock *)

InitState (n: Name) : State n :=
  match n with
    | Server => []
    | Agent n => false

(* 4 - handler for external input *)
HandleInp (n: Name) (s: State n) (inp: Inp) :=
  match n with
    | Server => nop   (* server performs no external IO *)
    | Agent n =>
      match inp with
        | Lock =>                    (* client requests lock   *)
          send (Server, LockMsg)     (* forward to Server      *)
        | Unlock =>                  (* client requests unlock *)
          if s == true then (        (* if lock held           *)
            s := false;;             (* update state           *)
            send (Server, UnlockMsg)) (* tell Server lock freed *)

(* 4 - handler for network messages *)
HandleMsg (n: Name) (s: State n) (src: Name) (msg: Msg) :=
  match n with
    | Server =>
      match msg with
        | LockMsg =>
          (* if lock not held, immediately grant *)
          if s == [] then send (src, GrantMsg);;
          (* add requestor to end of queue *)
          s := s ++ [src]
        | UnlockMsg =>
          (* head of queue no longer holds lock *)
          s := tail s;;
          (* grant lock to next waiting agent, if any *)
          if s != [] then send (head s, GrantMsg)
        | _ => nop (* never happens *)
    | Agent n =>
      match msg with
        | GrantMsg =>      (* lock acquired    *)
          s := true;;      (* update state     *)
          output Grant     (* notify listeners *)
        | _ => nop         (* never happens    *)
```

Figure 3. A simple lock service application implemented in Verdi, under the assumption of a reliable network. Verdi extracts these definitions into OCaml and links the resulting code with a runtime to send and receive messages over the network.

tion, an implementation satisfies `mutex` in a given semantics if `mutex` holds on all the traces produced under that semantics.

2.2 Implementation

Figure 3 shows the definitions a programmer provides to implement the lock service application in Verdi. (1) `Name` lists the names of nodes in the system. In the lock service application, there is a single Server node and an arbitrary number of Agents. (2) `Inp` and `Out` define the API of the lock service — the external input and output exchanged between an Agent and other local processes on its node. `Msg` defines network messages exchanged between Agents and the central Server. (3) `State` defines the state maintained at each node. Node state is defined as a dependent type where a node's name determines the data maintained locally at that node. In the lock service, the Server maintains a queue of Agent nodes, initially empty, where the head of the queue is the Agent currently holding the lock and the rest of the queue represents the Agents which are waiting to acquire the lock. Each Agent maintains a boolean, initially false, which is true exactly when that Agent holds the lock. (4) The handler functions `HandleInp` and `HandleMsg` define how nodes respond to external input and to network messages.

This implementation assumes a reliable network where machines never crash and packets may be reordered but are not dropped or duplicated. These assumptions reduce the programmer's effort in both implementing the application and proving it correct. Section 2.4 shows how Verdi can automatically transform the lock service application into a version that tolerates faults.

When the system runs, each node listens for events and responds by running the appropriate handler: `HandleInp` for external input and `HandleMsg` for network messages. When an Agent receives an external input that requests to acquire or release the lock, it forwards the request to the Server; in the `Unlock` case, it first checks to ensure that the lock is actually held, and it resets its local state to `false`. Because the network is assumed to be reliable, no acknowledgment of the release is needed from the Server. When the Server receives a `LockMsg` network message, if the lock is not held, the server immediately grants the lock, and always adds the requesting Agent to the end of the queue of nodes. When the Server receives an `UnlockMsg` message, it removes a node from the head of its queue of Agents and grants the lock to the next Agent in the queue, if any. When an Agent receives a `GrantMsg` message, it produces external output (`Grant`) to inform other processes running on its node that the lock is held.

The application will be deployed on some network, and *network semantics* capture assumptions about the network's behavior. For this example, we assume a semantics encoding a reliable network. In a reliable network, each step of execution either (1) picks an arbitrary node and delivers an arbitrary external input, runs that node's input handler, and updates the state, or (2) picks a message in the network, runs the recipient's message handler, and updates the state.

Figure 4 shows an execution of the lock service application with two agents. Agents A_1 and A_2 both try to acquire the lock. The service first grants the lock to A_1. Once A_1 releases the lock, the service grants it to A_2. Note that, because our network semantics does not assume messages are delivered in the same order in which they were sent, there is a potential race condition: an agent can attempt to re-acquire the lock before the server has processed its previous release. In that case, the server simply (and correctly) adds the sender to the queue again. Using Verdi, the lock service is guaranteed to behave correctly even in such corner cases.

2.3 Verifying the Lock Service Application

We briefly outline the proof of the `mutex` property for the lock service application in the reliable network environment (i.e., no machine crashes nor packet loss/duplication). The proof that `mutex` holds on all traces of the lock service application consists of three high-level steps: (1) prove an invariant about the reachable node and network states of the lock service application, (2) relate these reachable states to the producible traces, and (3) show that the previous two steps imply `mutex` holds on all producible traces.

The first step proves that all reachable system states satisfy the $\texttt{mutex}_{\texttt{state}}$ property:

$$\texttt{mutex}_{\texttt{state}}(\Sigma,\ P)\ :=$$
$$\quad \forall\, n\, m,\ n \neq m \to \neg\texttt{hasLock}(\Sigma,\ n) \vee \neg\texttt{hasLock}(\Sigma,\ m)$$

$$\texttt{hasLock}(\Sigma,\ n)\ :=$$
$$\quad \Sigma(\text{Agent}(n)) = \texttt{true}$$

The function Σ maps node names to their state, and P is the set of in-flight packets. The property $\texttt{mutex}_{\texttt{state}}$ ensures that at most one Agent node holds the lock at a time.

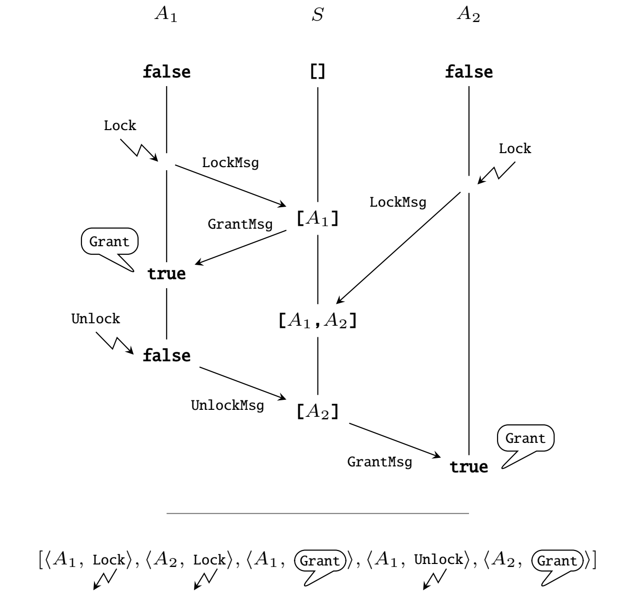
A_1
S
A_2
false
[]
false
Lock
LockMsg
Lock
LockMsg
GrantMsg
$[A_1]$
Grant
true
Unlock
$[A_1, A_2]$
false
UnlockMsg
$[A_2]$
Grant
GrantMsg
true
$[\langle A_1,$ Lock$\rangle, \langle A_2,$ Lock$\rangle, \langle A_1,$ Grant$\rangle, \langle A_1,$ Unlock$\rangle, \langle A_2,$ Grant$\rangle]$

Figure 4. The behavior of the lock service application, with one server S and two agents A_1 and A_2. Each agent starts with the state `false`, and the server starts with an empty queue. Time flows downward. In response to external input (drawn with lightning-bolt arrows) and network messages, the nodes exchange messages and update local state. External output is shown as speech bubbles. The trace of this execution is shown at the bottom; note that only externally-visible events (external input and output) appear in the trace.

