Shortest path forwarding using OpenFlow

Joris Soeurt
Iwan Hoogendoorn

University of Amsterdam

February 13, 2012

Acknowledgement

We would like to thank Ronald van der Pol (SARA) for his guidance and supervision of this research project.

Abstract

The first part of this paper describes the problems the Spanning Tree Protocol causes in some complex, modern networks (limited load balancing, suboptimal path, long convergence time). It also describes the main properties and gives a comparison of two possible successor protocols (802.1aq en TRILL). Both protocols use the IS-IS routing protocol to forward layer 2 frames to their destination along the shortest path available. The second part consists of a in depth look at OpenFlow and an implementation of shortest path forwarding using this protocol ("NOX routing module"). Our main conclusion is that although a shortest path forwarding algorithm can certainly be implemented in OpenFlow, with the current stage of development it can't compete with 'native' TRILL or 802.1aq in production networks. Because OpenFlow brings the complete control plane of switches to a programmable server (the controller), one would think that with enough effort, every network protocol could be implemented. Theoretically even 802.1aq or TRILL could be implemented in OpenFlow. But exactly the same reason that provides this kind of flexibility to OpenFlow also limits the ability of such an implementation. Because the control plane is moved to a remote server, the switches can not take a link failover decision locally and therefore latency of failover is increased. Also notable is that the controller is a single point of failure. In version 1.2 (expected March 2012) two new features are introduced, namely failover group and a master slave controller feature. The first feature makes it possible for switches to take the decision of using an alternative forwarding path on link failure locally, without asking the controller. The latter eliminates the single point of failure.

During our tests, we found two shortcomings in the NOX routing module, for which we created a improvement proposal in pseudocode. The two shortcomings are the lack of load balancing and link failover (at all) when flow entries aren't timed out.

Contents

1 Introduction **4**
1.1 Introduction to research . . . 4
1.2 Research question . . . 4
1.3 Related work . . . 4
1.4 Approach . . . 4

2 Shortcomings of the Spanning Tree Protocol **6**
2.1 Introduction to Spanning Tree . . . 6
2.2 Shortcomings . . . 6

3 Life beyond spanning tree **7**
3.1 Introduction spanning tree enhancements/successors . . . 7
3.1.1 Possible spanning tree sucessor protocols . . . 8
3.1.2 TRILL and 802.1aq . . . 8

4 TRILL **8**
4.1 Introduction to TRILL . . . 8
4.2 How TRILL works . . . 9
4.2.1 Learning of MAC addresses . . . 9
4.2.2 Forwarding of unicast frames . . . 9
4.2.3 Multicast, broadcast and packets with unknown destination . . . 10
4.2.4 Designated RBridges . . . 11
4.2.5 Unique feature . . . 11
4.2.6 Load balancing . . . 11
4.2.7 Pros and cons . . . 11

5 802.1aq (Shortest Path Bridging) **11**
5.1 Introduction to Shortest Path Bridging . . . 11
5.2 How 802.1aq works . . . 12
5.2.1 Learning of MAC addresses . . . 12
5.2.2 Forwarding of unicast frames . . . 12
5.2.3 Forwarding of non unicast frames . . . 12
5.2.4 Packet behaviour throughout the network . . . 13
5.2.5 Unique features . . . 13
5.2.6 Load balancing . . . 13
5.2.7 Pros and cons . . . 13
5.3 Shortest path bridging or routing? . . . 14

6 Comparison of spanning tree, TRILL and 802.1aq **15**

7 Introduction to OpenFlow **17**
7.1 What is OpenFlow? . . . 17
7.2 How we are going to use OpenFlow . . . 17
7.3 OpenFlow operation in essence . . . 17
7.4 NOX OpenFlow controller . . . 17
7.5 Status of OpenFlow . . . 18
7.6 Current vendors supporting OpenFlow . . . 18

8 Shortest path forwarding in OpenFlow 18
8.1 Introduction to NOX modules . . . 18
8.2 Switch registration to controller . . . 19
8.2.1 Switch registration . . . 19
8.2.2 Keep alives . . . 19
8.2.3 Timeout . . . 19
8.3 Introduction . . . 20
8.4 Working of pyswitch module . . . 21
8.4.1 Sending multicast/broadcast/unknown frames (ARP request) . . . 22
8.4.2 Sending unicast packets . . . 23
8.5 Working of the routing module . . . 23
8.5.1 Discovering topology . . . 24
8.5.2 Spanning tree . . . 25
8.5.3 Sending multicast/broadcast/unknown frames (ARP request) . . . 26
8.5.4 Sending unicast frames (ARP response) . . . 27
8.5.5 Sending unicast packets . . . 27
8.6 Weaknesses in NOX routing module . . . 28

9 Comparison of NOX routing module with SPB (802.1aq) and TRILL 30

10 Conclusion 32

11 Further research 33

12 Appendix A: Command summary 36

13 Appendix B: Compilation of OpenWRT image with OpenFlow support 37
13.1 OpenWRT . . . 37
13.2 Add OpenFlow extension . . . 37

14 Appendix C: Description of test environment 39
14.1 Network Overview . . . 40

15 Appendix D: Bugfixes 42
15.1 OpenFlow dissector Wireshark plugin . . . 42
15.2 Routing module . . . 42

16 Appendix E: Summary of tests and results 43
16.1 Introduction . . . 43
16.2 Method . . . 43
16.2.1 Data collected . . . 43
16.2.2 Testing . . . 43
16.2.3 Resetting test . . . 44
16.3 Topologies . . . 44
16.4 Description of tests . . . 45
16.5 Result summary . . . 47

17 Appendix F: Pseudocode 49
17.1 Introduction . . . 49
17.2 Component overview . . . 49
17.3 Events . . . 50
17.4 Library calls . . . 50
17.5 Functions . . . 50

17.6 Tables 54

1 Introduction

1.1 Introduction to research

Although Spanning Tree has served well in the past, this protocol can't live up to expectations of some complex, modern environments. The most evident weaknesses lie in the lack of control of the active topology and topology predictiveness after a link failure. Also the lack of load balancing and inability to use a redundant infrastructure to its fullest (although somewhat possible when using PVST (Per VLAN Spanning Tree)) contribute to the demand for a successor in large layer two networks.

In this paper we evaluate in what way OpenFlow can be used to implement shortest path forwarding on a network as successor to the Spanning Tree Protocol.

1.2 Research question

In what way can shortest path forwarding be implemented in OpenFlow as alternative to the Spanning Tree Protocol?

1.3 Related work

Will TRILL replace Spanning Tree Protocol in data center networks? (Shamus McGillicuddy) In this article McGillicuddy describes the limitations of the Spanning Tree Protocol and briefly describes the likeliness of it being replaced by TRILL.

OpenFlow: Enabling Innovation in Campus Networks (Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, Jonathan Turner) This whitepaper proposes OpenFlow: a way for researchers to run experimental protocols in the networks they use every day[1].

TRILL and 802.1aq are like apples and oranges (Ivan Pepelnjak) A brief comparison of TRILL and 802.1aq[11].

NOX components / Network apps / Routing This module keeps track of shortest path routes between two authenticated data paths in the network. It discovers all active MAC addresses in the network and which switch port they are bound to. It then calculates a path to every destination MAC address and installs a flow to that destination on each switch along the path. When sending a packet to a yet unknown MAC address, the packet is flooded[16].

Basic Spanning Tree (NOX module) A basic spanning tree module is currently maintained by Glen Gibb (grg@stanford.edu). The module attempts to build a spanning tree within an OpenFlow network. It does not interact with standard spanning tree protocols such as STP, MSTP, RSTP, PVST or R-PVST[17].

1.4 Approach

During this research we took the following approach:

- Enumerate and describe why the Spanning Protocol can't adhere to the needs of some complex networks anymore and a successor is needed.

- Compare the main competing successor protocols (802.1aq and TRILL) to see how they handle the problems described in the previous step.
- Build an OpenFlow powered test environment and get familiar with the techniques and (in)abilities.
- Research the architecture and working of a shortest path forwarding implementation (*NOX routing module*.
- Create improvement proposal in pseudocode.
- Compare successors and draw conclusion.

2 Shortcomings of the Spanning Tree Protocol

2.1 Introduction to Spanning Tree

Although Spanning Tree has served well in the past, this protocol can't live up to expectations of some complex, modern environments. Radia Perlman probably didn't know what impact the protocol would have and for how many years it would be in use when she created it in 1985. Perlman created the following poem while working on the protocol.

Algorhyme

I think that I shall never see
a graph more lovely than a tree.

A tree whose crucial property
is loop-free connectivity.

A tree that must be sure to span
so packets can reach every LAN.

First, the root must be selected.
By ID, it is elected.

Least-cost paths from root are traced.
In the tree, these paths are placed.

A mesh is made by folks like me,
then bridges find a spanning tree.

Radia Perlman[21]

2.2 Shortcomings

The main shortcomings of the Spanning Tree Protocol can be summarized as follows:

Recalculation, interruption & convergence Every time the topology changes, all traffic is blocked while a recalculation of the spanning tree takes place. With the standard Spanning Tree Protocol and suboptimal conditions, this can take up to 50 seconds.

Inefficient use of resources The Spanning Tree Protocol works by blocking ports that give access to redundant paths to avoid network loops. Although the mechanism is perfect for avoiding loops, it also reduces the available bandwidth on other links.

Suboptimal path The path that traffic between two nodes travels is based on the spanning tree calculated for the entire network instead of the shortest path between those two nodes.

Example of suboptimal path & inefficient use of resources

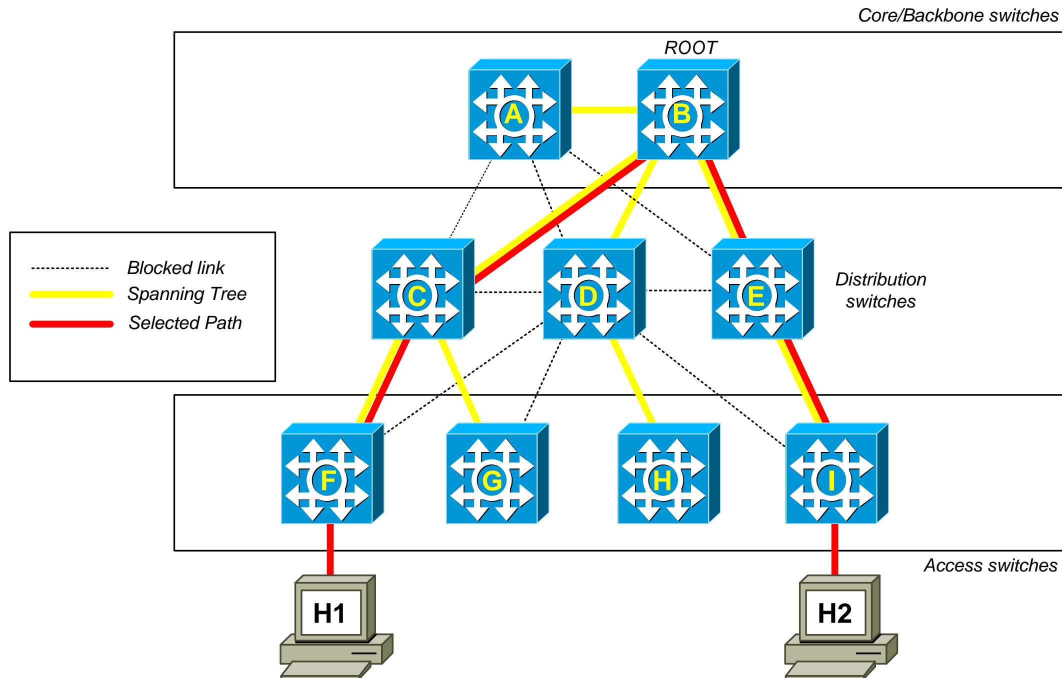


Figure 1: Spanning tree example

All traffic has to forwarded along the precalculated tree. Instead of forwarding traffic along the shortest path (H1 →F →D→I) traffic is forwarded over the tree. (H1 →F →C →B →E →I) No load balancing is possible because all redundant paths are blocked.

