OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch

Marco Bonola, CNIT/University of Roma “Tor Vergata”
Introduction and background

Stateful data plane and OpenState
OpenState background

• This presentation describes:
  • The basic OpenState concepts
  • FPGA based proof of concept implementation
  • Ongoing extension work

• OpenState is an OpenFlow extension that enables the execution of Finite State Machines (FSMs) directly at data plane

• In other words, OpenState gives a SDN switch the ability to auto adapt itself and update the forwarding behavior without requiring interaction with the SDN controller

• Project Homepage [http://openstate-sdn.org](http://openstate-sdn.org)
  • Userspace switch, controller and experimenter specification
Motivations

• OpenFlow's platform-agnostic programmatic interface permits to dynamically update match/action forwarding rules only via the explicit involvement of an external controller

• OpenFlow does not permit to deploy forwarding behaviors directly in the switches, i.e. describe how rules should evolve in time as a consequence of packet-level events

• Such static nature of the OpenFlow forwarding abstraction raises serious concerns regarding
  • Scalability
  • Latency
  • Security/reliability
Stateless vs. Stateful in SDN

Stateless data plane model (e.g. OpenFlow)

Controller
Global + local states

Switch
Stateless

Event notifications
Control enforcing

Stateful data plane model

Controller
Global states

Switch
Local states

Auto-adaption
Control delegation

SMART!
SMART!
SMART!
OpenState workflow

1. Key extractor (lookup-scope)
2. State table
3. Flow table
4. set_state(…)
5. Key extractor (update-scope)
6. pkt headers + other actions

pkt headers -> Key extractor (lookup-scope) -> State table -> Flow table

State table -> Key extractor (update-scope) -> pkt headers, state, next_state, timeouts
OpenState architecture

OpenState

**State table**

<table>
<thead>
<tr>
<th>key</th>
<th>state</th>
<th>timeouts</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><em>(any)</em></td>
<td>DEFAULT</td>
<td>n/a</td>
</tr>
</tbody>
</table>

 pkt headers + state

**Flow table**

<table>
<thead>
<tr>
<th>match</th>
<th>actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>headers</td>
<td>state</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>set_state(...)</td>
<td></td>
</tr>
</tbody>
</table>

Key extractor

<lookup-scope>

Key extractor

<update-scope>

(pkt headers + next_state + timeouts)
A simple use case

MAC learning with OpenState
Simple use case: MAC learning

Learn input port for each received frame

<table>
<thead>
<tr>
<th>MAC Addr</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>h3</td>
<td>3</td>
</tr>
</tbody>
</table>
Simple use case: MAC learning

Flood when destination unknown

<table>
<thead>
<tr>
<th>MAC Addr</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>h3</td>
<td>3</td>
</tr>
</tbody>
</table>
Simple use case: MAC learning

<table>
<thead>
<tr>
<th>MAC Addr</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>h3</td>
<td>3</td>
</tr>
<tr>
<td>h2</td>
<td>2</td>
</tr>
</tbody>
</table>

“Response” packets are unicast to h3
MAC learning

Mealy machine

\[ \text{in\_port}=i \]
\[ \text{Forward}(\text{state}[\text{eth\_dst}]) \]

State = Output port:
N switch ports → N + 1 states
0 (DEFAULT) = dest unknown → Flood()

Cross-flow state handling:
State update based on MAC source
Forward based on MAC destination
MAC learning

OpenState table configuration

Key extractors:
Lookup-scope = \{\texttt{eth\_dst}\}
Update-scope = \{\texttt{eth\_src}\}

<table>
<thead>
<tr>
<th>Priority</th>
<th>Match</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>in_port=1, state=0</td>
<td>set_state(1, 0), flood()</td>
</tr>
<tr>
<td>0</td>
<td>in_port=1, state=1</td>
<td>set_state(1, 0), output(1)</td>
</tr>
<tr>
<td>0</td>
<td>in_port=1, state=2</td>
<td>set_state(1, 0), output(2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>in_port=2, state=0</td>
<td>set_state(2, 0), flood()</td>
</tr>
<tr>
<td>0</td>
<td>in_port=2, state=1</td>
<td>set_state(2, 0), output(1)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>in_port=N, state=N</td>
<td>set_state(N, 0), output(N)</td>
</tr>
</tbody>
</table>

N ports switch: \(N^2 + N\) entries
Not only mac learning....

