Provable Data Plane Connectivity with Local Fast Failover
Introducing OpenFlow Graph Algorithms

Michael Borokhovich\textsuperscript{1}, Liron Schiff\textsuperscript{2}, Stefan Schmid\textsuperscript{3}
\textsuperscript{1} Ben Gurion University, Israel; \textsuperscript{2} Tel Aviv University, Israel; \textsuperscript{3} TU Berlin & T-Labs, Germany

ABSTRACT
Modern software-defined networks support the implementation of in-network failover mechanisms: mechanisms to quickly re-establish connectivity in the data plane without the interaction of the software controller. Interestingly, however, not much is known today about how to make use of these mechanisms.

This paper shows a very strong result: there exist failover implementations for OpenFlow that achieve a maximal robustness, in the sense that connectivity is always ensured as long as the underlying physical network is connected. In particular, we show that the problem of computing failover tables is related to graph search, and present three different algorithms achieving different tradeoffs, in terms of the number of required failover rules, the number of tags, as well as the resulting path lengths.

Our work can also be seen as a first attempt to implement classic graph algorithms in OpenFlow.

CATEGORIES AND SUBJECT DESCRIPTORS
C.2.2 [Computer-Communication Networks]: Network Protocols

GENERAL TERMS
Algorithms, Performance

KEYWORDS
Software-Defined Networking, Graph Exploration

1. INTRODUCTION

Software-defined network architectures distinguish between the data plane, consisting of the forwarding switches, and the control plane, consisting of one or multiple software controllers. Out-sourcing the control over the data plane elements to a logically centralized software controller simplifies the network management, and introduces new flexibilities and optimization opportunities.

However, indirections via the control plane can come at a cost, both in terms of communication overhead as well as latency. Indeed, the reaction time to data plane events in the control plane can be orders of magnitude slower compared to a direct reaction in the network. Especially for the recovery of failures, a slow reaction is problematic, and the indirect via the controller out of question. Accordingly, recent SDN versions support the implementation of local fast failover mechanisms: mechanisms that handle the failover in the data plane directly. For example, OpenFlow, the predominant SDN protocol today, supports conditional rules whose forwarding behavior depends on the local state of the switch; if links fail, an alternative forwarding port will be chosen according to these pre-defined rules, and without any controller interaction.

However, surprisingly little is known today about how to implement such fast-failover mechanisms, that is, how to algorithmically compute the conditional failover rules which ensure a high robustness in different failure scenarios. Moreover, hardly anything is known about the achievable robustness as well as the cost tradeoffs of different mechanisms, e.g., in terms of the number of required conditional rules or the resulting path lengths. This paper aims to close this gap.

Our Contribution. The main contribution of this paper are local-fast failover algorithms that guarantee a very strong notion of data plane robustness: connectivity is preserved under arbitrary link failures, subject to the (weakest and necessary) condition that the underlying physical network be connected. Interestingly, these algorithms can directly be implemented within modern OpenFlow standards. The mechanisms could hence also be used as part of a compiler for a higher-level programming language such as FatTire\textsuperscript{11}.

Our contribution can also be seen as a first study on how to introduce graph algorithms into OpenFlow. The first algorithm, Mod, is based on a simple modulo strategy to compute failover paths; the second algorithm, DFS, is based on a depth-first search strategy; and the third algorithm, BFS, is based on a breadth-first search strategy. We investigate the cost of our algorithms, in terms of the tag complexity (the number of used tags), the rule complexity (the number of conditional failover rules), as well as the resulting path lengths, and show that different tradeoffs can be achieved.