A programmer can verify the $\texttt{mutex}_{\texttt{state}}$ property by proving an *inductive state invariant*. A property ϕ is an inductive invariant if both (1) it holds in the initial state, $(\Sigma_0,\ \emptyset)$, where Σ_0 maps each node to its initial state and $\emptyset$ represents the initial, empty network, and also (2) whenever it holds in some state, $(\Sigma,\ P)$, and $(\Sigma,\ P)$ can step to $(\Sigma',\ P')$, then it holds in $(\Sigma',\ P')$.

One inductive state invariant for $\texttt{mutex}_{\texttt{state}}$ is:

$$\begin{aligned}
&(\forall\, n,\ \texttt{hasLock}(\Sigma,\ n) \rightarrow \texttt{atHead}(\Sigma,\ n))\\
&\wedge (\forall\, p \in P,\ p.\texttt{body} = \texttt{GrantMsg} \rightarrow \texttt{grantee}(\Sigma,\ p.\texttt{dest}))\\
&\wedge (\forall\, p \in P,\ p.\texttt{body} = \texttt{UnlockMsg} \rightarrow \texttt{grantee}(\Sigma,\ p.\texttt{source}))\\
&\wedge\ \texttt{at\_most\_one}\ \{\texttt{GrantMsg},\ \texttt{UnlockMsg}\}\ P
\end{aligned}$$

where

$$\begin{aligned}
\texttt{atHead}(\Sigma,\ n) \quad &:= \quad \exists\, t,\ \Sigma(\text{Server}) = n :: t\\
\texttt{grantee}(\Sigma,\ n) \quad &:= \quad \texttt{atHead}(\Sigma,\ n) \wedge \neg\texttt{hasLock}(\Sigma,\ n).
\end{aligned}$$

The first conjunct above ensures that the Server and Agents agree on who holds the lock. The second and third conjuncts state that `GrantMsg` is never sent to an agent that already holds the lock, and that `UnlockMsg` is never sent from an agent that still holds the lock. Finally, the last conjunct states that there is at most one in-flight message in the set $\{\texttt{GrantMsg}, \texttt{UnlockMsg}\}$; this is necessary to ensure that neither of the previous two conjuncts is violated when a message is delivered. We proved in Coq that this invariant is inductive and that it implies $\texttt{mutex}_{\texttt{state}}$; the proof is approximately 500 lines long.

The second step of the proof relates reachable states to the traces a system can produce:

$$\begin{aligned}
&\texttt{trace\_state\_agreement}(\tau,\ \Sigma)\ :=\\
&\quad \forall\, n,\ \texttt{lastGrant}(\tau,\ n) \leftrightarrow \texttt{hasLock}(\Sigma,\ n)\\
&\texttt{lastGrant}(\tau, n)\ :=\\
&\quad \tau = \tau_1 \mathbin{++} \langle n,\ \texttt{Grant}\rangle :: \tau_2 \wedge \forall\, m,\ \langle m,\ \texttt{Unlock}\rangle \notin \tau_2
\end{aligned}$$

This property requires that whenever a `Grant` output appears in the trace without a corresponding `Unlock` input, that agent's flag is true (and vice versa). The proof of this property is by induction on the possible behavior of the network.

The third step of the proof shows that together, $\texttt{mutex}_{\texttt{state}}$ and `trace_state_agreement` imply that `mutex` holds on all traces of the lock service application under the reliable semantics. This result follows from the definitions of `mutex`, $\texttt{mutex}_{\texttt{state}}$, and `trace_state_agreement`.

2.4 Verified System Transformers

We have proved the `mutex` property for a reliable environment where the network does not drop or duplicate packets and the server does not crash. Assuming such a reliable environment simplifies the proof by allowing the programmer to consider fewer cases. To transfer the property into an unreliable environment with network and machine failures, a programmer uses Verdi's verified system transformers. As illustrated by Figure 1 part ④, after verifying a distributed system in one network semantics, a programmer can apply a verified system transformer to produce another version of their system which provides analogous guarantees in another network semantics.

In general, there are two types of transformers in Verdi: *transmission transformers* that handle network faults like packet duplication and drops and *replication transformers* that handle node crashes. Below we describe an example transmission transformer for the lock service application and briefly overview replication transformers, deferring details to Section 7.

Tolerating network faults. Figure 3's implementation of the lock service application will *not* function correctly in a network where messages can be duplicated. If an `UnlockMsg` message is duplicated but the agent reacquires the lock before the second copy is delivered, the server will misinterpret the duplicated `UnlockMsg` message as releasing the second lock acquisition.

Realistically, most developers would not run into this issue, as correct TCP implementations reject duplicate transmissions. However, some distributed systems need to handle deduplication and retransmission at a higher level, or choose not to trust the guarantees provided by unverified TCP implementations.

As another option, a programmer could rewrite the lock service—for instance, by including a unique identifier with every `GrantMsg` and `UnlockMsg` message to ensure that they are properly paired. The developer would then need to re-prove system correctness for this slightly different system in the semantics that models packet-duplicating networks. This would require finding a new inductive invariant and writing another proof.

Verdi allows developers to skip these steps. Verdi provides a system transformer that adds sequence numbers to every outgoing packet and ignores packets with sequence numbers that have already been seen. Applying this transformer to the lock service yields both a new system *and a proof* that the new system preserves the `mutex` property even when packets are duplicated by the underlying network. Section 4 further details this transformer.

More generally, Verdi decouples the verification of application-level guarantees from the implementation and verification of fault-tolerance mechanisms. Verdi provides a collection of verified system transformers which allow the developer to transfer guarantees about a system in one network semantics to analogous guarantees about a transformed version of the system in another network semantics. This allows a programmer to build and verify their system against an idealized semantics and use a verified system transformer to obtain a version of the system that provably tolerates more realistic faults while guaranteeing end-to-end system correctness properties.

Tolerating machine crashes. Verdi also provides verified system transformers to tolerate machine crashes via replication. Such replication transformers generally create multiple copies of a node in order tolerate machine crashes. This changes the number of nodes when transforming a system, which we discuss further in Section 7. (By contrast, transmission transformers like the one described above generally preserve the number of nodes and the relationships between them when transforming a distributed system.)

2.5 Running the Lock Service Application

Now we have a formally verified lock service, written in Coq, that tolerates message duplication faults. To obtain an executable for deployment, a Verdi programmer invokes Coq's built-in extraction mechanism to generate OCaml code from the Coq implementation, compile it with the OCaml compiler, and link it with a Verdi shim. The shim is written in OCaml; it implements network primitives (e.g., packet send/receive) and an event loop that invokes the appropriate event handler for incoming network packets, IO, or other events.

