Ultimate Forwarding Resilience in OpenFlow Networks

Christopher Hannon
Illinois Institute of Technology
10 West 31st Street
Chicago, Illinois, 60616
channon@hawk.iit.edu

Dong Jin
Illinois Institute of Technology
10 West 31st Street
Chicago, Illinois, 60616
dong.jin@iit.edu

Chen Chen
Argonne National Laboratory
9700 S. Cass Avenue
Argonne,IL, 60439
morningchen@anl.gov

Jianhui Wang
Argonne National Laboratory
9700 S. Cass Avenue
Argonne,IL, 60439
jianhui.wang@anl.gov

ABSTRACT

Software defined networking is a rapidly expanding networking paradigm that aims to separate the control logic from the forwarding devices. Through centralized control, network operators are able to deploy and manage more efficient forwarding strategies. Traditionally, when the network undergoes a change through maintenance, failure, or cyber attack, the centralized controller processes these events and deploys new forwarding rules reactively. This work provides a strategy that does not require a controller in order to maintain connectivity while only using features within the existing OpenFlow protocol version 1.3 or greater. In this paper we illustrate why forwarding resiliency is desired in OpenFlow networks and provide an algorithm that computes the flow entries required to achieve maximal forwarding resiliency in presence of both multiple link and controller failures on any arbitrary network.

1. INTRODUCTION

OpenFlow is a standard that defines the communication and interaction between the forwarding and control layers of the SDN architecture. The protocol defines the technical specifications required by current SDN switches to support the interaction from OpenFlow controllers to the forwarding switches. SDN provides many features to improve network operations, including enhanced global visibility, greater customizability, and easier deployment. While historically motivated through data center management to optimize performance, minimize cost, and maximize utilization [11], [2], [9], SDN is being proposed for various other networks such as industrial control systems for increased efficiency, enhanced cyber security, and industry specific applications [6], [16].

Industrial control systems, such as the modern power grid, require networks that are resilient to cyber attack as well as natural disasters that may affect both the communication network as well as the power network. Fault tolerance is a well studied problem in computer networks. When a network is said to be fault tolerant, a communication channel from source to destination can remain established even if an intermediate link fails. In OpenFlow, a network relies on the central controller to make decisions for the data plane and then install the forwarding logic onto the switches with commands issued from the controller.

One widespread and growing networked industrial control system, the modern power system, is composed of heterogeneous networks, including PLC, Ethernet, cellular, and radio communication media. Redundant links are required to enable backup paths for traffic to ensure resiliency. In such heterogeneous networks, SDN is being proposed for resilient communications [3] and as a design architecture [12] to ensure successful and secure operations of the modern power system in the event of cyber attack and cyber errors. Additionally, other applications have been evaluated utilizing SDN in smart grid communications such as multi-rate multicast for Phasor Measurement Unit communications [8] further motivating the need for resilient communication in such networks that rely on accurate and reliable data streams for control purposes.

Communication resilience in the smart grid is important to the power applications that require high availability and minimal packet loss in the event of communication disruption. For example, power protection elements such as relays require low latency and are highly affected by long recovery times in the event of link failure during management events. As communications in the grid evolve to connectionless UDP traffic as the California ISO reports in [16], high availability becomes a significant design goal to minimize packet loss.

In this paper, we present a forwarding framework during multiple link failures with the following objectives: high availability, fast recovery time, and easy deployment. Although fault tolerance is native to SDN, the centralized controller opens up a new interface for cyber attack or failure. Thus, in presence of controller failure, SDN is at a disadvantage compared to traditional decentralized network recovery algorithms such as spanning tree protocols.

Our solution generalizes maximal resiliency for communication between hosts for mission critical traffic within OpenFlow networks. We guarantee that the same number of links

ACM ISBN 999-9-9999-9999-9/99/99...$15000000.00
DOI: http://dx.doi.org/11.111/1111111.1111111

Strategy	# Edges needed to remove in order to disconnect s and t	Recovery time	Description
This paper	|min-cut-st|	$O(ms)$	requires size of min-cut with s and t on disjoint subgraphs
k-disjoint-backup	$1 \leq k \leq$ |min-cut-st|	$O(ms)$	can be between 1 and size of|min-cut-st|
Backup at every intermediate switch	$2 \leq k \leq$ |min-cut-st|	$O(ms)$	if a backup path has a link failure, then at minimum 2 edges can be removed; if there are no backup paths then $k = 1 =$|min-cut-st|
Controller Based	|min-cut-st|	$O(ms)$	requires use of controller
RSTP / STP	|min-cut-st|	$O(seconds)$	slow and network cannot perform multi-path forwarding