3 Life beyond spanning tree

3.1 Introduction spanning tree enhancements/successors

Several enhancements have been made to the protocol to deal with these issues. Some of them made it to standards, others are proprietary. For example Cisco's **Per-VLAN Spanning Tree** to calculate a spanning tree per VLAN (and therefore load balance the physical network) and extensions like PortFast, BackboneFast and UplinkFast to reduce the time needed to converge on certain topology changing events. **IEEE 802.1w RSTP** (Rapid Spanning Tree) to reduce convergence time and **IEEE 802.1s MST** (Multiple Spanning Tree) as extension to rapid spanning tree to scale better, linking multiple VLANs to a spanning tree.

Although these enhancements have helped the Spanning Tree Protocol survive till today, they also made the

protocol more complex to configure and troubleshoot and less interoperable between vendors. One could argue that these enhancements have held back the innovation needed. Fortunately several protocols are being developed to push to real innovation.

3.1.1 Possible spanning tree sucessor protocols

There are several initiatives in the industry that can be used to solve The Spanning Tree shortcomings. Below a few of them are listed, but the only ones in scope for this research will be TRILL and 802.1aq.

- OTV (Overlay Transport Virtualization)[22].
- Portland[23].
- EVPN (Ethernet Virtual Private Network) / VPLS (Virtual Private LAN Service)[24].
- PBB-EVPN (Provider Backbone-Ethernet Virtual Private Network)[25].
- VL2[26].
- Seattle[27].
- Fabric Path[28].
- Moose (Multi-level Origin-Organised Scalable Ethernet)[29].
- 802.1Qbp[30].
- TRILL (Transparent Interconnection of Lots of Links).
- 802.1aq.

3.1.2 TRILL and 802.1aq

The IEEE is in the process of standardising the **802.1aq** protocol while the IETF is working on the **TRILL** specification.

In both protocols, bridges exchange link state information using a slightly adjusted version of the IS-IS routing protocol to achieve a global consistent view of all bridges, topology and location of end nodes. The bridges use this information to efficiently forward layer 2 frames to their destination using the shortest path available. This combination of using a routing protocol to forward frames on a layer 2 network, helps keep the ease of configuration of a layer 2 segment while adding the advantages of layer 3. These advantages can be summarized as multipathing (load balancing), shortest path forwarding and almost instant link failover. Although both protocols have the same main design goals and similar solutions, both differ on several points which are elaborated on in the paragraphs below.

4 TRILL

4.1 Introduction to TRILL

With the introduction of a successor of spanning tree, also a successor to Algorhyme was introduced by Radia's son. Algorhyme v2 is included in the IETF draft and can be read below.

Algorhyme v2

I hope that we shall one day see
A graph more lovely than a tree.

A graph to boost efficiency
While still configuration-free.

A network where RBridges can
Route packets to their target LAN.

The paths they find, to our elation,
Are least cost paths to destination!

With packet hop counts we now see,
The network need not be loop-free!

RBridges work transparently,
Without a common spanning tree.

Ray Perlner[7]

4.2 How TRILL works

4.2.1 Learning of MAC addresses

TRILL can use 4 different techniques to learn where a specific node is located (in other words where the MAC address of that node is located).

- Observation of source MAC address of locally received frames as in regular bridging.
- Observation of frames received from other RBridges (TRILL capable bridges) by examining the header and learning the combination of source RBridge and source MAC.
- Static configuration.
- The ESADI (End System Address Distribution Information) protocol. Using this protocol, RBridges can exchange end station addresses and associated RBridge with each other.

4.2.2 Forwarding of unicast frames

Forwarding of frames of which the location of the destination MAC address is known, are forwarded over the shortest path using the topology information gained by IS-IS. This happens much like forwarding layer 3 packets using normal routers.

When forwarding a frame to an end node:

1. The ingress RBridge adds two headers in order to deliver the frame to the egress RBridge to which the end node is connected to.
    - An inner TRILL header (containing ingress and egress RBridge, hop count and multi-destination flag).

- An outer ethernet header (with as source the MAC address of the sending/transmitting RBridge and as destination the MAC of the next hop RBridge).

2. The next hop RBridge is derived from the link state database acquired through IS-IS.
3. The frame is then forwarded to this next hop RBridge.
4. The next hop uses the information in the TRILL header (egress RBridge) and the IS-IS link state database to make a forwarding decision for yet another next hop.
5. The RBridge changes the outer ethernet header source address to it's own address and destination address to the chosen next hop. It also decrements the hop count by 1 and recalculates the FCS. It doesn't alter the TRILL header.
6. This process continues all the way till the frame is received by the egress RBridge, which strips both the outer ethernet header and TRILL header, leaving the original packet, which is then delivered to the destination host.
7. Because of this outer (regular) ethernet header, the frames appears to intermediate bridges just as a regular frame, which can be forwarded by looking at the destination MAC address.

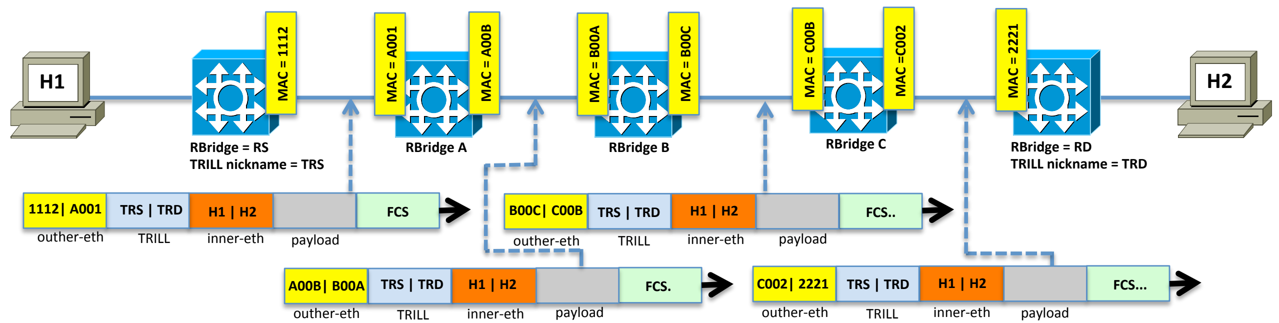


Figure 2: TRILL forwarding

4.2.3 Multicast, broadcast and packets with unknown destination

All packets which cannot be routed to one specific known MAC address should still be flooded through the network. To achieve this without creating loops, distribution trees are calculated. Multiple trees are calculated to achieve multipathing. The different trees are rooted at different RBridges. When an RBridge needs to flood a packet, it is first forwarded to the nearest distribution tree root (an RBridge) and then subsequently flooded through the network. These trees are individually calculated by all RBridges on basis of several parameters, priority and tiebreakers in a deterministic way, such that every RBridge calculates the same trees.

4.2.4 Designated RBridges

If multiple RBridges are present on a segment, a Designated RBridge (appointed forwarder) is selected. Frames on that specific segment are forwarded to their destination only by this RBridge. Without this mechanism, two RBridges on a segment could lead to a frame being delivered twice.

4.2.5 Unique feature

Addition of a hop count which can avoid loops in exception situations where somehow an error in the control plane occurs.

4.2.6 Load balancing

Redundant paths used using multiple spanning trees and equal cost load balancing.

4.2.7 Pros and cons

Pros

- Because the protocol has been designed without keeping current standards and formats too much in mind, the designers have been able to keep it fairly simple to understand.
- Intermediate bridges can be regular bridges and still forward frames between Rbridges in a regular way because the frames are encapsulated on the outside with a normal ethernet header.

Cons

- Because of the extra encapsulation, new ASICs have to be designed to forward the frames in hardware.
- Because unicast and non-unicast frames are sent using a different technique, they might get delivered out of order, when the MAC state transitions from unknown to known.
- TRILL forwarding is done a hop-by-hop basis. Because there is no simple way to determine the selected path for a particular flow, troubleshooting must be done hopby-hop at each node. [10]

5 802.1aq (Shortest Path Bridging)

5.1 Introduction to Shortest Path Bridging

The standard created by the IEEE is called 802.1aq. Two different versions are defined, SPB-V and SPB-M. Both versions use a different encapsulation and have different properties, but share the same main design principles.

Shortest Path Backbone Bridging (SPB-M) is aimed to be deployed in PBB (Provider Backbone) networks where all addresses are managed.

Shortest Path Bridging (SPB-V) is applicable in customer, enterprise or storage area networks.

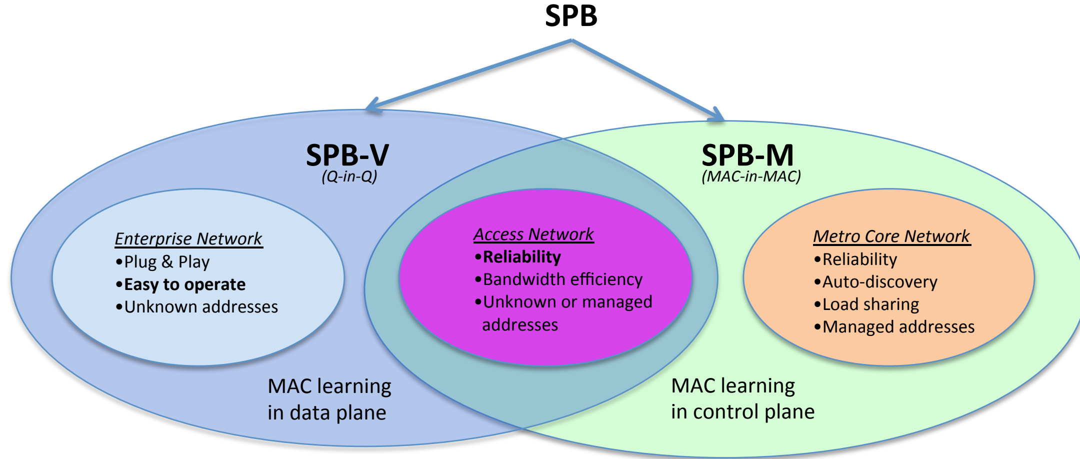


Figure 3: SPB-V and SPB-M charasteristics.

A huge benefit of SPB-M is that it protects all of the switches in the infrastructure (edge and core) from being seen by any end hosts due to the way of encapsulation. This was an important feature of the protocol design for many different environments, especially the campus and datacenter environments due to the fact that from the providers point of view they don't have all the MAC addresses stored in their switches.
SPB-M is aimed to be deployed in METRO and WAN (Wide Area Networks).

5.2 How 802.1aq works

5.2.1 Learning of MAC addresses

In SPB-M, MAC addresses are learned via the control plane; they are distributed between the bridges using an extension of IS-IS.
In SPV-V MAC addresses are learned via the data plane; by looking at the contents of incoming frames.

5.2.2 Forwarding of unicast frames

In comparison to TRILL. 802.1aq uses a different approach compared to TRILL for forwarding frames. After IS-IS has determined the topology, all edge bridges calculate at least one spanning tree to reach every destination. Meaning, the source bridge can be the root of the tree for every frame it sends and the complete end to end path is calculated and known before any frame has been forwarded. The calculation of these trees is deterministic in such a way that all intermediate bridges have identical views of each other's trees. Therefore, a frame only needs to be marked with the source and the destination bridge, and all intermediate bridges can forward the frame along the intended path without making hop by hop routing decisions. This path is calculated based on the information received by the IS-IS process. These trees are created in such a way that traffic between two bridges always takes the same path in both directions (traffic between two hosts is always symmetrical)

5.2.3 Forwarding of non unicast frames

Non unicast frames are forwarded over the exact same tree as unicast frames are forwarded over. This means the path of non-unicast traffic is congruent to unicast traffic. The difference is that unicast frames are only forwarded over a subset of the tree (the shortest path from source to destination) and non-unicast frames are forwarded over the complete tree to all leaf nodes.

