Backup rules in Software-Defined Networks

van Adrichem, Niels; Muhammad Iqbal, Farabi; Kuipers, Fernando

DOI
10.1109/NFV-SDN.2016.7919495

Publication date
2016

Document Version
Accepted author manuscript

Published in
IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)

Backup rules in Software-Defined Networks

Niels L. M. van Adrichem, Farabi Iqbal and Fernando A. Kuipers
Network Architectures and Services, Delft University of Technology
Mekelweg 4, 2628 CD Delft, the Netherlands
{N.L.M.vanAdrichem, M.A.F.Iqbal, F.A.Kuipers}@tudelft.nl

***Abstract*—The past century of telecommunications has shown that failures in networks are prevalent. Failure recovery processes are therefore needed. Failure recovery is mainly influenced by (1) detection of the failure, and (2) circumvention of the detected failure. However, especially in SDNs where controllers recompute network state reactively, this leads to high delays. Hence, next to primary rules, backup rules should be installed in the switches to quickly detour traffic once a failure occurs. In this work, we propose algorithms for computing an all-to-all primary and backup network forwarding configuration that is capable of circumventing link and node failures. After initial recovery, we recompute network configuration to guarantee protection from future failures. Our algorithms use packet-labeling to guarantee correct and shortest detour forwarding and are able to discriminate between link and node failures. The computational complexity of our solution is comparable to that of all-to-all shortest paths computations. Our experimental evaluation shows that network configuration complexity decreases significantly compared to classic disjoint paths computations. Finally, we provide a proof-of-concept OpenFlow controller in which our proposed configuration is implemented, demonstrating that it readily can be applied in production networks.**

I. Introduction

As society heavily depends on modern telecommunication networks, much has been done to prevent network failure, e.g., by improving the equipment environment and physical aspects of the material. However, the past century of telecommunications shows that network components still fail regularly [1].

In connection-oriented networks, e.g. wavelength-routed networks, network service interruptions due to the failure of network nodes or links can often be prevented by assigning at least two disjoint paths from the source node to the destination node of each connection [2]. When the primary path of a network connection fails, the connection can be reconfigured to use its backup path instead. Although finding a pair of (min-sum) disjoint paths from a source node to a destination node is polynomially solvable, the returned paths may be substantially longer than the shortest possible path between the nodes.

Packet-switched networks, e.g., Ethernet or IP networks, have no connection status, since packets are forwarded in a hop-by-hop manner through local inspection of headers at each router it traverses. Though using disjoint paths is possible in packet-switched networks through end-to-end liveliness detection monitoring schemes (such as Bidirectional Forwarding Detection (BFD) [3], Ethernet OAM/CFM [4] or IP Fast Reroute [5]), the approach is more constrained than in connection-oriented networks.

In packet-switched networks, each intermediate node along the primary path has the capability of forwarding packets through another link interface when necessary. Furthermore, after packets have been rerouted past the failure, packets are directed to the shortest remaining path towards the destination, possibly by following the remainder of the (initial) primary path that is unaffected by the failure. However, configuring an all-to-all configuration requires complex forwarding rule constructions, which is difficult to realize with traditional distributed routing protocols that operate on embedded systems with lower computational and memory capacities. A Software-Defined Networking (SDN) approach may facilitate implementing such network functionality.

SDN enables the use of a controller for recomputing the network state reactively upon a failure, but incurs high processing delays [6]. In [7], we provided an overview of SDN-specific related work on topology recovery mechanisms and have shown that failure recovery in OpenFlow-based SDN networks is best handled in three steps, being 1) fast failure detection through liveliness monitoring protocols, 2) failure protection through computation and configuration of backup rules prior to failure, which is the fastest recovery approach possible but may not deliver optimal network configuration, and 3) recomputation of optimal network state and new backup paths as soon as the failure detection has propagated to the network controller. Our proposal [7] showed very fast results, but assumed the configuration of backup rules to be present.

Our contributions in this paper are three-fold:

1) We derive the hard and soft constraints that should be incorporated by a resilient routing configuration.
2) We present and evaluate algorithms for computing paths that meet those constraints in circumventing link or node failures.
3) We implement and experiment with the presented algorithms in an SDN controller.

The remainder of the paper is organized as follows. In section II, we formally define the problem and give examples of what we need to compute and how traditional disjoint paths algorithms fail in doing so. Section III presents our algorithms for finding failure-disjoint paths, which we evaluate and analyze in section IV. Our prototype SDN controller implementation is presented in section V. Section VI presents related work. Finally, section VII concludes the paper. Further supportive details on the design optimization, evaluation and related work are available in our technical report [8].

