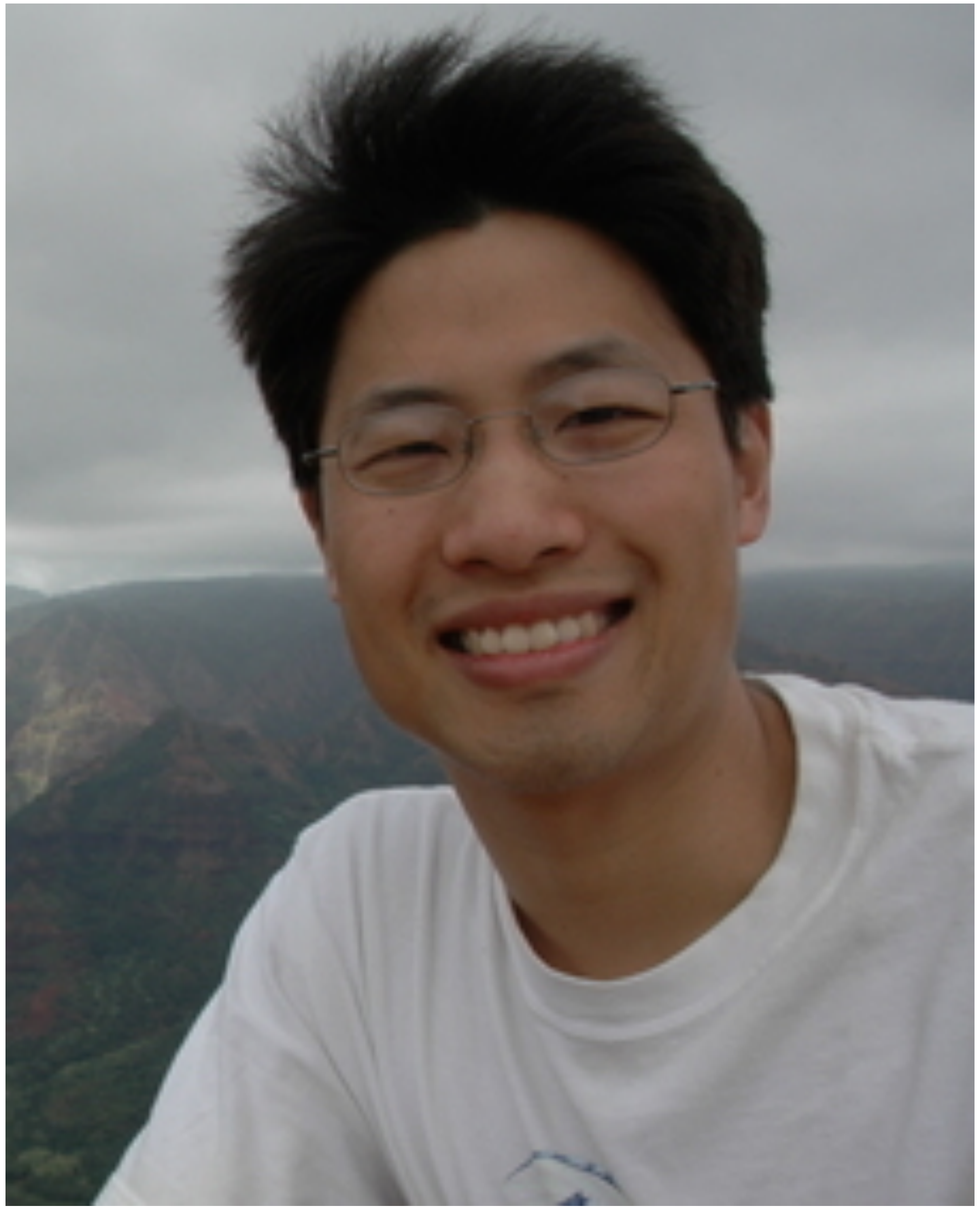
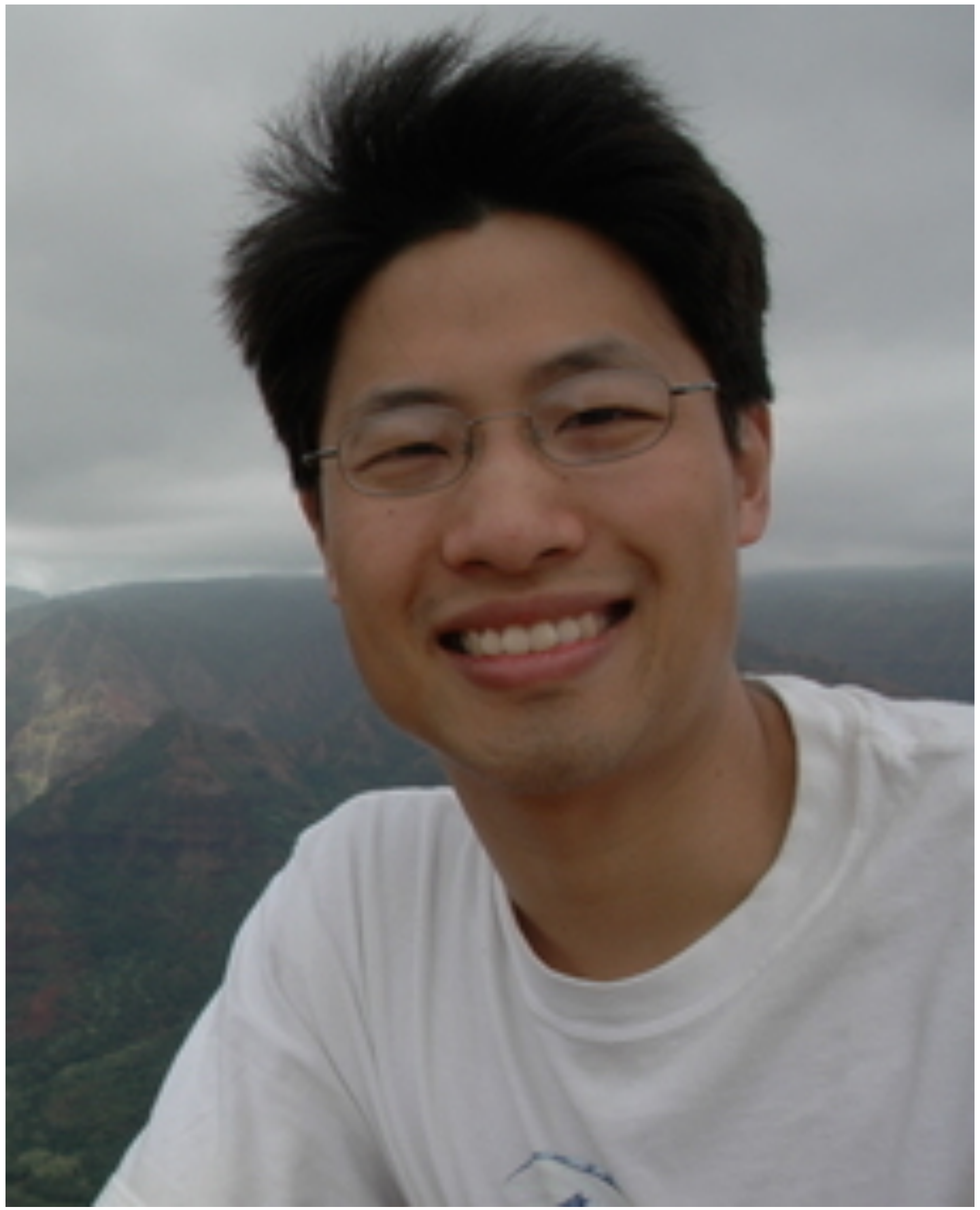
Minimizing Faulty Executions of Distributed Systems

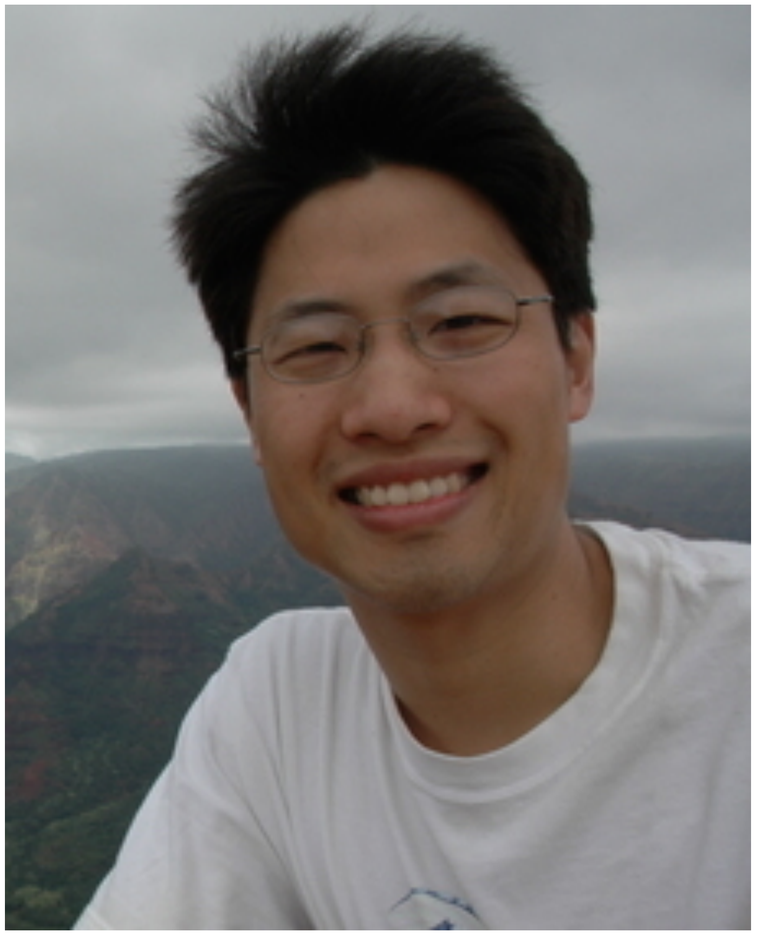
Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George Necula,
Arvind Krishnamurthy, Scott Shenker



