Provable Data Plane Connectivity with Local Fast Failover

Introducing OpenFlow Graph Algorithms

Michael Borokhovich
Ben Gurion University
Israel
borokhom@cse.bgu.ac.il

Liron Schiff
Tel Aviv University
Israel
schiffli@post.tau.ac.il

Stefan Schmid
TU Berlin & T-Labs
Germany
stefan@net.t-labs.tu-berlin.de

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.

HotSDN'14, August 22, 2014, Chicago, IL, USA.

ACM 978-1-4503-2989-7/14/08 ...$15.00.
http://dx.doi.org/10.1145/2620728.2620746 .

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. [5] Especially for the recovery of failures, a slow reaction is problematic, and the indirection 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* [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. [1] 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 attractive 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, Δ to the maximal node degree in G_0, and δ' to the maximal distance between two nodes in G'. We will use subscripts, e.g., Δ_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.

Algorithm 1 Algorithm MOD

Input: current node: v_i, packet dest: d, packet tag array: $\{pkt.v_j\}_{j\in[n]}$
Output: output port: *out*

```
1: if no tag then {same as pkt.v_i = 0}
2:    out ← default_route(i, d)
3: else
4:    out ← (pkt.v_i mod Δ_i) + 1
5:    pkt.v_i ← out
6:    while out failed do
7:       out ← (pkt.v_i mod Δ_i) + 1
8:       pkt.v_i ← out
9: return out
```

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 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 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 Δ_i ports, we define an arbitrary order on the outgoing ports $p_0 \prec p_2 \prec \ldots \prec p_{\Delta_i-1}$. 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, 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 MOD is summarized in Algorithm 1 in pseudo-code. We use the convention that $tag = 0$ 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 $default\_route(i, d)$.

Algorithm MOD can be implemented with the following tables, see Appendix and [13] for details. The modulo computations are realized using $2\Delta_i$ 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.

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 Δ_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 pseudocode. 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* $O(n \log \Delta)$.

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 δ', 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.

Algorithm 2 Algorithm DFS

Input: current node: v_i, input port: in, packet dest: d, packet failover global params: $pkt.start$, packet tag array: $\{pkt.v_j\}_{j \in [n]}$
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_i.par ← 0
6:       out ← 1
7: else
8:    if pkt.v_i.cur = 0 then
9:       pkt.v_i.par ← in
10:   out ← pkt.v_i.cur + 1
11:   if out = Δ_i + 1 then
12:      out ← pkt.v_i.par
13:      goto 19
14: while out failed or out = pkt.v_i.par do
15:   out ← out + 1
16:   if out = Δ_i + 1 then
17:      out ← pkt.v_i.par
18:      goto 19

