Platform-agnostic
Behavioral Forwarding and Stateful
Flow Processing at Wire Speed

Giuseppe Bianchi
CNIT / University of Roma Tor Vergata

Joint work with M. Bonola, A. Capone, C. Cascone, S. Pontarelli, D. Sanvito

Supported by EU ICT-05 grant:

Beba
BEhavioural BAse forwarding
Outline

→ Motivation and goal

→ Background and state of the art

→ OpenState: beyond OpenFlow

→ Beyond OpenState?

Platform-agnostic Behavioral Forwarding and Stateful Flow Processing at Wire Speed

Giuseppe Bianchi
Why flow processing?

➡️ Yesterday’s networks:
   ➡️ Increase bandwidth, decrease latency

➡️ Today’s networks:
   ➡️ Increase bandwidth, decrease latency
   ➡️ Add (very) specialized processing for network performance improvements
       ➡️ Load balancing, TCP acceleration, traffic policing, etc
   ➡️ Improve network security
       ➡️ Traffic classification and filtering, anomaly detection, DDoS detection, pattern/signature recognition, etc

Platform-agnostic Behavioral Forwarding and Stateful Flow Processing at Wire Speed

Giuseppe Bianchi
Why programmable flow processing?

➡️ Yesterday’s networks:
   ➡️ Increase bandwidth, decrease latency

➡️ Today’s networks:
   ➡️ Increase bandwidth, decrease latency
   ➡️ Add (very) specialized processing for network performance improvements
       ➡️ Load balancing, TCP acceleration, traffic policing, etc
   ➡️ Improve network security
       ➡️ Traffic classification and filtering, anomaly detection, DDoS detection, pattern/signature recognition, etc

➡️ Flexible approaches/techniques for different needs!

Platform-agnostic Behavioral Forwarding and Stateful Flow Processing at Wire Speed

Giuseppe Bianchi
Why flow processing at wire speed?

→ **Softwarisation does NOT necessarily imply x86**
  → General purpose _network/flow_ processors (domain-specific) can better fit high rate processing and retain VNF goals (repurposability, on-demand placement, etc)
    → General purpose, but domain specific, HW is still 2-3 orders of magnitude better than SW (think to TCAM’s wildcard matching)

→ **Packet-level processing needs**
  → ... if (pck==SYN) then forward remaining ...
  → ... if (# pck > X) then ...
  → ... if pck_fragment #1 followed by frag...
  → ... set (forwarding table = MAC_SRC → in-port)

→ **Flexibility @ packet-level time scale** → VERY hard!
  → @ 100 Gbps link: 64B packet = 5.12 nanoseconds
  → Hard (euphemism) to rely on CPU → confined to slow path

Platform-agnostic Behavioral Forwarding and Stateful Flow Processing at Wire Speed

——— Giuseppe Bianchi ————
Where is the problem?
Plenty of network processors...

Yes, but... ALL proprietary:
- proprietary programming model and languages (APIs)
- Flexibility up to the vendor

Yes but... what about P4? More later!
Platform-agnostic Behavioral Forwarding and Stateful Flow Processing at Wire Speed

Giuseppe Bianchi
Platform agnostic HW configuration: SDN foundation

Traditional networking

Software-Defined Networking

API to the data plane (e.g., OpenFlow)

Programmable switch

smart, slow, (logically) centralized

dumb, fast

Problem!! As of now, API to the data plane (OpenFlow) only for «static» forwarding rules
What we’d like to do?

OpenFlow / SDN

SMART!
Control-plane

OpenFlow switch

Data-plane
DUMB!

Our view / SDN

Central control ➔ still decides how switches shall «behave»

SMART!
Control-plane

Extended switch

Data-plane

Forwarding rules ➡️ Forwarding behavior:

Smart switches ➔ can dynamically update flow tables at wire speed

 SMART!

Forwarding rules AND how they should change or adapt to «events»

Platform-agnostic Behavioral Forwarding and Stateful Flow Processing at Wire Speed

Giuseppe Bianchi
What we’d like to do?

OpenFlow / SDN

Our view / SDN

Behavioral Forwarding in a nutshell:
Dynamic forwarding rules/states \(\rightarrow\)
some control tasks back (!) into the switch

(hard part: via platform-agnostic abstractions)

Data-plane

DUMB!

Data-plane

SMART!

