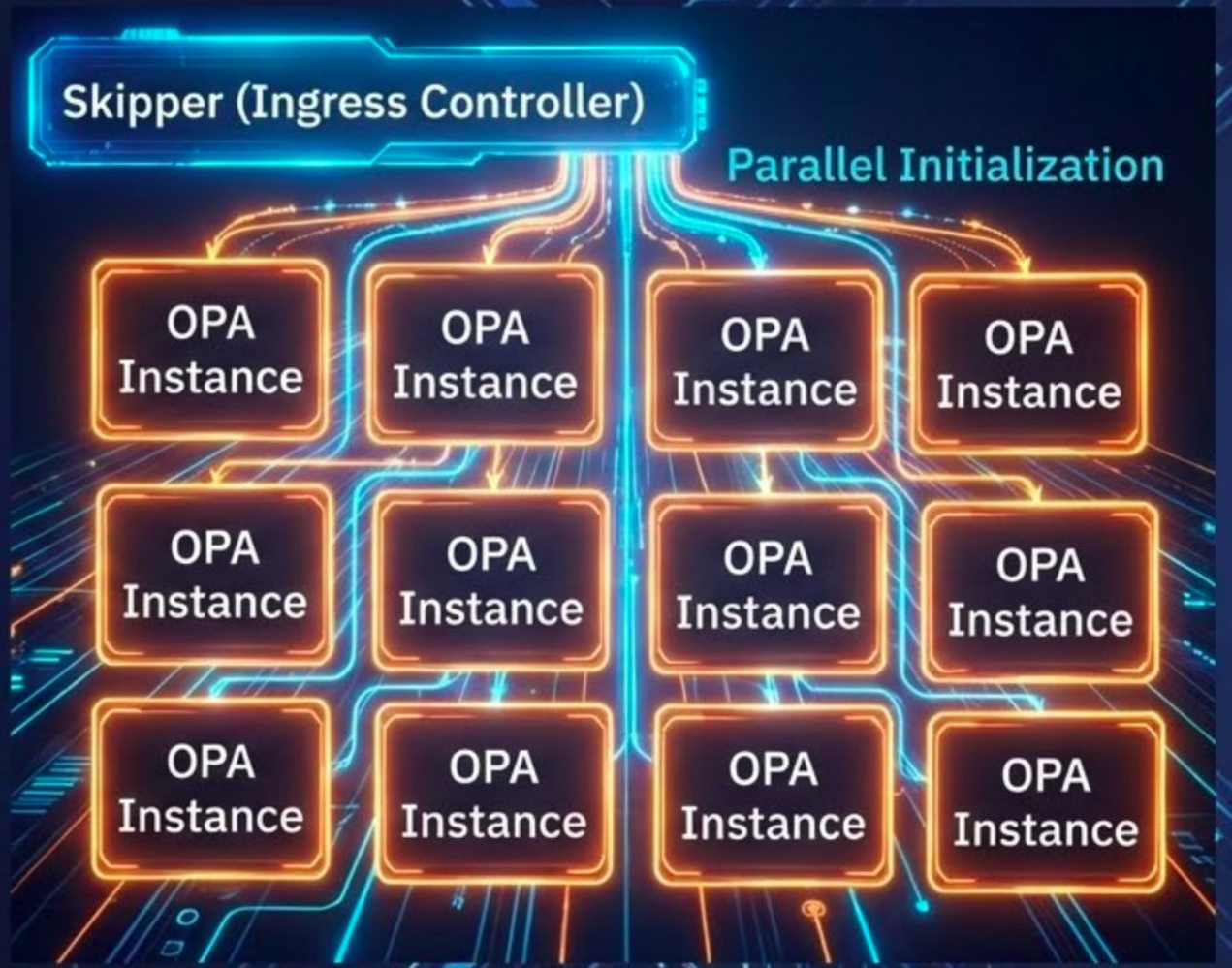
The Case of the Stale Snapshot

Debugging a multithreaded mystery where system lies to itself

Pushpalanka Jayawardhana

The Mission : Achieve Faster Startup

- We rolled out an optimization to Zalando's open-source ingress controller, Skipper
- **The Goal :** Modify the system to start all required Open Policy Agent(OPA) instances in parallel, allowing Skipper to be operational more quickly during large scale deployments
- This change, intended to improve performance, exposed a hidden time-dependent bug in OPA's plugin manager



