Bridging the Verifiability Gap

Why we need more from our specs and how we can get it

Jordan Halterman

An Operator Led Consortium

Overview

Distributed Systems at ONF

The Verifiability Gap

Model-Based Trace Checking

Model-Based Conformance Monitoring

What We Learned Along the Way

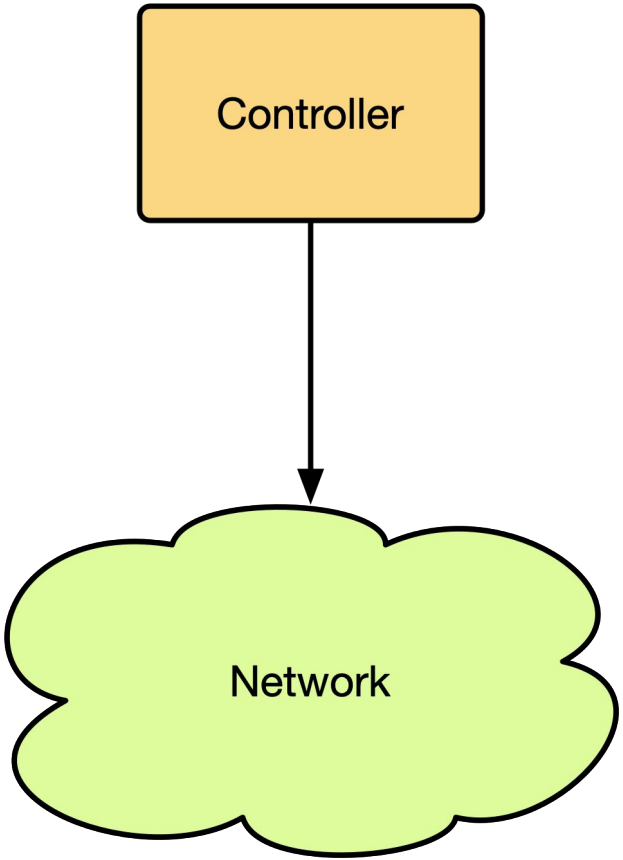
Distributed Systems at ONF

ONF

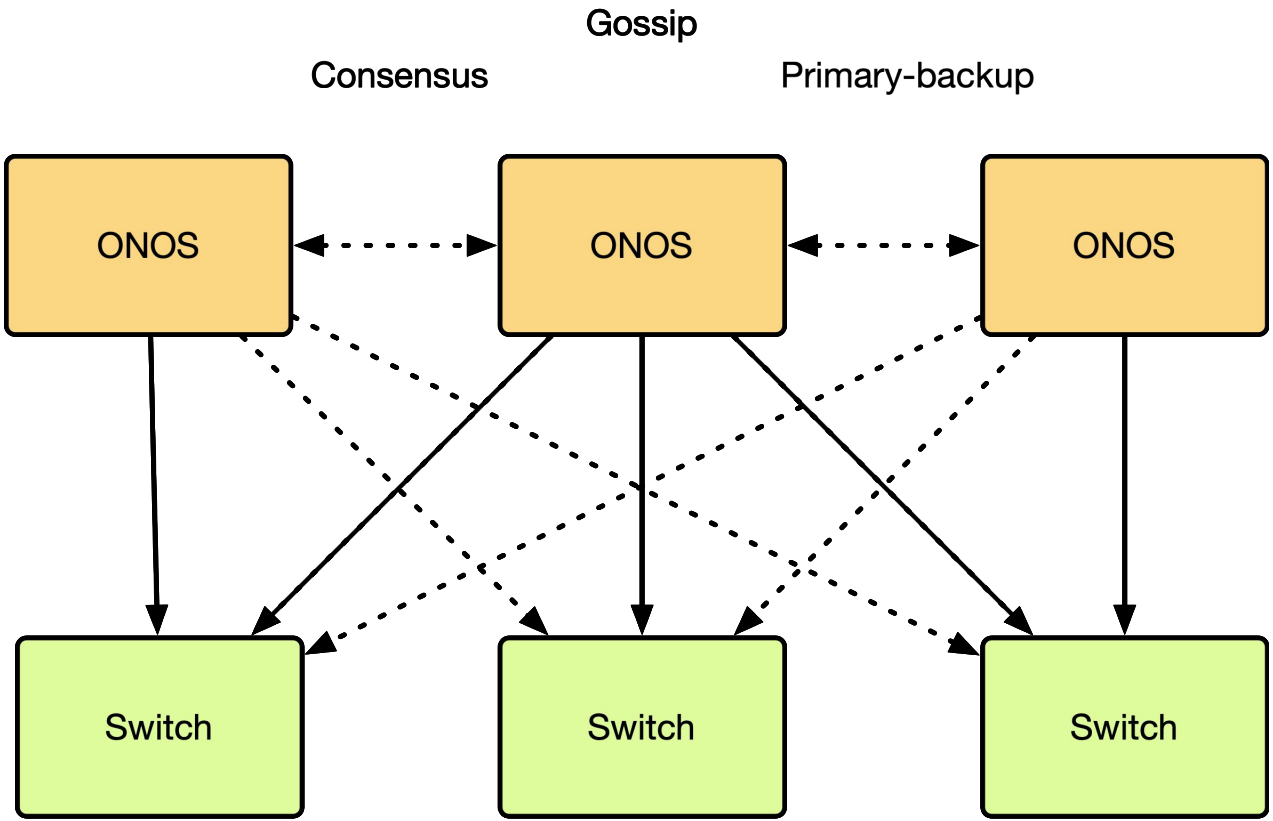
- The Open Networking Foundation is an industry funded open source foundation
- Dedicated to bringing software-defined networking technology to industry
- Small engineering staff develops ambitious projects

