Path Restoration in Source Routed Software Defined Networks

Saumya Hegde
Dept. of CSE
NITK, Surathkal, India.
hegdesaumya@gmail.com

Shashidhar G. Koolagudi
Dept. of CSE
NITK, Surathkal, India.
koolagudi@yahoo.com

Swapan Bhattacharya
Dept. of CSE
Jadavpur University, Kolkata, India.
bswapan2000@yahoo.co.in.

***Abstract*—Software defined networks have a central controller and central view of the network, allowing for source routing to be used as a scalable routing technique instead of the traditional destination based forwarding. However, with source routing the switches are reduced to simple forwarding devices, incapable of finding alternate paths in the event of link failures. In this paper we look at techniques to provide resiliency when packets are in transit and a network link failure occurs. Path restoration is one such mechanism wherein we use a bypass path for the failed link. Such bypass paths are stored locally on each of the switches, for all of its outgoing links. This mechanism ensures that the recovery mechanism is scalable since it avoids contacting the controller and takes local corrective measures. We have proposed two approaches for storing the bypass paths. In the first method we store the bypass path between all pairs of nodes. In the second method we store the bypass paths between few selected nodes. These node are the two hop neighbors, chosen using either the two colorable graph approach or the vertex cover approach. Our analysis shows that the second method, using the vertex cover approach reduces the total number of bypass paths stored, without compromising the resiliency. Also our solutions are topology dependent and not path dependent, allowing for most of the computations to be done proactively.**

Keywords: Software Defined Network, Data Plane Resilience, Path Restoration

I. Introduction

Software defined network (SDN) has caused a paradigm shift in networking due to its radical concept of separating the control plane from the data plane of the switches and placing them on a central controller. The controller has global view of the network and is responsible for installing the forwarding rules in the switches. SDN is gradually seeing real world deployment in several data centers. Source routing or label based routing is best suited for such centrally controlled networks. In destination based routing the forwarding information is stored in the switches, while in source routing the forwarding information is stored in the packet itself.

A basic requirement of SDN deployed in data center networks is high availability and resilience to failures. In order to cater to this the network must remain connected in the event of a networking element failure and packet forwarding must be carried out with as little disruption as possible. In this paper we focus on the issue of handling packets in transit when a link failure occurs.

Source routing in software defined data center networks (SDDCN) is scalable in terms of both the packet overhead and switch memory requirement because of the following two reasons

1) The length of paths in data centers is short, at most 6 - 8 hops[3]. This implies that even though source routing requires that the forwarding information must be stored in the packet, this overhead is limited to 8 hop information.
2) There is no need to store forwarding information on the intermediate switches. This makes the network scalable since the TCAM memory on these programmable switches are limited and expensive.

One of the issues with source routing is the fate of packets in transit when a link failure occurs. This is because unlike destination based forwarding where the switches learn the alternate paths on reconvergence, in source routing the switches are dumb forwarding devices which simply forward packets as per the next hop information in the packet. Hence for source routing to be a viable solution, the issue of packet forwarding in the event of link failure must be addressed.

There are two main techniques by which networks can handle packet forwarding when a link failure occurs, as follows:

I Path protection: Resend the packet from the source on an alternate, link disjoint path. The packet which was in transit is dropped.
II Path restoration: Continue to forward the packet in transit, without dropping it, via a path which bypasses the failed link, to a subsequent node along the path.

In this paper we explore the second technique. The information about the path which bypasses the broken link can be obtained using either of the following:

i The switch which detects one of its adjacent link has failed, contacts the controller and gets the bypass path
ii The switch stores the bypass paths for all of its neighbors in its local memory

The first method i has the problem that it takes time to contact the controller and also places extra load on the controller. It is therefor not a scalable option. On the other hand the second method ii requires storage space on the memory constrained switches. However the second option of path restoration is

faster and we propose a solution based on it and evaluate it in the following sections.

The rest of the paper is organized as follows. Prior work on data plane resiliency in software defined networks and our motivation for choosing this problem is presented in section 2. We formulate the problem in section 3. We propose our solution and analyze it is section 4 and 5. Some concluding remarks are given in section 6.

