IronFleet: Proving Practical Distributed Systems Correct

Authors: Hawblitzel et al. (Microsoft Research)
Presenter: Hammad Ahmad
Outline

• Motivation
• Background
• IronFleet Methodology
• Evaluation
• Conclusions
Distributed Systems Are Notoriously Complex And Hard To Debug

Figure 1: Typical Figure 2 from Byzantine fault paper: Our network protocol

[Mickens ’13]
And It Is Imperative That Distributed Systems Have No Bugs

Google, Gmail return after widespread service outages
Service problems were reported in Bay Area, Los Angeles and New York, internet giant says

By REX CRUM | rcrum@bayareanewsgroup.com | Bay Area News Group
Published: August 19, 2019 at 12:50 pm | Updated: August 20, 2019 at 4:12 am

Thousands of users of Google and its Gmail suffered service outages among several issues Monday — with reports saying that the internet giant’s services remained affected in some areas for several hours.

Down Detector, a website that tracks and monitors website outages and issues, reported a spike in issues with Gmail beginning about 7:15 a.m. PDT, while problems with Google itself started to rise shortly after 11 a.m. More than 5,000 reports of outages and issues involving Google were reported, according to Down Detector, while there were nearly 5,000 such reports about problems with Gmail.

AWS Outage that Broke the Internet Caused by Mistyped Command

Amazon says Tuesday’s mayhem resulted from mistake during a routine debugging exercise.

Yevgeniy Sverdlik | Mar 02, 2017

This past Tuesday morning Pacific Time an Amazon Web Services engineer was debugging an issue with the billing system for the company’s popular cloud storage service S3 and

Instagram and Facebook Down, and Social Media Is Outraged

By LIBBY BIRK — October 30, 2019 01:59 pm EDT

Instagram and Facebook users across the world experienced outage issues Wednesday morning, prompting many to wonder what was going on with the social media platforms. As documented by Down Detector, the reported amount of problems on Instagram shot up just before 11 a.m. ET Wednesday and extended into the early afternoon, while Facebook issues began just after 8 a.m. and continued into the rest of the day. While 81 percent of the most-reported Instagram problems had to do with the news feed, others reported issues with Stories and the web version of the social medium as well. Forty-one percent of reported Facebook problems were described as a "total blackout," while 38 percent were newsfeed issues.
Why Not Test For Bugs?

• Can reveal some faults

• But…
  • Probably not all faults
  • Limited efficacy for distributed systems

• Can we do better than testing?
  • In an ideal world: prevent all faults
Enter: Formal Verification

• Raise the standard of systems from “tested” to “correct”
• Prove system behavior consistent with spec
  • Can be handwritten or machine checked
• Strong guarantees of correctness
Methods of Formal Verification

- Deductive verification (theorem proving)
- Model checking
- Equivalence checking
- And others....
Formally Verifying Distributed Protocols

- Previous work
  - EventML (Rahli et al., DSN ’14)
  - Verdi (Wilcox et al., PLDI ’15)
- Generally limited to specifications, not implementations
- Only safety proofs, not liveness
Review: Safety and Liveness

- Safety: the system cannot perform incorrect actions
  - e.g. three Paxos learners learning different values for the same ballot

- Liveness: the system will eventually perform a useful action
  - e.g. a replicated chat service eventually responds to a client request
Outline

• Motivation

• Background

• IronFleet Methodology

• Evaluation

• Conclusions
Refinement

States reachable by the implementation must be consistent with those allowed by the specification!
Floyd-Hoare Verification

- First-order predicate logic (FOL) reasoning
- Annotate a program with assertions about program states
- Verifier checks that annotations hold for all program inputs

```java
method halve(x:int) returns (y:int)
    requires x > 0;
    ensures y < x;
    { y := x / 2; }
```
Dafny (LPAR ’10)

