IronFleet: Proving Practical Distributed Systems Correct

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

Microsoft Research
Buggy code update triggers Bing, Yahoo outage

Jan 5, 2015 9:01am
Filed under Search

Tags

Google blames software bug for Friday night Gmail outage

Promoted Content

Amazon Web Services suffers outage, takes down Vine, Instagram, others with it

Topic: Amazon

Summary: The cloud giant suffered an outage for about an hour on Sunday, showing once again the perils of an outsourced cloud service, as many AWS customers went down with it.

By Zack Whittaker for Between the Lines | August 26, 2013 -- 13:22 GMT (06:22 PDT)

Follow via: 

Comments 12  
Share on Facebook 0  
Tweet 1  
Share
"not one of the properties claimed invariant in [PODC] is actually true of it."

[Zave 2015]

Under the same assumptions made in the Chord papers, the [SIGCOMM] version of the protocol is not correct, and not one of the properties claimed invariant in [PODC] is actually invariantly true of it. The [PODC] version satisfies one invariant, but is still not correct. The results are presented by means of counterexamples to the invariants in Section 4. In preparation for the results, Section 2 gives a brief summary of the protocol and failure assumptions, and Section 3 introduces the invariants.
We show how to build complex, efficient distributed systems whose implementations are provably safe and live. Implementations are correct, not just abstract protocols.

First mechanical proof of liveness of a non-trivial distributed protocol, let alone an implementation.

Proof is subject to assumptions, not absolute.
What we built

IronRSL
Replicated state library

IronKV
Sharded key-value store

Complex with many features:
- state transfer
- log truncation
- dynamic view-change timeouts
- batching
- reply cache

You can build too!
IronFleet extends state-of-the-art verification to distributed systems

Key contributions of our work:
- Two-level refinement
- Concurrency control via reduction
- Always-enabled actions
- Invariant quantifier hiding
- Automated proofs of temporal logic
Not writing unit tests and having it work the first time: priceless
Outline

Introduction

Methodology

Methodology overview

Two-level refinement

Concurrency containment

Liveness

Libraries

Evaluation

Related work

Conclusions
Background: Refinement


method Main()
Specification is a simple, logically centralized state machine

```plaintext
type SpecState = int

function SpecInit(s: SpecState) : bool
{
    s == 0
}

function SpecNext(s: SpecState, s': SpecState) : bool
{
    s' == s + 1
}

function SpecRelation(realPackets: set<Packet>, s: SpecState) : bool
{
    forall p, i :: p in realPackets &&
        p.msg == MarshallCounterVal(i) ==> i <= s
}
```
method Main()
{
    var s: ImplState;
    s := ImplInit();
    while (true) {
        s := EventHandler(s);
    }
}
Proving correctness is hard

Subtleties of distributed protocols
- Maintaining global invariants
- Dealing with hosts acting concurrently
- Ensuring progress

Complexities of implementation
- Using efficient data structures
- Memory management
- Avoiding integer overflow
Two-level refinement
Protocol layer

Spec

Protocol

Implementation

```
function ProtocolNext(s: HostState, s': HostState) : bool
method EventHandler(s: HostState) returns (s': HostState)

type Message = MessageRequest() | MessageReply() | ...
type Packet = array<byte>
```
Protocol layer

Spec

Protocol

Implementation

Distributed system model
method EventHandler(s: HostState) returns (s’: HostState)
ensures ProtocolNext(s, s’);
Outline

Introduction

Methodology

Methodology overview

Two-level refinement

Concurrency containment

Liveness

Libraries

Evaluation

Related work

Conclusions
Cross-host concurrency is hard

Hosts are single-threaded, but we need to reason about concurrency of different hosts.
Cross-host concurrency is hard

Requires reasoning about all possible interleaving of the substeps
Concurrency containment strategy

Enforce that receives precede sends in event handler

Assume in proof that all host steps are atomic
Why concurrency containment works

Reduction argument: for every real trace...

[Diagram showing concurrency containment patterns]
Why concurrency containment works

Reduction argument: for every real trace...

...there’s a corresponding legal trace with atomic host steps

Constraining the implementation lets us think of the entire distributed system as hosts taking one step at a time.
Most of the work of protocol refinement is proving invariants

\[ P(j) && \text{SystemNext}(\text{states}[j], \text{states}[j+1]) \Rightarrow P(j+1) \]

\[ \text{SystemInit}(\text{states}[0]) \Rightarrow P(0) \]

Nearly all cases proved automatically, without proof annotations

automated theorem proving
### Introduction

**Methodology**

- Methodology overview
- Two-level refinement
- Concurrency containment
- Liveness

### Libraries

**Evaluation**

**Related work**

**Conclusions**
One constructs a liveness proof by finding a chain of conditions.

\[ C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4 \rightarrow \cdots \rightarrow C_n \]

- **Assumed starting condition**
- **Ultimate goal**
Simplified example

Client sends request

Replica receives request

Replica suspects leader

Leader election starts
Some links can be proven from assumptions about the network.