2. BACKGROUND AND MODEL

While our work can also be seen in a more general context, for the sake of concreteness, we will present our results using the concepts and terminology of recent versions of OpenFlow, the standard SDN protocol today. In general, the OpenFlow
protocol is based on a match-action concept: OpenFlow switches store rules (installed by the controller) consisting of a match and an action part. A packet matched by a certain rule will be subject to the associated action. For example, an action can define a port to which the matched packet should be forwarded. An action can also add or change a tag of a packet (a certain part in the packet header). Tagging is a useful technique to equip packet instances with meta-information; in fact, it is known that without tagging, failover mechanisms are severely limited. We will assume that each switch has a group table: a forwarding table whose rules include an ordered list of action buckets. Each action bucket contains a set of actions to execute, and provide the ability to define multiple forwarding behaviors. Each bucket in a fast failover type table (short: FF table), is associated with a parameter that determines whether the bucket is live; a switch will always forward traffic to the first live bucket. As the parameter to determine liveness, the programmer either specifies an output port or a group number (to allow several groups to be chained together).

Note that these failover tables need to be allocated before the failure(s) happen. We will sometimes refer to the initial network (before the failures occur) as \( G_0 \), and to the remaining “sub-graph” after the failure as \( G' \). The challenge of how to compute robust failover tables for \( G_0 \) without knowing the actual failure scenario, i.e., \( G' \), is addressed in this paper. Specifically, we will present an algorithm that ensures provable forwarding connectivity in \( G' \), i.e., each packet eventually reaches its destination, as long as \( G' \) is connected. Observe that this is a non-trivial result, given the limited configuration possibilities OpenFlow offers. In the following, we will often refer to the network switches as nodes.

Local fast failover mechanisms are often used together with additional, slower failover mechanisms. As such, the main priority of the failover is to quickly re-establish connectivity. Optimality e.g., in terms of path lengths or load balancing, plays a secondary role: paths can later be improved via the control plane. This two-stage approach is balancing, plays a secondary role: paths can later be improved via the control plane. This two-stage approach is appealing as it combines the advantages of both worlds. Nevertheless, fast failover mechanisms may come with different cost-performance tradeoffs, and we in this paper will study the following metrics.

1. **Rule complexity**: How many additional rules (per node) are required to implement the failover mechanism?

2. **Tag complexity**: How much tag space is needed in the packets? (We will assume that a protocol requiring \( x \) tags in the worst-case needs \( \Theta(\log x) \) bits tag space to store the different tags.)

3. **Route length**: How long is the longest forwarding path in \( G' \)? We are particularly interested in how much longer the path can be compared to the shortest paths in \( G' \).

In the following, \( n \) will refer to the number of nodes in the network, \( \Delta \) to the maximal node degree in \( G_0 \), and \( \delta \) to the maximal distance between two nodes in \( G' \). We will use subscripts, e.g., \( \Delta_i \), to denote the corresponding value for the given node \( v_i \). Finally, we will refer to the intended destination of a packet by \( d \), and assume that all switches in the network initially have a rule defining the port to which a packet destined to \( d \) should be forwarded. If the corresponding link is down, the failover mechanism kicks in.

### 3. CONNECTIVITY MECHANISMS

This section investigates robust failover mechanisms providing provable data plane connectivity. To get acquainted with the problem, we will first describe a simple solution called \( \text{Mod} \). Subsequently, we present two algorithms based on graph-search techniques, called DFS and BFS, exploring different points in the tradeoff space. While we will present the idea and pseudo-code directly in the text, to improve readability, some details of the OpenFlow failover tables appear in the Appendix resp. in our technical report [13].

#### 3.1 The Modulo Algorithm

The first algorithm \( \text{Mod} \) is simple: if a link is failed, an alternative forwarding port is used in a round-robin fashion. For this purpose, for switch \( v_i \) with \( \Delta_i \) ports, we define an arbitrary order on the outgoing ports \( p_0 < p_2 < \ldots < p_{\Delta_i-1} \). If a packet cannot be forwarded on a port \( p_i \) as planned (due to a link failure), the next port \( p_{i+1 \mod \Delta} \) is chosen in a modulo manner. This alone however is not sufficient to provide connectivity, as the network reached via \( p_{i+1 \mod \Delta} \) may not include \( d \). Hence, the packet may be returned to a given switch multiple times, in the search for an alternative path.