5.2.4 Packet behaviour throughout the network

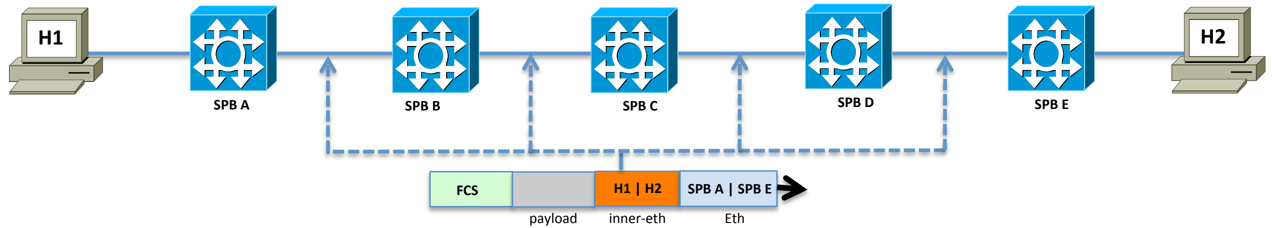


Figure 4: 802.1aq forwarding

1. The ingress SPB device adds two headers in order to deliver the frame to the SPB device to which the end node is connected to.
2. Based on the SPB type (V or M) the frame is encapsulated in a specific way.
3. One header contains the MAC addresses of the source and destination hosts and the other header contains the MAC addresses of the source SPB device and the destination of the SPB device.
4. The frame will remain unchanged until its delivered to the destination host. The complete path is known in advance and not determined hop-by-hop.

5.2.5 Unique features

- Use of I-SID identifiers to logically separate different ”services” in a physical network. These services can be seen like VLANs, but without the max of 4k VLAN restriction.
- The 802.1aq standard is based as much as possible on existing standards (for example 802.1ah, 802.1Q, 802.1ad).

5.2.6 Load balancing

The redundant path selection is determined based on various ECMT algorithms[31].

5.2.7 Pros and cons

Pros

- Existing ASICs can be used for forwarding in hardware.
- Complete path of frames is deterministic, known in advance and can be calculated offline. This makes debugging less hard.
- Unicast and non-unicast traffic is send over the same tree following a congruent path.

Cons

- Protocol is complex because compatibility with existing technology is maintained.

5.3 Shortest path bridging or routing?

Although both methods seem like routing on layer 2, using the term layer 2 routing is debatable. This is because both protocols have very specific layer 2 and specific layer 3 elements.

Layer 2 elements

- Unmodified layer 2 frames are delivered to end stations.
- No IP addresses have to be configured on the bridges to be able to communicate with each other.
- MAC addresses are (partly) learned from the data-plane.

Layer 3 elements

- TRILL uses a hop by hop lookup and forward mechanism for frames, just like routers use their routing table.
- The IS-IS routing protocol is used instead of the Spanning Tree Protocol to forward frames.
- MAC addresses can be learned by communication with other switches, using the control plane.

6 Comparison of spanning tree, TRILL and 802.1aq

<table>
<tr><th>Spanning tree (802.1d)</th><th>SPB (802.1aq)</th><th>TRILL</th></tr>
<tr><td colspan="3">Organization</td></tr>
<tr><td colspan="2">IEEE</td><td>IETF</td></tr>
<tr><td colspan="3">Load balancing</td></tr>
<tr><td>Redundant paths blocked</td><td>Redundant path selection determined based on various ECMT algorithms[31]</td><td>Redundant paths used using multiple spanning trees and equal cost load balancing.</td></tr>
<tr><td colspan="3">Calculation of shortest path for unicast frames with known destination</td></tr>
<tr><td>Uses spanning tree algorithm to calculate path with the use of BPDUs (Bridge Protocol Data Units).</td><td colspan="2">Use IS-IS protocol to calculate shortest path.</td></tr>
<tr><td colspan="3">Hardware</td></tr>
<tr><td colspan="2">Current ASICs can be used</td><td>New ASICs required in order to rewrite MAC addresses and change hop count (TTL). Intermediate bridges can handle these frame as there is no need to change hop count or MAC addresses.</td></tr>
<tr><td colspan="3">Extra loop prevention checks [1]</td></tr>
<tr><td>No extra mechanism to prevent loops; redundant ports are blocked</td><td>RPFC (Reverse Path Forwarding Check) to prevent loops</td><td>RPFC (Reverse Path Forwarding Check) & TTL (hop) field to prevent loops</td></tr>
<tr><td colspan="3">Forwarding of unicast packets</td></tr>
<tr><td>Forwarded over the tree using local switch MAC tables.</td><td>"After IS-IS builds the network topology, SPB creates the shortest paths based on link metrics and then assigns the traffic (Unicast and Multicast) to that path. Traffic with same source and destination combination is forwarded through path that is calculated in advance. Therefore it is very easy to predict the traffic flows through the meshed network since they are calculated once for the entire path. Forward and reverse path symmetric."[10]</td><td>Unicast traffic is forward along the most optimal path from source to destination RBridge. The forwarding decisions are made hop by hop, locally on each router. Forward and reverse paths are not symmetric.</td></tr>
</table>

[1] These checks are built-in to prevent loops in in certain exceptional situations, such as an error in the control plane.

Spanning tree (802.1d)	**SPB (802.1aq)**	**TRILL**
Forwarding of packets other than unicast		
Broadcasted over the (single) tree	Broadcasted, multiple trees possible. Paths of unicast and non-unicast traffic congruent.	Broadcasted, multiple trees possible.
Encapsulation		
No extra frame encapsulation	SPB-M - MAC-in-MAC (802.1ah) / SPB-V - Q-in-Q (802.1ad)	TRILL header & extra ethernet header

7 Introduction to OpenFlow

7.1 What is OpenFlow?

OpenFlow is a recent technology developed at Stanford University. This protocol strictly divides the control and data plane of all network equipment and moves the control plane to a centralised controller. Forwarding decisions are no longer made by network equipment (data-paths) itself, but by a server (controller) which subsequently passes these forwarding decisions on to the data-path (by means of the OpenFlow API) as a OpenFlow flow. This controller isn't bound to any network hardware limitations and vendor's closed software platforms. Because of this and because the controller communicates with all OpenFlow enabled devices (and therefore is aware of the network topology), this architecture supplies a very powerful means to program the data-flows in a network the most optimal way.

7.2 How we are going to use OpenFlow

Because all switches register to the OpenFlow controller, the controller can create a complete overview of all switches and links between the switches. Because of this overview the controller has, it can calculate the shortest path between nodes and instruct the switches to create flows that adhere to this shortest path. The major difference in this approach compared to TRILL and 802.1aq is that instead of letting all switches work together to decide the optimal path, all switches communicate to the controller where all the calculations are being done and actions are sent to the switches.

7.3 OpenFlow operation in essence

As the name suggests, OpenFlow is all about flows. OpenFlow switches cannot forward any frame without a flow entry and switches cannot create flows entries on their own. For the creation of these flow entries, a controller is needed. Whenever an OpenFlow switch receives a frame, it encapsulates this frame/packet into an OpenFlow packet and sends this to the controller. The controller responds to the switch with a flow entry. This switch installs this flow entry locally and uses it to forward frames that adhere to the criteria of this flow entry.

Such a flow entry can be divided in two parts; a match part and an action. The main properties of the match part are:

- Source and destination MAC address
- Source and destination IP4 address
- Protocol (apply action only if frame/packet is of this protocol)
- In port (apply action only if frame/packet is received on this interface)

The main actions are to output the frame to a specific interface or flood to all interfaces.

7.4 NOX OpenFlow controller

Our research is done using the NOX OpenFlow controller. This choice is based on the fact that this controller is fully open source and the modules (which dictate the behaviour of the controller) can be programmed in Python.

7.5 Status of OpenFlow

OpenFlow is currently used for testing and research purposes. This due to the fact that OpenFlow is immature, emerging, and still trying to sort out what direction its heading in[2].

Current production implementations are running on 1.0 and not on 1.1 for the simple reason 1.1 is a no-starter for vendors due to several issues that are solved in version 1.2. The specs that where introduced by several speakers where categorised as simplistic, but it could still be used for testing interesting things with it's limited functionality. Different features and bug-fixes are being introduced and integrated in upcoming releases 1.3, 1.4 and so on...[2]

7.6 Current vendors supporting OpenFlow

Several large, well known vendors have released OpenFlow hardware. The list of vendors consists of, but is not limited to:

- Broadcom
- NetFPGA
- Pronto
- HP
- NEC
- IBM

8 Shortest path forwarding in OpenFlow

8.1 Introduction to NOX modules

To research how a network can be programmed using OpenFlow as successor to Spanning Tree, we started by researching successor protocols (TRILL and 802.1aq). In this chapter we want to explain the architecture and working of a shortest path forwarding algorithm provided by the NOX OpenFlow controller (*NOX routing module*). Note that the name of the module is somewhat deceiving in the sense that it doesn't do layer 3 routing, but shortest path forwarding for layer 2 frames and layer 3 packets.

Unfortunately the documentation is very limited[16] and outdated (26-02-2010) and the source code (C) is not annoted. Therefore, we took another approach to understand this module. We set up different tests in our test-environment and treated the routing module as a black box. By monitoring every aspect (`tcpdump` on the hosts, Wireshark with OpenFlow dissector plugin on our controller, current flows on the switches and console output of the hosts and NOX controller) we both thoroughly tested the operation and gathered the information needed to understand the inner working.

To make the operation of the routing module more clear, we start by explaining the registration of the switches. Next we explain the working of the pyswitch module, which behaves just like a regular learning switch. Then we explain the behaviour of the routing module and end with a comparison, highlighting the main characteristics. We end with a complete overview of the inner-working and features of the routing module.

The tests we executed and the results can be found in appendix E.

8.2 Switch registration to controller

This part is independent of the NOX module currently active. It is repeated by every switch in order to set up a communication channel with the NOX controller. **SM** is a symmetric message while **CSM** is non-symmetric message (Control switch communication)

8.2.1 Switch registration

Switches		**Controller (NOX)**
Hello (SM)	→	
	←	Hello (SM)

Using the Hello packets, the switch identifies the controller of its existence and both agree on version number of the OpenFlow protocol they are going the speak.

	←	Features Request (CSM)

The controller sends a features request to see which ports are available.

	←	Set Config (CSM)

In this case, the controller asks the switch to use specific parameters.

Features Reply (CSM)	→	

The switch replies with its physical ports, port capabilites, supported actions, buffer sizes.

	←	Vendor (CSM)
Error (Vendor request not understood) (CSM)	→	

The controller sends the Vendor command to the switch, this command supported by our OpenFlow test switches.

	←	Flow Mod (delete all current flows) (CSM)

The controller asks the switch to delete all current flows to start with a clean slate.

8.2.2 Keep alives

Switches		**Controller (NOX)**
	←	Echo request (SM)
Echo reply (SM)	→	
Echo request (SM)	→	
	←	Echo reply (SM)

Both switch as controller send each other a Hello request with a random transaction ID, and reply to each others request, repeating the transaction ID of the request. This process is repeated every 15 seconds.

8.2.3 Timeout

For completeness, the following table is included. Every flow in an OpenFlow device has a timeout value. If no traffic has been matched against a rule before the specified idle-timeout value, the OpenFlow device removes the flow and notifies the controller.

Flow removal, controller notification

Switches		Controller (NOX)
Flow removed	→	

The switch notifies to the controller which flow it has removed after the timeout has expired.

8.3 Introduction

The working of the pyswitch and the routing module is explained by showing how the data and control frames/packets are sent between each host/device. This is done by sending an ARP request/response and ICMP echo request/response between H1 and H2.

Note that while we use this simple 2 switch topology for explaining the protocol communication, our tests were performed using a redudant 6 switch topology.

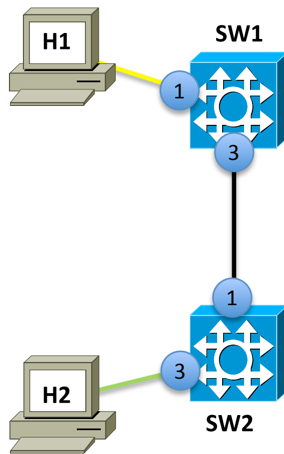


Figure 5: Topology used for explaing the NOX modules.

