Fast Control Plane Analysis Using an Abstract Representation

Aditya Akella

Aaron Gember-Jacobson, Raajay Viswanathan and Ratul Mahajan

UW-Madison and Microsoft
Control plane is...

- Essential → configuration errors may cause security/availability problems
- Complex → errors may not be immediately apparent

To: A

Datacenter Dynamics
The Business of Data Centers.

Microsoft: misconfigured network device led to Azure outage

30 July 2012 | By Yevgeniy Sverdlik

“The service interruption was triggered by a misconfigured network device that disrupted traffic to one cluster in our West Europe sub-region,” Mike Neil, general manager for Windows Azure, wrote in a blog post.

“Once a set device limit for external connections was reached, it triggered previously unknown issues in another network device within that cluster, which further complicated network management and recovery.”
Important functional invariants

- Always blocked
- Always isolated
- Always traverse middlebox
- Always equivalent paths

Challenge: Invariants violated under some (combinations of) failures
Analyze current data plane [HSA, Veriflow] → cannot verify invariants always hold

Generate data planes [Batfish] → time consuming

Proactive Verification

Blocked, isolated, waypoints, equivalence ...

- Properties of paths, not paths themselves
- Data centers, enterprises use a limited set of control plane constructs

Higher-level abstraction
Fast analysis
Abstract Representation for Control planes (ARC)

- Encodes the network’s forwarding behavior under *all* possible infrastructure faults
- Proactive verification boils down to checking simple graph-level properties \(\rightarrow\) fast
- Ignore which protocols used and how

Control plane configuration  
Abstract representation
Key requirements of ARC

1) **Sound & Complete**: each digraph contains *every feasible* path and *no infeasible* paths → verification of invariants

2) **Precise**: assign edge weights such that the *min-cost path* matches the real path → counter-examples, equivalence testing
• Why weighted digraphs?

• How to ensure soundness, completeness, precision?
Routing protocols used today

- **Commonality**: cost-based path selection algorithm
- **Differences**: granularity & currency
- Also must account for:
  - Traffic class specificity
  - Route redistribution
  - Route selection based on administrative distance

**Challenge**: determining the structure and edge weights of the graphs
Extended topology graphs (ETGs)

- One per traffic class
- **Vertices**: routing processes
- **Edges**: flow of data enabled by exchange of routing information

Edge-weights based on configured costs and administrative distances
ETG edge weights

- **Inter-device**: OSPF weights; unit cost per hop for BGP (each router is an AS)
- **Intra-device**: redistribution only: no cost within process; fixed-cost between processes + scaling

![Diagram showing OSPF and BGP processes with edge weights and paths]

**Precise**
(for DAG redistribution, AD graphs)

- Longest path = 0.5
- Shortest path = 1
- Gap = 1

![Diagram with OSPF and BGP processes connected with arrows and weights]
## ARC properties

<table>
<thead>
<tr>
<th>Construct</th>
<th>Sound &amp; Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSPF</td>
<td>✓</td>
</tr>
<tr>
<td>RIP</td>
<td>✓</td>
</tr>
<tr>
<td>eBGP</td>
<td>✓</td>
</tr>
<tr>
<td>Static Routes</td>
<td>✓</td>
</tr>
<tr>
<td>ACLs</td>
<td>✓</td>
</tr>
<tr>
<td>Route filters</td>
<td>✓</td>
</tr>
<tr>
<td>Route selection (based on Administrative Distance)</td>
<td>✓</td>
</tr>
<tr>
<td>Route redistribution</td>
<td>✓</td>
</tr>
</tbody>
</table>

Sound and complete for **100%**

Precise for **96%**
Verification

• *Always traverse middlebox*:  
  1) remove all edges with middleboxes  
  2) Src and Dst in same connected component?
Verification

- Always reachable with < k link failures: max-flow on unit-weight ETG ≥ k?

3 edge-disjoint paths

Max-flow = 3
## Verification

<table>
<thead>
<tr>
<th>Invariant</th>
<th>Graph property</th>
<th>Required ARC Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always blocked</td>
<td>Separate connected components</td>
<td>Sound &amp; Complete</td>
</tr>
<tr>
<td>Always reachable with &lt; k failures</td>
<td>Max flow $\geq k$</td>
<td>Sound &amp; Complete</td>
</tr>
<tr>
<td>Always traverse waypoint (chain)</td>
<td>Separate connected components</td>
<td>Sound &amp; Complete</td>
</tr>
<tr>
<td>Always isolated</td>
<td>No common edges</td>
<td>Sound &amp; Complete</td>
</tr>
<tr>
<td>Equivalence</td>
<td>Same structure &amp; weights</td>
<td>Sound, Complete, &amp; Precise</td>
</tr>
</tbody>
</table>

**Precision required to produce counter-examples**
Implementation and evaluation

- Implemented in Java using Batfish (parsing) and JGraphT (graph algorithms)
  [https://bitbucket.org/uw-madison-networking-research/arc](https://bitbucket.org/uw-madison-networking-research/arc)

- Configurations from 314 data center networks operated by a large online service provider

- 4-core 2.8GHz CPU
  24GB RAM
Evaluation: time to generate ARC

Most time is spent parsing

Fast (< 10 sec) even for large networks
Evaluation: verification time

- **Always blocked**: 
  - < 500 ms
  - (Batfish: 694 days!)

- **Always reachable with < k failures**: 
  - < 1 sec

- **Always isolated**: 
  - Up to 16 min

Verification time is proportional to the number of traffic classes; easily parallelized.
Next steps

- Precision under fewer assumptions
- Generality of ARCs
- Other uses...
Next steps: automated repair

1) Transform ETGs to have desired attributes (e.g., src and dst $\rightarrow$ always blocked)

2) Translate to config changes (e.g., remove edge $\rightarrow$ add ACL)

Challenge: finding a *minimal* repair (e.g., many ACLs vs. remove BGP neighbor) without side-effects
Next steps: Transition to SDN

Controller uses ETGs to drive forwarding plane configurations

Minimize controller involvement, churn?

Different underlying network topology?
Next steps: synthesis

• Operators require fine-grained control over routing: waypoints, isolation, traffic engineering
  – *Intents* $\rightarrow$ *configurations*
• Distributed routing based on shortest path – very difficult to program!
• One approach: input data planes $\rightarrow$ resilient ARCs $\rightarrow$ configs
Synthesis

- Edge weights
  - Input path to $dst$ must be the shortest path
  - Uniqueness of shortest path
- Route filtering
  - Disable edges for a destination to ensure path is shortest
- Backup paths
  - Weights such that backup path is chosen during link failures
Summary

• Presented an abstract representation for control planes
  – Fast and simple verification under arbitrary failures
  – Verification is based on graph-level properties
  – Up to 5 orders of magnitude speed-up

• Useful for repair, transition, synthesis, ...

Try it!
https://bitbucket.org/uw-madison-networking-research/arc
Backup
Evaluation: verification time

Always blocked using ARC
< 500 ms

Always blocked using Batfish
> 694 days!

Verification with ARC is 3 to 5 orders of magnitude faster!
Verification

- **Always blocked**: Src and Dst in same connected component?
Fast Control Plane Analysis
Using an Abstract Representation

Aditya Akella

Aaron Gember-Jacobson, Raajay Viswanathan and Ratul Mahajan

UW-Madison and Microsoft
Fast Control Plane Analysis Using an Abstract Representation

Aditya Akella