For further details on setup:
https://engineering.zalando.com/posts/2024/12/open-policy-agent-in-skipper-ingress.html

The Phantom Error : A Perfect Locked-Room Mystery

Consistent, but small error rate

1 Route
Only one failing route

1 Pod
Only one pod out of many

Zero
No error logs despite comprehensive logging

KEY Clue :

Log line that failing OPA instance became healthy, followed by another line where it became unhealthy in a fraction of a second

Ruling out the Usual Suspects

Our first step was to eliminate sources of errors through a process of deduction

Theory 1: A Buggy OPA Policy?

Verdict: Ruled Out. The exact same policy worked perfectly in every other pod.

RULED OUT

Theory 2: A Corrupted Bundle Download?

Verdict: Ruled Out. Our logging confirmed that all bundles were downloaded and parsed successfully, with no errors reported.

RULED OUT

Conclusion: With the simple explanations exhausted, we knew we were dealing with a deeper, more elusive issue.

Forcing the Bug Out of Hiding

- Despite many different approaches, issue didn't get reproduced with locally with a similar setup as of stakeholders.
- **The breakthrough**: I pushed one variable beyond stakeholder's setup, the scale

Local Setup: **OK**

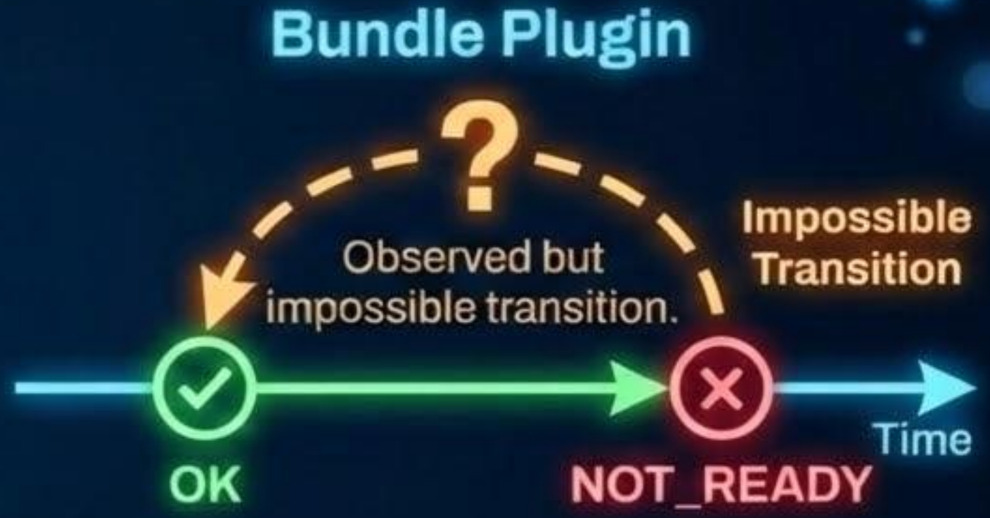
Local Setup @ Scale: **Reproduced!**

> "I kept on increasing the number of bundles in my local setup... as I go beyond 50 OPA instances, the mysterious issue start to happen consistently."

A Clue that Defies Logic

With the issue reproduced, more structured logs became my eyes inside the concurrency maze. Increased logging revealed a critical but confusing event:

- The bundle plugin was flipping its state from OK back to NOT_READY.
- This state change directly caused the brief healthy → unhealthy transition we had observed.



The Central Question:

The mystery deepened: There was no code path that could explain a transition from OK back to NOT_READY. How could our listener observe a state change that never actually happened in the code?

The Smoking Gun: One Log Line Reveals the Lie

I added one log line to print the plugin statuses received by the listeners, along with the statuses at the source of truth.

THE EVIDENCE

```
"UpdatePluginStatus:
  plugin status listener map[bundle:{NOT_READY} ...]
  status manager map[bundle:{OK} ...]
} ...]"
```

THE LIE
(Stale Snapshot)

THE TRUTH
(Current State)

The Revelation:

The log proved it. The listener was receiving a stale snapshot where the bundle was NOT_READY, even though the manager's internal state was already OK. The notification was a message from the past.