II. Related Work

Although [4] was not meant for SDN, it employs port switching to implement source routing, instead of using the switch identifications. They also ensure source routing is scalable by carrying out the path computation in the servers and keeping the routers stateless. One of the first papers to discuss source routing in SDN domain is [8]. They store the output port numbers of switches along the route for the packet, in the packet header. As the packet enters each switch, the reverse path is calculated, by storing the input port number via which the packet enters a switch, in the packet header. When a link failure occurs the switch chooses a secondary path using the information stored locally in the switch. With source routing the controller has to pass the path information only to the ingress switch and not all the switches along the path of a packet, as is the case in destination based forwarding. In [9] the authors discuss a source routing that is resilient to link failure. The architecture proposed here has a Plinko forwarding function which installs the secondary routes that protect the primary routes. In [6] the authors present a source routing algorithm that is resilient. They achieve this by computing a secondary path for every path and storing it in the packet header along with the primary path. They test this on fat tree topologies in data center environments. They extend this work for arbitrary topologies in [7]. In [1] the authors discuss how certain traditional header may not have a role to play in a SDN environment and how the removal of such headers will reduce the latency. In [2] the authors discuss scalability issues in SDN. They propose that the separation of edge and core controller. Here the edge controller carries out path installation functionality and the core controller carries out the network status collection functionality. Such an approach can improve scalability since the edge controller can quickly install paths as it is not burdened with network status collection functionality. Although [1], [2] are not works carried out on source routing, they do help us understand how source routing can be improved by replacing the unused header fields with source route information and how separating the status collection from path installation helps make SDN scalable. Source routing makes the SDN scalable by reducing the number of entries in the flow tables and reducing the number of rule installation the controller will have to make on the switches. Since the packets carry the forwarding information making use of multiple paths, involves only changes in header and not adding forwarding rules into the flow tables. This helps to spray packets of flows across multiple paths, making it fair for both elephant and mice traffic. This solution is presented in [5]

A. Motivation

Source routing is ideal for forwarding packets in networks where the control plane is decoupled and centralized such as in SDN. However, one of the main challenges with source routing is how packets in transit are handled when a link failure occurs. Since the packets themselves carry the path information the switches are simple forwarding devices with no route information. The switches can obtain the path information from the controller, but this is a time consuming process. Hence when a link failure occurs, the adjacent switch which detects the failure, must forward the packets without contacting the controller. This paper aims to provide data plane resiliency in the event of link failure. The solutions provided here are based on path restoration. The packet forwarding considered uses label based forwarding or source routing. The network that we consider is a large biconnected network under a single administrative domain and data center networks fulfill these requirements. The solution presented here provides protection against single link failure only.

III. Resilient Data Plane

In this paper we propose a resilient data plane in a source routed software defined data center network, when a link failure occurs, using the path restoration mechanism, while minimizing the communication between the switch and the controller and also minimizing the memory requirements on the switches.

We identify the following requirement for resilient source routing in SDDCN

- R1 In the event of link failure, packets in transit are re-routed correctly without having to retransmit them i.e. ensure path restoration.
- R2 Take local corrective measures so as to minimize the communications between the switch and the controller.
- R3 Minimize the average memory requirement of the switches in the network.
- R4 The solution that we propose must be topology dependent and not path dependent

A. The network model

A data center network is modeled as an undirected graph $G = (V, E)$ where each node v_i in the vertex set $V = \{v_1, v_2, \ldots, v_n\}$ represents a physical switch, $n = |V|$ is the total number of switches and the edge set E represents the links between the switches. Two vertices which are incident with a common edge represent adjacent switches.

B. Notations

Shortest Path: Given a graph $G(V, E)$ let $P(s, d)$ the shortest path from the source s to the destination d be a vertex set

$$P(s, d) = \{v_1, v_2, \ldots, v_m\} \quad (1)$$

where v_i is the i^{th} node in the path and m is the total number of nodes in the path.

Second Shortest Path: Given a graph $G(V, E)$ let $SP(s, d)$ the second shortest path from the source s to the destination d be a vertex set

$$SP(s, d) = \{v_1, v_2, \ldots, v_l\} \tag{2}$$

where v_i is the i^{th} node in the path and l is the total number of nodes in the path.

One Hop Neighbor Set: The one hop neighbor set of a node v_i be defined as

$$ON(i) = \{v_1, v_2, \ldots, v_d\} \tag{3}$$

where $v_k \in ON(i)$ is an adjacent node of v_i and $d = |ON(i)|$ is the degree of the node v_i.

Two Hop Neighbor Set: The two hop neighbor set of a node v_i be defined as

$$TN(i) = \{v_1, v_2, \ldots, v_t\} \tag{4}$$

where $v_k \in TN(i)$ is a node at a distance of two hop from v_i and $t = |TN(i)|$ is the total number of nodes at a distance of two hop from v_i.

