

Building a Reliable Database Protocol with AI, Formal Methods, and a Simulation Framework

GOTO



SPEAKERS



PhD, Anatoliy Bilenko is a Staff Engineer at Canonical, where he leads the dqlite team and runs the company's PerCorr Academy, training engineers and managers in performance analysis and formal verification. Spent a decade at Seagate leading teams working on distributed storage, performance, and transaction management



Alberto Carretero is a Software Engineer at Canonical, a key contributor to the dqlite team, where he focuses on correctness, formal methods, and the design and verification of distributed algorithms. Alberto has a background in both mathematics and computer science, though one gets more attention than the other these days.



PROBLEM STATEMENT: WHY PROTOCOLS ARE STILL HARD

Rare work in most teams

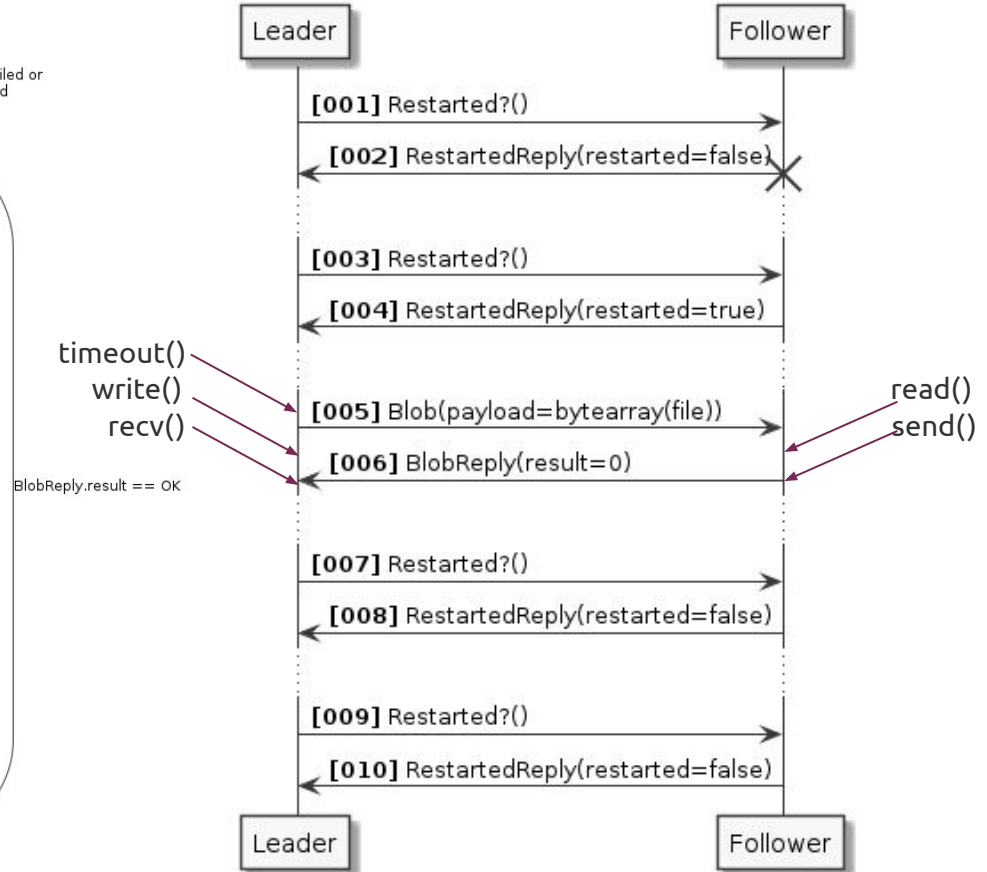
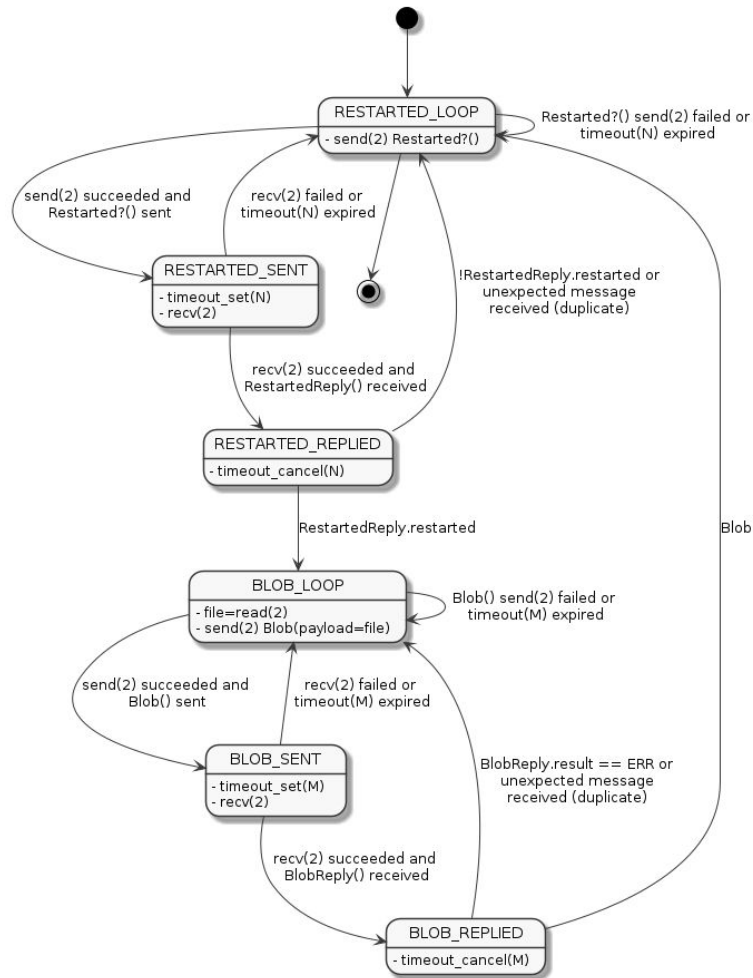
Failure assumptions stay implicit

