Minimizing Faulty Executions of Distributed Systems

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

Motivation in <60s

- Distributed systems are highly complex (concurrency, asynchrony, partial failure, …)
- ∴ Distributed systems are bug prone
- Popular way to catch bugs: QA (fuzz) testing
- Failing QA tests often hard to understand (hours long, multiple GBs in size)
- Much less time consuming to debug small traces

Dist'Sys Computational Model*

Distributed System: Collection of N processes

Each process *p:*

- Has unbounded internal storage
- Starts in a known initial state
- Changes states deterministically according to a transition function

*Inspired by FLP, JACM '85

Dist'Sys Computational Model

A message is a pair: *(p, m)*

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

Dist'Sys Computational Model

A *configuration* consists of:

- The internal state of each process
- The contents of the network buffer

Initially: empty network buffer, known initial states

Dist'Sys Computational Model

A *step* moves the system from one config to another

Normally:

- Network delivers some pending message *(p,m)*
- *p* enters a new state according to old state & transition function
- *p* sends a finite set of messages to other processes*

*May include *timer* messages to be delivered to itself later

Dist'Sys Computational Model

A *step* can also take another form: *external events*

Either:

- External message is delivered
- Process *p* fails, i.e. has its internal state reinitialized

Note: no need to explicitly model network partitions

Dist'Sys Computational Model

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

An *execution* is the corresponding sequence of steps.

Faulty executions

An invariant is a predicate *P* (a safety condition) over the internal state of each process at a particular configuration *C.*

A faulty execution is one that ends in *P(C) = false*

Problem Statement

Given: *P* and a schedule τ that produces an execution that violates *P*

Primary goal: find schedule τ' s.t.

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

Problem Statement

Secondary goal:

After finding τ', remove as many internal message events from τ' as possible while still causing violation

General Approach

Repeat:

1. Pick subsequence of external events from τ
2. Try to find a fault-inducing schedule containing those external events
3. If found: recurse

Key Challenge

Step 2 is intractable!

Research Agenda

- Find *heuristics* for exploring the schedule space s.t. fault-inducing schedules are found quickly
- Develop heuristics by *experimenting* with faulty executions found in real systems
- Formally characterize what program properties hold in those systems s.t. heuristics are effective

Contributions vs. previous paper

- Cleaner, more general problem statement
- New tool gives better control over the execution
- Broader system target (Spark, Raft, Pastry, JS)
- Novel heuristics, empirically validated
- Conjectures for explanatory program properties

New Tool

- Interpose on the Akka actor system
- Actors:
  - are single threaded, no shared state
  - compute by: i. receiving a message, ii. changing state, iii. sending out messages. (Atomic steps!)
  - send timer messages rather than reading from clock. (No dependence on wall-clock time!)

New Tool

- The Akka runtime is our “network”
- Interposition allows us to completely specify a linearizable sequence of steps to take
- Start with a test scenario written on top of Akka, then systematically explore schedules for that scenario

System Target

- akka-raft: consensus protocol written on top of akka
- Spark: control plane built on top of Akka
- Pastry: DHT written on top of kaka
- JavaScript: separate tool, similar computational model

System Target

- Find a variety of bug types in these systems (holy grail: find heuristics that are applicable to a wide range of bugs)
  - If possible, reproduce known bugs
  - Otherwise, find new bugs via fuzz testing

Example Bug

- akka-raft:
  - Two leaders elected in the same term
  - <Whiteboard>

Previous Heuristics

- Pick one schedule per subsequence
  - Define functional equivalence (fingerprints)
  - If unexpected event during replay, ignore
  - If absent event during replay, skip over