Vertex Cover Neighbor Set: The vertex cover neighbor set of a node v_i be defined as

$$VN(i) = \{v_1, v_2, \ldots, v_c\} \tag{5}$$

where $v_k \in VN(i)$ is a vertex cover node rechable from v_i without crossing another vertex cover node and $c = |VN(i)|$ is the total number of such neighboring vertex cover nodes of v_i.

One Hop Bypass Path: Given a node v_i and its one hop neighbor $v_k \in ON(i)$ then the one hop bypass path $OB(i, k)$ is given by

$$OB(i, k) = SP(i, k) \tag{6}$$

where SP is the second shortest path from i to k i.e. the shortest path from i to k without considering the direct link.

Two Hop Bypass Path: Given a node v_i and its two hop neighbor $v_k \in TN(i)$ then the two hop bypass path $TB(i, k)$ is given by

$$TB(i, k) = SP(i, k) \tag{7}$$

where SP is the second shortest path from i to k i.e. the shortest path from i to k without considering the direct two hop link.

Vertex Cover Bypass Path: Given a node v_i and its one hop neighbor $v_k \in VN(i)$ then the vertex cover bypass path $VB(i, k)$ is given by

$$VB(i, k) = SP(i, k) \tag{8}$$

where SP is the second shortest path from i to k i.e. the shortest path from i to k without considering the shortest link to the vertex cover neighbor.

C. Problem Definition

Given an edge l between two nodes v_i and v_j as $v_i <-> v_j$ along the path $P(s, d)$ has failed, the problem is defined as finding an alternate path between (s, d) which bypasses the link l without contacting the controller. This alternate path is used to forward the packets that have already left the egress switch, before the controller can update the new path information at the egress switch. The solution must also minimize the amount of information stored in the switches.

IV. Path Restoration

In order to meet the requirements 1 and 2, that we set in the previous section, we have to store bypass path in the intermediate switches, in order to enable local corrective measures to be taken. The challenge is to use as little of the switches memory as possible and to ensure that these bypass paths are topology dependent and not path dependent as per requirement 3 and 4.

We propose three approaches to solve this problem. In first approach we store the bypass path information on all switches. The information stored is the bypass path to all its neighboring switches. In the second approach we store the bypass information on only the two hop distance switches. The information stored is the bypass path to only the two hop neighbor switches. In third approach we store the bypass path information only on the vertex cover nodes. The information stored is the bypass path to only neighboring vertex cover nodes.

A. Bypass Path to One Hop Neighbor

Consider the example topology shown in Figure 1. Here the bypass routes stored in switch a are:

1) $<a, c, d, b>$ to the adjacent node b if link ab fails
2) $<a, b, d, c>$ for the adjacent node c if link ac fails

Node a threfore stores two entries which is the same as its degree.

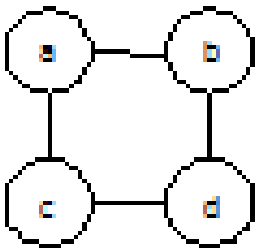

Fig. 1. Bypass route to one hop neighbor example

Every node $v_i \in V[G]$ will store the *one hop bypass path* $OB(i, k)$ for all $k \in ON(i)$.

Packet arriving at switch Consider a packet is at node v_i on its path from source node s to destination node d. Consider the edge v_i to v_k is down as a result of link failure. In such a case the bypass path $OB\,(i,k)$ stored at v_i is copied to the packet header, and may be used for source routing.

Figure 2 shows the header format. $v_1, \ldots v_k, v_l, \ldots$ represents the hop information along the primary path and m its length. $v_{k1}, v_{k2}, \ldots$ represents the hop information along the bypass path and l its length. p is the pointer to the next hop information along the primary path. sp is the pointer to the next hop information along the bypass path. m is decremented by every time the packet makes a hop along the primary path and a value of 0 indicates the packet has reached the destination. l is decremented by one every time the packet makes a hop along the bypass path and a value of 0 means the packet has taken the bypass and is now back on the primary path.

Fig. 2. Header format with bypass route

This will ensure the packet reaches node k via the bypass path. Thereafter it follows the original path embedded in the packet from node k to destination node d. The controller will eventually note that the link is down and identify an alternate path between s and d. For packets yet to be dispatched, this alternate path is used.

B. Bypass Path to Two Hop Neighbor

In order to reduce the total number of bypass path entries stored on the switches, we explore the idea of storing bypass path between only a few select nodes. This would imply that the packet may have to crank back along its path before it takes a bypass path. However, if the number of hops the packet would have to crank back is bounded then we maintain efficiency while we reduce the number of bypass paths stored.

