Online Testing of Distributed Systems

Dejan Kostić

EPFL, Switzerland
Networked Systems Laboratory
Distributed Systems

Foundation of our society's infrastructure

- Enterprise storage, Web services, Internet
Internet Routing is Unreliable

- Operator mistakes (**hijacks:** Pakistan 2008, China 2010)

Hijacked 10% of the Internet (Dell, CNN, Amazon.de, ...)

Popular site inaccessible for ~ 2 hours!
Software Bugs in Inter-Domain Routers

At 17:07:26 UTC on August 19, 2009 CNCI (AS9354), a small network service provider in Nagoya, Japan, advertised a handful of BGP updates containing an empty AS4_PATH attribute. [renesys blog]
...what could possibly go wrong?

10x increase ➔ routing instabilities

[renesys blog]
What went wrong (Cisco session reset flood)

Local testing is insufficient

Locally admissible action can have harmful global effect

Repeated service disruptions
Harness the increases in computational power and bandwidth to improve system reliability

Moore’s law

[http://www.nordicbroadbandblog.com/]
Detect Possible Faults and Remove Them!

Detection (in parallel)

Live

Time

A

B

C

D
Key Insight: Online Testing
• CrystalBall
  – Introduced live model checking to predict and avoid inconsistencies in distributed systems

• DiCE
  – Enabled systematic “what-if” exploration across heterogeneous, federated distributed systems

• NICE
  – Combined model checking and symbolic execution to test OpenFlow applications
Model Checking DS to Predict the Future

- State space exploration, e.g., MaceMC [Killian et al., NSDI‘07]
  - Run each enabled handler in a set of nodes (live code)
  - Check safety properties in each state

Exponential explosion in the number of states!

- Local actions
  - Timers
  - Node reset
  - Network messages
New concept: periodically collect consistent snapshot and execute state space exploration from current state.
CrystalBall Benefits

State-of-the-art in distributed debugging

- a) Classic model checking
- b) Replay-based/live predicate checking
- c) CB: deep online debugging
- d) CB: execution steering
Consequence Prediction

- Goal: quickly judge consequences of node actions
- State-of-the-art is too slow
  - 30 min for 6 steps

Key idea:
- Remove previously explored local node actions
  - Network messages can still be interleaved
- Factor of 100 speedup
Inconsistencies Found

<table>
<thead>
<tr>
<th>System</th>
<th>Inconsistencies not found by MaceMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandTree</td>
<td>2</td>
</tr>
<tr>
<td>Chord</td>
<td>2</td>
</tr>
<tr>
<td>Bullet’</td>
<td>3</td>
</tr>
</tbody>
</table>

- Previously, these systems were debugged both in local- and wide-area settings,
- Model checked using MaceMC (RandTree, Chord)
Execution Steering (ES)

• Goal: increase resilience of deployed systems
  → Initial focus on generic runtime mechanisms

• Goal: prevent inconsistencies without introducing new ones
  – As difficult as guaranteeing that the entire program is error-free
    • Hard problem
Execution Steering (ES)

• Approach:
  – Install an event filter to block offending action
  – Use *sound* filters (events normally occurring)
    • Break TCP connection/drop UDP message
  – Run consequence prediction to check the effect of the event filter on system
Paxos Execution Steering

- Paxos: protocol for achieving consensus
- Stress test: 2x100 live runs that expose Paxos safety violations due to two injected bugs
- One CPU core for CrystalBall

- < 5% False negatives
- Small performance impact (5%)

Bar chart showing violations avoided by immediate safety check and avoided by execution steering for bug1 and bug2.
Challenges in Widely-Deployed Systems

• Heterogeneity
  – Multiple implementations, versions, patches
  → Difficult/impossible to obtain source/binary code

• Federation
  – Multiple administrative domains
  → Cannot obtain internal state/configuration

Cannot use CrystalBall to detect Internet routing faults!
DiCE [USENIX ’11, TR ’11]

Automatically explore system behavior to detect faults:
1. Create an isolated snapshot of a node’s neighborhood
2. Subject node to many relevant inputs to exercise its actions
3. For each input, check if the snapshot misbehaves
Goal: Isolate testing from production while observing “societal” boundaries.

DiCE Snapshot

Local checkpoint of current state and configuration

BGP is a federated environment → Each router keeps its local checkpoint → Private state & config stays in the AS

To run one exploration (“what-if” scenario): lazily clone the snapshot
Exploration of Behavior

To systematically exercise node behavior: Use a path exploration engine

Is there an error?

阝 Clone of BGP process
Concolic (CONConcrete+symbOLIC) Execution
[Godefroid05, Cadar06, Cadar08, Burnim08, Crameri09]

X = 0 real input

X = 20 real input, for X > 0?

X = 5 real input, X > 0 && X < 10?

Automatically uncover test inputs using code itself!

Challenges:

• Coverage
  – Deal with exponential number of paths

• Large inputs
  – Handle systems with long running times

• Short running time
  – To allow for preventative measures
Managing Path Explosion and Generating Inputs

Input generation
- Config change
- Fuzz msg
- Perturb list

Explorer Node
- Current, live state
- Live (production)

Explorer Node
- (Message) Handler Code
- Exploration

Input control
- Path constraints
- Path exploration engine

Input control
- Input
Check properties that capture desired behavior (e.g., safety)

- **Example:** Harmful Global Events (session resets)
- Safety properties can be automatically inferred [SRDS’11]

**Report:**
List of packets that cause harmful global behavior

- Error count > threshold?

