Achieving Cycle-Free Fast-Failover Paths in a Software Defined Network

Joel, Helkey
jHelkey@wsu.edu

Anand, Raghuraman
anand.raghuraman@wsu.edu

April 18,2016

1 Abstract

2 Introduction

The purpose of this research project is to design an algorithm that will output a set of cycle-free fast-failover paths in a Software Defined Network (SDN) for k link failures, where k¿= 1, given an arbitrary network topology and a primary path from source (s) to target (t). This set of failover paths will be used to program by the controller to program the flow tables in the SDN switches.

What is Software Defined Networking?

The goal of Software Defined Networking is to bring about the separation of the network control plane from the data plane. It enables centralization of control (in the so called controllers) and implements flow based control in the switches. Switches match incoming packets with flow entries in a flow table and forward the packets as instructed by the flow table entry. The controller uses protocols such as OpenFlow (the first communications protocol to address this issue) to configure network devices and choose the optimal network path for packet traffic. This architecture allows for a relatively small number of expensive controllers with a lot of cheap switches.

Path protection and restoration mechanism must be implemented to provide a reliable network. Whenever there is a link failure in the network, OpenFlow controller can react by computing a new backup path and provide the information to the affected node or the node can switch to backup path locally using the predefined backup path table provided by the OpenFlow controller. Setting predefined backup paths, results in a faster switching time compared to backup path that establish on demand. However, it may lead to the use of non-optimal backup path. In this paper, we present a fast and efficient failover mechanism for redirecting traffic flows to optimal path when there is a link failure or congestion problem. Our method uses the switch flow entry expiry feature to reroute traffic to backup path and additional help of buffer in OpenFlow switches to reduce switching delay and packet loss.

The motivation for this line of research is that while it is known that setting predefined backup paths results in faster network rerouting time compared to using the controller to provide a backup path that establishes on demand. However, a naive approach to programming the fast-failover paths could result in creating a cycle.

3 Problem Definition

We are given a Network. We have to transmit a package from a source to a destination via a optimal path which is already been calculated based on Dijikstra's algorithm. We have a case at which there is a link broken in the optimal path. We are creating a algorithm to find out all failover paths in the network so that there would be no difficulties in a particular information packet to go from the source to the destination.

What is Failover??

Failover is a backup operational mode in which the functions of a system component (such as a processor, server, network, or database, for example) are assumed by secondary system components when the primary component becomes unavailable through either failure or scheduled down time. Used to make systems more fault-tolerant, failover is typically an integral part of mission-critical systems that must be constantly available. The procedure involves automatically offloading tasks to a standby system component so that the procedure is as seamless as possible to the end user. Failover can apply to any aspect of a system: within an personal computer, for example, failover might be a mechanism to protect against a failed processor; within a network, failover can apply

to any network component or system of components, such as a connection path, storage device, or Web server.

4 Algorithms

There are two casses in this process:
1) There is only 1 link broken (i.e, K=1)
2) There are multiple links broken(i.e, K¿=1)
The following algorithm illustrates the solution for cycle-free fast-failover paths in a software defined network

Explaination of algorithm We use iterative Dijikstra's algorithm to get fast-failover paths in order to help a packet to reach the destination even if the optimal path which it travels has a link broken.What our algorithm does is that it creates a flow table having all the links in the primary path. Then it breakes each link one by one and then uses dijikstra's algorithm from the link's parent node to find another optimal path to the destination and copies all of the links of the path to the flow table. This serves as a 'back-up' path.In this process, if they come across a cycle, then the algorithm backtracks to the parent of the link and chooses the next optimal path. In this process, if it goes to the source node and it does not have another path to which it can go to, it sends an error message to the user asking him or her to add new links.
The following algorithm is for when there are multiple links broken in the network(i.e, K¿1)

Explaination of algorithm
In this algorithm, we have a set of non-disjoint paths.We have the path calculated by Dijikstra's algorithm as our primary path and have the other non-disjoint paths as our backup path. Here, if there exists a link that is broken in one path, we backtrack all the way back to the source node and go to the next non-disjoint path.
Our assumption here is that number of non-disjoint paths(nd) is greater than the number of broken links(k). i.e, nd $\geq K$.

