Verifying Distributed Systems

Zachary Tatlock

Coq Workshop 2018
The Team

James Wilcox
Doug Woos
Pavel Panchekha
Ryan Doenges

Justin Adsuara
Keith Simmons
Steve Anton
Miranda Edwards

Karl Palmskog
Ilya Sergey
Xi Wang
Mike Ernst
Tom Anderson
The Team

James Wilcox
Doug Woos
Pavel Panchekha
Ryan Doenges

Justin Adsuara
Keith Simmons
Steve Anton
Miranda Edwards

Karl Palmskog
Illya Sergey
Xi Wang
Mike Ernst
Tom Anderson
The Team

Amazing researchers on the job market next year!
Distributed Systems
Distributed Systems
Distributed Apps
Distributed Infrastructure
One summer day...
One summer day...

**The New York Times**

The Stock Market Bell Rings, Computers Fail, Wall Street Cringes

By NATHANIEL POPPER  JULY 8, 2015

Problems with technology have at times roiled global financial markets, but the 223-year-old [New York Stock Exchange](https://www.nyse.com) has held itself up as an oasis of humans ready to step in when the computers go haywire.

On Wednesday, however, those working on the trading floor were left helpless when the computer systems at the exchange went down for nearly four hours in the middle of the day, bringing an icon of capitalism’s ceaseless energy to a costly halt.

The exchange ultimately returned to action shortly before the closing bell,
One summer day...
One summer day...
How distributed systems fail
How distributed systems fail

Challenges
concurrency
How distributed systems fail

Challenges
- concurrency
- message drops
- message dups
- message reorder
- machine crash
- machine reboot
How distributed systems fail
How distributed systems fail

Too many possible behaviors to effectively test!
How distributed systems fail

When exhaustive testing is impossible, our trust can only be based on proof.

Edsger W. Dijkstra
Under the Spell of Leibniz's Dream
Toward verified distributed systems
Toward verified distributed systems

Formalize *network semantics*

capture how faults can occur
Toward verified distributed systems

Formalize *network semantics*

*capture how faults can occur*

Separate app / fault reasoning
Toward verified distributed systems

Formalize network semantics
capture how faults can occur

Separate app / fault reasoning
develop and prove in simple fault model
Toward verified distributed systems

Formalize network semantics

capture how faults can occur

Separate app / fault reasoning

develop and prove in simple fault model
Toward verified distributed systems

Formalize network semantics

capture how faults can occur

Separate app / fault reasoning

develop and prove in simple fault model

apply generic verified fault handling
Toward verified distributed systems

Formalize *network semantics*

*capture how faults can occur*

Separate app / fault reasoning

*develop and prove in simple fault model*

*apply generic verified fault handling*
Toward verified distributed systems

The Verdi Framework

Verified Raft Consensus

TCB, Tools, Teaching

Enriching Models & Modularity
Toward verified distributed systems

The Verdi Framework

Verified Raft Consensus

TCB, Tools, Teaching

Enriching Models & Modularity
Formalizing distributed systems
Formalizing distributed systems

Diagram:

- S1
- S2
- S3
- S4
- S5

- Numbers 1, 2, 3, 4, 5, 6, 7, 8, 9

- Columns labeled S1, S2, S3, S4, S5
Formalizing distributed systems

timeouts
Formalizing distributed systems
Formalizing distributed systems

timeouts
msg delivery
state change
Formalizing distributed systems
Formalizing distributed systems
Formalizing distributed systems

1. Defining distributed systems
2. Giving systems semantics
3. Proving system safety
4. Reusable, verified fault-tolerance
1. Distributed sys as event handlers

Def mySys (P : params) : system :=

...
1. Distributed sys as event handlers

Def mySys (P : params) : system :=

   // types for state and I/O
Type msg := (* to/from internal nodes *)
Type cmsg := (* to/from external world *)
Type data := (* node-local state *)