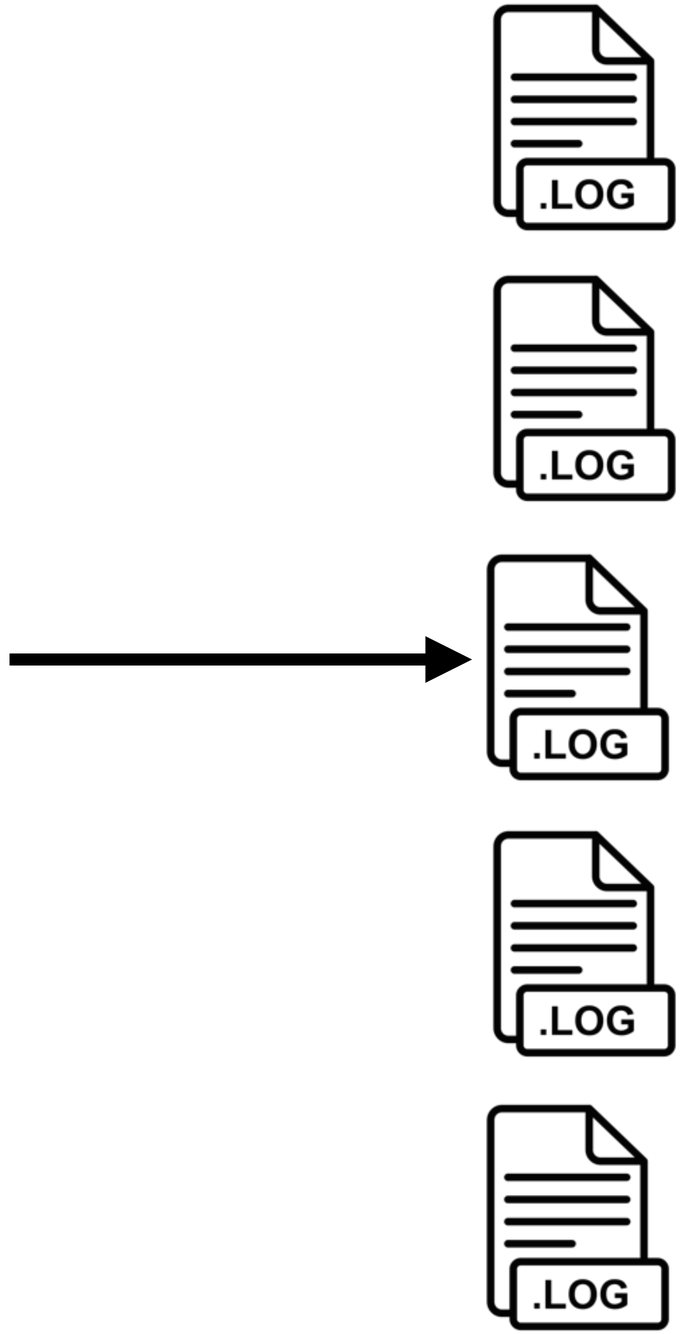
Alex Yip, SE at Nicira

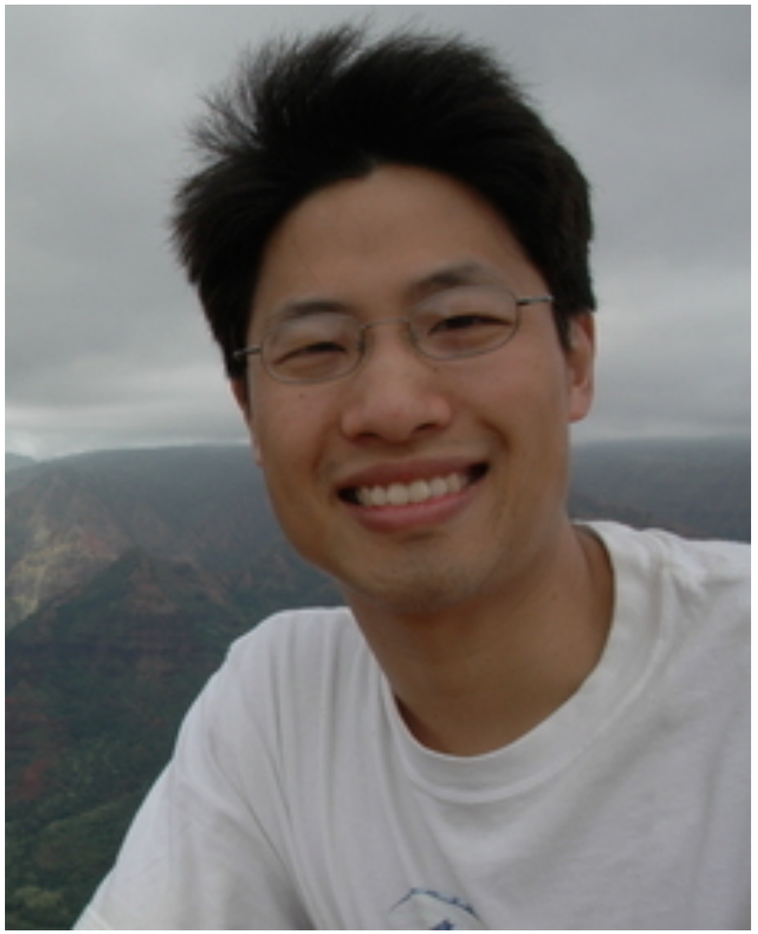


Alex Yip, ~~SE~~ at Nicira
Chief Debugger

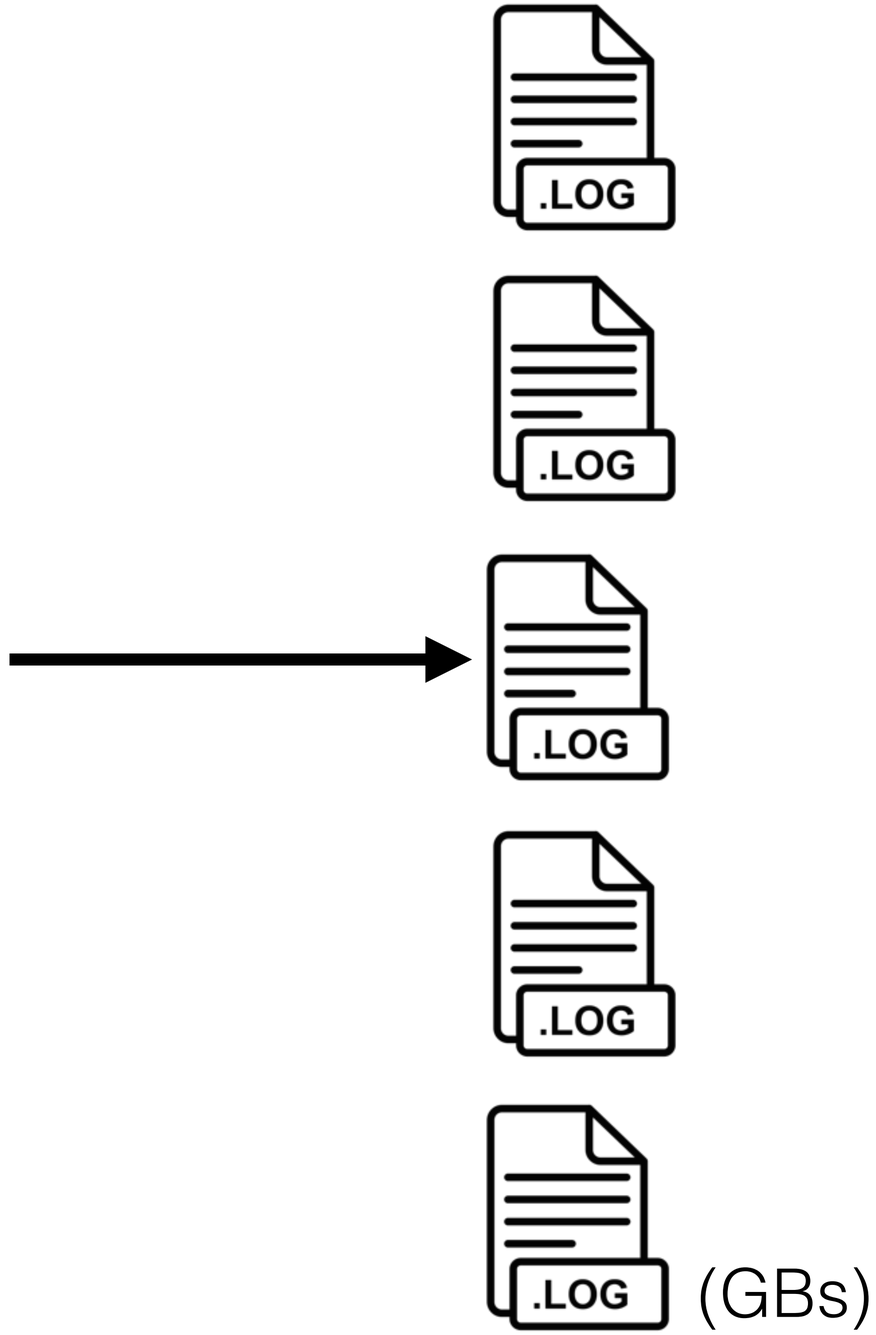


Alex Yip, ~~SE~~ at Nicira
Chief Debugger

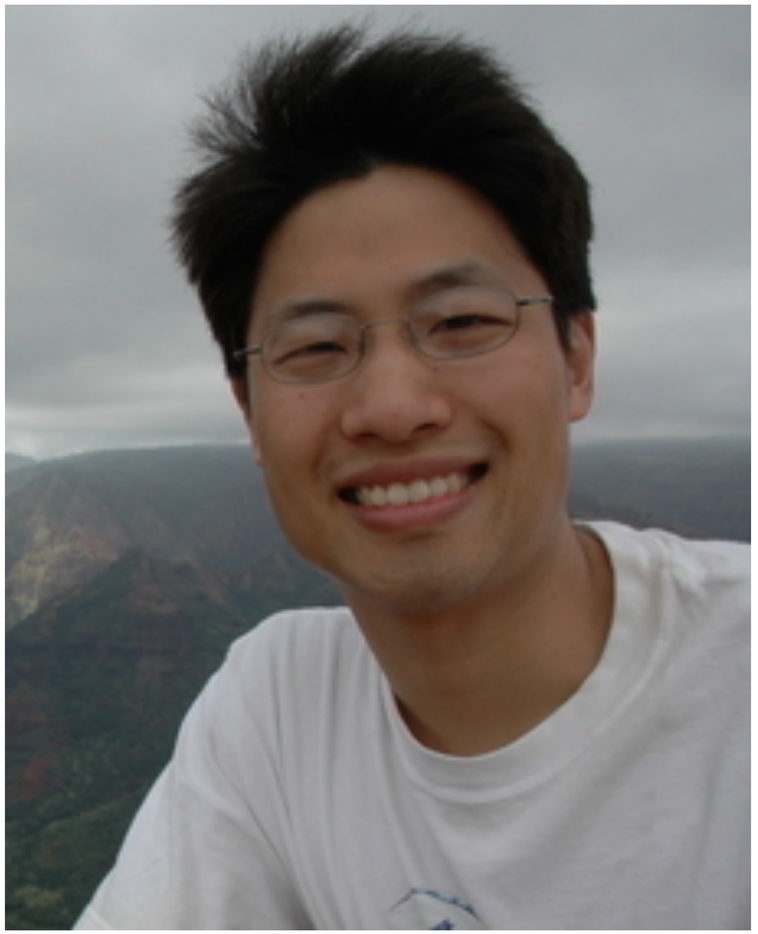




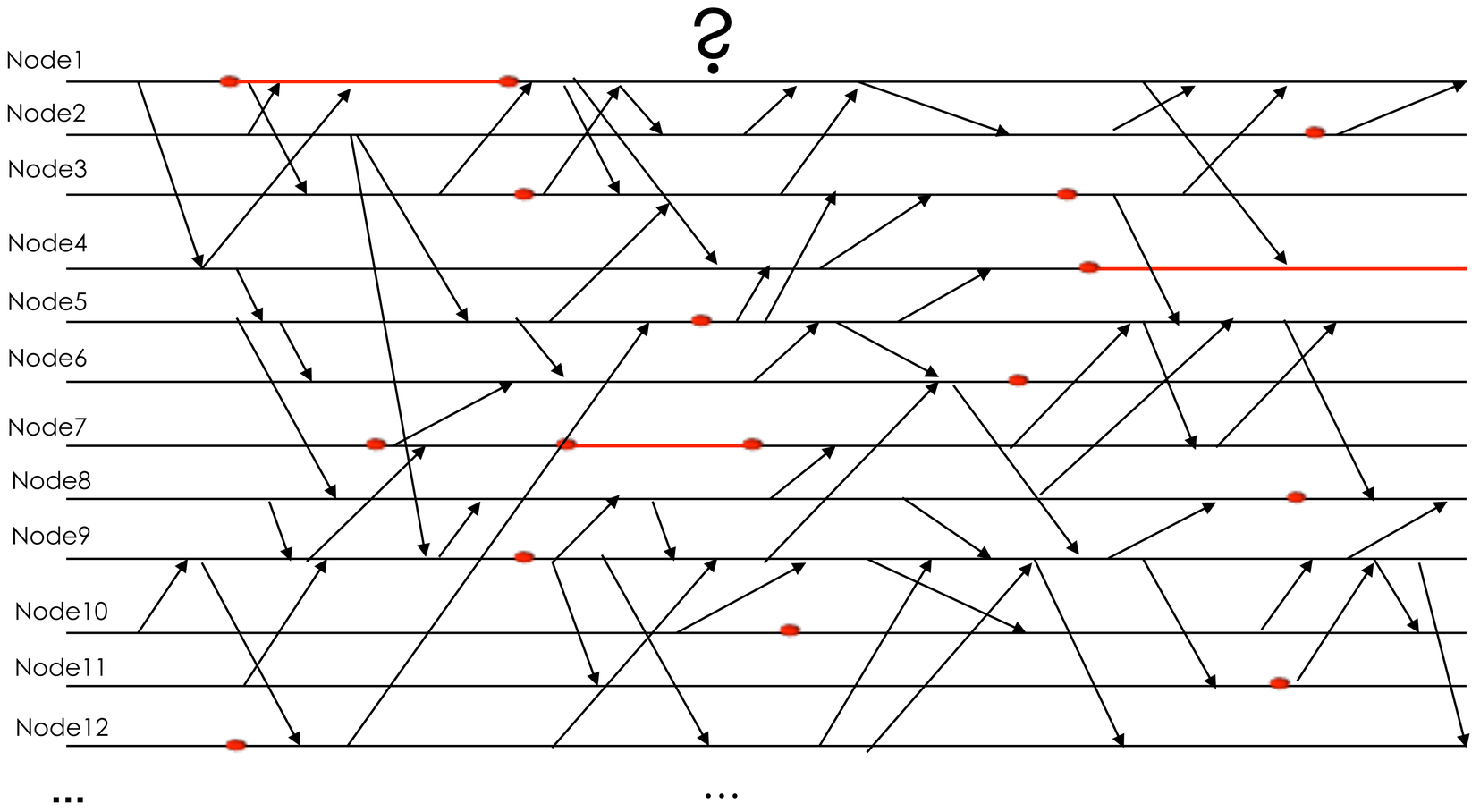
Alex Yip, ~~SE~~ at Nicira
Chief Debugger

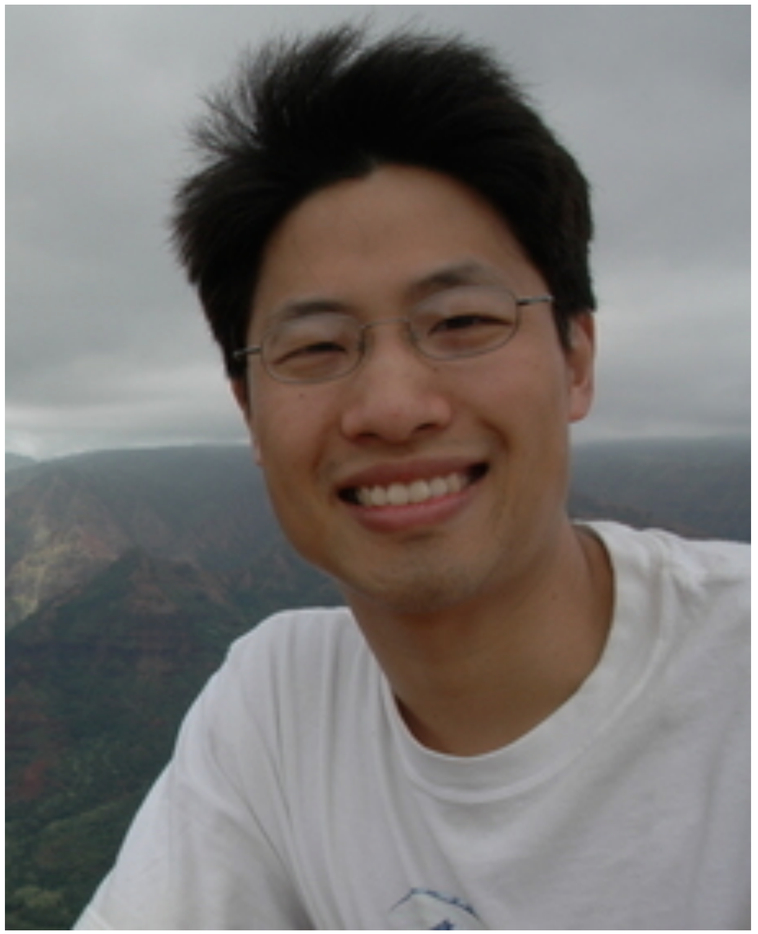


(GBs)

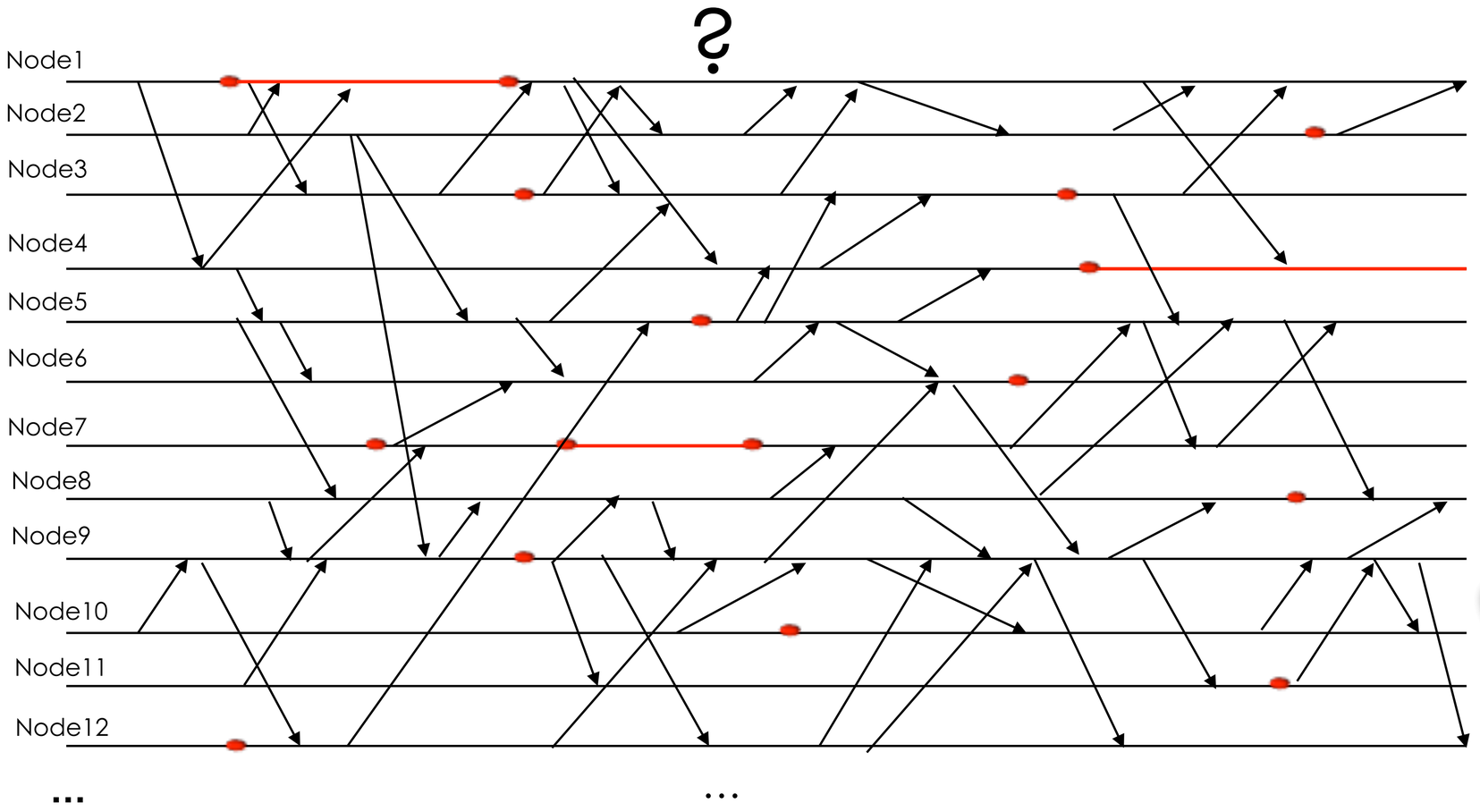


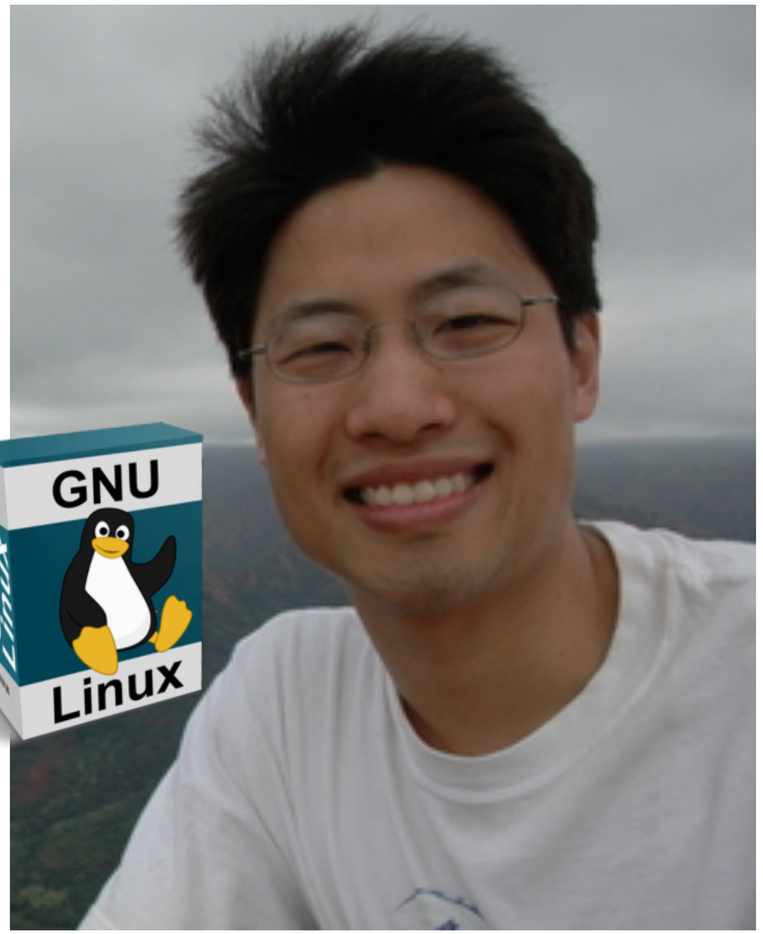
Alex Yip, ~~SE~~ at Nicira
Chief Debugger





