Stateful OpenFlow: Hardware Proof of Concept

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

HPSR ’15

03/07/2015 - Budapest, Hungary

Introduction and background

Stateful data plane and OpenState

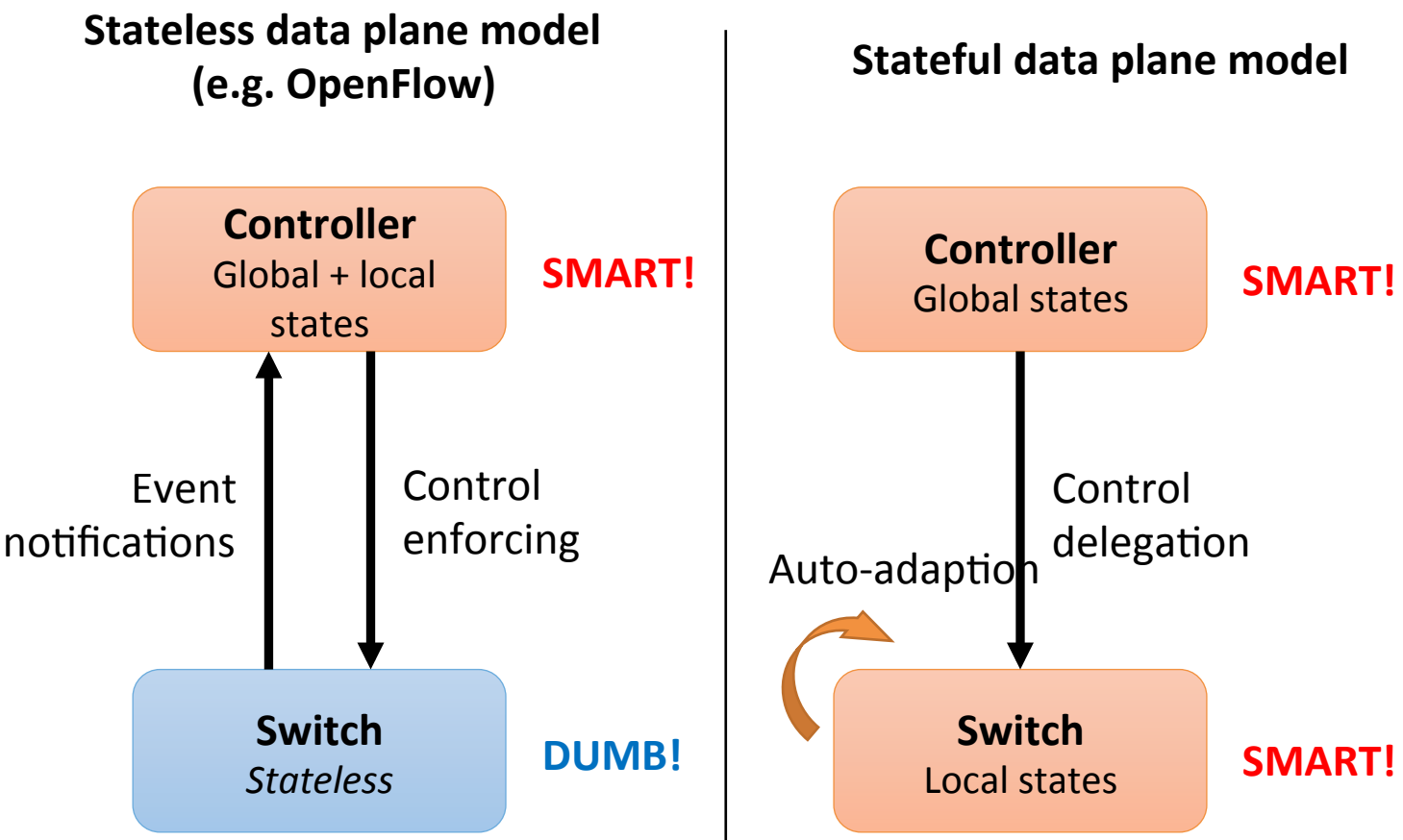
OpenState background

- This presentation describes an FPGA based proof of concept implementation of **OpenState**, a work originally published in ACM CCR (2014)
- 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***
  - Userspace switch, controller and experimenter specification