Table 1: Comparison of the existing failover schemes in event of link failure. Our approach guarantees the same number of links as a controller based recovery scheme as well as traditional decentralized mechanisms. The advantage of our approach is that we do not require a controller to guarantee this performance.

may fail as with a controller-based recovery strategy, specifically $|\text{min-cut-st}| - 1$, which is the minimum cut of the network, i.e., fewest number of edges required to fail to create two disjoint subgraphs with s and t on separate subgraphs. Therefore, as long as there exists a path from source to destination, our modified DFS algorithm, which transforms the network graph to a tree structure, will enable the correct forwarding independent of network topology. This work shows that the same resiliency properties can be established during controller failure as without or compared to decentralized spanning tree solutions. A controller based approach for recovery is a run-time solution for network link failure recovery. In our work we provide a compile-time recovery strategy that predetermines all backup paths and deploys them proactively. The related work only guarantees resilience in presence of single link failure. Therefore, for mission critical traffic in control systems, connection can be guaranteed. In this work, we design the algorithm and data structure to convert the graph representation of the network into flow tables and group tables on the OpenFlow switches that enforce the maximally resilient network property.

Section 2 discusses the related work, Section 3 shows how our strategy for providing maximal forwarding resiliency which extends from a single link failure and the challenges associated with multiple link failure. Section 4 presents our all-paths algorithm that changes the graph representation into an N-ary tree representation which allows us to easily determine the OpenFlow rules necessary for the network. Section 5 concludes our work and proposes future research directions for resilient forwarding in OpenFlow networks.

2. RELATED WORK

Many of the current strategies for fault tolerance and link failure rely on the central controller to calculate the optimal path from source to destination using a global view of the network topology updated with the failed edge. In [13] and [10] the authors show how to utilize the controller to update the network in event of link failure. When a link fails, the event is sent to the controller through the OpenFlow protocol, the controller then computes new paths for each flow and modifies the necessary flow rules on the switches to create new flows. This is the most straightforward approach using a centralized control architecture. Calculating the new optimal path however requires time to process the updated graph and time to install the new forwarding logic onto the corresponding switches. During this time packets may be lost.

In [15] the authors utilize a hybrid approach through backup routes and centralized recomputation by using two steps. In the first step, the forwarding devices locally detect a failure and resort to a failure mode where precomputed backup paths are taken. The second step is for the controller to compute the optimal paths based on that link failure. This ensures that a functional while non-optimal path can take over faster than the controller to recompute a new path. The authors extend the Open vSwitch to include the Bidirectional Forwarding Detection to report in the fast failover liveliness check. The design for the fast failover rules is to provide an alternative path, i.e., backup port to send the traffic to in the event of a failure. If the switch does not have a backup bath, then a backtracking method is used to return the packet to the previous switch where the process is repeated recursively. The authors compute backup paths so that each intermediate hop computes a backup path locally, implemented using the failover group table entry. If no backup path exists, the traffic gets sent back one hop until a backup path is located. They however do not consider the case of multiple link failures as our work does. The minimum number of link failures to disconnect the source and destination is two.

Similarly, [1] also implements a hybrid approach that uses the same mechanisms of [15] and show they can achieve recovery in 50 ms. These works share the same limitations of single link failure.

For the reliability of SDN controllers, ONOS [4], and [5], illustrate the benefits of distributed controllers. One such benefit is reliability during controller failure. In our work, we consider failure from a single SDN controller.

This paper presents a strategy that computes every possible path from source to destination and creates a flow table and group tables that provide ultimate resiliency in the face of multiple link failures. The contributions of the algorithm in this paper are as follows:

- rules are proactively installed, ensuring that the recovery process does not require a controller
- the approach enables multiple link failure resiliency in the network
- the approach minimizes packet loss in the event of multiple link failures

In this paper, we describe an algorithm that enables maximal resilience to link failures and evaluate empirically the effect it has on the switches' forwarding tables, as well as congestion and latency in the network. This algorithm requires no changes to the OpenFlow protocol and is computed offline and proactively installs rules onto the switches.

