Verdi: A Framework for Implementing and Formally Verifying Distributed Systems

Key-value store

VST

James R. Wilcox, Doug Woos, Pavel Panchekha, Zach Tatlock, Xi Wang, Michael D. Ernst, Thomas Anderson
Challenges
Challenges

Distributed systems run in unreliable environments
Challenges

Distributed systems run in unreliable environments

Many types of failure can occur
Challenges

Distributed systems run in unreliable environments

Many types of failure can occur

Fault-tolerance mechanisms are challenging to implement correctly
Challenges

Distributed systems run in unreliable environments

Many types of failure can occur

Fault-tolerance mechanisms are challenging to implement correctly
Challenges

Distributed systems run in unreliable environments

Many types of failure can occur

Fault-tolerance mechanisms are challenging to implement correctly

Contributions

Formalize network as operational semantics

Build semantics for a variety of fault models

Verify fault-tolerance as transformation between semantics
Verdi Workflow

Build, verify system in simple semantics
Verdi Workflow

Build, verify system in simple semantics

Apply verified system transformer
Verdi Workflow

Build, verify system in simple semantics

Apply verified system transformer

End-to-end correctness by composition
Contributions

Formalize network as operational semantics

Build semantics for a variety of fault models

Verify fault-tolerance as transformation between semantics
Contributions

Formalize network as operational semantics

Build semantics for a variety of fault models

Verify fault-tolerance as transformation between semantics

General Approach

Find environments in your problem domain

Formalize these environments as operational semantics

Verify layers as transformations between semantics
Verdi Successes

Applications

Key-value store
Lock service

Fault-tolerance mechanisms

Sequence numbering
Retransmission
Primary-backup replication
Consensus-based replication linearizability
Verdi Successes

Applications

- Key-value store
- Lock service

Fault-tolerance mechanisms

- Sequence numbering
- Retransmission
- Primary-backup replication
- Consensus-based replication linearizability
Important data
Replicated for availability
Replicated KV store

Replicated KV store

Replicated KV store

Replicated KV store
Crash
Reorder
Drop
Duplicate
Partition

Replicated KV store

Replicated KV store

Replicated KV store

Replicated KV store
Environment is unreliable
Decades of research; still difficult to implement correctly

Implementations often have bugs
Bug-free Implementations
Bug-free Implementations
Bug-free Implementations

Several inspiring successes in formal verification

\textit{CompCert, seL4, Jitk, Bedrock, IronClad, Frenetic, Quark}
Bug-free Implementations

Several inspiring successes in formal verification
*CompCert, seL4, Jitk, Bedrock, IronClad, Frenetic, Quark*

Goal: formally verify distributed system implementations
Formally Verify Distributed Implementations
Formally Verify Distributed Implementations

Separate independent system components
Formally Verify Distributed Implementations

Separate independent system components
Separate independent system components

Verify application logic independently from fault tolerance
Formally Verify Distributed Implementations

Separate independent system components

Verify **key-value store** independently from **consensus**
Formally Verify Distributed Implementations

1. Verify application logic
2. Verify fault tolerance mechanism
3. Run the system!

Separate independent system components

Verify key-value store independently from consensus
1. Verify Application Logic

- Client
- Key-value store

I/O
1. Verify Application Logic

Simple model, prove “good map”
I. Verify Application Logic

Simple model, prove “good map”
2. Verify Fault Tolerance Mechanism

Simple model, prove “good map”
2. Verify Fault Tolerance Mechanism

Simple model, prove “good map”
2. Verify Fault Tolerance Mechanism

Simple model, prove “good map”
2. Verify Fault Tolerance Mechanism

Simple model, prove “good map”

Apply verified system transformer, prove “properties preserved”
2. Verify Fault Tolerance Mechanism

Simple model, prove “good map”

Apply verified system transformer, prove “properties preserved”
2. Verify Fault Tolerance Mechanism

Simple model, prove “good map”

Apply verified system transformer, prove “properties preserved”

End-to-end correctness by composition
3. Run the System!

Extract to OCaml, link unverified shim
Run on real networks
Verifying application logic
Simple One-node Model

Key-value

State: {}
Simple One-node Model

Set "k" "v"

Key-value

State: {}
Simple One-node Model

Set “k” “v"

Key-value

State:
{"k": "v"}
Simple One-node Model

Set “k” “v” → Key-value
State: 
{“k”: “v”} → Resp “k” “v”
Simple One-node Model

Set “k” “v” → Key-value
State: {“k”: “v”} → Resp “k” “v”

Trace: [Set “k” “v”, Resp “k” “v”]
Simple One-node Model

System

State: $\sigma$
Simple One-node Model

Input: \( i \)