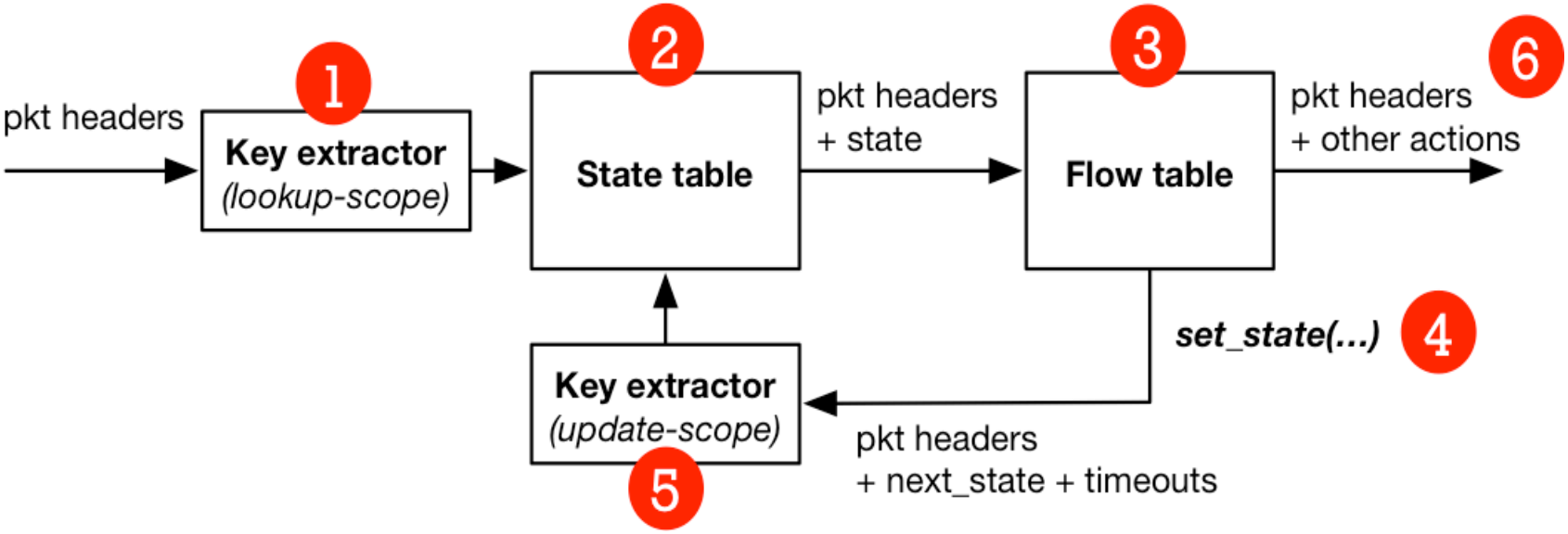
Motivations

- 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***

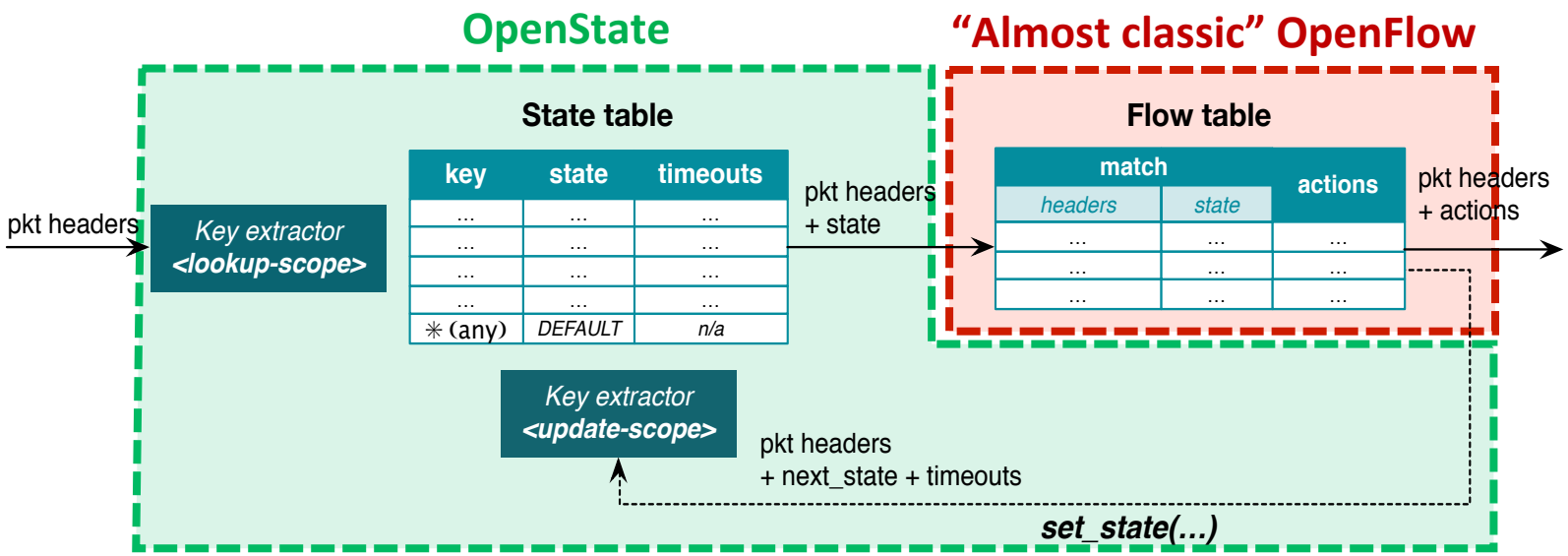
Stateless vs. Stateful in SDN



OpenState workflow



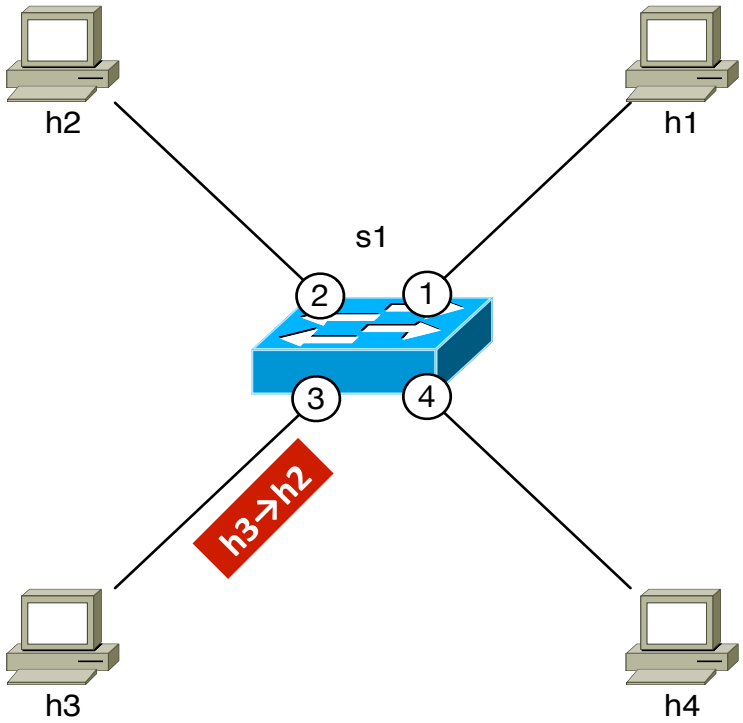
OpenState architecture



A simple use case

MAC learning with OpenState

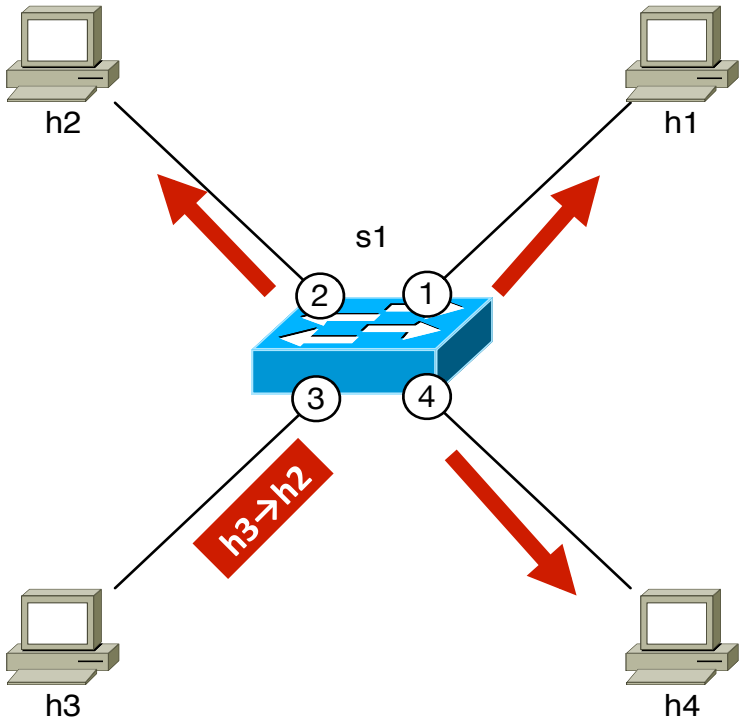
Simple use case: MAC learning



MAC Addr	Port
h3	3