Smart switches \(\rightarrow\) can dynamically update flow tables at wire speed

Platform-agnostic Behavioral Forwarding and Stateful Flow Processing at Wire Speed

Giuseppe Bianchi
Outline

- Motivation and goal

- Background and state of the art

- OpenState: beyond OpenFlow

- Beyond OpenState?

Platform-agnostic Behavioral Forwarding and Stateful Flow Processing at Wire Speed

Giuseppe Bianchi
The emergence of Software Defined Networking

➜ Traditional concern: how to configure and manage multi-vendor networks
   ➜ Configuration interfaces not only differ for different vendors/devices but even vary across different firmware versions of same device!

➜ Before 2008: many ideas/proposals, no real world impact

➜ 2008 breakthrough: OpenFlow
   ➜ First platform-agnostic API to the switch HW – key: pragmatic compromise!
   ➜ “Before OpenFlow, the ideas underlying SDN faced a tension between the vision of fully programmable networks and pragmatism that would enable real-world deployment. OpenFlow struck a balance between these two goals by enabling more functions than earlier route controllers and building on existing switch hardware, through the increasing use of merchant-silicon chipsets in commodity switches”. [quote from Feamster, Rexford, Zegura, 2014]

➜ Since then:
   ➜ Excellent work on network-wide programming abstractions & controllers
   ➜ Excellent work on “softwarization” (virtualization) of network functions
   ➜ But very limited (significant) work on beyond OpenFlow HW abstractions
       ➜ At least up to 2013/14; Then POF, P4, our OpenState… more later

——— Giuseppe Bianchi
SDN key concept

Lots of advantages.... simpler centralized control, immediate deployment, cheaper HW, uniform interface to the switch, high level abstractions and programming models, etc, etc etc

--- Giuseppe Bianchi ---
SDN enabler: the OpenFlow compromise
[original quotes: from OF 2008 paper]

➡ Best approach: “persuade commercial name-brand equipment vendors to provide an open, programmable, virtualized platform on their switches and routers”

➡ Plainly speaking: open the box!! No way...

➡ Viable approach: “compromise on generality and seek a degree of switch flexibility that is

➡ High performance and low cost

➡ Capable of supporting a broad range of research

➡ Consistent with vendors’ need for closed platforms.
OpenFlow match/action abstraction

Programmable logic

Matching Rule

Vendor-implemented

Action

1. FORWARD TO PORT
2. ENCAPSULATE&FORWARD
3. DROP
4. ...

Extensible

Pre-implemented matching engine

Switch Port  MAC src  MAC dst  Eth type  VLAN ID  IP Src  IP Dst  IP Prot  TCP sport  TCP dport

Giuseppe Bianchi
### Example

<table>
<thead>
<tr>
<th>Description</th>
<th>Port</th>
<th>MAC src</th>
<th>MAC dst</th>
<th>Eth type</th>
<th>VLAN ID</th>
<th>IP Src</th>
<th>IP Dest</th>
<th>TCP sport</th>
<th>TCP dport</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 switching</td>
<td>*</td>
<td>*</td>
<td>00:1f..</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Port6</td>
</tr>
<tr>
<td>L3 routing</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>5.6.<em>.</em></td>
<td>*</td>
<td>Port6</td>
</tr>
<tr>
<td>Micro-flow handling</td>
<td>3</td>
<td>00:20..</td>
<td>00:1f..</td>
<td>0x80</td>
<td>Vlan1</td>
<td>1.2.3.</td>
<td>5.6.7.</td>
<td>4</td>
<td>17264</td>
<td>Port4</td>
</tr>
<tr>
<td>Firewall</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>22</td>
</tr>
<tr>
<td>VLAN switching</td>
<td>*</td>
<td>*</td>
<td>00:1f..</td>
<td>Vlan1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>Port6, port7, port8</td>
</tr>
</tbody>
</table>

---

Readily implemented in legacy TCAM
Ternary Content Addressable Memory

---

Giuseppe Bianchi
How to populate flow states?

Packet F1 -> No match

Giuseppe Bianchi
How to populate flow states?

Take «smart» decision

Controller

Packet F1

Set flow state (flow-mod command)

Encapsulate & forward
How to populate flow states?
Centralization: not a panacea!