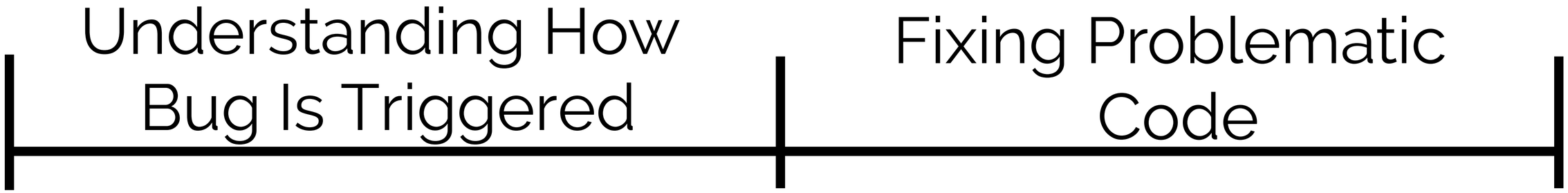
Alex Yip, ~~SE~~ at Nicira
Chief Debugger





Alex Yip, ~~SE~~ at Nicira
Chief Debugger

~49% of developers' time spent on debugging![1]



[1] LaToza, Venolia, DeLine, ICSE' 06

Our Goal

Allow Developers To Focus on Fixing the Underlying Bug

Problem Statement

Identify a **minimal** sequence of events that triggers the bug

Why Minimization?

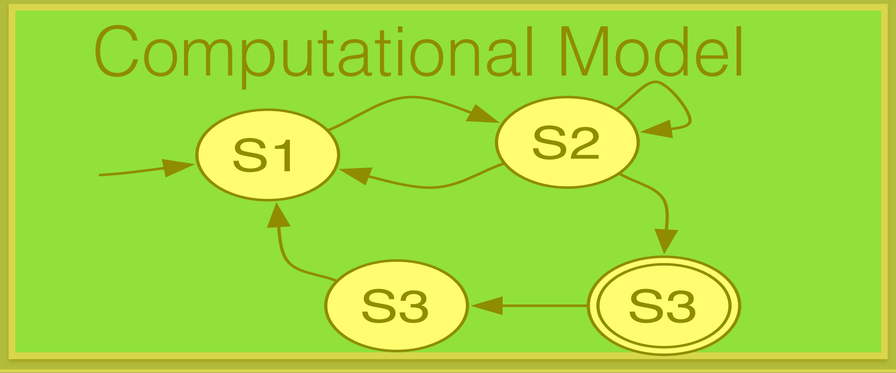
Smaller event traces are easier to understand

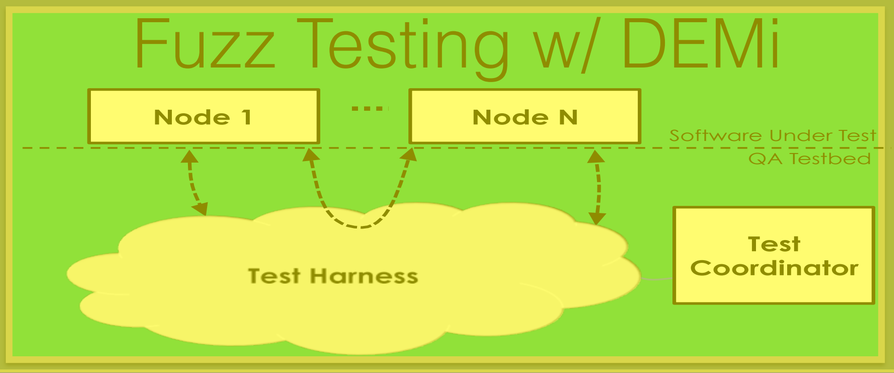
G. A. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. Psychological Review ’56.

Outline

Introduction

Background



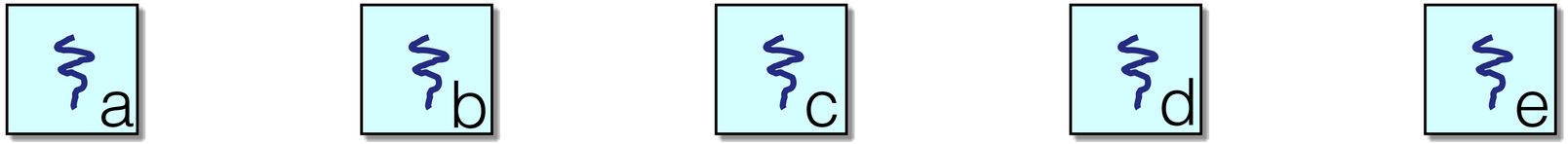


Minimization

Evaluation

Conclusion

Computational Model

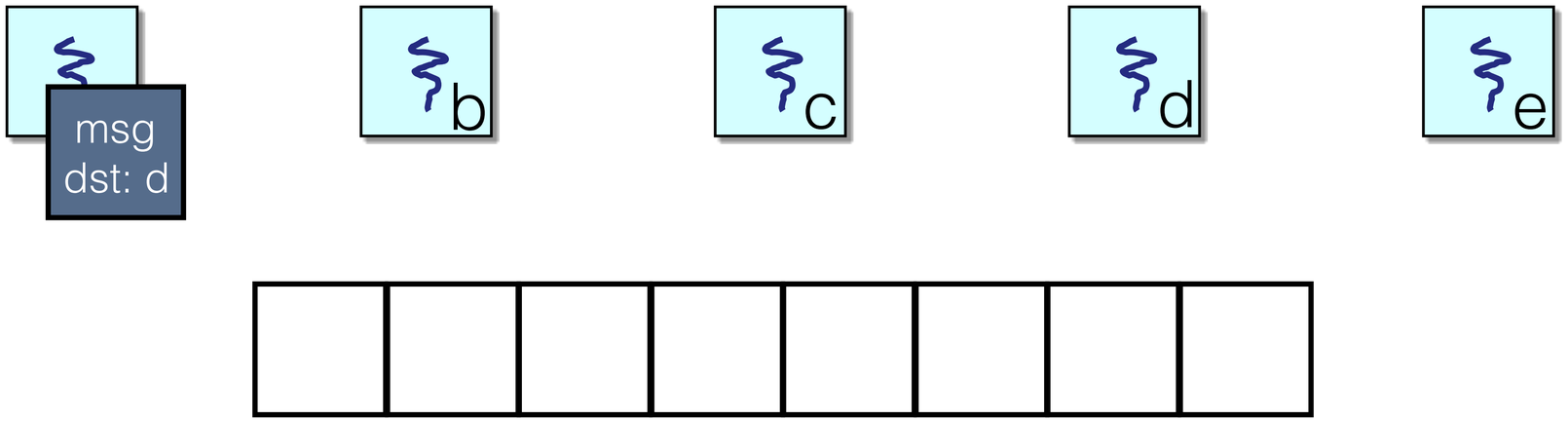


Distributed System: Collection of N processes

Each process p:

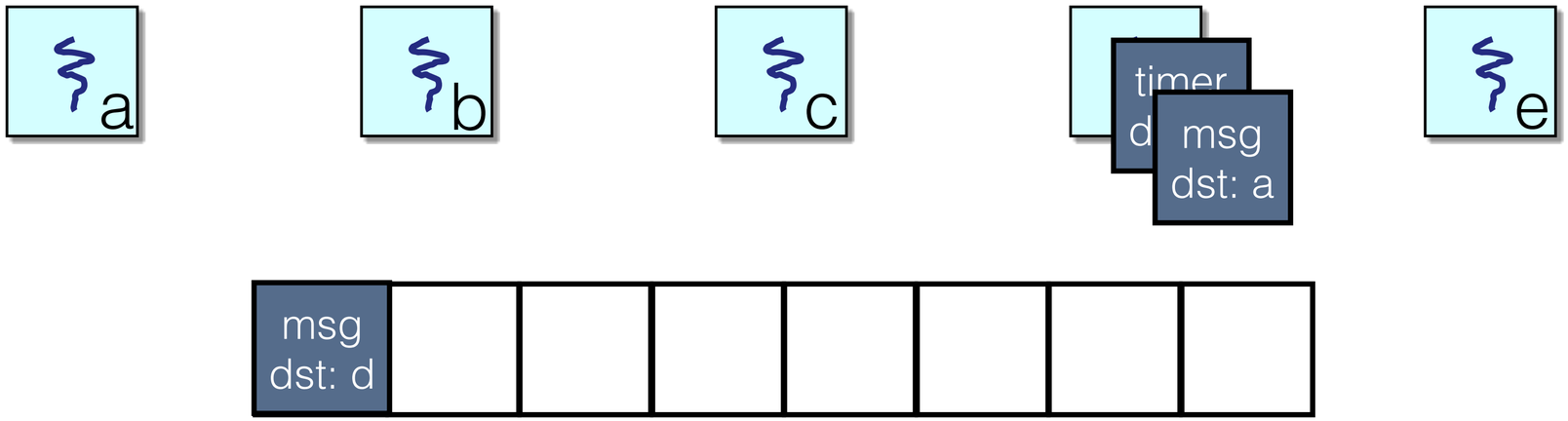
- Has unbounded memory
- Starts in a known initial state
- Changes states deterministically

Computational Model



The network maintains a buffer of sent but not yet delivered messages

Computational Model

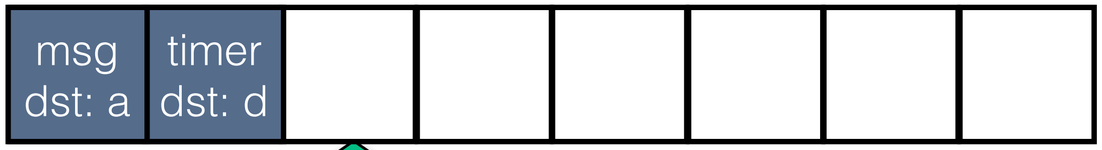


Message deliveries occur one at a time:

- destination enters a new state according to old state & message
- destination sends a finite set of messages to other processes*

*May include timer messages to be delivered to itself later

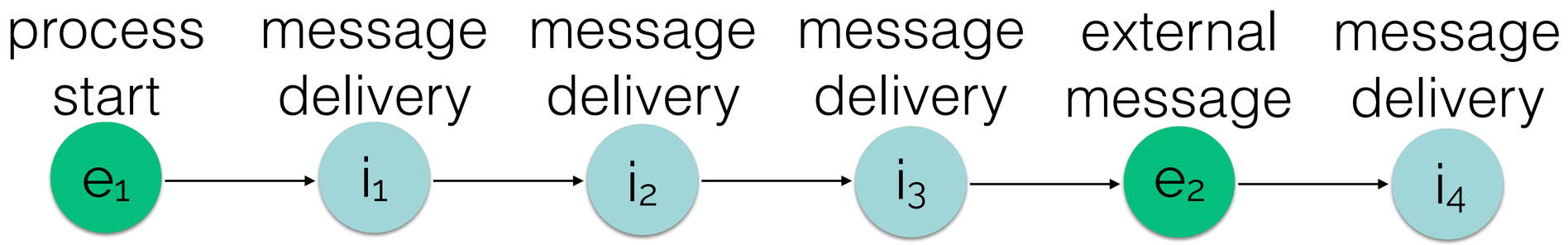
Computational Model



Steps may also be *external:*

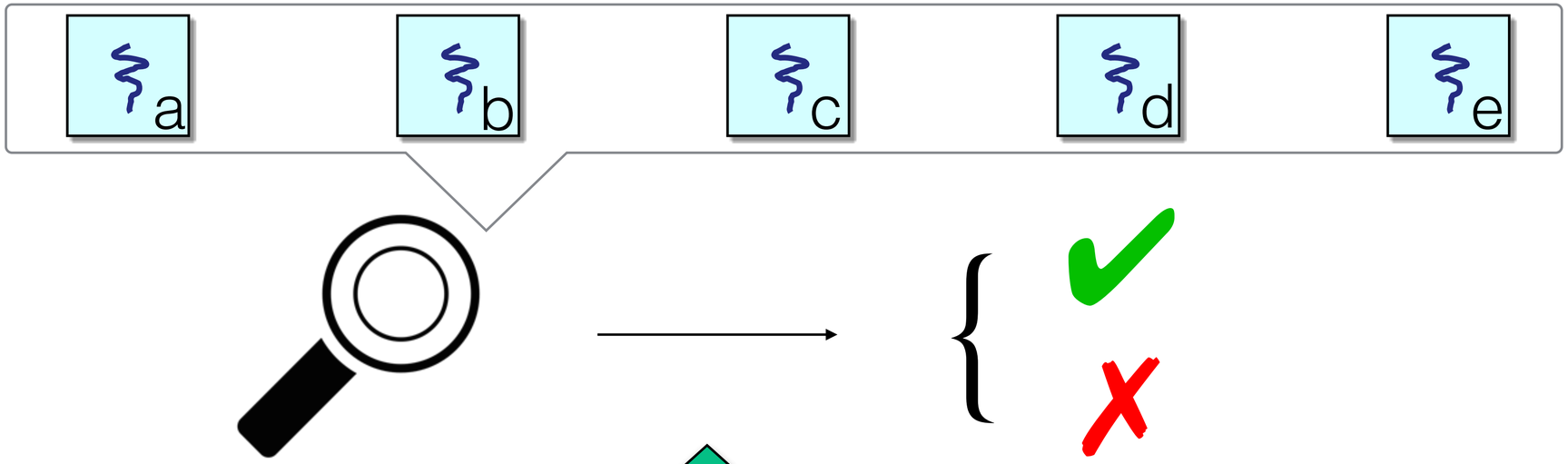
- External message is sent
- Process is created
- Process crash-recovers

Computational Model



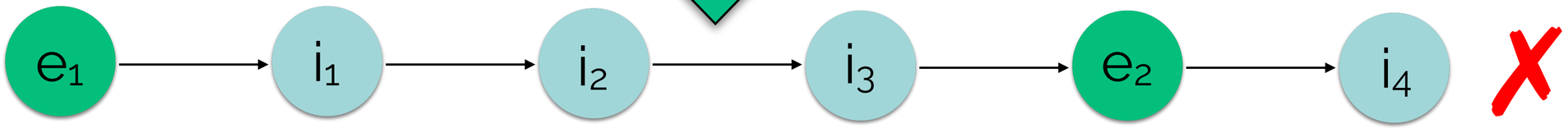
A schedule τ is a sequence of events (either external or internal message deliveries) that can be applied in turn starting from the initial configuration.

