Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

SDN RESILIENCY TO CONTROLLER FAILURE IN MOBILE CONTEXTS

David M. Nicol
Rakesh Kumar

Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, IL USA 61801

ABSTRACT

Software defined networking (SDN) is a technology for management of computer networks. We are interested in SDNs where the switches are mounted on vehicles, the controller is airborne, and connectivity changes dynamically as a result of vehicular movement. The challenge of interest is that when a controller fails the switches are left with whatever configuration they had at the point of failure. Without a controller, as the switches move and links become unuseable, flows configured to use those links are broken unless the configuration has anticipated such events and provided alternative routes. We describe a high fidelity SDN network simulator which uses snapshots of actual switch configurations and analyzes their resilience to impacts of controller failure. We propose an algorithm for backup paths which is guaranteed to not cause loops, and conduct a simulation study to understand the types of domains where have the greatest impact on network performance.

1 INTRODUCTION

Software Defined Networking (SDN) increases flexibility and control over establishment and maintenance of connectivity. Under SDN the logically centralized *control-plane* implemented in one or more controllers decides which packet forwarding and header transformations occur at switches, while the *data-plane* (i.e. individual switches) performs these actions on the packets. The control-plane dynamically sets up and tears down particular routes in response to connectivity needs, and network conditions. SDNs appear most frequently in data centers, where if a switch is programmed to use a particular egress link for forwarding some traffic and it discovers that that link has failed, it reports the failure to the SDN controller. In turn the controller computes alternative routes for traffic which use that link, and issues modified rules to switches to implement the new path.

In the operational context studied here is of military interest, and was suggested to us by our sponsor, Boeing. Switches are mounted on vehicles, and use directed high powered wireless communication between themselves, and with one or more airborne controllers. Mobile hosts communicate with a switch through low power wireless means, and the switches provide longer-range host-to-host connectivity. Hardware and protocols independent of a switch keep it apprised of its connectivity with other switches; from the SDN protocol point of view, the fact the source of the link state is immaterial. The topology and connectivity needs are dynamic, but nevertheless may change slowly. Vehicles may move, and links may become disconnected because of distance, and/or because a vehicle loses line-of-sight with another. Changes which disconnect some links can enable the connection of others. Therefore if the changes do not happen too rapidly the airborne controller can respond with new routes when its switches report link failure. Links that become disconnected can become connected again when the causes of the disconnect subside.

With OpenFlow protocols since OpenFlow 1.3 it has been possible to place logic in switches for dealing with link failure. This is accomplished using a so-called "fast failover" table. After a packet has advanced to the egress step in its pipeline, it looks up an ordered list of links to be used when directing the packet towards the destination declared elsewhere in the packet. The packet is pushed through the first link in the table which is live. The important point is that the controller can perform analysis which creates the table, and introduces backup paths to respond to failed links. It is important though to understand the properties of the backup paths so created, e.g., that they do not create loops.

The challenge posed to us by Boeing was to develop methodologies and tools capable of evaluating the compliance of the configurations their controller creates with user defined expectations (a.k.a. "policy".) This paper describes the initial result of our response to that challenge.

This paper describes three contributions. First, we created an SDN network simulator specifically for this task, the *High Fidelity Flow Analyzer*, or HF^2A. This tool accepts descriptions of experiments from the user, and executes those experiments using real SDN rules read up from real SDN switches. The user's experimental description includes a description of the physical connectivity of the SDN, specifies a set F of host-to-host flows to be carried by the network, enumerates flow policies and describes how link failures and link recoveries are to be simulated. The HF^2A loads the rules into switch simulators which faithfully emulate all the actions that an OpenFlow 1.3 switch can take. The HF^2A exercises the network model, following the user directives for the experiments. The kinds of questions the experiments may be designed to answer include

- If 5% of the links fail, on average what fraction of flows in F are routed to their destination without interruption?
- If 5% of the links fail, on average what fraction of the flows in F meet all policy objectives?
- Given any 3 link failures, is every flow in F delivered to its destination?
- Given (random) link failure, what is the probability that the performance of a given flow in F may be impacted either by an increase in the number of switches traversed exceeding its policy, or by overloading the bandwidth capacity of a link?
- If a controller fails at time s with the switches in a given configuration, what is the probability that by time $t > s$ link failures will cause some flow f to fail to be delivered?

Clearly at the heart of this problem is an ability to assess what impact link failures have on flows. With the high fidelity emulation of traffic passing through switches we are able to determine precisely what will happen after starting with a snapshot of rules, we cause (simulated) links to fail and assess the impact on the set F of user flows.

Our second contribution is an algorithm for creating backup paths which provably contain no loopbacks, regardless of the set of links which fail. This algorithm is built on top of the results of shortest path computations and yields routes that are loop-free and require no additional state information in packet headers, in the presence of arbitrary link failures. The technique is simple, and can in principle be easily added to any rule synthesis algorithm not in conflict its shortest path assumptions.

Our final contribution is a empirical study where we consider the extent to which the back-up paths created by the proposed techniques improve the connectivity of the network over configurations which have no backup paths. For this study we consider models of switch mobility and assumptions under which links fail as a result of that mobility.