II. Problem Statement

Figure 1 shows an example of a shortest path through a sample network, and a link failure between nodes C and

Algorithm 1 Per-link approach

Input: Adjacency matrix $adj = G(N, L)$
Output: Forwarding matrix fw containing primary and backup rules

1: set fw to all-to-all shortest paths matrix
2: **for** each node $n \in N$
3: **for** each outgoing link l of n
4: set $tAdj$ to shadow copy of adj
5: remove link l from $tAdj$
6: set $\{n'\}$ from N where $nextLink = l$
7: compute 1-to-$\{n'\}$ shortest paths from $tAdj$:
8: store all $nextLink$ as $fw[(curNode, l)][n']$
9: **return** fw

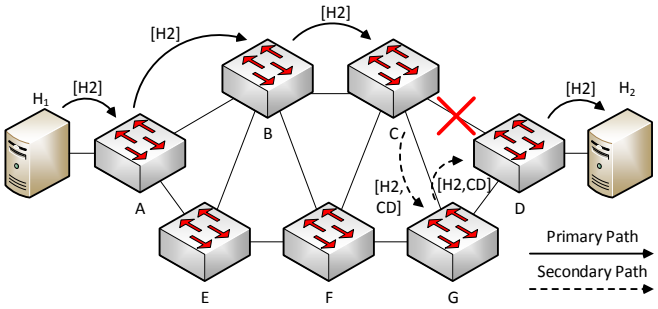


Figure 1: Failure-disjoint path and labels from H_1 to H_2

D. Although we are looking for an all-to-all solution, for illustration purposes we will use the example of traffic flowing from node H_1 to node H_2 in the network. The primary path of the traffic, which is the shortest path, breaks by the failure of link l_{CD}, an event only noticeable by node C, which is an intermediate node along the primary path. In order for the traffic to arrive at node H_2, there must be an alternative rule to revert to at node C that will ultimately route the traffic to node H_2. In essence, we are looking for an all-to-all solution in which all nodes are preconfigured with backup forwarding rules to overcome any such single link or node failure in the network. Moreover, since those rules will be computed for each possible specific single link/node failure, both the primary and backup paths will be as short as possible in length, which is a big gain over standard path disjoint protection schemes. The problem can be formally defined as follows.

Single Failure Avoidance Rule Assignment (SFARA) problem: Given a directed network G of a set N of $|N|$ nodes and a set L of $|L|$ directed links. Each link $l_{uv} \in L$ connects nodes u and v, and is characterized by a link weight ℓ_{uv} and a boolean link status s_{uv} indicating link functionality. $s_{uv} = up$ implies that link l_{uv} is functioning normally, while $s_{uv} \neq up$ implies that link l_{uv} is not functioning. Find an overall set of primary and backup forwarding rules such that any possible source node $x \in N$ can send packets to any possible destination node $y \in N$ when all links are operational ($\forall l_{uv} \in L : s_{uv} = up$), or under a single link (or node) failure ($\exists! l_{uv} \in L : s_{uv} \neq up$).

The following constraints exist for the SFARA problem:

1) The status s_{uv} of each link $l_{uv} \in L$ is only available from its adjacent nodes u and v, and may be used in the forwarding logic of nodes u and v. For example, $(s_{uv} = up)?(output(l_{uv})) : (output(l_{uw}))$ describes the forwarding logic where node u forwards packets to node v when link l_{uv} is operational, or to node w over link l_{uw} otherwise. Node u thereby relies on node w to have a suitable backup path towards the destination.
2) A set of forwarding actions can be performed on a packet at each node, including (a) dropping it, (b) rewriting, adding or removing any of its labels and (c) forwarding it to the next node by outputting it to a specific output port or link.
3) The appropriate forwarding actions for each packet are selected from a forwarding table based on properties such as: (a) the packet's incoming port, (b) (wildcard) matching on packet labels, such as its Ethernet addresses, IP addresses, TCP or UDP source and destination address, VLAN tags, MPLS labels, etc., and (c) status of the outgoing links of the router or switch.

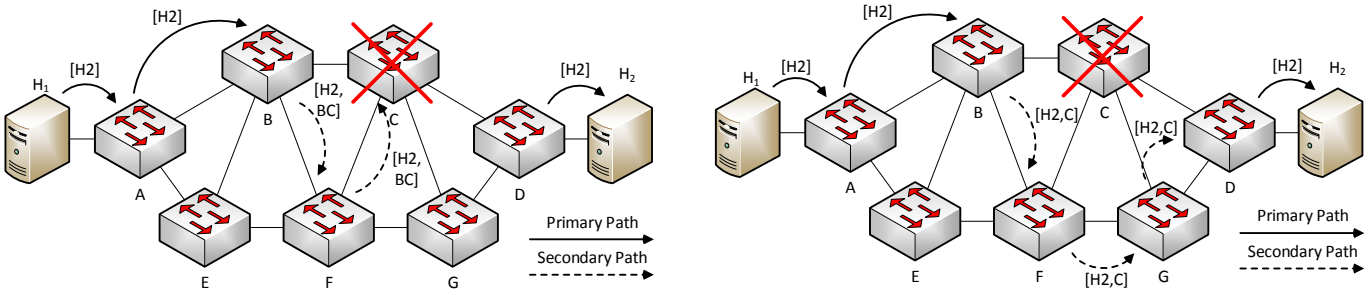
Algorithm 2 Per-node approach, changes to alg. 1 underlined

