IronFleet

Dmitry Bondarenko, Yixuan Chen
A short survey

- How many people have the confidence that your Paxos implementation has no bug?
- How many people have proved that the implementation is correct?
- Why not?
- Solution: IronFleet!
Existing Tools

- Coq
  - Are you serious?
- TLA+
  - Good, but hard to use
  - Not related to implementation
- Dafny
  - Good, but does only support single thread model
IronFleet

- IronFleet supports distributed model with UDP connections
- IronFleet employs multiple layers refinement verification
- The top most layer is the specification
  - A program with spec can never go wrong!
- The bottom most layer is the concrete implementation
  - The code that will be compiled and distributed
- The middle layer is the distributed protocol layer
  - Connects the spec and the implementation
IronFleet - Basic Idea: Refinement

- Refinement is the transformation of an abstract specification into a concrete executable program.
- Refinement in the state machine model is represented by the mapping of states
  - Mapping from initial state to initial state
  - Mapping transition to (transition)*
IronFleet - Basic Idea: Dafny

- Dafny is a language designed for automatic verification
- Dafny uses Z3 smartly to automatically prove lemmas for developers
- In IronFleet, Dafny is used as the framework language
  - We write libraries in Dafny, and IronFleet codes invoke the library
IronFleet - Basic Idea: Example Dafny Code

1 // Multiply two numbers by addition to the next smaller multiple,
2 // the smaller multiple being computed recursively.
3 // Can you find the error below?
4
5 method Mul(x: int, y: int) returns (r: int)
6 requires 0 <= x && 0 <= y
7 ensures r == x*y
8 decreases x
9 {
10   if x == 0 {
11     r := 0;
12   } else {
13     var m := Mul(x-1, y);
14     r := m + x;
15   }
16 }
Specification Layer

- Specify the behavior in state machine model
  - Init - where the state machine starts
  - Next - how the states transits
  - SpecInit and SpecNext

```plaintext
datatype SpecState = SpecState(history: seq<HostId>)
predicate SpecInit (ss: SpecState)
{ |ss.history| == 1 & & ss.history[0] in AllHostIds() }
predicate SpecNext (ss_old: SpecState, ss_new: SpecState)
{ exists new_holder :: new_holder in AllHostIds() & &
  ss_new.history == ss_old.history + [new_holder] }
```
Distributed Protocol Layer

- A more detailed state machine
  - Still defined with Init and Next
  - HostInit and HostNext
- Why bother another layer?
  - Get rid of the complex
    implementation details when we are
    reason about the protocol

```plaintext
datatype Host = Host(held:bool, epoch:int)
predicate HostInit(s:Host, id:HostId, held:bool)
  { s.held==held && s.epoch==0 }
predicate HostGrant(s_old:Host, s_new:Host, spkt:Packet)
  { s_old.held && !s_new.held && spkt.msg.transfer?
    && spkt.msg.epoch == s_old.epoch+1 }
predicate HostAccept(s_old:Host, s_new:Host, rpkt:Packet, spkt:Packet)
  { !s_old.held && s_new.held && rpkt.msg.transfer?
    && s_new.epoch == rpkt.msg.epoch == spkt.msg.epoch
    && spkt.msg.lock? }
predicate HostNext(s_old:Host, s_new:Host, rpkt:Packet, spkt:Packet)
  { HostGrant(s_old,s_new,spkt) ||
    HostAccept(s_old,s_new,rpkt,spkt) }
```
How they connect?

- We need to define a relation (or a function) \( \text{PRef} \), which maps from protocol state to spec state.
- There are several constraints:
  - HostInit is mapped to SpecInit
  - HostNext is mapped to a sequence SpecNext
Implementation Layer

- We are finally at the place when we need to write imperative code
  - Why there is no code example?
  - They are so ugly and tedious that we don’t want to present them
- Implementation does not contain any math features
- The UDP related functions are ultimately trusted
- The implementation is runned inside a main event loop
Again, how they connect?

- Each impl state is mapped to exactly one protocol state
  - This is why we need the protocol layer
- Similar constraints still applies
- Furthermore, we need to prove for multiple hosts
Putting things together

- The connection of spec and protocol
- The connection of protocol and implementation
- Thus, we have proven that the implementation implements the spec!
  - Just need to make sure that the state mapping is correct, according to SpecRelation
One more thing

- Each state transitions in implementation are not necessarily atomic
- Goal: by proving one execution, automatically proves for all valid execution
- Reduction: an argument for equivalence between executions
One more thing

- We claim without machined-checked proof (future work) that
  Every valid execution is equivalent to the very execution where we always receive first, and then send, in a single state transition

- If our code is written in this way, we are confident with its correctness
- Extension: support clock access by enforcing the sequence:
  receive -> at most one clock access -> send
Liveness

- In the previous sections, we have proven the safety property
- We need liveness as well to prove it's correct
- Problems
  - How to encode temporal logic inside our language, specifically □ and ◇
  - How to teach the verifier to verify it
Liveness - Expressing TLA

- The temporal logic of actions is defined by a map from int(time) to states
  - The operators are just quantifiers now
- Extra hints are provided to Z3 since it’s not clever enough
  - Heuristic to select future steps together
Liveness - Proving

- Of all the liveness properties, we focus on this specific form

condition $C_i$ leads immediately to condition $C_{i+1}$.