- The Open Network Operating System (ONOS)
- Open source network controller
- The first project created at ONF
- Brought to production in a nationwide network

ONOS



ONOS


Gossip
Consensus
Primary-backup
ONOS
ONOS
ONOS
Switch
Switch
Switch

The Verifiability Gap

ONOS in Production

- In 2018 we began field trials of ONOS
- Production scale testing exposed distributed systems bugs that had laid dormant for years
- Spent hours and often days scanning trace logs to identify bugs
- After years of work, ONOS was eventually deployed in production in a nationwide network

TLA+ at ONF

- TLA+ a critical tool for productionizing ONOS
  - Designing new distributed systems protocols
  - Improving existing distributed systems protocols
- TLA+ in ONOS
  - Extending the Raft consensus protocol
  - Distributed locking algorithms
  - Custom primary-backup protocols
  - Network-optimized consensus protocols
- Helped validate solutions for numerous bugs
- Could have been more effective if used in initial design

A New ONOS

- In 2019, the ONOS team began a complete rewrite of ONOS using cloud native architecture

Opportunity!

A New ONOS

- Focus on testing and debugging infrastructure
- How can we reduce the number of bugs?
- How can we making debugging easier?

A New Commitment to TLA+

- Began using TLA+ to design new systems
- Document and verify algorithms
- Provide a foundation for experimenting with enhancements
- Used to
  - Design new leader election algorithm
  - Verify control loop logic
  - Design distributed cache

Now we know the algorithm is correct...

How do we know the code is correct?

The Ideal Solution

- Design a new algorithm with TLA+
- Verify the new algorithm with TLC
- Implement the algorithm with Go/Java/etc
- Verify the implementation against TLA+ spec
- Debug the implementation using TLA+ spec

Why TLA+?