19: pkt.v_i.cur ← out
20: return out
```

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

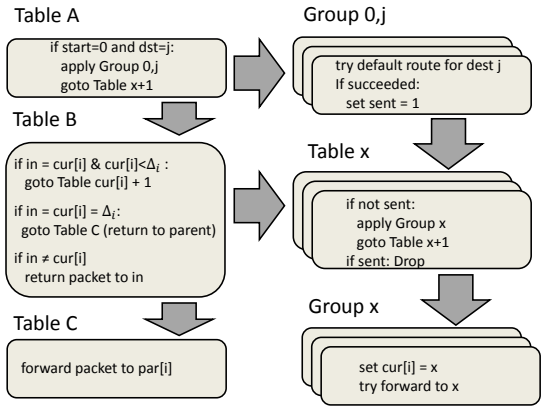

Figure 1: Overview of DFS tables for node i.

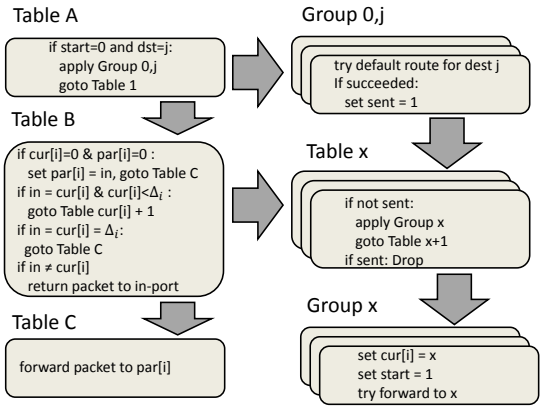

Figure 2: Overview of BFS tables for node i.

Algorithm 3 Algorithm BFS

Input: current node: v_i, input port: in, packet dest: d, packet failover global params: $pkt.start$, packet tag array: $\{pkt.v_j\}_{j\in[n]}$
Output: output port: out

1: **if** $pkt.start = 0$ **then**
2: $out = default\_route(i, d)$
3: **if** out failed **then**
4: $pkt.start \leftarrow 1$
5: $pkt.v_i.par \leftarrow 0$
6: $out \leftarrow 1$
7: **else**
8: **if** $pkt.v_i.cur = 0$ **and** $pkt.v_i.par = 0$ **then**
9: $pkt.v_i.par \leftarrow in$
10: $out \leftarrow in$
11: **return** out
12: **if** $pkt.v_i.cur \neq in$ **and** $pkt.v_i.par \neq in$ **then**
13: $out \leftarrow in$
14: **return** out
15: $out \leftarrow pkt.v_i.cur + 1$
16: **if** $out = \Delta_i + 1$ **then**
17: $out \leftarrow pkt.v_i.par$
18: $pkt.v_i.cur \leftarrow 0$
19: **goto** 27

20: **while** out failed **or** $out = pkt.v_i.par$ **do**
21: $out \leftarrow out + 1$
22: **if** $out = \Delta_i + 1$ **then**
23: $out \leftarrow pkt.v_i.par$
24: $pkt.v_i.cur \leftarrow 0$
25: **goto** 27

26: $pkt.v_i.cur \leftarrow out$

27: **if** $out = 0$ **then**
28: $out \leftarrow 1$
29: **while** out failed **do**
30: $out \leftarrow out + 1$
31: **if** $out = \Delta_i + 1$ **then**
32: Drop
33: $pkt.v_i.cur \leftarrow out$
34: **return** out

implementation, no extra rules are needed for the counting. Finally, the path length depends on the number of nodes in the neighborhood of the source at distance at most δ'. Let Γ_r denote the number of nodes at distance at most r from the source node s, and define $D = \sum_{r=1}^{\delta'} \Gamma_r(s)$, then at most $O(D)$ nodes are visited by a packet: the traffic remains more local.

THEOREM 3. BFS *ensures data plane connectivity whenever the data plane is physically connected.* BFS *has rule complexity* $O(n \log \Delta)$*, tag complexity* $O(\Delta)$*, and path lengths are at most* $O(D)$*, where* D *is the sum of nodes at distances up to* δ'*.*

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 $\Omega(n)$, independently of δ'. 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 become unreachable. Nevertheless, it is desirable that local failover mechanisms "behave well" in such settings. In particular: 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 20. 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, 14], where "shortest" often refers to congestion [12, 15]. Alternative 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* [9], 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 [3], 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

[1] M. Borokhovich and S. Schmid. How (not) to shoot in your foot with sdn local fast failover: A load-connectivity tradeoff. In *Proc. 17th International Conference on Principles of Distributed Systems (OPODIS)*, 2013.

[2] J. F. et al. Ba: On the resilience of routing tables. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, pages 237–238, 2012.

[3] E. Gafni and D. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *Communications, IEEE Transactions on*, 29(1):11–18, Jan 1981.

[4] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving convergence-free routing using failure-carrying packets. In *Proc. SIGCOMM*, pages 241–252, 2007.

[5] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker. Ensuring connectivity via data plane mechanisms. In *Proce. 10th USENIX NSDI*, pages 113–126, 2013.

[6] J. Liu, B. Yan, S. Shenker, and M. Schapira. Data-driven network connectivity. In *Proc. HotNets*, pages 8:1–8:6, 2011.

