End-to-end network programmability

From core switches to end hosts

Gregorio Procissi gregorio.procissi@unipi.it
Giuseppe Lettieri giuseppe.lettieri@unipi.it

Agenda

Network programmability all over the network

- Part I (Gregorio Procissi)
  - From original SDN to programmable network data-plane
  - In-network computing: programmable switches and the P4 language
  - P4 in practice: running a programmable software switch in an emulated network environment

- Part II (Giuseppe Lettieri)
  - End-host computing
    - in-kernel networking with extended Berkeley Packet Filter (eBPF)
  - eBPF in practice
    - programming simple applications on a Linux machine

Part I

Gregorio Procissi

In-network data plane programmability

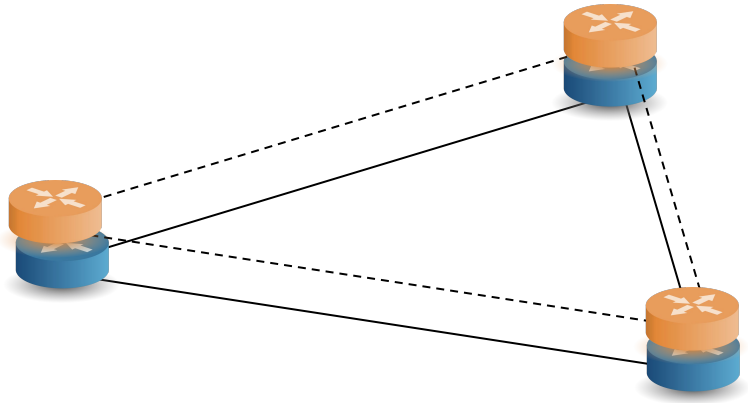
Data Plane/Control Plane...

- Data Plane (Forwarding plane)
  - Processing of data packets, in particular forwarding

- Control plane
  - Intelligence of the network, defines how to handle packets
  - example: *routing*

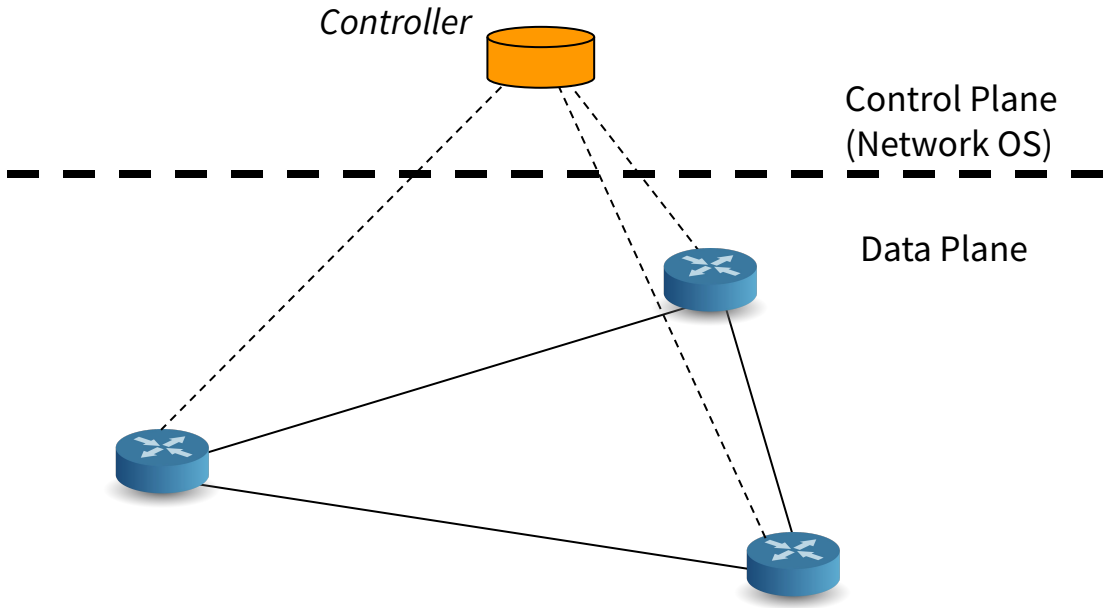
… In traditional networks

- Data and control planes vertically integrated
  - in every network node
  - control plane distributed across nodes (routers)
- ***Destination based forwarding***