1. **Client sends request**
2. **Replica receives request**
3. **Replica suspects leader**
4. **Leader election starts**
5. **Network eventually delivers packets in bounded time**
Most links involve reasoning about host actions

- Client sends request
- Replica has request
- Replica suspects leader
  - One action that event handler can perform is “become suspicious”
- Leader election starts
Lamport provides a rule for proving links

Tricky things to prove:
• Action is enabled (can be done) whenever $C_i$ holds
• If Action is always enabled it’s eventually performed

Enablement poses difficulty for automated theorem proving
Always-enabled actions

- Handle a client request

- If you have a request to handle, handle it; otherwise, do nothing
Always-enabled actions allow a simpler form of Lamport’s rule.

- Action is performed infinitely often

\[ C_i \rightarrow C_{i+1} \]
To execute each action infinitely often, event handler is a simple scheduler.

Straightforward to prove that if event handler runs infinitely often, each action does too.
# Outline

<table>
<thead>
<tr>
<th>Introduction</th>
<th>Methodology</th>
<th>Evaluation</th>
<th>Related work</th>
<th>Conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Methodology overview</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two-level refinement</td>
<td>Concurrency containment</td>
<td>Liveness</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Libraries</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jay Lorch, Microsoft Research
We built general-purpose verified libraries

Temporal logic

Parsing and marshalling

Assumptions about hardware and network

Implementation

Proof

Induction

Liveness

\[
\begin{align*}
\text{uint64} & \quad \text{P} & \quad \text{uint64} \\
S_0 & \quad \rightarrow & \quad S_1 & \quad \rightarrow & \quad S_2 & \quad \rightarrow \\
\text{P} & \quad \rightarrow & \quad \text{P} & \quad \rightarrow & \quad \text{P} & \quad \rightarrow \\
\cdots & & & & &
\end{align*}
\]
Outline

Introduction

Methodology

- Methodology overview
  - Two-level refinement
  - Concurrency containment
  - Liveness

Libraries

Evaluation

Related work

Conclusions
Few lines of spec in TCB
Safety proof-to-code ratio is 5:1, comparable to Ironclad (4:1) and seL4 (26:1)
Including liveness, proof-to-code ratio is 8:1
IronRSL performance

We trade some performance for strong correctness, but no fundamental reason verified code should be slower.
IronKV performance

Throughput (req/s)

<table>
<thead>
<tr>
<th>Value size</th>
<th>Get (IronKV)</th>
<th>Get (Redis)</th>
<th>Set (IronKV)</th>
<th>Set (Redis)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 B</td>
<td>30000</td>
<td>30000</td>
<td>18000</td>
<td>18000</td>
</tr>
<tr>
<td>1 KB</td>
<td>20000</td>
<td>20000</td>
<td>11000</td>
<td>11000</td>
</tr>
<tr>
<td>8 KB</td>
<td>15000</td>
<td>15000</td>
<td>5000</td>
<td>5000</td>
</tr>
</tbody>
</table>

Throughput 25%-75% of Redis
Related work

• Single-system verification
  – seL4 microkernel [Klein et al., SOSP 2009]
  – CompCert C compiler [Leroy, CACM 2009]
  – Ironclad end-to-end secure applications [Hawblitzel et al., OSDI 2014]

• Distributed-system verification (but no liveness properties)
  – EventML replicated database [Rahli et al, DSN 2014]
    • Framework with components useful for building verified distributed systems
    • Proof only about consensus algorithm, not state machine replication
    • Uses only one layer of refinement, so all optimizations done by compiler
  – Verdi framework and Raft implementation [Wilcox et al., PLDI 2015]
    • Uses interactive theorem proving to build verified distributed systems
    • Missing features like batching, state transfer, etc.
    • Uses system transformers, a cleaner approach to composition
It’s now possible to build provably correct distributed systems...

...including both safety and liveness properties

...despite implementation complexity necessary for features and performance

To build on our code: https://github.com/Microsoft/Ironclad

To try Dafny: http://rise4fun.com/dafny

Thanks for your attention!
Running example: IronRSL replicated state library

Safety property:
Equivalence to single machine

Liveness property:
Clients eventually get replies
Specification approach:
Rule out *all* bugs by construction

- Invariant violations
- Race conditions
- Integer overflow
- Buffer overflow
- Parsing errors
- Marshalling errors
- Deadlock
- Livelock

...
Outline

Introduction

Methodology

Methodology overview

Two-level refinement

Concurrency containment

Liveness

Evaluation

Libraries

Related work

Conclusions