Unexpected interleavings multiply

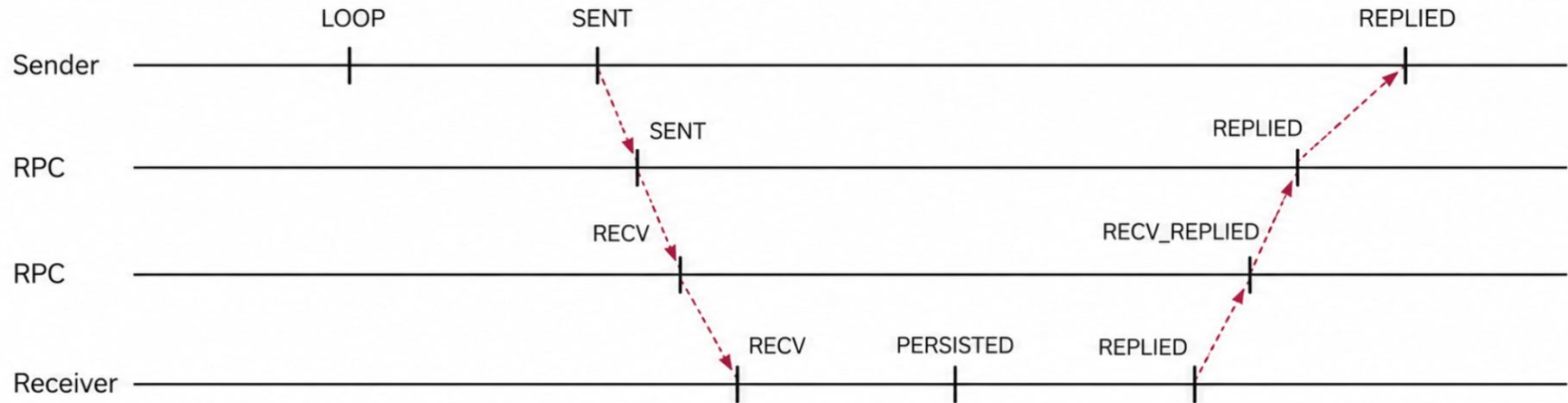
Network, disk, process, and time

Design drifts away from code

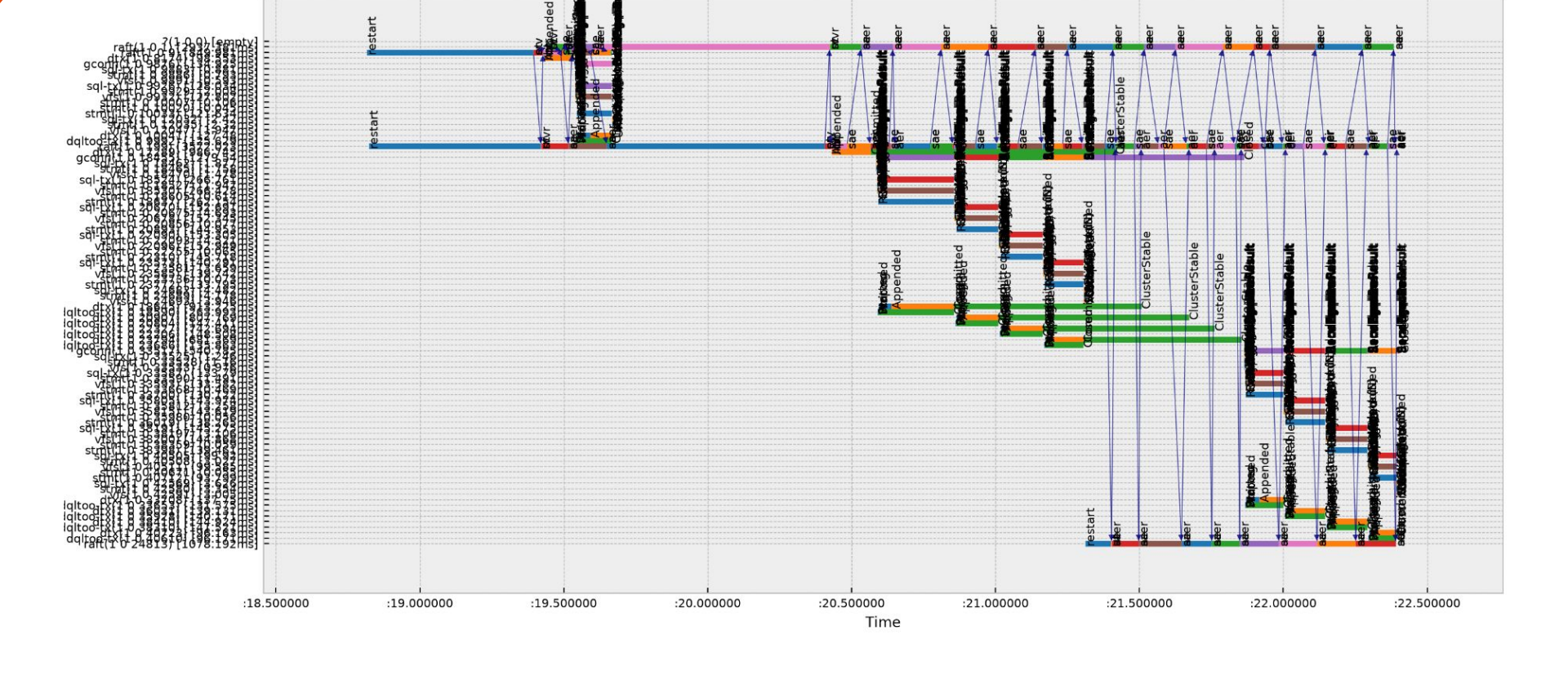
Social effects



COMPOSITION: HIERARCHICAL STATE MACHINES, HISTORIES



HISTORY: EXAMPLE



DISTRIBUTED SYSTEM PROPERTIES

Safety

- Something bad never happens;
- E.g. system never deadlocks, loses consistency, failures still happen;

Example: At most one request in flight; A reply matches a prior request

Liveness

- Something good eventually happens;
- E.g. if safety satisfied then the algorithm eventually transmits the file, failures still happen.

Example: A sent message eventually arrives

DISTRIBUTED SYSTEM SPECIFICATIONS, BEHAVIOUR SPECIFICATION

$vars \triangleq \langle sent, delivered, out \rangle$

$Init \triangleq sent = \emptyset \wedge delivered = \emptyset$
 $\wedge out = \langle \rangle$

$Send(k) \triangleq k \in Keys \setminus sent$
 $\wedge sent' = sent \cup \{k\}$
 $\wedge UNCHANGED \langle delivered, out \rangle$

$OnDelivery(k) \triangleq k \in sent$
 $\wedge IF k \in delivered$
 THEN UNCHANGED $vars$
 ELSE $delivered' = delivered \cup \{k\}$
 $\wedge out' = Append(out, k)$
 $\wedge UNCHANGED sent$

Send

Record the logical message key $k = \langle p, q, m \rangle$. The lower link sends it to q .