Reconstructing the Crime: An Event Ordering Race

- The root cause was a subtle race condition in how OPA's manager notified listeners.

1. A goroutine prepares a status snapshot (NOT_READY).
2. Before it can notify, it gets pre-empted by the OS.
3. Another goroutine updates the state to OK, prepares a new snapshot, and successfully notifies the listener.
4. The original, pre-empted goroutine wakes up and delivers its stale NOT_READY snapshot, overwriting the correct state.

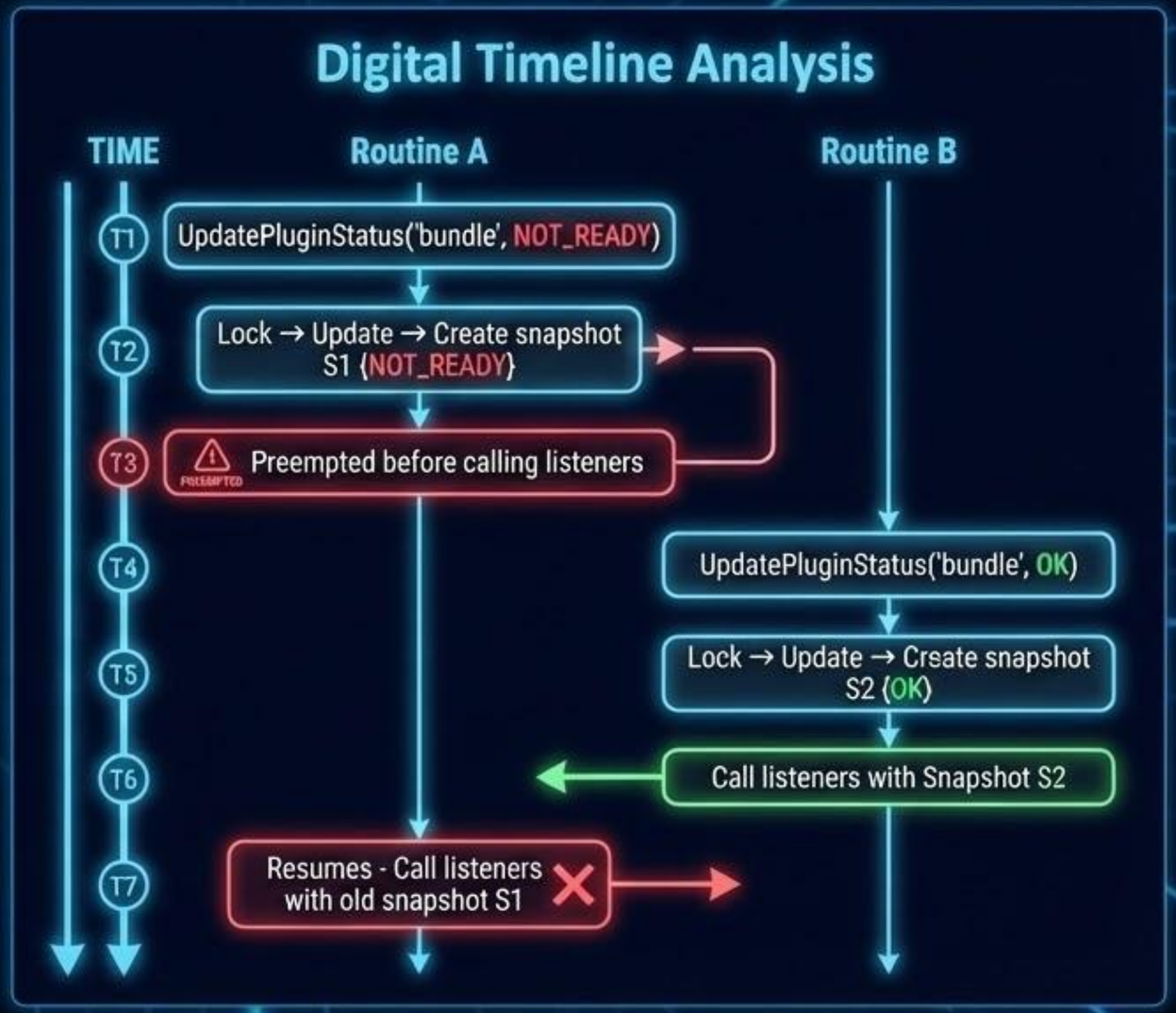


Exhibit A: The Unprotected Gap

How the CPU scheduler hijacked the execution order.

```
// UpdatePluginStatus updates a named plugins status. Any registered
// listeners will be called with a copy of the new state of all
// plugins.
func (m *Manager) UpdatePluginStatus(pluginName string, status *Status) {
 var toNotify map[string]StatusListener
 var statuses map[string]*Status

 func() {
  m.mtx.Lock()
  defer m.mtx.Unlock()
  m.pluginStatus[pluginName] = status
  toNotify = make(map[string]StatusListener, len(m.pluginStatusListeners))
  maps.Copy(toNotify, m.pluginStatusListeners)
  statuses = m.copyPluginStatus()
 }()

    // 🔴 Lock released here — listeners called concurrently
 for _, l := range toNotify {
  l(statuses)
 }
}
```

Closing the Case: The One-Line Fix

Since notification system was unreliable under CPU contention, the fix was to bypass it. Instead of trusting the passed in snapshot, we query the manager directly for the current state.

BEFORE: TRUSTING THE SNAPSHOT (Risky)

```
// manager.RegisterPluginStatusListener("...",
func(status map[string]*plugins.Status) {
    opa.healthy.Store(allPluginsReady(status, ...))
})
```

AFTER: QUERYING THE SOURCE OF TRUTH (Fixed)

```
// manager.RegisterPluginStatusListener("...",
func(_ map[string]*plugins.Status) {
    // Get fresh status to workaround OPA issue #8009
    status := opa.manager.PluginStatus()
    opa.healthy.Store(allPluginsReady(status, ...))
})
```

Always get the latest state directly.

Why It Works: We always read the real-time state from the source of truth (the manager), guaranteeing we never act on stale data delivered out of order.

The Detective's Notebook : Lessons from the Case

Learning 1 : Logs Beat Debuggers to Troubleshoot Concurrency Issues