The tables below can be read in the following way:

Data	Ctrl	Src.	Dest.	Content
Sequence number of data frame/packet, sent between end-hosts. If marked with a,b,c.. this means it is the same frame/packet, but in a different stage of forwarding.	Sequence number control packets, sent between OpenFlow switch and controller.	Source of frame/packet	Destination of frame/packet	Description of content of frame/packet.

8.4 Working of pyswitch module

Description The pyswitch module behaves just like a normal learning switch. It learns the MAC address to port binding by examine incoming packets and looking at the source address. It uses this information to forward incoming frames. If the destination is unknown, the packet is flooded. It is not possible to have redundant links because this would result in a loop.

Traffic flows The source host sends frames/packets to destination host. The switch asks the NOX controller how to handle these frames. This is decided by the pyswitch module.

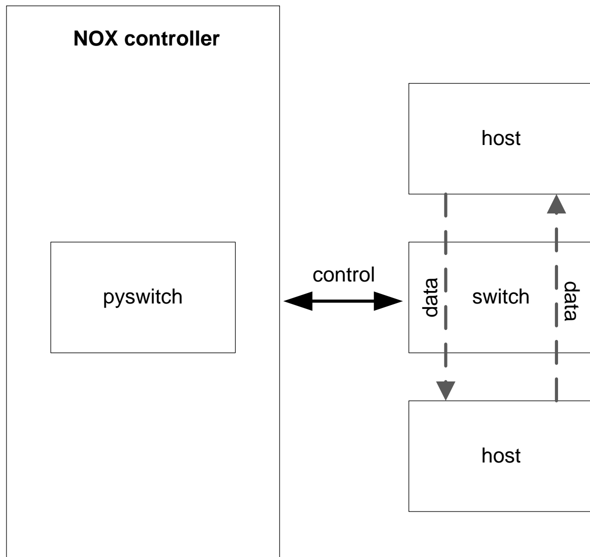


Figure 6: Component interaction of pyswitch module

8.4.1 Sending multicast/broadcast/unknown frames (ARP request)

ARP request

Data	Ctrl	Src.	Dest.	Content
1a		Host 1	Broadcast	ARP: Who has 10.10.10.2 ?

Host 1 sends out ARP request to the broadcast MAC address.

	1	Switch 1	Controller	OFP encoded frame 1 (ARP request)

The switch has no flows yet and does not know how to forward this frame. It encapsulates the frame in a OpenFlow packet and sends it to the controller.

	2	Controller	Switch 1	Flood frame out all ports (except originating)

The controller orders the switch to send the frame out of all ports (except for the orginating)

1b		Switch 1	Flood	ARP: Who has 10.10.10.2 ?

The switch indeed floods the frame. When switch 2 receives the frame, this process repeats the same way:

	3	Switch 2	Controller	OFP encoded frame 1 (ARP request)
	4	Controller	Switch 2	Flood frame out all ports (except originating)

As switch 2 floods the frame, it is received by host 2.

1c		Switch 2	Flood	ARP: Who has 10.10.10.2 ?

ARP response

2a		Host 2	Host 1	ARP: I'm 10.10.10.2, my MAC is 52:54:00:D5:4F:0D

Host 2 (now aware of the MAC address of host 1) replies back.

	5	Switch 2	Controller	OFP encoded frame 2 (ARP response)

The switch does not know what to do with the frame and asks the controller using an OpenFlow control packet.

	6	Controller	Switch 2	Create flow: Prot: ARP, SRC 52:54:00:34:FA:2C, DST 52:54:00:D5:4F:0D →IN = 3, OUT = 1

The controller knows where the destination MAC address exists on the network and orders the switch to create a flow. This flow can be used by the switch if any subsequent frames with the same properties should arrive.

2b		Switch 2	Switch 1	ARP: I'm 10.10.10.2, my MAC is 52:54:00:D5:4F:0D

Switch 2 forwards the frame to switch 1.

	7	Switch 1	Controller	OFP encoded frame 2 (ARP response)

The process repeats; the controller knows where the destination MAC address exists on the network and orders the switch to create a flow. This flow can be used by the switch if any subsequent frames with the same properties should arrive.

	8	Controller	Switch 2	Create flow: ARP reply from 52:54:00:D5:4F:0D to 52:54:00:34:FA:2C →inport = 3 outport = 1

Finally the frame is deliverd to host 1.

2c		Switch 1	Host 1	ARP: I'm 10.10.10.2, my MAC is 52:54:00:D5:4F:0D

8.4.2 Sending unicast packets

ICMP echo request

Data	Ctrl	Src.	Dest.	Content
1a		Host 1	Host 2	ICMP echo request

Now that host 1 knows the MAC address of host 2, it can send a ICMP echo request.

	1	Switch 1	Controller	OFP encoded packet 1 (ICMP echo request)

Again, the switch doesn't know what to do with the packet, so it encodes it in a OpenFlow packet and asks the controller.

	2	Controller	Switch 1	Create flow: Prot: ICMP, SRC 10.10.10.1, DST 10.10.10.2 →IN = 3, OUT = 1

During the ARP session, the controller has learned the location of the destination host and can now instruct the switch to create a flow for this specific traffic.

1b		Switch 1	Switch 2	ICMP echo request

Switch 1 forwards the frame to switch 2

	3	Switch 2	Controller	OFP encoded packet 1 (ICMP echo request)

Switch 2 also ask the controller what to do with the incoming packet.

	4	Controller	Switch 2	Create flow: Prot: ICMP, SRC 10.10.10.1, DST 10.10.10.2 →IN = 3, OUT = 1
1c		Switch 2	Host 2	ICMP echo request

Again, the controller instructs the switch to create a flow for traffic forwarding.

ICMP echo response

The sending of the ICMP echo response happens in the same way as the REQUEST packet is forwarded:

2a		Host 2	Host 1	ICMP echo reply
	5	Switch 2	Controller	OFP encoded packet 2 (ICMP echo reply)
	6	Controller	Switch 2	Create flow: Prot: ICMP, SRC 10.10.10.2, DST 10.10.10.1 →IN = 1, OUT = 3
2b		Switch 2	Switch 1	ICMP echo reply
	7	Switch 1	Controller	OFP encoded packet 2 (ICMP echo reply)
	8	Controller	Switch 1	Create flow: Prot: ICMP, SRC 10.10.10.2, DST 10.10.10.1 →IN = 1, OUT = 3
2c		Switch 1	Host 1	ICMP echo reply

8.5 Working of the routing module

Description The routing module provides shortest path forwarding within the OpenFlow network. The module depends on several other modules, of which the most important ones are the discovery module and the spanning tree module. The spanning tree module is used to flood traffic with an unknown or unspecified destination throughout the network. The routing module is used to forward traffic along the shortest path

between source and destination. This can only be done for unicast traffic of which the location of the destination host is known.

Traffic flows When the hosts send traffic to each other, the switch asks the NOX controller how to handle this traffic. The discovery module discovers the topology and gives input to the routing module (for known destinations traffic) and spanning tree module (for unknown destination traffic).

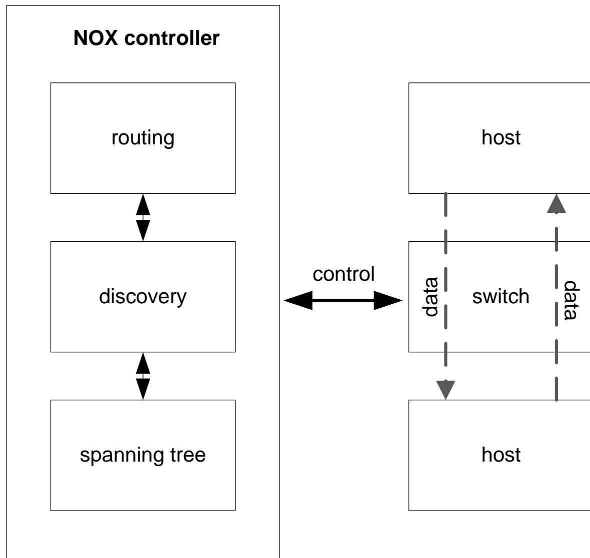


Figure 7: Component interaction in routing module

8.5.1 Discovering topology

Discovery of the topology is needed for the spanning tree module to be able to calculate a spanning tree of the network. The module works by instructing every registered switch to send LLDP frames out of every interface. These frames originate at the controller and are transported to the switch using the OpenFlow protocol. Subsequently, when an outgoing frame of an arbitrary switch has been received by another arbitrary switch, that frame is routed back to the controller, using the OpenFlow protocol. By extracting the frame from the OpenFlow packet, and examining its content, the controller knows which two switches are connected and with what interfaces. It can therefore build up a complete topology overview.

LLDP (iterated every 10 seconds				
Data	**Control**	**Source**	**Destination**	**Content**
	1	Controller	Switch 1	Send LLDP frame out port 1
1		Switch 1	Port 1	LLDP frame
	2	Controller	Switch 1	Send LLDP frame out port 2
2		Switch 1	Port 2	LLDP frame
	3	Controller	Switch 1	Send LLDP frame out port 3
3		Switch 1	Port 3	LLDP frame
	4	Switch 2	Controller	**Received** this LLDP frame at port 1
	5	Controller	Switch 1	Send LLDP frame out port 4
4		Switch 1	Port 4	LLDP frame

Of the 4 LLDP frames switch 1 has send out, one has reached switch 2. Switch 2 subsequently send the frame back to the controller indicating the interface it has received the frame on (port 1).

Data	Control	Source	Destination	Content
	6	Controller	Switch 2	Send LLDP frame out port 1
5		Switch 2	Port 1	LLDP frame
	7	Switch 1	Controller	**Received** this LLDP frame at port 3
	8	Controller	Switch 2	Send LLDP frame out port 2
6		Switch 2	Port 2	LLDP frame
	9	Controller	Switch 2	Send LLDP frame out port 3
7		Switch 2	Port 3	LLDP frame
	10	Controller	Switch 2	Send LLDP frame out port 4
8		Switch 2	Port 4	LLDP frame

Of the 4 LLDP frames switch 2 has send out, one has reached switch 1. Switch 1 subsequently send the frame back to the controller indicating the interface it has received the frame on (port 3). The controller now knows switch 1 and 2 are connected by a single link between port 1 and 3.

8.5.2 Spanning tree

Now that the topology is known, a the spanning tree can be calculated. The spanning tree is not (only) kept in memory, but programmed to the active switches. This is done by using portmod commands, indicating which ports should and which ports shouldn't output a packet defined with "output action" flood. This means that even if a switch received the command to flood a packet/frame, the packet/frame is only sent out on specific interfaces, thereby eliminating loops.

Flood deactivation

Switches		Controller (NOX)
	←	Port Mod (port 1)
	←	Port Mod (port 2)
	←	Port Mod (port 3)
	←	Port Mod (port 4)

On activation, the controller immediately sends a portmod command to every switch for every interface the switch has. In the portmod command, the "Do not include this port when flooding" -flag is set to "active"

Partial flood reactivation to build spanning tree

	←	Port Mod (port X)
	←	Port Mod (port Y)

After the "spanning tree module" has received the topology information from the "topology module", it reactivates some ports for flooding by de-activating the "active" flag using the portmod command.

8.5.3 Sending multicast/broadcast/unknown frames (ARP request)

The mechanism of sending packets with an unspecific destination is illustrated by the process of an ARP request.

ARP request

Data	Ctrl	Source	Destination	Content
1a		Host 1	Broadcast	ARP: Who has 10.10.10.2 ?

Host 1 broadcast a normal ARP request.

	1	Switch 1	Controller	OFP encoded frame 1 (ARP request)

Switch 1 wraps the frame into an OpenFlow packet, sends it to the controller and asks what to do with it.

	2	Controller	Switch 1	Create flow: Prot: ARP, SRC 52:54:00:34:FA:2C, DST FF:FF:FF:FF:FF:FF →IN = 3, OUT = Flood
	3	Controller	Switch 1	Flood frame out all ports (except originating)
1b		Switch 1	Flood	ARP: Who has 10.10.10.2 ?