OnDelivery

The lower link can deliver k repeatedly. The first delivery emits an indication; later copies leave the state unchanged.

Assumptions

The lower link cannot create messages. It eventually delivers messages between correct processes.

DISTRIBUTED SYSTEM SPECIFICATIONS, PROPERTIES

$Keys \triangleq Proc \times Proc \times Msg$

$sent, delivered \subseteq Keys$

$out \in Seq(Keys)$

$CountOut(k) \triangleq |\{i \in 1..Len(out) : out[i] = k\}|$

$PL1 \triangleq \forall k \in Keys :$

$(k[1] \in Correct \wedge k[2] \in Correct \wedge k \in sent)$

$\rightsquigarrow (k \in delivered)$

$PL2 \triangleq \Box(\forall k \in Keys : CountOut(k) \leq 1)$

$PL3 \triangleq \Box(delivered \subseteq sent)$

PL1 Reliable delivery

Liveness: every message sent between correct processes is eventually delivered. Assumes fair delivery by the lower link.

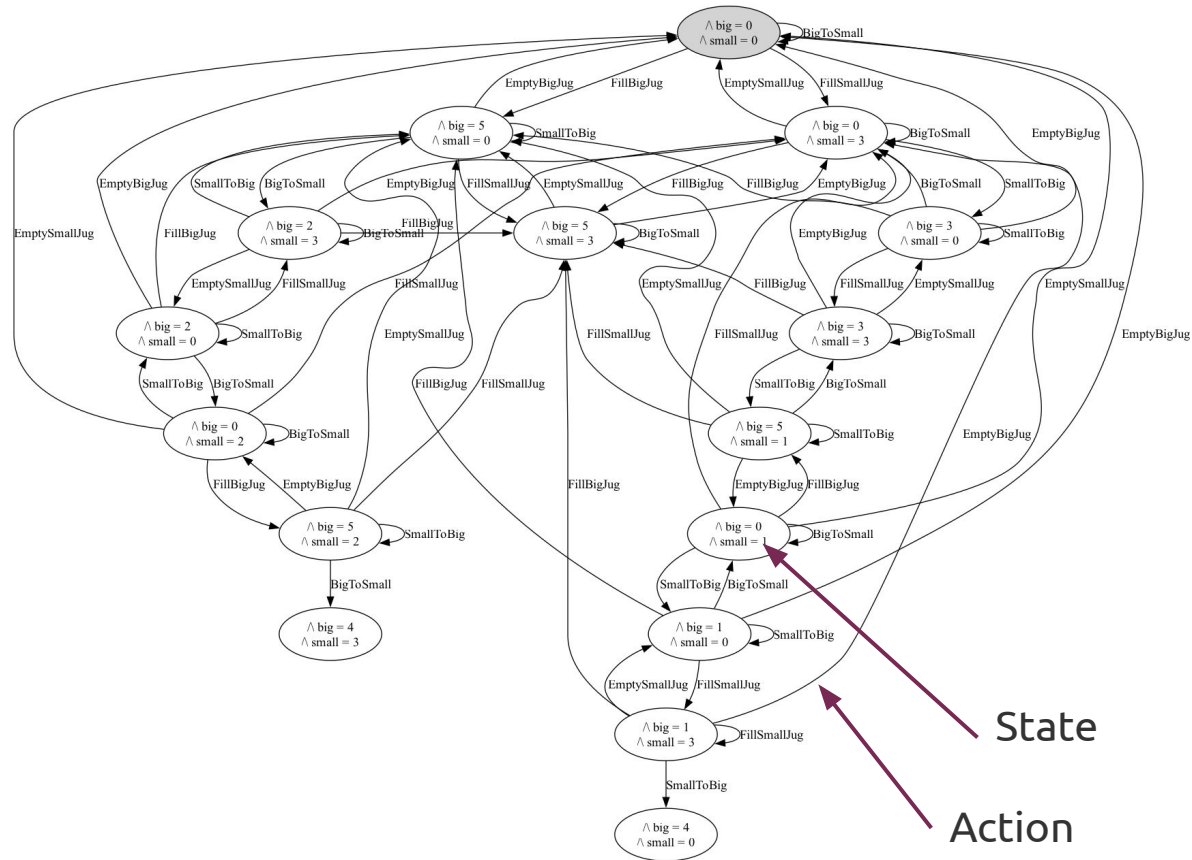
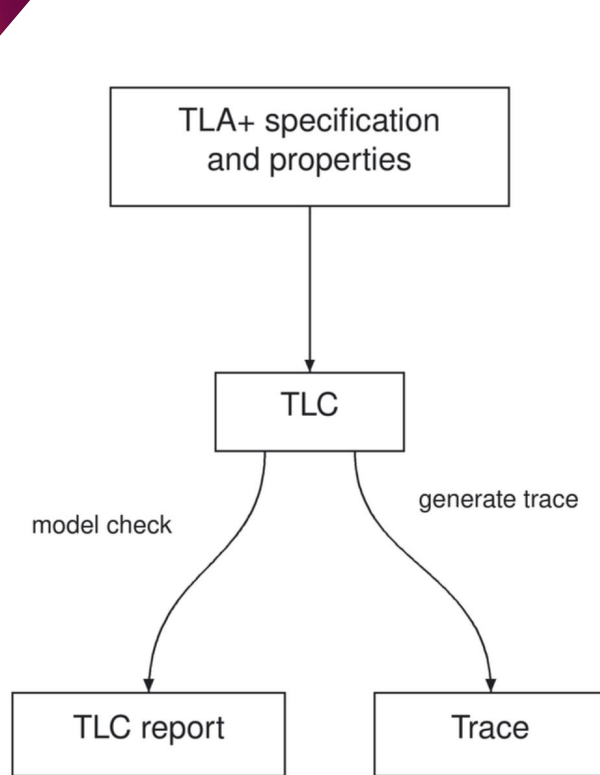
PL2 No duplication

Safety: a logical message produces at most one delivery indication.

PL3 No creation

Safety: every delivered message was previously sent.