3. MOTIVATION

This section considers a single path from source to destination rather than every flow on the network simultaneously. The s node represents the source while the t node is the destination node. G represents the network with switches as nodes and links as edges. This notation will be used throughout the paper. The following examples provide the motivation and explain the mechanisms of the maximally resilient forwarding algorithm described in the next section which generalizes the process for arbitrary networks.

3.1 Single Link Failure Recovery

During a link failure, the switches with ports connected to the failed link, detect the failure. Upon detection, fast failover group tables will allow for a different rule set to be placed on the packets destined to failed link using the liveliness monitor feature. Group table actions consist of sending the packet to a new port, a new group table, or even back to the flow tables [14]. In this work, the group table for fast failover is used to monitor the outgoing port and forwards to a backup in case of failure on the main path.

In the network depicted in Figure 1, the main path is $s \rightarrow s1 \rightarrow s2 \rightarrow s3 \rightarrow s4 \rightarrow s8 \rightarrow t$. When the edge from $s1$ to $s2$ fails, and switch $s1$ detects this through its liveliness monitor for port y in the failover table and resorts to port x. Therefore the traffic can now be routed through $s \rightarrow s1 \rightarrow s5 \rightarrow s6 \rightarrow s7 \rightarrow s8 \rightarrow t$ and can do this without any reactive influence from a centralized controller. The link failure can be detected using a mechanism such as bidirectional forwarding detection.

Only relying on failover tables to chose an alternate route locally may not satisfy every case. For instance, if a different edge experiences a failure ($s2 \rightarrow s3$ on the primary path), $s2$ does not have any backup routes locally as it only has a degree of 2. Therefore a more complex scheme must be used involving backtracking. This backtracking is inspired from the crankback algorithm used in MPLS. Crankback [7] allows for sending a message backward when setting up a connection in routing between networks. By sending the message backwards, a new path can be established that meets the requirements of the flow at the cost of latency and bandwidth consumption.

In our case, if a path becomes infeasible (i.e. no failover paths locally) the packet gets forwarded back to the previous switch and is forwarded from there. The rule for a returned packet is the same as it would be as if the edge it just returned on was failed. This process of crankback can happen recursively if there are no backup links to send to. If the edge ($s2 \rightarrow s3$) fails, $s2$ requires a group table to monitor the liveliness of port x and send through y as a backup. $s1$ would need a rule in the flow table to match on incoming port y and send to x.

In this case, the traffic flows from $s \rightarrow s1 \rightarrow s2 \rightarrow s1 \rightarrow s5 \rightarrow s6 \rightarrow s7 \rightarrow s8 \rightarrow t$. Although $s1$ must process the packet twice, the information stream from s to t remains intact and does this without reactive influence from a controller. Additionally, if the link from $s2 \rightarrow s3$ comes back online, this optimal path would be taken automatically without any configuration changes due to the liveliness status of group tables. In all the related work, the network resiliency without controller influence stops here, however the examples presented only experience a single link failure. Our work shows how this approach can be extended for link failure in $|min-cut(G,s,t)|-1$ edges regardless of network size or topology where $|min-cut(G,s,t)|$ is the size of the min-cut where s and t are on separate disjoint subgraphs of G.