2 RELATED WORK

Prior efforts to model a software-defined network for the policy validation purpose use a model for the entire network and a model for representing all possible flows in the network. Kang et. al (Kang et al. 2013) modeled the entire network as a single switch and provided a way to specify flows of interest. Kazemian et. al. (Kazemian et al. 2013) modeled the network as a graph of flow rules and specification of policy using a specialized language called $flowexp$. Khurshid et. al. (Khurshid et al. 2012) proposed a network model that represents the network using a specialized trie. However, none of these models allow specification of behavior of the network when the topology of the network changes. These approaches also did not model switches that are capable of the fast failover mechanism. This mechanism was added to the switch specification in the OpenFlow version 1.3 (Open Networking Foundation ONF) and allows choosing alternative ports for traffic without the intervention of the controller. To these ends, we proposed the port graph model that allows modeling of the network as a graph of ports and a policy specification that specifies changes in the network topology (Kumar and Nicol 2016).

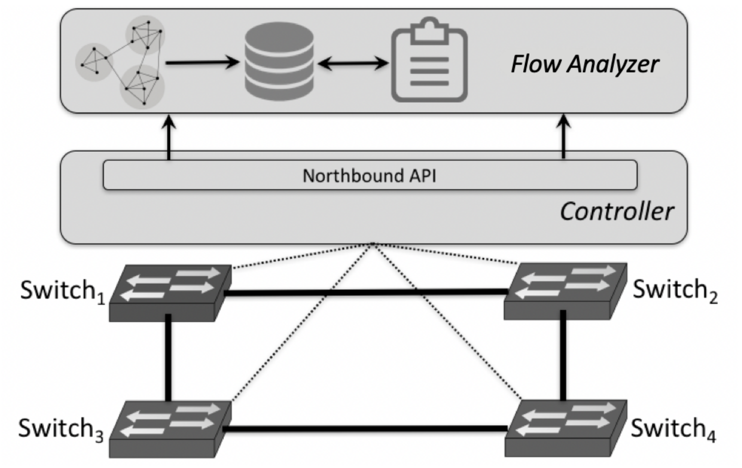
Furthermore, instead of validating the control-plane state of an SDN against a stated policy, there is prior work that synthesizes the data-plane state so that it provides certain resiliency guarantees network by using the fast-failover mechanism. Elhourani et. al. (Elhourani et al. 2014) proposed an approach that assumes k-connected topological graphs and uses arborescence to minimize the length of the failover path in case of link failures. However, Hannon et. al. (Hannon et al. 2017) proposed an approach that can be used to synthesized resilient paths on topologies that are not k-connected at the cost of longer path lengths.

Finally, there is prior work that models the traditional distributed control-plane networks for policy validation. The errant behavior of a network device in such networks is typically rooted in user mis-configuration. `Fireman` (Yuan et al. 2006) is a firewall-specific static analysis tool that allows checking for inconsistencies in configuration of the access control. `Anteater` (Mai 2011) and `Libra` (Zeng et al. 2014) illustrate policy validation by using only the data-plane state from campus and data-center networks respectively. These tools parse the individual device configuration and the data-plane state of heterogeneous devices; thus individual device models become bottlenecks to predicting overall network behavior.

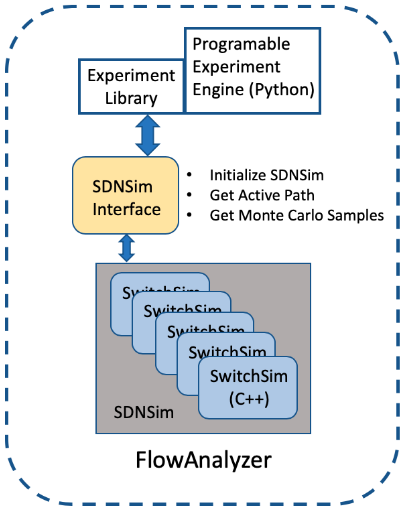
3 THE HIGH FIDELITY FLOW ANALYZER

3.1 The HF^2A Architecture

In order to acquire rules that have been issued by a controller, our approach requires access to the logically centralized *state* of the SDN: the network topology and the forwarding rules installed on switches. In the SDN architecture, the controller is the purveyor of this state and makes it available via a *northbound* API to be used by the applications. As illustrated in Figure 1a, our tool uses the northbound API to access a current snapshot of the SDN state. It should be noted that this requirement is for rules acquisition only. HF^2A can be run in isolation from the real SDN network, provided that it is given access to rules that **were** extracted from an actual network.



(a) An SDN containing four switches connected in a ring.



(b) Architecture of the SDN Simulator and Interface.

Figure 1: Architecture of the High Fidelity Flow Analyzer.