Learn input port for each received frame

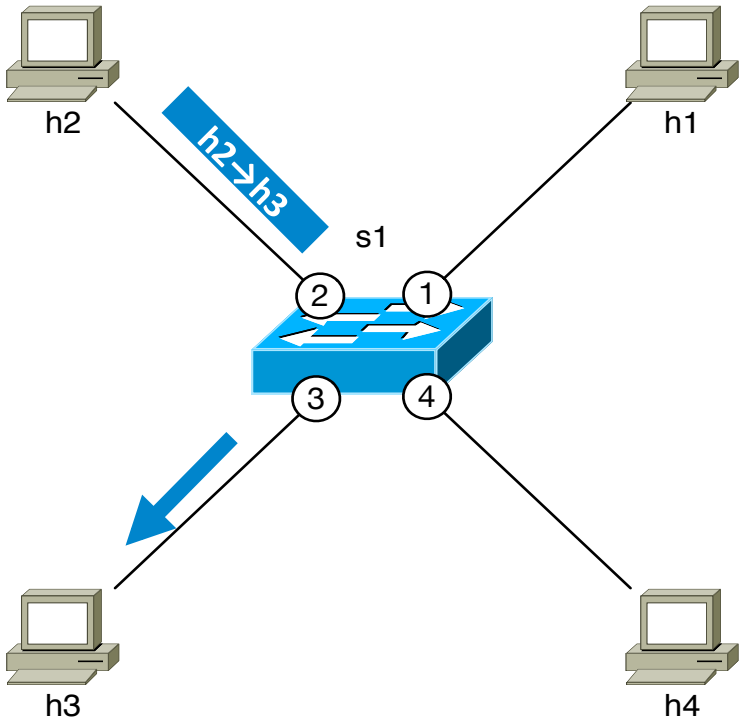
Simple use case: MAC learning



MAC Addr	Port
h3	3

Flood when destination unknown

Simple use case: MAC learning



MAC Addr	Port
h3	3
h2	2

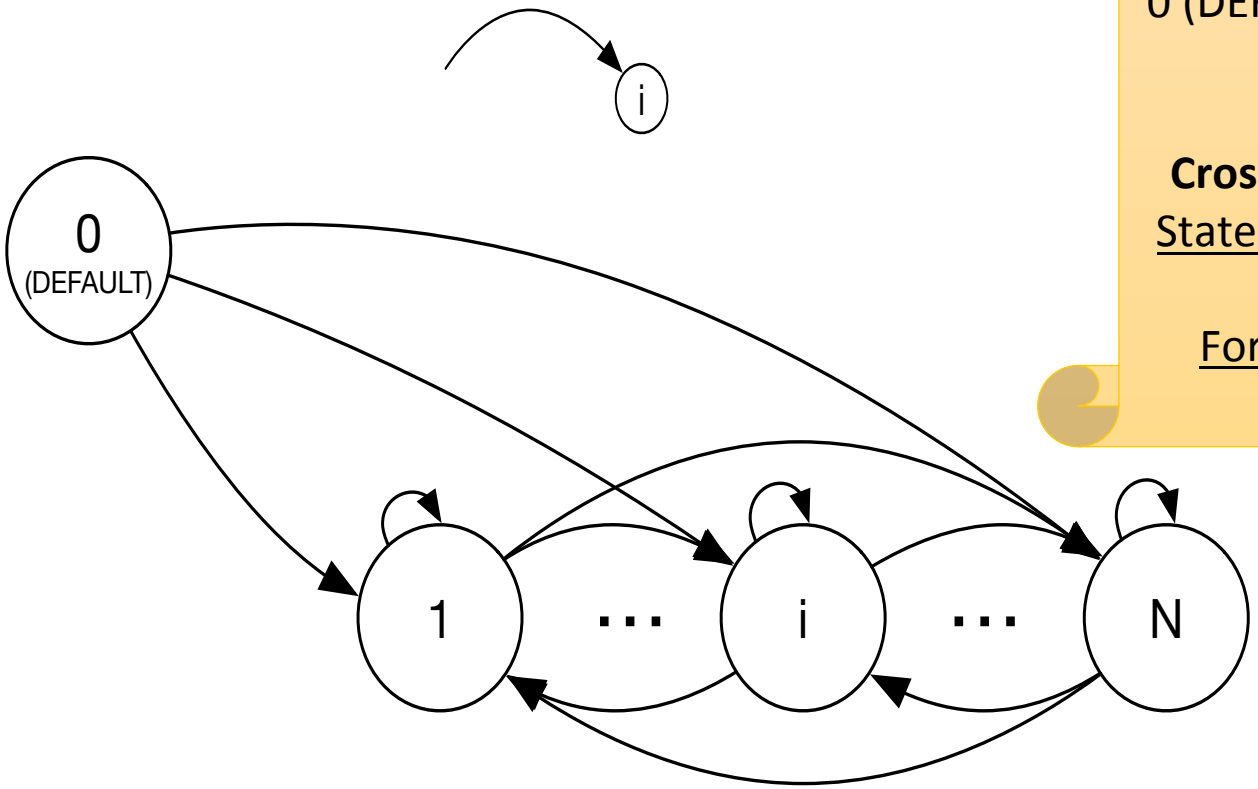
“Response” packets are unicasted to h3

MAC learning

Mealy machine

in_port=i

Forw*ard(**state[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 = **{eth_dst}**

Update-scope = **{eth_src}**

Priority	Match	Actions
0	in_port=1, state=0	set_state(1, 0), flood()
0	in_port=1, state=1	set_state(1, 0), output(1)
0	in_port=1, state=2	set_state(1, 0), output(2)
	...	
0	in_port=2, state=0	set_state(2, 0), flood()
0	in_port=2, state=1	set_state(2, 0), output(1)
	...	
0	in_port=N, state=N	set_state(N, 0), output(N)

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

….