System

State: \( \sigma \)
Simple One-node Model

Input: $i$

System

State: $\sigma$

$H_{\text{inp}}(\sigma, i)$
\[ H_{\text{inp}}(\sigma, i) \]
Simple One-node Model

Input: $i$  \hspace{1cm}  System  \hspace{1cm}  Output: $o$

State: $\sigma$  

State: $\sigma'$

$H_{\text{inp}}(\sigma, i)$
Simple One-node Model

\[ H_{\text{inp}}(\sigma, i) = (\sigma', o) \]
Simple One-node Model

Input: $i$  
Output: $o$

System
State: $\sigma$
State: $\sigma'$

$H_{\text{inp}}(\sigma, i) = (\sigma', o)$
Simple One-node Model

Input: \( i \) \[\xrightarrow{\text{System}}\] Output: \( o \)

State: \( \sigma \) \[\xrightarrow{\text{State: } \sigma'}\]

Trace: \([i, o]\)

\[H_{\text{inp}}(\sigma, i) = (\sigma', o)\]

\[\left(\sigma, \quad \right) \xrightarrow{\sim} \left(\sigma', \quad \right)\]
Simple One-node Model

\[ H_{\text{inp}}(\sigma, i) = (\sigma', o) \]

\[ (\sigma, T) \xrightarrow{\sim} (\sigma', \text{Output: } o) \]
Simple One-node Model

Input: $i$ \quad \rightarrow \quad \text{System} \quad \rightarrow \quad \text{Output: } o

State: $\sigma$  
State: $\sigma'$

Trace: $[i, o]$

$H_{\text{inp}}(\sigma, i) = (\sigma', o)$

$(\sigma, T) \leadsto (\sigma', T ++ \langle i, o \rangle)$
Simple One-node Model

Client \[\xrightarrow{\text{I/O}}\] Key-value store
Simple One-node Model

Spec: operations have expected behavior (good map)
Set, Get
Del, Get
Simple One-node Model

Spec: operations have expected behavior (good map)
  Set, Get
  Del, Get

Verify system against semantics by induction
  Safety Property
Verifying Fault Tolerance
The Raft Transformer

Consensus provides a replicated state machine

Same inputs on each node

Calls into original system
The Raft Transformer

Consensus provides a replicated state machine

Same inputs on each node

Calls into original system

Log of operations
The Raft Transformer

Consensus support for replicated state

Same inputs on each node

Calls into original system

Log of operations

Original system

Raft

Raft

Raft
The Raft Transformer

When input received:

Add to log
Send to other nodes

When op replicated:

Apply to state machine
Send output
The Raft Transformer

For KV store:

Ops are Get, Set, Del

State is dictionary
Raft Correctness

Correctly transforms systems

Preserves traces

Linearizability
Fault Model

Model global state

Model internal communication

Model failure
Fault Model: Global State
Fault Model: Global State

Machines have names
Fault Model: Global State

Machines have names

Σ maps name to state
Fault Model: Messages

1
\[ \Sigma[1] \]

2
\[ \Sigma[2] \]

3
\[ \Sigma[3] \]
Fault Model: Messages

1
Σ[1]

2
Σ[2]

3
Σ[3]

Vote?

Vote?
Fault Model: Messages

Network

\langle 1, 2, \text{"Vote?"} \rangle
\langle 1, 3, \text{"Vote?"} \rangle
Fault Model: Messages

Network

<1,3,"Vote?">

<1,2,"Vote?">

1
Σ[1]

Vote?

2
Σ[2]

3
Σ[3]

Vote?
Fault Model: Messages

\[ H_{\text{net}}(dst, \Sigma[dst], src, m) \]
Fault Model: Messages

\[ H_{\text{net}}(dst, \Sigma[dst], src, m) \]
Fault Model: Messages

\[
H_{\text{net}}(\text{dst}, \Sigma[\text{dst}], \text{src}, m)
\]
Fault Model: Messages

$$H_{\text{net}}(dst, \Sigma[dst], src, m)$$
Fault Model: Messages

Network

\(<1, 3, "Vote?">\)
\(<2, 1, "+1"\)
Fault Model: Messages

\[ \text{Network} \]

\[ <1,3, "Vote?">, <2,1, "Vote?">, <1,2, "Vote?">, <1,3, "Vote?"> \]

Output: \( o \)

\[ H_{\text{net}}(\text{dst}, \Sigma[\text{dst}], \text{src}, m) = (\sigma', o, P') \]
Fault Model: Messages

Network

\[ \langle 1, 3, "Vote?" \rangle \]
\[ \langle 2, 1, "+1" \rangle \]

\[ \langle 1, 2, "Vote?" \rangle \]

\[ \sum[2] = \sigma' \]

Output: \( o \)

\[ H_{\text{net}}(\text{dst}, \Sigma[\text{dst}], \text{src}, m) = (\sigma', o, P') \]
\[ \Sigma' = \Sigma[\text{dst} \mapsto \sigma'] \]
Fault Model: Messages

