Provable Data Plane Connectivity with Local Fast Failover

Introducing OpenFlow Graph Algorithms

Michael Borokhovich (Ben Gurion Uni, Israel)
Liron Schiff (Tel Aviv Uni, Israel)
Stefan Schmid (TU Berlin & T-Labs, Germany)
Provable Data Plane Connectivity with Local Fast Failover

Introducing OpenFlow Graph Algorithms

Michael Borokhovich (Ben Gurion Uni, Israel)
Liron Schiff (Tel Aviv Uni, Israel)
Stefan Schmid (TU Berlin & T-Labs, Germany)
Before failover:

- **Link failures** today are not uncommon

- Modern networks provide **robust routing mechanisms**
  - i.e., routing which reacts to failures
  - example: MPLS local and global path protection

After failover:
Fast In-band Failover

- Important that failover happens fast = in-band
  - Reaction time in control plane can be orders of magnitude slower [1]
- For this reason: **OpenFlow Local Fast Failover Mechanism**
  - Supports conditional forwarding rules (depend on the local state of the link: live or not?)
- Gives fast but local and perhaps "suboptimal" forwarding sets
  - Controller improves globally later…

![Diagram of network](image)
• Important that failover happens fast = in-band
  • Reaction time in control plane can be orders of magnitude slower
• For this reason: **OpenFlow Local Fast Failover Mechanism**
  • Supports conditional forwarding rules (depend on the local state of the link: live or not?)
• Gives fast but local and perhaps “suboptimal” forwarding sets
  • Controller improves globally later…

However, not much is known about how to *use* the OpenFlow fast failover mechanism. E.g.: **How many failures** can be tolerated without losing connectivity?
However, not much is known about how to use the OpenFlow fast failover mechanism. E.g.: **How many failures** can be tolerated without losing connectivity?

How to use mechanism is a **non-trivial problem** even if underlying network stays connected: (1) conditional failover rules need to be allocated **ahead of time**, without knowing actual failures, (2) views at runtime are **inherently local**.

How not to **shoot in your foot** with local fast failover (e.g., create forwarding loops)?
Contribution: Very Robust Routing Possible with OpenFlow

Theorem: «Ideal» Forwarding Connectivity Possible

There exist algorithms which guarantee that packets always reach their destination, independently of the number and locations of failures, as long as the remaining network is connected.
Contribution: Very Robust Routing Possible with OpenFlow

Theorem: «Ideal» Forwarding Connectivity Possible

There exist algorithms which guarantee that packets always reach their destination, independently of the number and locations of failures, as long as the remaining network is connected.

Three algorithms:
- Modulo
- Depth-First
- Breadth-First

Essentially classic graph algorithms (routing, graph search) implemented in OpenFlow. Make use of tagging to equip packets with meta-information to avoid forwarding loops.
Contribution: Very Robust Routing Possible with OpenFlow

Theorem: «Ideal» Forwarding Connectivity Possible

There exist algorithms which guarantee that packets always reach their destination, independently of the number and locations of failures, as long as the remaining network is connected.

Three algorithms:
- Modulo
- Depth-First
- Breadth-First

Essentially classic graph algorithms (routing, graph search) implemented in OpenFlow. Make use of tagging to equip packets with meta-information to avoid forwarding loops.

Analysis of their complexity: maximum stretch (route length compared to ideal route), number of tags, number of OpenFlow rules.
Overview of Contributions

High-Level Algorithms

Algorithm 1 Algorithm MOD

Input: current node: \( v_i \), packet dest: \( d \), packet tag array: \( \{p(t,v)\}_{v \in \mathcal{V}} \)
Output: output port: \( o \)

1: if \( o = \text{default} \) then
2: \( o = \text{default} \)
3: else
4: if \( o = \text{out} \)
5: \( o = \text{out} \)
6: while \( o = \text{out} \) do
7: \( o = \text{out} \)
8: \( o = \text{out} \)
9: \( o = \text{out} \)
10: \( o = \text{out} \)
11: \( o = \text{out} \)
12: \( o = \text{out} \)
13: \( o = \text{out} \)
14: \( o = \text{out} \)
15: \( o = \text{out} \)
16: \( o = \text{out} \)
17: \( o = \text{out} \)
18: \( o = \text{out} \)
19: \( o = \text{out} \)
20: \( o = \text{out} \)