...
1. Distributed sys as event handlers

Def mySys (P : params) : system :=

// types for state and I/O
Type msg := (* to/from internal nodes *)
Type cmsg := (* to/from external world *)
Type data := (* node-local state *)

Type resp := data * list cmsg * list msg

// event handlers
Def onMsg : data * msg -> resp
Def onTmOut : data * unit -> resp
Def onClient : data * cmsg -> resp
2. Network semantics
2. Network semantics

\[(P, \Sigma, T)\] state of the world
2. Network semantics

\[(P, \Sigma, T)\]

- state of the world
- packets in flight
2. Network semantics

\((P, \Sigma, T)\)

- state of the world
- packets in flight
- data @ nodes
2. Network semantics

\[(P, \Sigma, T)\]

- state of the world
- packets in flight
- history of client I/O
- data @ nodes
2. Network semantics

\[(P, \Sigma, T)\]
2. Network semantics

\[(P, \Sigma, T) \rightsquigarrow (P', \Sigma', T')\]

Good old small step operational semantics.
Example rule: message delivery

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

\[
((\text{src}, \text{dst}, m)) \uplus P, \Sigma, T) \leadsto (P \uplus P', \Sigma', T++\langle o \rangle)
\]
Example rule: message delivery

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

\[
(\{(\text{src}, \text{dst}, m)\} \cup P, \Sigma, T) \leadsto (P \cup P', \Sigma', T + + \langle o \rangle)
\]

if this message is in the network
Example rule: message delivery

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

\[ \{(\text{src}, \text{dst}, m)\} \uplus P, \Sigma, T \leadsto (P \uplus P', \Sigma', T ++ \langle o \rangle) \]

run handler on message

if this message is in the network
Example rule: message delivery

\[
H_{\text{net}}(\text{dst}, \Sigma[\text{dst}], \text{src}, m) = (\sigma', o, P') \\
(\{(\text{src}, \text{dst}, m)\} \uplus P, \Sigma, T) \rightsquigarrow (P \uplus P', \Sigma', T ++ \langle o \rangle)
\]

- run handler on message
- get response
- if this message is in the network
Example rule: message delivery

If this message is in the network, get response resulting new global state.

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

$$(\{(\text{src}, \text{dst}, m)\} \cup P, \Sigma, T) \leadsto (P \cup P', \Sigma', T ++ \langle o \rangle)$$

- run handler on message
- get response
2. Network semantics

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

Deliver

\[
((\text{src}, \text{dst}, m) \cup P, \Sigma, T) \leadsto (P \cup P', \Sigma', T ++ \langle o \rangle)
\]

Deliver

\[
p \in P
\]

Duplicate

\[
(P, \Sigma, T) \leadsto (P \cup \{p\}, \Sigma, T)
\]

Duplicate

\[
(\{p\} \cup P, \Sigma, T) \leadsto (P, \Sigma, T)
\]

Drop

\[
H_{\text{tmt}}(n, \Sigma[n]) = (\sigma', o, P') \quad \Sigma' = \Sigma[n \mapsto \sigma']
\]

Timeout

\[
(P, \Sigma, T) \leadsto (P \cup P', \Sigma', T ++ \langle \text{tmt}, o \rangle)
\]
Library of network semantics

Type sem := state -> state -> Prop

Def sync_sem := (* in-order delivery *)
Def async_sem := (* + reordering *)
Def flaky_sem := (* + drops, timeouts *)
Def busy_sem := (* + duplicates *)
Def crash_sem := (* + crash, reboot *)
Library of network semantics

Type sem := state -> state -> Prop

Def sync_sem := (* in-order delivery *)

Def async_sem := (* + reordering *)

Def flaky_sem := (* + drops, timeouts *)

Def busy_sem := (* + duplicates *)

Def crash_sem := (* + crash, reboot *)