→ Central view of the network
  → Network as a “whole”
  → Network states
  → Multi-node coordination

→ Signalling & latency!
  \[ \Rightarrow O(100 \text{ ms}) \]
  → 100ms = 20M packets lost @ 100 gbps

Great idea for network-wide states and «big picture» decisions

Poor idea for local states/decision, (way!) better handled locally (less delay, less load)

A considered «solution»: proactive flow states: pre-populate flow tables. Yes, but how to?!

--------- Giuseppe Bianchi ---------------------------------------------
Distributed controllers
the «common» way to address such cons

Proprietary controller extensions?  
Back to Babel?

A non-solution!  
still slow path latency!!

«true» fast path solution: update forwarding rules in 1 packet time – 5 ns @ 64B x 100 Gbps

3 ns = 60cm signal propagation…
OpenFlow evolutions

→ Pipelined tables from v1.1
  ⇒ Overcomes TCAM size limitation
  ⇒ Multiple matches natural
  ⇒ Ingress/egress, ACL, sequential L2/L3 match, etc.

→ Extension of matching capabilities
  ⇒ More header fields
  ⇒ POF (Huawei, 2013): complete matching flexibility!

→ OpenFlow «patches» for (very!) specific processing needs and states
  ⇒ Group tables, meters, synchronized tables, bundles, typed tables (sic!), etc
  ⇒ Not nearly clean, hardly a «first principle» design strategy
  ⇒ A sign of OpenFlow structural limitations?

----- Giuseppe Bianchi
Programming the data plane: The P4 initiative

  - Dramatic flexibility improvements in packet processing pipeline
    - Configurable packet parser → parse graph
    - Target platform independence → compiler maps onto switch details
    - Reconfigurability → change match/process fields during pipeline

- Feasible with HW advances
  - Reconfigurable Match Tables, SIGCOMM 2013
  - Intel’s FlexPipe™ architectures

- P4.org: Languages and compilers
  - Further support for «registry arrays» and counters meant to persist across multiple packets
  - Though no HW details, yet

--- Giuseppe Bianchi
Programming the data plane: The P4 initiative

  - Dramatic flexibility improvements in packet processing pipeline
  - Configurable packet parser
  - Parse graph
  - Target platform independence
  - Compiler maps onto switch details
  - Reconfigurability
    - change match/process fields during pipeline
  - Feasible with HW advances
    - Reconfigurable Match Tables, SIGCOMM 2013
    - Intel's FlexPipe™ architectures
  - P4.org: Languages and compilers
    - Further support for «registry arrays» and counters meant to persist across multiple packets
    - Though no HW details, yet

OpenFlow 2.0 proposal

- Stateful processing, but only «inside» a packet processing pipeline!

- Not yet (clear) support for stateful processing «across» subsequent packets in the flow
Outline

➔ Motivation and goal

➔ Background and state of the art

➔ OpenState: beyond OpenFlow

➔ Beyond OpenState?

Platform-agnostic Behavioral Forwarding and Stateful Flow Processing at Wire Speed

Giuseppe Bianchi
Our group, SIGCOMM CCR 2014; surprising finding: an OpenFlow switch can «already» support stateful evolution of the forwarding rules

With almost marginal (!) architecture modification

OpenFlow / SDN

SMART!
Control-plane

Forwarding rules

OpenFlow switch

DUMB!

Data-plane

OpenState / SDN

Central control still decides how switches shall «behave»

Forwarding behavior:
Forwarding rules AND how they should change or adapt to «events»

Smart switches can dynamically update flow tables at wire speed

SMART!
Control-plane

Forwarding rules

OpenState switch

SMART!
Control-plane

Data-plane
Our findings at a glance

- Any control program that can be described by a Mealy (Finite State) Machine is already (!) compliant with OF1.3

- MM + Bidirectional flow state handling requires minimal hardware extensions to OF1.1+


---------- Giuseppe Bianchi ---------------
### Remember OF match/action API

#### Programmable logic

<table>
<thead>
<tr>
<th>Matching Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch Port</td>
</tr>
</tbody>
</table>

#### Vendor-implemented

1. FORWARD TO PORT
2. ENCAPSULATE & FORWARD
3. DROP
4. ...

#### Pre-implemented matching engine

- Extensible

### Matching Rule

- Action
  - 1. FORWARD TO PORT
  - 2. ENCAPSULATE & FORWARD
  - 3. DROP
  - 4. ...