• Fault tolerance and fast failover
• Data driven routing
• Traffic engineering
• Security/monitoring
• Stateful firewall
• CCN

....
FPGA prototype

HW proof of concept implementation of OpenState
FPGA prototype

• The OpenState hardware prototype has been designed using as target development board the INVEA COMBO-LXT, an express PCI x8 mother card equipped with the XILINX Virtex5 FPGA

• The prototype exploits the two 10 GbE interfaces of the board, and the PCI express bus to configure the development board as a 4 port switch

• The FPGA is clocked at 156.25 MHz, with a 64 bits data path from the Ethernet ports, corresponding to a 10 gbps throughput per port

• The 4-input 1-output mixer block aggregates the packets using a round robin policy. The output of the mixer is a 320 bits data bus able to provide an overall throughput of 50 Gbps
Prototype architecture

Four ingress queues collect the packets coming from the ingress ports
A 4-input 1-output mixer block aggregates the packets using a round robin policy. The output of the mixer is a 320 bits data bus able to provide an overall throughput of 50 Gbps.
A delay queue stores the packet during the time need by the OpenState tables to operate
**Protoype architecture**

The **look-up and update extractor blocks** that build the keys that are used to read/update the state table. The 128 bit output is given as input to the state lookup and update.
The state table is realized by the d-left hash table (4k entries, MHT without moving capability) and a small TCAM (32 entries * 128 bits) and a companion SRAM (configured as dual port RAM)
The FSM table is realized by the second TCAM/SRAM pair. The TCAM has 128 entries * 160 bits and the RAM store the next state and an action (if any)
The action block applies the selected actions and forward the packet to the output queues.
Performance

Performance along with the estimation for possible ASIC implementation

• Throughput: 40 Gbps on FPGA @156MHz, 640 Gbps on ASIC @1GHz
• Number of flows in hash table: 4K on FPGA, up to 2M on ASIC
• Number of flows in TCAM: 64 on FPGA, up to 256K on ASIC

FPGA resource occupation
• Number of Slice LUTs: 10,691 out of 24,320 (43%)
• RAM blocks: 53 out of 212 (25%)
Limitation: system latency

• The interval between the first lookup and the last update is 5 clock cycles (5 packets)
• The *feedback loop* may present a problem: the state update performed for a packet at the fifth clock cycle would be missed by pipelined packet
  • This could be an issue for packets belonging to a same flow arriving back-to-back (consecutive clock cycles)

**Considerations and possible workaround**

1. Also the standard control update mechanism of OpenFlow does not allow to exactly determinate at which time instant a new rule is installed in the flow tables
2. By aggregating $n \geq 5$ different links the mixer’s round robin policy will separate two packets coming from the same link of $n$ clock cycles (latency will not increase)
The waveform shows the ingress bus (only one of the ingress queues is presented in the waveform), the four egress queues, and some signals of the hash table and of the TCAM1 and TCAM2 blocks for the port knocking use case.
Beyond Openstate?

Can we support real XFSM?
Mealy Machine: nice but insufficient!

• **True!** Flow processing requires memory, registries, counters, etc
  • State alone is insufficient

• **True!** Flow processing requires operations (compare, add, shift, etc)
  • OpenFlow (forwarding) actions are insufficient

• **True!** Flow processing requires... «processing»
  • Processing = CPU: cannot afford any ordinary CPUs at ns time scales wire speed!

Can we further evolve OpenState into an architecture equivalent to a “full” CPU (Without using any CPU?)

AND CAPABLE OF EXECUTING A PLATFORM AGNOSTIC ABSTRACTION?
Extended finite state machines: much more general!

• Mealy Machines: 4-tuple
  • I, O, S
  • T:S×I×S×O

• XFSM: 7-tuple
  • I, O, S (Input symbols, output symbols, states)
    • As before, S = User-defined
    • D=D1×...×Dn n-dimensional linear space
      • Registries!!! Global or (user-defined) per flow!!
  • F = set of enabling functions fi:D{0,1}
    • Boolean Conditions on registries!!!
  • U = set of update functions ui:DD
    • Update of the registry values!
  • T:S×I×FS×O×U the actual XFSM transition
    • A map can be implemented by the TCAM!
Towards an Open Flow Processor

• HW architecture «executing» an XFSM
  • Seems feasible, via appropriate extension of OpenState

Stage n

Flow Extraction → Flow Context retrieval → Condition Evaluation → Match, action, update functions

To Stage n+1
An OFP program = a platform agnostic abstract XFSM!
(example: a TCP SYN scan detection+mitigation)

**DEFAULT**
- NEW_TCP_FLOW
  - if \( R0 = 0 \)
  - if \( R2 = \text{pkt.ts} \)
  - [OUT]

**DROP**
- ANY_PACKET
  - if \( R1 < \text{pkt.ts} \)
  - if \( R0 = 0 \)
  - if \( R2 = \text{pkt.ts} \)
  - [DROP]

**MONITOR**
- NEW_TCP_FLOW
  - if \( R0 < G0 \)
  - if \( R0 = \text{ewma}(R0,R2,\text{pkt.ts}) \)
  - if \( R2 = \text{pkt.ts} \)
  - [OUT]

**IDLE_TIMEOUT_EXPIRED**
- REMOVE_FLOW_ENTRY

**R0**: TCP SYN rate (EWMA)
**R1**: DROP state expiration timestamp
**R2**: last packet timestamp

**G0**: rate threshold (global)
**G1**: DROP duration (global)

**NEW_TCP_FLOW**

**ANY_PACKET**

**ANY_PACKET**
Acknowledgement

This work has been partly funded by the European Commission in the context of the BEBA H2020 project (Grant Agreement: 644122)

BEBA Homepage http://www.beba-project.eu
OpenState Homepage http://openstate-sdn.org