Algorithm 2 Algorithm DFS

Input: current node: \( v_i \), input port: \( i \), packet dest: \( d \), packet falloff global param: \( p(t,v) \), packet tag array: \( \{p(t,v)\}_{v \in \mathcal{V}} \)
Output: output port: \( o \)

1: if \( o = \text{default} \) then
2: \( o = \text{default} \)
3: else
4: if \( o = \text{out} \)
5: \( o = \text{out} \)
6: while \( o = \text{out} \) do
7: \( o = \text{out} \)
8: \( o = \text{out} \)
9: \( o = \text{out} \)
10: \( o = \text{out} \)
11: \( o = \text{out} \)
12: \( o = \text{out} \)
13: \( o = \text{out} \)
14: \( o = \text{out} \)
15: \( o = \text{out} \)
16: \( o = \text{out} \)
17: \( o = \text{out} \)
18: \( o = \text{out} \)
19: \( o = \text{out} \)
20: \( o = \text{out} \)

Flow-Table Implementations

Flow Table \( \Delta - 1 \)

Flow Table \( \Delta \)

Flow Table A (Start)

Flow Table B

Complexity Analysis

**Theorem 1.** MOD ensures data plane connectivity when:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Packet Memory</th>
<th>Message Count</th>
<th>Rules Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>( n \log d )</td>
<td>( n )</td>
<td>( O(n^d) )</td>
</tr>
<tr>
<td>BFS</td>
<td>( n \log d )</td>
<td>( 2(n) )</td>
<td>( O(n^d) )</td>
</tr>
<tr>
<td>BFS*</td>
<td>( k \log^2 + \log n )</td>
<td>( 2(n) )</td>
<td>( O(n^d) )</td>
</tr>
</tbody>
</table>

Related Work

- Borokhovich, OPODIS’13
- [1] Liu et al. NSDI’13
- Graph-search literature
Overview of Contributions

We expect that our algorithms scale up to 500-node networks (ignoring link capacities) (e.g., using our NoviKit 250 switches, with 32MB flow table space and full support for extended match fields).

High-Level Algorithms

Flow-Table Implementations

Complexity Analysis

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Packet Memory</th>
<th>Message count</th>
<th>Rules space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>nlogd</td>
<td>Exp(n)</td>
<td>O(n^d)</td>
</tr>
<tr>
<td>DFS</td>
<td>nlogd</td>
<td>2(E)</td>
<td>O(n^d)</td>
</tr>
<tr>
<td>BFS</td>
<td>nlogd</td>
<td>2kn</td>
<td>O(n^d)</td>
</tr>
<tr>
<td>BFS*</td>
<td>(logd+logn)</td>
<td>2kn</td>
<td>O(n^d(d+k))</td>
</tr>
</tbody>
</table>

Related Work

- Borokhovich, OPODIS’13
- [1] Liu et al. NSDI’13
- Graph-search literature
Overview of Contributions

High-Level Algorithms
Flow-Table Implementations

Complexity Analysis

Related Work
- Borokhovich, OPODIS’13
- [1] Liu et al. NSDI’13
- Graph-search literature

We expect that our algorithms scale up to 500-node networks (ignoring link capacities) (e.g., using our NoviKit 250 switches, with 32MB flow table space and full support for extended match fields).

Inherent tradeoffs between robustness and network load of failover without tagging.

Same objective: ideal connectivity. But their link-reversal algorithms not applicable to OpenFlow: require dynamic state at router.

Lower bounds with implications on optimality of our algorithms.
Conclusion

• Fast failover: example of a function that should be kept in the data plane

• Our result shows that non-trivial functions can be computed in the OpenFlow data plane!