Input: Adjacency matrix $adj = G(N, L)$
Output: Forwarding matrix fw containing primary and backup rules

1: set fw to all-to-all shortest paths matrix
2: **for** each node $n \in N$
3: **for** each outgoing link l of n
4: set $tAdj$ to shadow copy of adj
5: set n^R to node opposite of link l
6: remove node n^R and adjacent links from $tAdj$
7: set $\{n'\}$ from N where $next - link = l$
8: compute 1-to-$\{n'\}$ shortest paths from $tAdj$:
9: store all $nextLink$ as $fw[(curNode, n^R)][n']$
10: **return** fw

III. Per-failure precomputation for affected shortest paths

We explain our algorithm for finding and configuring link-failure disjoint paths using labeling techniques in subsection III-A, and later modify it to node-failure disjoint paths in subsection III-B. Knowing whether a link or node failure has manifested can be difficult, since each node can only determine that an adjacent link is broken, while the failure may only be limited to the reported link or may include the adjacent node (and all of its links). A conservative approach would be to assume that all link failures imply node failures, but this leads to higher detours and possibly false negatives in determining whether there exists a detour path to the destination node. Subsection III-C thus presents our hybrid adaptation from the link- and node-failure disjoint paths where we use a labelling technique to "upgrade" a link failure to a node failure only when necessary, and adapt the forwarding strategy accordingly. Finally, section III-D discusses how we optimize routing table complexity by removing redundant rules.



(a) Node failure at node C incorrectly handled by link-disjoint backup path

(b) Proper detour around failure of node C

Figure 2: Failure disjoint paths and labels used in forwarding

A. *Link-failure disjoint paths*

Algorithm 1 presents our algorithm for computing primary and backup forwarding rules for all possible source-destination pairs given that at most one link is broken at any time. The algorithm first optimizes the length of the primary path, and then optimizes the length of the detour towards the destination node for all possible link failures.

Line 1 computes a regular all-to-all shortest paths matrix, using a shortest paths algorithm. Lines 2 and 3 iterate through all nodes' outgoing links. Since any link connects exactly two nodes this results in a combined complexity of $2|L|$, leading to an intermediate complexity determined by $|N|$ times the one-to-all shortest paths computations and $\mathcal{O}(|L|)$ for the following procedure. Line 4 creates a shadow copy of the adjacency matrix, which stores only the changes from the original. Calls to the shadow copy check for changes first and if absent return the original result from the original table. Since we remove at most one link, our shadow copy suffices to be a function call to the original table that filters out the one link before looking up the value in the matrix. Hence, creation and lookup both have a constant complexity. Line 5 removes the link under evaluation from the shadow copy, which has a complexity of $\mathcal{O}(1)$. Line 6 selects all destinations whose shortest paths go through the removed link. Sets containing the shortest path destinations denoted per link can be created within a time complexity contained by any of the suggested shortest path algorithms and hence does not add to the overall complexity of the algorithm. Selection of the sets is done in constant time. Finally, lines 7 and 8 compute and store the backup paths using a regular one-to-all shortest paths computation with a slight change to the stop-criterion. First of all, line 7 indicates that the algorithms may stop when all currently unreachable nodes $\{n'\}$ have been found again, there is no need to find the shortest paths to all nodes. Line 8 adds the found forwarding rules to the original forwarding matrix. A distinction between the original and backup shortest path forwarding rules from a node n to its destination forwarding rules is made by saving it under a label identifying the specific failure, in this case link l. As presented in figure 1, the node that initiates sending packets through backup paths should add a label identifying the failure it is detouring from. From this label nodes along the backup path derive that these packets need special treatment until they reach their destination or a shortest path that is not affected by the failure anymore. In the latter case, the label may be removed.

The overall complexity of the algorithm is mostly defined by the chosen shortest path algorithm. In general, our algorithm has a worst-case complexity of $\mathcal{O}(|N| + |L|)$ times the complexity of the implemented shortest path algorithm, since we need $|N|$ iterations to derive the all-to-all forwarding table and need to recompute broken shortest paths twice for all $|L|$ links.

B. *Node-failure disjoint paths*

In the case of a node failure, algorithm 1 may not work as the node opposite to the detected broken link is not excluded from the backup path. Figure 2a shows how the selection of a link-disjoint path may send packets right back towards the broken node. Even if node F would select its link-disjoint backup path towards node H_2, this path is not guaranteed to be loop-free from a previous backup path. As suggested in figure 2b, in the case of a node failure we need a node-disjoint backup path that eliminates the failed node instead of individual links from the backup paths. Algorithm 2 presents our solution that computes primary and backup forwarding rules for all-to-all paths given that at most one node is broken. The algorithm is almost equal to algorithm 1, except for minor changes. The biggest change is found in lines 5 and 6, where instead of the removal of link l, its opposite node n^R is removed from the shadow copy. The stored label n^R is used in forwarding. The computational complexity remains unchanged.