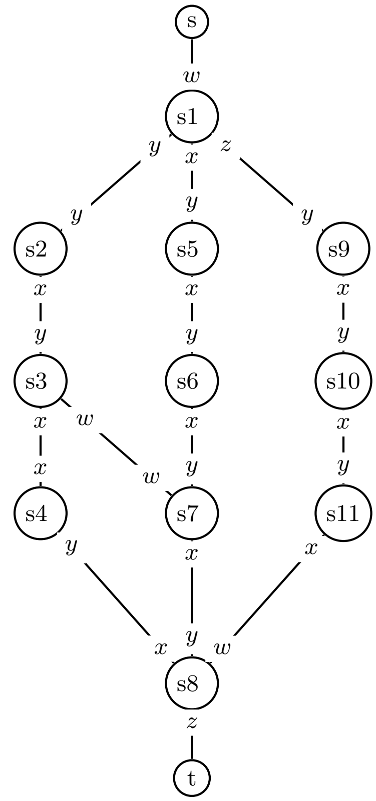

Figure 1: The main path is $s \rightarrow s1 \rightarrow s2 \rightarrow s3 \rightarrow s4 \rightarrow s8 \rightarrow t$. Case 1: $s1 \rightarrow s2$ failed. In this case, the group table installed on $s1$ of type failover will be sent to the next live port, x, to complete the flow $s \rightarrow s1 \rightarrow s5 \rightarrow s6 \rightarrow s7 \rightarrow s8 \rightarrow t$. Case 2: $s2 \rightarrow s3$ failed. In this case, the group table installed on $s2$ returns the packet to $s1$ in crankback style routing. Case 3: Multiple link failure. $s4 \rightarrow s8$ and $s7 \rightarrow s8$ failed. In this case, the packet has the ultimate path $s \rightarrow s1 \rightarrow s2 \rightarrow s3 \rightarrow s4 \rightarrow s3 \rightarrow s7 \rightarrow s3 \rightarrow s2 \rightarrow s1 \rightarrow s9 \rightarrow s10 \rightarrow s11 \rightarrow s8 \rightarrow t$. Table 2 shows rules to perform this forwarding. Case 4: Multiple link failure. If edges $s2 \rightarrow s3$, $s7 \rightarrow s8$ and $s11 \rightarrow s8$ failed, the rules in Table 2 do not find the successful path. A more general approach is necessary.

3.2 Multiple Link Failure Recovery

The introduction of loops and more complicated networks requires additional logic when faced with multiple link failures. For example, consider if the network in Figure 1 contains multiple link failures, $s4 \rightarrow s8$ and $s7 \rightarrow s8$ with corresponding flow and group tables defined in Table 2. Switches $s2, s4, s9, s10, s11$, all follow the same format as the table $s1$:FT1 and are excluded for brevity in Table 2. The failover

Table 2: Switch Configurations for the Example in Figure 1

Flow Table s_1:FT1			Flow Table s_2:FT1			Flow Table s_7:FT1		
FlowID	Match Field	Action	FlowID	Match Field	Action	FlowID	Match Field	Action
Flow 1	InPort: w	Group GT1	Flow 1	InPort: y	Group GT1	Flow 1	InPort: w	Group GT1
Flow 2	InPort: y	OutPort: z	Flow 2	InPort: x	OutPort: y	Flow 2	InPort: x	OutPort: w
Group Table s_1:GT1 Type: FF			**Group Table s_2:GT1 Type: FF**			**Group Table s_7:GT1 Type: FF**		
Bucket ID	WatchPort	Action	Bucket ID	WatchPort	Action	Bucket ID	WatchPort	Action
Bucket 1	y	OutPort: y	Bucket 1	x	OutPort: x	Bucket 1	x	OutPort: x
Bucket 2	z	OutPort: z	Bucket 2	y	OutPort: y	Bucket 2	w	OutPort: w
Flow Table s_3:FT1			**Group Table s_3:GT1Type: FF**			**Flow Table s_8:FT1**		
Flow ID	Match Field	Action	Bucket ID	WatchPort	Action	Bucket ID	WatchPort	Action
Flow 1	InPort: y	Group GT1	Bucket 1	x	OutPort: x	Flow 1	InPort: *	OutPort z
Flow 2	InPort: x	OutPort w	Bucket 2	w	OutPort: w			
Flow 3	InPort: w	OutPort y	Bucket 3	y	OutPort: y			

rule returns the packet to the previous switch to support crankback routing.

In this network the main path from s to t is $s \rightarrow s1 \rightarrow s2 \rightarrow s3 \rightarrow s4 \rightarrow s8 \rightarrow t$. The path from $s4 \rightarrow s8$ is failed, therefore $s4$ routes the traffic back according to its failover group. $s3$ has a rule to route $s3 \rightarrow s7 \rightarrow s8 \rightarrow t$. However, upon reaching $s7$, port x reports that the edge from $s7 \rightarrow s8$ is also down. The fast failover rule routes the frame back to $s3$ according to its failover rule. $s3$ has nowhere to send the packet so using Flow 3 returns it to $s1$ via $s2$. At this point since the path between $s5$ and t (without going through $s1$) is blocked, the switch $s1$ should forward the frame to $s9 \rightarrow s10 \rightarrow s11 \rightarrow 8 \rightarrow t$. Therefore $s1$ has rule FT2 to perform this action. In this case the packet is able to arrive at t with an ultimate route of $s \rightarrow s1 \rightarrow s2 \rightarrow s3 \rightarrow s7 \rightarrow s3 \rightarrow s2 \rightarrow s1 \rightarrow s9 \rightarrow s10 \rightarrow s11 \rightarrow s8 \rightarrow t$. While this is not the most direct route, this strategy successfully enables forwarding in the presence of 2 failed links on the primary and backup paths.