Accordingly, \( \text{Mod} \) maintains meta-information in the packet header, using tagging. Concretely, for each node \( v \) (a switch), a set of \( O(\log \Delta) \) bits are reserved in the tag space, to implement a modulo counter \( c(v) \): whenever a packet with counter \( c(v) \) arrives at switch \( v \), the packet is forwarded to the port \( c(v) \mod \Delta \), and the counter for this switch is incremented \((c(v) \leftarrow c(v) + 1)\).

Algorithm \( \text{Mod} \) is summarized in Algorithm [4] in pseudo-code. We use the convention that \( \text{tag} = 0 \) denotes no tag; positive tags indicate that the failover mechanism is active. In the absence of failures, node \( v_i \) should forward the packet destined to \( d \) to port \( \text{default}_\text{route}(i, d) \).

Algorithm \( \text{Mod} \) can be implemented with the following tables, see Appendix and [13] for details. The modulo computations are realized using \( 2\Delta \) flow tables. The first table, Table 0, tries to forward untagged packets via the default route (stored in the group table). Packets with tag \( x > 0 \) are sent to the corresponding Table \( x \), where the next forwarding port is tried; otherwise (sent bit \( sb = 0 \)), the counter is incremented.

---

### Algorithm 1 Algorithm \( \text{Mod} \)

**Input:** current node: \( v_i \), packet dest: \( d \), packet tag array: \( \{\text{pkt}.v_j\}_{j\in[n]} \)

**Output:** output port: \( \text{out} \)

1. **if** no \text{tag} **then** \{same as \text{pkt}.\text{v} = 0\}
2. \text{out} \leftarrow \text{default}_\text{route}(i, d)
3. **else**
   4. \text{out} \leftarrow (\text{pkt}.\text{v} \mod \Delta_i) + 1
   5. \text{pkt}.\text{v} \leftarrow \text{out}
5. **while** \text{out} failed **do**
   6. \text{out} \leftarrow (\text{pkt}.\text{v} \mod \Delta_i) + 1
   7. \text{pkt}.\text{v} \leftarrow \text{out}
9. return \text{out}
Note that if the remaining network $G'$ is connected, a packet will eventually reach its destination $d$. The proof is by induction over the distance from $d$: a neighbor $v$ of $d$ will eventually forward a packet to $d$ (at most after $\Delta - 1$ other attempts). Similarly, a neighbor $v'$ of $v$ in $G'$ will eventually forward the packet to $v$. And so on, until the entry port of the packet is reached.

Overall, a linear number of rules are needed to implement this scheme, and for each switch $v_i$ we need to store a counter assuming values between 0 and $\Delta_i$. Hence, we have the following result.

**Theorem 1.** Mod ensures data plane connectivity whenever the data plane is physically connected. Mod has rule complexity $O(n)$ and tag complexity $O(n \cdot \log \Delta)$.

While simple, the Mod comes with a big drawback: in the worst-case, a packet may have to travel far before reaching its destination $d$.

### 3.2 The Depth-First Algorithm

There exist algorithms that ensure connectivity with shorter paths. In particular, forwarding a packet $pkt$ to its destination $d$ in the unknown graph $G'$ can be seen as a graph search problem. Accordingly, in the following we show how to realize a depth-first search in OpenFlow.

Upon a link failure, the DFS algorithm explores alternative routes in a depth-first fashion, implicitly constructing a spanning tree. Towards this goal, for each node $v_i$, we reserve a certain part of the packet header $pkt.v_i.par$, to store the parent $p(v_i)$ of $v_i$, the node from which the packet was received the first time (indicated by the incoming port $in$).

Algorithm DFS is summarized in Algorithm 2 in pseudo-code. Here, $pkt.v_i = 0$ denotes that no parent has been selected yet. A node tries to forward the packet to each neighbor, and only if this fails, returns the packet to the parent.

