**CELL iteration 3 10/10/16: Tasks 3.5, POX spanning Tree NetApps; Dottie Kessler 10/2/16**

1. Start VM, login (mininet/mininet) and obtain ip (**sudo dhclient eth1**)
2. Run Xming: then xterm from within mininet
3. Putty to VM; run wireshark then filter of ; capture->interfaces->loopback (**sudo wireshare &**)
4. Putty to VM run ODL (**cd ~/opendaylight; sudo ./run.sh -virt ovsdb**)
5. Browser to host open IP of VM (admin/admin) [**http://192.168.56.101:8080**](http://192.168.56.101:8080)

**Task 3.5 Spanning Tree Program (POX)**

**Task instructions**

1. Spanning Tree, you need to get files from your laptop to the VM
2. My attempts to have the VM read my usb were unsuccessful
3. I did a copy paste of the file into the vm
4. The filenames in the directions have \_ underscore, it should be – hyphen
5. topo-2sw-2host.py.dak spanning-tree.py.dak
6. Save the original files this will be replacing
7. Uses forwarding\_l2 learning
8. The L2 Learning component makes OpenFlow switches act like Ethernet learning switches
9. Stores a MAC-to-port table for each DPID.

**Task execution**

1. **start mininet with the replaced custom topology python file**

**cd ~mininet/custom**

**sudo mn --custom topo-2sw-2host.py --topo mytopo --controller remote**

mininet@mininet-vm:~/mininet/custom$ **ls**

README topo-2sw-2host.py topo-2sw-2host.py.dak topo-2sw-2host.py.orig topox.py

mininet@mininet-vm:~/mininet/custom$ **mv topo-2sw-2host.py.dak topo-2sw-2host.py**

mininet@mininet-vm:~/mininet/custom$ **sudo mn --custom topo-2sw-2host.py --topo mytopo --controller remote**

\*\*\* Creating network

\*\*\* Adding controller

Unable to contact the remote controller at 127.0.0.1:6633

\*\*\* Adding hosts:

h1 h2 h3 h4

\*\*\* Adding switches:

s3 s4 s5 s6

\*\*\* Adding links:

(h1, s3) (s3, s4) (s3, s5) (s4, h2) (s4, s5) (s5, h3) (s6, h4) (s6, s3) (s6, s4)

\*\*\* Configuring hosts

h1 h2 h3 h4

\*\*\* Starting controller

c0

\*\*\* Starting 4 switches

s3 s4 s5 s6 ...

\*\*\* Starting CLI:

mininet>

1. **pingall fails here, there is no controller yet**

mininet> **pingall**

\*\*\* Ping: testing ping reachability

h1 -> X X X

h2 -> X X X

h3 -> X X X

h4 -> X X X

\*\*\* Results: 100% dropped (0/12 received)

1. **Replace spanning\_tree.py with file from the zip**

mininet@mininet-vm:~/freshpox/pox/pox/openflow$ **cp spanning\_tree.py spanning\_tree.py.orig**

mininet@mininet-vm:~/freshpox/pox/pox/openflow$ **cp $HOME/dak/iteration3/spanning\*** .

mininet@mininet-vm:~/freshpox/pox/pox/openflow$ **ls**

debug.py flow\_table.py libopenflow\_01.py of\_01.py spanning\_tree.py util.py

debug.pyc \_\_init\_\_.py libopenflow\_01.pyc of\_01.pyc spanning-tree.py.dak util.pyc

discovery.py \_\_init\_\_.pyc nicira\_ext.py of\_json.py spanning\_tree.py.orig webservice.py

discovery.pyc keepalive.py nicira.py of\_service.py topology.py

mininet@mininet-vm:~/freshpox/pox/pox/openflow$ **mv spanning-tree.py.dak spanning\_tree.py**

mininet@mininet-vm:~/freshpox/pox/pox/openflow$

**/home/mininet/freshpox/pox/pox/openflow**

1. **Run POX spanning tree**

mininet@mininet-vm:~/freshpox/pox/pox/openflow$ **/home/mininet/freshpox/pox/pox.py log.level --DEBUG** **samples.spanning\_tree**

POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.

[samples.spanning\_tree ] Using forwarding: pox.forwarding.l2\_learning

[core ] POX 0.2.0 (carp) going up...

[core ] Running on CPython (2.7.6/Mar 22 2014 22:59:56)

[core ] Platform is Linux-3.13.0-24-generic-x86\_64-with-Ubuntu-14.04-trusty

[core ] POX 0.2.0 (carp) is up.