As illustrated in Figure 1b, the HF^2A is further compartmentalized. For the sake of computational efficiency, it has a component written in C++ called SDNSim. It contains high fidelity SDN Switch Simulators (SwitchSim) modules. These modules closely mimic the forwarding actions and transformations made on a packet passing through it. To understand the importance of this element of the design it is important to know that the OpenFlow specification supports more than just simple routing. The switch might block a packet entirely. It might change IP and port numbers in a packet's header to implement Network Address Translation. It might alter other bits in the packet header. The point is that HF^2A is designed to be used in an operational environment with potentially complex switch configurations, and so switches in SDNSim are executable implementations of the OpenFlow specification. An SDN emulator such as mininet provides the same functionality, but in our experience does not scale with a large number of switches.

The SDNSim exposes an API (SDNSim Interface) which can be used to perform its initialization with a snapshot of SDN State, to query whether an active flow path between two nodes would exist when a set of links in the topology have failed, and to perform Monte Carlo experiments in parallel. These actions are all performed under the direction of Python scripts resident in the Programmable Experiment Engine. Indeed all of the experiments described earlier would be developed and executed as Python scripts. As we develop classes of experiments we save them in an Experiment Library, and include these as modules in the Programmable Experiment Engine's scripts.

HF^2A brings together the fidelity of a compiled interpreter of OpenFlow rules with the flexibility of a scripting language to set up experiments, change conditions, and make calls upon the SDNSim to compute how the network will respond to those changes. This is a powerful mix. It allows us to design and execute experiments that

- introduce mobility to the SDN switches and assess the impact that mobility may have on network connectivity,
- use Monte Carlo estimation of network metrics, potentially including advanced variance reduction techniques,
- introduce models of external forces that may alter connectivity, e.g., a jamming attack.

3.2 The Switch

An SDN switch contains a table processing pipeline and multiple physical ports. Packets arrive at one of the ports, and are processed by the pipeline comprised of one or more flow tables. Each flow table contains rules ordered by their priority. Each flow rule defines a matching criteria, and an *action* taken if the packet header satisfies the rule criteria. During the processing of a single packet, these actions can modify the packet, forward it out of the switch, or drop it.

Rules are tested in decreasing order of priority until a matching rule is found. A packet arriving at a switch is first matched against the rules in the first table and assigned a set of actions to be applied at the end of table processing pipeline. The instructions in a matching rule in any table can choose to manipulate this set and can also apply some actions to the packet before it is processed by the next table in the pipeline. For our purposes the important point is that SDNSim implements OpenFlow 1.3 in the switches, and the rules created for the switches are our starting point, our ground truth. We are primarily concerned with the fast failover tables, but it is certainly possible for a switch to modify the header of a passing packet so that the destination changes and so potentially also the fast failover table used at egress. This means that for fidelity we have to be able to interpret SDN configurations, not just analyze abstractions of the switches. Unlike other analysis which compute all possible actions for all possible flows, in HF^2A we are focused on specifically identified flows, and ask questions about how the network handles these flows in the presence of link failures.

4 FAILOVER PATHS

OpenFlow's mechanism for supporting fast failover is to include so called "fast failover" tables. Inside of a switch packets pass through a pipeline of stages, and selection of an egress port is the last step. The switch can be programmed at this stage to choose a prioritized table of potential egress ports, chosen as a function of all the bits in the packet's header, in particular (but not necessarily exclusively) bits describing the packet's destination, say, d. The first "live" port appearing in the table is selected for egress. If there is no such port the packet is dropped, and the controller is notified of the particulars of the switch and packet.

Our technique for creating loop-free fast failover tables is built around Dijkstra's algorithm (Cormen, Leiserson, Rivest, and Stein 2009) for unidirected weighted graphs. This algorithm takes a particular graph node d and finds the shortest path from every other node to d. The algorithm uses a min-priority heap. An element (c,n) on the heap represents a graph node n, and the cost c of a minimum path from n to d. For the purposes of stability, given elements (c,n_1) and (c,n_2) with the same cost but different nodes, we will break the tie in favor of the element whose node has smaller identity index, i.e., take $(c,n_1) < (c,n_2)$.

While Dijkstra's algorithm is well-known, it does not typically save certain intermediate values that our algorithm uses and which are needed in a proof of loopback freedom. At the risk of seeming pedantic, we write out the algorithm in Figure 2. This is Python (2.7) code. It presumes a node class object, with the graph nodes stored in an array *Nodes*. The node object has Boolean attribute visited, an integer id, and a variable sized array of edges. Each edge is described as a tuple (c,m) where c is the cost of the edge, and m is the integer identity of the connected neighbor, switch s_m. The algorithm is focused around a min priority heap (here we use a standard Python module heapq). The key idea is that the priority heap contain descriptions of an as-yet-unvisited graph node, and the minimum cost of traversing the graph from that node to the named destination, d. Pulling the node(c,m) from the heap we mark s_m as visited and examine all of its unvisited neighbors (if any). If there is an unvisited neighbor s_k, then the least cost path from s_k to s_d which passes first through s_m follows the least cost path from s_m to s_d. That is recorded, and an entry is pushed onto the heap which identifies k and the least cost path from s_k to s_m to d. Once all the nodes have been visited the min priority heap will be empty and the algorithm terminates.