How can this graph search algorithm be implemented in OpenFlow? Figure 1 gives an overview of the required tables. Overall, there are six types of tables. Table A first tries to forward the packet along the default route (we have Group 0,j for each destination j), otherwise the packet is sent to the tables A, B, C which are used to establish the spanning tree; if no neighbor reaches the destination, the packet is sent back to the parent (Table E). Finally, for each port $x$, Table x is used to try and forward to $x$. The main OpenFlow tables for this algorithm appear in the Appendix.

The number of hops traveled by a packet is upper bounded by $O(n)$, which improves significantly over Mod: a node is visited at most twice. The tag complexity is $O(n \log \Delta)$: a parent pointer has to be stored in the packet for each node. Finally, per node, $O(\Delta)$ rules are required.

**Theorem 2.** DFS ensures data plane connectivity whenever the data plane is physically connected. DFS yields paths of length at most $O(n)$, and has rule complexity $O(\Delta)$ and tag complexity $O(n \cdot \log \Delta)$.

### 3.3 The Breadth-First Algorithm

The paths computed by the DFS algorithm are still inefficient in the sense that the actual length of the route taken by a packet does not depend on $\delta_i$, the shortest distance to the destination $d$ in the remaining network $G'$. Naturally, a breadth-first search algorithm has the potential to find more competitive paths.

It turns out that breadth-first search algorithms can also be implemented in OpenFlow. In the following, we present a solution called BFS. Algorithm BFS is summarized in Algorithm 3. BFS controls the radius at which the destination $d$ is searched in each phase in a clever way: each node $v$ (other than the source) behaves as a leaf, and the first time reached, $v$ returns the packet right back to its parent; on any other occasion $v$ behaves as a DFS node, trying all his neighbors and then returning to the parent. Again, a spanning tree is constructed, and tags are used to store the discovered parents. The full tables can be found at 13.

The tag complexity of BFS is $O(n \log \Delta)$, due to the parent pointers, and the number of rules is $O(\Delta)$: note that in our 18:

![Figure 1: Overview of DFS tables for node i.](image-url)
Table A

if start = 0 and dst = i:
  apply Group 0
  goto Table 1

Table B

if cur[i] = 0 & par[i] = 0:
  set par[i] = in, goto Table C
if in = cur[i] & cur[i] = 0:
  goto Table C
if in = cur[i] = A:
  goto Table C
if in = cur[i]
  return packet to in-port

Table C

forward packet to par[i]

Group 0,j

try default route for dest:
  if succeeded:
    set sent = 1

Table x

if not sent:
  apply Group x
  goto Table x+1
if sent:
  Drop

Group x

set cur[i] = x
set start = 1
try forward to x

Figure 2: Overview of BFS tables for node i.

Algorithm 3 Algorithm BFS

Input: current node: vi, input port: in, packet dest: d, packet failover global params: pkt start, packet tag array: \{ pkt,vi \}_{i \in \mathcal{N}}

Output: output port: out
1. if pkt start = 0 then
2.   out = default_route(i,d)
3. if out failed then
4.   pkt start ← 1
5. pkt.v.par ← 0
6. out ← 1
else
8. if pkt.v.cur = 0 and pkt.v.par = 0 then
9.   pkt.v.par ← in
10. out ← in
11. return output
12. if pkt.v.cur ≠ in and pkt.v.par ≠ in then
13.   out ← in
14. return output
15. out ← pkt.v.cur + 1
16. if out = Δi + 1 then
17.   out ← pkt.v.par
18. pkt.v.cur ← 0
19. goto \[27]\n20. while out failed or out = pkt.v.par do
21.   out ← out + 1
22. if out = Δi + 1 then
23.   out ← pkt.v.par
24. pkt.v.cur ← 0
25. goto \[27]\n26. pkt.v.cur ← out
27. if out = 0 then
28.   out ← 1
29. while out failed do
30.   out ← out + 1
31. if out = Δi + 1 then
32. Drop
33. pkt.v.cur ← out
34. return output

3.4 Remarks

We conclude the section with some remarks.