You can't step-through race conditions in an IDE. When an issue is random and concurrency is suspected, structured logging is more effective.

Pro Tip: To compensate for losing interactive stack traces, print stack traces directly in logs at critical state transitions.

Learning 2 : Strategic Logging Over Verbose Logging

Good logging saves time, but more is not always better. Log key state transitions with context, not every single operation.

Guiding Question: "Will this log help me confirm or rule out something critical?"

Learning 3 : Race Detectors Don't See the Whole Picture

Go's race detector is excellent at finding unsynchronized memory access (data races). It did not detect this event ordering race, as no memory was accessed improperly.

Key Insight: Multi-threading bugs are often about event ordering and state consistency, which require manual reasoning.

The Detective's Notebook : Lessons from the Case Ctd.

Learning 4: Question the Assumptions Explicitly 🔍

False Assumption	Reality
NOT_READY means the plugin is broken.	The real state was OK; the snapshot was stale.

Learning 5: AI is a productivity tool, Not a Debugger (Yet) 🤖

AI models excel at accelerating the process: generating log parsing commands, documenting findings, or explaining unfamiliar code.
The actual debugging—forming hypotheses and finding the root cause—still requires human intuition and expertise.

Key Insight:
AI is excellent for generating code and explaining concepts, but human reasoning is still essential for complex debugging and root cause analysis.

Learning 6: Every Bug is Reproducible (With enough persistence) ⏱️

Reproducibility isn't binary; it's about finding the right conditions. By systematically varying parameters (like scale) and investing consistent effort, we found a way to trigger the bug reliably.

Takeaway:
Don't give up on "flaky" tests when the stakes are high. Systematically isolate variables and invest to find the specific conditions that make a bug reproducible

Beyond Data Races : The Hidden Dangers of Information Propagation

The Lesson from the Bug

Complex multi-threading issues are not always about locks and data races. They can be about logical races in information propagation and timing assumptions.

Final Thought

Understanding how information flows through the system is as critical as protecting access to shared memory.