MODEL CHECKING AND TRACE GENERATION



LOCAL PROPERTY TESTS

```
fn sender_sm_invariant(sm: &SenderSm) -> bool {  
    match sm.state {  
        SenderState::BlobLoop => !sm.file.is_open(),  
        SenderState::BlobSent => sm.file.is_open() &&  
            sm.is_timeout_set(),  
        SenderState::BlobReplied => sm.file.is_closed() &&  
            !sm.is_timeout_set(),  
    }  
}
```

GLOBAL PROPERTIES

Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3

Figure 3: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

$ElectionSafety \triangleq$

$\forall e1, e2 \in elections :$

$e1.eterm = e2.eterm \Rightarrow e1.eleader = e2.eleader$

$LeaderAppendOnly \triangleq$

$\forall e \in elections :$

$(state[e.eleader] = Leader \wedge currentTerm[e.eleader] = e.eterm)$

$\Rightarrow \wedge Len(e.elog) \leq Len(log[e.eleader])$

$\wedge \forall k \in 1 \dots Len(e.elog) :$

$log[e.eleader][k] = e.elog[k]$

$LogMatching \triangleq$

$\forall i, j \in Server :$

$\forall k \in 1 \dots Min(\{Len(log[i]), Len(log[j])\}) :$

$log[i][k].term = log[j][k].term$

$\Rightarrow \forall k2 \in 1 \dots k : log[i][k2] = log[j][k2]$

$LeaderCompleteness \triangleq$

$\forall c \in committed :$

$\forall e \in elections :$

$c.eterm < e.eterm \Rightarrow$

$\wedge c.index \in 1 \dots Len(e.elog)$

$\wedge e.elog[c.index] = c.entry$

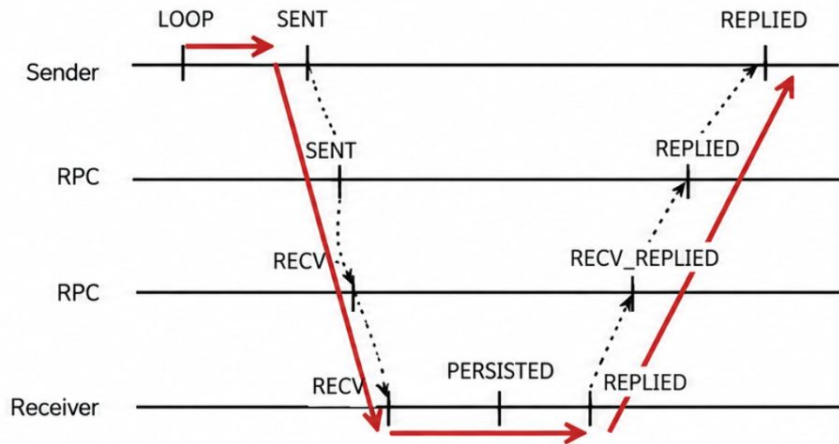
$StateMachineSafety \triangleq$

$\forall i, j \in Server :$

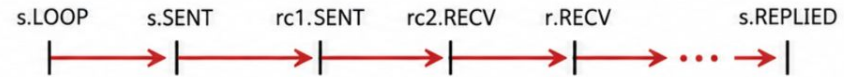
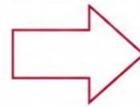
$\forall k \in 1 \dots Min(\{commitIndex[i], commitIndex[j]\}) :$