… and in Software Define Networks

- Data plane and control plane are logically separated



Software Defined Networking (SDN)

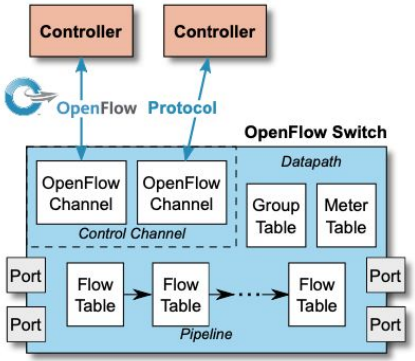
- Flow based forwarding
  - **flow**: set of header fields values used as **matching rules**
  - **actions**: to be applied to packets belonging to the same flow

- Extended concept of forwarding node:
  - a switch may be **instructed** to be a router, firewall, load-balancer, and so on

- At this stage network applications run on top of the SDN controller
  - just as processes run on top of operating systems
  - programmability is quite classic
  - does it **scale** at high traffic rates...?

Openflow

- The first successful incarnation of the SDN paradigm, from Stanford University
- from OF paper (2008): "*amenable to high performance and low-cost implementation; capable of supporting a broad range of research; and consistent with vendors' need for closed platforms*"
- Essentially, OpenFlow is an **API** to a switch flow table



OpenFlow tables (examples)

L2 switch

Switch Port	Mac src	Mac dst	Eth type	IP src	IP dst	TCP s_port	TCP d_port	Action
*	*	68:5b:35:83:c8:56	*	*	*	*	*	Forward to Port P2

L3 forwarding

Switch Port	Mac src	Mac dst	Eth type	IP src	IP dst	TCP s_port	TCP d_port	Action
*	*	*	0x0800	*	131.114.52.*	*	*	Forward to Port P3

L4 Firewall

Switch Port	Mac src	Mac dst	Eth type	IP src	IP dst	TCP s_port	TCP d_port	Action
*	*	*	0x0800	131.114.52.*	*	*	6969	Drop

Match-Action Tables

- Match-Action Table
  - Match:
    - binary match (0,1)
    - ternary match (0,1,*)
  - Action:
    - drop, forward, modify, go to another table

Switch Port	Mac src	Mac dst	Eth type	IP src	IP dst	TCP s_port	TCP d_port	Action
*	*	68:5b:35:83:c8:56	*	*	*	*	*	Forward to Port P2

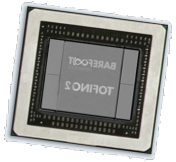
OpenFlow Switching platforms

- Commodity CPU: software switch (OVS, OFSoftSwitch, etc…)

- FPGA, NetFPGA

- Network Processor (NPU)

- Switching chips
  - ~ 100x faster than CPUs
  - ~ 10x faster than NPU



Is it really data plane programming?

