IronFleet: Proving Practical Distributed Systems Correct

Distributed systems

Concurrency
Communication cos
Partial failure

Ironfleet goal: A **practical** verified system: Good performance.

Examples:

- Paxos-based replicated state machine
- Rebalancing key-value store

Question
In the IronFleet paper, what is the *spec* for an overall system, what is the *code* that they prove meets the spec, and what is the sequence of arguments (e.g., different types of refinement) that the authors use in their proof? Which of these proof arguments are machine-checked, and which are done on paper?

Ironfleet layers

0) Spec S: centralized. Includes a *spec relation* that defines what's visible.

1) Protocol P: host-host messages

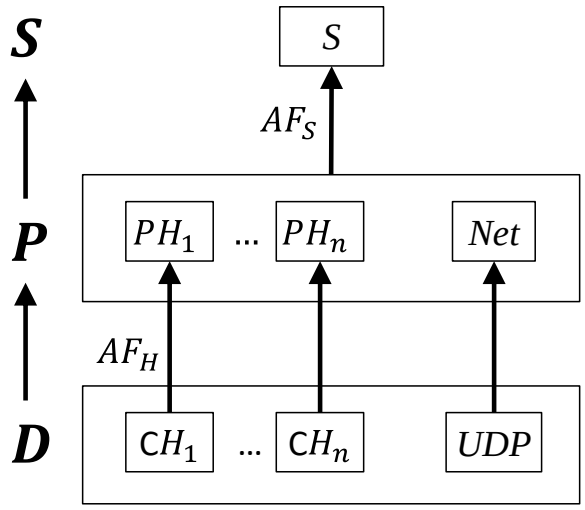
Use **TLA**. Hence, proof by invariants

Abstract state: unbounded integers, sets, sequences, messages
Visible: messages sent or received
Host PH: atomic actions $PHNext(s, s')$ on host and network state
Network: just a set of messages, no actions

Abstraction function AF_S from P state to S state.

2) Host CH: Single-host atomic actions, imperative code.

Use **sequential** reasoning + **reduction**.



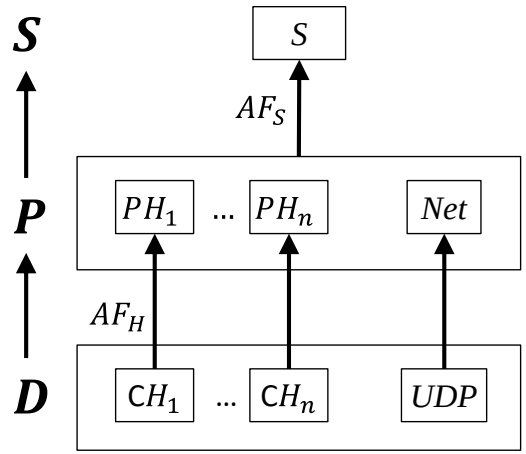
Ironfleet layers cont'd

2) Host CH: Single-host atomic actions, imperative code.
   Use sequential reasoning + reduction.
   Host code refines protocol host spec with AF_H
      $CHNext(c, c') \Rightarrow PHNext\big(AF_H(c), AF_H(c')\big)$, likewise for $Init$
      Code for each host action is sequential, usually deterministic

3) Network: UDP packets. Ghost journal of all packet sends/receives

4) Distributed system $D = N$ hosts + network, refines P

5) D refines S
   By composing P refines S and D refines P.
   The P to S abstraction function defines external visibility.



Lock example

var $held$: **seq** Id // spec state
$Init = \textbf{var}\ id \in Id; lh \coloneqq [id]$
$Next = \textbf{var}\ newId \in Id; lh \coloneqq lh +\!\!+ [newId]$

type $Pkt = [src: Id, n: \text{Nat}\,]$
var net: $[xfer, acq$: **set** $Pkt \coloneqq \{\,\}]$ // code state, net
$AR(net, held) = \forall p \in net.acq \mid p.src = held[p.n]$ // src held lock n

var $hosts$: $Id \to [lk$: Bool, n: Nat$]$ // code state, hosts

Invariant: one host holds the lock, or it's in an $xfer$ message
let $i = \{id \mid hosts(id).lk\}.\text{size}, n_{max} = \{x \in hosts.rng \mid x.n\}.\text{max}$
in $i \leq 1 \wedge (i = 0 \Rightarrow \exists p \in xfer \mid p.n = n_{max} + 1)$

Lock invariants

Atomicity of a is not enough. You also need a precondition and a postcondition for a.

That's what the lock invariant gives you. It has to be an invariant, because it's not enough for the postcondition to hold just for the first step after a.

With state machines, there's nothing special about the lock invariant. It's just a conjunct of the global invariant that holds whenever the PC is outside of a critical section.

$$I_{global} = (pc \in CS \Rightarrow I_{lock}) \wedge I_{other}$$

Quantifier hiding

Instead of $\forall x \exists y \mid p(x, y)$, provide a witness:

$y \coloneqq findY(p) \approx \{y' \mid p(x, y')\}.choose$

Makes the theorem prover's life easier: no need to search for a witness.