Invariant Checking



An invariant is a predicate *P* over the state of all processes.

A faulty execution is one that ends in an invariant violation.



Formal Problem Statement

Given: schedule τ that results in violation of P

Find: schedule τ' s.t.

- τ' violates P, $|\tau'| \leq |\tau|$
- τ' contains a subsequence of the external events of τ
- if we remove any single external event e from τ',
- $\neg\exists\ \tau''$ containing same external events - e, s.t. τ'' violates P

Formal Problem Statement

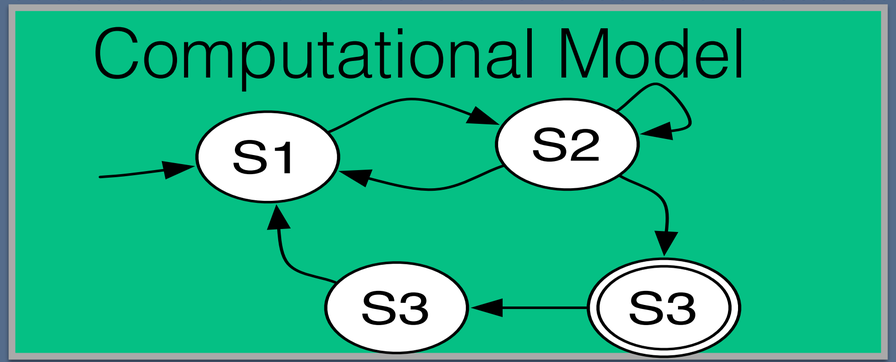
After finding τ':

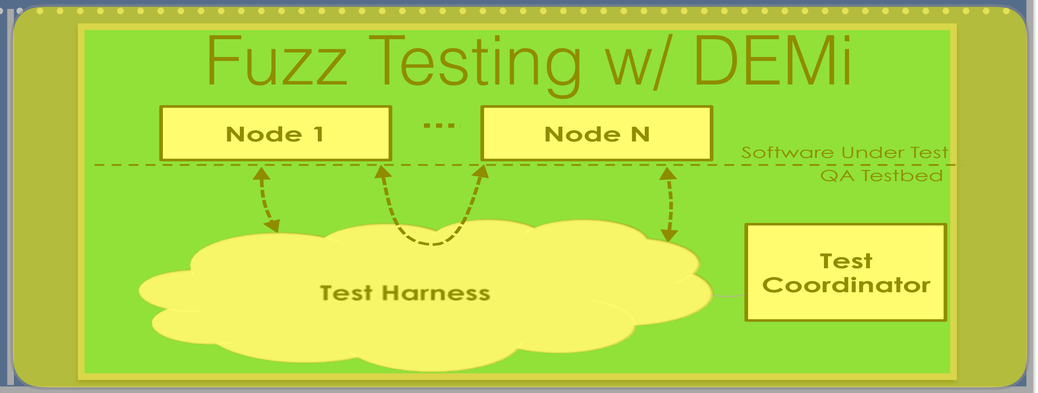
- remove extraneous message deliveries from τ'