[openflow.of\_01 ] Listening on 0.0.0.0:6633

[openflow.of\_01 ] [00-00-00-00-00-05 1] connected

[openflow.discovery ] Installing flow for 00-00-00-00-00-05

[forwarding.l2\_learning] Connection [00-00-00-00-00-05 1]

[openflow.of\_01 ] [00-00-00-00-00-03 3] connected

[openflow.discovery ] Installing flow for 00-00-00-00-00-03

[forwarding.l2\_learning] Connection [00-00-00-00-00-03 3]

[openflow.of\_01 ] [00-00-00-00-00-06 4] connected

[openflow.discovery ] Installing flow for 00-00-00-00-00-06

[forwarding.l2\_learning] Connection [00-00-00-00-00-06 4]

[openflow.of\_01 ] [00-00-00-00-00-04 2] connected

[openflow.discovery ] Installing flow for 00-00-00-00-00-04

[forwarding.l2\_learning] Connection [00-00-00-00-00-04 2]

[openflow.discovery ] link detected: 00-00-00-00-00-05.1 -> 00-00-00-00-00-03.3

[openflow.discovery ] link detected: 00-00-00-00-00-05.2 -> 00-00-00-00-00-04.3

[openflow.discovery ] link detected: 00-00-00-00-00-03.3 -> 00-00-00-00-00-05.1

[openflow.discovery ] link detected: 00-00-00-00-00-03.4 -> 00-00-00-00-00-06.1

[openflow.discovery ] link detected: 00-00-00-00-00-03.2 -> 00-00-00-00-00-04.1

[openflow.discovery ] link detected: 00-00-00-00-00-06.1 -> 00-00-00-00-00-03.4

[openflow.discovery ] link detected: 00-00-00-00-00-06.2 -> 00-00-00-00-00-04.4

[openflow.discovery ] link detected: 00-00-00-00-00-04.3 -> 00-00-00-00-00-05.2

[openflow.discovery ] link detected: 00-00-00-00-00-04.1 -> 00-00-00-00-00-03.2

[openflow.discovery ] link detected: 00-00-00-00-00-04.4 -> 00-00-00-00-00-06.2

[openflow.spanning\_tree] \*\*\* SPANNING TREE \*\*\*

3 : [4, 5, 6]

4 : [3]

5 : [3]

6 : [3]

[openflow.spanning\_tree] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[openflow.spanning\_tree] 10 ports changed

[openflow.spanning\_tree] \*\*\* SPANNING TREE \*\*\*

3 : [4, 5, 6]

4 : [3]

5 : [3]

6 : [3]

[openflow.spanning\_tree] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[openflow.spanning\_tree] \*\*\* SPANNING TREE \*\*\*

3 : [4, 5, 6]

4 : [3]

5 : [3]

6 : [3]

[openflow.spanning\_tree] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[openflow.spanning\_tree] \*\*\* SPANNING TREE \*\*\*

3 : [4, 5, 6]

4 : [3]

5 : [3]

6 : [3]

[openflow.spanning\_tree] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**So from this example output we see that the spanning tree protocol removes the links (removes the loops) from switch 4 to switch 6 and switch 4 to switch 5**



1. **Now pingall should work**

mininet> **pingall**

\*\*\* Ping: testing ping reachability

h1 -> h2 h3 h4

h2 -> h1 h3 h4

h3 -> h1 h2 h4

h4 -> h1 h2 h3

\*\*\* Results: 0% dropped (12/12 received)

mininet>

1. **Follow the Spanning Tree Instructions and see if the results from changing the program match the displays that are included**

**Dottie Kessler(4:36:33 PM):** I have a question on iteration 3, task 3.5 spanning tree. I followed the instructions and replaced the topo-2sw-2host.py and the spanning-tree.py, and my results matched the expected where links 4-6 and 4-5 were no longer there. The next things in the iteration, says to follow the spanning tree instructions below. Where are those instructions, I only see a python file, it isn't clear to me what is wanted here.

**Jerry Cheng(4:53:41 PM):** hi Dorothy, I think the exercise is to look through the python script and understand what the Spanning Tree script is doing, and read the comments within the script that explains each part of the script is doing and make sure your spanning tree script is doing exactly what is posted on iteration 3.5.