The return from a call *shortestPath(d)* is a Python dictionary. The presence of element sp[i][j] implies that switches s_i and s_j share an edge, and that the least cost path from s_i to s_d which first traverses that edge has cost sp[i][j].

We can use the results of a procedure like *shortestPath* to create fast failover tables. Imagine a topology graph whose nodes are hosts and switches; an edge may exist between a pair of switches or a host and a switch, but not between a pair of hosts. If a host connects to only one switch we can simplify the model and think of switches as destinations rather than hosts, assuming that the underlying addressing scheme and packet header allows a switch to compute a target switch identity. Nominally the cost of an edge could be 1, so the min-cost is the minimum number of hops. Alternatively one could define some other cost function that makes sense in the domain, so long as it is strictly positive.

Whatever the method and however the costs are defined, we can use the results of calling *shortestPath* with target d as an argument to create fast failover tables supporting d at all switches. The first link in each such table is the primary link created during rule synthesis. We engage fast failover only when that link fails. Now imagine switch s_i and suppose the primary link used to route to d is p_d. Every neighbor node of s_i with an entry in sp[i] (excluding the one led to by following p_d) is a potential next hop in a fast failover route to d. Observe that it is not necessary that every neighbor s_k have an entry in sp[i]. By the time node s_k and a cost came off the priority heap, switch s_i may have already been visited, and so be skipped (see lines 20 and 21). In this case s_i is a way to d from s_k, but not vice-versa. Note also it might be that sp[i] has no members other than p_n, which implies an inability to use this method to create a fast failover table at i. However, assuming there are neighbors through which s_i can reach d, the links to those neighbors are candidates for entries in a fast failover table.

There is a trade-off to be managed. Fast failover tables take up memory in a switch. One option is to include just a single fast failover option. It is intuitive to choose the neighbor s_j for which sp[i][j] is

```
import heapq
def shortestPath(d):

    sp = {}
    # sp[i][j] = least cost of path i->n through j
    for i in xrange(0,len(Nodes)):
        Nodes[i].visited = False  # list of graph nodes
        sp[i] = {}

    heapq.heappush(q,(0,d))  # note tuple comparison.
    while q:
        # next shortest path from n
        (c,m) = heapq.heappop(q)

        Nodes[m].visited = True  # don't come around here no more
        m_node = Nodes[m]        # get to data structure

        for nbrIdx in xrange(0, len(m_node.edges)):
            (nc,nbr) = m_node.edges[nbrIdx]
            if Nodes[ nbr ].visited:
                continue

            sp[nbr][m] = nc+c    # least cost of nbr to m to d

            # expand from also
            heapq.heappush(q,(nc+c,nbr))
    return sp
```

Figure 2: Python routine computing shortest paths from d to every other node.

minimized among all s_j not reached through the primary link. In the experimental section we'll call this the sp-1 option. An option at the other extreme is to make the fast failover table as large as possible, so the table includes all links to neighbors s_j appearing in sp[i] which are not the primary link. We might as well order them by increasing cost, although for the no-loop properties we establish this does not matter. In the experimental section we'll call this the sp-* option. Obviously sp-* offers a longer network life, but at the cost of more storage.

The current literature on back-up routing focuses on properties such as guaranteed tolerance up to k arbitrary link failures. However, the larger the value of k, the stronger the assumptions have to be made about connectivity of the network. The operational assumptions of the mobile switches scenario preclude our ability to assert strong connectivity properties. Given that, it seems a desirable property to retain is that in the presence of an arbitrary set of link failures, the fast failover routes used have no loops. Happily, we can prove this property for fast failover links defined as we have above. With the previously stated understanding that a source or destination may be either switch or host and that all intermediary steps in a path be through switches, we avoid the cumbersomeness of maintaining the distinction and simply talk about paths through "objects".

Theorem 1 Let s_0 and d be source and destination objects, and imagine that some number of edges has failed. For $i = 1, ...i$ let $s_1, s_2, \ldots, s_k$ be objects such that s_i is a object reached by following the highest priority live link of the fast failover table of s_{i-1} for destination d, and either $s_1 = d$, or $s_i \neq d$ for $i = 1, \ldots, s_{k-1}$. Then the elements of this sequence are unique, i.e., there are no cycles.

Proof. Suppose not. Then there exists s_j and $i < j$ such that $s_{j+1} = s_i$. Suppose that j is the largest index with this property. We will say that a link from object a to b is *primal* if among all neighbors of a, Algorithm 2 reached a first (at line 23) through object b (at line 13). Observe that the link from s_j to s_i cannot be primal. According to Algorithm 2, the edge between s_i and s_j would have been established when s_i was pulled off the priority heap and extensions of the shortest path from s_i to d to its neighbors were considered. However, since the link from s_j to s_i is not primal, there exists an object b_{j+1} such that the link from s_j to b_{j+1} *is* primal. s_i is a different neighbor of s_j, and the fact that the shortest path from s_j to d is through $b_{j+1} \neq s_i$ means that b_{j+1} is pulled off the priority queue *before* object s_i is pulled off. This means that s_j is marked as visited before s_i is taken off the priority queue, which means that when s_i *is* taken off the queue, it will not touch already-visited neighbor s_j, and thus will not establish a link from s_j to s_i. This provides the contradiction.