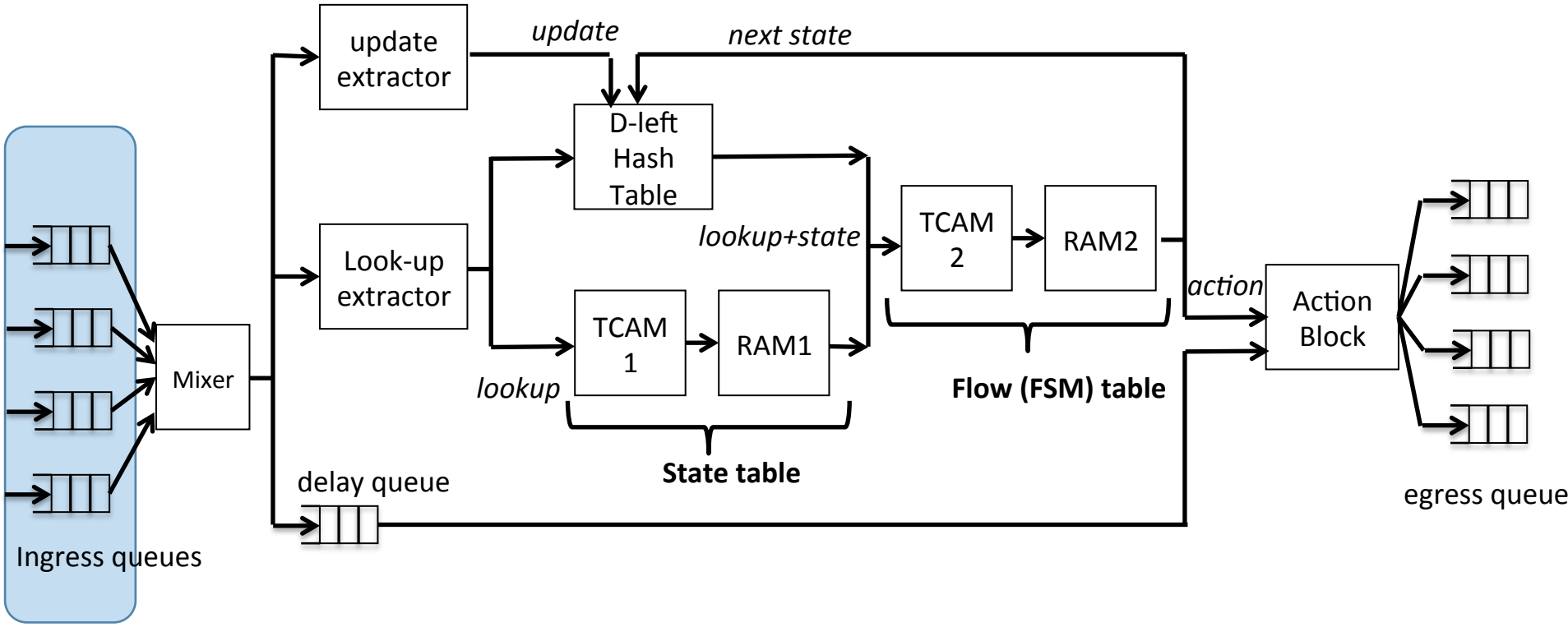
FPGA prototype

HW proof of concept implementation of OpenState

FPGA prototype

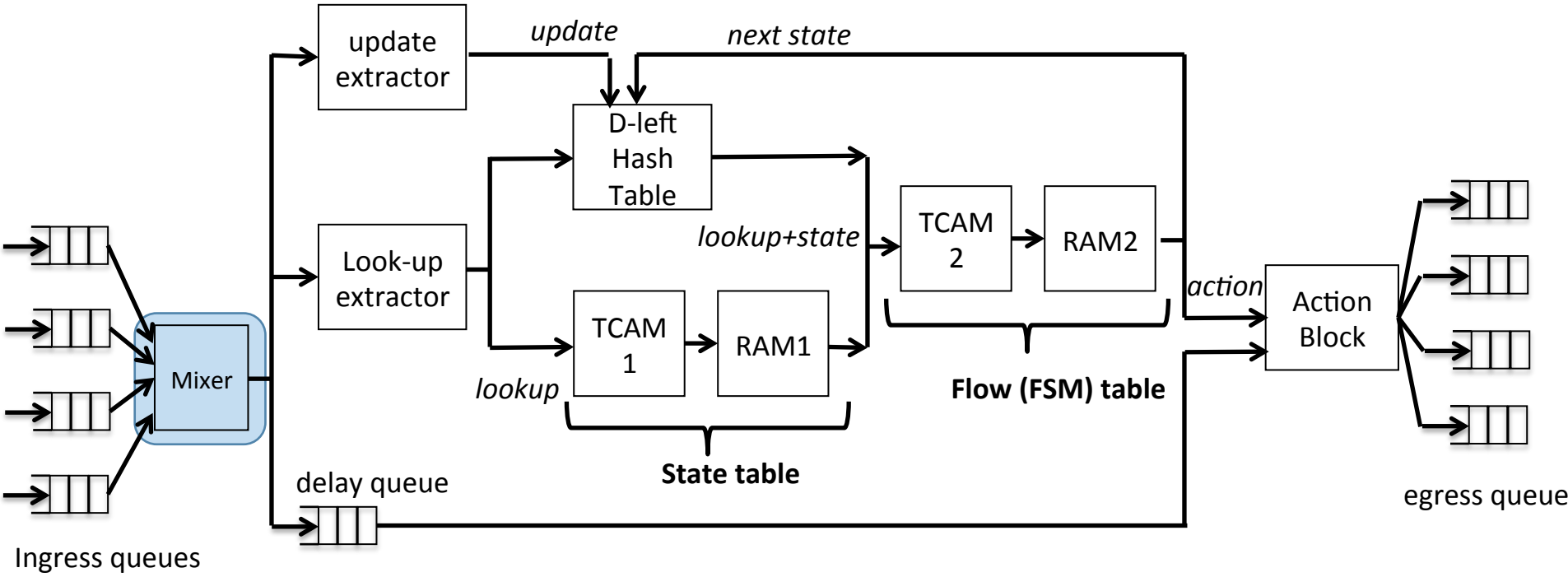
- The OpenState hardware prototype has been designed using as target development board the INVEA COMBO-LXT, an express PCI x8 mother card equipped with the XILINX Virtex5 FPGA
- The prototype exploits the two 10 GbE interfaces of the board, and the PCI express bus to configure the development board as 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