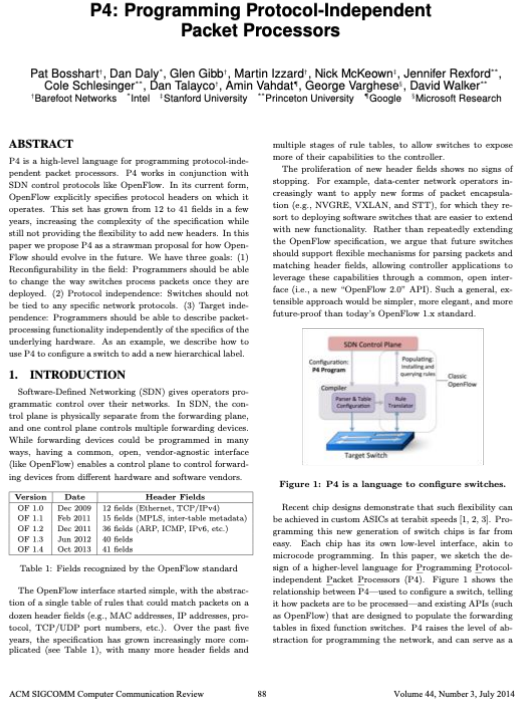
- To add *expressiveness* the protocol has become more and more **complex**
  - 12 fields match on OF 1.0 ---> 41 in OF 1.4
- **Fixed** type tables
- **No custom actions** upon matching
  - only a set of “pre-cooked” actions
- No room for defining custom **state variables** (across different packets)

```
enum oxm_ofb_match_fields {
  OFPXMT_OFB_IN_PORT,
  OFPXMT_OFB_IN_PHY_PORT,
  OFPXMT_OFB_METADATA,
  OFPXMT_OFB_ETH_DST ,
  OFPXMT_OFB_ETH_SRC ,
  OFPXMT_OFB_ETH_TYPE ,
  OFPXMT_OFB_VLAN_VID ,
  OFPXMT_OFB_VLAN_PCP ,
  OFPXMT_OFB_IP_DSCP ,
  OFPXMT_OFB_IP_ECN ,
  OFPXMT_OFB_IP_PROTO ,
  OFPXMT_OFB_IPV4_SRC ,
  OFPXMT_OFB_IPV4_DST ,
  OFPXMT_OFB_TCP_SRC ,
  OFPXMT_OFB_TCP_DST ,
  OFPXMT_OFB_UDP_SRC ,
  OFPXMT_OFB_UDP_DST ,
  OFPXMT_OFB_SCTP_SRC ,
  OFPXMT_OFB_SCTP_DST ,
  OFPXMT_OFB_ICMPV4_TYPE ,
  OFPXMT_OFB_ICMPV4_CODE ,
  OFPXMT_OFB_ARP_OP ,
  OFPXMT_OFB_ARP_SPA ,
  OFPXMT_OFB_ARP_TPA ,
  OFPXMT_OFB_ARP_SHA ,
  OFPXMT_OFB_ARP_THA ,
  OFPXMT_OFB_IPV6_SRC ,
  OFPXMT_OFB_IPV6_DST ,
  OFPXMT_OFB_IPV6_FLABEL ,
  OFPXMT_OFB_ICMPV6_TYPE ,
  OFPXMT_OFB_ICMPV6_CODE ,
  OFPXMT_OFB_IPV6_ND_TARGET,
  OFPXMT_OFB_IPV6_ND_SLL,
  OFPXMT_OFB_IPV6_ND_TLL,
  OFPXMT_OFB_MPLS_LABEL,
  OFPXMT_OFB_MPLS_TC,
  OFPXMT_OFP_MPLS_BOS,
  OFPXMT_OFB_PBB_ISID,
  OFPXMT_OFB_TUNNEL_ID,
  OFPXMT_OFB_IPV6_EXTHDR,
  OFPXMT_OFB_PBB_UCA
};
```

```
enum ofp_action_type {
  OFPAT_OUTPUT,
  OFPAT_COPY_TTL_OUT,
  OFPAT_COPY_TTL_IN,
  OFPAT_SET_MPLS_TTL,
  OFPAT_DEC_MPLS_TTL,
  OFPAT_PUSH_VLAN,
  OFPAT_POP_VLAN,
  OFPAT_PUSH_MPLS,
  OFPAT_POP_MPLS,
  OFPAT_SET_QUEUE,
  OFPAT_GROUP,
  OFPAT_SET_NW_TTL,
  OFPAT_DEC_NW_TTL,
  OFPAT_SET_FIELD,
  OFPAT_PUSH_PBB,
  OFPAT_POP_PBB,
  OFPAT_EXPERIMENTER
};
```

Deeper network programmability

- Why not defining flexible parsers to match arbitrary header fields?
- Why not defining custom tables for matching rules?
- Why not implementing custom actions to be applied upon matching rules?
- Why not supporting custom stateful operations?
- The answer… again came from Stanford University… and is… **Programming Protocol-independent Packet Processors + PISA**



P4: Programming Protocol-Independent Packet Processors

Pat Bosshart[†], Dan Daly[*], Glen Gibb[†], Martin Izzard[†], Nick McKeown[†], Jennifer Rexford[**], Cole Schlesinger[**], Dan Talayco[†], Amin Vahdat[*], George Varghese[‡], David Walker[**]
[†]Barefoot Networks [*]Intel [‡]Stanford University [**]Princeton University [*]Google [‡]Microsoft Research