C. *Hybrid approach*

The biggest problem with link-failure disjoint paths is that they may have problems when the node opposite of the detected failed link is broken. The node that detects link failure cannot determine whether the link failure is a result of a single link failure or a node failure that affects all the failed node's links. In practice, link failures occur more than node-failures. We therefore prefer a link-failure disjoint path whenever possible, and a node-disjoint path otherwise.

In order to accomplish such routing, as depicted in figure 5, we let the asserting node assume a link-failure and act accordingly by adding a label denoting link failure and forwarding through the link-failure disjoint path. If any node along this

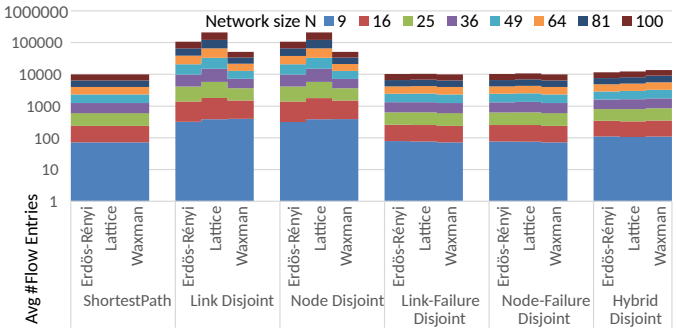


Figure 3: Average number of Flow entries in each network, categorized per network type and disjoint computation and incrementally stacked per network size

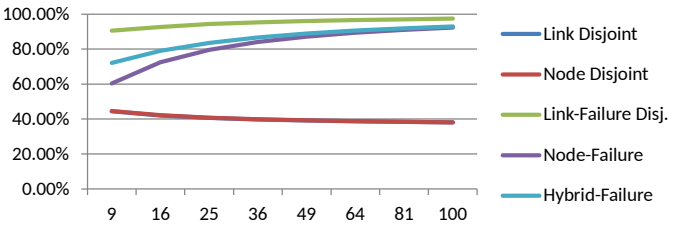


Figure 4: Ratio of forwards to Group table entries for Erdös-Rényi generated random networks of size $N = 100$ nodes

backup path has a primary forwarding rule to the failed node through another of its links, it assumes node failure based on the local link-failure detection combined with the label on the incoming packets indicating it is not the first broken link of that node. Furthermore, this knowledge is added to the attached label. When every attached label of a failed link is a concatenation of its interconnecting nodes ($\{u, v\}$), a forwarding rule wildcard match, such as $\{*, v\}$, can detect previous link failures to node v.

To compute these rules, we compute both node- and link-failure disjoint paths and place these using their unique labels in the shared forwarding matrix. Note that the initial forwarding matrix only needs to be computed once, and removal of links and nodes and their respective recomputations may occur sequentially. This procedure runs in the same worst-case time complexity as the previous two algorithms.

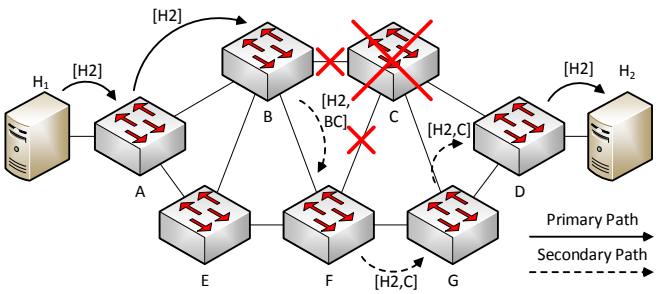


Figure 5: Link-, then node-failure disjoint approach

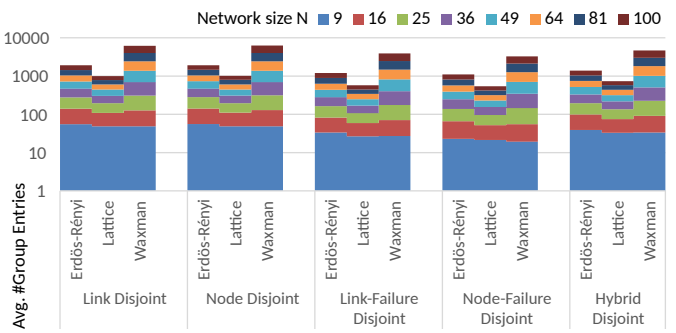


Figure 6: Average number of distinct Group Entries in each network, categorized per network type and of disjoint computation and incrementally stacked per network size

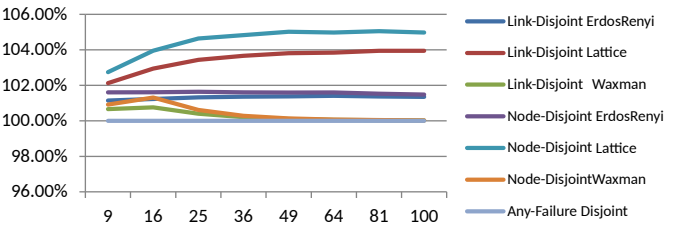


Figure 7: Increase in primary path per network size, categorized by algorithm and network type.