The controller responds by instructing the switch to add a flow for this specific traffic. It then gives the command to flood this specific frame out of all ports. (Except the ones with the no flooding flag set by the spanning tree module). This process is now repeated identically for switch 2, until the frame reaches it's destination (host 2)

	4	Switch 2	Controller	OFP encoded frame 1 (ARP request)
	5	Controller	Switch 2	Create flow: Prot: ARP, SRC 52:54:00:34:FA:2C, DST FF:FF:FF:FF:FF:FF →IN = 3, OUT = Flood
	6	Controller	Switch 2	Flood frame out all ports (except originating)
1c		Switch 2	Flood	ARP: Who has 10.10.10.2 ?

8.5.4 Sending unicast frames (ARP response)

Now that the controller knows where both sender and receiver are located on the network and the replying host knows the MAC address of the receiver, the ARP response can be forwarded in a much more efficient way. For unicast frames with known destination, the complete end to end path is calculated and subsequently programmed to all switches along the path.

ARP response

2a		Host 2	Host 1	ARP: I'm 10.10.10.2, my MAC is 52:54:00:D5:4F:0D

Host 2 sends out the ARP response

	7	Switch 2	Controller	OFP encoded frame 2 (ARP response)

Switch 2 asks the controller what to do with the frame.

	8	Controller	Switch 2	Create flow: Prot: ARP, SRC 52:54:00:D5:4F:0D, 10.10.10.2, DST 52:54:00:34:FA:2C, 10.10.10.1 →IN = 3, OUT = 1
	9	Controller	Switch 2	Create flow: Prot: ARP, SRC 52:54:00:D5:4F:0D, 10.10.10.2, DST 52:54:00:34:FA:2C, 10.10.10.1 →IN = 3, OUT = 1

The controller creates a flow on both switches simulteanously.

		Controller	Switch 2	Perform action in flow table for frame 2

The controller tells switch 2 what to do with this specific frame.

2b		Switch 2	Switch 1	ARP: I'm 10.10.10.2, my MAC is 52:54:00:D5:4F:0D
2c		Switch 1	Host 1	ARP: I'm 10.10.10.2, my MAC is 52:54:00:D5:4F:0D

The frame can be delivered to host 1 with the earlier created flow.

8.5.5 Sending unicast packets

To explain the forwarding of unicast packets, we chose to use ICMP echo because of it's triviality and widespread familiarity. We chose not to annote the entries in this table because all individual steps have been annoted in previous tables and this improves the readability.

ICMP echo

Data	Ctrl	Source	Destination	Content
1		Host 1	Host 2	ICMP echo request
	1	Switch 1	Controller	OFP encoded packet 1 (ICMP echo request)
	2	Controller	Switch 1	Create flow: Prot: ICMP, SRC 10.10.10.1, DST 10.10.10.2 →IN = 3, OUT = 1
	3	Controller	Switch 2	Create flow: Prot: ICMP, SRC 10.10.10.1, DST 10.10.10.2 →IN = 3, OUT = 1
	4	Controller	Switch 1	Perform action in flow table for packet 1
2		Switch 1	Switch 2	ICMP echo request
3		Switch 2	Host 2	ICMP echo request
4		Host 2	Host 1	ICMP echo reply
	5	Switch 2	Controller	OFP encoded packet 1 (ICMP echo reply)
	6	Controller	Switch 2	Create flow: Prot: ICMP, SRC 10.10.10.1, DST 10.10.10.2 →IN = 3, OUT = 1
	7	Controller	Switch 1	Create flow: Prot: ICMP, SRC 10.10.10.1, DST 10.10.10.2 →IN = 3, OUT = 1
	8	Controller	Switch 2	Perform action in flow table for packet 4
5		Switch 2	Switch 1	ICMP echo reply
6		Switch 1	Host 1	ICMP echo reply

8.6 Weaknesses in NOX routing module

By testing the routing module, we found the following shortcomings:

No immediate link failover During the testing we found out that when a change in topology occurs, for example a link that is removed, all active flows using that link are not rerouted. The flow is not removed from the data-path and as long as packets keep coming in, the flow idle-timeout is reset and not removed. Only when the packet stream is stopped, the flow times out. When the packet stream is started again, the route through the network is reprogrammed along another path. This can probably be solved relatively easy using OpenFlow protocol version 1.1. Unfortunately, we only found this out at the end of our research. Also, version 1.1 has not been implemented for our test hardware. In version 1.1 the concept of failover groups is introduced in which the switch can take a failover decision to another interface without having to wait for the controller.

We created a implementation proposal that solves this issue in the current version by keeping a local cache of all installed flows. When a topology change is detected, all effected flows are remove from the active switches. Because after removal no flow is available for subsequent frames, a recalcuation is triggered yielding a new active path. This proposal can be found in Appendix F.

No extra loop prevention The module contains a bug that makes hosts authenticate to different data-paths. This authentication is a registration of which hosts can be reached through which data-path. In exceptional cases, the sender of a broadcast packet that is flooded through the network can get associated with multiple switches. This means that all these switches think they are the best data-path to reach that host. This results in loops. A mechanism such as a TTL field which is used in TRILL could prevent the packets from going into an endless loop when there is a problem in the control plane such as this one. However, modifying or adding headers is not possible using the OpenFlow protocol. [20]

Controller is single point of failure When the controller goes down the network can no longer be controlled, in the OpenFlow versions 1.0 & 1.1 the controller high availability feature is not present. This feature is planned however in OpenFlow version 1.2[3].

No load sharing During the test we found that no load sharing is used, all traffic follows one specific shortest path. This is both the case when traffic is flooded (because only one tree is used) as with traffic that is forwarded along the shortest path from source to destination.

9 Comparison of NOX routing module with SPB (802.1aq) and TRILL

NOX routing module	**SPB (802.1aq)**	**TRILL**
Organization		
No specific organization pushing the implementation, because it is not a standard. OpenFlow is maintained by the Open Networking Foundation.	IEEE	IETF
Load balancing		
None	Redundant path selection determined based on various ECMT algorithms[31]	Redundant paths used using multiple spanning trees and equal cost load balancing.
Calculation of shortest path for unicast frames with known destination		
Based on hop count, with weight of each hop being 1. The algorithm is based on the paper "A New Approach to Dynamic All Pairs Shortest Paths" by Demetrescu et. al. [18][19]	Use IS-IS protocol to calculate shortest path.	
Hardware		
New hardware is needed. All intermediate switches between source and destination need to have OpenFlow support.	Current ASICs can be used	New ASICs required because of new frame format. Intermediate bridges can handle these frame as if normal ethernet frames (unaware of extra TRILL header)
Extra loop prevention checks [2]		
By calculating flood ports using spanning tree and programming into network. No extra built-in securities to prevent loops in "exception" cases.	RPFC to prevent loops	RPFC & TTL field to prevent loops
Forwarding of unicast packets		
Frames and packets of which the controller has learned the location of destination node are forwarded over the shortest path. This path is programmed in advance to all switches along the path.	"After IS-IS builds the network topology, SPB creates the shortest paths based on link metrics and then assigns the traffic (Unicast and Multicast) to that path. Traffic with same source and destination combination is forwarded through path that is calculated in advance. Therefore it is very easy to predict the traffic flows through the meshed network since they are calculated once for the entire path. Forward and reverse path symmetric."[10]	Unicast traffic is forward along the most optimal path from source to destination RBridge. The forwarding decisions are made hop by hop, locally on each router. Forward and reverse paths are not symmetric.

[2] These checks are built-in to prevent loops in in certain exceptional situations, such as an error in the control plane.

Shortest path forwarding using OpenFlow	**SPB (802.1aq)**	**TRILL**
Forwarding of packets other than unicast		
Frames of which the destination is unknown are flooded to all OpenFlow switches using the precomputed & programmed spanning tree.	Broadcasted, multiple trees possible. Paths of unicast and non-unicast traffic congruent.	Broadcasted, multiple trees possible.
Encapsulation		
No extra encapsulation for data frames. Frames sent to controller encapsulated with OpenFlow header.	SPB-M - MAC-in-MAC (802.1ah) / SPB-V - Q-in-Q (802.1ad)	TRILL header & extra ethernet header

10 Conclusion

This paper has explained both the problems the Spanning Tree Protocol faces as well as different successor protocols to solve these problems. During our research we tested a shortest path forwarding algorithm supplied with the NOX OpenFlow controller (*NOX routing module*). This has given us a good understanding of the OpenFlow architecture and how a shortest path forwarding algorithm can be implemented leveraging this protocol.

All three methods (SPB; 802.1aq, TRILL and the NOX routing module) are designed around their own design principles and therefore have their own advantages and disadvantages for deployment in specific situations. SPB (802.1aq) is strongly designed around reusing existing standards and specifications to make implementation more smoothly. TRILL on the other hand has been designed more pragmatic and has focussed on a solution which keeps things simple.

OpenFlow is not a protocol for shortest path forwarding per se but a very powerful protocol to program networks in virtually any way one would want to. Decisions are not made distributed (in agreement of all switches to each other), but centralized, with a (single) controller instructing all switches what to do. OpenFlow is designed as a means to implement and test new protocols. It is therefore also relatively easy to implement a shortest path forwarding algorithm such as the *NOX routing module* described in this paper. Because OpenFlow brings the complete control plane of switches to a programmable server (the controller), with enough effort, every network protocol could be implemented. Theoretically even 802.1aq or TRILL could be implemented in OpenFlow. But, in the current state we don't think such an implementation can compete with 'native' SPB (802.1aq) and TRILL. OpenFlow is still in development and while version 1.1 is the current stable version, work is being done on version 1.2 with lots of added and changed features. Moreover, because of the centralized architecture, overhead and latency will always be worse than with native, specialized, one purpose protocols.

One of these new features, failover group, is needed to implement a sub second path failover on link failure. With this feature an alternative path can be programmed to the switch in advance. When the switch notices a link failure, it can forward frames using an alternative path without waiting for the controller. The other protocols don't have this issue because of their use of the IS-IS routing protocol and ability to calculate alternative paths locally. Another feature introduced in version 1.2 is the ability to register a switch to both a master as slave controller to eliminate the single point of failure of the controller.

For now we must conclude that although OpenFlow is very promising and most certainly will be deployed more and more the coming years, in the current state it doesn't provide an Spanning Tree alternative such as 'native' TRILL and 802.1aq. The two main reasons for this being the heavy development still going on and the same reason that gives this protocol its flexiblity; the control plane being located on a remote server.

11 Further research

Further research can be divided into two main parts:

Leveraging new features Implementation of the group failover and master slave features.

Optimization of algorithm The routing module lacks support for load balancing. Adding support to this module would be a great improvement.