Consider the example topology shown in Figure 3. Here we choose only the two hop neighbors to store the bypass path information between them. We represent them as black nodes a, c, e.The second shortest path between a and c is represented as $bp1$ and the second shortest path between c and e is represented as $bp2$. The bypass path information stored on node a:

1) $<a, bp1, c>$ to the two hop neighbor c if link ab or ac fails

The bypass path information stored on node c:

1) $<c, bp1, a>$ to the two hop neighbor a if link cb or ba fails
2) $<c, bp2, e>$ to the two hop neighbor e if link cd or de fails

The bypass path information stored on node e:

1) $<e, bp2, c>$ to the two hop neighbor c if link ed or dc fails

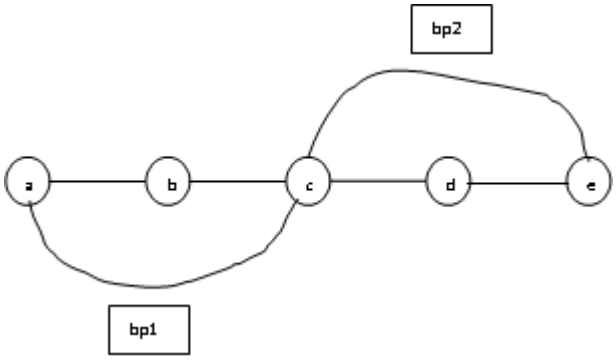

Fig. 3. Bypass route to two hop neighbor example

If the link coming after the black node along the path fails then the packets can directly take the bypass path to the two hop neighbor along the path. However if the link is not preceded by the black node along the path fails then the packet will have to crank back to the black node and then take the bypass path. For example in Figure 3 for a packet at node a along the path a to e if link ab fails, then the packet can take the bypass path $<a, bp, c>$ to node c. However, if the packet is at node b, and link bc fails, the packet would have to crank back to node a and then take the bypass path $<a, bp, c>$ to node c, since node b does not store any bypass paths.

Identifying the two hop neighbors in a graph, is essentially identifying if a graph is two colorable. If it is then we choose all the nodes with one color as the two hop neighbors. Identifying these two hop neighbors is topology dependent and not path dependent, and can be done proactively by the SDN controller.

The above approach involves carrying out the following steps proactively by the SDN controller

1) Identify if a graph is two colorable.
2) If step 1 is true, label the nodes of the graph with colors black and white.
3) Store the bypass path information only between the black nodes which are two hop neighbors.

V. Analysis

Source routing does not burden the switch by storing forwarding rules in it. However, as discussed in previous sections, in order to introduce resiliency we have to store the bypass path information locally in the switches. In this section we analyze the overhead due to this information store.

A. Bypass Path to One Hop Neighbor

Switch Memory

We store the *one hop bypass paths* on the all nodes for each of its neighbors. This memory requirement across the entire network $MOB\,(G)$ is given by

$$MOB\,(G) = \sum_{i=1}^{n} \sum_{k=1}^{d} |OB\,(i,k)| + 1 \tag{9}$$

where $|OB(i,k)|+1$ gives the bypass path length in terms of number of hops.
Since we consider a SDDCN the bypass path has a minimum length of 2 hops and a maximum lenght of 8 hops. Hence

$$2*n*d \leq MOB(G) \leq 8*n*d \tag{10}$$

Packet Header Space
Source routing introduces an overhead on each packet since it has the path embedded in it. This overhead is increased when fault tolerance is provided, as discussed in our proposed technique. However we show that this overhead is minimal. This header requirement on the packets $POB(s,d)$:

$$4 \leq POB(s,d) \leq 16 \tag{11}$$

since now the packet has to carry both the shortest path from s to d and the bypass path from i to k

Latency
The overall latency of source routing is reduced because of two reasons.

1) The controller does not have to install the forwarding rule on all the switches along the path, rather it passes the path information to the ingress switch so that it can replace the MAC header of packet. It later contacts the egress switch inorder to restore the MAC header.
2) The forwarding itself is faster, since the switch can forward packet based on just header content and it does not have to do a flow table lookup.