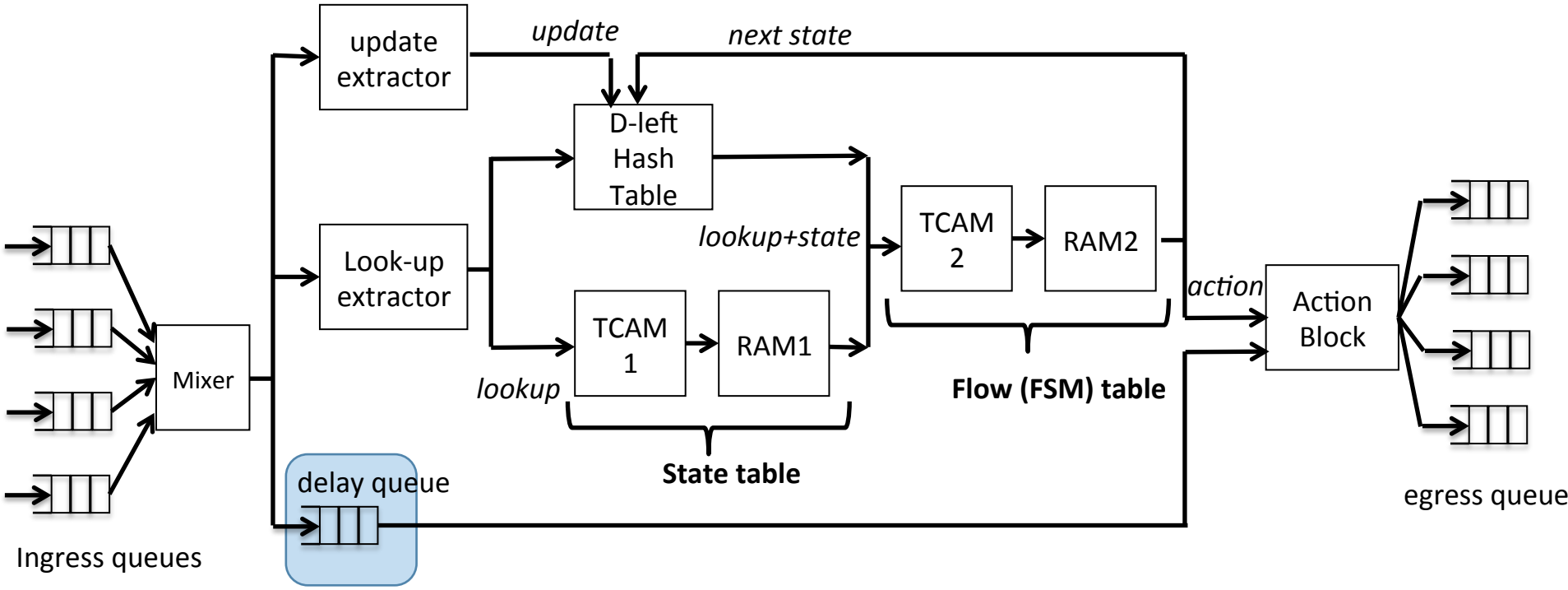
Four ingress queues collect the packets coming from the ingress ports

Prototype architecture



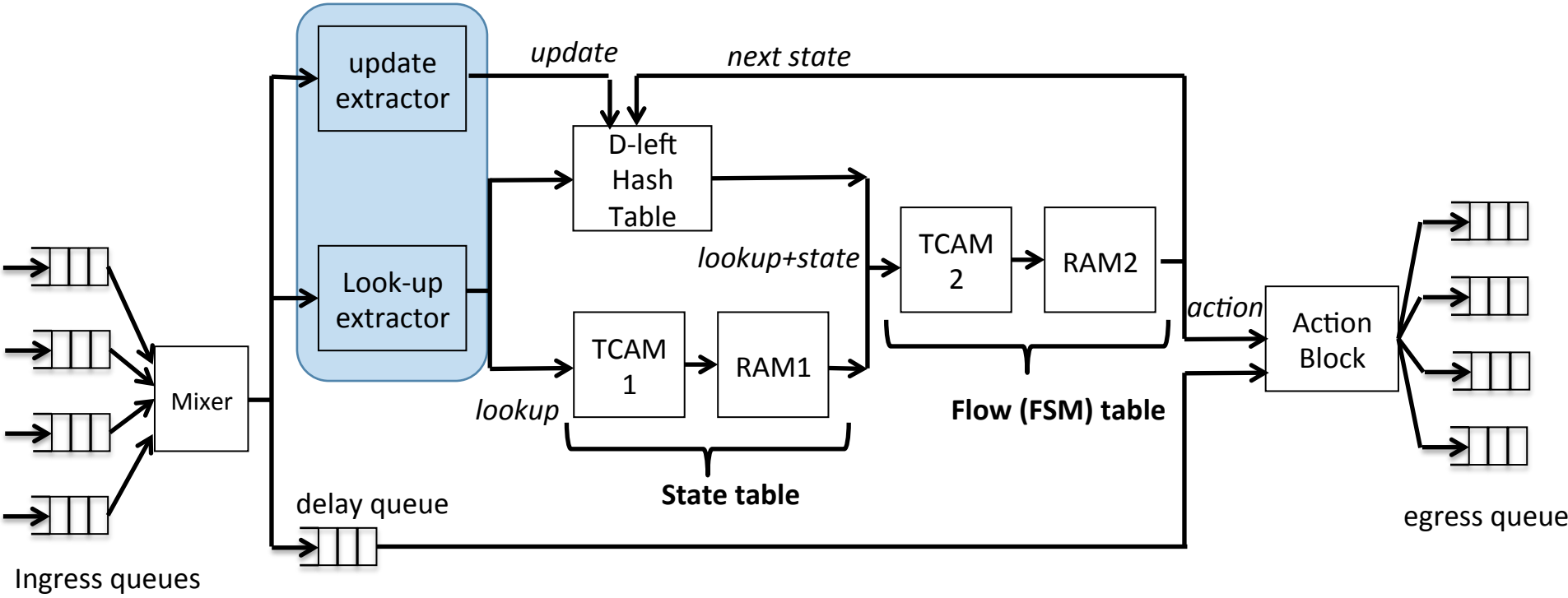
A 4-input 1-output mixer block aggregates the packets using a round robin policy. The output of the mixer is a 320 bits data bus able to provide an overall throughput of 50 Gbps