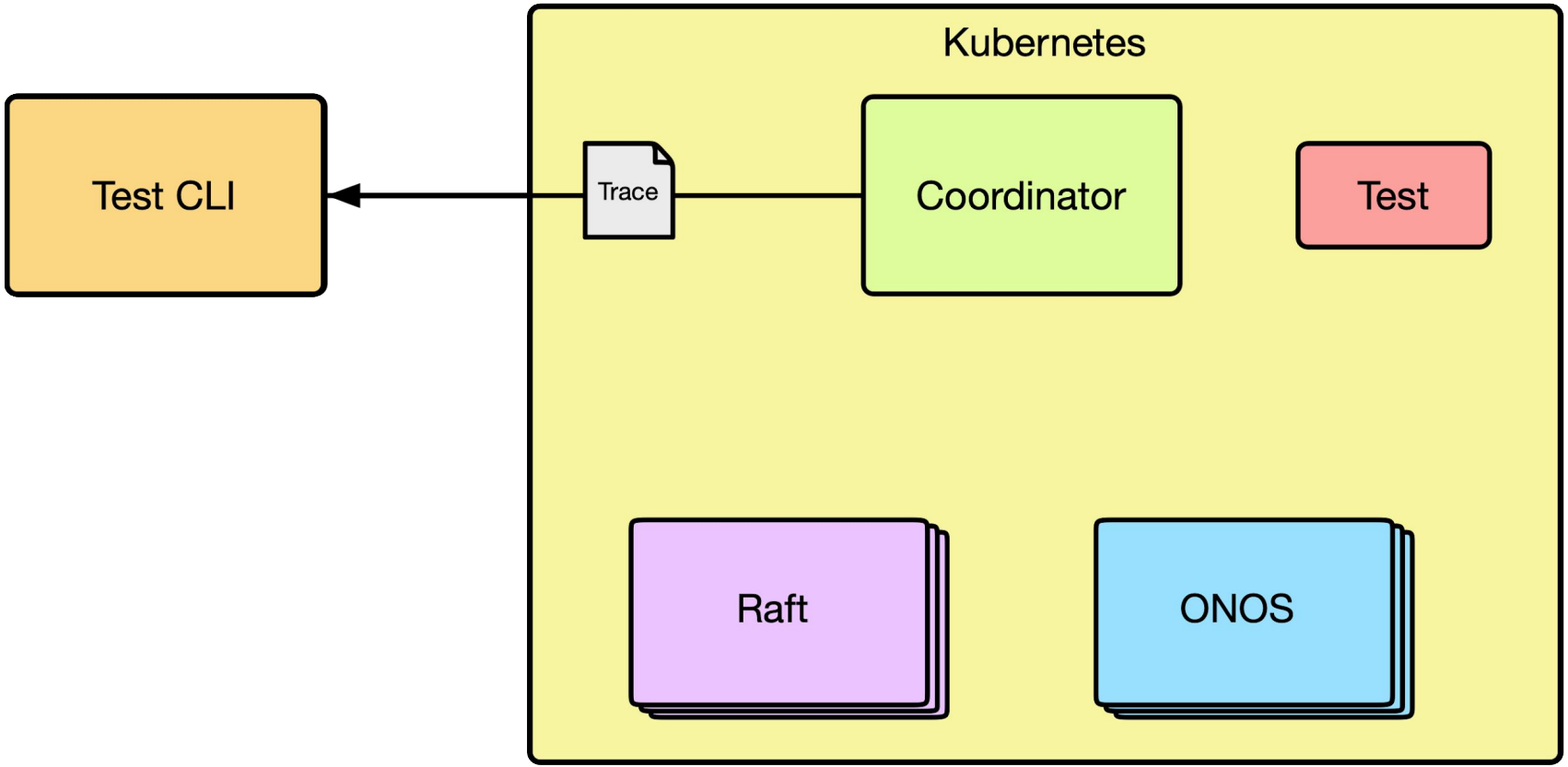
- Algorithms already specified in TLA+
- Using an alternative tool would present the same problem: maintaining consistency with the TLA+ spec
- Could help encourage the use of TLA+ to design new algorithms

Model-Based Trace Checking

Model-Based Trace Checking

- Run application
- Log structured (e.g. JSON) traces
- Consume structured traces in TLA+
- Change model state
- Verify state adheres to invariants

Test Framework



Trace Checking

MODULE *MapCacheTrace*

EXTENDS *Naturals*, *Sequences*, *TLC*, *Trace*

VARIABLE *reads*

VARIABLE *events*

VARIABLE *i*

INSTANCE *MapHistory* WITH *history* ← *reads*, *events* ← *events*

$$
\begin{aligned}
&Read \triangleq \\
&\quad \text{LET } record \triangleq Trace[i'] \\
&\quad \text{IN} \\
&\qquad \lor \land \lor record.op = \text{“put”} \\
&\qquad\quad \lor record.op = \text{“get”} \\
&\qquad\quad \lor record.op = \text{“remove”} \\
&\qquad\quad \land RecordRead(record.process,\ record.key,\ record.version) \\
&\qquad\quad \land \text{UNCHANGED } \langle events \rangle \\
&\qquad \lor \land record.op = \text{“event”} \\
&\qquad\quad \land RecordEvent(record.process,\ record.key,\ record.version) \\
&\qquad\quad \land \text{UNCHANGED } \langle reads \rangle
\end{aligned}
$$

$$
\begin{aligned}
&Init \triangleq \\
&\quad \land i = 0
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ & \lor i < Len(Trace) \land i' = i + 1 \land Read \\
& \lor \text{UNCHANGED } \langle i,\ reads,\ events \rangle
\end{aligned}
$$