• Our algorithms: may serve in compilers for higher-level languages, e.g., FatTire
Backup Slides
Complexity

• Today switches allow to match a few hundreds bits which can support a network of few dozens elements

• Some advanced experimental switches allow to match any offset in the packet thereby supporting huge networks of a few hundreds elements

• The ability to match every offset is expected to be supported by future versions of OpenFlow standard
OpenFlow Failover in a Nutshell

OpenFlow 101

- **OpenFlow** based on a pipeline of forwarding tables: each switch has multiple flow tables and a group table.
- Each **flow table** in the switch contains a set of flow entries; each flow entry consists of match fields, counters, and an ordered list of action buckets.
- **Groups** can be applied on a packet while processed.
- Each **action bucket** contains a set of actions to execute, and provides the ability to define multiple forwarding behaviors.
- The **group table** consists of multiple groups, where different groups can have different types, e.g., fast failover.

Without controller, an OpenFlow switch forwards according to:

- Static configuration
- Links status
- Packet header
- Input port

Each packet carries an Action set: empty at the start, updated while packet is processed, executed at the end.
Related Theory Literature

• Automata and Labyrinths [Budach 1978]
  • No finite automaton can explore all graphs
• Graph exploration by a finite automaton [Fraigniaud, Ilcinkas, Peerb, Pelcc, Peleg 2005]
  • $\Omega(\log n)$ memory for $n$ nodes graph
  • $\Theta(D \log d)$ for a graph with diameter $D$ and maximum degree $d$ (DFS is optimal).
• An Agent Exploration in Unknown Undirected Graphs with Whiteboards [Sudo, Baba, Nakamura, Ooshita, Kakugawa, Masuzawa 2010]
  • $O(\log d)$ memory in each node.
Algorithm 1 Algorithm MOD
Input: current node: $v_i$, packet dest: $d$, tags array:
\{$pkt.v_j\}_{j \in [n]}$
Output: new next hop: next
1: if no tag then \{same as $pkt.v_i = 0$\}
2: \quad next $\leftarrow$ default_route($i, d$)
3: else
4: \quad next $\leftarrow (pkt.v_i \mod \Delta_i) + 1$
5: \quad pkt.v_i $\leftarrow$ next
6: while $(v_i, v_{next})$ is failed do
7: \quad next $\leftarrow (pkt.v_i \mod \Delta_i) + 1$
8: \quad pkt.v_i $\leftarrow$ next
9: return next
**DFS Algorithm**

**Algorithm 2 Algorithm DFS**

**Input:** current node: $v_i$, packet dest: $d$, tags array: $T$

**Output:** new next hop: $next$

1. if $pkt.start = 0$ then
2. $next \leftarrow$ default_route($i,d$)
3. if $(v_i, next)$ failed then
4. $pkt.start \leftarrow 1$
5. $pkt.v_i.par \leftarrow in$
6. $next \leftarrow 1$
7. else
8. $next \leftarrow pkt.v_i.cur + 1$
9. if $next = \Delta_i + 1$ then
10. $next \leftarrow pkt.v_i.par$
11. goto 17
12. while $(v_i, next)$ failed or $next = pkt.v_i.par$ do
13. $next \leftarrow next + 1$
14. if $next = \Delta_i + 1$ then
15. $next \leftarrow pkt.v_i.par$
16. goto 17
17. $pkt.v_i.cur \leftarrow next$
18. return $next$

---

**Figure 1: DFS Tables illustration for node $i$.**
BFS Algorithm

Figure 2: BFS Tables illustration for node $i$. 
## Complexity

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Packet Memory</th>
<th>Message count</th>
<th>Rules space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>n log d</td>
<td>Exp(n)</td>
<td>O(n*d)</td>
</tr>
<tr>
<td>DFS</td>
<td>n log d</td>
<td>2</td>
<td>E</td>
</tr>
<tr>
<td>BFS</td>
<td>n log d</td>
<td>2kn</td>
<td>O(n*d)</td>
</tr>
<tr>
<td>BFS*</td>
<td>k(log d+log n)</td>
<td>2kn</td>
<td>O(n*(d+k))</td>
</tr>
</tbody>
</table>