2.6 Summary

We have demonstrated how to use Verdi to establish a strong guarantee of the `mutex` property for the lock service application running in a realistic environment. Specifically, a programmer first specifies, implements, and verifies an application assuming a reliable environment. The programmer then applies system transformers to obtain a version of their application that handles faults in a provably correct way.

Verdi's trusted computing base includes the following components: the specifications of verified applications, the assumption that Verdi's network semantics match the physical network, the Verdi shim, Coq's proof checker and OCaml code extractor, and the OCaml compiler and runtime.

Verdi currently supports verifying safety properties, but not liveness properties, and none of Verdi's network semantics currently capture Byzantine fault models. We believe that Verdi could be extended to support these features: liveness properties could be verified by supporting infinite traces and adding fairness hypotheses as axioms as in TLA [19], while Byzantine fault models can be supported by adding more non-determinism in the network semantics.

3. Network Semantics

The correctness of a distributed system relies on assumptions about its environment. For example, one distributed system may assume a reliable network, while others may be designed to tolerate packet reordering, loss, or duplication. To enable programmers to reason about the correctness of distributed systems in the appropriate environment model, Verdi provides a spectrum of *network semantics* that encode possible system behaviors using small-step style derivation rules.

This section presents the spectrum of network semantics that Verdi provides, ranging from single-node systems that do not rely on the network, through models of increasingly unreliable packet delivery (reordering, drops, and duplication), and culminating with a model that permits arbitrary node crashes under various recovery assumptions. Each of these semantics is useful for reasoning about different types of systems. For example, the properties of single-node systems can be extended to handle node failures using protocols like Raft, while packet duplication semantics is useful for verifying packet delivery even in the face of reconnection, something that raw TCP does not support.

$$\frac{H_{\mathrm{inp}}(\sigma,\ i) = (\sigma',\ o)}{(\sigma,\ T) \rightsquigarrow_{\mathrm{s}} (\sigma',\ T \mathbin{++} \langle i,\ o\rangle)}\ \textsc{Input}$$

Figure 5. Single-node semantics. The derivation rule above encodes possible behaviors of a single-node system that does not rely on the network. When the node is in state σ with input/output trace T, it may receive an arbitrary input i, and respond by running its input handler $H_{\mathrm{inp}}(\sigma,\ i)$, which generates both the next state σ' and a list of outputs o. The Input rule relates the two states of the world $(\sigma,\ T) \rightsquigarrow_{\mathrm{s}} (\sigma',\ T \mathbin{++} \langle i,\ o\rangle)$ to reflect that the node has updated its state to σ' and sent outputs o in response to input i. Verifying properties of such single-node systems (i.e. state machines) is useful when they are replicated over a network to provide fault tolerance.

In Verdi, network semantics are defined as step relations on a "state of the world". The state of the world differs among network semantics, but always includes a trace of the system's external input and output. For example, many semantics include a bag of in-flight packets that have been sent by nodes in the system but have not yet been delivered to their destinations. Each network semantics is parameterized by system-specific data types and handler functions. Below we detail several of the network semantics Verdi currently provides.

Single-node semantics We begin with a simple semantics for single-node systems that do not use the network, i.e. state machines. This semantics is useful for proving properties of single-node systems; these can be extended, using a verified system transformer based on Raft, to provide fault tolerance. The single-node semantics, shown in Figure 5, models systems of a single node that respond to input by modifying their state and producing output. The node's behavior is described by a handler H, which takes the current local state and an input and returns the new state and a list of outputs. The state of the world in this semantics is the node's state σ paired with a trace T that records the inputs sent to the system along with the outputs the system generates. The only step, Input, delivers an arbitrary input i to the handler H and records the results in the next state. The squiggly arrow between two states indicates that a system in the state of the world on the left of the arrow may transition to the state of the world on the right of the arrow when all of the preconditions above the horizontal bar are satisfied. The node's state is updated, and the trace is extended with the input i and the output o.

Reordering semantics The reordering semantics, shown in Figure 6, models a system running on multiple nodes where packets are always delivered but may be arbitrarily reordered. This was the "reliable" semantics initially used for the lock service implementation in Section 2. Each node communicates with other processes running on the same host via input and output, just as in the single-node semantics. Nodes can also exchange *packets*, which are tuples of the form (source, destination, message), over a network that may reorder packets arbitrarily but does not drop, duplicate, or fabricate them. The behavior of nodes is described by two handler functions. The input handler, H_{inp}, is run whenever a node receives input from another process on the same host. H_{inp} takes as arguments the node on which it is running, the current local state, and the input that was delivered. It returns the new local state and a list of outputs and packets to be processed by the semantics. Similarly, the network handler, H_{net}, is run whenever a packet is delivered from the network. H_{net} takes as arguments the receiver of the packet, the sender of the packet, the local state, and the message that was delivered.

$$\frac{H_{\mathrm{inp}}(n,\ \Sigma[n],\ i) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[n \mapsto \sigma']}{(P,\ \Sigma,\ T) \rightsquigarrow_{\mathrm{r}} (P \uplus P',\ \Sigma',\ T \mathbin{+\!\!+} \langle i,\ o \rangle)}\ \textsc{Input}$$

$$\frac{\begin{array}{c} H_{\mathrm{net}}(dst,\ \Sigma[dst],\ src,\ m) = (\sigma',\ o,\ P') \\ \Sigma' = \Sigma[n \mapsto \sigma'] \end{array}}{(\{(src,\ dst,\ m)\} \uplus P,\ \Sigma,\ T) \rightsquigarrow_{\mathrm{r}} (P \uplus P',\ \Sigma',\ T \mathbin{+\!\!+} \langle o \rangle)}\ \textsc{Deliver}$$

Figure 6. Reordering semantics. The derivation rules above encode the behavior of systems running on networks that may arbitrarily reorder packet delivery. The network is modeled as a bag (i.e. a multiset) P of *packets*, which contain source and destination node names as well as a message. The state at each node in the network is a map Σ from node names to system-defined data. The INPUT rule passes arbitrary input i to the input handler H_{inp} for a given node n in state σ, which generates the next state σ', a list of outputs o, and a multiset of new packets P'. The outputs are added to the externally-visible trace, while the packets are added to the network (using the multiset-union operator $\uplus$). The DELIVER rule works similarly, except that instead of running a handler in response to arbitrary input, the network handle H_{net} is run on a packet taken from the network.

$$\frac{p \in P}{(P,\ \Sigma,\ T) \rightsquigarrow_{\mathrm{dup}} (P \uplus \{p\},\ \Sigma,\ T)}\ \textsc{Duplicate}$$

Figure 7. Duplicating semantics. The duplicating semantics includes all the derivation rules from the reordering semantics, which we elide for space. In addition, it includes the DUPLICATE rule, which duplicates an arbitrary packet in the network. This represents a simple class of network failure in which a network misbehaves by delivering the same packet multiple times.

A state of the world in the reordering semantics consists of a bag of in-flight packets P, a map from nodes to their local state Σ, and a trace T. The two rules in the reordering semantics, INPUT and DELIVER, respectively, model input from other processes on the node's host (i.e. the "outside world") and delivery of a packet from the network, where the corresponding handler function executes as described above. Delivered packets are removed from the bag of in-flight packets. Input and output are recorded in the trace; new packets are added to the bag of in-flight packets.