• Imperative and functional features
  • Limited support for OO programming

• Specification constructs
  • Describe intended program behavior

• Verifier checks program behavior matches specification
Reduction

• Map behavior from concurrent systems to equivalent behavior of atomic steps

• Two steps can swap places in behavior iff swapping them has no effect on execution outcome
Temporal Logic of Actions (TLA)

• Used to reason about safety and liveness

• TLA formulas: predicates about system’s current and future states

Let $P := \text{Host } h \text{ holds the lock currently}$

$\lozenge P := h \text{ eventually holds the lock}$

$\Box P := h \text{ holds the lock currently and forever}$
Some More TLA Formulas

\[
\forall h \in \text{Hosts} : \Box \Diamond P := \text{Every host eventually holds the lock}
\]

\[
\exists h \in \text{Hosts} : \Diamond \Box P := \text{Some host eventually holds the lock forever}
\]
Outline

• Motivation
• Background
• IronFleet Methodology
• Evaluation
• Conclusions
Introducing IronFleet

• “How to build complex distributed systems whose implementations are provably safe and live”

• Why should we care?

  • Easier to implement correct, bug-free distributed systems

  • Not writing unit tests and knowing your code will work correctly the first time
A High-Level View Of The IronFleet Approach

IronFleet: a single “automated” verification framework for distributed protocols
Assumptions

• Asynchronous network
  • Network does not tamper with packets
• Dafny, .NET compiler, and Windows OS assumed correct
• Assumptions for liveness properties
The Iron-Systems

- IronRSL
  - Replicated state machine library

- IronKV
  - Sharded key-value store
The IronFleet Methodology: Two-Level Refinement

- **High-level spec**
  - $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \ldots \rightarrow S_n$

- **Distributed Protocol**
  - $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_m$

- **Implementation**
  - $I_0 \rightarrow I_1 \rightarrow I_2 \rightarrow \ldots \rightarrow I_m$
Why Do We Need A Third Layer For Refinement?

Spec-Related Difficulties
- Threats to progress
- Concurrent hosts
- Global invariants

Implementation-Related Difficulties
- Language/platform limitations
- Using efficient data structures
- Memory management
Proving Safety Properties About A System

- Invariant: a property that must hold at all times during execution

$P$ is known as an *inductive* invariant
Dealing With Non-Atomicity

- Two-level refinement proofs assume atomic protocol steps in implementation

- But, implementation event handlers are not atomic…
  - Multiple hosts concurrently sending and receiving packets of data

- Need to bridge this gap
  - Use a “reduction” argument
The “Reduction” Argument

- Take original order of events of system and posit an alternate order s.t.
  - Each host receives the same packets in the same order
  - Packet send ordering is preserved
  - Packets are never received before being sent
  - Ordering of operations on hosts is preserved
The “Reduction” Argument (cont.)

Trace of atomic host steps admits a direct refinement to the distributed protocol layer.

Equivalent Execution

Actual Execution
Why Does Reduction Work?

Observation:

If it is possible to see some packets signifying incorrect behavior in the real trace, it is also possible to do so in the corresponding atomic trace.

Implication:

Proving that the atomic trace does not have any incorrect behavior implies that the real trace does not have any incorrect behavior (by contrapositive reasoning).
Verifying Liveness

• Utilize a chain of conditions

• Prove that first condition *eventually* leads to the second, and so on…

![Diagram showing a chain of conditions]

- $C_0$ → $C_1$ → $C_2$ → $C_3$ → $\ldots$ → $C_n$

  - e.g. “Client sends a request”
  - e.g. “A replica receives the request”
  - e.g. “Client gets a response”
Proving Links

- To prove $C_i$ leads to $C_{i+1}$, need to prove:

  1. If $C_i$ holds, it does so as long as $C_{i+1}$ does not
  2. If Action occurs when $C_i$ holds, it causes $C_{i+1}$ to hold
  3. Action occurs infinitely often
Lamport’s Idea

(Yes, Lamport is involved in everything Distributed Systems)

“If action A is always enabled, i.e. always possible to do, the implementation must eventually do it.”
The IronFleet Approach: Adopt Always Enabled Actions