How optimal are the presented algorithms?

It turns out that there is no algorithm that ensures path
length less than \( O(n) \), independently of \( \delta' \). This follows
from literature on agent-based graph exploration, e.g., \[10\],
where data can be stored in the visited nodes (referred to
as “putting pebbles” or “using white-boards”). While our
algorithms can store information not only in the nodes but
also in the packets (tags), our model is equivalent to an
algorithm that has only pebbles. Hence, our DFS algorithm
achieving \( O(n \log \Delta) \) tag complexity with only two traversals
per edge, is competitive with known algorithms.

What if \( G' \) is not connected?

If \( G' \) is disconnected, certain destinations necessarily be-
come unreachable. Nevertheless, it is desirable that local
failover mechanisms “behave well” in such settings. In partic-
ular: they should discover the disconnection and terminate.

There is a simple technique to render our algorithms robust
to such failures. We simply need to add two bits per every
node to the tag array in the packet’s header (i.e., the tag
complexity is increased by \( 2n \) additional bits). The first bit,
\( b_1 \), will be raised by a node on the first time it is visited.
The second bit, \( b_2 \), will be raised by a node once it “tried”
sending the packet to all of its neighbors. Each node, upon
packet reception, checks the \( b_1, b_2 \) bits of all the nodes. Any
node which detects that all the visited nodes also “tried” all
their neighbors, will discard the packet. The above condition
implies that all possible paths in the connected component
were traversed by the packet.

For the DFS algorithm, the stopping condition can also be
implemented by simply checking if the DFS root has “tried”
all its neighbors. Notice, in Algorithm \[2\], the DFS root node
will have value 0 for its parent. Thus, we can check the out
value before returning it in Line \[29\]. Once the out value is 0,
the packet will be discarded.

4. RELATED WORK

Outages due to link failures are not uncommon, \[5, 8\] and
robust routing mechanisms are provided by many networks
protocols today. For example, robust Multiprotocol Label
Switching (MPLS) supports local and global path protection
to compute shortest backup paths around an outage area \[9, 12\],
where “shortest” often refers to congestion \[12, 15\]. Alter-
native solutions to make routing more resilient rely on
special header bits (e.g., to determine when to switch from
primary to backup paths, as in MPLS Fast Reroute \[8\], or to
encode failure information to make failure-aware forwarding
decisions \[4, 7\], or on the fly table modifications \[6\].
Recently, Feigenbaum et al. [2] made a first step towards a better theoretical understanding of resilient routing tables. The authors prove that routing tables can provide guaranteed resilience (i.e., loop-freeness) against a single failure, when the network remains connected. In [5], Liu et al. introduce the notion of ideal forwarding-connectivity: for any failure scenario where the network remains physically connected, the forwarding choices should guide packets to their intended destinations. Ideal forwarding-connectivity is the strongest resilience that can be provided by the data plane, and is equivalent to the guarantees given by our algorithms. In contrast to [5], we in this paper study failover mechanisms for software-defined networks. In particular, the failover scheme in [5] is based on link-reversal algorithms [2], which has the limitation that it requires to maintain dynamic state in the routers and is not applicable to OpenFlow. In this paper, we have presented algorithms without this limitation, and also analytically studied the corresponding complexity tradeoffs. The paper closest to ours is [1], which studies SDN local fast failover mechanisms without tagging. The authors prove that without tagging, local fast failover mechanisms are severely limited, in the sense that connectivity cannot be provided already for a small number of link failures, and even if the physical network is still well-connected. Moreover, the paper shows a tradeoff between robustness and load-balancing. In contrast, we in this paper have presented algorithms that maintain connectivity under arbitrary link failures.

5. CONCLUSION
This paper presented the first fast failover mechanism for OpenFlow networks which provably guaranteed data plane connectivity whenever this is possible. Such a mechanism can be attractive in many scenarios, e.g., in order to find in-band paths to the controller, or for network troubleshooting.