It is clear from this argument that any path which follows only primal links and links built into fast failover tables can have no cycles, so it follows that cycle-freedom holds under arbitrary link failures. It is also clear that the packets need not carry any additional information in their headers to take advantage of this capability. $\square$

5 EVALUATION

5.1 Simulation Model

We describe now the simulation model which resides in the Programmable Experiment Engine in the Flow Analyzer of Figure 1.

We expect that network behaviour depends considerably on context, and we have the question then of what constitutes a reasonable context? Modeling SDN use in a data center we would describe a link failure with some stochastic hazard rate function, and randomly sample links and times of failure. But the mobile SDN scenario is different. Our intuition is that the link failures are more likely to be caused by external factors that interfere with communication, possibly only temporarily. Two things come to mind as potential contributors to such failures, increased distance, and/or obstacles obscuring line-of-sight between the mobile switches. Another difference is that the causes of the link failures may disappear, making it possible to restore a failed link and use it again. Just as we assume that the radio supporting the switch can detect loss of connection and so cause a link to be marked as failed, we assume that that same radio can detect when new edges are possible, in particular when a lost link known to the configuration reappears.

We correspondingly build a model with the following elements.

- Random placement of n switches within the unit square $[0,1] \times [0,1]$.
- Uniformly random selection of vehicle bearing, sampled over $[0, 2\pi]$.
- Random speed (governed by user input).
- Maximum distance threshold for communication (governed by user input).
- Uniformly random placement of circular obstacles in $[0,1] \times [0,1]$, with number and radius of obstacles governed by user input.
- Random selection of flow endpoints (the number of flows is specified by the user.)

Following the initial placement, an edge is considered to exist between two switches if they are "close enough" (according to a user defined threshold,) and a straight line between them does not intersect any obstacle. Figure 3 illustrates a sample initial starting point for the same set of switches, in one case when there are no obstacles and in the other case when there are. Switches have little directional arrows indicating

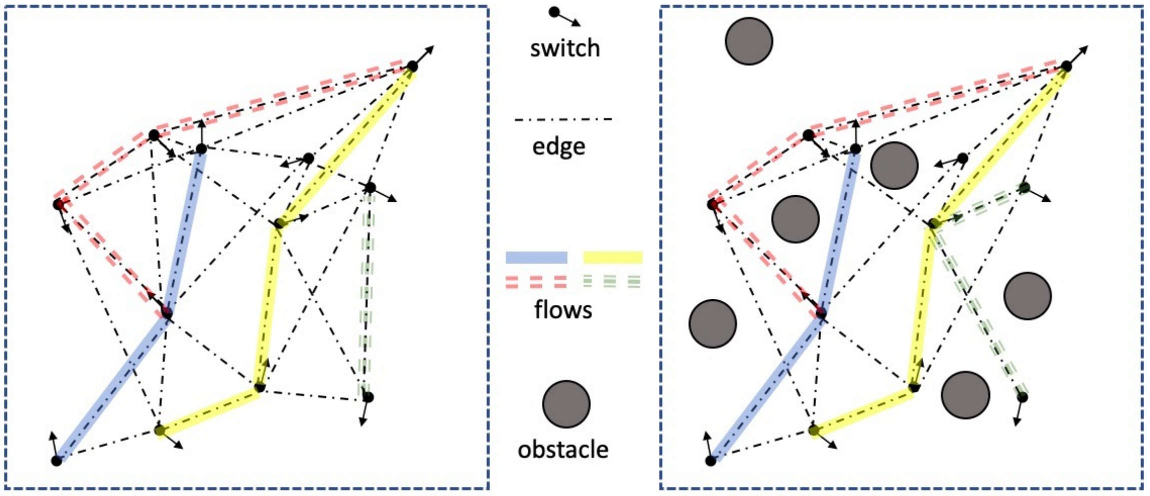


Figure 3: Moving Switch Model for SDN Resiliency Evaluation.

their bearing. In these examples, remembering that the spatial coordinates are the unit square, switches must be within distance 0.3 to be connected, and in the presence of obstacles, have a clear line of sight. Sample flows are overlaid on the primal links that carry them.

5.2 Routing

Switch rules are synthesized to route flows along the shortest number of hops, and to provide fast failover in one of four *modes*. The base mode provides no failover. Recalling our earlier definition of a link on the fail-free route as being primal, the failure of any primal link will disconnect any flow configured to use that link. A second option, we call the sp-0 mode, provides one failover alternative for primal links (if possible from the shortest path algorithm of §4,) but only for primal links. It is a way of providing some protection, but keeps the memory cost of that protection low, e.g., not having a failover link for destination d on any switch which does not carry flows to d. It is possible for a flow to survive the failure of multiple links, but only if the failover paths bring the route back to the primal path, and the links which fail along its path are all primal. A third option we called the sp-1 mode in §4, provides up to one failover alternative to *every* link, primal or not. The memory requirements for a switch's fast failover tables is bounded by two times the number of flow destinations. The fourth mode, called sp-* in §4, may provide multiple failover links for every link as a result of running the shortest-path algorithm for every flow's destination. Under sp-* the memory requirements for a switch's fast failover tables is bounded by the number of flow destinations times the number of ports in the switch.