This strategy is specifically tailored to work with the failed edges. The previous rule set fails if the following edges ($s2 \rightarrow s3$, $s7 \rightarrow s8$, $s11 \rightarrow s8$) are down even though there exists a path $s \rightarrow s1 \rightarrow s5 \rightarrow s6 \rightarrow s7 \rightarrow s3 \rightarrow s4 \rightarrow s8 \rightarrow t$. Therefore a few additional rules so that $s7$ can check to see if the route through $s3$ is a candidate can be added to the current strategy allowing for this path to be tried.

However a problem quickly emerges. With the additional rules, specifically the $s7 \rightarrow s3$, an infinite loop is easily created. A loop can occur if the link from $s4 \rightarrow s8$ is down, the usual behavior is to send this packet back to $s2$ using the crankback technique. Eventually this packet will return to s through $s3 \rightarrow s2$ or be dropped if that link is down. This brings motivation for adding a mechanism to remember the history of the switches that have already processed the packet as well as treating the switch differently depending on the source to ensure proper crankback. The challenge is that OpenFlow does not support switch to switch communication for this problem. We utilize the packet header space as a ledger which we show in the following section.

4. ALGORITHM

We present a modified depth first search algorithm that finds all paths from source to destination recursively. In this modified DFS algorithm, the data structure created is an N-ary tree that represents all paths that can lead to the destination node. In this N-ary tree, the root node is the source and each leaf is the destination t. Each intermediate node has a set of children ≥ 1 that can reach the destination t. Not every edge from an intermediate node is represented in the data structure as some may not have a path to the destination. Loops in the network may introduce multiple paths p_1, p_2 from source to destination, where $p_1 \cap p_2 \neq \emptyset$. This implies that p_1 and p_2 are not disjoint. Such occurrences create duplicate nodes in the tree that are marked accordingly.

The first path on node n is from parent to child 1, if child 1 port is down, then it goes to child 2 and this process continues until there are no children remaining of n. If this is the case there is no path from n to t so the packet is sent back to the parent in the style of crankback routing. Similarly, if there is a downstream link down from $n.child_k$ to t then the packet returns to n and should be sent to $n.child_{k+1}$. However a different fast failover table must be used to ensure that if port $n.child_{k+1}$'s port is down that the fast failover sends the packet to $n.child_{k+2}$ if possible. Therefore, there exists a flow entry for each port sending it to a fast failover table. The flow table for intermediate node i follows the format in Table 3.

As seen in table 3, each InPort sends the packet to a different group Table. Because failover rules can only forward in case of local link failures, there needs to be a different group table for fast failover for each port on a switch to prevent loops. The group table 1 on a node m is the table that is arrived from the port connected to the parent of m. Subsequent group tables are for each child of m. Each child's group table is a subset of group table 1. For example, if a node m has three children, the first failover rule in group table 1 will be child 1 then child 2 then child 3. If downstream of child 1 there is a link failure and the packet returns to the node m, the packet should be forwarded to child 2. Therefore the group table for m that matches child 1's incoming port is group table 1 less the entry for child 1. If this link is broken the fast failover should send it to child 3. If the original fast failover group is used, then the packet would be sent back down to child 1 where it is known that there is no path thus causing a loop. Hence, the group table for child 2 is group table 2 less the entry for child 2. Tables 4-6 illustrate the general case for forwarding with multiple children of a node with a total degree of $n+1$.