Duplicating semantics The duplicating semantics, shown in Figure 7, extends the reordering semantics to model packet duplication in the network. In addition to the INPUT and DELIVER rules from the reordering semantics, the duplicating semantics includes the rule DUPLICATE, which adds an additional copy of an in-flight packet to the network.

Dropping semantics Figure 8 specifies a network that drops arbitrary in-flight packets. The DROP rule allows any packet in the in-flight bag P to be dropped. However, simply adding this rule to the semantics would make it very difficult to write working systems, since handler functions only execute when packets are delivered and packets may be arbitrarily dropped. Real networked systems handle the possibility that packets can be dropped by setting timeouts, which execute if a certain amount of time has elapsed without receiving some other input or packet. We model this behavior in the TIMEOUT rule: a timeout can be delivered to any node at any time, and will execute the node's H_{tmt} handler.

$$\frac{}{(\{p\} \uplus P,\ \Sigma,\ T) \rightsquigarrow_{\mathrm{drop}} (P,\ \Sigma,\ T)}\ \textsc{Drop}$$

$$\frac{H_{\mathrm{tmt}}(n,\ \Sigma[n]) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[n \mapsto \sigma']}{(P,\ \Sigma,\ T) \rightsquigarrow_{\mathrm{drop}} (P \uplus P',\ \Sigma',\ T \mathbin{+\!\!+} \langle \mathrm{tmt},\ o \rangle)}\ \textsc{Timeout}$$

Figure 8. Dropping semantics. The dropping semantics includes the two rules above in addition to all the derivation rules from the duplicating semantics. The DROP rule allows a network to arbitrarily drop packets. Systems which tolerate dropped packets need to retransmit some messages, so the dropping semantics also includes a TIMEOUT rule, which fires a node's timeout handler H_{tmt}. The Verdi shim implements this by setting system-defined timeouts after every event; if another event has not occurred on a given node before the timeout fires, the system's H_{tmt} handler is executed. Note that the semantics do not explicitly model time and allow timeouts to occur at any step.

$$\frac{n \notin F}{(P,\ \Sigma,\ F,\ T) \rightsquigarrow_{\mathrm{fail}} (P,\ \Sigma,\ \{n\} \cup F,\ T)}\ \textsc{Crash}$$

$$\frac{n \in F \qquad H_{\mathrm{rbt}}(n,\ \Sigma[n]) = \sigma' \qquad \Sigma' = \Sigma[n \mapsto \sigma']}{(P,\ \Sigma,\ F,\ T) \rightsquigarrow_{\mathrm{fail}} (P,\ \Sigma',\ F - \{n\},\ T)}\ \textsc{Reboot}$$

Figure 9. Failure semantics. The node failure semantics represents a network in which nodes can both stop and start, and adds a set of failed nodes F to the state of the world. The node failure semantics includes all the derivation rules from the dropping semantics in addition to the rules above. The rules from the drop semantics are modified to only run when node n is not in the set of failed nodes F. The CRASH rule simply adds a node to the set of failed nodes F. Crashed nodes may re-enter the network via the REBOOT rule, at which point their state is restored according to the H_{rbt} function.

Node failure There are many possible models for node failure. Some systems assume that nodes will always return after a failure, in which case node failure is equivalent to a very long delay. Others assume that nodes will never return to the system once they have failed. Verdi's semantics for node failure, illustrated in Figure 9 assumes that nodes can return to the system and that all, some, or none of their state will be preserved (i.e. read back in from non-volatile storage). The state of the world in the node failure semantics includes a set F containing the nodes which have failed. The rules from the drop semantics are included in the failure semantics, but each with an added precondition to ensure that only live nodes (i.e. nodes that are not in F) can receive external input, network packets, or timeouts. A node can fail (be added to F) at any time, and failed nodes can return at any time. When a failed node returns, the H_{rbt} (reboot) function is run on its pre-failure state to determine what state survives the failure.

Low-level details Verdi's network semantics currently elide low-level network details. For example, input, output, and packets are modeled as abstract datatypes rather than bits exchanged over wires, and system details such as connection state are not modeled. This level of abstraction simplifies Verdi's semantics and eases both implementation and proof. Lower-level semantics could be developed and connected to the semantics presented here via system transformers, as described in the next section. This would further reduce Verdi's trusted computing base and increase our confidence in the end-to-end guarantees Verdi provides.

4. Verified System Transformers

Verdi's spectrum of network semantics enable the programmer to reason about their system running in the fault model corresponding to their environment. However, directly verifying a system in a realistic fault model requires establishing both application-level guarantees and the correctness of fault-tolerance mechanisms simultaneously. Verdi provides verified system transformers to separate these concerns and enable a modular approach to building and verifying distributed systems. The programmer can assume an idealized network while verifying application-level guarantees and then apply a transformer to obtain a system that tolerates more faults while providing analogous guarantees.

For common fault models, the distributed systems community has developed standard techniques to handle failures. For example, as discussed in Section 2, by adding a unique sequence number to every message and ignoring previously received messages, systems can handle packet duplication. Verdi supports such standard fault-tolerance mechanisms through verified system transformers, which *transform* systems from one semantics to another while guaranteeing that analogous system properties are preserved. For example, in the transformer that handles deduplication, any property that holds on the underlying system is true of the transformed system when sequence numbers are stripped away.

System transformers are implemented as wrappers around the system's state, messages, and handlers. Messages and state are generally transformed to include additional fields. Handlers in the transformed system call into underlying handlers and implement additional functionality. The underlying handlers are called with underlying state and underlying messages, capturing the intuition that the underlying handlers are unable to distinguish whether they are running in their original network semantics or the new semantics targeted by the system transformer.

System transformers in Verdi are generally either *transmission transformers*, which tolerate network faults by adding functionality to every node in a system, or *replication transformers*, which tolerate node failures by making several copies of the underlying nodes. The sequence numbering transformer discussed below is an example of a transmission transformer. Sections 6 and 7 discuss replication transformers.

4.1 Sequence Numbering Transformer

Sequence numbering is a technique for ensuring that messages are delivered at most once. Senders tag each outgoing message with a sequence number that is unique among all messages from that sender. Message recipients keep track of all ⟨`number`, `sender`⟩ pairs they have seen. If a message arrives with a ⟨`number`, `sender`⟩ pair that the destination has seen before, the message is discarded.

Figure 10 shows the Verdi implementation of the sequence numbering transformer, `SeqNum`. It takes a distributed system S as input and produces a new distributed system that implements sequence numbering by wrapping the message, state, and handler definitions in S. `SeqNum` leaves the `Name`, `Inp`, and `Out` types unchanged. It adds an integer field to each message which is used as a sequence number to uniquely identify messages. `SeqNum` also adds a list of (`Name`, `int`) pairs to the state to track the sequence numbers received from other nodes in the system, as well as an additional counter to track the local node's current maximum sequence number. The initial state in the wrapped system is constructed by building the initial state for the underlying system and then setting all sequence numbers to zero. To handle messages, the wrapped handler checks the input message to determine if it has previously been processed: if so, the message is simply dropped; otherwise, the message is passed to the message handler of S. Messages sent by the underlying handler are paired with fresh sequence numbers and the sequence number counter is incremented appropriately using the helper function `wrap_result`. The input handler passes input through to the input handler from S and wraps the results.