The latency for packet forwarding can be analyzed as follows: Given that m is the number of nodes in the path between a source s and destination d, p is the probability of one link failing along the path, l is the length of the bypass path across the broken link, t is the the time it takes to extract and act upon the next hop information from the packet and u is the time it takes to copy the bypass route to the packet header from the bypass table in the switch, then the latency $L(s,d)$ is given as follows

$$\begin{aligned} L(s,d) &= p*((m-1+l)*t+u)+(1-p)*m*t \\ &= p*((l-1)*t+u)+m*t \end{aligned} \tag{12}$$

B. Bypass Path to Two Hop Neighbor

Switch Memory
We store the *two hop bypass paths* on the black nodes for each of its neighbors. This memory requirement across the entire network can be analyzed as follows

If E_w and E_b give the number of two hop bypass path entries stored in the white and black nodes respectively then the total number of entries E in the network is given by

$$\begin{aligned} E &= E_w + E_b \\ &= 0 + E_b \end{aligned} \tag{13}$$

as we do not store any entries on the white nodes The number of black nodes in the graph G with n nodes is

$$N_b = v_1, v_2, \ldots, v_{\frac{n}{2}} \tag{14}$$

Similarly the number of white nodes is

$$N_w = v_1, v_2, \ldots, v_{\frac{n}{2}} \tag{15}$$

The number of two hop neighbor of a black node v_i, $TN(i)$ is dependent on the number of one hop neighbors or degree of v_i's white node neighbors.

$$TN(i) = ON(j)\, except \quad v_i \forall v_j \in ON(i) \tag{16}$$

$$t = \sum_{j=1}^{d} |ON(j)| - 1 \forall j \in ON(i) \tag{17}$$

where t is the number of entries in black node.

Equation 17 implies that the number of entries in a black node is directly proportional to the degree of its white neighbor nodes in $|ON(j)|$ or degree of j. This in turn implies that the number of bypass entries on a black node are also directly proportional to the degree of its white neighbor nodes.

The total number of entries in the network is given by

$$E = E_b = \sum_{k=1}^{\frac{n}{2}} t \forall k \in N_b \tag{18}$$

Hence the number of bypass entries will reduce if the nodes with lower degree are chosen as white nodes. However, this cannot be ensured with two color algorithms.

The packet header space and latency experienced by the packet is similar to that of one hop bypass path method discussed in previous section.

From the above observation we find that by choosing high degree nodes as black nodes, we reduce the number of entries in the switches. We therefore choose to identify minimum vertex cover nodes as the black nodes, because minimum vertex cover nodes are generally nodes of high degree.So instead of using $TN(i)$ in equation 16 we use $VN(i)$ as given in equation 5.

VI. Conclusion

In this paper we have proposed a solution to ensure resilient source routing in software defined wireless networks. Our solution aims to provide a backup path for packets in transit when a link failure occurs. Our aim was to enable the switches to take local corrective measures without having to consult the controller. Also, we needed to ensure our solution was scalable with respect to the switch memory space and the packet overhead. Our proposed approach caters to these requirements by storing a bypass path for the failed link at the switches. The number of such bypass paths is limited by the number of neighboring switches and the bypass path length itself is limited to 6 - 8 hops.

References

[1] *8th International Conference on Network and Service Management, CNSM 2012, Las Vegas, NV, USA, October 22-26, 2012*. IEEE, 2012.

[2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: minimal near-optimal datacenter transport. In *SIGCOMM*, pages 435–446, 2013.

[3] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[4] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 15:1–15:12, New York, NY, USA, 2010. ACM.

[5] Saumya Hegde, Shashidhar G. Koolagudi, and Swapan Bhattacharya. Scalable and fair forwarding of elephant and mice traffic in software defined networks. *Comput. Netw.*, 92(P2):330–340, December 2015.

[6] Ramon Marques Ramos, Magnos Martinello, and Christian Esteve Rothenberg. Data center fault-tolerant routing and forwarding: An approach based on encoded paths. *Dependable Computing, Latin-American Symposium on*, 0:104–113, 2013.

[7] Ramon Marques Ramos, Magnos Martinello, and Christian Esteve Rothenberg. Slickflow: Resilient source routing in data center networks unlocked by openflow. In *LCN*, pages 606–613, 2013.

[8] Mourad Soliman, Biswajit Nandy, Ioannis Lambadaris, and Peter Ashwood-Smith. Source routed forwarding with software defined control, considerations and implications. In *Proceedings of the 2012 ACM Conference on CoNEXT Student Workshop*, CoNEXT Student '12, pages 43–44, New York, NY, USA, 2012. ACM.

[9] Brent Stephens, Alan L. Cox, and Scott Rixner. Plinko: Building provably resilient forwarding tables. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 26:1–26:7, New York, NY, USA, 2013. ACM.