Precisely characterize fault model for sys.
Library of network semantics

Type \( \text{sem} := \text{state} \rightarrow \text{state} \rightarrow \text{Prop} \)

Def sync\_sem := (* in-order delivery *)

Def async\_sem := (* + reordering *)

Def flaky\_sem := (* + drops, timeouts *)

Def busy\_sem := (* + duplicates *)

Def crash\_sem := (* + crash, reboot *)

Precisely characterize fault model for sys.

more behaviors \( \rightarrow \) harder proof
3. Verifying system safety

Definition: $\text{ok} : \text{state} \rightarrow \text{Prop}$
3. Verifying system safety

\[ \text{Def ok : state} \rightarrow \text{Prop} \]

\[(P, \Sigma, T)\]
3. Verifying system safety

Definition: \( \text{ok} : \text{state} \rightarrow \text{Prop} \)}

Initial state: \((P, \Sigma, T)\)
3. Verifying system safety

Def ok : state -> Prop

init state

\((P, \Sigma, T)\)
3. Verifying system safety

**Def ok : state → Prop**

need to show all reachable states ok
3. Verifying system safety

Def ok : state -> Prop
3. Verifying system safety

Def ok : state -> Prop

As usual, problem is specs not inductive.
3. Verifying system safety

Def ok : state -> Prop

As usual, problem is specs not inductive.

Strengthen “ok” to inductive “ok_ind”.
3. Verifying system safety

When verifying systems in a particular semantics, need to repeat similar fault tolerance reasoning for every system.
4. Verifying system *transformers*

Implement fault tolerance as wrapper

\[
\text{Def tcp : system} \rightarrow \text{system}
\]

Transfer proofs across semantics

\[
\text{Theorem tcp_ok : forall s P, P s \rightarrow lift_tcp P (tcp s)}
\]

Separate app proof / fault tolerance

*handles class of faults once and for all* can compose transformers, proofs
4. Verifying system *transformers*

**Raft Consensus**
- App
  - Consensus
  - Primary
  - Backup

**Primary Backup**
- App

**Seq # and Retrans**
- App
  - Seq + Retrans
  - Retrans

**Ghost Variables**
- App
  - Ghost
Toward verified distributed systems

The Verdi Framework

Verified Raft Consensus

TCB, Tools, Teaching

Enriching Models & Modularity
Example: verifying Raft in Verdi
Example: verifying Raft in Verdi

critical components must not fail
Replication for fault tolerance
Replication for fault tolerance

available if n/2 nodes are up
Replication for fault tolerance
Replication correctness
Replication correctness
Replication correctness

linearizability

cluster looks like a single node (state machine) to clients
Defining Raft

\[
\text{Def raft(sm: state machine, \ldots) :=}
\]

\[
\ldots
\]
Defining Raft

Def raft(sm: state machine, ...) :=

// types for state and I/O
cmsg := sm.cmsg

...
Defining Raft

Def raft(sm: state machine, ...) :=

// types for state and I/O

cmsg := sm.cmsg
msg := (* ??? *)
data := (* ??? *)

// event handlers
Def onMsg := (* ??? *)
Def onTmOut := (* ??? *)
Def onClient := (* ??? *)
Raft: election and replication terms
Raft: election and replication terms
Raft: leader election
Raft: leader election

Candidate
Raft: leader election

Candidate

Followers

- ReqVote

Term 1
Term 2
Term 3
Raft: leader election

Candidate → Followers

- ReqVote
- Vote

Term 1
Term 2
Term 3
Raft: election and replication terms
Raft: log replication

- Leader
- Followers
  - Append
  - AppendAck
Defining Raft

\[
\text{Def \ raft(sm: state machine, ...) :=} \\
\\
// types for state and I/O \\
cmsg := sm.cmsg \\
msg := \text{ReqVote} | \text{Vote} | \text{Append} | \ldots \\
data := \{ \text{sm.data, list sm.op, \ldots} \}
\]