```
(* S describes a system in the reordering semantics *)
SeqNum (S) :=
  Name := S.Name

  Inp := S.Inp
  Out := S.Out
  Msg := { seqnum: int; underlying_msg: S.Msg }

  State (n: Name) := { seen: list (Name * int);
                       next_seqnum: int;
                       underlying_state: S.State n }

  InitState (n: Name) := { seen := [];
                           next_seqnum := 0;
                           underlying_state := S.InitState n }

  HandleInp (n: Name) (s: State n) (inp: Inp) :=
    wrap_result (S.HandleInp (underlying_state s) inp)

  HandleMsg (n: Name) (s: State n) (src: Name) (msg: Msg) :=
    if not (contains s.seen (src, msg.seqnum)) then
      s.seen := (src, msg.seqnum) :: s.seen;;
      (* wrap_result adds sequence numbers to messages while
         incrementing next_seqnum *)
      wrap_result (S.HandleMsg n (underlying_state s)
                               src (underlying_msg msg))
```

Figure 10. Pseudocode for the sequence numbering transformer.

4.2 Correctness of Sequence Numbering

Given a proof that property Φ holds on every trace of an underlying system, the correctness of a system transformer should enable a programmer to easily establish an analogous property Φ' of traces in the transformed system.

Each verified system transformer T provides a function `transfer` which translates properties of traces in the underlying semantics $\leadsto_1$ to the target semantics $\leadsto_2$:

$$\forall\ \Phi\ S, \texttt{holds}(\Phi,\ S,\ \leadsto_1) \rightarrow \texttt{holds}(\texttt{transfer}(\Phi),\ T(S),\ \leadsto_2)$$

where $\texttt{holds}(\Phi,\ S,\ \leadsto)$ asserts that a property Φ is true of all traces of a system S under the semantics defined by $\leadsto$. Crucially, the `transfer` function defines how properties of the underlying system are translated to analogous properties of the transformed system.

For the sequence numbering transformer, $\leadsto_1$ is $\leadsto_{\mathrm{r}}$ (the step relation for the reordering semantics) and $\leadsto_2$ is $\leadsto_{\mathrm{dup}}$ (the step relation for the duplicating semantics). The `transfer` function is the identity function: properties of externally visible traces are precisely preserved by the transformation. Intuitively, the external output depends only on the wrapped state of the system, and the wrapped state is preserved by the transformer.

We prove that the wrapped state is preserved by *backward simulation*: for any step the transformed system $T(S)$ can take, the underlying system S can take an equivalent step. We specify this using helper functions `unwrap` and $\texttt{dedup}_{\texttt{net}}$. Given the global state of the transformed system, `unwrap` returns the underlying state at each node. Given the global state of the transformed system and the bag of in-flight messages, $\texttt{dedup}_{\texttt{net}}$ returns a bag of packets which includes only those messages which will actually be delivered to the underlying handlers—non-duplicate packets which have not yet been delivered. The simulation is specified as follows, where $\leadsto_{\mathrm{dup}}^{\star}$ and $\leadsto_{\mathrm{r}}^{\star}$ are the reflexive transitive closures of the duplicating semantics and the reordering semantics, respectively:

$$(\Sigma_0,\ \emptyset,\ \emptyset) \rightsquigarrow^{\star}_{\text{dup}} (\Sigma,\ P,\ T) \rightarrow$$

$$(\texttt{unwrap}(\Sigma_0),\ \emptyset,\ \emptyset) \rightsquigarrow^{\star}_{\text{r}} (\texttt{unwrap}(\Sigma),\ \texttt{dedup}_{\texttt{net}}(\Sigma, P),\ T)$$

The proof is by induction on the step relation. For DUPLICATE steps, $\rightsquigarrow^{\star}_{\text{r}}$ holds reflexively, since $\texttt{dedup}_{\texttt{net}}$ returns the same network when a packet is duplicated and the state and trace are unchanged. For DELIVER steps, the proof shows that either the delivered packet is ignored by the destination node, in which case $\rightsquigarrow^{\star}_{\text{r}}$ holds reflexively, or that the underlying handler is run normally, in which case the underlying system can take the analogous DELIVER step. For both the DELIVER and INPUT steps, the proof shows that wrapping the sent packets results in a deduplicated network that is reachable in the underlying system. These proofs require several facts about the internal state of the sequence numbering transformer, such as the fact that all nodes correctly maintain their `next_seqnum` field. These internal state properties are proved by induction on the execution.

4.3 Ghost Variables and System Transformers

Many program verification frameworks support *ghost variables*: state which is never read during program execution, but which is necessary for verification (e.g. to provide sufficiently strong induction hypotheses). In Verdi, ghost variables are implemented via a system transformer. Like the sequence numbering transformer, the ghost variable transformer adds information to the system's state while ensuring that the wrapped state is preserved. The system's original handlers are called in order to update the wrapped state and send messages; the new handlers only update the ghost state. The indistinguishability result shows that the ghost transformer does not affect the externally-visible trace or the wrapped state. In this way, ghost state can be added to Verdi systems for free, without requiring any additional proof effort to show that properties verified in the ghost system hold for the underlying system as well.

5. Case Study: Key-Value Store

As a case study, we implemented a simple key-value store as a single-node system in Verdi. The key-value store accepts `get`, `put`, and `delete` operations as input. When the system receives input $\texttt{get}(k)$, it outputs the value associated with key k; when the system receives input $\texttt{put}(k,\ v)$, it updates its state to associate key k with value v; and when the system receives input $\texttt{delete}(k)$, it removes any associations for the key k from its state. Internally, the mapping from keys to values is represented using an association list.

The key-value store's correctness is specified in terms of traces. First, operations on a single key are specified using an interpreter over trace input/output events, which runs each operation and returns the final result. For instance,

$$\texttt{interpret}\ \left[\texttt{put}\ \text{“foo”},\ \texttt{put}\ \text{“bar”},\ \texttt{get}\right] = \text{“bar”}$$

Trace correctness is then defined using the interpreter: for every $\langle\texttt{input},\ \texttt{output}\rangle$ pair in the trace, `output` is equal to the value returned by running the interpreter on all operations on that key up to that point. This trace-based specification allows the programmer to change the backing data structure and implementation of each operation without changing the system's specification. Moreover, additional operations can be added to the specification via small modifications to the interpretation function.

We prove the key-value store's correctness by relating its trace to its current state: for all keys, the value in the association list for that key is equal to interpreting all the operations on that key in the trace. The proof is by induction on the execution, and is approximately 280 lines long.

In the next section, we will see how a state-machine replication system can be implemented and verified using Verdi. Combining the key-value store with the replication transformer provides an end-to-end guarantee for a replicated key-value store without requiring the programmer to simultaneously reason about both application correctness and fault tolerance.