$log[i][k] = log[j][k]$

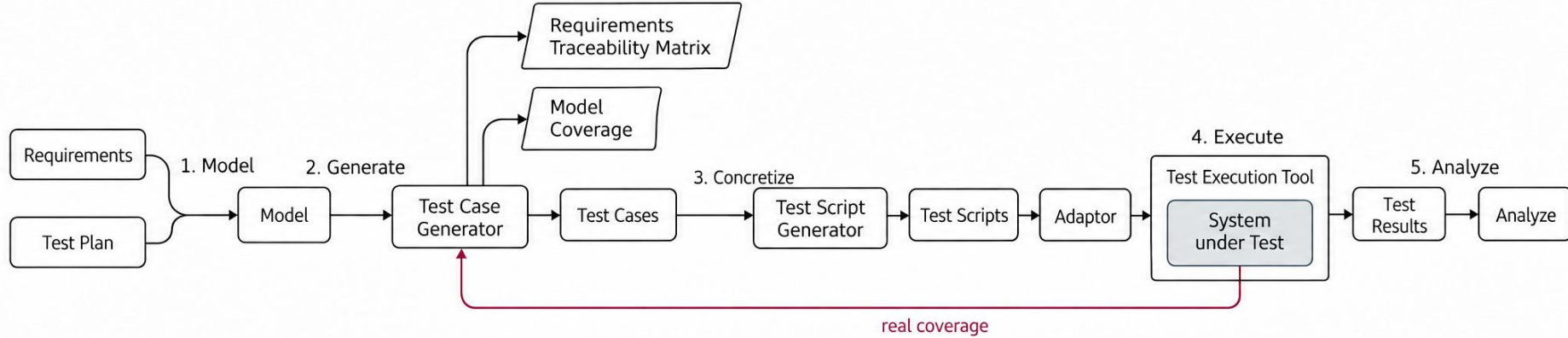
RECONSTRUCT GLOBAL STATE FROM LOGS



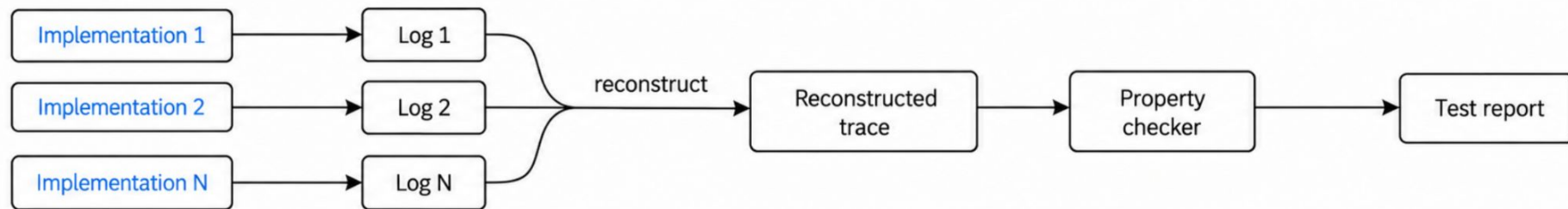
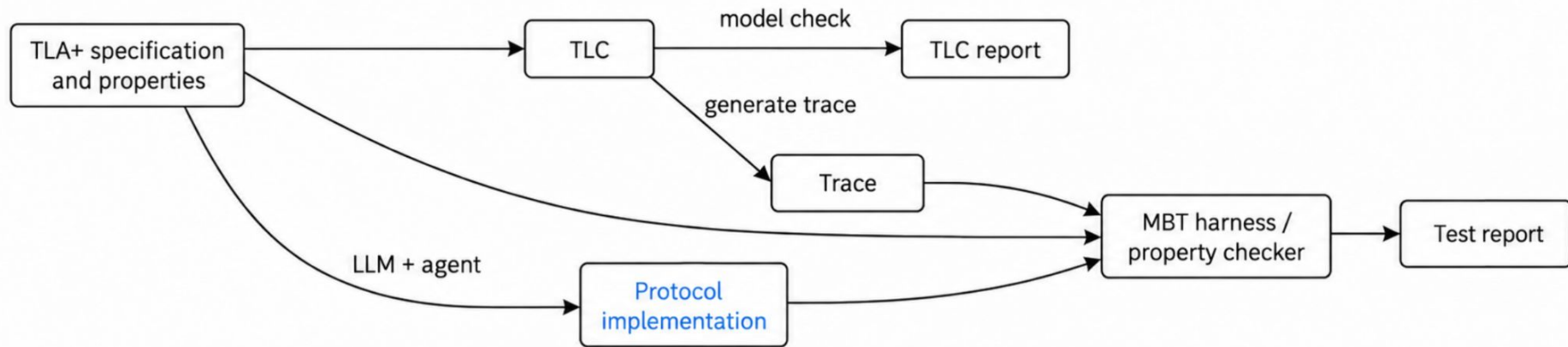
topological sort



BRING MODEL TRACES INTO THE TEST CODE



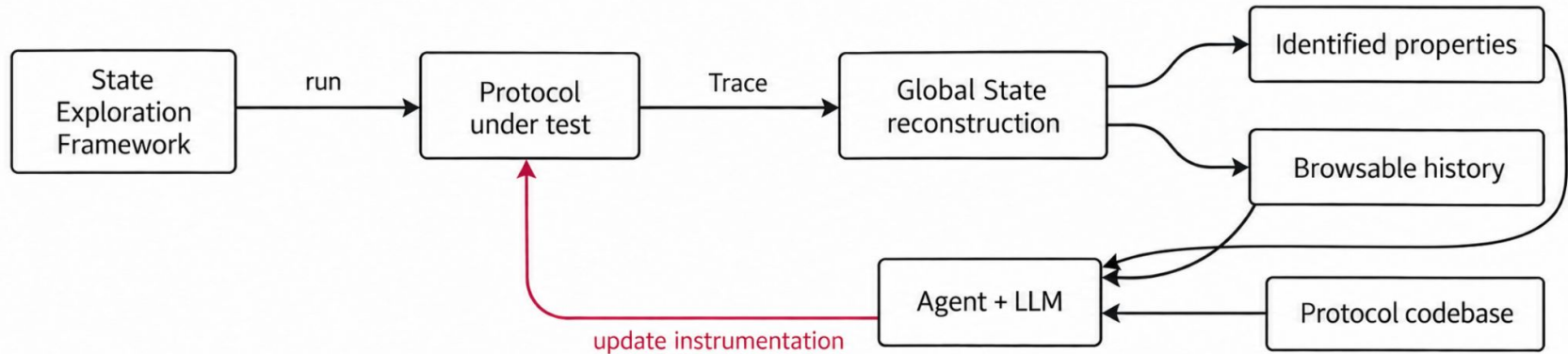
TOOL FLOW



EXPLORE BEHAVIOUR OF LEGACY SYSTEMS

Exploration frameworks:

- DST (Antithesis)
- Randomized (Jepsen)





Limitations & solutions

LIMITATIONS: TLC, PROBLEM

Process A actions:

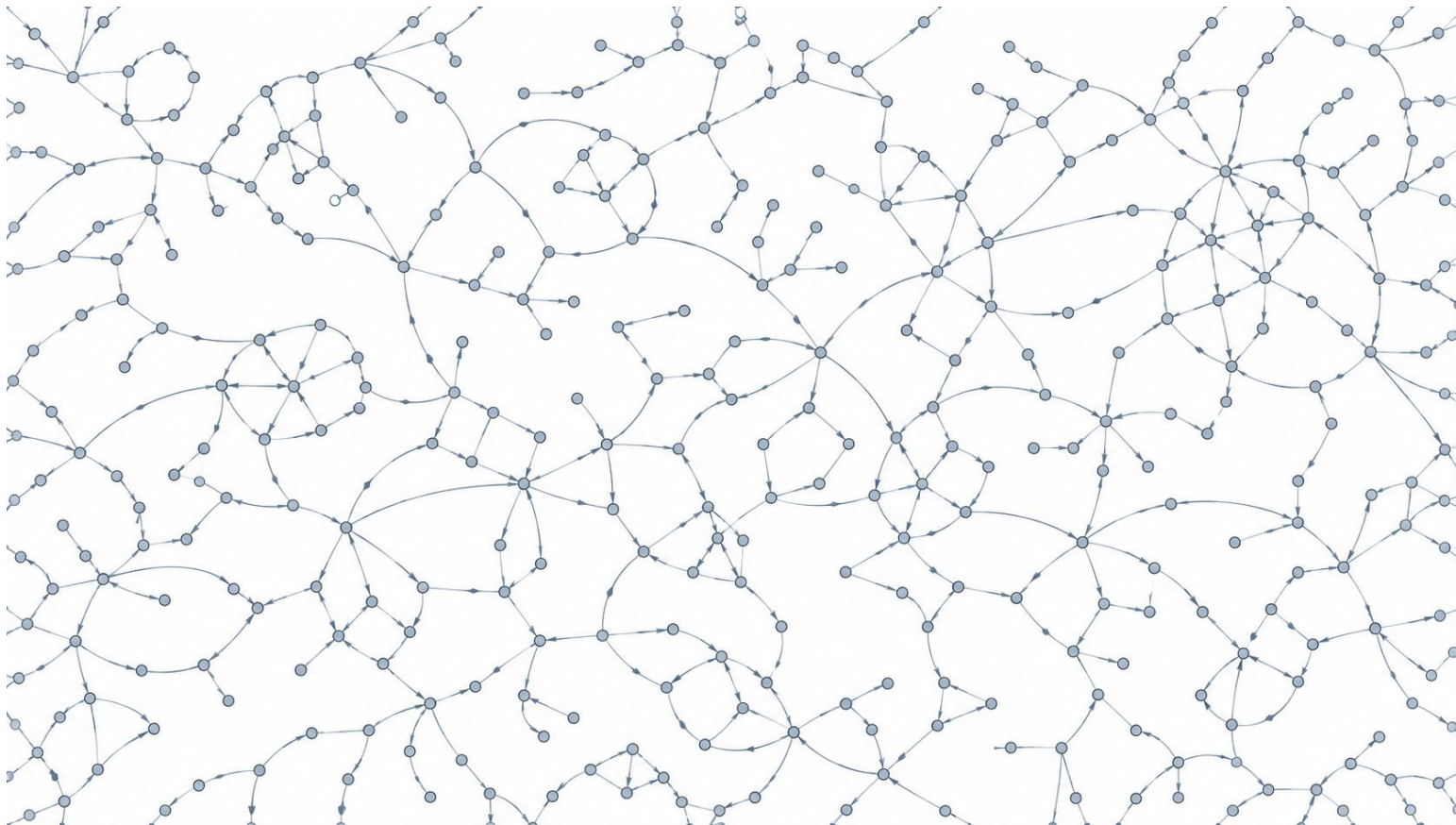
1. Read shared counter
2. Increment local copy
3. Update shared counter