D. Routing Table Optimization

The procedures described in the previous subsections look for min-min link-, node- and hybrid-failure disjoint paths. However, without optimizing forwarding rule complexity, this results in a state explosion of forwarding rules.

Considering that detoured packets at a certain point follow the default shortest paths from intermediate nodes on the backup path to destination, unaffected by the found failure, a first optimization is found by removing the failure-identifying label once a suitable default shortest path is found.

We further optimize rulespace utilization by removing link-failure disjoint forwarding rules in the hybrid computation when they are equal to their respective node-failure disjoint rule.

The results of the positive effect of the above optimization are presented, for the realistic USnet topology, in [8].

IV. Evaluation

In this section, we study the performance of our algorithms through simulation in three network topologies, Erdös-Rényi random networks [9], lattice networks, and Waxman networks [10]. For our generated Erdös-Rényi random networks, we choose $\frac{2log|N|}{|N|}$ as the probability for link existence. The lattice network is useful in representing grid-based networks, which may resemble the inner core of an ultra-long-reach optical data plane system [11]. We choose a square lattice network of $i \times i$ dimension, where $i = \sqrt{|N|}$, for our generated lattice networks. The Waxman network is frequently used to model communication networks and the Internet topology [12], due to its unique property of decaying link existence over distance. In the Waxman network, nodes are uniformly positioned in the

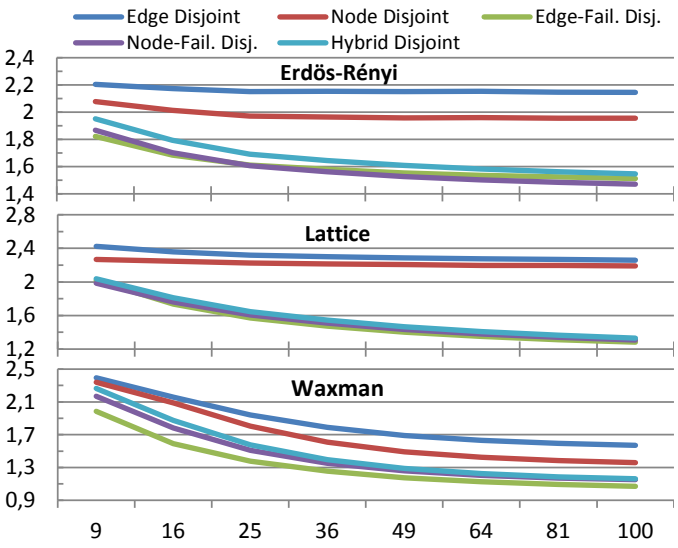


Figure 8: Average length of backup paths relative to length of their respective shortest path

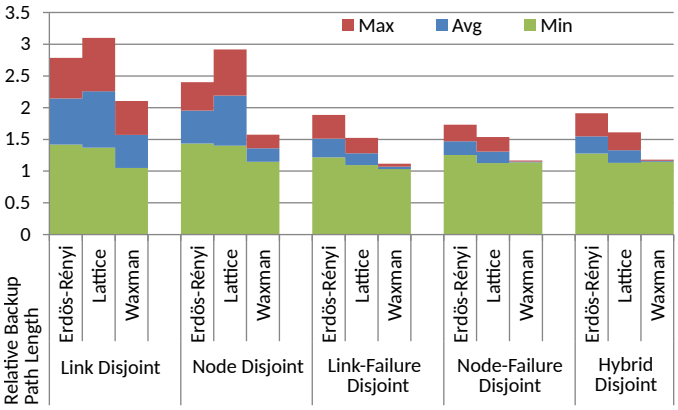


Figure 9: Comparison between minimal, average and maximal relative backup path lengths for networks of size $N = 100$

plane, and link existence is reflected by $ie^{\frac{\ell_{uv}}{j\,a}}$, where the link weight ℓ_{uv} is set to the Euclidean distance between nodes u and v, a is the maximum distance between any two nodes in the plane, and i and j can vary between 0 to 1. We set $i = 0.5$ and $j = 0.5$. We consider only two-connected generated graphs, such that the network can never be disconnected by a single node or link failure. In the Erdös-Rényi and lattice networks, each link has a random link weight between 0 and 1. No self-loops or parallel links are allowed. Simulations were conducted on an Intel(R) Core i7-3770K 3.50 GHz machine with 16GB RAM memory, and all results are averaged over a 1000 runs and grouped by the network sizes 9, 16, 25, 36, 49, 64, 81 and 100 due to the dimension of the lattice network.

We compute and compare the results of different disjoint algorithms, being our link-, node- and hybrid-failure disjoint approaches and min-sum pairs of fully link- and node-disjoint paths obtained via Bhandari's algorithm.