```
PB (S) :=
  Name := Primary | Backup

  Msg := Replicate S.Inp | Ack
  Inp := S.Inp
  Out := { request: S.Inp; response: S.Out }
  State (n: Name) = { queue: list S.Inp;
                      underlying_state: S.State }

  InitState (n: Name) = { queue := [];
                          underlying_state := S.InitState n }

  HandleInp (n: Name) (s: State n) (inp: Inp) :=
    if n == Primary then
      append_to_queue inp;;
      if length s.queue == 1 then
        (* if not already replicating a request *)
        send (Backup, Replicate (head s.queue))

  HandleMsg (n: Name) (s: State n) (src: Name) (msg: Msg) :=
    match n, msg with
     | Primary, Ack =>
        out := apply_entry (head s.queue);;
        output { request := head s.queue; response := out };;
        pop s.queue;;
        if s.queue != [] then
          send (Backup, Replicate (head s.queue))
     | Backup, Replicate i =>
        apply_entry i;;
        send (Primary, Ack)
```

Figure 11. Pseudocode for the primary-backup transformer. The primary node accepts commands from external input and replicates them to the backup node. During execution, the primary node keeps a queue of operations it has received but not yet replicated to the backup node. The backup node applies operations to its local state and notifies the primary node. Once the primary node receives a notification, it responds to the client.

6. Case Study: Primary-Backup Transformer

In this section, we introduce the primary-backup replication transformer, which takes a single-node system and returns a replicated version of the system in the reordering semantics. A primary node synchronously replicates requests to a backup node: when a request arrives, the primary ensures that the backup has processed it before applying it locally and replying to the client. Whenever a client gets a response, the corresponding request has been processed by both the primary and the backup. Pseudocode for the primary-backup transformer is shown in Figure 11.

The primary-backup transformer's correctness is partially specified in terms of traces the primary may produce: any sequence of inputs and corresponding outputs produced by the primary node is a sequence that could have occurred in the original single-node system, and thus any property Φ of traces of the underlying single-node system also holds on all traces at the primary node in the transformed system. This result guarantees indistinguishability for the primary-backup transformer.

The primary-backup transformer specification also relates the backup node's state to the primary node's state. Because the primary replicates entries synchronously, and one at a time, the backup can fall arbitrarily behind the input stream at the primary. However, the primary does not send a response to the client until the backup has replicated the corresponding request. Thus, the state at the backup is closely tied to that at the primary. In particular, we were able to show that either the primary and the backup have the same state or the backup's state is one step ahead of the primary. This property

Table 1. Messages, inputs, and outputs used in Verdi's implementation of Raft.

	Name	Purpose
Messages	`AppendEntries` `AppendEntriesReply`	Log Replication
	`RequestVote` `RequestVoteReply`	Leader Election
Inputs	`ClientRequest`	Client inputs
Outputs	`ClientResponse` `NotLeader`	Successful execution Resubmit

provides some intuitive guarantees about potential failure of the primary: namely, that manual intervention could restore service with the guarantee that any lost request must not have been acknowledged. It makes sense that manual intervention is necessary in the case of failure: the composed system is verified against the reordering semantics, where the developer assumes that machine crashes require manual intervention.

Once implemented and verified, the primary-backup transformer can be used to construct replicated applications. Applying it to the case study from Section 5 results in a replicated key-value store. The resulting system is easy to reason about because of the transformer's indistinguishability result. For example, we were able to show (in about 10 lines) that submitting a `put` request results in a response that correctly reflects the `put`.

7. Case Study: Raft Replication Transformer

Fault-tolerant, consistent state machine replication is a classic problem in distributed systems. This problem has been solved with *distributed consensus* algorithms, which guarantee that all nodes in a system will agree on which commands the replicated state machine has executed and in what order, and that each node has a consistent copy of the state machine.

In Verdi, we can implement consistent state machine replication as a system transformer. The consistent replication transformer lifts a system designed for the state machine semantics into a system that tolerates machine crashes in the failure semantics. We implemented the replication transformer using the Raft consensus algorithm [30]. Our implementation of Raft in Verdi is described in Section 7.1.

A Verdi system transformer lifts a safety properties of an input system into a new semantics. The consensus transformer provides an indistinguishability result for *linearizability*, which states that any possible trace of the replicated system is equivalent to some valid trace of the underlying system under particular constraints about when operations can be re-ordered. We have proved that Raft's *state machine safety* property implies linearizability; our proof of state machine safety is still in progress as of this writing. We discuss this result further in Section 7.2.

7.1 Raft Implementation

Raft is structured as a set of remote procedure calls (RPCs). In Verdi, we implement each RPC as a pair of messages. Raft's `message` type is shown in Table 1. Raft divides time into *terms* of arbitrary length, and guarantees that there can be at most one leader per term. If a node n suspects that the leader has failed, that node advances its term and attempts to become the leader by sending `RequestVote` messages to every other node in the system. If a quorum of nodes votes for n, then n becomes the leader for its term. Since nodes can only vote for one leader in a given term, there is guaranteed to be at most one leader per term.

Once a leader has been elected, it can begin replicating log entries. A log entry stores a command (i.e. an input) for the underlying state machine, as well as the term in which it was created and a monotonically increasing `index`. Entries are created by the leader in response to `ClientRequest` inputs. When the leader creates an entry e, it sends `AppendEntries` messages to every other node in order to replicate e in other nodes' logs. Once e is in a quorum of logs, its command can safely be executed against the underlying state machine. More details about Raft can be found in the original Raft paper [30] and Ongaro's thesis [29].

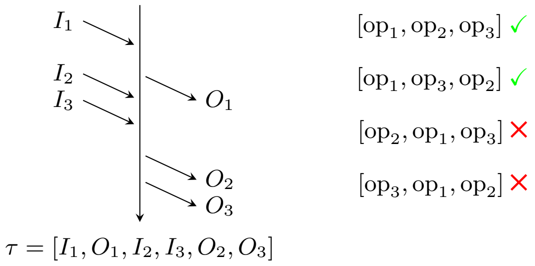
I_1
I_2
I_3
O_1
O_2
O_3
$\tau = [I_1, O_1, I_2, I_3, O_2, O_3]$
$[\mathrm{op}_1, \mathrm{op}_2, \mathrm{op}_3]$ ✓
$[\mathrm{op}_1, \mathrm{op}_3, \mathrm{op}_2]$ ✓
$[\mathrm{op}_2, \mathrm{op}_1, \mathrm{op}_3]$ ✗
$[\mathrm{op}_3, \mathrm{op}_1, \mathrm{op}_2]$ ✗

Figure 12. An example trace, with permitted and forbidden operation orderings. Since O_1 happens before I_2 and I_3, op_1 must happen before op_2 and op_3. The operations op_2 and op_3, however, can happen in either order.

The Verdi implementation of Raft includes the basic Raft algorithm, but does not include extensions of Raft which are described in the paper and useful in practice. In particular, it does not include log compaction, which allows a server to garbage-collect old log entries to save space, or membership changes, which allow nodes to be added and removed from a Raft cluster. We leave these features for future work.

7.2 Raft Proof

As discussed above, the indistinguishability result for Raft is linearizability. Linearizability [12] guarantees that clients see a consistent view of the state machine: clients see a consistent order in which operations were executed, and any request issued after a particular response is guaranteed to be ordered after that response.