$$Spec \triangleq Init \land \Box[Next]_{\langle i,\ reads,\ events \rangle}$$

Trace Checking

Record a read to the history

$$
\begin{aligned}
&RecordRead(c,\ k,\ v) \triangleq \\
&\quad \land \lor \land c \in \text{DOMAIN } history \\
&\qquad\ \ \land k \in \text{DOMAIN } history[c] \\
&\qquad\ \ \land history' = [history \text{ EXCEPT } ![c][k] = Append(history[c][k],\ v)] \\
&\qquad \lor \land c \in \text{DOMAIN } history \\
&\qquad\ \ \land k \notin \text{DOMAIN } history[c] \\
&\qquad\ \ \land history' = [history \text{ EXCEPT } ![c] = history[c] \mathbin{@@} (k :> \langle v \rangle)] \\
&\qquad \lor \land c \notin \text{DOMAIN } history \\
&\qquad\ \ \land history' = history \mathbin{@@} (c :> [i \in \{k\} \mapsto \langle v \rangle])
\end{aligned}
$$

Trace Checking

The state invariant checks that the client's history never go back in time

$$
\begin{aligned}
&\mathit{StateInvariant} \triangleq \\
&\quad \land \forall\, c \in \text{DOMAIN } \mathit{history} : \\
&\qquad \land \forall\, k \in \text{DOMAIN } \mathit{history}[c] : \\
&\qquad\quad \land \forall\, r \in \text{DOMAIN } \mathit{history}[c][k] : \\
&\qquad\qquad r > 1 \Rightarrow \mathit{history}[c][k][r] \geq \mathit{history}[c][k][r-1]
\end{aligned}
$$

Challenges

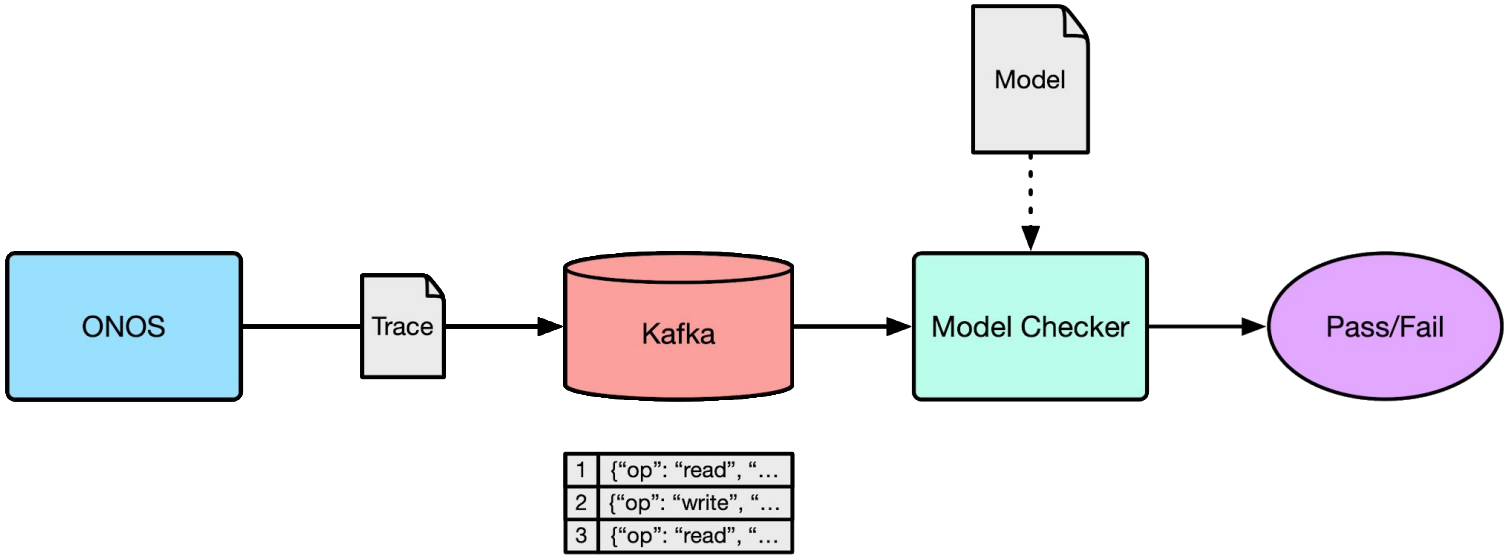
- Worked great for checking client-centric consistency models
- Still not obvious how to ensure the code correctly implements every step in the spec
- Production experience limits confidence in our ability to produce safety violations in test environments
- Need to be able to detect bugs when they occur rather than relying on our ability to make them occur