Process B actions:

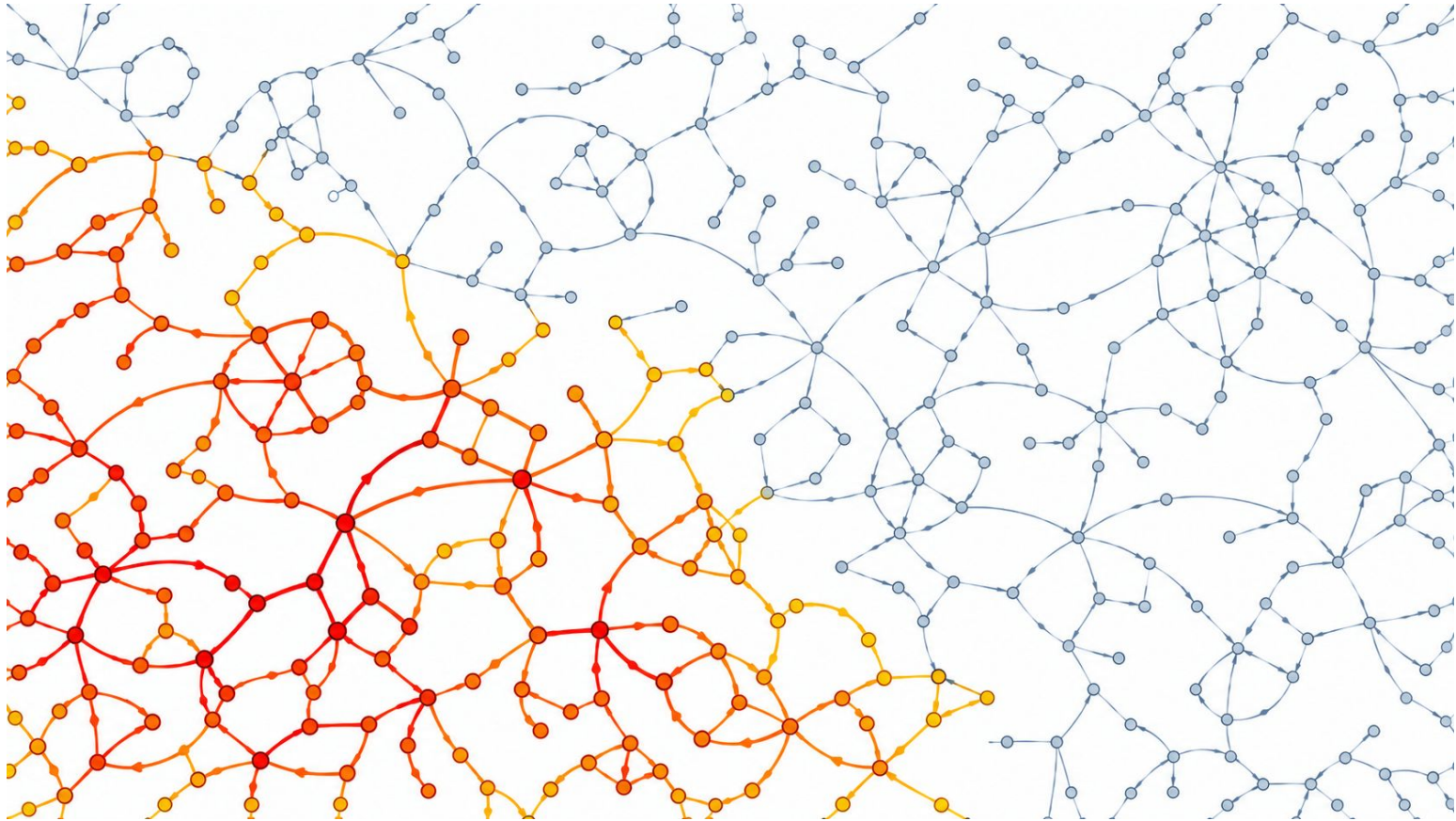
1. Read shared counter
2. Increment local copy
3. Update shared counter

TLC reports 100% coverage!