[7] S. S. Lor, R. Landa, and M. Rio. Packet re-cycling: eliminating packet losses due to network failures. In *Proc. HotNets*, pages 2:1–2:6, 2010.

[8] D. Madory. Renesys blog: Large outage in pakistan. *Blog*, 2011.

[9] P. Pan, G. Swallow, and A. Atlas. Fast reroute extensions to RSVP-TE for LSP tunnels. In *RFC 4090*, 2005.

[10] D. Peleg. *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[11] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proc. HotSDN*, pages 109–114, 2013.

[12] H. Saito and M. Yoshida. An optimal recovery LSP assignment scheme for MPLS fast reroute. In *Proc. NETWORKS*, 2002.

[13] Tech Report. http://www.net.t-labs.tu-berlin.de/∼stefan/hotsdn14tr.pdf. Technical report, pdf, 2014.

[14] J.-P. Vasseur, M. Pickavet, and P. Demeester. *Network Recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS*. Morgan Kaufmann Publishers Inc., 2004.

[15] D. Wang and G. Li. Efficient distributed bandwidth management for MPLS fast reroute. *IEEE/ACM Trans. Netw.*, 2008.

APPENDIX

A. MOD FAILOVER TABLES

Flow Table A (Start)

Match		Instructions
$pkt.v_i.cur$	dst	
0	1	Gr 0.1, Table 1
0	2	Gr 0.2, Table 1
...	...	...
0	n	Gr 0.n, Table 1
$*$	$*$	Table B

Flow Table B

Match	Instructions
$pkt.v_i.cur$	
1	Table 2
2	Table 3
...	...
$\Delta_i - 1$	Table Δ_i
Δ_i	Table 1

Table 1: Mod: Flow Tables for switch i.

Group	Actions
Gr 0.1	$\langle sb \leftarrow 1, \text{Fwd } Route(1)\rangle$
Gr 0.2	$\langle sb \leftarrow 1, \text{Fwd } Route(2)\rangle$
...	...
Gr 0.n	$\langle sb \leftarrow 1, \text{Fwd } Route(n)\rangle$
Gr 1	$\langle sb \leftarrow 1, T(i) \leftarrow 1, \text{Fwd } 1\rangle$
Gr 2	$\langle sb \leftarrow 1, T(i) \leftarrow 2, \text{Fwd } 2\rangle$
...	...
Gr Δ_i	$\langle sb \leftarrow 1, T(i) \leftarrow \Delta_i, \text{Fwd } \Delta_i\rangle$

Table 2: Mod: Group Table of switch i.

Flow Table 1

Match *sb*	Instructions
0	Gr 1, Table 2
1	Drop

...

Flow Table Δ_i

Match *sb*	Instructions
0	Gr Δ_i, Table $\Delta_i + 1$
1	Drop

...

Flow Table $2\Delta_i - 1$

Match *sb*	Instructions
0	Gr $\Delta_i - 1$, Drop
1	Drop

Table 3: Mod: Flow Tables for switch i.

B. DFS FAILOVER TABLES

Group	Actions
Gr 0.1	$\langle sb \leftarrow 1, \text{Fwd } Route(1)\rangle$
Gr 0.2	$\langle sb \leftarrow 1, \text{Fwd } Route(2)\rangle$
...	...
Gr 0.n	$\langle sb \leftarrow 1, \text{Fwd } Route(n)\rangle$
Gr 1	$\langle sb \leftarrow 1, pkt.v_i.cur \leftarrow 1, pkt.start \leftarrow 1, \text{Fwd } 1\rangle$
Gr 2	$\langle sb \leftarrow 1, pkt.v_i.cur \leftarrow 2, pkt.start \leftarrow 1, \text{Fwd } 2\rangle$
...	...
Gr Δ_i	$\langle sb \leftarrow 1, pkt.v_i.cur \leftarrow \Delta_i, pkt.start \leftarrow 1, \text{Fwd } \Delta_i\rangle$

Table 4: DFS: Group Table of switch i.

Flow Table 1

Match *sb*	Instructions
0	Gr 1, Table 2
1	Drop

...

Flow Table $\Delta_i - 1$

Match *sb*	Instructions
0	Gr $\Delta_i - 1$, Table Δ_i
1	Drop

Flow Table Δ_i

Match *sb*	Instructions
0	Gr Δ_i,Table 0.2 (Send-Parent)
1	Drop

Table 5: DFS: Flow Tables of switch i.

Flow Table A (Start)

Match		Instructions
pkt.start	*dst*	
0	1	Gr 0.1, Table 1
0	2	Gr 0.2, Table 1
...	...	...
0	n	Gr 0.n, Table 1
$*$	$*$	Table B

Flow Table B

Match			Instructions
in	*pkt.v_i.cur*	*pkt.v_i.par*	
1	0	$*$	*pkt.v_i.par* $\leftarrow$ *in*, Table 2
1	1	2	Table 3
2	2	3	Table 4
3	3	4	Table 5
...	...	...	...
$\Delta_i - 2$	$\Delta_i - 2$	$\Delta_i - 1$	Table Δ_i
$\Delta_i - 1$	$\Delta_i - 1$	Δ_i	Table C
$*$	0	$*$	*pkt.v_i.par* $\leftarrow$ *in*, Table 1
1	1	$*$	Table 2
2	2	$*$	Table 3
3	3	$*$	Table 4
...	...	...	...
$\Delta_i - 1$	$\Delta_i - 1$	$*$	Table Δ_i
Δ_i	Δ_i	$*$	Table C
1	$*$	$*$	Fwd 1
2	$*$	$*$	Fwd 2
...	...	...	...
Δ_i	$*$	$*$	Fwd Δ_i

Flow Table C (Send-Parent)

Match *pkt.v_i.par*	Instructions
1	Fwd 1
2	Fwd 2
...	...
Δ_i	Fwd Δ_i

Table 6: DFS Flow Tables of switch i.