Network

\[ <1,3,"Vote?"> \]
\[ <2,1,"+1"> \]

\[ <1,2,"Vote?"> \]

\[ \begin{align*}
\Sigma[1] & \rightarrow \Sigma[2], \\
\Sigma[2] & \rightarrow \Sigma'[2] = \sigma', \\
\Sigma'[2] & \rightarrow \Sigma[3]
\end{align*} \]

Output: \( \sigma \)

\[ H_{\text{net}}(dst, \Sigma[dst], src, m) = (\sigma', o, P') \]
\[ \Sigma' = \Sigma[dst \mapsto \sigma'] \]
\[ ((src, dst, m)} \cup P, \Sigma, T) \]
Fault Model: Messages

Network

\[ \langle 1,3, \text{"Vote?"} \rangle \]
\[ \langle 2,1, \text{"+1"} \rangle \]

\[ \langle 1,2, \text{"Vote?"} \rangle \]

\[ \Sigma[1] \]

\[ \Sigma[2] \]
\[ \Sigma'[2] = \sigma' \]

Output: 0

\[ H_{\text{net}}(\text{dst}, \Sigma[\text{dst}], \text{src}, m) = (\sigma', o, P') \]
\[ \Sigma' = \Sigma[\text{dst} \mapsto \sigma'] \]

\[ \{(\text{src, dst, m})\} \cup P, \Sigma, T \sim (P \cup P', \Sigma', T ++ \langle o \rangle) \]
Fault Model: Failures

Network

\[ <1,2,"\text{Vote?"} > \]
\[ <1,3,"\text{Vote?"} > \]

\[ \Sigma[1] \]

\[ \Sigma[2] \leftrightarrow \Sigma[3] \]
Fault Model: Failures

Message drop

Network

\(<1, 3, "Vote?"\>
Fault Model: Failures

Message drop

Message duplication

Network

\[ <1,3,\text{"Vote?"}> \]

\[ \Sigma[1] \]

\[ \Sigma[2] \]

\[ \Sigma[3] \]
Fault Model: Failures

Message drop

Message duplication

Machine crash

Network

<1,3,"Vote?"> <1,3,"Vote?">
Fault Model: Drop

$\langle 1,2,\text{"hi"} \rangle$

$\langle 1,3,\text{"hi"} \rangle$

\[
(\{p\} \oplus P, \Sigma, T) \rightsquigarrow (P, \Sigma, T)
\]
Fault Model: Drop

Network

\((\{p\} \uplus P, \Sigma, T) \leadsto (P, \Sigma, T)\)
Fault Model: Drop

\[
\begin{align*}
\text{Network: } & \quad \langle 1,2,"hi" \rangle \\
& \quad \langle 1,3,"hi" \rangle \\
\hline
\{p\} \uplus P, \Sigma, T & \sim \Rightarrow (P, \Sigma, T)
\end{align*}
\]
Toward Verifying Raft

General theory of linearizability

1k lines of implementation, 5k lines for linearizability

State machine safety: 30k lines

Most state invariants proved, some left to do
Verified System Transformers

Functions on systems

Transform systems between semantics

Maintain equivalent traces

Get correctness of transformed system for free
Verified System Transformers
Verified System Transformers

Raft Consensus

App

Consensus

App

Consensus

App
Verified System Transformers

Raft Consensus

Primary Backup

App

Consensus

App

Consensus

App

Primary

App

Backup
Verified System Transformers

Raft Consensus

Primary Backup

Seq # and Retrans
Verified System Transformers

Raft Consensus

Primary Backup

Seq # and Retrans

Ghost Variables

App

Consensus

Primary

Backup

Seq + Retrans

Ghost
Running Verdi Programs
Running Verdi Programs

Coq extraction to Ocaml

Thin, unverified shim

Trusted compute base: shim, Coq, Ocaml, OS
Performance Evaluation

Compare with etcd, a similar open-source store

10% performance overhead

Mostly disk/network bound

etcd has had linearizability bugs
Previous Approaches

EventML [Schiper 2014]

Verified Paxos using the NuPRL proof assistant

MACE [Killian 2007]

Model checking distributed systems in C++

TLA+ [Lamport 2002]

Specification language and logic
Contributions

Formalize network as operational semantics

Build semantics for a variety of fault models

Verify fault-tolerance as transformation between semantics
Contributions

Formalize network as operational semantics

Build semantics for a variety of fault models

Verify fault-tolerance as transformation between semantics

http://verdi.uwplse.org
Contributions

Formalize network as operational semantics

Build semantics for a variety of fault models

Verify fault-tolerance as transformation between semantics

http://verdi.uwplse.org

Thanks!