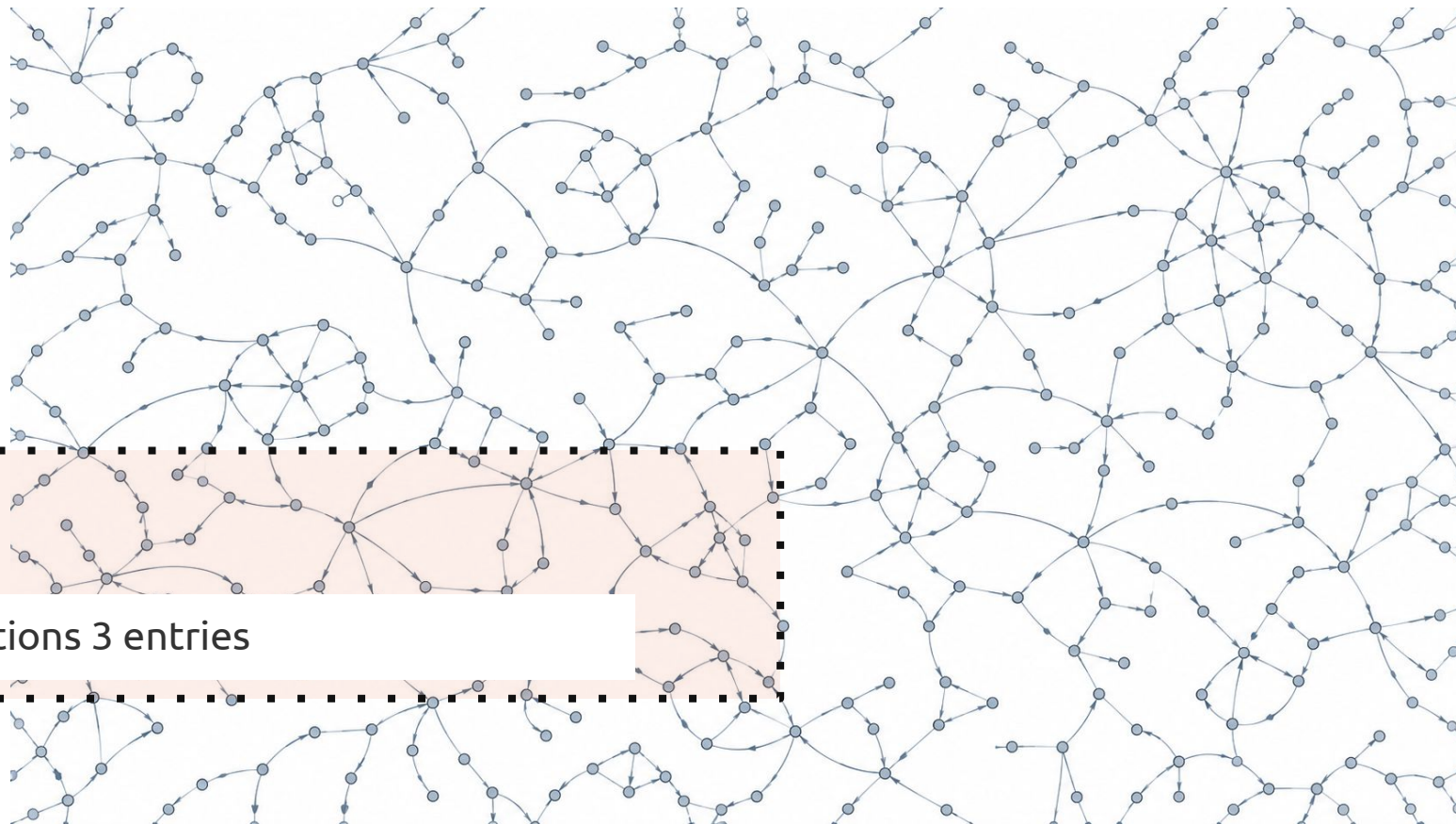
LIMITATIONS: TLC, PROBLEM



LIMITATIONS: TLC, PROBLEM



SOLUTIONS: TLC



2 elections 3 entries

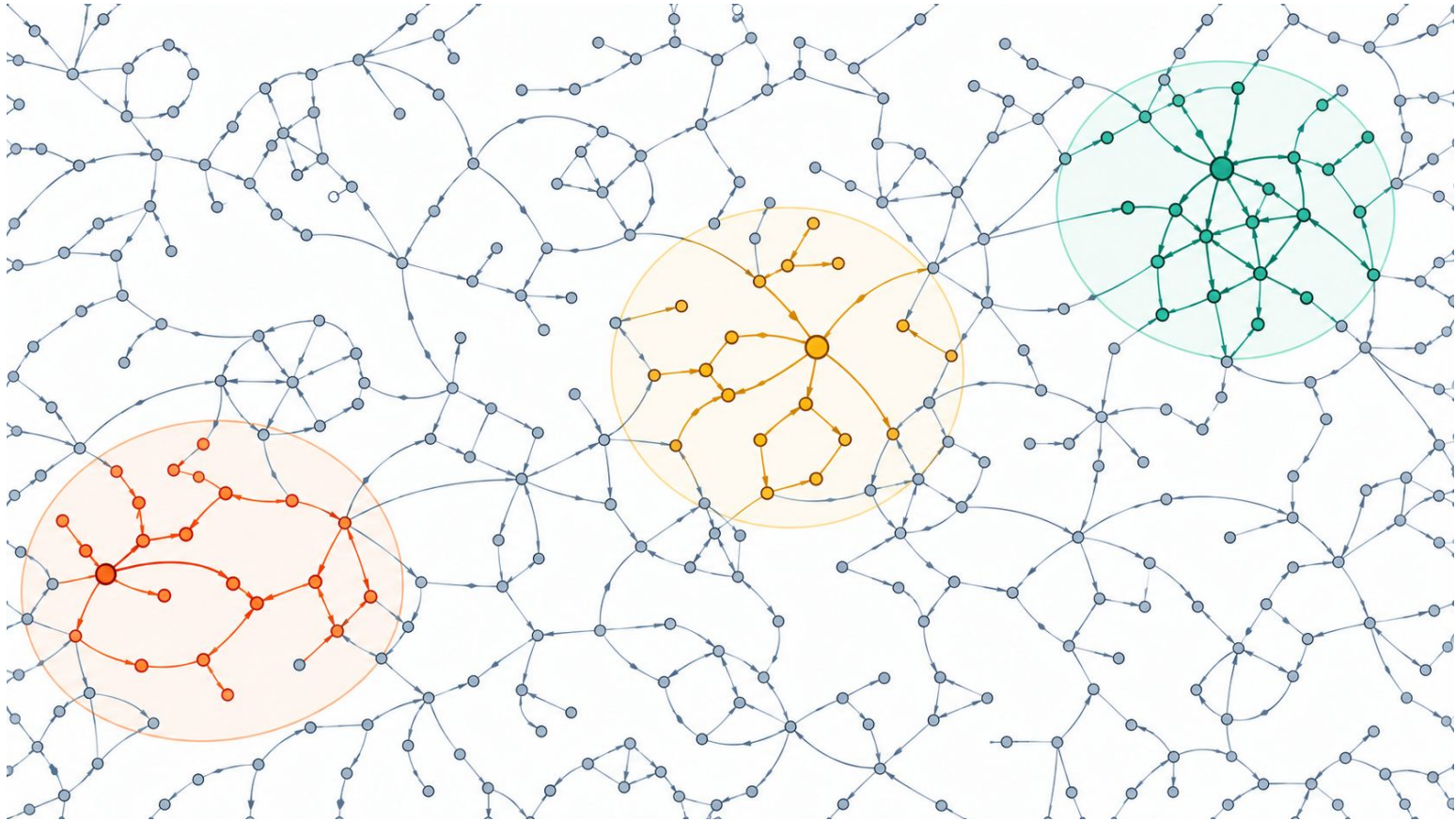
SOLUTIONS: TLC

- Define “interesting” predicates as coverage. For example, “two nodes start an election simultaneously”.