Prototype architecture



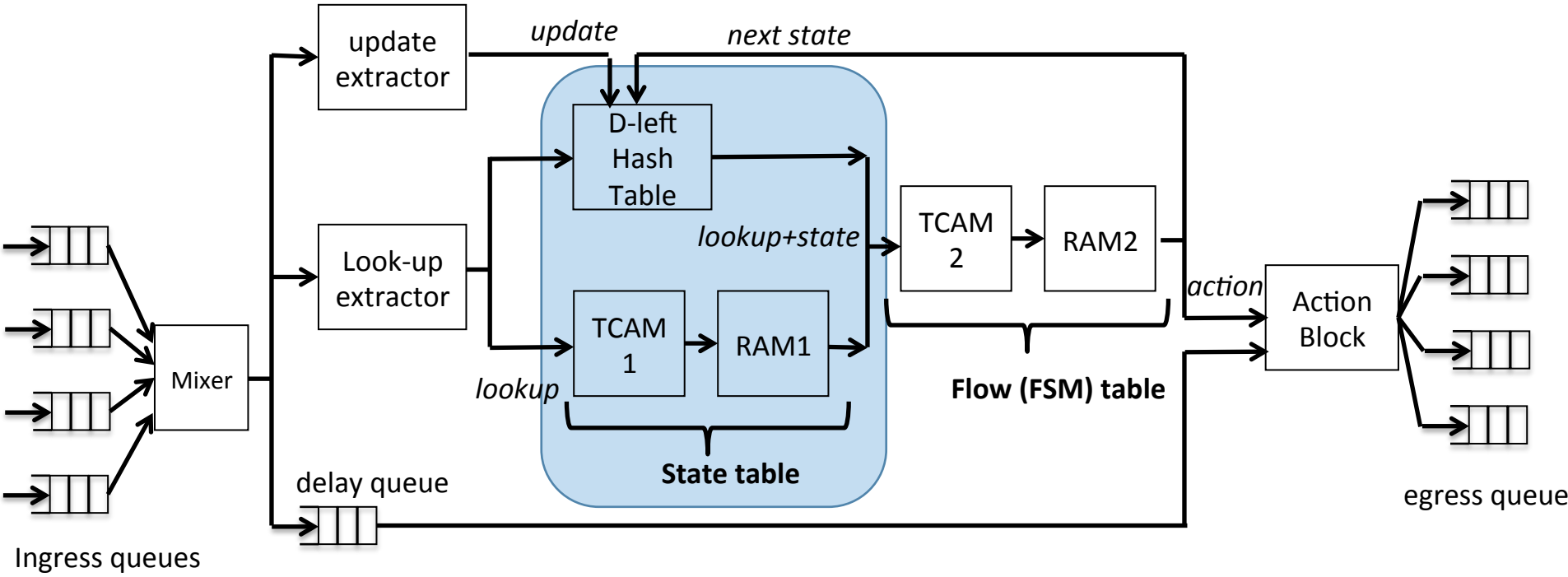
A delay queue stores the packet during the time need by the OpenState tables to operate

Prototype architecture



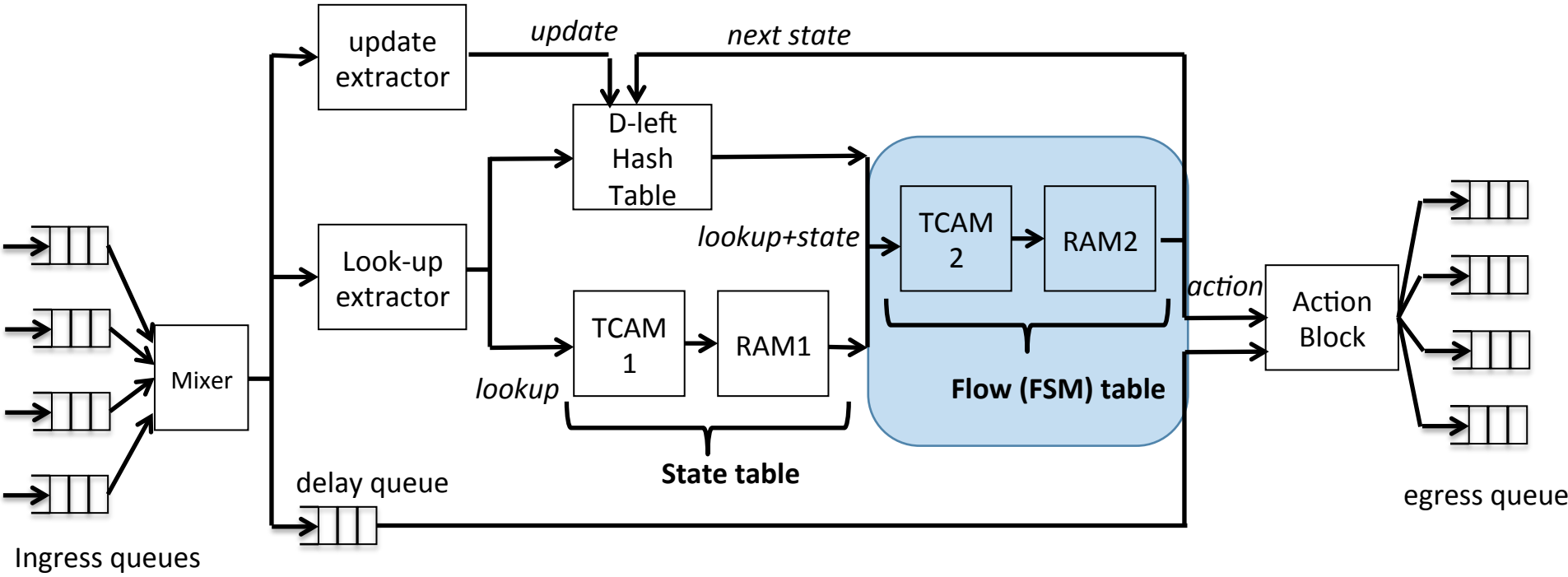
The look-up and update extractor blocks that build the keys that are used to read/update the state table. The 128 bit output is given as input to the state lookup and update