References

[1] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, Jonathan Turner, *OpenFlow: Enabling Innovation in Campus Networks*
`http://www.openflow.org/documents/openflow-wp-latest.pdf`
14 March 2008

[2] Ethan Banks, *OpenFlow State of the Union: Reflections on the OpenFlow Symposium*
`http://packetpushers.net/OpenFlow-state-of-the-union-reflections-on-the-OpenFlow-symposium`
27 October 2011

[3] Jean Tourrilhes *OpenFlow 1.2 proposals*
`http://www.OpenFlow.org/wk/index.php/OpenFlow_1_2_proposal`

[4] Yiannis Yiakoumis, Julius Schulz-Zander, Jiang Zhu *OpenFlow for OpenWRT*
`http://www.OpenFlow.org/wk/index.php/Pantou_:_OpenFlow_1.0_for_OpenWRT`

[5] Interworking Task Group of IEEE 802.1, *IEEE P802.1aq Draft Standard for Local and Metropolitan Area Networks - Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks*
13 December 2011

[6] J. Touch, R. Perlman, *Transparent Interconnection of Lots of Links (TRILL): Problem and Applicability Statement*
`http://tools.ietf.org/html/rfc5556`
May 2009

[7] D. Eastlake 3rd, D. Dutt, S. Gai, A. Ghanwani, *Routing Bridges (RBridges): Base Protocol Specification*
`http://tools.ietf.org/html/rfc6325`
July 2011

[8] A. Banerjee, D. Dutt, R. Perlman, A. Ghanwani, *Transparent Interconnection of Lots of Links (TRILL) Use of IS-IS*
`http://tools.ietf.org/html/rfc6326`
July 2011

[9] R. Perlman, A. Ghanwani, D. Dutt, V. Manral, *Routing Bridges (RBridges): Adjacency*
`http://tools.ietf.org/html/rfc6327`
July 2011

[10] Avaya, *Compare and Contrast SPB and TRILL*
`http://www.avaya.com/uk/resource/assets/whitepapers/SPB-TRILL_Compare_Contrast-DN4634.pdf` 2011

[11] Ivan Pepelnjak, *TRILL and 802.1aq are like apples and oranges.*
`http://blog.ioshints.info/2010/08/trill-and-8021aq-are-like-apples-and.html`
02 August 2010

[12] John Herbert, *Layer 2 Routing (sort of) and TRILL*
`http://lamejournal.com/2011/05/16/layer-2-routing-sort-of-and-trill`
16 May 2011

[13] Greg Ferro, *Network Fabric:TRILL for Server and Network People. Welcome RBridges* `http://etherealmind.com/trill-introduction-review-overview-why-what-how` 30 March 2009

[14] Jose Miguel Huertas, *TRILL: The end of Spanning-Tree?*
http://blog.initialdraft.com/archives/1412
9 March 2011

[15] NANOG 50, *The Great Debate: TRILL Versus 802.1aq (SPB)*
http://www.nanog.org/meetings/nanog50/presentations/Monday/NANOG50.Talk63.NANOG50_TRILL-SPB-Debate-Roisman.pdf
4 October 2010

[16] *NOX routing module main Wiki page*
http://noxrepo.org/noxwiki/index.php/Routing

[17] Glen Gibb, *NOX Spanning Tree module*
http://www.openflow.org/wk/index.php/Basic_Spanning_Tree

[18] Camil Demetrescu, Giuseppe F. Italiano,*A New Approach to Dynamic All Pairs Shortest Paths*
http://www.dis.uniroma1.it/~demetres/docs/dapsp-full.pdf 2003

[19] NOX repo, *How does the routing module calculate shortest path?*
http://noxrepo.org/pipermail/nox-dev/2010-December/007039.html

[20] NOX repo, *One MAC might get authenticated to more than one datapath*
http://noxrepo.org/pipermail/nox-dev/2010-March/006063.html

[21] Radia Perlman, *Algorhyme*
http://www.csua.berkeley.edu/~ranga/humor/algorhyme.txt

[22] H. Grover, D. Rao, D. Farinacci, V. Moreno, emphOTV (Overlay Transport Virtualization)
http://tools.ietf.org/id/draft-hasmit-otv-03.txt
08 Juli 2011

[23] Portland
http://cseweb.ucsd.edu/~vahdat/papers/portland-sigcomm09.pdf

[24] EVPN (Ethernet Virtual Private Network) / VPLS (Virtual Private LAN Service)
http://tools.ietf.org/html/rfc4761/http://tools.ietf.org/html/rfc4762

[25] PBB-EVPN (Provider Backbone-Ethernet Virtual Private Network
http://tools.ietf.org/html/draft-sajassi-l2vpn-pbb-evpn-00

[26] VL2
http://research.microsoft.com/apps/pubs/default.aspx?id=80693

[27] Seattle
http://www.cs.princeton.edu/~chkim/Research/SEATTLE/seattle.pdf

[28] Fabric Path
http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9402/white_paper_c11-605488.html

[29] Moose (Multi-level Origin-Organised Scalable Ethernet)
http://www.cl.cam.ac.uk/~mas90/MOOSE

[30] 802.1Qbp
http://www.ieee802.org/1/pages/802.1bp.html

[31] Equal Cost Multi Tree
http://en.wikipedia.org/wiki/IEEE_802.1aq#Equal_Cost_Multi_Tree_-_ECMT

12 Appendix A: Command summary

dpctl	
dpctl dump-flows tcp:*localhost*	Display current flows.
dpctl dump-desc tcp:*localhost*	Display vendor + version information.
dpctl show tcp:*localhost*	Display switch port state & capabilities.
dpctl show-protostat tcp:*localhost*	Display traffic statistics.
dpctl add-flow tcp:*localhost* in_port=*in_port*,actions=*action*	Create forwarding flow on switch.
dpctl del-flows tcp:*localhost*	Remove all current flows from switch.

tcpdump	
tcpdump -ennqti any \(arp or icmp \)	Display summary of all received ICMP and ARP packets.

NOX controller	
sudo ./nox_core -v -v -i ptcp:6633 routing	Start NOX controller with routing module in verbose mode. Listen on port TCP 6633

13 Appendix B: Compilation of OpenWRT image with OpenFlow support

The OpenWRT image available on the OpenFlow wiki[4] did not function correctly on our hardware (TP-Link WR1043ND v1.8). Although the image booted correctly and didn't output any errors, the creation of flows did not lead to the forwarding of traffic. The problem is most probably related to the hardware version of our equipment. The wiki states that version 1.7 had been tested, while we had received switches of hardware version 1.8. We solved this problem by compiling our own OpenWRT image from the latest SVN / GIT branches.

This appendix is based on the instructions available on: `http://www.OpenFlow.org/wk/index.php/OpenFlow_1.0_for_OpenWRT` Some steps, are removed, others are made more clear. Several errors have been corrected.

13.1 OpenWRT

Create a working directory for the compilation process.

```
mkdir ~/openwrt
```

Install all dependencies needed for the compilation process:

```
apt-get quilt git install build-essential binutils flex bison autoconf gettext texinfo
sharutils subversion libncurses5-dev ncurses-term zlib1g-dev gawk
```

Create a working directory, checkout latest SVN source for OpenWRT and and prepare source code.

```
cd ~/openwrt
svn co svn://svn.openwrt.org/openwrt/branches/backfire
cd ~/openwrt/backfire
./scripts/feeds update -a
./scripts/feeds install -a
```

Select platform specific parameters:

Select "Atheros AR71xx/AR7240/AR913x" under "Target System".
Select "TP-LINK TL-WR1043ND v1" under "Target profile"
Save and exit

```
make menuconfig
```

Check if you have all prerequisites installed with:

```
make prereq
```

13.2 Add OpenFlow extension

Go to your working directory and download the OpenFlow extension.

```
cd ~/openwrt/
git clone git://gitosis.stanford.edu/OpenFlow-openwrt
```

Move to the TP-Link specific branch.

```
cd ~/OpenFlow_openwrt
git checkout -b OpenFlow-1.0/tplink origin/OpenFlow-1.0/tplink
```

Add the OpenFlow extensions to the backfire directory.

```
cd ~/openwrt/backfire/package/
ln -s ~/openwrt/OpenFlow-openwrt/OpenFlow-1.0/
```

Add basic configuration files for OpenWRT

```
cd ~/openwrt/backfire/
ln -s ~/openwrt/OpenFlow-openwrt/OpenFlow-1.0/files/
```

Add the related package to your configuration:

Select "OpenFlow" under "Network".
Select "tc" under "Network".
Select "kmod-tun" under "Kernel Modules, Network Support".
Save and Exit

```
make menuconfig
```

Add support for queueing:

Select "Hierarchical Token Bucket (HTB)" under "Networking Support, Networking options, QoS and/or fair queueing"
Save and Exit

```
make kernel_menuconfig
```

Build the image

```
make
```

Load the image (~/openwrt/backfire/bin/ar71xx/openwrt-ar71xx-tl-wr1043nd-v1-squashfs-*[sysupgrade][factory]*.bin) to the device. For instructions on how to load the image, see the OpenWRT wiki (`http://wiki.openwrt.org/toh/tp-link/tl-wr1043nd`

14 Appendix C: Description of test environment

Summary of hard- and software used in test environment.

Servers

Hostname	**IP address**	**Description**	**Hardware**	**Software**	**Version**
C1	145.100.37.185/27	NOX controller	N/A	Ubuntu	10.04
				NOX	0.9.1
H1	145.100.37.186/27 10.10.10.1/24	Test host 1	Virtual	Ubuntu	10.04
H2	145.100.37.187/27 10.10.10.2/24	Test host 2	Virtual	Ubuntu	10.04
H3	145.100.37.188/27 10.10.10.3/24	Test host 3	Virtual	Ubuntu	10.04
H4	145.100.37.189/27 10.10.10.4/24	Test host 4	Virtual	Ubuntu	10.04

OpenFlow switches

Hostname	**IP address**	**Descrip.**	**Hardware**	**Software**	**Version**
SW1	145.100.37.163/27	Switch 1	TP-LINK TL-WR1043ND	OpenWRT	Backfire, 10.03.1, r29685
				OpenFlow	v1.0 latest (13-05-11)
SW2	145.100.37.164/27	Switch 2	TP-LINK TL-WR1043ND	Ubuntu	10.04
				OpenFlow	v1.0 latest (13-05-11)
SW3	145.100.37.165/27	Switch 3	TP-LINK TL-WR1043ND	Ubuntu	10.04
				OpenFlow	v1.0 latest (13-05-11)
SW4	145.100.37.166/27	Switch 4	TP-LINK TL-WR1043ND	Ubuntu	10.04
				OpenFlow	v1.0 latest (13-05-11)
SW5	145.100.37.167/27	Switch 5	TP-LINK TL-WR1043ND	Ubuntu	10.04
				OpenFlow	v1.0 latest (13-05-11)
SW6	145.100.37.168/27	Switch 6	TP-LINK TL-WR1043ND	Ubuntu	10.04
				OpenFlow	v1.0 latest (13-05-11)

Management switch

Hostname	**IP address**	**Description**	**Hardware**	**Software**	**Version**
VLAN-SW	145.100.37.162/27	Management switch	TP-LINK TL-WR1043ND	Cisco IOS	12.1(22)EA14

Choice for TP-Link TL-WR1043ND equipment[4]

We choose to use TP-Link TL-WR1043ND switches because of their low price and the ability to run custom firmware. Because of these properties, work has been done to implement a OpenWRT image with OpenFlow support. Although performance is low, it is a very attractive choice to build a testing network with.