- Unaffected router
- Affected router

Detecting Faults
Integration Details

• Integrated DiCE with BGP in the BIRD router
  – Open-source router, coded in C
  – Fuzzed path attributes in BGP messages
  – Marked symbolic inputs in BIRD (a few LoC)

• Integrated DiCE with MaraDNS
  – Open-source DNS server, coded in C
  – 74 LoC to integrate with the concolic engine

### BGP UPDATE

<table>
<thead>
<tr>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Withdrawn Routes</td>
</tr>
<tr>
<td>Path Attributes</td>
</tr>
<tr>
<td>Attribute Type</td>
</tr>
<tr>
<td>Network Layer Reachability</td>
</tr>
<tr>
<td>NLRI Length</td>
</tr>
</tbody>
</table>
BGP Evaluation

• Microbenchmark:
  – Less than 5% performance impact during exploration
    • on a single CPU core, while processing routing updates

• Emulated 27-router network
  – Using ModelNet [OSDI ‘02]

• BGP properties that capture resilient faults
  1. Harmful Global Events
  2. Origin Misconfiguration
  3. Policy Conflicts
BGP Evaluation Results

Explored all paths in the UPDATE handlers + across the Internet-like testbed in ~4 min avg (11 min max)

Detected session reset and origin misconfiguration
Presence of a logically centralized controller does not eliminate bugs!
Bugs in OpenFlow Apps

Controller

OpenFlow program

Install rule (delayed)

Switch 1

Switch 2

Host A

Packet

Host B

Time
Subject unmodified Apps to a multitude of inputs:

1. Values of packet header fields
2. Orderings of packet arrivals and events

Exercise possible corner cases by systematically exploring the state space via model checking
Building a State-Space Model

Transition system:
- Components
- Channels
- State
- Transitions
- System execution

Switch and network state is large

Introduced simplified switch model for 7x size reduction
Overcoming Scalability Challenges

Large # of possible packets

Data-plane driven

Equivalence classes of packets

Enumerating all inputs and event orderings is intractable

Large # of possible event orderings

Complex network behavior

Domain-specific search heuristics
Combating Huge Space of Packets

💡 Use Symbolic Execution
  – Code itself reveals relevant packets

• 1 path = 1 eq. class = 1 packet to inject

Example equivalence classes:
1. Broadcast destination
2. Unknown unicast destination
3. Known unicast destination

Unreachable from initial state

Install rule and forward packet

Flood packet
State 1

New relevant packets:

\[ \text{pkt}_2, \text{pkt}_3 \]

Enable new transitions:

\[ \text{client}_1 / \text{send} (\text{pkt}_2) \]

\[ \text{client}_1 / \text{send} (\text{pkt}_3) \]

Symbolic execution of packet_in handler:

Controller state changes:

Controller state

Augmenting Model Checking with Symbolic Execution

discover packets transition:
Combating Huge Space of Orderings

OpenFlow-specific search strategies for up to \textbf{20x} state-space reduction:

5x faster than state-of-the-art tools (e.g., JPF)
Specifying App Correctness

• Library of common properties
  – No forwarding loops
  – No black holes
  – Direct paths (no unnecessary flooding)
  – ...

• Exposed API to define app-specific properties
  – Python code snippets that check network state
Prototype & Experiences

• Built a NICE prototype in Python
• Tested 3 unmodified NOX OpenFlow Apps
  – MAC-learning switch (pyswitch.py)
  – Web Server Load Balancer [Wang et al., HotICE ’11]
  – Energy-Efficient Traffic Engineering [CoNEXT ’11]
• NICE found 11 property violations \( \rightarrow \) bugs
  – Violations found in a few seconds (30 min max)
  – 1-2 simple bugs (forgotten packet in ctrl)
  – 3 insidious bugs due to subtle race conditions
Future Work

• Test with real OpenFlow switches
• Incorporate online safety checks into NICE
• Simplify development and deployment of distributed systems
More information about NICE

• NICE source code + technical report:

http://prophet.epfl.ch
Related Work

• Model checking
  – CMC [Musuvathi02], MaceMC [Killian07], MODIST [Yang09]

• Offline/online predicate checking
  – Friday [Geels07], OverLog/P2 [Singh06], D3S [Liu08]

• Symbolic execution
  – Exe [Cadar06], Klee [Cadar08], KleeNet [Sasnauskas10], S2E [Chipounov11]

• Model checking + Symbolic execution
  – Java PathFinder [Visser03]

• Static analysis of network configurations
  – RCC [Feamster05], FlowChecker [Al-Shaer10], AntEater [Mai11]

• Runtime mechanisms
  – {Peer,Net}Review [Haeberlen07,09], Bouncer [Costa07],
    Rx [Qin05], BTR [Keller09], ShadowConfig [Alimi09]
Conclusion

• Hard to make distributed systems reliable
  – Even harder when they are heterogeneous and federated

• Our vision:
  – At runtime, detect possible deviations from correct behavior
  – Steer execution away from the predicted faults
  – New way of achieving reliability in distributed systems

• The systems we built, CrystalBall/DiCE:
  – Quickly detect important classes of faults, with low overhead

• NICE automates testing of unmodified OpenFlow apps
  – Combines model checking with symbolic execution
  – Uses domain-specific search strategies
Thanks!

**CrystalBall:**  Maysam Yabandeh, Nikola Knežević, Viktor Kuncak

**DiCE:**  Marco Canini, Daniele Venzano, Vojin Jovanović, Gautam Kumar, Dejan Novaković, Boris Spasojević, Olivier Crameri

**NICE:**  Marco Canini, Daniele Venzano, Peter Perešíni, Jennifer Rexford
Questions?