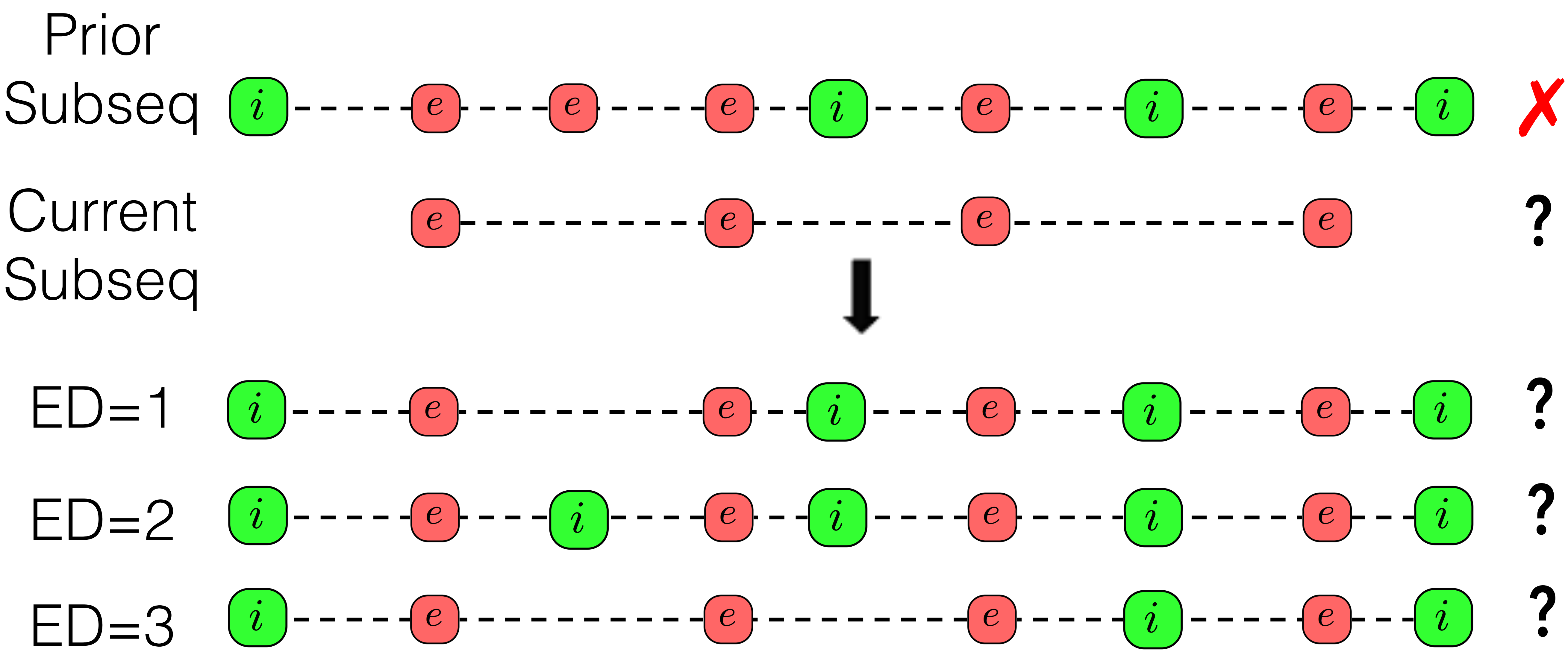
Results w/ Previous Heuristics

	Short Run	Long Run
Original Messages Deliveries	90	360
Removed by Provenance	26	25
Removed by DDmin	9	62
Removed by internal minimization	20	192
Final # of events	35 (7 external)	81 (8 external)

Old heuristics are suboptimal!

	Short Run	Long Run
Original Messages Deliveries	90	360
Removed by Provenance	26	25
Removed by DDmin	9	62
Removed by internal minimization	20	192
Final # of events	35 (7 external)	81 (8 external)

Heuristic: Bounded Edit Distance



Evaluation Methodology

- Empirically compare effectiveness of heuristics
- Key metrics:
  - Closeness to minimal result (if known)
  - # of schedules explored before terminating

Program Properties

- Data Independence
- Bounded Edit-Distance
- Commutativity
- Recency of State

Stop Gap: Fairness

- Enforce fixed computational budget for every external event subsequence, spread evenly over external events

Related Work

- traditional minimization (DDmin, QuickCheck's shrinking)
  - not immediately applicable to systems with intermediate external events
- best-effort minimization (QuickCheck applied to concurrent systems, field failures)
  - don't systematically handle concurrency
- model checking minimization: MAX-Sat, interpolation (ConcBugAssist, DSPs)
  - many disadvantages of model checking. See Q & A.
- deterministic replay log minimization (schedule minimization, program analysis)
  - don't allow divergence during replay! i.e. minimization specification is overly specific -> minimization results aren't great
  - program flow analysis work is tied to a single language, + high perf overhead
- model inference, log summarization (synoptic, csight, invariant mining)
  - summarize the events that occurred, but don't actually minimize the test case. Apply this after you minimize the test case with our techniques.
- program slicing, automated debugging (experimental, program analysis)
  - minimize program statements, but not the test case itself. Useful for debugging, but not troubleshooting. Apply this after you minimize the test case with our techniques.

Q & A

- Why not analyze the program?
  - ties you to one language
  - parsers usually aren't complete; will sometimes have to make assumptions that aren't realistic
  - requires engineering effort to get working
  - high computational overhead of tracing
  - by solving a harder problem, we come up with more interesting solutions

- Is QA testing going to be around forever? Why not use newer more sophisticated testing tools?
  - As long as you have bugs in production, you will need minimization.