Model-Based Conformance Monitoring

Conformance Monitoring

- Near-real time trace checking
- Log application traces to stream
- Consume stream in TLC process
- Update model state
- Verify state adheres to invariants
- Alert when invariant is violated

Conformance Monitoring



https://github.com/onosproject/tlaplus-monitor

Conformance Monitoring

MODULE *MapCacheTrace*

EXTENDS *Naturals*, *Sequences*, *TLC*, *Trace*

VARIABLE *reads*

VARIABLE *events*

VARIABLE *offset*

INSTANCE *Traces*

INSTANCE *MapHistory* WITH *history* ← *reads*, *events* ← *events*

```
Read ≜
    ∧ offset' = offset + 1
    ∧ LET
        record ≜ Trace(offset')
      IN
        ∨ ∧ ∨ record.op = "put"
              ∨ record.op = "get"
              ∨ record.op = "remove"
          ∧ RecordRead(record.process, record.key, record.version)
          ∧ UNCHANGED ⟨events⟩
        ∨ ∧ record.op = "event"
          ∧ RecordEvent(record.process, record.key, record.version)
          ∧ UNCHANGED ⟨reads⟩

Init ≜
    ∧ offset = 0
    ∧ reads = [p ∈ {} ↦ [k ∈ {} ↦ ⟨⟩]]
    ∧ events = [p ∈ {} ↦ [k ∈ {} ↦ ⟨⟩]]
Next ≜
    ∨ Read
    ∨ UNCHANGED ⟨offset, reads, events⟩

Spec ≜ Init ∧ □[Next]⟨offset, reads, events⟩
```

Conformance Monitoring

$$
\begin{aligned}
Read \triangleq\ & \wedge\ offset' = offset + 1 \\
& \wedge\ record \triangleq Trace(offset') \\
& \wedge\ \vee\ \wedge\ \vee\ record.op = \text{“put”} \\
& \qquad\ \vee\ record.op = \text{“get”} \\
& \qquad\ \vee\ record.op = \text{“remove”} \\
& \qquad\ \wedge\ RecordRead(record.process,\ record.key,\ record.version) \\
& \qquad\ \wedge\ \text{UNCHANGED}\ \langle events \rangle \\
& \quad\ \vee\ \wedge\ record.op = \text{“event”} \\
& \qquad\ \wedge\ RecordEvent(record.process,\ record.key,\ record.version) \\
& \qquad\ \wedge\ \text{UNCHANGED}\ \langle reads \rangle
\end{aligned}
$$

Conformance Monitoring

The state invariant checks that the client's history never go back in time

$$
\begin{aligned}
&StateInvariant \triangleq \\
&\quad \lor \land \forall\, c \in \text{DOMAIN } history : \\
&\qquad \land \forall\, k \in \text{DOMAIN } history[c] : \\
&\qquad\quad \land \forall\, r \in \text{DOMAIN } history[c][k] : \\
&\qquad\qquad r > 1 \Rightarrow history[c][k][r] > history[c][k][r-1]
\end{aligned}
$$

$$Alert([msg \mapsto \text{“Invariant was violated”}])$$

Challenges

- Difficult to limit the size of the trace in an infinite stream
- Ordering can be established within a single process
- Must rely on timestamps for ordering across processes
- May work best for client-centric consistency models
- Modern ns-scale clock synchronization protocols (Huygens, DPTP, etc) could help
- Still need a sorting step

What We Learned Along the Way

What We Learned Along the Way

- Generally possible to use TLA+ to check traces against system invariants both offline and online
- Simpler to test local invariants than global invariants in a distributed system
- Not so easy to check traces using original design specs
- Specs still need to be written for trace checking
- Modularity of TLA+ does allow specs to share logic

What We Learned Along the Way

- Still see significant value in trace checking with TLA+
- Significant success in using it to verify API guarantees
- But not internal implementations, which was the goal
- By making it part of our testing infrastructure
  - Detect bugs before they're seen in production
  - Reduce the effort required to debug systems
  - Find ways to generalize trace checking for TLA+

Questions?