1. **Spanning Tree Protocol** (**STP**) is a Layer 2 **protocol** that runs on bridges and switches. The specification for **STP** is IEEE 802.1D. The main purpose of **STP** is to ensure that you do not create loops when you have redundant paths in your network. Loops are deadly to a network.
2. **Layer 2 Switching loops** may cause serious problem to network performance. Layer 2 Switching loops are prevented in networks using [Spanning Tree Protocol](http://www.omnisecu.com/cisco-certified-network-associate-ccna/what-is-spanning-tree-protocol-stp.php).
3. The *Spanning Tree* component is required in cases where the topology of the network contains loops. It works with the OpenFlow Discovery component to build a view of the network topology and constructs a spanning tree by disabling flooding on switch ports that aren’t on the tree. The options *no-flood* and *hold-down* are used to ensure no packets are flooded in the network before the component creates the spanning tree.
4. The *Spanning Tree* component will respond to changes in the network topology. If a link is broken, and if an alternate link exists, it can maintain connectivity in a network by creating a new tree that enables flooding on the ports connected to the alternate link.
5. When using the *Spanning Tree* component, also use a forwarding component that creates flows that have a timeout value set. In this example, we used the *L2 Learning* component.
6. A good explanation

<http://archive.openflow.org/wk/index.php/Basic_Spanning_Tree>

**mininet@mininet-vm:~/dak/iteration3$ cat spanning-tree.py.dak**

# Copyright 2012,2013 James McCauley

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at:

#

# http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

"""

Creates a spanning tree.

This component uses the discovery component to build a view of the network

topology, constructs a spanning tree, and then disables flooding on switch

ports that aren't on the tree by setting their NO\_FLOOD bit. The result

is that topologies with loops no longer turn your network into useless

hot packet soup.

This component is inspired by and roughly based on the description of

Glenn Gibb's spanning tree module for NOX:

http://www.openflow.org/wk/index.php/Basic\_Spanning\_Tree

Note that this does not have much of a relationship to Spanning Tree

Protocol. They have similar purposes, but this is a rather different way

of going about it.

"""

from pox.core import core

import pox.openflow.libopenflow\_01 as of

from pox.lib.revent import \*

from collections import defaultdict

from pox.openflow.discovery import Discovery

from pox.lib.util import dpidToStr

from pox.lib.recoco import Timer

import time

log = core.getLogger()

# Might be nice if we made this accessible on core...

#\_adj = defaultdict(lambda:defaultdict(lambda:[]))

def \_calc\_spanning\_tree ():

"""

Calculates the actual spanning tree

Returns it as dictionary where the keys are DPID1, and the

values are tuples of (DPID2, port-num), where port-num

is the port on DPID1 connecting to DPID2.

"""

def flip (link):

return Discovery.Link(link[2],link[3], link[0],link[1])

adj = defaultdict(lambda:defaultdict(lambda:[]))

switches = set()

# Add all links and switches

for l in core.openflow\_discovery.adjacency:

adj[l.dpid1][l.dpid2].append(l)

switches.add(l.dpid1)

switches.add(l.dpid2)

# Cull links -- we want a single symmetric link connecting nodes

for s1 in switches:

for s2 in switches:

if s2 not in adj[s1]:

continue

if not isinstance(adj[s1][s2], list):

continue

assert s1 is not s2

good = False

for l in adj[s1][s2]:

if flip(l) in core.openflow\_discovery.adjacency:

# This is a good one

adj[s1][s2] = l.port1

adj[s2][s1] = l.port2

good = True

break

if not good:

del adj[s1][s2]

if s1 in adj[s2]:

# Delete the other way too

del adj[s2][s1]

q = []

more = set(switches)

done = set()

tree = defaultdict(set)

while True:

q = sorted(list(more)) + q

more.clear()

if len(q) == 0: break

v = q.pop(False)

if v in done: continue

done.add(v)

for w,p in adj[v].iteritems():

if w in tree: continue

more.add(w)

tree[v].add((w,p))

tree[w].add((v,adj[w][v]))

if True:

log.info("\*\*\* SPANNING TREE \*\*\*")

for sw,ports in tree.iteritems():

#print " ", dpidToStr(sw), ":", sorted(list(ports))

print " ", sw, ":", [l[0] for l in sorted(list(ports))]

#log.info((" %i : " % sw) + " ".join([str(l[0]) for l in

# sorted(list(ports))]))

log.info("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

return tree

# Keep a list of previous port states so that we can skip some port mods

# If other things mess with port states, these may not be correct. We

# could also refer to Connection.ports, but those are not guaranteed to

# be up to date.

\_prev = defaultdict(lambda : defaultdict(lambda : None))

# If True, we set ports down when a switch connects

\_noflood\_by\_default = False

# If True, don't allow turning off flood bits until a complete discovery

# cycle should have completed (mostly makes sense with \_noflood\_by\_default).

\_hold\_down = False

def \_handle\_ConnectionUp (event):