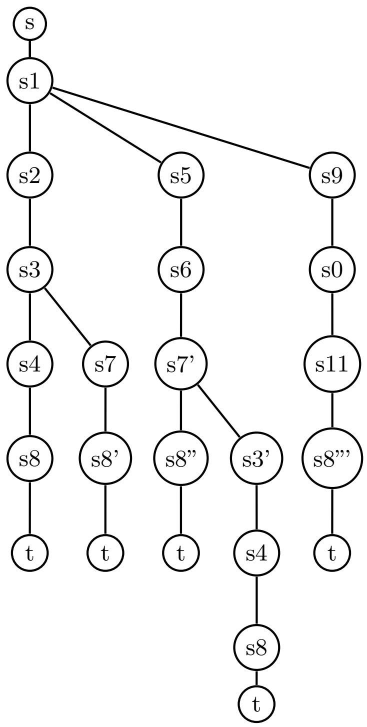

Figure 2: The resulting N-ary tree from Algorithm 1 on the network in Figure 1. The nodes that are depicted with ' (prime) are backedges. The rule from a node with child and with prime notation checks the VLAN field to see if the switch has processed the packet previously; if not, uses the VLAN tag to return the packet to the correct parent, thus preventing forwarding loops.

4.1 Tag

Many networks that enable resiliency will contain loops. The major difficulty for multiple paths routing through loops is that there are already rules on the switches making it difficult to keep track of which routes have been traversed before. When a crossedge points to a node $s \to v$ where v is the node already explored from, a set of rules need to be in place to allow the packet to return to s through the crossedge if no path can be found from $v \to t$. This is done by *tagging* the edge. Tagging the edge with a VLAN through the PUSH VLAN and SET VLAN actions in OpenFlow, the crossedge can be identified and corresponding rules can be applied to the v node, allowing the VLAN ID to be used to match on in the flow tables of the switches along with port number. Note that any other field can be used, but we use the VLAN for example to require only layer 2 information. In this case, the children of v are viable candidates to reach t however if v has already processed this packet earlier, then there is no possible way for the packet to reach t according to lemma 1.

Lemma 1: If a packet p travels through a crossedge $b = (s \to v)$, where v has already been visited, there is no way for p to reach t through b.

Proof: Suppose that it is possible for p to reach t through

Algorithm 1 Modified DFS Algorithm

function ALGO($G=(V,E), s, t, M$)
  for all $v \in N(s)$ **do**
    if $v \in T$ **then** ▷ Node visited before
      if $v \in M$ **then** ▷ Case 1: backedge
        Return
      else if $v \in R$ **then** ▷ Case 2: crossedge
        $Tag(v, s, M)$
      else ▷ Case 3:no path to t
        Return
      end if
    else
      $M = M \cup \{v\}$ ▷ Add v to M
      $M[s].child = v$
      if $v = t$ **then** ▷ Case 4: reached t
        $R = R \cup M$ ▷ Add M to T
      else ▷ Case 5:unexplored node
        $T = T \cup \{v\}$
        $Algo(G, v, t, M)$
      end if
    end if
  end for
end function

R={ } ▷ N-ary tree, contains all paths
M={ } ▷ tmp working tree
T={ } ▷ list of marked nodes
s = source
t = destination
Algo(G,s,t,M)

function TAG(v, s, M)
  $A = copy(v)$ ▷ creates copy of v
  $M = M \cup \{A\}$ ▷ add A to M
  $M[s].child = A$
  $Copy\_down(v, A, M)$ ▷ child of v **will** reach t
  $Copy\_up(v, A, M)$ ▷ parent of v **may** reach t
end function

function COPY_DOWN(v, A, M)
  for all $v' \in children(v) \notin M$ **do**
    $A' = copy(v')$
    $M[A].child = A'$
    if $v = t$ **then**
      $R = R \cup M$
    else
      $Copy\_down(v, A', M)$
    end if
  end for
end function

function COPY_UP(v, A, M)
  if $v.parent \in M$ **then**
    Return
  else
    $A' = copy(v.parent)$
    $M[A].child = A'$
    $Tag(A', v.parent, M)$ ▷ treat as crossedge
  end if
end function

Table 3: Flow Table for intermediate node si

FlowID	Match Field(s)	Action
FT1	InPort: $si.port.parent$	Group $GT1$
FT2	InPort: $si.port.child_1$	Group $GT2$
FTk+1	InPort: $si.port.child_k$	Group $GTk+1$
FTn+1	InPort:$si.port.child_n$	OutPort $si.port.parent$