SOLUTIONS: TLC

- Define “interesting” predicates as coverage. For example, “two nodes start an election simultaneously”.
- Define “interesting” known starting points.

SOLUTIONS: TLC



SOLUTIONS: TLC

- Define “interesting” predicates as coverage. For example, “two nodes start an election simultaneously”.
- Define “interesting” known starting points.
- Guide the model-checker: Apache json-rpc server, Q-learning strategies for TLC.

SOLUTIONS: TLC

- Define “interesting” predicates as coverage. For example, “two nodes start an election simultaneously”.
- Define “interesting” known starting points.
- Guide the model-checker: Apache json-rpc server, Q-learning strategies for TLC.
- Create a simulator and a fuzzer.

SOLUTIONS: TLC

- Define “interesting” predicates as coverage. For example, “two nodes start an election simultaneously”.
- Define “interesting” known starting points.
- Guide the model-checker: Apache json-rpc server, Q-learning strategies for TLC.
- Create a simulator and a fuzzer.
- Use a simulator and a fuzzer: Antithesis.

LIMITATIONS: ANTITHESIS, JEPSEN

Antithesis:

- Deterministic
- Can go back in time before the failure
- Guided exploration, good coverage
- Deployment is somewhat limited as it runs in a sandbox

Jepsen:

- Very flexible failures and testing
- Works in any environment / setup
- Can't go back in time
- Not deterministic
- Random, not guided

OTHER APPROACHES: AI + PROOFS

► Codex prove my spec is safe, no mistakes!

...

All obligations proved

LIMITATIONS: TLAPS/AI

- With proofs you know the model is correct but you have no information about the implementation.

LIMITATIONS: TLAPS/AI

- With proofs you know the model is correct but you have no information about the implementation.
- LLM is very good at “known” proofs but you still need to guide it.
- LLM cheats.

ASSUME Cheating == Statement

...

...

THEOREM Statement

BY Cheating

LIMITATIONS: TLAPS/AI

- With proofs you know the model is correct but you have no information about the implementation.
- LLM is very good at “known” proofs but you still need to guide it.
- LLM cheats.
- Proving bigger specs is exponentially harder.

ASSUME Cheating == Statement

...

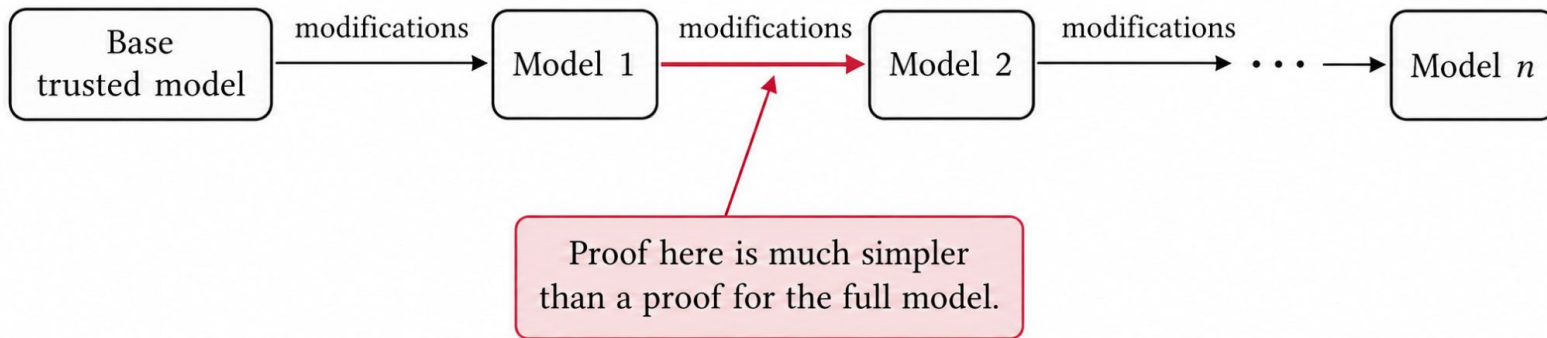
...

THEOREM Statement

BY Cheating

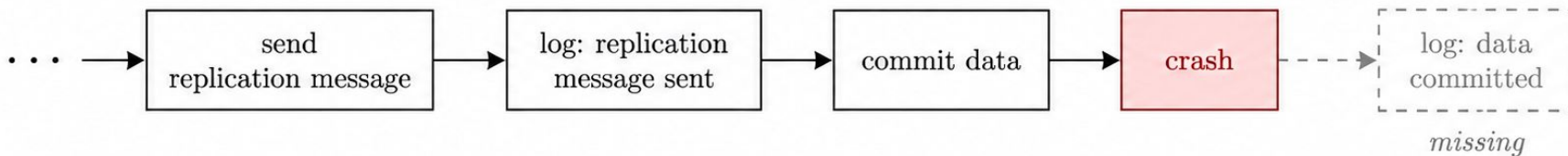
SOLUTIONS: TLAPS/AI

- Instead of big safety proofs, prove that the transformations you introduce to the spec are safe assuming the original one is.
- Use existing TLA+ specs as ground truths.



LIMITATIONS: GLOBAL STATE RECONSTRUCTION

- Hard in the present of failures.



- Distributed snapshots are hard in general, algorithms like Chandy-Lamport or Lai-Yang.

SUMMARY

Design:

- Model / State machines
- Properties
- TLA+ specs
- Proofs

Reasoning about running system:

- Traces
- Globally reconstructed traces

Testing:

- Model-based Testing
- Deterministic Simulation Testing
- Property-based testing
- Randomized testing
- Fuzzing



Questions