Case File : Primary Sources & Evidence

For a deeper technical dive, the original issues and pull requests are available for review

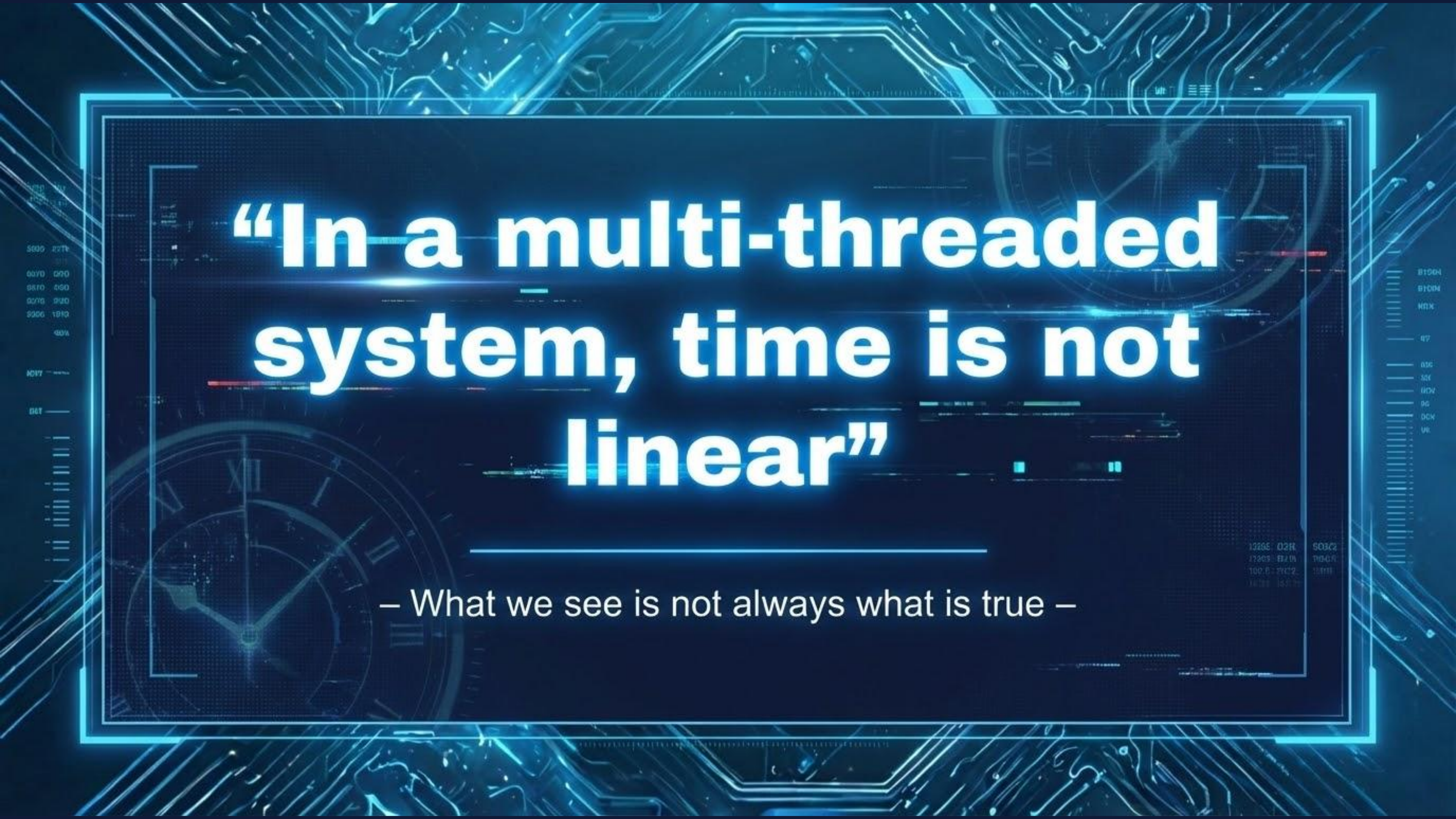
OPA Issue Report: github.com/open-policy-agent/opa/issues/8009

The Exposing Feature in Skipper: PR #3562 (Preloading OPA instances in parallel)

The Workaround in Skipper: PR #3692

The Original Article: ‘Beyond Race Detectors’ by Pushpalanka Jayawardhana

OPA Integration Details: Zalando Tech Blog : ‘OPA Integration in Skipper’


“In a multi-threaded system, time is not linear”
– What we see is not always what is true –

Thank you!