Figure 3 presents the average number of Flow table entries for each generated network. A regular shortest paths computation always generates exactly $|N|(|N|-1)$ Flow table entries (from each node to each other node). This number increases when more complex path computations are used. Specifically, we see a strikingly high increase in Flow table entries when fully-disjoint paths are used, which is caused by the fact that each forwarding rule has to take both source and destination into account for primary path forwarding, as well as the incoming port for crankback routing. As also shown in table I, our failure-disjoint proposal shows an increase in Flow table entries varying from 15.7% to 38% for a network size of $N = 100$ nodes, whereas for the fully-disjoint computations this is limited from no increase to 7.7%. Given that fully-disjoint paths lead to an increased table usage by a factor of 21, our method appears to be much more conservative in Flow table usage. Whereas we found that our proposal uses significantly less flow table entries, figure 4 shows up to 94% of these are forwarded to Group table entries compared to a worst case of 44% for a fully disjoint path. Although this looks like a significant increase, the absolute number of Flow entries forwarding to Group table entries remains much lower in all cases. Moreover, table I and figure 6 show that our proposal contains a significantly lower usage of distinct Group table entries in each network, which are considered scarce resources.

Besides a smaller configuration complexity, also the primary paths taken are better. While the primary path in our proposal always defaults to the shortest path, figure 7 shows that using fully-disjoint paths leads to an increase of primary path lengths of up to 5.0%, and thus incurs higher network operation costs. Although the increase of primary path length of disjoint paths in most cases grows and at a certain point seems to stabilize, with Waxman generated networks the path increase decreases over time implying that the design of the network has implications for the relative cost of robustness.

Figure 8 shows that besides a shorter primary path, our proposal on average also has significantly shorter average backup paths. In order to determine the average backup path for a node pair, we took its primary path and for each link or node on the path computed the length of the path if that specific link or node would fail and averaged accordingly. Hence, as figure 9 shows, the average backup path deviates significantly based on the link that fails. Especially the fully-disjoint paths suffer from a high deviation due to the high order of crankback routing that is involved when a link further down the primary path breaks. Figures 10 and 11 additionally show that the ratio and deviation of crankback paths is much larger for fully-disjoint paths than for our approach. Furthermore, crankback paths only exist temporarily in our proposal, since the controller reconfigures the network by applying the protection scheme to its newly established topology once it is notified of the failure, thereby removing existing crankback subpaths from the shortest paths.

The hybrid-failure disjoint path lengths are only shown for a node failure, since the path lengths for a respective link failure are equal to the results in the link-disjoint approach by design. Although the number of Flow and Group table entries, as well as the secondary path and crankback length for node failures slightly increases in the hybrid-failure approach, we claim this number is justified by the merits of shorter paths for the more often occurring link failures.

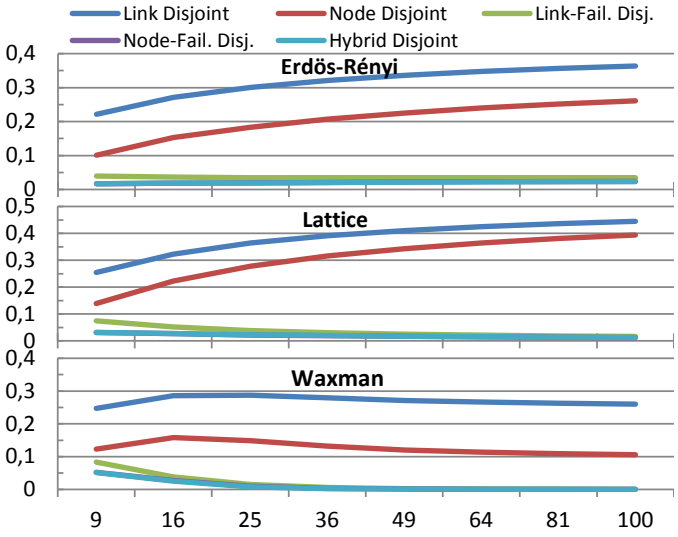


Figure 10: Average length of crankback paths relative to length of their respective shortest path

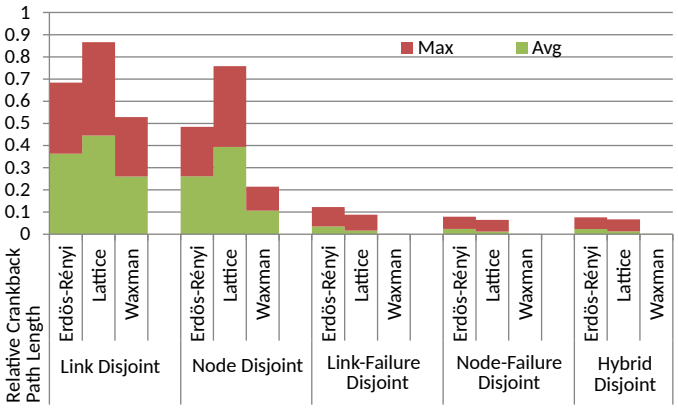


Figure 11: Comparison between minimal, average and maximal relative crankback lengths for networks of size $N = 100$

V. Software Implementation

In order to evaluate failure circumventing methods as described in the previous two sections, we have implemented an open-source prototype OpenFlow controller module that configures Software-Defined Networks with such backup rules.