- Grant lock to the next host (Wrong)
- If you hold the lock, grant it to the next host; otherwise, do nothing (Correct)
Proving Links

- To prove $C_i$ leads to $C_{i+1}$, need to prove:
  1. If $C_i$ holds, it does so as long as $C_{i+1}$ does not
  2. If Action occurs when $C_i$ holds, it causes $C_{i+1}$ to hold
  3. Action occurs infinitely often
Proving An Action Runs Infinitely Often

```plaintext
method Main() {
  var s := ImplInit();
  while (true) {
    InvariantImplInvariant(s);
    
    ghost var journal_old := get_event_journal();
    ghost var ios_performed: seq<IoEvent>;
    s, ios_performed := ImplNext(s);
  
    assert get_event_journal() ==
            journal_old + ios_performed;
    assert ReductionObligation(ios_performed);
  }
}
```

Runs infinitely often
Outline

- Motivation
- Background
- IronFleet Methodology
- Evaluation
- Conclusions
## LOC and Verification Time

<table>
<thead>
<tr>
<th></th>
<th>Spec (source lines of code)</th>
<th>Proof</th>
<th>Time to Verify (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High-Level Spec:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IronRSL</td>
<td>85</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>IronKV</td>
<td>34</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Temporal Logic</td>
<td>208</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Distributed Protocol:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IronRSL Protocol</td>
<td>–</td>
<td>–</td>
<td>1202</td>
</tr>
<tr>
<td>Refinement</td>
<td>35</td>
<td>–</td>
<td>3379</td>
</tr>
<tr>
<td>Liveness</td>
<td>167</td>
<td>–</td>
<td>7869</td>
</tr>
<tr>
<td>IronKV Protocol</td>
<td>–</td>
<td>–</td>
<td>726</td>
</tr>
<tr>
<td>Refinement</td>
<td>36</td>
<td>–</td>
<td>3998</td>
</tr>
<tr>
<td>Liveness</td>
<td>98</td>
<td>–</td>
<td>2093</td>
</tr>
<tr>
<td>TLA Library</td>
<td>–</td>
<td>–</td>
<td>1824</td>
</tr>
<tr>
<td><strong>Implementation:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IO/Native Interface</td>
<td>591</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Common Libraries</td>
<td>134</td>
<td>833</td>
<td>7690</td>
</tr>
<tr>
<td>IronRSL</td>
<td>6</td>
<td>2941</td>
<td>7535</td>
</tr>
<tr>
<td>IronKV</td>
<td>6</td>
<td>1340</td>
<td>2937</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1400</td>
<td>5114</td>
<td>39253</td>
</tr>
</tbody>
</table>

**Verification Time**

**Code**

**Proof**
The Iron-Systems: Latency and Throughput

**Figure 13.** IronRSL’s performance is competitive with an unverified baseline. Results averaged over 3 trials.

**Figure 14.** IronKV’s performance is competitive with Redis, an unverified key-value store. Results averaged over 3 trials.
Conclusions

• Possible to implement provably correct distributed systems

• Verify protocol specs and implementations

• Correctness of both safety and liveness properties

• But…
Utopia? Not Really….

- Strong guarantees of correctness come with an added developer effort
  - Time cost of writing proofs in the order of person-months
  - Non-negligible learning curve for new developers to write verified code
- Trade-offs…
  - At what point is the increased cost worth the correctness guarantee?
Limitations and Future Work

- Requires considerably more developer effort
- Requires knowledge about the protocols
  - Non-negligible learning curve
- Iron-systems do not yet match baseline performances
- Future work by others attempts to fix some of these limitations
  - IVy (Padon et al., PLDI ’16)
  - I4 (Ma et al., SOSP ’19)
Conclusions

• Possible to implement provably correct distributed systems
  • Verify protocol specs and implementations
  • Correctness of both safety and liveness properties
  • But…
    • With add a significant added time/effort cost!

Thank you!