Outline

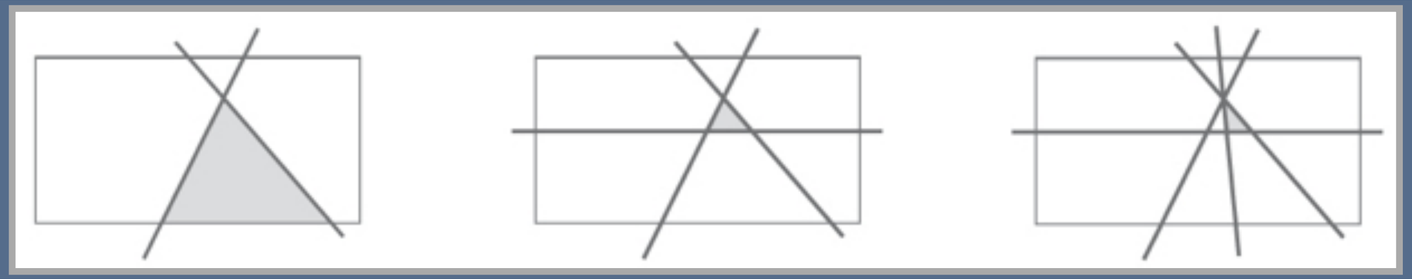
Introduction

Background

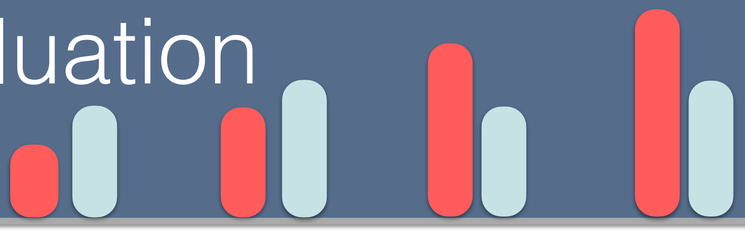




Minimization

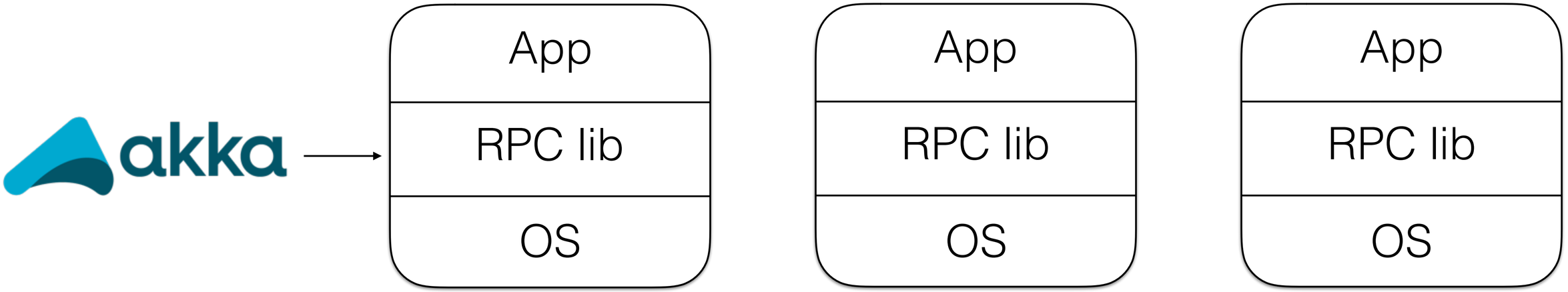


Evaluation

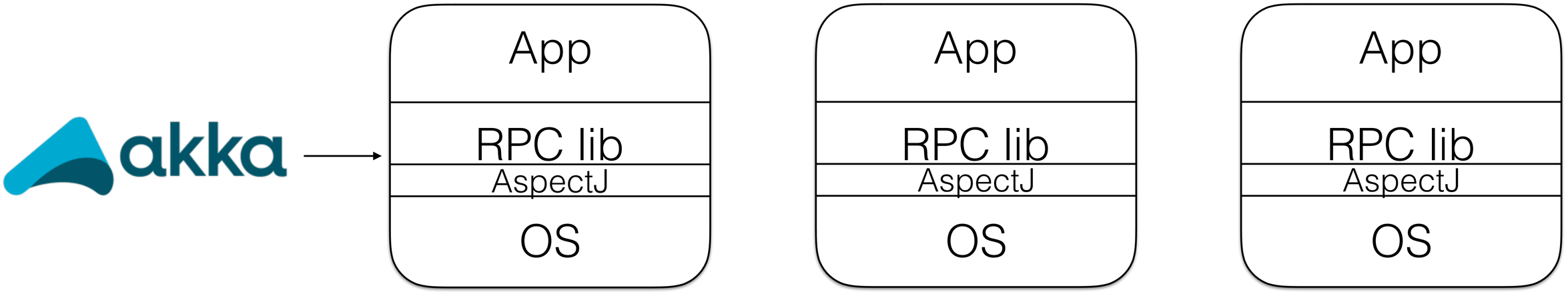


Conclusion

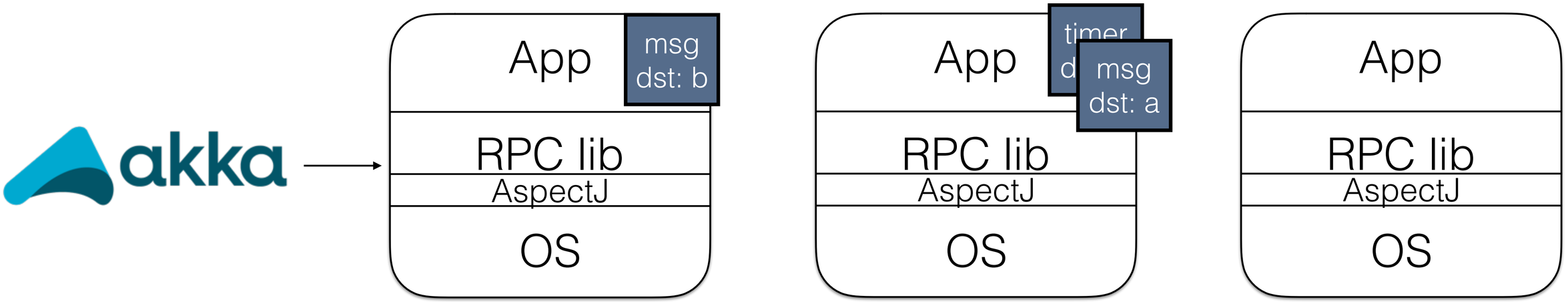
Fuzz Testing with DEMi

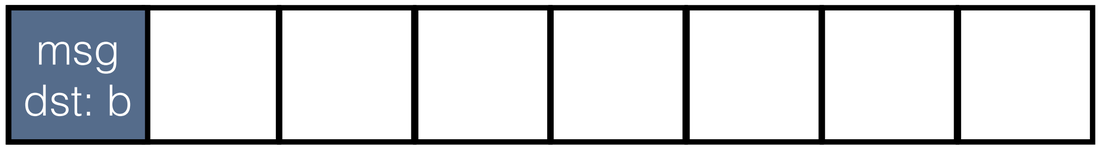


Fuzz Testing with DEMi

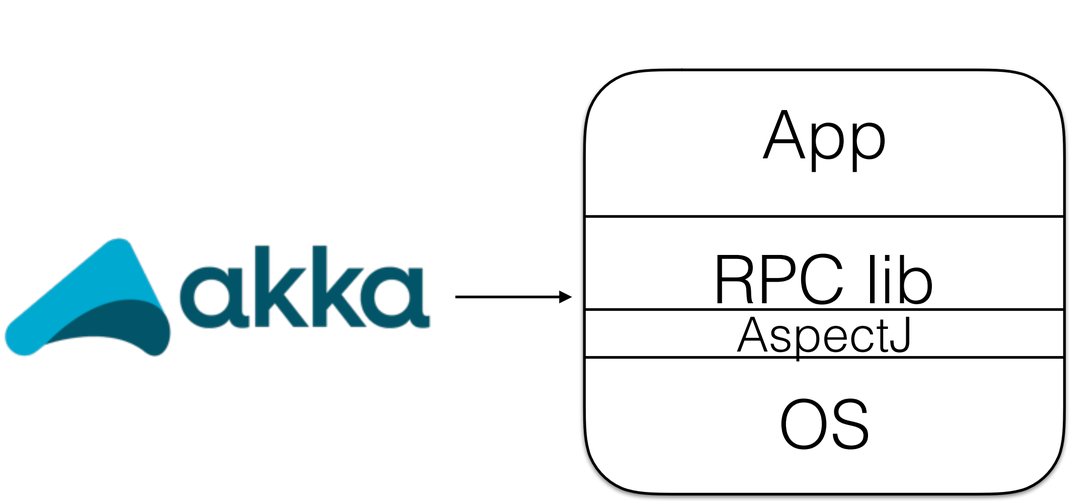


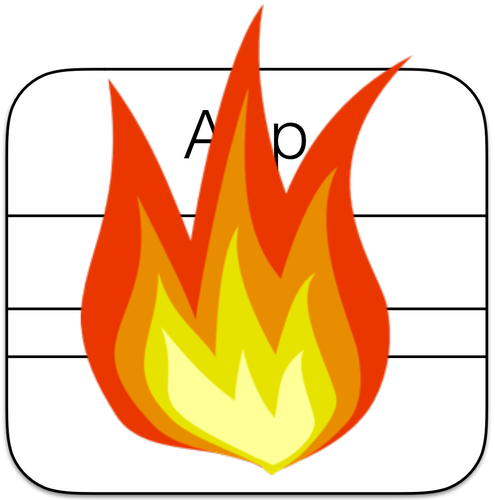
Fuzz Testing with DEMi

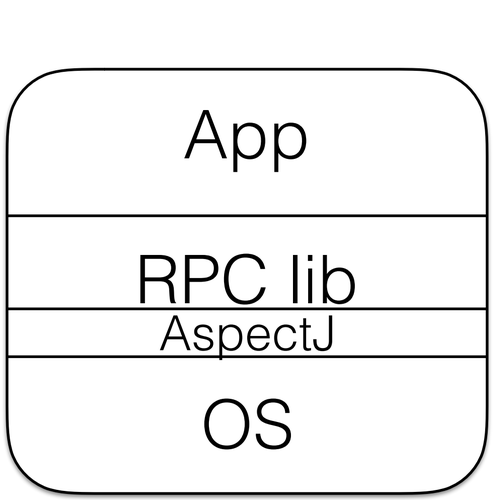


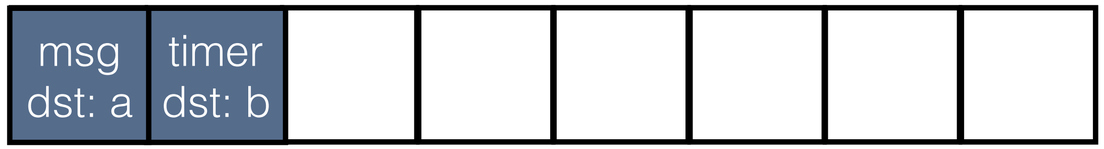


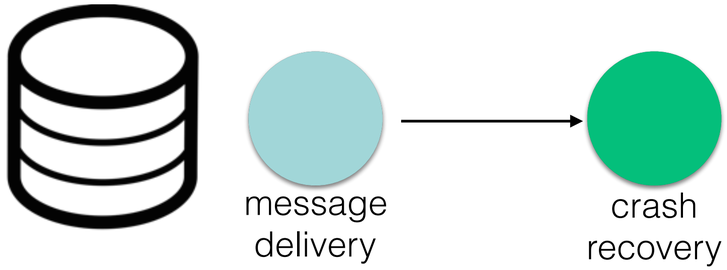
Fuzz Testing with DEMi



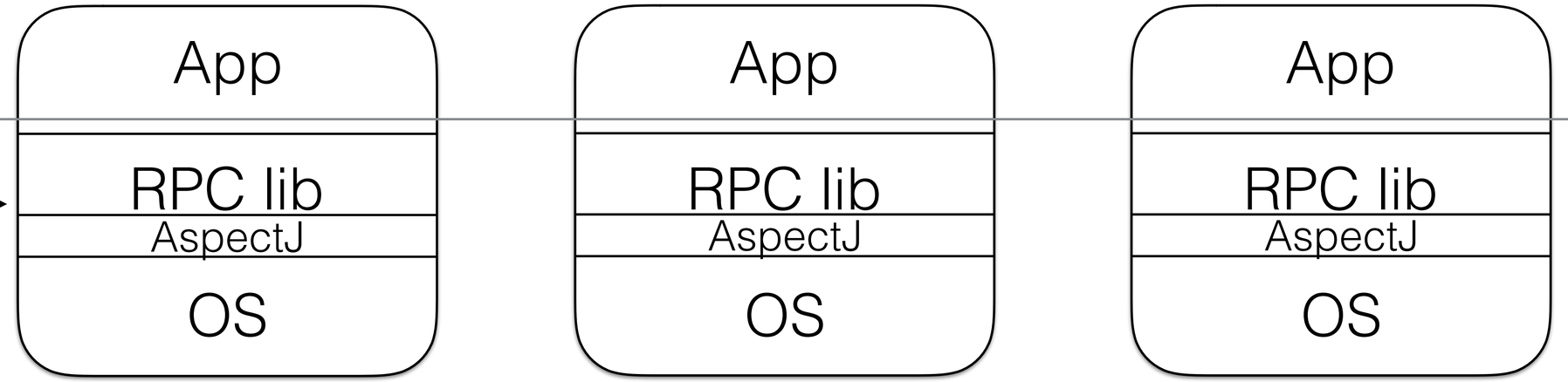


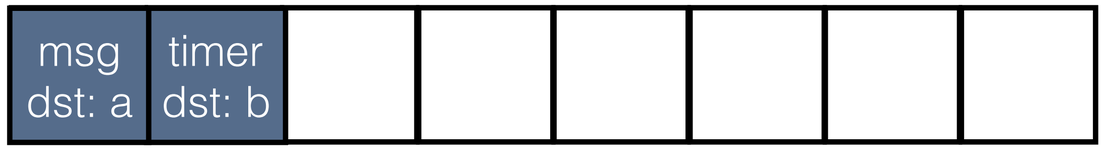


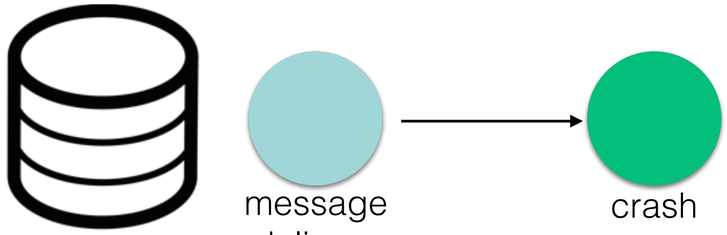




Fuzz Testing with DEMi



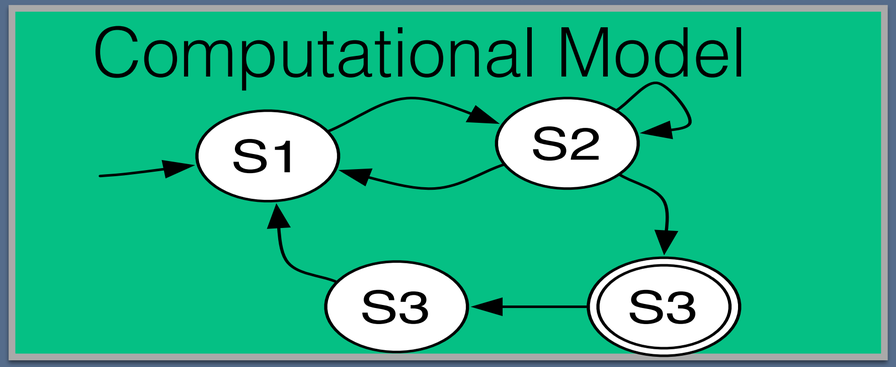


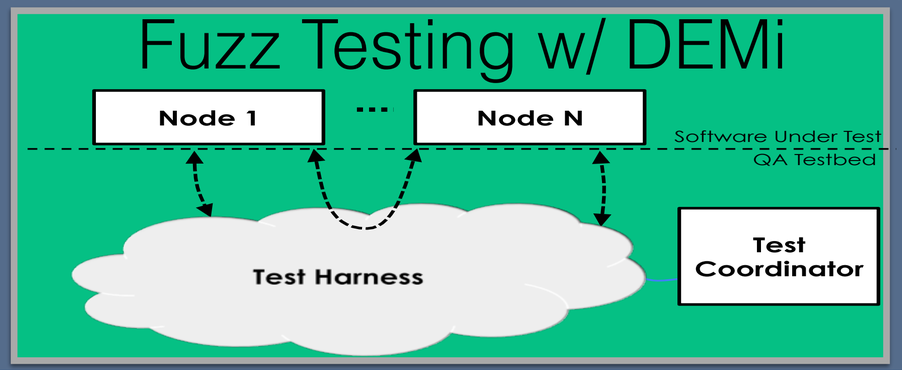


Outline

Introduction

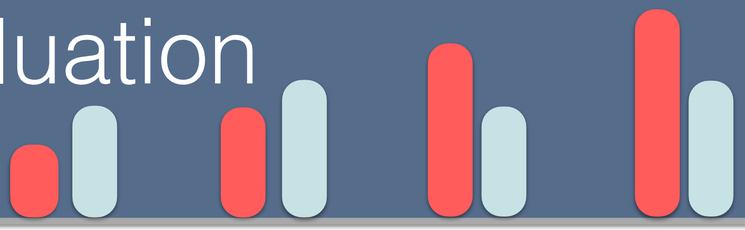
Background





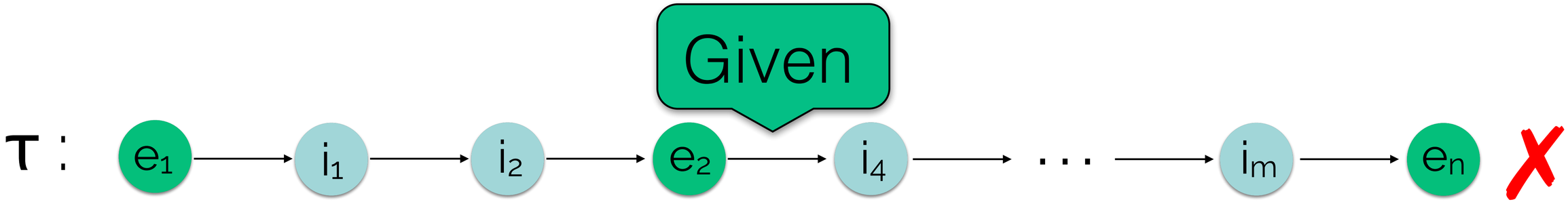
Minimization

Evaluation



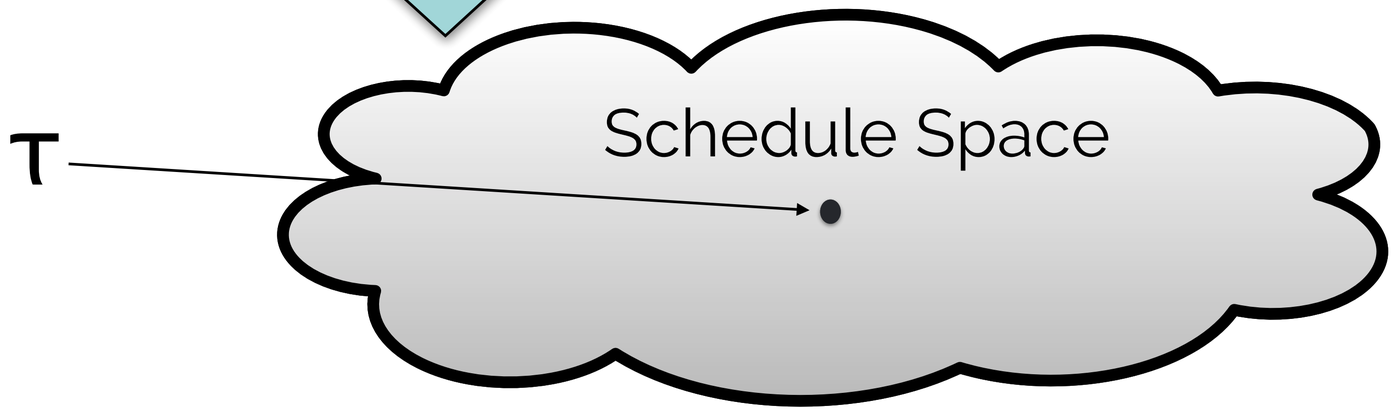
Conclusion

Minimization



Straightforward approach:

- Enumerate all schedules $|\tau'| \leq |\tau|$,
- Pick shortest prefix that reproduces ✗



$$O(n!)$$

Observation #1: many schedules are commutative

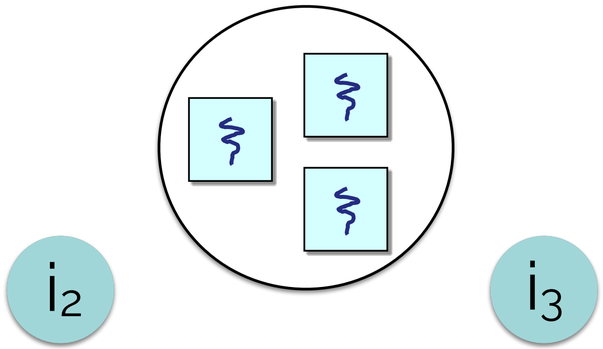
i_2

i_3

- $i_2 \nrightarrow i_3$
- $i_3 \nrightarrow i_2$
- $dst(i_2) \neq dst(i_3)$

Observation #1: many schedules are commutative

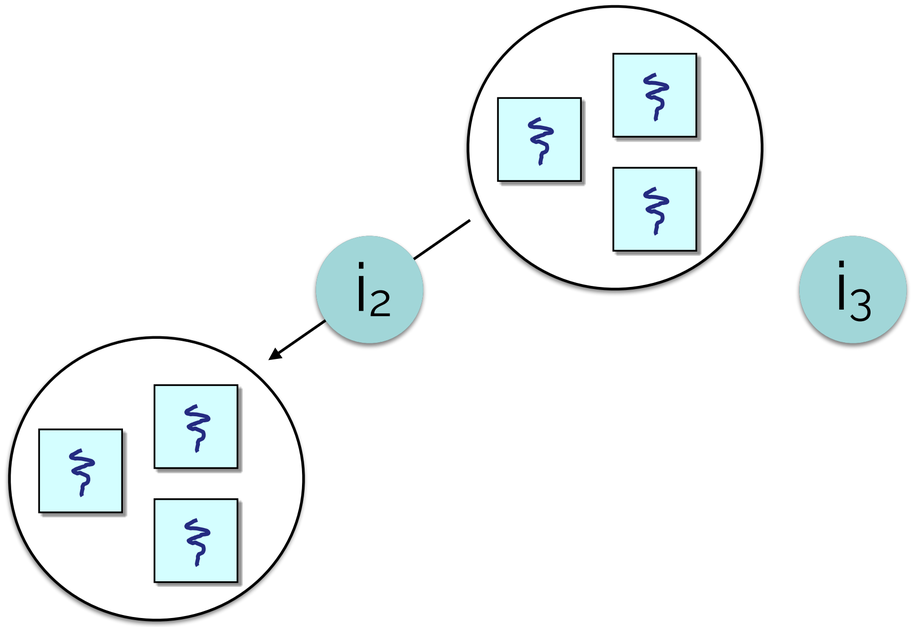
Step n:



- $i_2 \nrightarrow i_3$
- $i_3 \nrightarrow i_2$
- $dst(i_2) \neq dst(i_3)$

Observation #1: many schedules are commutative

Step n:



Step n+1:

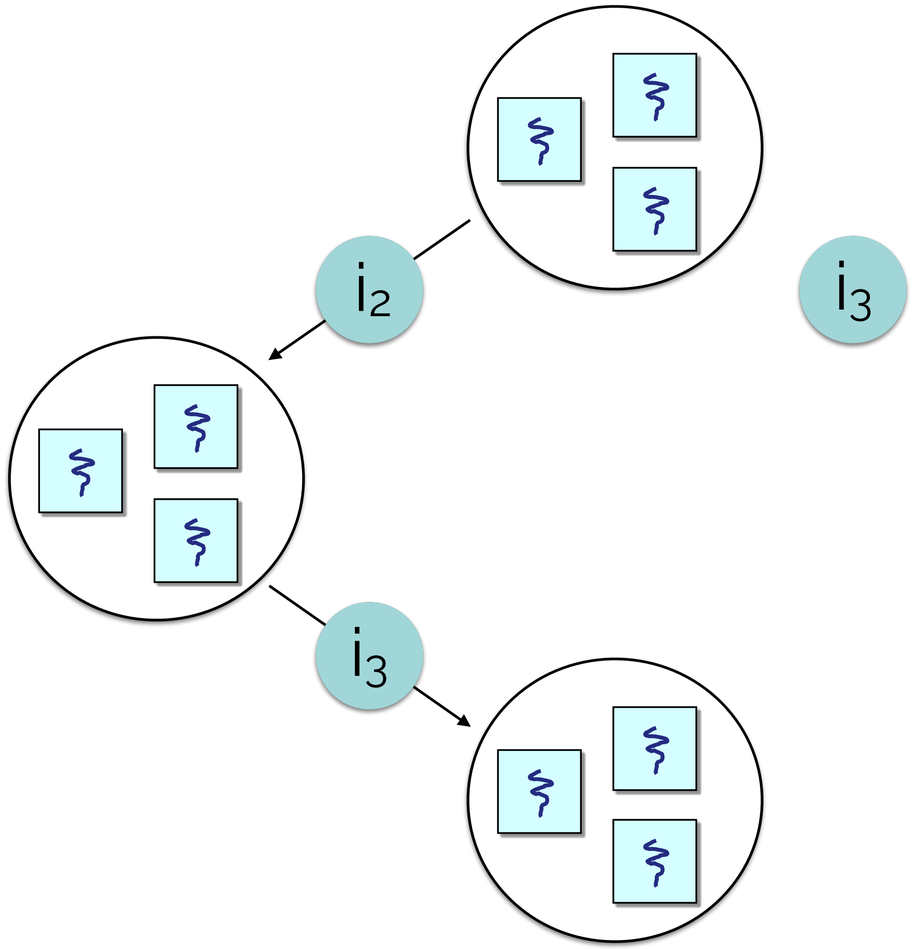
- $i_2 \nrightarrow i_3$
- $i_3 \nrightarrow i_2$
- $dst(i_2) \neq dst(i_3)$

Observation #1: many schedules are commutative

Step n:

Step n+1:

Step n+2:



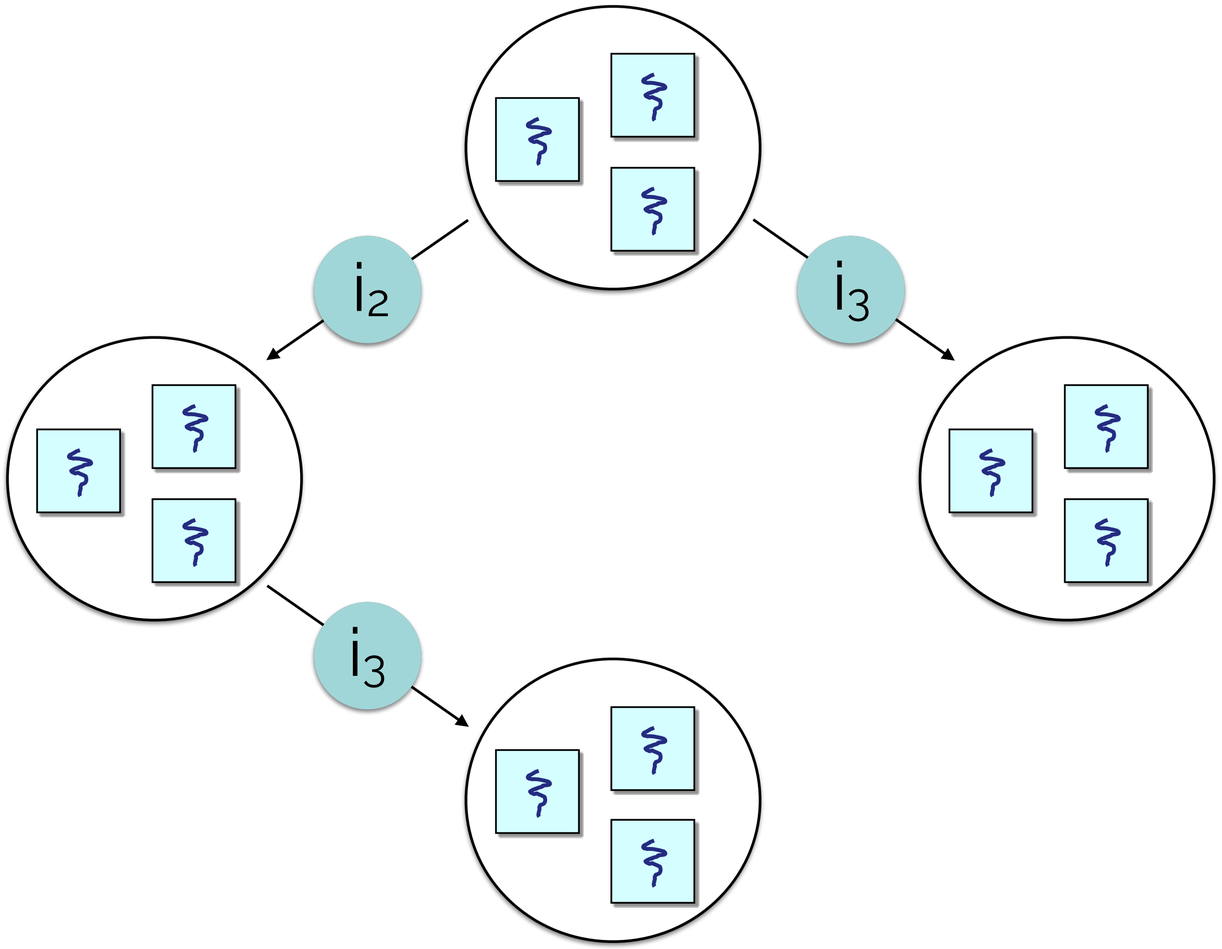
- $i_2 \nrightarrow i_3$
- $i_3 \nrightarrow i_2$
- $dst(i_2) \neq dst(i_3)$

Observation #1: many schedules are commutative

Step n:

Step n+1:

Step n+2:



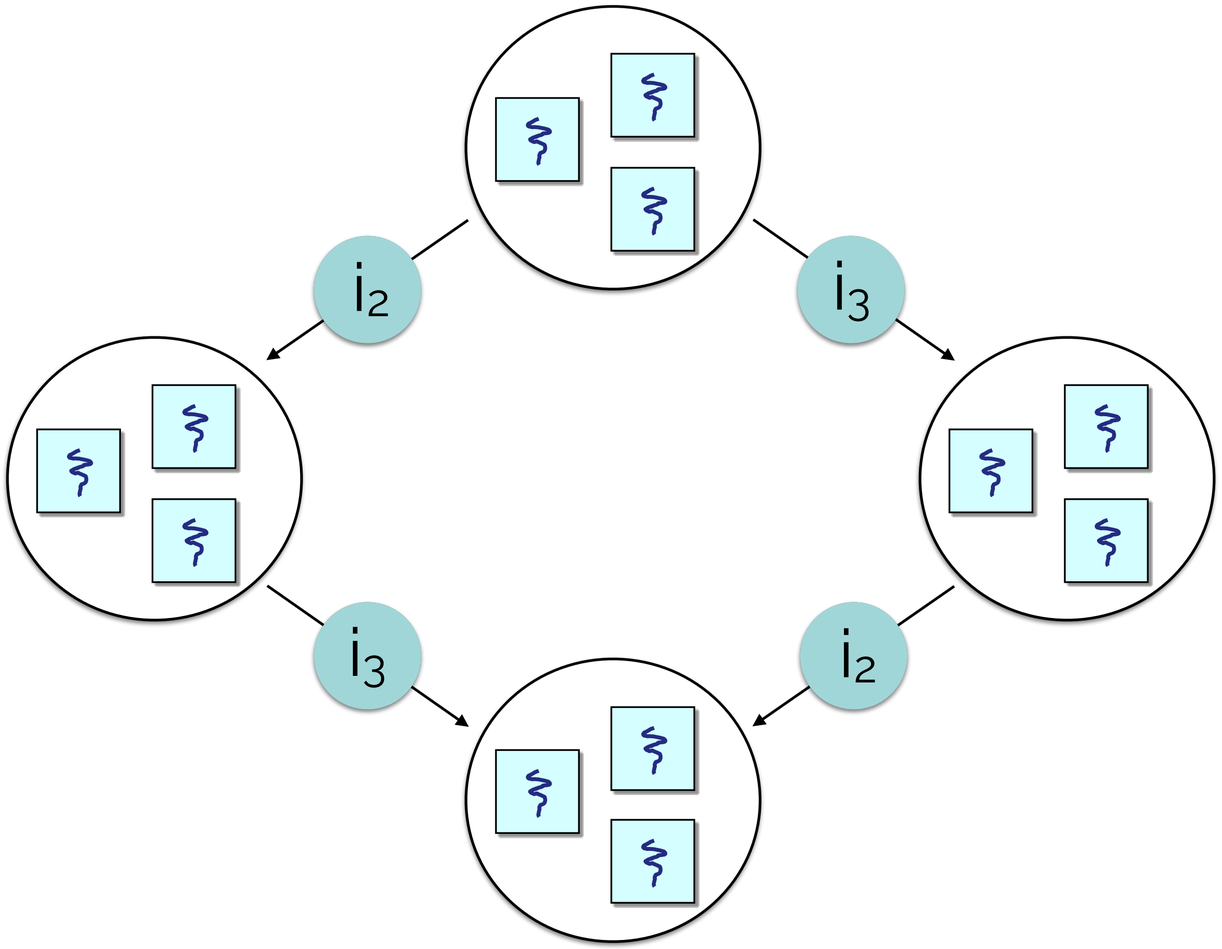
- $i_2 \nrightarrow i_3$
- $i_3 \nrightarrow i_2$
- $dst(i_2) \neq dst(i_3)$

Observation #1: many schedules are commutative

Step n:

Step n+1:

Step n+2:

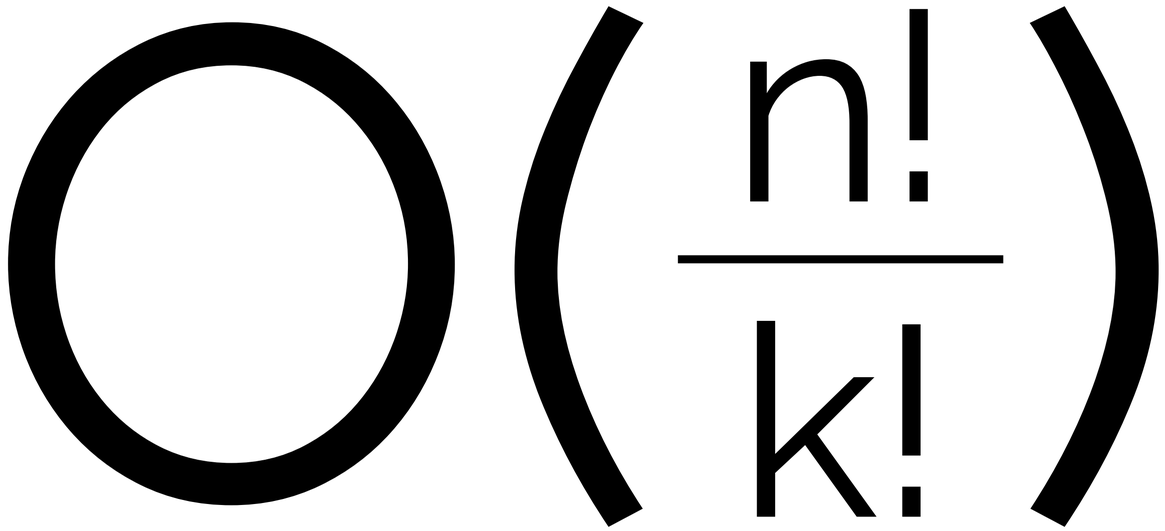


- $i_2 \nrightarrow i_3$
- $i_3 \nrightarrow i_2$
- $dst(i_2) \neq dst(i_3)$

Observation #1: many schedules are commutative

Adopt DPOR:
Dynamic **P**artial **O**rder **R**eduction

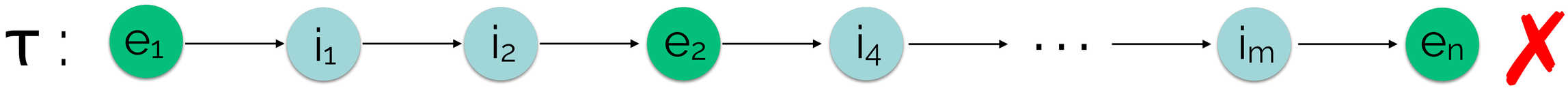
C. Flanagan, P. Godefroid, “Dynamic Partial-Order Reduction for Model Checking Software”, POPL ‘05


O(n!/k!)

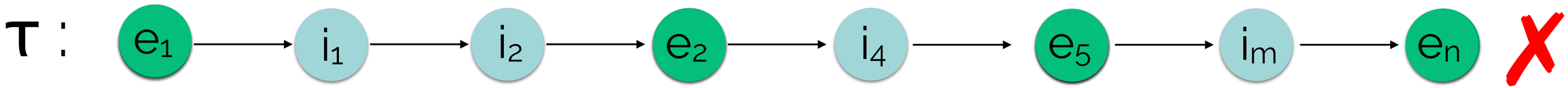
Approach: prioritize schedule space exploration

Assume: fixed time budget
Objective: quickly find small failing schedules

Observation #2: stay close to original schedule!

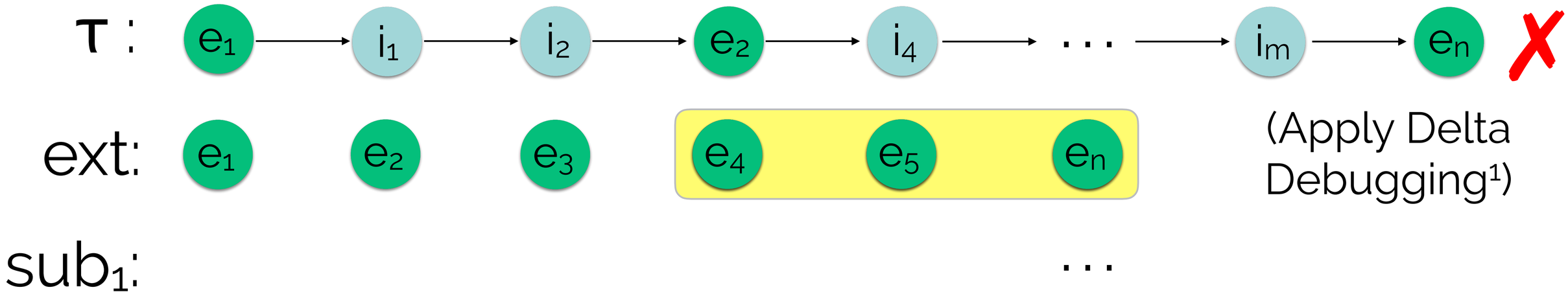


Observation #2: stay close to original schedule!



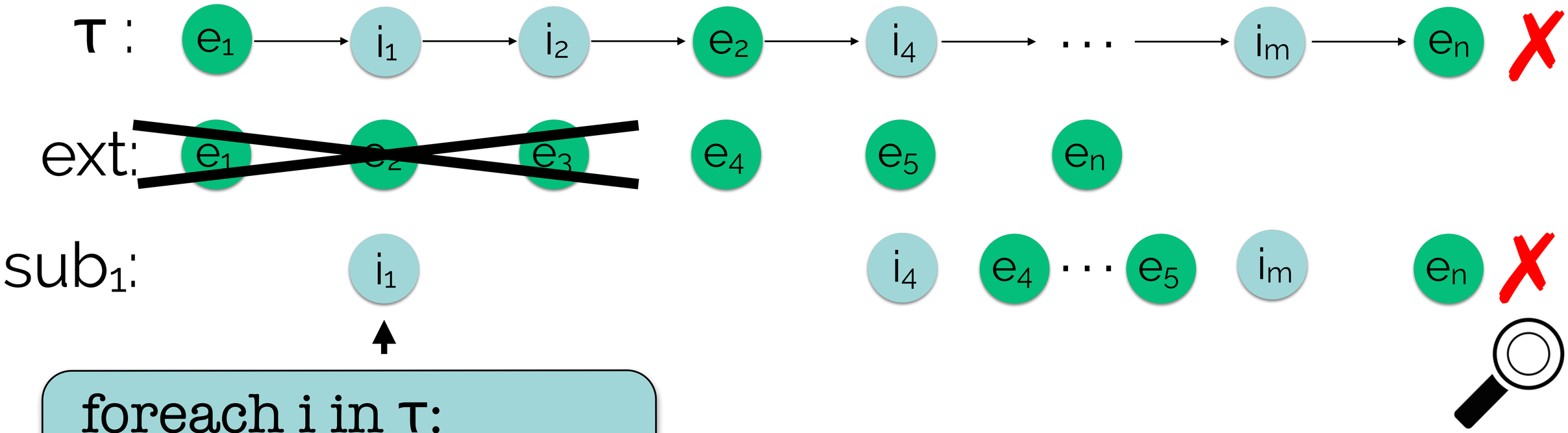
ext:

Observation #2: stay close to original schedule!



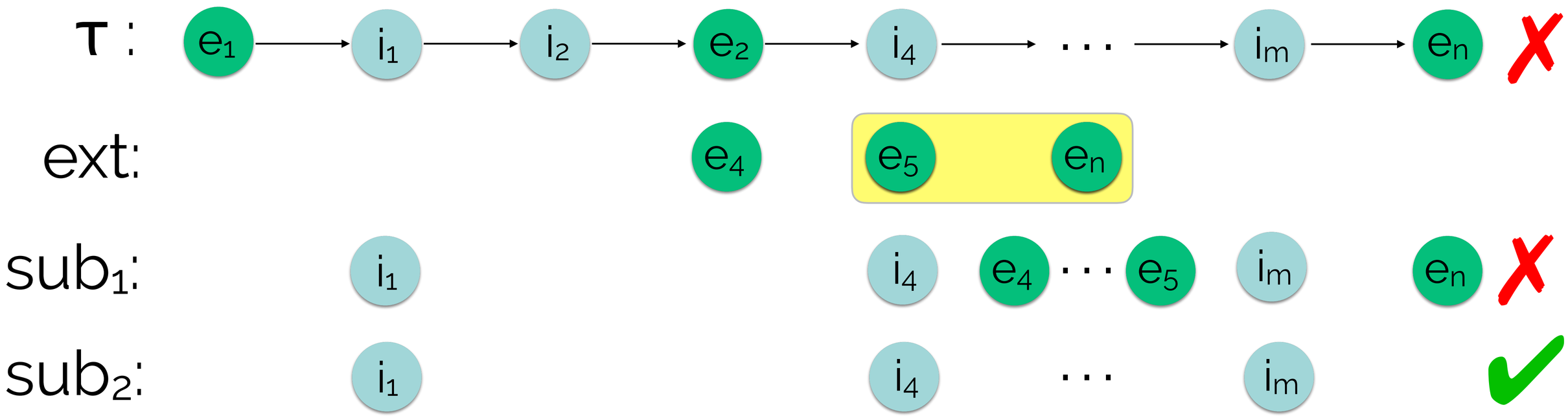
[1]A Zeller, R. Hildebrandt, “Simplifying and Isolating Failure-Inducing Input”, IEEE ‘02

Observation #2: stay close to original schedule!




```
foreach i in τ:
   if i is pending:
     deliver i
# ignore unexpected
```