// event handlers \\
Def onMsg := \\
Def onTmOut := \\
Def onClient :=
Verifying Raft
Verifying Raft

linearizability
Raft internal correctness
Raft internal correctness

linearizability follows from internal correctness: state machine safety
Proving Raft in Verdi
Proving Raft in Verdi

Graphical representation showing the reduction of complexity from large to smaller graphs.
State machine safety

Nodes’ logs match on committed entries

since only committed entries executed

proof by induction over executions
State Machine Safety: Proof
State Machine Safety: Proof

not inductive!
State Machine Safety: Proof
State Machine Safety: Proof

I \Rightarrow \text{Diagram} \Rightarrow I
State Machine Safety: Proof
State Machine Safety: Proof

$I$ true initially

$I$ preserved

$I \Rightarrow \text{network image} \Rightarrow I$
State Machine Safety: Proof

90 invariants in total

Lemma Lemma Lemma ...

\[ \text{true initially} \quad \Rightarrow \quad \text{preserved} \]

\[ \text{I} \Rightarrow \text{I} \]

\[ \text{I} \Rightarrow \text{I} \]
State Machine Safety: Proof

\[ I \text{ true initially} \quad \rightarrow \quad I \text{ preserved} \]

\[ I \Rightarrow \text{Lemma Lemma Lemma \ldots} \]
State Machine Safety: Proof

I

true initially

I

preserved

I

Lemma
Lemma
Lemma

I

I

I

I
State Machine Safety: Proof

I  true initially  I  preserved

I  \[\Rightarrow\]  green square  I

Lemma  Lemma  Lemma  \ldots
The burden of proof

Re-verification is the primary challenge:
- invariants are not inductive
- not-yet-verified code is wrong
- need additional invariants
The burden of proof

Re-verification is the primary challenge
The burden of proof

Re-verification is the primary challenge

Proof engineering techniques help:
- affinity lemmas
- intermediate reachability
- structural tactics
- information hiding
Ghost state: example

Capture all entries received by a node
Ghost state: example

Capture all entries received by a node
Ghost state: example

Capture all entries received by a node

Log (real)

Leader  A,B,C
Ghost state: example

Capture all entries received by a node

Log (real)

Follower

Leader
Ghost state: example

Capture all entries received by a node

- Follower
  - Log (real): A, D
  - allEntries (ghost): \{A, D\}

- Leader
  - Log (real): A, B, C
  - allEntries (ghost): \{A, B, C\}
Ghost state: example

Capture all entries received by a node

Leader

Follower

Append [A],B,C

Log (real) A,B,C

allEntries (ghost) \{A,B,C\}

A,D

\{A,D\}
Ghost state: example

Capture all entries received by a node

Leader

Follower

Log (real)  allEntries (ghost)

A,B,C  {A,B,C,D}

[A],B,C  {A,B,C}
Affinity lemmas: example

\[ e \in \text{allEntries} \implies e.\text{term} > 0 \]
Affinity lemmas: example

\[ e \in \log \Rightarrow \]
\[ e.\text{term} > 0 \]

\[ e \in \text{allEntries} \Rightarrow e.\text{term} > 0 \]
Affinity lemmas: example

e ∈ log ⇒

e.term > 0

\textbf{Affinity Lemma}

e ∈ allEntries ⇒ e.term > 0
Affinity lemmas: example

\[ e \in \log \Rightarrow e.\text{term} > 0 \]

Every invariant of entries in logs is invariant of entries in `allEntries`

\[ e \in \text{allEntries} \Rightarrow e.\text{term} > 0 \]
Affinity lemmas: example

\[ e \in \log \Rightarrow P(e) \]

\[ e \in \text{allEntries} \Rightarrow P(e) \]

\[
\text{every invariant of entries in logs is invariant of entries in allEntries}
\]
More affinity lemmas