We expect that our algorithms scale up to 500-node networks using switches like our NoviKit 250 switch, with 32MB flow table space and full support for extended match fields. In such networks, the packet header overhead can reach 1KB which limits the size of the data section of the packet to less than 0.5KB. For scenarios where high data transfer rates are crucial, a more compact version is possible, using only about 600 bits of packet header, which supports up to 40 nodes.

Acknowledgments. Michael Borokhovich is supported in part by the Israel Science Foundation (grant 1549/13). Liron Schiff is supported by the European Research Council (ERC) Starting Grant no. 259085 and by the Israel Science Foundation Grant no. 1386/11. Stefan Schmid is supported by the EIT ICT project Mobile SDN.

6. REFERENCES

APPENDIX
A. MOD FAILOVER TABLES

<table>
<thead>
<tr>
<th>Flow Table A (Start)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
</tr>
<tr>
<td>pkt.v,cur</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flow Table B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
</tr>
<tr>
<td>pkt.v,cur</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Δi−1</td>
</tr>
<tr>
<td>Δi</td>
</tr>
</tbody>
</table>

Table 1: Mod: Flow Tables for switch i.
Table 2: Mod: Group Table of switch $i$.

<table>
<thead>
<tr>
<th>Group</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gr 0.1</td>
<td>$(sb ← 1, \text{Fwd Route(1)})$</td>
</tr>
<tr>
<td>Gr 0.2</td>
<td>$(sb ← 1, \text{Fwd Route(2)})$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Gr 0.n</td>
<td>$(sb ← 1, \text{Fwd Route(n)})$</td>
</tr>
<tr>
<td>Gr 1</td>
<td>$(sb ← 1, T(i) ← 1, \text{Fwd 1})$</td>
</tr>
<tr>
<td>Gr 2</td>
<td>$(sb ← 1, T(i) ← 2, \text{Fwd 2})$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Gr $\Delta_i$</td>
<td>$(sb ← 1, T(i) ← \Delta_i, \text{Fwd } \Delta_i)$</td>
</tr>
</tbody>
</table>

Table 3: Mod: Flow Tables for switch $i$.

B. DFS FAILOVER TABLES

Table 4: DFS: Group Table of switch $i$.

<table>
<thead>
<tr>
<th>Group</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gr 0.1</td>
<td>$(sb ← 1, \text{Fwd Route(1)})$</td>
</tr>
<tr>
<td>Gr 0.2</td>
<td>$(sb ← 1, \text{Fwd Route(2)})$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Gr 0.n</td>
<td>$(sb ← 1, \text{Fwd Route(n)})$</td>
</tr>
<tr>
<td>Gr 1</td>
<td>$(sb ← 1, pkt.v_i.cur ← 1, pkt.start ← 1, \text{Fwd 1})$</td>
</tr>
<tr>
<td>Gr 2</td>
<td>$(sb ← 1, pkt.v_i.cur ← 2, pkt.start ← 1, \text{Fwd 2})$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Gr $\Delta_i$</td>
<td>$(sb ← 1, pkt.v_i.cur ← \Delta_i, pkt.start ← 1, \text{Fwd } \Delta_i)$</td>
</tr>
</tbody>
</table>

Table 5: DFS: Flow Tables of switch $i$.

<table>
<thead>
<tr>
<th>Group</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gr 1</td>
<td>$(sb ← 1, T(i) ← 1, \text{Fwd 1})$</td>
</tr>
<tr>
<td>Gr 2</td>
<td>$(sb ← 1, T(i) ← 2, \text{Fwd 2})$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Gr $\Delta_i$</td>
<td>$(sb ← 1, T(i) ← \Delta_i, \text{Fwd } \Delta_i)$</td>
</tr>
</tbody>
</table>

Table 6: DFS Flow Tables of switch $i$.