### Programmable logic

- Pre-implemented matching engine
What is the OF abstraction, formally?

Packet header match = “Input Symbol” in a finite set 
$I = \{i_1, i_2, \ldots, i_M\}$. 
⇒ One input symbol = any possible header match  
⇒ Possible matches pre-implemented; cardinality depends on match implementation  
⇒ Theoretically, it is irrelevant how the Input Symbols’ set $I$ is established  
  ⇒ i.e. each input symbol = Cartesian combination of multiple header field matches, further including “wildcard” matches; 
  ⇒ E.s. incoming packet destination port = 5238 AND source IP address is 160.80.82.1, and the VLAN tag is 1111, etc.

OpenFlow actions = “Output Symbols” in finite set 
$O = \{o_1, o_2, \ldots, o_N\}$  
⇒ Pre-implemented actions

OpenFlow’s match/action abstraction: a map $T: I \rightarrow O$  
⇒ all what the third party programmer can specify!

Giuseppe Bianchi
Reinterpreting (and extending) the OpenFlow abstraction

OpenFlow map is trivially recognized to be a very special and trivial case of a Mealy Finite State Machine

\[ T : \{\text{default-state}\} \times I \rightarrow \{\text{default-state}\} \times O, \]

i.e. a Finite State Machine with output, where we only have one single (default) state!

By adding (per-packet) retrieval and update of states, OpenFlow can be turned it into a Mealy machine executor!!
If an application can be «abstracted» in terms of a mealy Machine...

Example: Port Knocking firewall
knock «code»: 5123, 6234, 7345, 8456 → then open Port 22
... it can be transformed in a Flow Table!

MATCH: \(<\text{state}, \text{port}> \rightarrow \text{ACTION}: \langle\text{drop/forward, state_transition}\rangle\)

Plus a state lookup/update
Putting all together

1) State lookup

<table>
<thead>
<tr>
<th>Flow key</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPsrc=1.2.3.4</td>
<td>... ... ...</td>
</tr>
<tr>
<td>IPsrc=5.6.7.8</td>
<td>OPEN</td>
</tr>
<tr>
<td>IPsrc= ... ...</td>
<td>... ... ...</td>
</tr>
<tr>
<td>IPsrc= no match</td>
<td>DEFAULT</td>
</tr>
</tbody>
</table>

2) XFSM state transition

<table>
<thead>
<tr>
<th>Match fields</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>event</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>Port=5123</td>
</tr>
<tr>
<td>STAGE-1</td>
<td>Port=6234</td>
</tr>
<tr>
<td>STAGE-2</td>
<td>Port=7345</td>
</tr>
<tr>
<td>STAGE-3</td>
<td>Port=8456</td>
</tr>
<tr>
<td>OPEN</td>
<td>Port=22</td>
</tr>
<tr>
<td>OPEN</td>
<td>Port=*</td>
</tr>
<tr>
<td>*</td>
<td>Port=*</td>
</tr>
</tbody>
</table>

3) State update

1 «program» XFSM table for all flows
(same knocking sequence)

N states, one per (active) flow
Cross-flow state handling

Yes but what about MAC learning, multi-port protocols (e.g., FTP), reverse path forwarding, bidirectional flow handling, etc?

MACdst | MACsrc
--- | ---

State Table

<table>
<thead>
<tr>
<th>Flow key</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 bit MAC addr</td>
<td>Port #</td>
</tr>
</tbody>
</table>

XFSM Table

<table>
<thead>
<tr>
<th>state</th>
<th>event</th>
<th>action</th>
<th>Next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port#</td>
<td>*</td>
<td>forward</td>
<td>In-port</td>
</tr>
</tbody>
</table>

MACdst | MACsrc
--- | ---

State Table

<table>
<thead>
<tr>
<th>Flow key</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 bit MAC addr</td>
<td>Port #</td>
</tr>
</tbody>
</table>

DIFFERENT lookup/update scope

Field 1 | Field 2 | Field N
--- | --- | ---

Flowkey selector

Read/write signal

---

Giuseppe Bianchi
(HW) OpenState Architecture

HW implementation details:

- First TCAM only for static states (e.g. packets belonging to a given subnet)
- 5 clock (2 SRAM read + 2 TCAM + 1 SRAM write)
- 10 Gbps just requires 156 MHz clock TCAM, trivial

HW implementation Details in HPSR 2015, Pontarelli, Bonola, Bianchi, Capone, Cascone
Outline

→ Motivation and goal

→ Background and state of the art

→ OpenState: beyond OpenFlow

→ Beyond OpenState?