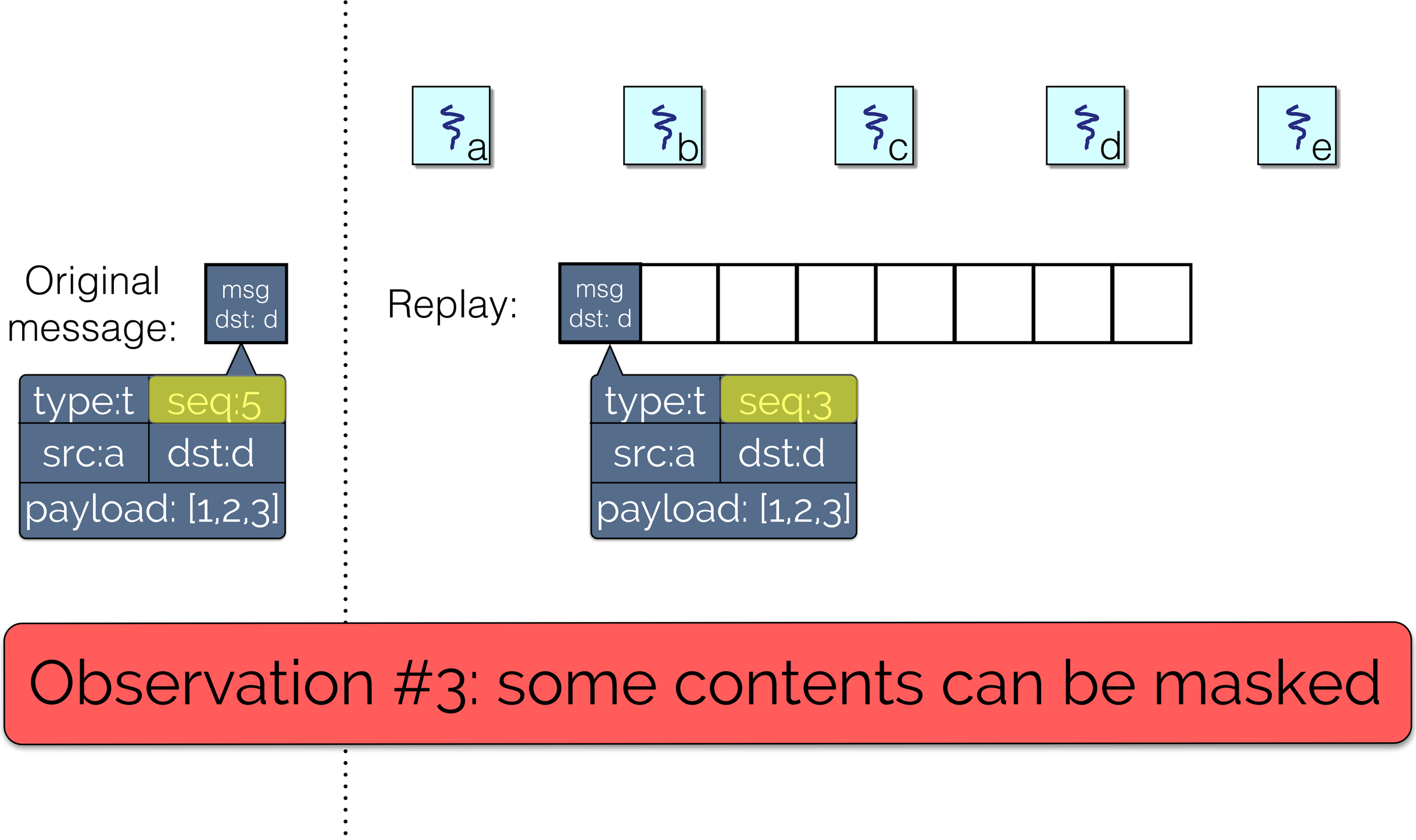
Observation #2: stay close to original schedule!



Explore backtrack points until (i) ✗ or (ii) time budget for sub_2 expired

. . .

Message contents may differ across executions!



Observation #3: some contents can be masked

Observation #3: some contents can be masked

Observation #3: some contents can be masked

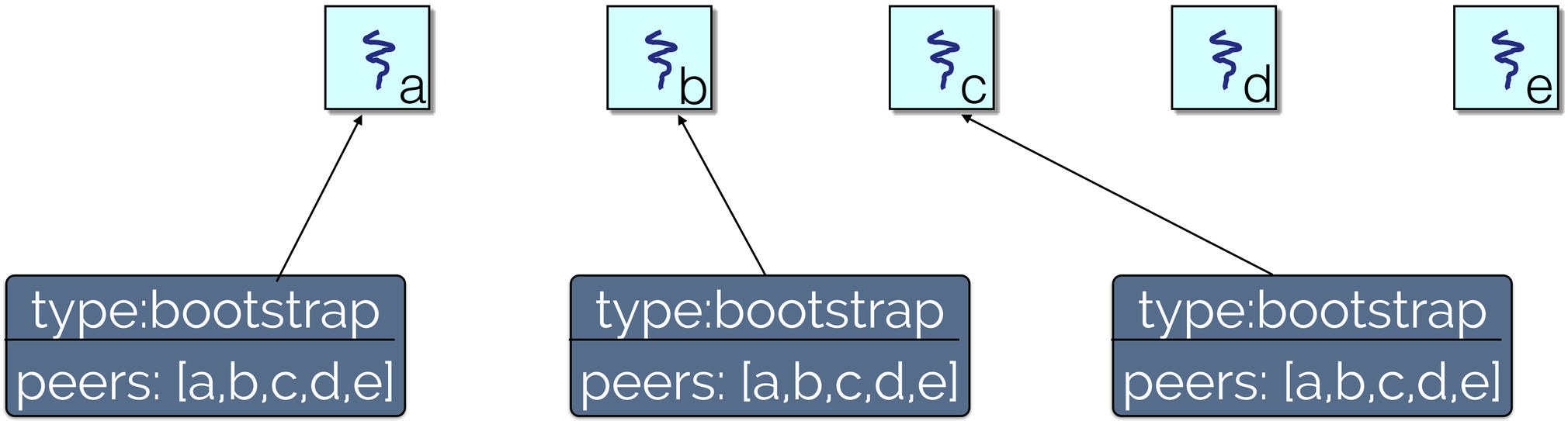
Phase 1:

- Match messages by user-defined “fingerprint”

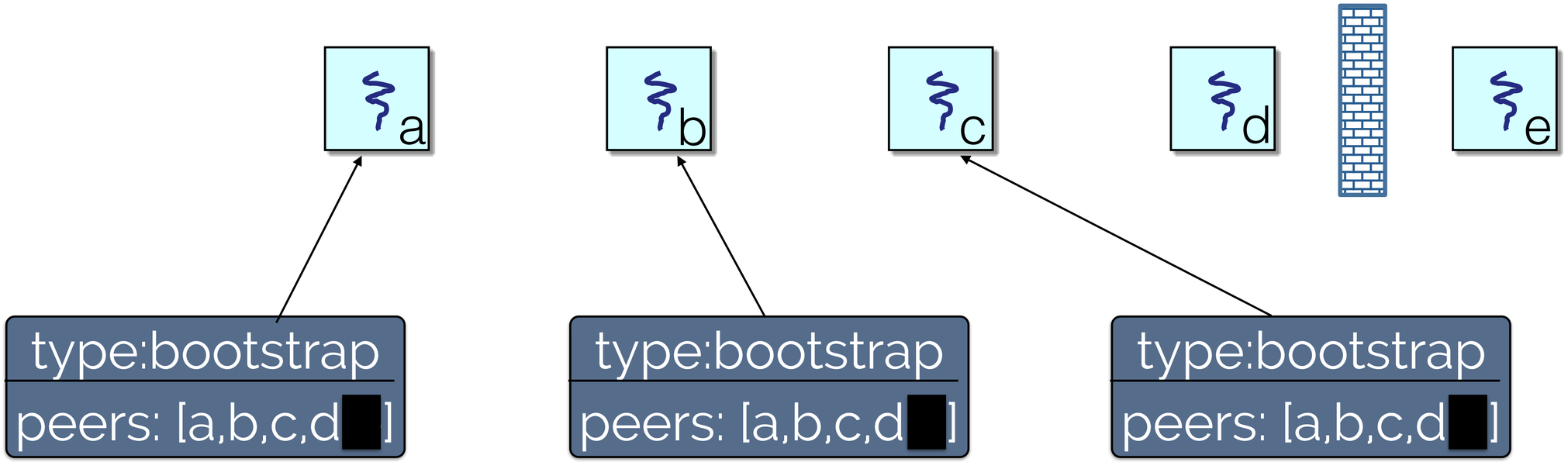
Phase 2:

- Match messages by type only
- Backtrack whenever multiple pending messages match by type

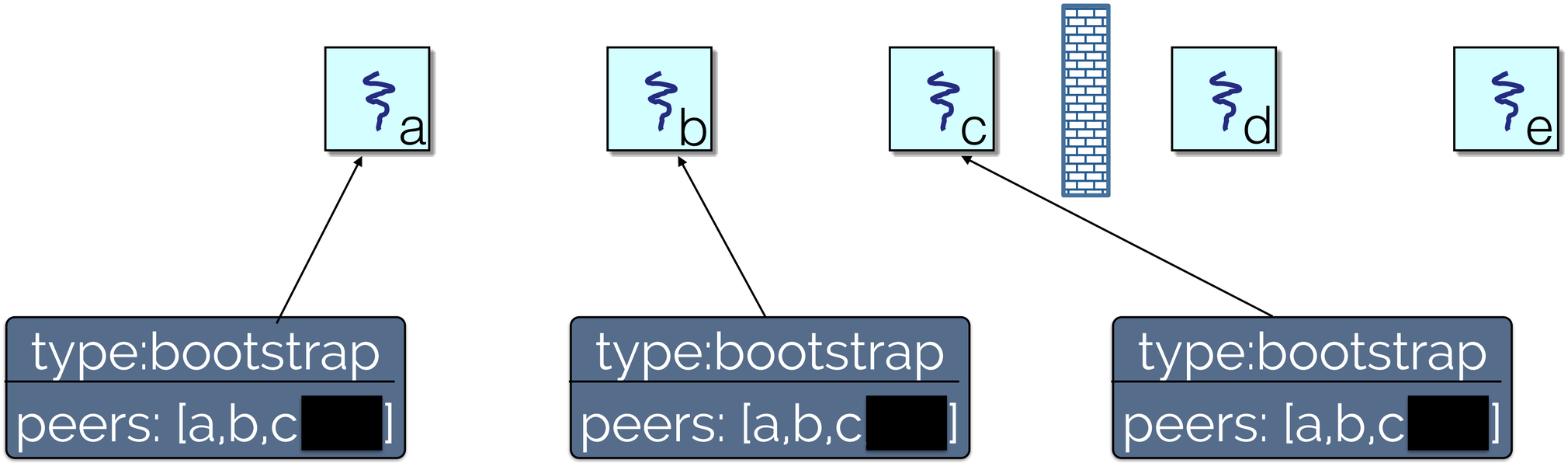
Observation #4: shrink external message contents



Observation #4: shrink external message contents



Observation #4: shrink external message contents



Goal: find minimal schedule that produces violation

Observation #1: many schedules are commutative

Approach: prioritize schedule space exploration

Observation #2: stay close to original schedule!

Observation #3: some contents can be masked

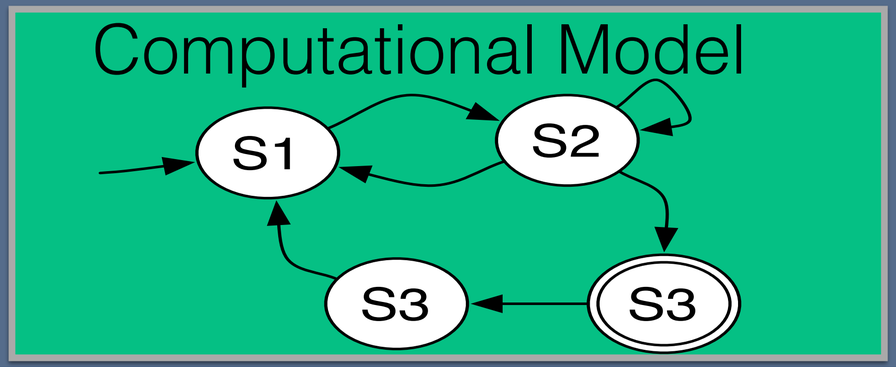
Observation #4: shrink external message contents

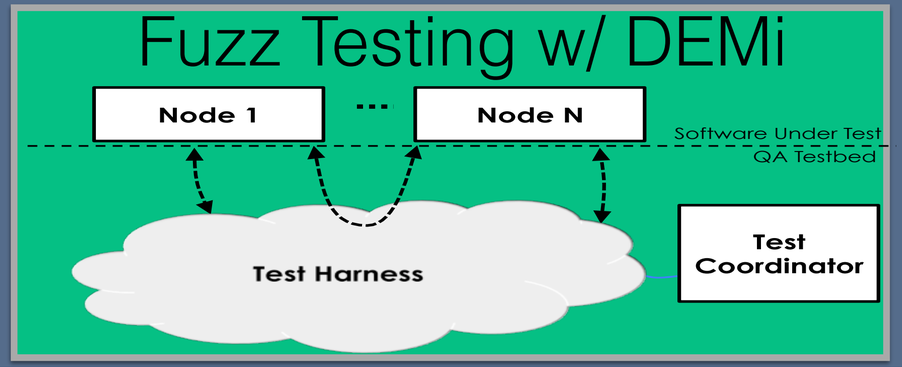
Minimize internal events after externals minimized

Outline

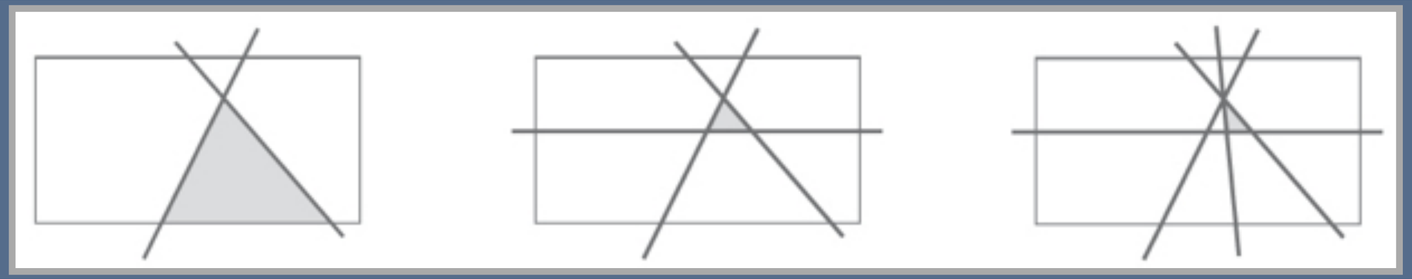
Introduction

Background





Minimization

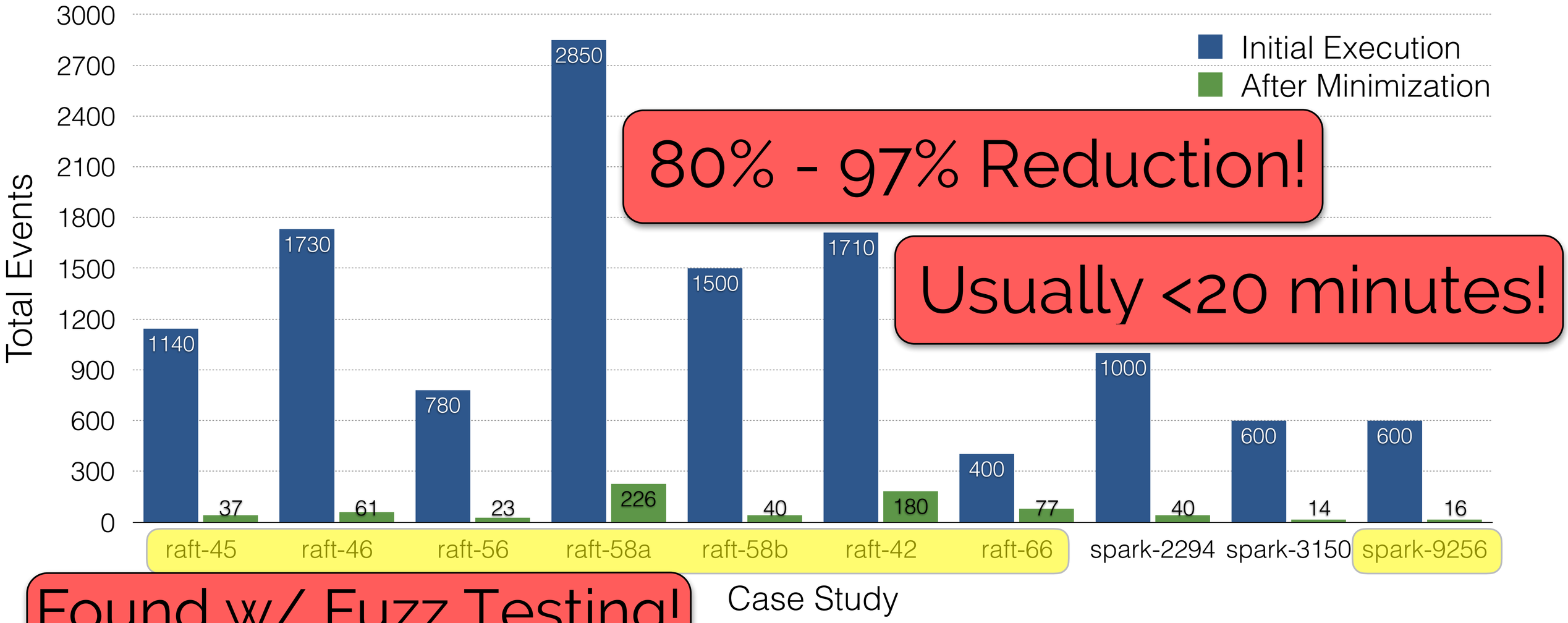


Evaluation

Conclusion

Target Systems

How well does DEMi work?



