SPIDER: Fault Resilient SDN Pipeline with Recovery Delay Guarantees

Carmelo Cascone*, †, Luca Pollini^, Davide Sanvito^, Antonio Capone*, Brunilde Sansò†

* Politecnico di Milano, Italy
^CNIT – Consorzio Nazionale Universitario per le Telecomunicazioni, Italy
† Ecole Polytechnique de Montreal, Canada

Supported by EU Project
Outline

• Motivations
• Goal
• SPIDER
• Numerical results
• European H2020 research project

• Started January 2015

• Goal: programmable stateful packet processing in the fast-path
  – Allow pre-configuration of different sets of forwarding rules to be applied according to the observed network state
  – in-switch fast state evolution according to packet-level events/time-based events/flow level measurements

www.beba-project.eu
Fault Resiliency in SDN

- Weak support by current data plane abstractions
- OpenFlow
  - Stateless match+action
  - requires remote controller to reconfigure the forwarding
    - Additional overhead and latency $\Rightarrow$ hard to obtain carrier-grade recovery times ($<50\text{ ms}$)
  - Fast-Failover group type:
    - limited to local failures protection
    - Liveness checking out of spec. $\Rightarrow$ No guarantees on detection delays ($1\text{ms} - 500\text{ms}$)
Related Works

• Integration of a BFD deamon with OF Fast-Failover
• fine-tuning of parameters in Open vSwitch’s BFD process
• OF extension to implement in-switch link monitoring functions
• OF extension with a flow entry auto-rejecting mechanism based on port status

Mainly based on patching OF Fast-Failover and BFD $\rightarrow$ slow-path
SPIDER Goal

• Provide a forwarding pipeline design to allow:
  – End-to-end proactive protection independent from controller reachability
  – Programmable sub-millisecond detection delay

• Inspired by legacy technologies
  – BFD
  – MPLS Fast Reroute
Stateful Dataplane

- Switch maintains flow memory across different packets
- Forwarding is based on packet fields and current flow state
- The controller can delegate to switches local changes in the forwarding
OpenState

• Stateful dataplane
  – Statefulness in the fast-path → state updates at wire-speed!

• Stateful OpenFlow extension

• Pipeline of stateless/stateful stages

• Forwarding behaviour modeled as Finite State Machine (FSM)


Other examples of stateful dataplane

• OVS: learn() action
  – OF extension → slow-path only

• P4: stateful memories
  – We can describe OpenState
OpenState Stateful Stage

- **State Table**
  - Associates a flow key (exact match) with a state
  - Flow key extractor (lookup-scope and update-scope)

- **Flow Table**
  - Classic OF match+action table
  - New *state* match field
  - New *set state* action
FSM

- OpenState stateful forwarding abstraction is based on FSM
- Forwarding depends on current state + packet header fields
- State transitions
  - Packet-driven
  - Time-based

- FSM structure is defined by the controller at boot-time
  - by inserting flow table entries
  - by configuring lookup-scope and update-scope

- The switch executes the FSM at run-time
  - by storing flow states in the state table
  - by updating the states
SPIDER
Stateful Programmable failure Detection and Recovery

- Fault resilient SDN pipeline design
- Fully programmable failure detection and recovery in the fast-path
  - Sub-millisecond detection & reroute (device timeout granularity)
- Based on stateful dataplane abstraction
  - Implementation in OpenState
- Instantaneous in-switch recovery from any pre-planned failure scenario
  - Controller intervention needed only in case of unplanned failures
- Programmable failure detection
  - BFD-like
- Fast reroute
  - Inspired by MPLS
  - For both local and non-local failures
  - Path probing
  - Flowlet-aware rerouting
Preplanning of Primary/Backup Paths

- Given:
  - network topology
  - set of demands

We need to provide the controller with a set of primary path (PP) and backup paths (BP) for each possible failure affecting the PP of a given demand.

- The controller then creates the switch pipeline configuration
  - FSM instantiation
  - Flow table entries
  - Forwarding based on L2 src-dst addresses and MPLS label

Failure Detection

Assumption:
As long as packets are received from a given port, that port can be also used to transmit packets.

- If no packet is received from port $x$ within a $\delta_1$ interval:
  - Next data packet towards port $x$ is tagged with a special value (Heartbeat request)
  - Port $x$ is declared down if adjacent node does not send back a copy (Heartbeat reply) within a $\delta_2$ interval

- Configurable trade off: overhead vs. failover responsiveness
  - Heartbeat requests generation timeout ($\delta_1$)
  - Heartbeat reply timeout ($\delta_2$) before the port is declared down

- Guaranteed max detection delay: $\delta_1 + \delta_2$
Failure Detection FSM

$\delta_1 =$ HB requests generation timeout
$\delta_2 =$ HB reply timeout

Lookup-scope = [metadata]
Update-scope = [metadata]
Fast Reroute (local)

- MPLS label used to distinguish between different forwarding:
  - No tag $\rightarrow$ forward packet on the primary path
  - tag=Fi $\rightarrow$ forward packet on the detour for the i-th failure

- Zero losses after failure detection
- No controller intervention
- What if no local alternative path is available?
Fast Reroute (remote)

- Packets are tagged and bounced back up to a proper redirect point
- Tagged packets trigger a state transition:
  - updating the routing of the involved connections
- Still zero losses after failure detection!
- Tagged data packets as signalling
- No controller intervention!
Path Probing

- How to restore the forwarding on the primary path?
- Programmable periodic probing for primary path availability
Flowlet-aware Rerouting

- Failover activation/deactivation can be post-poned
  - In order to minimize out-of-sequence, packets are kept on the primary path up to expiration of a burst of packets
  - Programmable idle timeout/hard timeout
Putting all together: Fast reroute FSM
Results: Detection Mechanism