Platform-agnostic Behavioral Forwarding and Stateful Flow Processing at Wire Speed

Giuseppe Bianchi
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?

----- Giuseppe Bianchi -----------------------------------------
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:D→D
    ⇒ Update of the registry values!
  ⇒ T:S×I×F→S×O×U the actual XFSM transition
    ⇒ A map can be implemented by the TCAM!

---

Giuseppe Bianchi
Extended finite state machines: much more general!

→ Mealy Machines: 4-tuple
  ⇒ \( I, O, S \)
  ⇒ \( T : S \times I \rightarrow S \times O \)
→ XFSM: 7-tuple
  ⇒ \( I, O, S \)
  ⇒ \( T : S \times I \times F \rightarrow S \times O \times U \)

Question to CS formal theorists in the room: can an XFSM execute any arbitrary algorithm?

If not, we are very close to: Abstract State Machines (further slight generalization) indeed can [Gurevich]

⇒ Update
⇒ Check Conditions on \( D \)
⇒ Update symbol to \( D \)
⇒ State 2

→ A map can be implemented by the TCAM!

--- Giuseppe Bianchi ---
Towards an Open Flow Processor

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

→ Details to be presented in a follow up presentation
An OFP program = a platform agnostic abstract XFSM!
(example: a TCP SYN scan detection+mitigation)

**DEFAULT**

NEW_TCP_FLOW
<\(D_0 = 0\)>
<\(D_1 = \text{pkt.ts}\)>
[OUTPUT 1]

**DROP**

ANY_PACKET
if \(D_2 > \text{pkt.ts}\)
<\(D_0 = 0\)>
<\(D_1 = \text{pkt.ts}\)>
[DROP]

IDLE_TIMEOUT_EXPIRED
<REMOVE_FLOW_ENTRY>

**MONITOR**

ANY_PACKET
if \(D_2 < \text{pkt.ts}\)
<\(D_0 = 0\)>
<\(D_1 = \text{pkt.ts}\)>
[OUTPUT 1]

NEW_TCP_FLOW
<\(D_0 \geq G_0\)>
<\(D_1 = \text{pkt.ts} + G_1\)>
[DROP]

NEW_TCP_FLOW
if \((\text{\text{\text{NEW_TCP_FLOW}}}_0 < G_0)\)
<\(D_0 = \text{rate}(D_0, D_1)\)>
<\(D_1 = \text{pkt.ts}\)>
[OUTPUT 1]

\(D_0\): TCP SYN rate
\(D_1\): last packet timestamp
\(D_2\): DROP state expiration timestamp

\(G_0\): rate threshold (global)
\(G_1\): DROP duration (global)
An OFP program = a platform agnostic abstract XFSM!
(example: a TCP SYN scan detection+mitigation)

DEFUALT

NEW_TCP_FLOW
\(<D_0 = 0>\)
\(<D_1 = \text{pkt.ts}>\)
[OUTPUT 1]

D_0: TCP SYN rate
D_1: last packet timestamp
D_2: DROP state expiration timestamp

G_0: rate threshold (global)
G_1: DROP duration (global)

IDLE_TIMEOUT_EXPIRED
\(<\text{REMOVE FLOW ENTRY}>\)

ANY_PACKET
if \(D_2 < \text{pkt.ts}\)
\(<D_0 = 0>\)

MONITO

NEW_TCP_FLOW
\(<D_0 = 0>\)
\(<D_1 = \text{pkt.ts}>\)
[OUTPUT 1]

Note: guaranteed to run at wire speed!
(does not rely on any CPU: it is directly «executed» by the architecture, with the TCAM performing state transitions!

Giuseppe Bianchi
Conclusions

Platform-agnostic programming of control intelligence inside devices’ fast path (!) seems not only possible but even viable

- Viable on today’s HW (implemented!)
- TCAM as «State Machine processor»
- Support for full XFSM, beyond OpenState’s Mealy Machine

Rethinking control-data plane SDN separation?

- Back to active networking 2.0? (but now with a clearcut abstraction in mind)