14.1 Network Overview

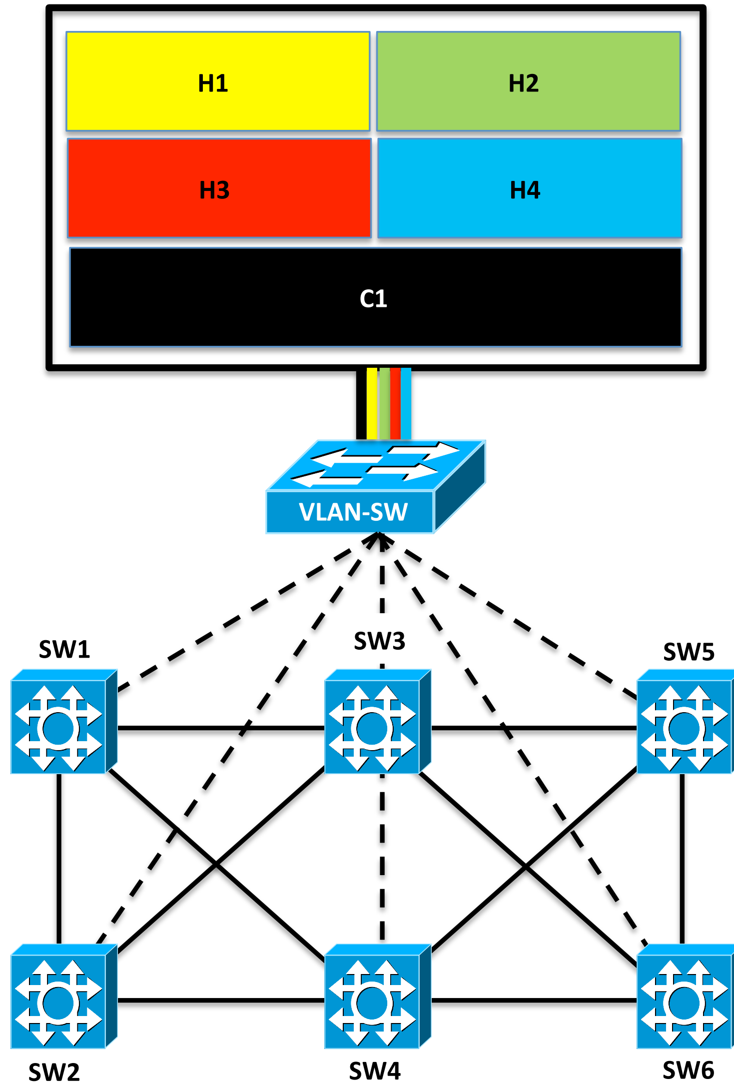


Figure 8: Logical network diagram

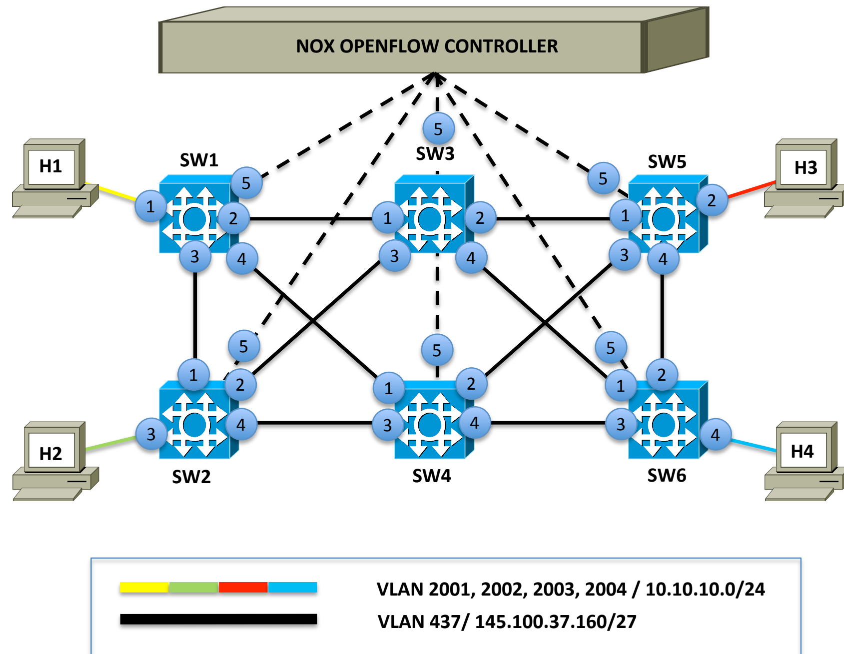


Figure 9: Physical network diagram

15 Appendix D: Bugfixes

15.1 OpenFlow dissector Wireshark plugin

To use the OpenFlow dissector Wireshark plugin (version 1.0) with Wireshark (version 1.6.4) we had to change line 769 of `packet-OpenFlow.c` from:

```
dissector_add(TCP_PORT_FILTER, global_OpenFlow_proto, OpenFlow_handle);
```

to:

```
dissector_add_uint(TCP_PORT_FILTER, global_OpenFlow_proto, OpenFlow_handle);
```

Based on `http://www.mail-archive.com/OpenFlow-discuss@lists.stanford.edu/msg00969.html`

15.2 Routing module

The dependency of the routing module on the spanning tree module is not correctly configured in the corresponding `meta.json` file. Without fixing this problem, loops are created and the module doesn't function at all.

Change:

```
{
    "name": "routing" ,
    "library": "sprouting" ,
    "dependencies": [
        "routing_module",
        "authenticator",
    ]
},
```

To:

```
{
    "name": "routing" ,
    "library": "sprouting" ,
    "dependencies": [
        "routing_module",
        "authenticator",
        "spanning_tree"
    ]
},
```

16 Appendix E: Summary of tests and results

16.1 Introduction

During the research we performed a number of tests. The aim of these tests was to understand the OpenFlow protocol and bridges, working of and interaction with the NOX controller and innerworking of the routing module.

16.2 Method

16.2.1 Data collected

For the tests, we collected the following data (where applicable).

- tcpdump log of ICMP and ARP packets/frames on hosts.
- Wireshark logging on NOX controller (with help of OpenFlow dissector plugin.
- Console output of NOX controller.
- Current flows on OpenFlow switches.

  To capture the flow table on different switches in specific point in time, we created a script. This script uses a private-public key construction to automatically login to all 6 OpenFlow switches at the same time, dump the current flows, and save them to a non-volatile location.

16.2.2 Testing

The steps we iterated for each test (where applicable):

Test 1a & 1b:

- Start tcpdump on hosts.
- Create manual flows on switch.
- Start data stream from source to destination host.

Other tests (with NOX controller):

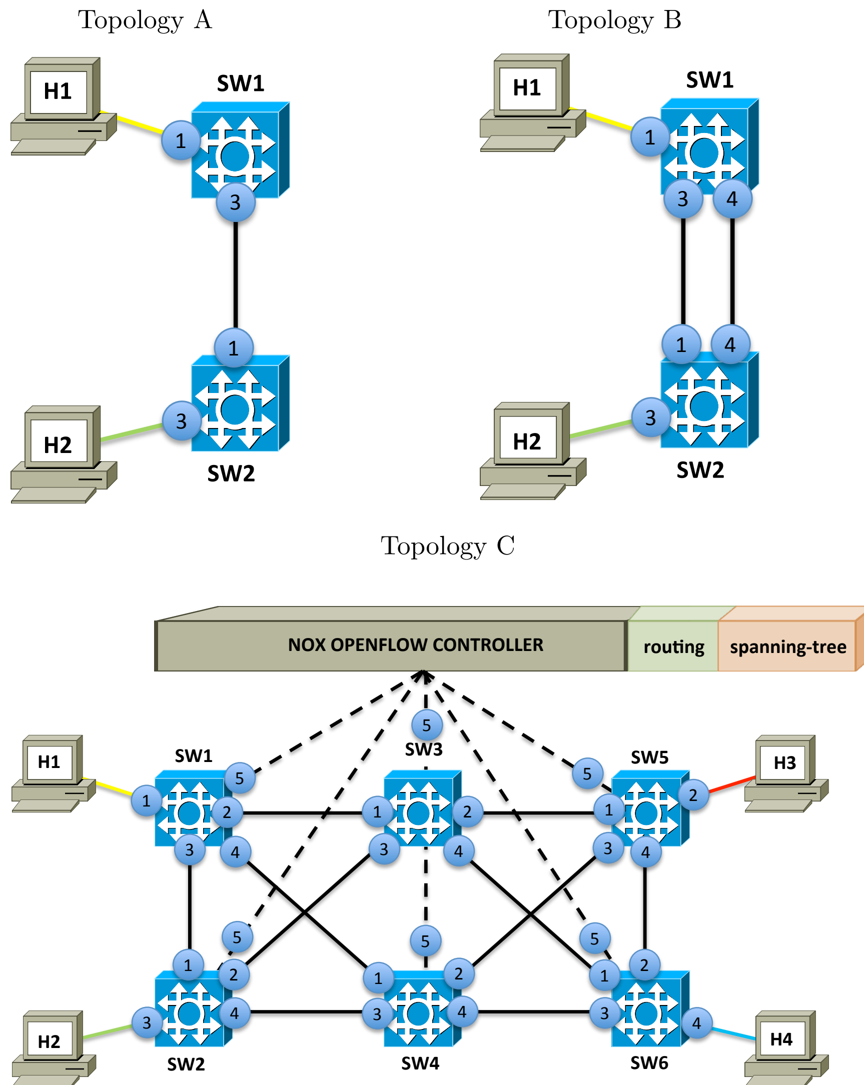
- Start Wireshark logging.
- Start tcpdump on hosts.
- Start controller with specific module.
- Wait for 1 minute till controller is fully started.
- Send PING request from source to destination host.
- Collect current flows snapshot from all relevant router using collect script.

16.2.3 Resetting test

Between each test, we resetted the test environment using the following steps (where applicable):

- Clear ARP entries on hosts.
- Restart NOX controller
- Delete all current flows from switches
- Reset Wireshark and tcpdump

16.3 Topologies



16.4 Description of tests

Manual flow creation

Create flows manual through CLI, verify connectivity using PING from host 1 to host 2. Execute test for connectivity both ways (source and destination host swapped)

Flows test 1a (identical for both switches)

```
dpctl add-flow tcp:localhost in_port=1,actions=output:3
dpctl add-flow tcp:localhost in_port=3,actions=output:1
```

Flows test 1b (switch 1)

```
dpctl add-flow tcp:localhost in_port=1,actions=output:3,4
dpctl add-flow tcp:localhost in_port=3,4,actions=output:1
```

Flows test 1b (switch 2)

```
dpctl add-flow tcp:localhost in_port=3,actions=output:1,4
dpctl add-flow tcp:localhost in_port=1,4,actions=output:3
```

	Purpose	**Topology**
Test 1a	Verify setup and understanding of technology.	Topology A
Test 1b	Verify setup and understanding of technology.	Topology B

pyswitch module

*Load **pyswitch** module in NOX controller, verify connectivity using PING from host 1 to host 2. Execute test for connectivity both ways (source and destination host swapped)*

Command: `sudo ./nox_core -v -v -i ptcp:6633 pyswitch`

	Purpose	**Topology**
Test 2a	Verify setup and understanding of technology.	Topology A
Test 2b	Verify setup and understanding of technology.	Topology B

routing module

*Load **routing** module in NOX controller, verify connectivity using PING from host 1 to host 2. Execute test for connectivity both ways (source and destination host swapped)*

Command: `sudo ./nox_core -v -v -i ptcp:6633 routing`

	Purpose	**Topology**
Test 3a	Understand and verify working of routing module	Topology A
Test 3b	Understand and verify working of routing module	Topology B

*Load **routing** module in NOX controller, send PING from host 1 to host 4. Monitor which path is actually being used. Execute test for both ways (source and destination host swapped). Iterate 10 times to determine actual path taken.*

Command: `sudo ./nox_core -v -v -i ptcp:6633 routing`

Test 4	Determine path determination algorithm	Topology C

*Load **routing** module in NOX controller, send PING from host 1 to host 4. Remove active ethernet link to verify link failover algorithm.*

Command: `sudo ./nox_core -v -v -i ptcp:6633 routing`

Test 5	Determine link failover algorithm	Topology C

*Load **routing** module in NOX controller, start multiple TCP streams (using iperf) from host 1 to host 4. Monitor which path are actually being used and if traffic is load balanced across network.*

Command: `sudo ./nox_core -v -v -i ptcp:6633 routing`

Test 6	Determine load balancing capability	Topology C

16.5 Result summary

Manual flow creation

	Result
Test 1a	Success
Test 1b	Success, traffic was delivered twice; H1→SW1-1 →SW2-1 →H2 →SW1-2 →SW2-2 →H2

Pyswitch

	Result
Test 2a	Success
Test 2b	Success

Routing

	Result
Test 3a	Success
Test 3b	Success
Test 4	Success
	Host 1 →host 2 Path taken: **(1) H1**→(**SW1**-1 →SW1-2) →(**SW3**-1 →SW3-4) →(**SW6**-1 →SW6-4) →**H2** Path taken: **(2) H1**→(**SW1**-1 →SW1-2) →(**SW3**-1 →SW3-4) →(**SW6**-1 →SW6-4) →**H2** Path taken: **(3) H1**→(**SW1**-1 →SW1-2) →(**SW3**-1 →SW3-4) →(**SW6**-1 →SW6-4) →**H2** Path taken: **(4) H1**→(**SW1**-1 →SW1-2) →(**SW3**-1 →SW3-4) →(**SW6**-1 →SW6-4) →**H2** Path taken: **(5) H1**→(**SW1**-1 →SW1-2) →(**SW3**-1 →SW3-4) →(**SW6**-1 →SW6-4) →**H2** Path taken: **(6) H1**→(**SW1**-1 →SW1-2) →(**SW3**-1 →SW3-4) →(**SW6**-1 →SW6-4) →**H2** Path taken: **(7) H1**→(**SW1**-1 →SW1-2) →(**SW3**-1 →SW3-4) →(**SW6**-1 →SW6-4) →**H2** Path taken: **(8) H1**→(**SW1**-1 →SW1-2) →(**SW3**-1 →SW3-4) →(**SW6**-1 →SW6-4) →**H2** Path taken: **(9) H1**→(**SW1**-1 →SW1-2) →(**SW3**-1 →SW3-4) →(**SW6**-1 →SW6-4) →**H2** Path taken: **(10) H1**→(**SW1**-1 →SW1-2) →(**SW3**-1 →SW3-4) →(**SW6**-1 →SW6-4) →**H2**