Unidirectional demand h1->h2 @1000 pkt/s

The plot shows the number of packets lost by tuning:

- Heartbeat requests generation timeout ($\delta_1$)
- Heartbeat reply timeout ($\delta_2$)
Results: Heartbeat Overhead

Unidirectional demand h1->h2 @100 pkt/s
Unidirectional demand h2->h1 from 200 to 0 pkt/s

HB req rate = 1/δ₁

Hearbeat packets are requested only if incoming traffic rate is lower than 1/ δ₁

Overhead does not affect link available capacity!
Results: comparison with OpenFlow

We compared SPIDER to a reactive OF application:
- failure detection with Fast-Failover Group Table
- controller installs new forwarding rules

Losses in SPIDER: detection phase only
Losses in OF FF: detection phase + failover phase
Results: Complexity Analysis

\( n \times n \) grid networks with a traffic demand for each pair of outer nodes of the grid

Number of flow entries per node is \( O(E^2 \times N) \)

Worst case scenario: E2E path protection

With a more efficient protection scheme (segment) we can even obtain a lower number of rules per node

<table>
<thead>
<tr>
<th>Net</th>
<th>D</th>
<th>E</th>
<th>C</th>
<th>min</th>
<th>avg</th>
<th>max</th>
<th>( E^2 \times N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5x5</td>
<td>240</td>
<td>16</td>
<td>9</td>
<td>443</td>
<td>775</td>
<td>968</td>
<td>6400</td>
</tr>
<tr>
<td>6x6</td>
<td>380</td>
<td>20</td>
<td>16</td>
<td>532</td>
<td>1115</td>
<td>1603</td>
<td>14400</td>
</tr>
<tr>
<td>7x7</td>
<td>552</td>
<td>24</td>
<td>25</td>
<td>795</td>
<td>1670</td>
<td>2404</td>
<td>28224</td>
</tr>
<tr>
<td>8x8</td>
<td>756</td>
<td>28</td>
<td>36</td>
<td>1069</td>
<td>2232</td>
<td>3726</td>
<td>50176</td>
</tr>
<tr>
<td>9x9</td>
<td>992</td>
<td>32</td>
<td>49</td>
<td>1368</td>
<td>2884</td>
<td>4509</td>
<td>82944</td>
</tr>
<tr>
<td>10x10</td>
<td>1260</td>
<td>36</td>
<td>64</td>
<td>1188</td>
<td>3584</td>
<td>6153</td>
<td>129600</td>
</tr>
<tr>
<td>11x11</td>
<td>1560</td>
<td>40</td>
<td>81</td>
<td>1409</td>
<td>4249</td>
<td>7558</td>
<td>193600</td>
</tr>
<tr>
<td>12x12</td>
<td>1892</td>
<td>44</td>
<td>100</td>
<td>1185</td>
<td>5124</td>
<td>9697</td>
<td>278784</td>
</tr>
<tr>
<td>13x13</td>
<td>2256</td>
<td>48</td>
<td>121</td>
<td>2062</td>
<td>6218</td>
<td>11025</td>
<td>389376</td>
</tr>
<tr>
<td>14x14</td>
<td>2652</td>
<td>52</td>
<td>144</td>
<td>1467</td>
<td>7151</td>
<td>15436</td>
<td>529984</td>
</tr>
<tr>
<td>15x15</td>
<td>3080</td>
<td>56</td>
<td>169</td>
<td>3715</td>
<td>8461</td>
<td>16347</td>
<td>705600</td>
</tr>
</tbody>
</table>

SPIDER worst case scenario

Big-O analysis
Software Implementation

- SW implementation based on OpenState
  - Ryu* controller
  - CPqD OpenFlow 1.3 softswitch*
  - https://github.com/OpenState-SDN/spider

- SW implementation in P4 based on openstate.p4 library

*modified with OpenState support http://openstate-sdn.org
Conclusions

• Failure detection and recovery in SDN (OpenFlow) is a major problem
• Statefulness in the data plane allows to implement fast detection and rerouting (<1ms)
  – Independent on controller reachability
  – With guaranteed detection delays
• SPIDER is an example of a pipeline design providing such features
Thank you!

davide.sanvito@polimi.it
BACKUP SLIDE
Example: port knocking

An IP address is allowed to access an UDP server after a secret knock sequence is received.

UDP secret sequence: 10,11,12,13
UDP server port: 22
Example: port knocking (2)

UDP secret sequence: 10,11,12,13
UDP server port: 22

Key extractors:
Lookup-scope = {ip_src}
Update-scope = {ip_src}

<table>
<thead>
<tr>
<th>Priority</th>
<th>Match</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>arp</td>
<td>flood()</td>
</tr>
<tr>
<td>10</td>
<td>state=0, ip, udp_dest=10</td>
<td>set_state(1), drop()</td>
</tr>
<tr>
<td>10</td>
<td>state=1, ip, udp_dest=11</td>
<td>set_state(2), drop()</td>
</tr>
<tr>
<td>10</td>
<td>state=2, ip, udp_dest=12</td>
<td>set_state(3), drop()</td>
</tr>
<tr>
<td>10</td>
<td>state=3, ip, udp_dest=13</td>
<td>set_state(4), drop()</td>
</tr>
<tr>
<td>10</td>
<td>state=4, ip, udp_dest=22</td>
<td>output(2)</td>
</tr>
<tr>
<td>0</td>
<td>ip, udp</td>
<td>set_state(0), drop()</td>
</tr>
</tbody>
</table>
State table memory requirements

• Failure detection state machine
  P state entries (where P is the number of ports)
  5 possible states

• Failover state machine
  D_n state entries (D_n is the number of demands for which node n is a reroute node)
  1 + 4F_n possible states (where F_n is the number of remote failures managed by node n)