Table 4: Group Table 1 for switch sn

Group Table ID: GT1		Type: FF
Bucket ID	WatchPort	Actions
Bucket 1	$sn.port.child_1$	OutPort: $sn.port.child_1$
Bucket i	$sn.port.child_i$	OutPort: $sn.port.child_i$
Bucket n	$sn.port.child_n$	OutPort: $sn.port.child_n$
Bucket n+1	$sn.port.parent$	OutPort: $sn.port.parent$

Table 5: Group Table k for switch sn

Group Table ID: GT2		Type: FF
Bucket ID	WatchPort	Actions
Bucket 1	$sn.port.child_k$	OutPort: $sn.port.child_k$
Bucket i-k	$sn.port.child_i$	OutPort: $sn.port.child_i$
Bucket n-k	$sn.port.child_n$	OutPort: $sn.port.child_n$
Bucket n-k+1	$sn.port.parent$	OutPort: $sn.port.parent$

a crossedge $b = (s \rightarrow v)$ where v has already processed p, the packet p will have to reach t either through a child of v or the parent of v. If p reaches t through a child of v then this path would have been discovered earlier at the first processing of p, similarly p reaching t through the parent of v would require the parent to have a path to t where the edge $v \rightarrow v.parent$ acts as the crossedge making this argument recursive until reaching the root.

Using Lemma 1, an additional rule is installed on both switch v and s in a crossedge $s \rightarrow v$. The rule installed on v is to *sign* the frames that it processes upon crankback. The rule installed on s is to check for v's signature. If s sees $v's$ signature, then it uses the same group table entry as if sent back from $v \rightarrow s$. This feature of using the VLAN field as a ledger enables the passing of primitive messages between switches to allow future switches to act differently based on the past history of the packet. The ability to match using a mask allows for checking a single bit (or arbitrary number bits) of a VLAN ID. A similar mechanism allows the setting of a single bit in the VLAN ID using OpenFlow.

4.2 CopyDown / CopyUp

The copy down routine allows for the copying of part of the existing N-ary tree into the current working tree. Because the existing N-ary tree has all leaves as t, we can copy each branch where the crossedge has reached. This is this the first part of handling a crossedge. The nice feature of the N-ary tree is that there are no loops so that copying the sub tree at the cross edge enforces this property. Because the tagging routine tags the destination of the crossedge, this subtree will only be traversed if the node has not already been visited by the packet. If it has not already been visited, it utilizes the VLAN to ensure the correct path, i.e., back through the crossedge is taken if no path to t can be found. On the other hand, if a packet p travels through a crossedge $b = (s \rightarrow v)$, where v has not already been visited, it means that between v and the root there is a link failure. In this case it is possible for p to reach t through $s \rightarrow v$. Therefore the algorithm accounts for this case by treating the edge $e = (v, v.parent)$ as a crossedge recursively.

4.3 Analysis

In this work, we do not consider the transient state of links failing, we consider the steady state where each link may be in one of two possible states: okay and failed. If links are reported as failed and later report as alive again, then the subsequent packets will be processed as normal because no rule is ever deleted, i.e., timeout on all rules set to infinity. Given the graph in Figure 1, the size of the minimum cut of the graph with s and t on disjoint subgraphs is 3, if we exclude the link $s \rightarrow s1$ and $s8 \rightarrow t$. This means that the minimum number of links that can be removed to disconnect the source and destination is 3 links, i.e., $s1 \rightarrow s2$, $s5 \rightarrow s6$ and $s11 \rightarrow s8$. The maximum number of edges that can be removed is 9, as long as there exists a path from source to destination the forwarding strategy we propose will find the path. Compared to a controller based recovery strategy, we can guarantee that our strategy will have equal performance. The benefit of our approach is we can guarantee this at compile-time rather than at run-time since our backup strategy is installed proactively. In addition to SDN based strategies, the traditional mechanisms for network recovery, namely spanning tree protocols, can guarantee the same performance, but take much longer on the order of seconds to re-converge after a link failure. For mission critical traffic this is too long, therefore our recovery strategy using OpenFlow meets our objective of high availability and fast recovery time. Comparing our strategy in this example with the related work, the disjoint paths approach and all backup paths from $s1 \rightarrow s8$ both only find three paths, while our scheme provides all 5 backup paths.