5.3 Executing an Experiment

Prior to any runs we randomly sample the initial position of switches, and the position of any obstacles. While random, the construction is done in a way which ensures that the initial network is connected. The links are discovered, and the flow endpoints are randomly sampled. At the beginning of every run the switches are returned to their initial positions, and the links are returned to their initial state. At the beginning of each run the bearing and speed of the switches are randomly sampled; these per-run samples are what makes the network behavior in one run different from that of another run.

With every time-step (of magnitude given by the user) the position of each switch is updated in accordance with its speed and bearing. If a prospective update would take the switch out of the unit square, we randomly re-sample the bearing until we find one that takes the switch back into the domain. As a result of a position change an edge may disappear, either because the switches become too far apart, or because their movement brings an obstacle between them. By the same token an edge may appear, in particular, an edge that once carried a flow but was lost may reappear. We cause that link to be recognized within the switch as being live. After having identified which links change state following an update, the SDNSim is notified of those changes. The SDNSim is then queried as to the state (and path) of every flow. Within SDNSim if a link a flow previously followed from a switch has failed, the fast failover mechanism may reroute the flow. The particulars of the flow—whether it reaches its destination, the path the flow follows—are returned for analysis against policy requirements. For the simple case under discussion the only datum of interest is whether the flow succeeded in reaching the destination.

5.4 Metrics

With every time-step we can count the number of user flows which are supported as being active, as a function of the failover mode. We illustrate a sample behavior in Figure 4a, plotting the fraction of live flows (out of 100 initial flows), as the simulation goes through 1000 time-steps. The immediate drop in flow connectivity happens because the network starts in a fully connected state, supporting all flows, but then movement causes link state to change. We can clearly see the impact of being able to re-incorporate links that had failed but became operational again. This graph also plots any topological connectivity, i.e., assumes that if there is a topological path between endpoints then a flow between those endpoints can be routed. Thus the 'topo' curve illustrates the best that is possible at any given point, given the state of the topology.

We can summarize the aggregate connectivity of a run like this by the area under a life-time curve, also known as the space-time product. This is easily computed by accumulating over all time-steps the product of the time-step size times the fraction of flows that can be routed at that time. For the example in Figure 4a the space-time for each mode is given in within the figure as spt(base), and so on. Note that these metrics have been normalized by dividing the computed space-time product by the length of the simulated interval.

If we were to use SDNSim exclusively for these experiments we would run the simulation over the differing configurations, but would do so having synchronized the random number streams so that switches start in exactly the same place and move in exactly the same way for every set of synchronized runs. While we have used our link failure models to drive SDNSIM evaluation of network behavior using fast failover links, the data we later present is taken from a simpler, faster network simulator which is focused entirely on routing, and can within a single run assess which flows would fail and when as a function of failover mode. As all of the stochastic behavior is external to SDNSim and as we have validated SDNSim, the data we would obtain using SDNSim would be exactly the same as that presented here.

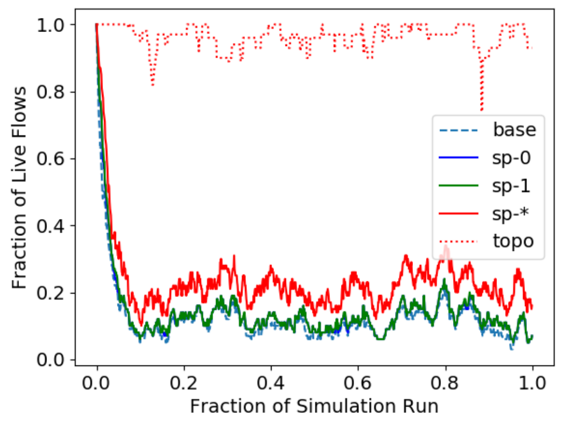
Thus every Monte Carlo repetition produces random space-time metrics spt_b, spt_0, spt_1, sp_* and spt_t (with the subscripts coding the now repetitious failover modes. These values may vary significantly from context to context. With the intention of better understanding when using failover links improves aggregate network connectivity we compute relative gain measures

$$g_0 = \frac{spt_0 - spt_b}{spt_b}, \quad g_1 = \frac{spt_1 - spt_b}{spt_b}, \quad g_* = \frac{spt_* - spt_b}{spt_b}, \quad g_t = \frac{spt_t - spt_b}{spt_b}.$$

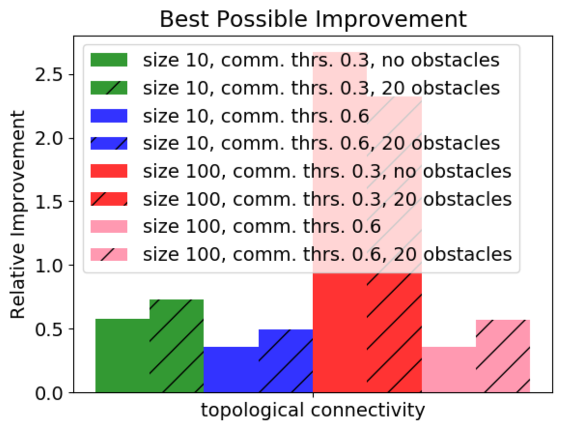
We then use standard output analysis techniques of Monte Carlo simulation to estimate their means, and construct confidence intervals around those means.