Algorithm 1 $k = 1$

INPUT: AM, s, t, pp

OUTPUT: Set of flow tables with fast-failover entries

Initialize flow table with primary path information

for Link $\in$ *LinkQueue* **do**

T:=AM

In T, set value of link.from $\rightarrow$ link.to to zero

BackupPath := Dijiktra's algorithm on T with source as link.from

if BackupPath $\neq$ Emptyset **then**

for each bLink in BackupPath **do**

if Adding bLink does not create cycle **then**

add bLink failover information to flow table

else

loop:

if node $\neq$ Source **then**

Rerun algorithm with last link that caused cycle

else

Set error condition "Add more nodes/links"

end if

end if

end for

else

set Error condition "Add more nodes/links"

end if

end for

Algorithm 2 FindAllVertexIndependentPaths

INPUT: Network V, s, t

OUTPUT:Set of vertex independent paths

$n := 0$

for each vertex $\in V$ **do**

  if vertex $\neq s$ **then**

    Label vertex as "available"

  else

    Label vertex as "unavailable"

  end if

end for

do

  $m :=$ new path from s to t within available vertices

  if path exists **then**

    $n := n + 1$

    mark all vertices along the path with value $k = n$

  else

    break

  end if

while possible candidates for path from s to t

Return V

Algorithm 3 Case 2: $k \geq 1$

INPUT: AM, s, t, pp, k

OUTPUT: Set of flow tables with fast-failover entries

$T := AM$ minus nodes in pp and edges incident to nodes in pp

$VI :=$ FindAllVertexIndependentPaths(T)

$n := |VI| + 1$

if $n > k$ **then**

  VI:=VI+pp (making pp the first path in the set)

  for each $bPath$ in VI **do**

    for $link \in pp$ **do**

      if $link.from \neq s$ **then**

        add primary flow entry in direction of $bPath$

        add failover flow entry backwards along $bPath$

      else

        on start node, flow entry will switch to next

        vertex-independent path if it exists

      end if

    end for

  end for

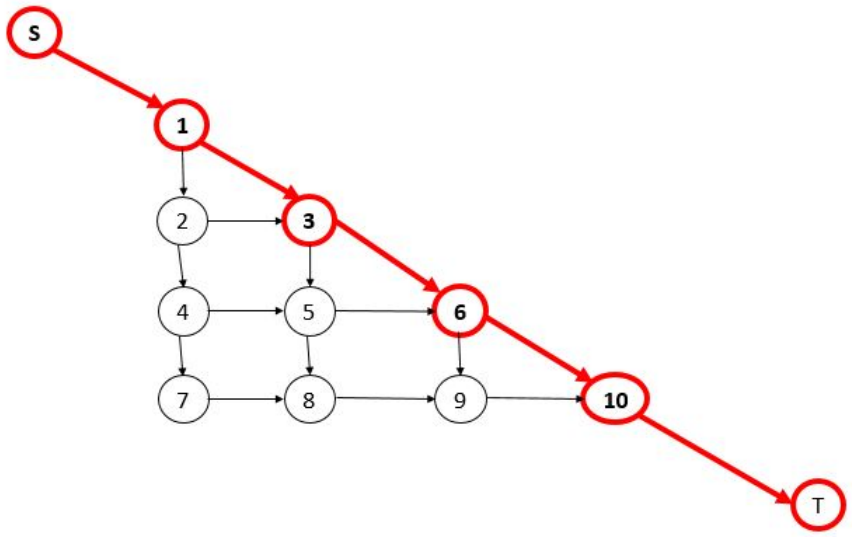
else

  Set error condition "Add more nodes/links"

end if

5 Implementation

We have implemented the algorithm for the first case(K=1) in the below network:

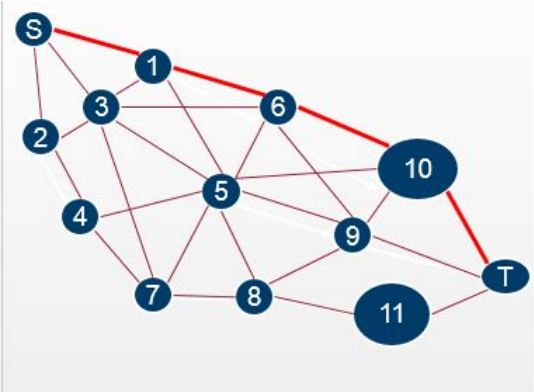


The final flow table is below:

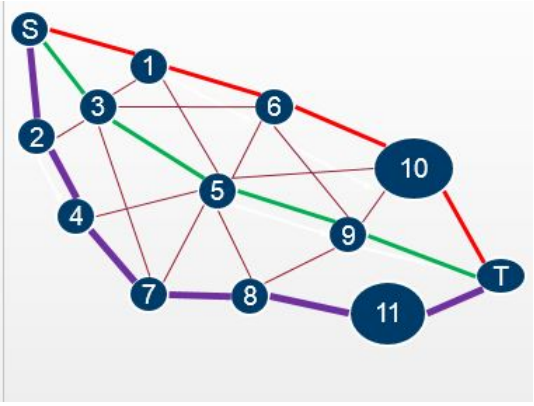
Final flow table:

Node	From	To
1	S	3
	S	2
3	1	6
	1	5
5	3	6
6	3	10
	3	9
9	6	10
10	6	T
	9	T

For the second case, K¿1, we have implemented the algorithm in the below network:



After implementing the algorithm, we get the different paths as:



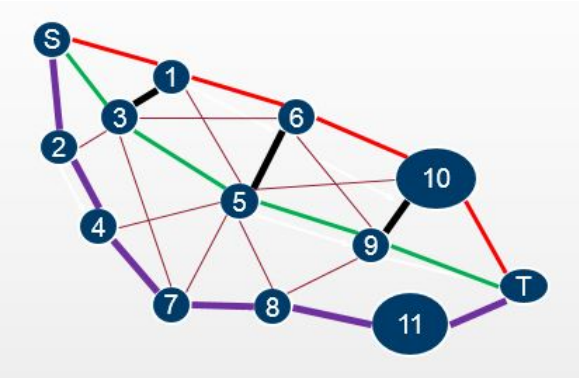
Here, Red is the primary path, violet in the tertiary path.

6 Results and Discussion

7 Future work

As a future work, we would optimize Algorithm 2. Our idea is to try and make each node of one vertex-dependent path to directly shift to a similarly placed node in another path rather than going back to the root node. We feel it would help in transmitting the information between source and target more efficient and fast.

Refer the below Picture:



Here, we have designed a network in such a way that when there is a failure of a link in the primary path, it is able to 'jump' to a node in another path(the link is in black) without going back to the root node.

8 Related work

There are two papers that were published realted to this research. One of the paper is titled "**DeadLock-Free Local Fast Failover for Arbitrary Data Center Networks**".With the data center networks sizes and bursty workloads we have today , it is likely that at any given moment there is a packet loss due to some type of that can be combined network failures, namely congestion and routing failures. Through this paper, we would get the first ever Deadlock free approaches to local fast failover. The papers evaluation shows that DF-EDST(Deadlock-Free-edge disjoint spanning trees) resilience can improve fault tolerance without adversely impacting performance when compared to a state-of-the-art approach to deadlock-free routing. This paper has introduced and evaluated two different approaches to implementing both deadlock-free and fault-tolerant forwarding for arbitrary data center networks, Df-FI resilience and DF-EDST resilience.The main contribution of the paper is their second approach which overcomes the scalability limitations of DF-FI resilience because it does not require the network to provide virtual channels.Their approach(called DF-EDST resilience) is such that packets transition between the paths defined by different EDSTs on network failures. This removes all the tree transitions that could cause cyclic dependencies from EDST resilience. Thus if the network provides even a limited number of virtual channels, DF-EDST resilience can exploit them to improve the performance. The paper then proves that DF-EDST resilience is deadlock-free and evaluate the DF-EDST resilience.

Another paper is titled "**Openflow Path Fast Failover Fast Convergence Mechanism**.The purpose of this research paper is to improve the OpenFlow controller fast failover mechanism for redirecting traffic flows to a backup path when there is a link failure in a Software Defined Network.The motivation for this line of research is that while it is known that setting predefined backup paths results in faster network rerouting time compared to backup path that establishes on demand. However, it may result in the use of a sub-optimal backup path. In this paper, the authors present a fast and efficient failover mechanism for redirecting traffic flows to a more optimal backup path when there is a link failure. They introduce a switch flow entry expiry mechanism to immediately reroute traffic to the backup path to reduce the network restoration time.The strengths of this paper are that it ignores fast failover groups which have been available since OpenFlow 1.1 (2011) and the resulting paths generated might not be loop-free in large network. The weaknesses are that it seems if main path is expired and evicted from flow table, what happens when link goes back up. There is no restoration of main path. Also, the modification to Dijkstras algorithm includes parameters that are unlikely to be known apriori; the length of queues, length of packet, and transmission rate.

9 Conculsion

10 Bibliography