Relate ghost state to real state

*transfer properties once and for all*

Relate current messages to past

*response => past request*
Structured handlers

handler = update_state ; respond
Structured handlers

\[
\text{handler} = \text{update\_state} ; \text{respond}
\]

\[
\text{net}
\]

\[
\text{net'}
\]
Structured handlers

\[
\text{handler} = \text{update\_state} \ ; \ \text{respond}
\]
Structured handlers

handler = update_state ; respond

handler

net

I

net

I

net’

update_state

net

net_i

respond

net’
Structured handlers

\[
\text{handler} = \text{update\_state} ; \text{respond}
\]
First formal verification of Raft

50k lines of Coq
18 person-months
Considerable pizza and beer
First formal verification of Raft

50k lines of Coq
18 person-months
Considerable pizza and beer

Diego Ongaro
@ongardie

@wilcoxjay so that's it then. You win.
First formal verification of Raft

50k lines of Coq
18 person-months
Considerable pizza and beer
Toward verified distributed systems

The Verdi Framework

Verified Raft Consensus

TCB, Tools, Teaching

Enriching Models & Modularity
Verified $\neq$ perfect

Network semantics shim is delicate

  atomicity, fairness, serialization,…

Verdi users need Coq + distr sys skills

  notorious learning curves hinder impact

Regular development still tricky

  maintenance, extension, management
Network semantics shim is delicate

\[ H_{\text{net}}(\text{dst}, \Sigma[\text{dst}], \text{src}, m) = (\sigma', o, P') \quad \Sigma' = \Sigma[\text{dst} \mapsto \sigma'] \]
\[ (\{(\text{src}, \text{dst}, m)\} \cup P, \Sigma, T) \leadsto (P \cup P', \Sigma', T ++ \langle o \rangle) \] \hspace{1cm} \text{DELIVER}

\[ p \in P \]
\[ (P, \Sigma, T) \leadsto (P \cup \{p\}, \Sigma, T) \] \hspace{1cm} \text{DUPLICATE}

\[ (\{p\} \cup P, \Sigma, T) \leadsto (P, \Sigma, T) \] \hspace{1cm} \text{DROP}

\[ H_{\text{tmt}}(n, \Sigma[n]) = (\sigma', o, P') \quad \Sigma' = \Sigma[n \mapsto \sigma'] \]
\[ (P, \Sigma, T) \leadsto (P \cup P', \Sigma', T ++ \langle \text{tmt}, o \rangle) \] \hspace{1cm} \text{TIMEOUT}
Network semantics shim is delicate

\[
H_{\text{net}}(dst, \Sigma[dst], src, m) = (\sigma', o, P') \quad \Sigma' = \Sigma[dst \mapsto \sigma']
\]
\[
\{(src, dst, m)\} \cup P, \Sigma, T \leadsto (P \cup P', \Sigma', T ++ \langle o \rangle)
\]

\[
p \in P
\]
\[
(P, \Sigma, T) \leadsto (P \cup \{p\}, \Sigma, T)
\]

\[
\{(p) \cup P, \Sigma, T\} \leadsto (P, \Sigma, T)
\]

\[
H_{\text{tmt}}(n, \Sigma[n]) = (\sigma', o, P') \quad \Sigma' = \Sigma[n \mapsto \sigma']
\]
\[
(P, \Sigma, T) \leadsto (P \cup P', \Sigma', T ++ \langle \text{tmt}, o \rangle)
\]

Note that all steps are atomic in semantics!
Shim must carefully persist to ensure fidelity.
Network semantics shim is delicate

Node in singleton cluster never becomes leader #40

tschottendorf commented 24 days ago

I'm trying to run the benchmarks against a single-node system:

```bash
$ ./vard.native disp /tmp/vard.sock -port 9000.0 -e node 0,127.0.0.1:10000 -debug
```

unordered shim running setup for vard

unordered shim ready for action

client 11551202 connected on 127.0.0.1:49446

client 11551202 disconnected: client closed socket

The client logged above is the following invocation:

```python
python2 bench/setup.py --service vard --keys 50 --cluster 127.0.0.1:10000
```