Prototype architecture



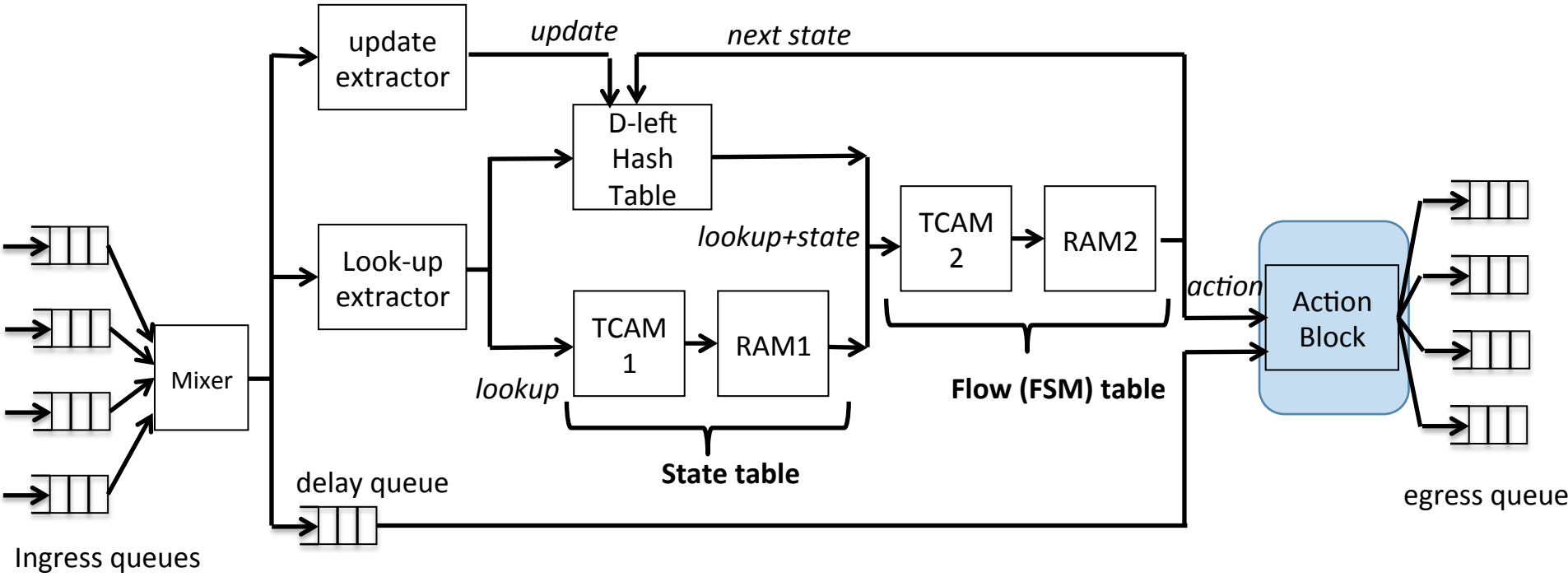
The state table is realized by the d-left hash table (4k entries, MHT without moving capability) and a small TCAM (32 entries * 128 bits) and a companion SRAM (configured as dual port RAM)

Prototype architecture



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)

Prototype architecture



The action block applies the selected actions and forward the packet to the output queues

Performance

Performance along with 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)

Synthesis and simulation

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

Conclusions

- We have presented a proof of concept HW implementation of OpenState showing the feasibility of the proposed stateful approach
- We showed that the OpenState extension can be easily developed reusing the same building blocks of a standard OpenFlow implementation along with simple combinatorial hardware blocks
- Since flow states can be easily stored in a d-left hash table, the scalability of the system is related to the maximum size of the SRAM memory (more than 2 millions of flows can be stored in a 32 MB embedded SRAM)
- The number of clock cycles required to update a state can be limited to few clock cycles

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***