# When a switch connects, forget about previous port states

\_prev[event.dpid].clear()

if \_noflood\_by\_default:

con = event.connection

log.debug("Disabling flooding for %i ports", len(con.ports))

for p in con.ports.itervalues():

if p.port\_no >= of.OFPP\_MAX: continue

\_prev[con.dpid][p.port\_no] = False

pm = of.ofp\_port\_mod(port\_no=p.port\_no,

hw\_addr=p.hw\_addr,

config = of.OFPPC\_NO\_FLOOD,

mask = of.OFPPC\_NO\_FLOOD)

con.send(pm)

\_invalidate\_ports(con.dpid)

if \_hold\_down:

t = Timer(core.openflow\_discovery.send\_cycle\_time + 1, \_update\_tree,

kw={'force\_dpid':event.dpid})

def \_handle\_LinkEvent (event):

# When links change, update spanning tree

(dp1,p1),(dp2,p2) = event.link.end

if \_prev[dp1][p1] is False:

if \_prev[dp2][p2] is False:

# We're disabling this link; who cares if it's up or down?

#log.debug("Ignoring link status for %s", event.link)

return

\_update\_tree()

def \_update\_tree (force\_dpid = None):

"""

Update spanning tree

force\_dpid specifies a switch we want to update even if we are supposed

to be holding down changes.

"""

# Get a spanning tree

tree = \_calc\_spanning\_tree()

log.debug("Spanning tree updated")

# Connections born before this time are old enough that a complete

# discovery cycle should have completed (and, thus, all of their

# links should have been discovered).

enable\_time = time.time() - core.openflow\_discovery.send\_cycle\_time - 1

# Now modify ports as needed

try:

change\_count = 0

for sw, ports in tree.iteritems():

con = core.openflow.getConnection(sw)

if con is None: continue # Must have disconnected

if con.connect\_time is None: continue # Not fully connected

if \_hold\_down:

if con.connect\_time > enable\_time:

# Too young -- we should hold down changes.

if force\_dpid is not None and sw == force\_dpid:

# .. but we'll allow it anyway

pass

else:

continue

tree\_ports = [p[1] for p in ports]

for p in con.ports.itervalues():

if p.port\_no < of.OFPP\_MAX:

flood = p.port\_no in tree\_ports

if not flood:

if core.openflow\_discovery.is\_edge\_port(sw, p.port\_no):

flood = True

if \_prev[sw][p.port\_no] is flood:

#print sw,p.port\_no,"skip","(",flood,")"

continue # Skip

change\_count += 1

\_prev[sw][p.port\_no] = flood

#print sw,p.port\_no,flood

#TODO: Check results

pm = of.ofp\_port\_mod(port\_no=p.port\_no,

hw\_addr=p.hw\_addr,

config = 0 if flood else of.OFPPC\_NO\_FLOOD,

mask = of.OFPPC\_NO\_FLOOD)

con.send(pm)

#log.info("port no: %i config: %i", p.port\_no, flood)

\_invalidate\_ports(con.dpid)

if change\_count:

log.info("%i ports changed", change\_count)

except:

\_prev.clear()

log.exception("Couldn't push spanning tree")

\_dirty\_switches = {} # A map dpid\_with\_dirty\_ports->Timer

\_coalesce\_period = 2 # Seconds to wait between features requests

def \_invalidate\_ports (dpid):

"""

Registers the fact that port info for dpid may be out of date

When the spanning tree adjusts the port flags, the port config bits

we keep in the Connection become out of date. We don't want to just

set them locally because an in-flight port status message could

overwrite them. We also might not want to assume they get set the

way we want them. SO, we do send a features request, but we wait a

moment before sending it so that we can potentially coalesce several.

TLDR: Port information for this switch may be out of date for around

\_coalesce\_period seconds.

"""

if dpid in \_dirty\_switches:

# We're already planning to check

return

t = Timer(\_coalesce\_period, \_check\_ports, args=(dpid,))

\_dirty\_switches[dpid] = t

def \_check\_ports (dpid):

"""

Sends a features request to the given dpid

"""

\_dirty\_switches.pop(dpid,None)

con = core.openflow.getConnection(dpid)

if con is None: return

con.send(of.ofp\_barrier\_request())

con.send(of.ofp\_features\_request())

log.debug("Requested switch features for %s", str(con))

def launch (no\_flood = True, hold\_down = True):

global \_noflood\_by\_default, \_hold\_down

if no\_flood is True:

\_noflood\_by\_default = True

if hold\_down is True:

\_hold\_down = True

def start\_spanning\_tree ():

core.openflow.addListenerByName("ConnectionUp", \_handle\_ConnectionUp)

core.openflow\_discovery.addListenerByName("LinkEvent", \_handle\_LinkEvent)

log.debug("Spanning tree component ready")

core.call\_when\_ready(start\_spanning\_tree, "openflow\_discovery")