We verified linearizability of the Verdi Raft implementation as a consequence of Raft's state machine safety property, which states that every node applies the same state machine command at a given index. We believe that this is the first formal proof (machine-checked or otherwise) of Raft's linearizability. A proof that state machine safety holds for our implementation is currently in progress [37]. A pencil and paper proof of state machine safety for a TLA model of Raft was given in Ongaro's thesis [29].

We formalized a general theory of linearizable systems in Coq as follows. A trace τ of requests $I_1, \ldots, I_n$ and responses $O_1, \ldots, O_m$ (where there is a total ordering on requests and responses) is linearizable with respect to an underlying state machine if there exists a trace of *operations* (i.e. request and response pairs) τ' such that: (1) τ' is a valid, sequential execution of the underlying state machine (meaning that each response is the one produced by running the state machine on the trace); (2) every response in τ has a corresponding operation in τ'; and (3) if a response to an operation op_1 occurs before a request for an operation op_2 in τ, then op_1 occurs before op_2 in τ'. Some examples of permitted and forbidden τ' for a particular τ are shown in Figure 12. Note that the primary-backup transformer described in Section 6 trivially provides linearizability: its traces are traces of the underlying system and it does no reordering.

Raft's I/O trace consists of `ClientRequests` and `ClientResponses`. The key to the proof is that Raft's internal log contains a linearized ordering of operations. The desired underlying trace, then, is just the list of operations in the order of the log. The rest of the proof involves showing that this order of operations satisfies the conditions

Table 2. Verification effort: size of the specification, implementation, and proof, in lines of code (including blank lines and comments).

System	Spec.	Impl.	Proof
Sequence numbering	20	89	576
Key-value store	41	138	337
Primary-backup	20	134	1155
KV+PB	5	N/A	19
Raft (Linearizability)	170	520	4144
Verdi	148	220	2364

above. To prove condition (1), we show that the state machine state is correctly managed by Raft and that entries are applied in the order they appear in the log. Condition (2) follows from the fact that Raft never issues a `ClientResponse` before the corresponding log entry is applied to the state machine. Finally, condition (3) holds because Raft only appends entries to the log: if a `ClientResponse` has already been issued, then that entry is already in the log, so any subsequent `ClientRequest` will be ordered after it in the log.

8. Evaluation

This section aims to answer the following questions:

- How much effort was involved in building the case studies discussed above?
- To what extent do system transformers mitigate proof burden when building modular verified distributed applications?
- Do Verdi applications correctly handle the faults they are designed to tolerate?
- Can a verified Verdi application achieve reasonable performance relative to analogous unverified applications?

8.1 Verification Effort

Table 2 shows the size of the specification, implementation, and proof of each case study. The Verdi row shows the number of lines in the shim, the network semantics from Section 3, and proofs of reusable, common lemmas in Verdi. The KV+PB row shows the *additional* lines of code required to state and prove a simple property of the key-value store with the primary-backup transformer applied. This line shows that verified system transformers mitigate proof burden by preserving properties of their input systems.

8.2 Verification Experience

While verifying the case studies, we discovered several serious errors in our system implementations. The most subtle of these errors came from our implementation of Raft: servers could delete committed entries when a complex sequence of failures occurred. Such a sequence is unlikely to arise in regular testing, but proving Raft in Verdi forced us to reason about all possible executions. The Raft linearizability property we proved prevents such subtle errors from going unnoticed.

8.3 Verification and Performance

We applied the consensus transformer described in Section 7 to the key-value store described in Section 5; we call the composed system `vard`.[1] We performed a simple evaluation of its performance. We ran our benchmarks on a three-node cluster, where each node had eight 2.0 GHz Xeon cores, 8 GB main memory, and 7200 RPM, 500 GB hard drives. All the nodes were connected to a gigabit switch and had ping times of approximately 0.1 ms. First, we ran the composed system and killed the leader node; the system came back as expected. Next, we measured the throughput and latency of the composed system and compared it to etcd [6], a production fault-tolerant key-value store written in the Go language which also uses Raft internally. We used a separate node to send 100 random requests using 8 threads; each request was either a put or a get on a key uniformly selected from a set of 50.

Table 3. A performance comparison of etcd and our `vard`.

	Throughput	**Latency**	
	(req./s)	`get` (ms)	`put` (ms)
`etcd`	38.9	205	198
`vard`	34.3	232	232

As shown in Table 3, `vard` achieves comparable performance to etcd. We believe that etcd has slightly better throughput and latency because of better data structures and because requests are batched. `vard` is not feature complete with respect to etcd, which uses different internal data structures and a more complex network protocol. Nonetheless, we believe this benchmark shows that a verified Verdi application can achieve roughly equivalent performance compared to existing, unverified alternatives.

9. Related Work

This section relates Verdi to previous approaches for building reliable distributed systems.

Proof assistants and distributed systems. EventML [32] provides expressive primitives and combinators for implementing distributed systems. EventML programs can be automatically abstracted into formulae in the Logic of Events, which can then be used to verify the system in NuPRL [5]. The ShadowDB project implements a total-order broadcast service using EventML [36]. The implementation is then translated into NuPRL and verified to correctly broadcast messages while preserving causality. A replicated database is implemented on top of this verified broadcast service. Unlike `vard` (described in Section 8), the database itself is unverified.

Bishop et al. [2] used HOL4 to develop a detailed model and specification for TCP and the POSIX sockets API, show that their model implements their specification, and validate their model against existing TCP implementations. Rather than verifying the network stack itself, in Verdi we chose to focus on verifying high-level application correctness properties against network semantics that are assumed to correctly represent the behavior of the network stack. These two lines of work are therefore complementary.

Ridge [34] verified a significant component of a distributed message queue, written in OCaml. His technique was to develop an operational semantics for OCaml which included some basic networking primitives, encode those semantics in the HOL4 theorem prover, and prove that the message queue works correctly under those semantics. Unlike in Verdi, the proofs for the system under failure conditions were done only on paper.

Verdi's system transformers enable decomposing both systems and proofs. This allows developers to establish end-to-end correctness guarantees of the implementation of their distributed systems, from the low-level network semantics to a high-level replicated key-value store, while retaining flexibility and modularity. The system transformer abstraction could integrated into these other approaches; for example, ShadowDB's consensus layer could be implemented as a system transformer along the lines of Verdi's Raft implementation.

Ensemble. Ensemble [11] layers simple *micro protocols* to produce sophisticated distributed systems. Like Ensemble micro protocols, Verdi's system transformers implement common patterns in distributed systems as modular, reusable components. Unlike

[1] Pronounced *var-DEE*.

Ensemble, Verdi's systems transformers come with correctness theorems that translate guarantees made against one network semantics to analogous guarantees against another semantics. Unlike Verdi, Ensemble enables systems built by stacking many layers of abstraction to achieve efficiency equivalent to hand-written implementations via cross-protocol optimizations. These micro protocols are manually translated to IO automata and verified in NuPRL [13, 23]. In contrast, Verdi provides a unified framework that connects the implementation and the formalization, eliminating the formality gap.