We have used the Ryu controller framework [13] as basis for our implementation, which loads and executes our network application. The network topology is discovered by Ryu's built-in *switches* component, while host detection occurs by a simple MAC learning procedure in our application.

Our application is OpenFlow 1.1+ [14] compatible, since it depends on the Fast-Failover Group Tables to perform the switchover to backup paths. Tests have been performed using OpenFlow 1.3 [15] which is considered the current stable version of OpenFlow.

The application configures the network according to our protection scheme, enabling it to circumvent link or node failures independent of (slow) controller intervention. After the controller is notified of an occurred failure, it reapplies the protection scheme to the new network topology, reestablishing protection from future topology failure where possible. Reconfiguration occurs without traffic interruption using a Flow entry update strategy as explained in [16]. Our open-source OpenFlow controller is published on our GitHub webpage [17].

VI. Related Work

For an overview of disjoint paths algorithms we refer to [2] and [8], an overview of SDN-specific related work in topology recovery mechanisms is presented in [7], a survey on disaster-resiliency in SDNs is given in[18].

In terms of work related to configuration computation in Software-Defined Networks, Capone et al. [19] derive and compute an MILP formulation for preplanning recovery paths including QoS metrics. Their approach relies heavily on crankback routing, which results in long backup paths and redundant usage of links compared to our approach. Their follow-up work SPIDER [20] implements the respective failure rerouting mechanism using MPLS tags. The system relies heavily on OpenState [21] to perform customized failure detection and data plane switching, making it incompatible with existing networks and available hardware switches. Furthermore, the system does not distinguish between link and node failures as our approach does.

IBSDN [22] achieves robustness through running a centralized controller in parallel with a distributed routing protocol. Initially, all traffic is forwarded according to the controller's configuration. Switches revert to the path determined by the traditional routing protocol once a link is detected to be down. The authors omit crankback paths through crankback detection using a custom local monitoring agent. The proposed system is both elegant and simple, though does require customized hardware, since switches need to connect to a central controller, run a routing protocol, and implement a local agent to perform crankback detection. Moreover, the time it takes the routing protocol to converge to the post-failure situation may be long and cannot outpace a preconfigured backup plan.

Braun et al. [23] apply the concept of Loop-Free Alternates (LFA) from IP networks to SDNs, where nodes are preprogrammed with single-link backup rules when not creating loops. Through applying an alternative loop-detection method more backup paths are found than using traditional LFA, although full protection requires topological adaptations.

VII. Conclusion

In this paper we have derived, implemented and evaluated algorithms for computing an all-to-all network forwarding configuration capable of circumventing link and node failures. Our algorithms compute forwarding rules that include failure-disjoint backup paths offering preprogrammed protection from future topology failures. Through packet labelling we guarantee correct and loop-free detour forwarding. The labeling technique allows packets to return on primary paths unaffected by the failure and carries information used to upgrade link-failures to node-failures when applicable. Furthermore, we have implemented a proof-of-concept network controller that configures OpenFlow-based SDN switches according to this

Table I: Results for the evaluated algorithms and networks of network size $N = 100$

Network	Disjoint	Flow Entries	Distinct Groups	Primary Path Ratio	Backup Path Ratio	- Min	- Max	Crankback Ratio	- Max
Erdös-Rényi	Link	106852.298	1908.580	1.014	2.146	1.419	2.787	0.363	0.684
Erdös-Rényi	Node	106632.844	1913.405	1.015	1.956	1.434	2.402	0.261	0.484
Erdös-Rényi	Link-Failure	10160.248	1187.016	1.000	1.512	1.218	1.887	0.035	0.123
Erdös-Rényi	Node-Failure	10149.929	1096.475	1.000	1.471	1.254	1.733	0.024	0.079
Erdös-Rényi	Hybrid-Failure	11451.396	1388.225	1.000	1.547	1.277	1.914	0.023	0.076
Lattice	Link	207585.132	1002.772	1.039	2.259	1.369	3.101	0.445	0.866
Lattice	Node	207642.592	1006.752	1.050	2.190	1.403	2.919	0.394	0.758
Lattice	Link-Failure	10381.567	571.113	1.000	1.283	1.095	1.525	0.016	0.088
Lattice	Node-Failure	10663.192	542.702	1.000	1.308	1.127	1.538	0.012	0.065
Lattice	Hybrid-Failure	12320.495	735.551	1.000	1.331	1.131	1.612	0.013	0.067
Waxman	Link	50885.088	6208.709	1.000	1.570	1.049	2.105	0.260	0.528
Waxman	Node	50572.606	6247.912	1.000	1.359	1.147	1.576	0.106	0.214
Waxman	Link-Failure	9900	3874.098	1.000	1.070	1.032	1.117	0.000	0.001
Waxman	Node-Failure	9900	3268.110	1.000	1.152	1.139	1.166	0.000	0.000
Waxman	Hybrid-Failure	13671.039	4660.458	1.000	1.163	1.147	1.180	0.000	0.000

approach, showing that these types of failover techniques can be applied to production networks. Although implemented in OpenFlow, our method is applicable to all networks in which central controllers have an equally granular topology overview and the ability to match and add or rewrite protocol labels.