On the other hand, because no rules are ever added or deleted (unless administratively), there will be high overhead in the number of rules needed to be installed on a given switch. For each switch v in the network there will be $deg\ v$ flow table entries and group tables required for a single flow $s \rightarrow t$. Crossedges will require additional $deg\ v$ group tables and flow entries to enable correct forwarding.

The total number of group tables and flow entries is at most $2E$ which is at maximum $O(n^2)$ flow entries and group tables for highly connected networks. Additionally, due to the crankback style of routing there will be additional traffic within the network. Because a packet never returns to the same switch except during crankback, a packet may be transmitted at most twice across any single link. If $flow(s,t)$ represents the quantity of traffic sent from s to t then at maximum each link can experience $2 * flow(s,t)$ traffic. Finally, due to the crankback routing and sub-optimal path selection, if there remains a path between s and t then the latency is at most

$$O(\sum_{e \in E} 2 * prop(e) + \sum_{v \in V} deg(v) * process(v))$$

$prop$ is the propagation delay in the links, $process$ is the processing delay at the switch including the queue delays.

5. CONCLUSION

We have presented a general scheme to implement resilient forwarding in OpenFlow networks with minimal packet loss.

Our scheme works without reactive influence from a controller, enabling larger resilience than traditional SDN approaches. Our algorithm provides the OpenFlow rules for multiple link failures equivalent to a centralize controller approach enabling the hosts to remain connected. Due to the overhead incurred by this method of forwarding in presence of multiple link failure, it is designed for mission critical communication applications as to prevent congestion within the network. Our future work includes researching techniques to reduce the overhead in terms of number of tables on the switches for multiple source-destination pairs through rule compression, increasing the scalability of the VLAN ledger mechanisms, and evaluating the trade-off between overhead and resiliency by formulating this resiliency problem as an optimization problem.

6. REFERENCES

[1] R. Ahmed, E. Alfaki, and M. Nawari. Fast failure detection and recovery mechanism for dynamic networks using software-defined networking. In *2016 Conference of Basic Sciences and Engineering Studies (SGCAC)*, pages 167–170, Feb 2016.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[3] A. Aydeger, K. Akkaya, M. H. Cintuglu, A. S. Uluagac, and O. Mohammed. Software defined networking for resilient communications in smart grid active distribution networks. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2016.

[4] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[5] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed sdn controller. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 7–12. ACM, 2013.

[6] X. Dong, H. Lin, R. Tan, R. K. Iyer, and Z. Kalbarczyk. Software-defined networking for smart grid resilience: Opportunities and challenges. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, CPSS '15, pages 61–68, New York, NY, USA, 2015. ACM.

[7] e. a. Farrel. Crankback signaling extensions for mpls and gmpls rsvp-te. *Network Working Group*, RFC 4920, 2007.

[8] A. Goodney, S. Kumar, A. Ravi, and Y. H. Cho. Efficient pmu networking with software defined networks. In *Smart Grid Communications (SmartGridComm), 2013 IEEE International Conference on*, pages 378–383, Oct 2013.

[9] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. Elastictree: Saving energy in data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association.

[10] H. Kim, M. Schlansker, J. R. Santos, J. Tourrilhes, Y. Turner, and N. Feamster. Coronet: Fault tolerance for software defined networks. In *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–2, Oct 2012.

[11] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zupdate: Updating data center networks with zero loss. *SIGCOMM Comput. Commun. Rev.*, 43(4):411–422, Aug. 2013.

[12] S. Rinaldi, P. Ferrari, D. BrandÃčo, and S. Sulis. Software defined networking applied to the heterogeneous infrastructure of smart grid. In *Factory Communication Systems (WFCS), 2015 IEEE World Conference on*, pages 1–4, May 2015.

[13] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester. Enabling fast failure recovery in openflow networks. In *Design of Reliable Communication Networks (DRCN), 2011 8th International Workshop on the*, pages 164–171, Oct 2011.

[14] The Open Networking Foundation. OpenFlow Switch Specification, Jun. 2016.

[15] N. L. M. v. Adrichem, B. J. v. Asten, and F. A. Kuipers. Fast recovery in software-defined networks. In *2014 Third European Workshop on Software Defined Networks*, pages 61–66, Sept 2014.

[16] T. Williams. The role of software-defined networking and wan virtualization in securing scada systems: An alternative to pki and vpns for securing telemetry data.