Ex: Every reply message has a matching request = $\forall\, reply \mid \exists\, req \in sent$

$ReplyToReq(reply, \ldots)$ **pre** $reply \in net$ **post** $req \in net$

$\{\ldots req \coloneqq \cdots\}$

Reduction (movers)

We saw this in Armada and in reasoning for locks: **commute allows move**

$\boldsymbol{a_1}; \boldsymbol{a_2}; b; c; \boldsymbol{a_3} = b; \boldsymbol{a_1}; \boldsymbol{a_2}; \boldsymbol{a_3}; c$

Reduction for locks: conflict ≡ non-commuting.

$\boldsymbol{acq_1}; b; \boldsymbol{p}; \boldsymbol{acq_2}; \boldsymbol{t}; \boldsymbol{rel_1}; c; d; \boldsymbol{rel_2} = b; \boldsymbol{acq_1}; \boldsymbol{p}; \boldsymbol{acq_2}; \boldsymbol{t}; \boldsymbol{rel_1}; \boldsymbol{rel_2}; c; d$

After rel, conflicting actions are possible.

Reduction for messages: Receive/Process*; Read time; Send/Process*

$\boldsymbol{rcv_1}; b; \boldsymbol{p}; \boldsymbol{rcv_2}; \boldsymbol{t}; \boldsymbol{snd_1}; c; d; \boldsymbol{snd_2} = b; \boldsymbol{rcv_1}; \boldsymbol{p}; \boldsymbol{rcv_2}; \boldsymbol{t}; \boldsymbol{snd_1}; \boldsymbol{snd_2}; c; d$

This rule is essential. Without it, you could have an external memory.

P and PRS commute, so P is a both mover.

R commutes with R, S commutes with S.

For R and S, it's one-way: $R_A; S_B \rightarrow S_B; R_A$—can't receive before send.

$R_A; PRS_B \rightarrow PRS_B; R_A$ // R is a right mover

$PRS_B; S_A \rightarrow S_A; PRS_B$ // S is a left mover

Reduction from obligation

Paper proof: obligation (what we can prove) ⇒ reduction (what we want)

Dafny checks that code satisfies obligation: $RP^*; T; SP^*$ in a host action.

(More recently this was also proved in Dafny.)

Standard main loop enforces the obligation

$Main()$ **{var** $s := Init()$; **while** true {

var $ev := GetEvents(), myIOs$: **seq** IO;

$s, myIOs := Next(s)$; **assert** $getEvents() = ev + myIOs$;

assert $ReductionObligation(myIOs)$ }**}**

Liveness

- Harder than safety.
- Based on *fairness* assumptions.
- For RSM, depends on complicated assumptions about failures.

Paxos-based replicated state machine (RSM)

- Deterministic machine does a sequence of commands.
- All replicas agree on the next command.
- Agreement by the Paxos consensus protocol.
- IronRSM spec: same output as single machine.

- Idea of Paxos: A sequence of rounds. In each one
  - *Propose* a command c.
  - Try for a quorum of *acceptors* (two quorums intersect).
    - Acceptor accepts c, or abstains if it's seen a later round
  - If you *learn* a quorum for c, that's the outcome.
  - Else try a new round for c', but **if $\boldsymbol{outcome = c}$ is *possible*, $\boldsymbol{c' = c}$**.

Ironfleet's RSM is practical

- Batching to amortize the cost of consensus across multiple requests
- Log truncation to constrain memory usage
- Responsive view-change timeouts to avoid timing assumptions
- State transfer to recover from extended disconnection
- A reply cache to avoid unnecessary work

- Liveness:
  - Impossible in an asynchronous system (FLP)
  - Show implied by timing assumption: a quorum stays accessible for long enough and doesn't get overloaded.

IronKV

- Non-trivial because of rebalancing.
- Otherwise it’s about sequential reasoning plus reliable messaging.
- Key invariant: every key belongs to one host, or an in-flight packet.
- Tricky data structure: finite representation of infinite key → host map.

Assumptions

- The spec for each system and the main-event loop are correct.
- The network does not tamper with packets
- Dafny, the .NET compiler and runtime, the underlying Windows OS, and the underlying hardware are correct
  - Ironclad shows how to compile Dafny code into verifiable assembly code to avoid these dependencies.
- The paper proof of obligation ⇒ reduction is correct.
- Liveness properties depend on further assumptions ☹

Pragmatics

- Verified libraries
  - Concrete → abstract refinement for common data structures
  - Marshalling: serializing and parsing
  - Properties of sets, sequences, etc.

- Ghost variables for permanent history
  - Notable, network messages

- Functional first, then refine to imperative

- Automation challenges: annotations

- Roughly 4x more code than an unverified system
  - Also needs verification experts