Traceback (most recent call last):

File "bench/setup.py", line 34, in <module>
main()

File "bench/setup.py", line 27, in main
   host, port = client.find_leader(args.cluster)
   File "/users/tschottendorf/tin/verdi-raft/extraction/vard-src/vard/NoLeader
```

paismog commented 23 days ago

I'm pretty sure this is a liveness bug (and thus an issue outside the scope of election safety, which is guaranteed). What happens is that the singleton node never manages to elect itself leader - it waits forever for a `requestVoteReply` message.

The `tryToBecomeLeader` function in raft/Raft.v is called when a timeout occurs. However, `tryToBecomeLeader` does not immediately check whether the candidate wins the vote. This is only done once a `RequestVoteReply` message is received, using a `call` to `voteElectoral`.

The original Go implementation of Raft uses a general loop for the Candidate state that first sends all necessary `RequestVote` messages and then immediately checks whether it has enough votes (and becomes leader if possible). The bug could be fixed by adding a similar check to `tryToBecomeLeader`, but I'm not sure how much that would mess with the proofs. Arguably, there is no point in running Raft in a singleton node cluster anyway - it's enough to run a system that directly uses the underlying state machine (vard).
Network semantics shim is delicate

User stumbled across liveness bug for single node cluster.
An Empirical Study on the Correctness of Formally Verified Distributed Systems

Pedro Fonseca    Kaiyuan Zhang    Xi Wang    Arvind Krishnamurthy
University of Washington
{plfonseca, kaiyuan, xi, arvind}@cs.washington.edu

Abstract
Recent advances in formal verification techniques enabled the implementation of distributed systems with machine-checked proofs. While results are encouraging, the importance of distributed systems warrants a large scale evaluation of the results and verification practices.

This paper thoroughly analyzes three state-of-the-art, formally verified implementations of distributed systems: Iron-Fleet, Verdi, and Chapur. Through code review and testing, we found a total of 16 bugs, many of which produce serious consequences, including crashing servers, returning incorrect results to clients, and invalidating verification guarantees. These bugs were caused by violations of a wide-range of assumptions on which the verified components relied. Our results revealed that these assumptions referred to a small fraction of the trusted computing base, mostly at the inter-

![Diagram](image)

Figure 1: An overview of the workflow to verify a distributed system implementation.

finding tools [26, 37, 53, 54], and formal verification techniques [22, 29, 34, 52].
An Empirical Study on the Correctness of Formally Verified Distributed Systems

Pedro Fonseca  Kaiyuan Zhang  Xi Wang  Arvind Krishnamurthy
University of Washington
{pfonseca, kaiyuanz, xi, arvind}@cs.washington.edu

Abstract
Recent advances in formal verification techniques enabled the implementation of distributed systems with machine-checked proofs. While results are encouraging, the importance of distributed systems warrants a large scale evaluation of the results and verification practices.

This paper thoroughly analyzes three state-of-the-art, formally verified implementations of distributed systems: IronFleet, Verdi, and Chaper. Through code review and testing, we found a total of 16 bugs, many of which produce serious consequences, including crashing servers, returning incorrect results to clients, and invalidating verification guarantees. These bugs were caused by violations of a wide-range of assumptions on which the verified components relied. Our results revealed that these assumptions referred to a small fraction of the trusted computing base, mostly at the inter-

“CSmith” paper for verified distr sys
Network semantics shim is delicate

- Client-server marshaling allows users to inject commands and to crash the server
  #42 by pfons was closed on May 21, 2017

