IronFleet:
Proving Practical Distributed Systems Correct

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

Presenter: Oreoluwa Alebiosu
Iron Fleet

- Build complex, efficient distributed systems whose implementations are provably safe and live.
  - Implementations are correct, not just abstract protocols
  - Proofs are machine checked
- First work to produce mechanical proof of liveness of non-distributed protocol and implementation
  - Proofs are not absolute and assume correctness of some things
  - Work is on proving correctness of your code
Iron Fleet

• Toolset modification

• Methodology
  – Two-level refinement
  – Concurrency control via reduction
  – Always-enabled actions (Liveness)
  – Invariant quantifier hiding
  – Automated proofs of temporal logic

• Libraries
Iron Fleet

Builds upon..

• Single-machine system verification (sel4)
• SMT Solvers
• Dafny

https://docs.google.com/document/d/1KaoFQrt8rQCfww939WW4p8uUeYbMx2xtNHgadWBb4OFDVA/edit?usp=sharing
Implementation

• IronRSL: Replicated state library
  — Complex with many features:
    • state transfer
    • log truncation
    • dynamic view-change timeouts
    • batching
    • reply cache

• IronKV: Sharded key-value store
IronRSL

- Safety property: Equivalence to single machine
IronRSL

- Safety property: Equivalence to single machine
- Liveness property: Clients eventually get replies
Specification approach: Rule out *all* bugs by construction

<table>
<thead>
<tr>
<th>Invariant violations</th>
<th>Parsing errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Race conditions</td>
<td>Marshalling errors</td>
</tr>
<tr>
<td>Integer overflow</td>
<td>Deadlock</td>
</tr>
<tr>
<td>Buffer overflow</td>
<td>Livelock</td>
</tr>
</tbody>
</table>
Refinement

Spec

Implementation

I0 → I1 → I2 → I3 → I4
Refinement

Spec

Implementation

S0 → S1 → S2 → S3 → S4 → S5 → S6 → S7

I0 → I1 → I2 → I3 → I4
Refinement

Spec

Implementation

S0 S1 S2 S3 S4 S5 S6 S7
I0 I1 I2 I3 I4
Proving correctness is hard

<table>
<thead>
<tr>
<th>Subtleties of distributed protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintaining global invariants</td>
</tr>
<tr>
<td>Dealing with hosts acting concurrently</td>
</tr>
<tr>
<td>Ensuring progress</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complexities of implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using efficient data structures</td>
</tr>
<tr>
<td>Memory management</td>
</tr>
<tr>
<td>Avoiding integer overflow</td>
</tr>
</tbody>
</table>
Refinement
Two Level Refinement

Spec

Protocol

Impl

S0 → S1 → S2 → S3 → S4 → S5 → S6 → S7

P0 → P1 → P2 → P3 → P4

I0 → I1 → I2 → I3 → I4
One constructs a liveness proof by finding a chain of conditions:

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

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

1. Client sends request
2. Replica receives request
3. Replica suspects leader
4. Leader election starts

Diagram shows three replicas labeled A, B, and C with arrows indicating the flow of communication and decision-making in the Paxos protocol.
Some links can be proven from assumptions about the network.
Most links involve reasoning about host actions

1. Client sends request
2. Replica has request
3. Replica suspects leader
4. Leader election starts

One action that event handler can perform is “become suspicious”
Lamport provides a rule for proving links

- \( C_i \) holds
- 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 enabled (can be done) whenever $C_i$
- If Action is always enabled, it’s eventually performed
- Action is performed infinitely often

$C_i \rightarrow C_{i+1}$

\[\text{Action}\]
Much more in the paper!

- General Purpose verifying libraries
- Invariant quantifier hiding
- Embedding temporal logic in Dafny
- Reasoning about time
- Strategies for writing imperative code
- Tool improvements
Safety proof-to-code ratio is 5:1
Liveness proof-to-code ratio is 8:1
IronRSL performance

Adding batching (~2-3 person-months) improved performance significantly

Maximum throughput (req/s)

<table>
<thead>
<tr>
<th></th>
<th>IronRSL</th>
<th>Baseline (EPaxos's Multipaxos)</th>
</tr>
</thead>
<tbody>
<tr>
<td>With batching</td>
<td>25000</td>
<td>5000</td>
</tr>
<tr>
<td>Without batching</td>
<td>15000</td>
<td>5000</td>
</tr>
</tbody>
</table>
IronKV performance

Throughput 25%-75% of Redis
Conclusions

It’s now possible to build provably correct distributed systems...

...including both safety and liveness properties

...despite implementation complexity necessary for features and performance