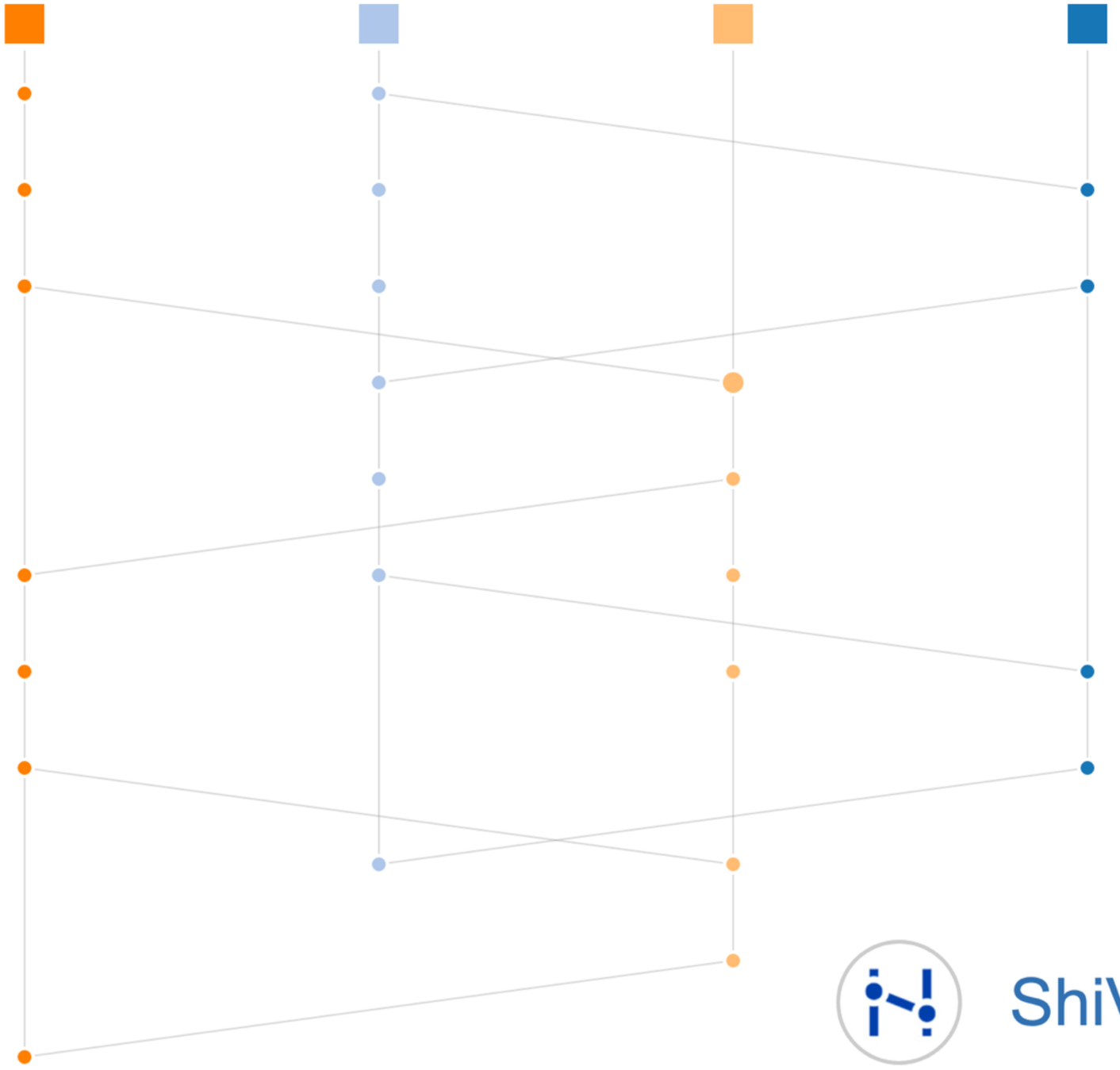
80% - 97% Reduction!

Usually <20 minutes!

Found w/ Fuzz Testing!

Log lines | Motifs

```
Sending to raft-member-4: RequestVote(Te
Sending to raft-member-2: BeginElection

Received message from raft-member-3: Req
Received timer: Timer(election-timer,Ele
Received message from raft-member-2: Beg

Sending to raft-member-3: VoteCandidate(
Sending to raft-member-3: BeginElection
Sending to raft-member-1: RequestVote(Te

Received message from raft-member-4: Vot
25 Received message from raft-member-2: R
   equestVote(Term(2),Actor[akka://new-sy
   stem-0/user/raft-member-2#-69475157
   3],Term(0),0)
Rec
Sending to raft-member-2: VoteCandidate(

Sending to raft-member-4: RequestVote(Te
Received message from raft-member-1: Vot
Received timer: Timer(election-timer,Ele

Received message from raft-member-3: Req
Received message from raft-member-2: Beg
Received timer: Timer(election-timer,Ele

Sending to raft-member-3: VoteCandidate(
Sending to raft-member-1: RequestVote(Te

Received message from raft-member-4: Vot
Received message from raft-member-2: Req

Sending to raft-member-2: VoteCandidate(

Received message from raft-member-1: Vot
```



ShiViz

See the paper for…

- How we handle non-determinism
- Handling multithreaded processes
- Supporting other RPC libraries
- Sketch for minimizing production traces
- More in-depth evaluation
- Related work
- …

Other DEMi Features

- Record/Replay for Debugging
- Time warping
- Automatically generating regression tests
- ...

Conclusion

You should minimize your bug reports!

Open source tool: github.com/NetSys/demi

Read our paper!
eecs.berkeley.edu/~rcs/research/nsdi_draft.pdf

Contact me! cs@cs.berkeley.edu

Thanks for your time!

Attributions

Inspiration for slide design: Jay Lorch's IronFleet slides

Graphic Icons: thenounproject.org
logfile: mantisshrimpdesign
magnifying glass: Ricardo Moreira
disk: Anton Outkine
hook: Seb Cornelius

Production Traces

Model: feed partially ordered log into single machine DEMi

Require:

- Partial ordering of all message deliveries
- All crash-recoveries logged to disk

Instrumentation Complexity

	akka-raft	Spark
Message Fingerprint	59	56
Non-Determinism	2	~250

Table 4: Instrumentation complexity (lines of code) needed to define message fingerprints, and to mitigate non-determinism.

Related Work

- Thread Schedule Minimization
  - Isolating Failure-Inducing Thread Schedules. SIGSOFT '02.
  - A Trace Simplification Technique for Effective Debugging of Concurrent Programs. FSE '10.
- Program Flow Analysis.
  - Enabling Tracing of Long-Running Multithreaded Programs via Dynamic Execution Reduction. ISSTA '07.
  - Toward Generating Reducible Replay Logs. PLDI '11.
- Best-Effort Replay of Field Failures
  - A Technique for Enabling and Supporting Debugging of Field Failures. ICSE '07.
  - Triage: Diagnosing Production Run Failures at the User's Site. SOSP '07.

DDmin in more detail

Input: $T_{\text{✘}}$ s.t. $T_{\text{✘}}$ is a trace and $\mathit{test}(T_{\text{✘}}) = \text{✘}$. Output: $T'_{\text{✘}} = \mathit{ddmin}(T_{\text{✘}})$ s.t. $T'_{\text{✘}} \subseteq T_{\text{✘}}$, $\mathit{test}(T'_{\text{✘}}) = \text{✘}$, and $T'_{\text{✘}}$ is minimal.

$$\mathit{ddmin}(T_{\text{✘}}) = \mathit{ddmin}_2(T_{\text{✘}}, \emptyset) \quad \text{where}$$

$$\mathit{ddmin}_2(T'_{\text{✘}}, R) = \begin{cases} T'_{\text{✘}} & \text{if } |T'_{\text{✘}}| = 1 \text{ (“base case”)} \\ \mathit{ddmin}_2(T_1, R) & \text{else if } \mathit{test}(T_1 \cup R) = \text{✘} \text{ (“in } T_1\text{”)} \\ \mathit{ddmin}_2(T_2, R) & \text{else if } \mathit{test}(T_2 \cup R) = \text{✘} \text{ (“in } T_2\text{”)} \\ \mathit{ddmin}_2(T_1, T_2 \cup R) \cup \mathit{ddmin}_2(T_2, T_1 \cup R) & \text{otherwise (“interference”)} \end{cases}$$

where $\mathit{test}(T)$ denotes the state of the system after executing the trace T, ✘ denotes an invariant violation, $T_1 \subset T'_{\text{✘}}$, $T_2 \subset T'_{\text{✘}}$, $T_1 \cup T_2 = T'_{\text{✘}}$, $T_1 \cap T_2 = \emptyset$, and $|T_1| \approx |T_2| \approx |T'_{\text{✘}}|/2$ hold.

DDmin assumptions

- *Monotonic:*

$$P \oplus C = \chi \Rightarrow P \oplus (C \cup C') \neq \checkmark$$

- *Unambiguous:*

$$P \oplus C = \chi \wedge P \oplus C' = \chi \Rightarrow P \oplus (C \cap C') \neq \checkmark$$

- *Consistent*

$$P \oplus C \neq ?$$

Local vs. global minima

Definition 8 (Global minimum). *A set* $c \subseteq c_\chi$ *is called the* global minimum *of* c_χ *if:* $\forall c' \subseteq c_\chi \cdot \big(|c'| < |c| \Rightarrow test(c') \neq \chi\big)$ *holds.*

Definition 10 (n-minimal test case). *A test case* $c \subseteq c_\chi$ *is* n-minimal *if:* $\forall c' \subset c \cdot |c| - |c'| \leq n \Rightarrow \big(test(c') \neq \chi\big)$ *holds. Consequently,* c *is 1-minimal if* $\forall \delta_i \in c \cdot test\big(c - \{\delta_i\}\big) \neq \chi$ *holds.*

Minimization Pace

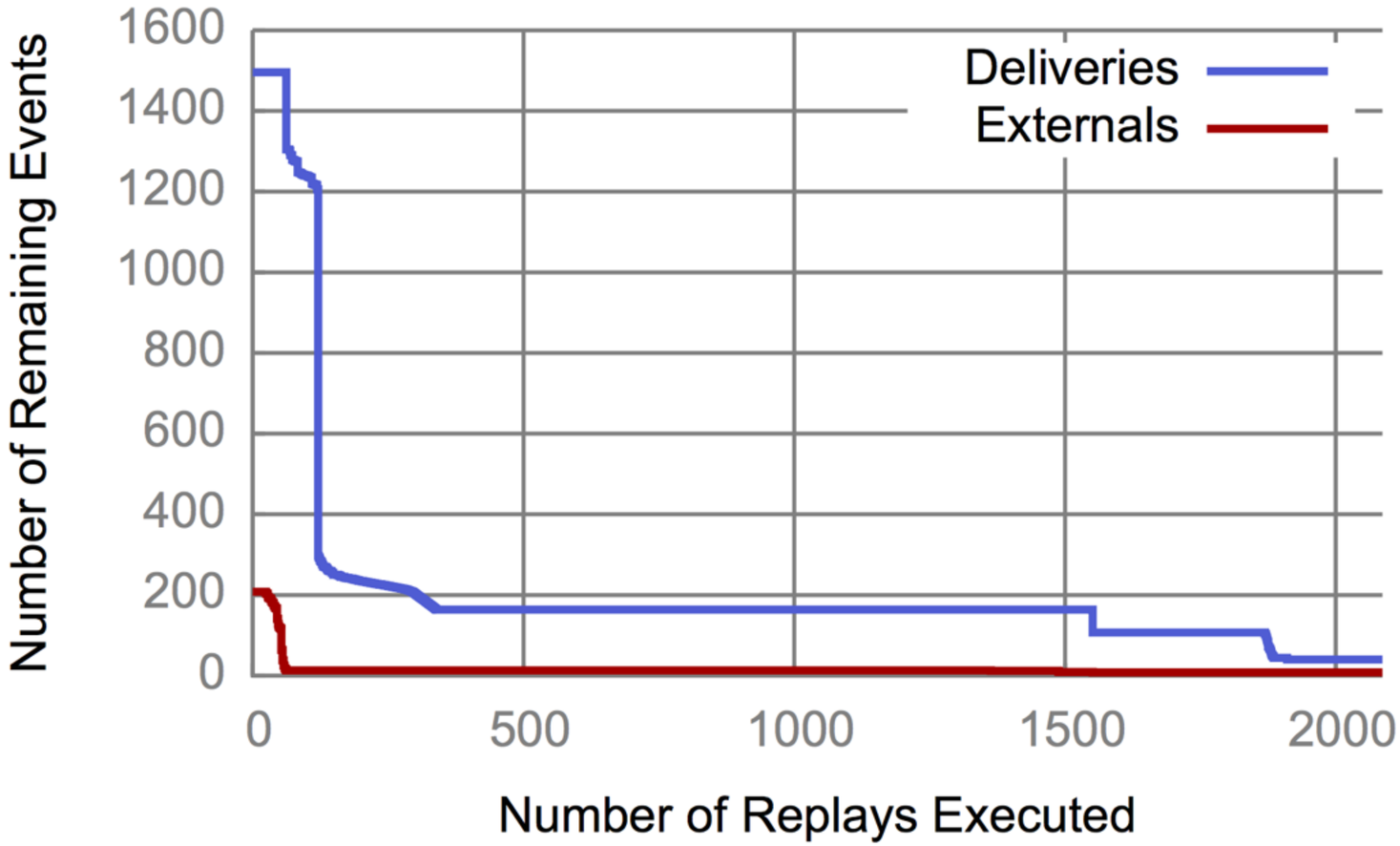


Figure 2: Minimization pace for raft-58b. Significant progress is made early on, then progress becomes rare.

Dealing With Threads

If you're lucky: threads are largely independent (Spark)
If you're unlucky: key insight:
A write to shared memory is equivalent to a message delivery

Approach:

- interpose on virtual memory, thread scheduler
- pause a thread whenever it writes to shared memory / disk

Cf. "Enabling Tracing Of Long-Running Multithreaded Programs Via Dynamic Execution Reduction", ISSTA '07

Dealing With Non-Determinism

Interpose on:

- Timers
- Random number generators
- Unordered hash values
- ID allocation

Stop-gap: replay each schedule multiple times

Integrating with other RPC libs