5.5 Experiments

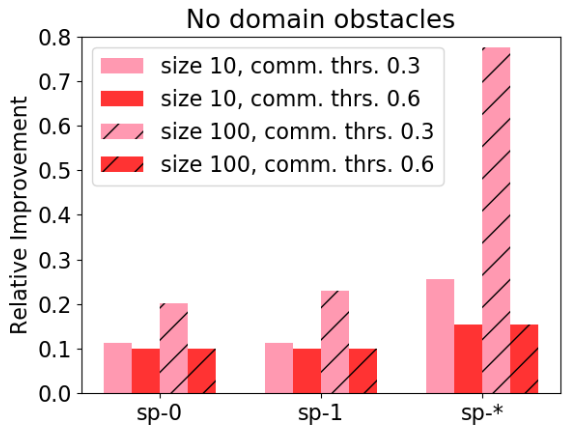
The network model we describe and study here is of the same genus as the physicist's spherical chicken, useful for understanding principles, but not particularly reflective of reality. With this understanding what we can hope to observe by simulation is how factors we believe might impact behavior, do. The factors we explore in our experiments are these:



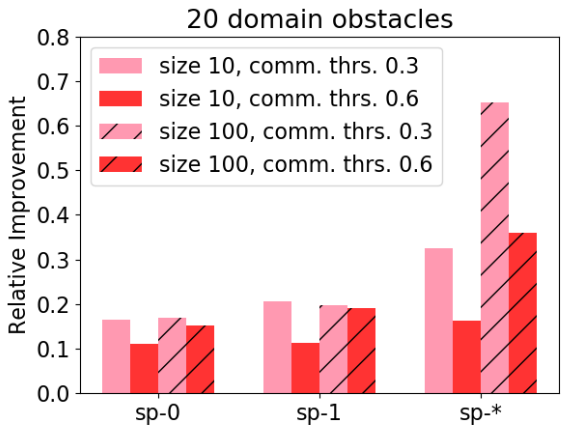
(a) Sample Trajectory of Live Flows.



(b) Maximum possible improvement.



(c) Improvement using failover links, no obstacles.



(d) Maximum possible improvement, obstacles present

Figure 4: Relative Improvement of Space-Time Connectivity Due To Failover Modes.

Network Density. Given the fixed size of the domain, we increase network density by increasing the number of switches. Our experiments use either 10 switches with 10 uniformly sampled flows (meaning that their endpoints are uniformly sampled, and no flow's source and destination switches are the same,) or 100 switches with 100 uniformly sampled flows.

Presence of Obstacles. We introduced the possibility of obstacles interfering with communication as a potential explanation of how and when an edge might fail. The question is whether this model feature appears to impact the relative performance gain of failover links.

Link Connection Distance. We vary the maximum distance between switches which will support a link, between 0.3 and 0.6. We found that with the smaller network sizes any threshold smaller than 0.3 fragments the network so greatly that hardly any flows at all are supported.

Figures 4b, 4c, and 4d illustrate the means of the relative gain metrics. With 25 replications for each configuration the 95% confidence intervals were within 5% of the mean and so are omitted here. For reference we plot in Figure 4b the best possible relative improvement, under the assumption that if a topological path between a flow's endpoints exists then the flow is routed. With one notable exception, in the experiments we ran the best we can hope to do using failover routing is about 50% better than if we didn't try at all.

5.6 Implications

What did we learn from these experiments? Failover edges never hurt, but whether they add significant connectivity relative over using no failover edges depends on the topological context. In these experiments there are three ways to increase topological connectivity: by increasing network density, by increasing the communication radius, and by removing obstacles from the domain. Here are broad-brush characterizations of this data, obtained by comparing the relative performance of parameters that differ in only one attribute.

- Increasing the communication radius decreases relative performance.
- Removing obstacles tends to decrease relative performance (with notable exceptions).
- Increasing density tends to improve relative performance when there are no obstacles, but has impact when there are obstacles only for the sp-* mode.