- Server assumes that it can read the entire client request with a single recv call
  #41 by pfons was closed on May 21, 2017

- Transient system call errors during recovery cause inconsistent re-initialization
  #40 by pfons was closed on May 21, 2017

- Crash during update of snapshot causes loss of data
  #39 by pfons was closed on May 21, 2017

- Server is unable to recover when disk log is incomplete due to a crash while writing an entry
  #38 by pfons was closed on May 21, 2017

- Server crashes when trying to produce large packets because of buffer overflow
  #37 by pfons was closed on May 21, 2017
Network semantics shim is delicate

Pedro et al. found several bugs, **BUT none in any verified components.**
Network semantics shim is delicate

Pedro et al. found several bugs, **BUT** none in any verified components.

**Cheerios:**
New system transformer with correct serialization implemented and verified.

---

Pedro et al. found several bugs, **BUT** none in any verified components.
Training the next generation

There will always be a TCB

we’ll always need informed judgement

Engineers unlikely to pick this up at work

but courses great evangelism opportunity

How to get this into ugrad canon?

need reusable labs and tools
Training the next generation
Training the next generation
Proof engineering

Verdi Proofalytics

- 2018-07-05 at 09:04:59 on 5d3d4d291544 in H:
  max ltac: input_serialize_deserialize_id (1024.45)
  max qed: input_serialize_deserialize_id (1024.45)
  build time: 1674 s
  admits: 0

- 2018-04-27 at 22:15:30 on 2b85412550cc in H:
  max ltac: input_serialize_deserialize_id (1064.80)
  max qed: input_serialize_deserialize_id (1064.80)
  build time: 1819 s
  admits: 0

- 2018-04-25 at 03:24:24 on fd3df3e359835 in H:
  max ltac: input_serialize_deserialize_id (1148.57)
  max qed: input_serialize_deserialize_id (1148.57)
  build time: 2000 s
  admits: 0
Proof engineering

Finally catching the interest of the SE community: ASE ’17, ICSE ’18, ISSTA ’18
Toward verified distributed systems

The Verdi Framework

Verified Raft Consensus

TCB, Tools, Teaching

Enriching Models & Modularity
Churn = nodes joining & leaving a system at run time
Punctuated safety properties

Reachable under churn (→)

Safety after churn stops (→)

Ryan Doenges
Punctuated safety properties

Reachable under churn

Safety after churn stops

System recovers

Churn stops

Error

Time
Toward verifying churn tolerance

Tree aggregation

aggregate data in sensor networks
designated root node eventually correct

Chord

distributed hash table
protocol bugs found [Zave 2015]
ing ring should eventually stabilize
Composition: A way to make proofs harder
Composition: A way to make proofs harder

“In 1997, the unfortunate reality is that engineers rarely specify and reason formally about the systems they build. It seems unlikely that reasoning about the composition of open-system specifications will be a practical concern within the next 15 years.”
“Horizontal composition”: eliminate closed world hypothesis
“Horizontal composition”: eliminate closed world hypothesis
Compositional Verif of Distr Sys

Challenges
Client reasoning
Invariants
Separation

Solutions
Protocols
\text{WITHINV} \text{ rule}
\text{FRAME} \text{ rule/Hooks}

Disel: $\vdash \{P\} \ c \ \{Q\}$

[POPL 18]
Toward verified distributed systems

The Verdi Framework

Verified Raft Consensus

TCB, Tools, Teaching

Enriching Models & Modularity
Reflections on Verdi experience

Distributed sys good fit for verification
  critical, expert-written, I/O bound cases

Biggest challenge is proof engineering
  reproving and managing scale daunting

Lots of low-hanging fruit left
  dynamic update, concurrency, optimization
The most important ingredients

James Wilcox
Doug Woos
Pavel Panchekha
Ryan Doenges

Justin Adsuara
Keith Simmons
Steve Anton
Miranda Edwards

Karl Palmskog
Ilya Sergey
Xi Wang
Mike Ernst
Tom Anderson
Thank You!

The Verdi Framework

Verified Raft Consensus

TCB, Tools, Teaching

Enriching Models & Modularity

http://distributedcomponents.net