Verified SDN. Formal verification has previously been applied to software-defined networking, which allows routing configurations to be flexibly specified using a simple domain specific language (see, e.g. Verified NetCore [9]). As in Verdi, verifying SDN controllers involves giving a semantics for OpenFlow, switch hardware, and network communication. The style of formalization and proof in and Verdi are quite similar and address orthogonal problems. Verified NetCore is concerned with correct routing protocol configuration, while Verdi is concerned with the correctness of distributed algorithms that run on top of the network.

Specification Reasoning. There are many models for formalizing and specifying the correctness of distributed systems [7, 31, 35]. One of the most widely used models is TLA, which enables catching protocol bugs during the design phase [20]. For example, Amazon developers reported their experience of using TLA to catch specification bugs [27]. Another approach of finding specification bugs is to use a model checker. For example, Zave applied Alloy [15] to analyzing the protocol of the Chord distributed hash table [43]. Lynch [25] describes algorithm transformations which are similar to Verdi's verified system transformers.

On the other hand, Verdi focuses on ensuring that implementations are correct. While this includes the correctness of the underlying algorithm, it goes further by also showing that the actual running system satisfies the intended properties.

Model checking and testing. There is a rich literature in debugging distributed systems. Run-time checkers such as Friday [8] and D^{3}S [24] allow developers to specify invariants of a running system and detect possible violations on the fly or offline. Model checkers such as Mace [16, 17], MoDist [40], and CrystalBall [38] explore the space of executions to detect bugs in distributed systems. These tools are useful for catching bugs and easy to use for developers, as they only need to write invariants. On the other hand, Verdi's proof provide correctness guarantees.

For example, Mace provides a full suite of tools for building and model checking distributed systems. Mace's checker has been applied to discover several bugs, including *liveness* violations, in previously deployed systems. Mace provides mechanisms to explicitly break abstraction boundaries so that lower layers in a system can notify higher layers of failures. Verdi does not provide liveness guarantees nor mechanisms to break abstraction boundaries, but enables stronger guarantees via full formal verification.

Verification. Several major systems implementations have been verified fully formally in proof assistants. The CompCert C compiler [22] was verified in Coq and repeatedly shown to be more reliable than traditionally developed compilers [21, 41]. Our system transformers are directly inspired by the translation proofs in CompCert, but adapted to handle network semantics where faults may occur.

The Reflex framework [33] provides a domain-specific language for reasoning about the behavior of reactive systems. By carefully restricting the DSL, the authors were able to achieve high levels of proof automation. Bedrock [4] and Ynot [26] are verification frameworks based on separation logic and are useful for verifying imperative programs in Coq, but also consider only the behavior of a single node and do not model faults. These frameworks consider only the behavior of a single node and do not model faults.

10. Conclusion

This paper presented Verdi, a framework for building formally verified distributed systems. Verdi's key conceptual contribution is the use of verified system transformers to separate concerns of application correctness and fault tolerance, which simplifies the task of implementing and verifying distributed systems end-to-end. This modularity is enabled by Verdi's encoding of distinct fault models as separate network semantics. We demonstrated how to apply Verdi to writing and verifying several practical applications, including the Raft state replication library and the `vard` fault-tolerant key-value store, with the help of verified system transformers. These applications provide strong correctness guarantees and acceptable performance while imposing reasonable verification burden. We believe that Verdi provides a promising first step toward our overarching goal of easing the burden for programmers to implement correct, high-performance, fault-tolerant distributed systems.

Acknowledgments

The authors thank Steve Anton, Richard Eaton, Dan Grossman, Eric Mullen, Diego Ongaro, Dan R. K. Ports, Vincent Rahli, Daniel T. Ricketts, and Ryan Stutsman. We also thank Nate Foster for shepherding our paper, and the anonymous reviewers for their helpful and insightful feedback.

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] Amazon. Summary of the Amazon EC2 and Amazon RDS service disruption in the US East Region, Apr. 2011. `http://aws.amazon.com/message/65648/`.

[2] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 55–66, Charleston, SC, Jan. 2006.

[3] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, Portland, OR, Aug. 2007.

[4] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, San Jose, CA, June 2011.

[5] 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.

[6] CoreOS. etcd: A highly-available key value store for shared configuration and service discovery, 2014. `https://github.com/coreos/etcd`.

[7] S. J. Garland and N. Lynch. Using I/O automata for developing distributed systems. In *Foundations of Component-based Systems*. Cambridge University Press, 2000.

[8] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 285–298, Cambridge, MA, Apr. 2007.

[9] A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 483–494, Seattle, WA, June 2013.

[10] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Proceedings of the HotOS XIV*, Santa Ana Pueblo, NM, May 2013.

[11] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[12] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[13] J. J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 119–133, York, UK, Mar. 2009.

[14] High Scalability. The updated big list of articles on the Amazon outage, May 2011. `http://highscalability.com/blog/2011/5/2/the-updated-big-list-of-articles-on-the-amazon-outage.html`.

[15] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Feb. 2012.

[16] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 179–188, San Diego, CA, June 2007.

[17] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 243–256, Cambridge, MA, Apr. 2007.

[18] 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 *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, Oct. 2009.

[19] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, July 2002.

[20] L. Lamport. Thinking for programmers, Apr. 2014. `http://channel9.msdn.com/Events/Build/2014/3-642`.

[21] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 216–226, Edinburgh, UK, June 2014.

[22] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[23] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. L. Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 80–92, Kiawah Island, SC, Dec. 1999.

[24] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^3S: Debugging deployed distributed systems. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 423–437, San Francisco, CA, Apr. 2008.

[25] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1558603484.

[26] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Victoria, British Columbia, Canada, Sept. 2008.

[27] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. Use of formal methods at Amazon Web Services, Sept. 2014. `http://research.microsoft.com/en-us/um/people/lamport/tla/formal-methods-amazon.pdf`.

[28] NYTimes. Amazon's trouble raises cloud computing doubts, Apr. 2011. `http://www.nytimes.com/2011/04/23/technology/23cloud.html`.

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

[30] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014.

[31] J. L. Peterson. Petri nets. *ACM Computing Surveys*, pages 223–252, Sept. 1977.

[32] V. Rahli. Interfacing with proof assistants for domain specific programming using EventML. In *Proceedings of the 10th International Workshop On User Interfaces for Theorem Provers*, Bremen, Germany, July 2012.

[33] D. Ricketts, V. Robert, D. Jang, Z. Tatlock, and S. Lerner. Automating formal proofs for reactive systems. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 452–462, Edinburgh, UK, June 2014.

[34] T. Ridge. Verifying distributed systems: The operational approach. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, pages 429–440, Savannah, GA, Jan. 2009.

[35] D. Sangiorgi and D. Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0521781779.

[36] N. Schiper, V. Rahli, R. van Renesse, M. Bickford, and R. L. Constable. Developing correctly replicated databases using formal tools. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 395–406, Atlanta, GA, June 2014.

[37] Verdi. Verdi, 2014. `https://github.com/uwplse/verdi`.

[38] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 229–244, San Francisco, CA, Apr. 2008.

[39] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 99–110, Toronto, Canada, June 2010.

[40] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MoDist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 213–228, Boston, MA, Apr. 2009.

[41] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, San Jose, CA, June 2011.

[42] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 249–265, Broomfield, CO, Oct. 2014.

[43] P. Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, Apr. 2012.