By increasing the communication radius or by removing obstacles, for a fixed set of switches we induce more edges. This certainly gives more opportunity to find and exploit failover paths, so a *decrease* in relative performance is at first blush counter-intuitive. On reflection though we realize this is explained by the change inducing a larger benefit to the *base* case. Flows that were not connected owing to failed links are revitalized by the presence of links that can now carry them. Now the prominent exception to this rule is the sp-* mode, whose relative gain improves by removing obstacles. We understand this as a result of sp-* being the most comprehensive of the failover modes for building failover paths. For this mode the increased number of edges gives it an increased number of failover paths and makes it more resilient to failures. sp-0 and sp-1 do not significantly increase the number of failover paths as more edges are available. This same observation explains why the case of sp-* with highest density, no obstacles and low communication radius stands out comparatively. The depth of failover penetration it provides matters most when the communication radius is low. That advantage is clearly lost though with higher communication radius. Increasing the radius restores flows for the base case at a greater rate than the added edges provide better failover backup paths.

So while the specific context is not especially realistic, we see that reasoning about the root causes of the behavior we see gives us better understanding into what domain factors impact the relative utility of providing failover paths.

6 CONCLUSION

The use of software defining networking in mobile contexts is novel, and brings with it new problems. Mobile SDN networks are ideally managed by a controller, but controllers may fail, leaving us with the question of how well and how long the network can continue to operate using the configurations frozen into the switches at the time of the failure. The problem as posed to us was to devise means of assessing how well configurations we did not ourselves synthesize would support connectivity after a controller failure. To address this challenge we constructed a high fidelity SDN network simulator whose core engine interprets SDN configurations generated by real SDN controllers. We couple this with an external environment simulator that allows us to model mobility and the causes of link failure, then observe the impact on the real SDN network of the simulated link failures.

This paper (a) describes the tool, (b) proves that an intuitive means of providing so-called failover paths in anticipation of failed links needs no modification to packet headers and does not create loops, and (c) performs a simulation study to help understand the characteristics of a domain in which provisioning of SDN backup links has the greatest positive benefit to connectivity.

ACKNOWLEDGEMENTS

This research was supported in part by a grant from the Boeing Corporation.

REFERENCES

Cormen, T., C. Leiserson, R. Rivest, and C. Stein. 2009. *Introduction to Algorithms (3rd Edition)*. MIT Press.

Elhourani, T., A. Gopalan, and S. Ramasubramanian. 2014. "IP Fast Rerouting for Multi-link Failures". In *INFOCOM, 2014 Proceedings IEEE*, 2148–2156. IEEE.

Hannon, C., D. Jin, C. Chen, and J. Wang. 2017. "Ultimate Forwarding Resilience in OpenFlow Networks". In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, 59–64. ACM.

Kang, N., Z. Liu, J. Rexford, and D. Walker. 2013. "Optimizing the One Big Switch Abstraction in Software-Defined Networks". In *Proceedings of the Ninth ACM conference on Emerging Networking Experiments and Technologies*, 13–24. ACM.

Kazemian, P., M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. 2013. "Real Time Network Policy Checking Using Header Space Analysis". In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 99–112. Berkeley, CA, USA: USENIX Association.

Khurshid, A., W. Zhou, M. Caesar, and P. Godfrey. 2012. "Veriflow: Verifying Network-wide Invariants in Real Time". *ACM SIGCOMM Computer Communication Review* 42(4):467–472.

Kumar, R., and D. M. Nicol. 2016. "Validating Resiliency in Software Defined Networks for Smart Grids". In *Smart Grid Communications (SmartGridComm), 2016 IEEE International Conference on*, 441–446. IEEE.

Mai, H. 2011. "Diagnose Network Failures via Data-plane Analysis". https://www.researchgate.net/publication/49176983_Diagnose_network_failures_via_data-plane_analysis.

Open Networking Foundation ONF. "OpenFlow Switch Specification 1.3". "https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf".

Yuan, L., H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra. 2006. "Fireman: A Toolkit for Firewall Modeling and Analysis". In *Security and Privacy, 2006 IEEE Symposium on*, 15–pp. IEEE.

Zeng, H., S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. 2014. "Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks". In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 87–99. Berkeley, CA, USA: USENIX Association.

AUTHOR BIOGRAPHIES

DAVID M. NICOL is the Franklin W. Woeltge Professor of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign, and Director of the Information Trust Institute. He is the PI for two national centers for infrastructure resilience: the DHS-funded Critical Infrastructure Reliance Institute, and the DoE funded Cyber Resilient Energy Delivery Consortium, and is the Director of the Advanced Digital Sciences Center in Singapore, and PI of its Trustworthy and Secure Cyber Plexus program. His research interests include trust analysis of networks and software, analytic modeling, and parallelized discrete-event simulation, research which has lead to the founding of start-up company Network Perception, and election as Fellow of the IEEE and Fellow of the ACM. He is the inaugural recipient of the ACM SIGSIM Outstanding Contributions award. He received the M.S. (1983) and Ph.D. (1985) degrees in computer science from the University of Virginia, and the B.A. degree in mathematics (1979) from Carleton College. His email address is dmnicol@illinois.edu.

RAKESH KUMAR is a Ph.D. candidate in Electrical and Computer Engineering at University of Illinois at Urbana-Champaign (UIUC). He received the M.S. (2010) and B.S. (2007) degrees from UIUC and FAST-NU Karachi respectively. His research interests include communications, networking and distributed computation. His email address is kumar19@illinois.edu.