Compared to traditional link- or node-disjoint paths, our method shows to have significantly shorter primary and backup paths. Furthermore, we observe significantly less crankback routing when backup paths are activated in our approach. Besides shorter paths, our approach outperforms traditional disjoint path computations in terms of respectively the needed Flow and Group table configuration entries by factors up to 20 and 1.9. Our approach allows packets that encounter a broken link or node along their path, to travel the second-to-shortest path to their destination taken from the node where the link or node failure is detected. We apply Software-Defined Networking, specifically the OpenFlow protocol, to configure computer networks according to the derived protection scheme, allowing them to continue functioning without (slow) controller intervention. After the network controller is notified of the link or node failure it reconfigures the network by applying the protection scheme to its newly established topology, therewith reassuring protection from future topology failure within reasonable time. For future work, we suggest researching the protection of multi-link and -node failures as well as strictly guaranteeing QoS constraints under failure.

REFERENCES

[1] C. Doerr and F. Kuipers, "All quiet on the Internet front?" *IEEE Communications Magazine*, vol. 52, no. 10, pp. 46–51, 2014.

[2] F. A. Kuipers, "An overview of algorithms for network survivability," *ISRN Communications Networking*, 2012.

[3] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)," IETF RFC 5880 (Proposed Standard), Jun. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5880.txt

[4] "IEEE Standard for Local and Metropolitan Area Networks Virtual Bridged Local Area Networks Amendment 5: Connectivity Fault Management," *IEEE 802.1ag*, 2007.

[5] M. Shand and S. Bryant, "IP Fast Reroute Framework," IETF RFC 5714 (Informational), Jan. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5714.txt

[6] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Openflow: meeting carrier-grade recovery requirements," *Elsevier Computer Communications*, 2012.

[7] N. L. M. van Adrichem, B. J. van Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *IEEE Third European Workshop on Software Defined Networks (EWSDN'14)*, 2014.

[8] N. L. M. van Adrichem, F. Iqbal, and F. A. Kuipers, "Computing backup forwarding rules in software-defined networks," *arXiv preprint arXiv:1605.09350*, 2016.

[9] P. Erdős and A. Rényi, "On random graphs," *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290–297, 1959.

[10] B. M. Waxman, "Routing of multipoint connections," *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 9, pp. 1617–1622, 1988.

[11] A. R. Moral, P. Bonenfant, M. Krishnaswamy, and O. In, "The optical internet: architectures and protocols for the global infrastructure of tomorrow," *IEEE/ACM Transactions on Networking*, vol. 39, no. 7, pp. 152–159, 2001.

[12] M. Naldi, "Connectivity of Waxman topology models," *Elsevier Computer Communications*, vol. 29, pp. 24–31, 2005.

[13] "Ryu SDN Framework," 2015, [Accessed: 2015-12-22]. [Online]. Available: http://osrg.github.io/ryu/

[14] "Ryu SDN Framework," 2011, [Accessed: 2015-12-22]. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf

[15] "Ryu SDN Framework," 2015, [Accessed: 2015-12-22]. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf

[16] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM, 2011, p. 7.

[17] N. L. M. van Adrichem, F. Iqbal, and F. A. Kuipers, "TUDelftNAS/SDN-OpenFlowBackupRules," 2016, [Accessed: 2016-03-23]. [Online]. Available: https://github.com/TUDelftNAS/SDN-OpenFlowBackupRules

[18] C. M. Machuca, S. Secci, P. Vizarreta, F. Kuipers, A. Gouglidis, D. Hutchison, S. Jouet+, D. Pezaros+, A. Elmokashfi, P. Heegaard++ *et al.*, "Technology-related disasters: A survey towards disaster-resilient software defined networks," pp. 181–185, 2015.

[19] A. Capone, C. Cascone, A. Q. Nguyen, and B. Sanso, "Detour planning for fast and reliable failure recovery in sdn with openstate," in *Design of Reliable Communication Networks (DRCN), 2015 11th International Conference on the*. IEEE, 2015, pp. 25–32.

[20] C. Cascone, L. Pollini, D. Sanvito, A. Capone, and B. Sansò, "Spider: Fault resilient sdn pipeline with recovery delay guarantees," *arXiv preprint arXiv:1511.05490*, 2015.

[21] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 44–51, 2014.

[22] O. Tilmans and S. Vissicchio, "Igp-as-a-backup for robust sdn networks," in *Network and Service Management (CNSM), 2014 10th International Conference on*. IEEE, 2014, pp. 127–135.

[23] W. Braun and M. Menth, "Loop-free alternates with loop detection for fast reroute in software-defined carrier and data center networks," *Journal of Network and Systems Management*, pp. 1–21, 2016.