	Host 2 →**host 1** Path taken: **(1) H2**→(**SW6-**4 →SW6-1) →(**SW3-**4 →SW3-1) →(**SW1-**2 →SW1-1) →**H1** Path taken: **(2) H2**→(**SW6-**4 →SW6-1) →(**SW3-**4 →SW3-1) →(**SW1-**2 →SW1-1) →**H1** Path taken: **(3) H2**→(**SW6-**4 →SW6-1) →(**SW3-**4 →SW3-1) →(**SW1-**2 →SW1-1) →**H1** Path taken: **(4) H2**→(**SW6-**4 →SW6-1) →(**SW4-**4 →SW4-1) →(**SW1-**2 →SW1-1) →**H1** Path taken: **(5) H2**→(**SW6-**4 →SW6-1) →(**SW3-**4 →SW3-1) →(**SW1-**2 →SW1-1) →**H1** Path taken: **(6) H2**→(**SW6-**4 →SW6-1) →(**SW4-**4 →SW4-1) →(**SW1-**4 →SW1-1) →**H1** Path taken: **(7) H2**→(**SW6-**4 →SW6-1) →(**SW3-**4 →SW3-1) →(**SW1-**2 →SW1-1) →**H1** Path taken: **(8) H2**→(**SW6-**4 →SW6-1) →(**SW4-**4 →SW4-1) →(**SW1-**4 →SW1-1) →**H1** Path taken: **(9) H2**→(**SW6-**4 →SW6-1) →(**SW3-**4 →SW3-1) →(**SW1-**2 →SW1-1) →**H1** Path taken: **(10) H2**→(**SW6-**4 →SW6-1) →(**SW3-**4 →SW3-1) →(**SW1-**2 →SW1-1) →**H1**
Test 5	Traffic path before link removal: **H1**→(**SW1-**1 →SW1-2) →(**SW3-**1 →SW3-4) →(**SW6-**1 →SW6-4) →**H2** After removal of the link between switch 1 and 3, the traffic was interrupted and not (automatically) resumed. After removal of the link between switch 3 and 6, the traffic was resume after the current flows (in switch 3 and 6) had timed out. The flow on switch 1 (towards to removed link) did not timeout, because packets kept coming in from host 1. If the traffic flow was stopped until the failing flow had timed out, or the flow was removed manually, the controller reinstalled a new flow forwarding traffic along another path. In other words, traffic was only send along a new path after the current flows where removed from the switch by timeout or manual action. The controller itself didn't notify the switch to remove the failing flow rule.
Test 6	During this test, all traffic flows took the same path (**H1**→(**SW1-**1 →SW1-2) →(**SW3-**1 →SW3-4) →(**SW6-**1 →SW6-4) →**H2**) No load balancing of traffic between the hosts occurred.
Notes	While working with the routing module, we noticed sometimes traffic did get stuck in a loop. This most probable reason for this is when a host gets authenticated (associated to) multiple datapaths (switches) in looped topology[20].

17 Appendix F: Pseudocode

17.1 Introduction

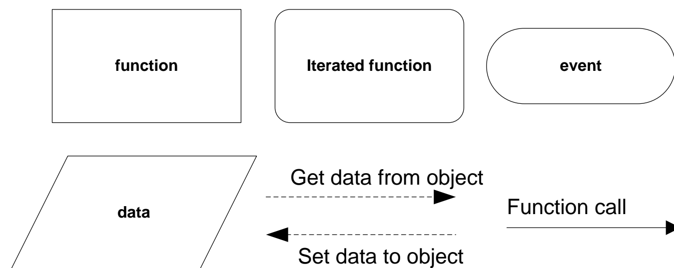
This proposal tries to solve the issue of non effective failover on link failure as described in this paper. It does this by keeping a local cache of all flows installed on the switches in the network. Whenever a topology change occurs, all effected flows are removed. This in turn triggers a recalculation of the active path.

17.2 Component overview

Function interaction

data_packet_in
dispatcher
known_destination?
yes
No
spanning_tree
Dijkstra
route_packet
flood_packet
host_table
link_timeout_watch
topology_table
registered_
datapaths
topology_packet_in
receive_topology_
packet
Flow expired
notification
flow_expired
flow_table
send_topology_packet

Legend



17.3 Events

Event	Handler
Receival of data packet	dispatcher(dpid_src, in_port, packet)
Receival of topology discovery packet	receive_topology_packet(dpid,packet)
Notification of switch that flow has expired and been removed from local flow table	update_flow_table(dpid, flow)

17.4 Library calls

Method	Description
create_flow	API call to create a flow on a datapath
delete_flow	API call to remove a flow from a datapath
dijkstra	Library call to calculate shortest path between two nodes
spanning_tree	Library call to calculate a spanning tree and program to the datapath ports using portmod packets
send_packet	API call to send packet to datapath

17.5 Functions

function dispatcher(dpid_src, in_port, packet)

When a data packet is received, the dispatcher function is called. This function first insert the source MAC and port into the MAC to port mapping table. Then it determines whether the frame should be flooded or routed to its destination.

Gets data from	Description
host_table	To determine if host is already known.
Calls to other functions	**Description**
route_packet	To forward packet if destination is known.
flood_packet	To forward packet if destination is unknown.

```
#Called on data_packet_in event, determines whether packet should be should be
    flooded or routed.

#Learn src MAC and store into local MAC to port mapping table.
        host_table[dpid_src,in_port ] = packet.src

#Check if DST mac has already been learned in local MAC to port mapping table.
        if packet.dst in host_table
                call route_packet(dpid_src, in_port, packet)

        #If DST mac is unknown; flood packet through tree:
        else:
                call flood_packet(dpid_src, in_port, packet)
```

---

function route_packet(dpid_src, in_port, packet)

If the destination of the packet is known, it can be routed to its destination. This function gets the destination datapath and port from the host_table and calls the dijkstra function to calculate the shortest path. It then programs an appropriate flow to all intermediate datapaths between source and destination. It also stores the newly created flows in a local table, needed to determine which routes should be recalculated in a topology change.

Gets data from	**Description**
host_table	Get location of destination host.
Sets data to	**Description**
flow_table	Store newly created flow in local table.
Calls to other functions	**Description**
dijkstra	Function to calculate shortest path.
create_flow	Function to program flow to datapaths.

```
#Get destination switch from host_table
        destination = host_table[packet.dst]

#Split destination into destination datapath and outport.
        dpid_dst = destination[1]
        action = destination[2]

#Get shortest path from source to destination. Function returns array with all
     flow entries that should be programmed. The source datapath is needed
    because this is the switch where the complete path should originate.
        best_route = call dijkstra(dpid_src,in_port,dpid_dst,out_port)

#Program route to switches
        for each (dpid, in_port, action) in best_route do:
                call create_flow(dpid, in_port, action, packet.src, packet.dst
                    , packet.prot)

#Update flow_table with newly created flow
        for each (dpid, in_port, action) in best_route do:
                flow_table.add(dpid, in_port, action, packet.src, packet.dst,
                    packet.prot)
```

---

function flood_packet(dpid_src, in_port, packet)

Because the location of the destination address is unknown this function is called to flood the frame through the network.

Gets data from	**Description**
registered_datapaths	To get a list of all active datapaths, itering through and create flow with flood action.
Calls to other functions	**Description**
create_flow	Function to program the flow with action FLOOD to all datapaths.

```
#Program flood route to switches
action =  FLOOD

#Create flood flow for all registered switches
        for each dpid in registered_datapaths do:
                call create_flow(dpid, in_port, action, packet.src, packet.dst
                   , packet.prot)
```

---

function send_topology_packet

Send topology discovery packet out every port of every registered datapath. Packet contents: source datapath, source_port. Function is iterated every 1 second.

Gets data from	**Description**
registered_datapaths	Get a list of all registered datapaths, iterate through and ask switch to send topology packet.
Calls to other functions	**Description**
send_packet	Function to ask switch to send topology packet.

```
#Iterate through all datapaths and send out topology discovery packets for
    each port.
        for each(id,dpid,port,speed) in registered_datapaths do:
                send_packet(dpid[dpid,portid])
```

---

function receive_topology_packet(dpid,packet)

For any incoming topology_packet evaluate link and update `topology_table` to reflect actual topology. For links that already have been discovered, update the last received timer value. This is needed to discard outdated link information.

Sets data to	**Description**
topology_table	When a new link is discovered, use information in packet to update this table.
Gets data from	**Description**
topology_table	When a known link is rediscovered, update timer value in table.

```
#Set data into variables
        dpid_src = packet.src
        dpid_dst = dpid
        outport = packet.outport
        inport = packet.inport

#Check if the link has already been discovered. If so, only update last_update
     timer value.
```

```
if (dpid_src ,outport ,dpid_dst ,inport) in topology_table do:
        topology_table(id ,dpid_src ,outport ,dpid_dst ,inport).update(
           last_update) = "0"
else
#Else add link to topology table.
        topology_table.add(dpid_src ,outport ,dpid_dst ,inport ,"0")
```

---

function flow_expired(packet.src,packet.dst,dpid,in_port,action,packet.prot)

When a datapaths notifies the controller of the expiration of a flow, this function is called. It is used to keep the local flow table up to date.

Sets data to	**Description**
flow_table	Update the flowtable when a datapath notifies the controller that is has removed a certain flow.

```
#Remove expired flow from local flow table
        flow_table.remove(id ,packet.src ,packet.dst ,dpid ,in_port ,action ,packet.
           prot)
```

---

function link_timeout_watch

If any link in the topology table has not been discovered for more than 3 seconds, delete link from topology table. Then check in flow_table if this link is used in current, active flows and if so, delete flow from flow_table and datapaths.

Gets data from	**Description**
flow_table	Check if timed out link is used in current flows.
topology_table	Check if link is timed out.
Sets data to	**Description**
topology_table	If link is timed out, remove from table.
flow_table	If timed out link is used in a any current flows, remove flows from this table.

```
for each (id ,dpid_src ,src_port ,dpid_dst ,dst_port ,last_update) in
    topology_table do:
        if last_update > 3
                topology_table.remove(id)

#Check if link is used in current flows, if so, delete flow local table and
    switches.
                for each (id ,src ,dst ,dpid ,in_port ,action ,prot) in flow_table
                    do:
                        if ((dpid = dpid_src) and (in_port = src_port) and (
                            action=dst_port))
```

```
#Remove flow from local table
flow_table.remove(id)

#Remove flow from switch
delete_flow(dpid,src,dst,in_port,action,prot)
```

17.6 Tables

These tables represent examples of the variables in use.

topology_table

id	dpid_src	src_port	dpid_dst	dst_port	last_update
1	1	2	3	1	1
2	1	3	2	1	2
3	1	4	4	1	3
4	2	1	1	3	4
5	2	2	3	3	2
6	2	4	4	3	1
7	3	1	1	2	3
8	3	2	5	1	1
9	3	3	5	2	3
10	3	4	6	3	1
11	4	1	1	4	3
12	4	2	5	3	2
13	4	3	2	4	2
14	4	4	6	3	3

host_table

id	**MAC**	**dpid, port**
1	00:00:00:00:00:00:01	1,1
2	00:00:00:00:00:00:02	2,3
3	00:00:00:00:00:00:03	5,2
4	00:00:00:00:00:00:04	6,4

flow_table

id	src	dst	dpid	in_port	out_port	prot
1	00:00:00:00:00:00:01	00:00:00:00:00:00:04	1	1	2	arp

registered_datapaths

id	dpid	port	speed
1	1	1	1000
2	1	2	1000
3	1	3	1000
4	1	4	1000
5	2	1	1000
6	2	2	1000
7	2	3	1000
8	2	4	1000
9	3	1	1000
10	3	2	1000