- The chain of this statement leads to proof of arbitrary happens before relation proof.
- By restating the statement provides more fine-grained property (for example, a time limit)
- We are proving lemmas about condition, which is a predicate over state
  - Similar, we are talking about action: a predicate over state transitions
Liveness - Proving

- How to prove this statement?
- This statement holds if the following three statements hold
  - If $C_i$ holds, then it continue to hold as long as $C_{i+1}$ does not.
  - If an action $A$ occurs when $C_i$ holds, the state machine will transit into a state where $C_{i+1}$ holds
  - The transitions satisfying $A$ will happens infinitely often
- The first two statements can be proved by considering every possible pair of state transitions
- The last statement can be proved if the actions is executed fairly
  - If the action is always available to execute, then it will eventually be executed
Liveness - Proving Fairness

- We have reduce the task to prove fairness
- We want eliminate the hypothesis because of its problematic nature
  - Every transition in implementation layer is always-enabled
  - If it's not “in logic”, then it will do nothing
- If the main event loop is fair, then the fairness is ensured
  - Specifically, lemmas have been provided for the round-robin scheduler
System Implementation

- Use IronFleet to implement and verify real distributed systems
- IronRSL - Replicated State Machine Library
  - Fault-tolerant deterministic state machine
  - Guaranteed safety and liveness
  - Support for advanced features
- IronKV - Sharded Key-Value Store
  - Distributed hash table
  - Use several nodes to maximize throughput
IronRSL High-Level Spec

• Provide linearizability
  ○ Same output as single node
  ○ Use MultiPaxos for consensus
IronRSL Distributed-Protocol Layer

- Protocol
  - Proposer, acceptor, learner
- Protocol invariants
  - Agreement
- Protocol refinement
  - Replicas are equivalent to high-level state machine

```python
predicate ExistsProposal(m_set:set(Msg1b), op:Op)
  { exists p :: p in m_set && op in p.msg.votes }
predicate ProposeBatch(s:Proposer, s':Proposer)
  { if |s.lbMsgs| < quorumSize then no_op()
    else if ExistsProposal(s.lbMsgs, s.nextOp) then
      var new_batches := s.proposedBatches[s.nextOp := BatchFromHighestBallot(s.lbMsgs, s.nextOp)];
      s' := s[nextOp := s.nextOp + 1]
      [proposedBatches := new_batches]
    else ... }```

IronRSL Implementation Layer

- Transform abstract types to concrete data structures
- Find efficient ways to compute results
- Add and prove invariants
IronKV

- **High-level spec**
  - Equivalent to hash table
- **Distributed-protocol layer**
  - State for each host: hash table & delegation map
  - Uses reliable transmission
  - Every key is claimed by exactly one host
- **Implementation layer**
  - Delegation map of infinite size
  - Each host keeps list of key ranges
Common Libraries

- Generic refinement
  - Refinement between abstract types and concrete structures
- Marshalling and parsing
  - Hide heap interaction
- Collection properties
  - Reasoning about sequences, sets, maps, etc.
Evaluation

- Dafny provides real-time verification
  - 1-10 seconds to verify single method
  - 6-8 minutes to verify full system
- Small ratio of proof annotation
  - 3.6 : 1 proof annotation to code
- Code worked the first time

<table>
<thead>
<tr>
<th></th>
<th>Spec</th>
<th>Impl</th>
<th>Proof</th>
<th>Time to Verify</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High-Level Spec:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IronRSL</td>
<td>85</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>IronKV</td>
<td>34</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Temporal Logic</td>
<td>208</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Distributed Protocol:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IronRSL Protocol</td>
<td>–</td>
<td>–</td>
<td>1202</td>
<td>4</td>
</tr>
<tr>
<td>Refinement</td>
<td>35</td>
<td>–</td>
<td>3379</td>
<td>26</td>
</tr>
<tr>
<td>Liveness</td>
<td>167</td>
<td>–</td>
<td>7869</td>
<td>115</td>
</tr>
<tr>
<td>IronKV Protocol</td>
<td>–</td>
<td>–</td>
<td>726</td>
<td>2</td>
</tr>
<tr>
<td>Refinement</td>
<td>36</td>
<td>–</td>
<td>3998</td>
<td>12</td>
</tr>
<tr>
<td>Liveness</td>
<td>98</td>
<td>–</td>
<td>2093</td>
<td>23</td>
</tr>
<tr>
<td>TLA Library</td>
<td>–</td>
<td>–</td>
<td>1824</td>
<td>2</td>
</tr>
<tr>
<td><strong>Implementation:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IO/Native Interface</td>
<td>591</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Common Libraries</td>
<td>134</td>
<td>833</td>
<td>7690</td>
<td>13</td>
</tr>
<tr>
<td>IronRSL</td>
<td>6</td>
<td>2941</td>
<td>7535</td>
<td>152</td>
</tr>
<tr>
<td>IronKV</td>
<td>6</td>
<td>1340</td>
<td>2937</td>
<td>42</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1400</td>
<td>5114</td>
<td>39253</td>
<td>395</td>
</tr>
</tbody>
</table>
Performance - IronRSL

- 3 replicas
- 1-256 parallel clients
- Baseline: MultiPaxos implemented in Go
- Peak throughput within 2.4x
Performance - IronKV

- Server with 1000 keys
- 1-256 parallel threads
- Baseline: K/V store in C++
Discussion

- **Weaknesses:**
  - Weaker performance
  - Lack of existing libraries
  - Developer effort (3.7 person-years)
  - Familiarity with Dafny and C#

- **Strengths:**
  - Better distributed systems
    - Formally verified
    - Provably correct
Questions?

Team behind IronFleet