ABSTRACT

P4 is a high-level language for programming protocol-independent packet processors. P4 works in conjunction with SDN control protocols like OpenFlow. In its current form, OpenFlow explicitly specifies protocol headers on which it operates. This set has grown from 12 to 41 fields in a few years, increasing the complexity of the specification while still not providing the flexibility to add new headers. In this paper we propose P4 as a strawman proposal for how OpenFlow should evolve in the future. We have three goals: (1) Reconfigurability in the field: Programmers should be able to change the way switches process packets once they are deployed. (2) Protocol independence: Switches should not be tied to any specific network protocols. (3) Target independence: Programmers should be able to describe packet-processing functionality independently of the specifics of the underlying hardware. As an example, we describe how to use P4 to configure a switch to add a new hierarchical label.

1. INTRODUCTION

Software-Defined Networking (SDN) gives operators programmatic control over their networks. In SDN, the control plane is physically separate from the forwarding plane, and one control plane controls multiple forwarding devices. While forwarding devices could be programmed in many ways, having a common, open, vendor-agnostic interface (like OpenFlow) enables a control plane to control forwarding devices from different hardware and software vendors.

Version	Date	Header Fields
OF 1.0	Dec 2009	12 fields (Ethernet, TCP/IPv4)
OF 1.1	Feb 2011	15 fields (MPLS, inter-table metadata)
OF 1.2	Dec 2011	36 fields (ARP, ICMP, IPv6, etc.)
OF 1.3	Jun 2012	40 fields
OF 1.4	Oct 2013	41 fields

Table 1: Fields recognized by the OpenFlow standard

The OpenFlow interface started simple, with the abstraction of a single table of rules that could match packets on a dozen header fields (e.g., MAC addresses, IP addresses, protocol, TCP/UDP port numbers, etc.). Over the past five years, the specification has grown increasingly more complicated (see Table 1), with many more header fields and multiple stages of rule tables, to allow switches to expose more of their capabilities to the controller.

The proliferation of new header fields shows no signs of stopping. For example, data-center network operators increasingly want to apply new forms of packet encapsulation (e.g., NVGRE, VXLAN, and STT), for which they resort to deploying software switches that are easier to extend with new functionality. Rather than repeatedly extending the OpenFlow specification, we argue that future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to leverage these capabilities through a common, open interface (i.e., a new "OpenFlow 2.0" API). Such a general, extensible approach would be simpler, more elegant, and more future-proof than today's OpenFlow 1.x standard.

Figure 1: P4 is a language to configure switches.

Recent chip designs demonstrate that such flexibility can be achieved in custom ASICs at Tbit speeds [1, 2, 3]. Programming this new generation of switch chips is far from easy. Each chip has its own low-level interface, akin to microcode programming. In this paper, we sketch the design of a higher-level language for Programming Protocol-independent Packet Processors (P4). Figure 1 shows the relationship between P4—used to configure a switch, telling it how packets are to be processed—and existing APIs (such as OpenFlow) that are designed to populate the forwarding tables in fixed function switches. P4 raises the level of abstraction for programming the network, and can serve as a

Reconfigurable Match Tables (RMTs)

- **Reconfigurable Match Table** (RMT)
  - Set of pipelined stages each with a match table of **arbitrary depth** and **width**
    - ex: IP match: 32-bit depth, 256k width
    - ex: Ethernet: 48-bit addresses width, 64k width
- RMT introduces reconfigurability:
  - field definitions *can be altered* and *new fields* added
  - the number, topology, widths, and depths of match tables *can be specified*, subject only to an overall resource limit on the number of matched bits
  - *new actions may be defined*, such as writing new header fields
  - arbitrarily modified packets can be placed in specified queue(s), for output at any subset of ports, with a queuing discipline specified for each queue
- Configuration managed by an SDN controller

Mac src	Mac dst	Eth type	Action
*	68:5b:35:83:c8:56	*	Goto Table 2
...	...	...	...
*	Aa:bb:00:cc:00:aa	*	Goto Table 2

Table 1

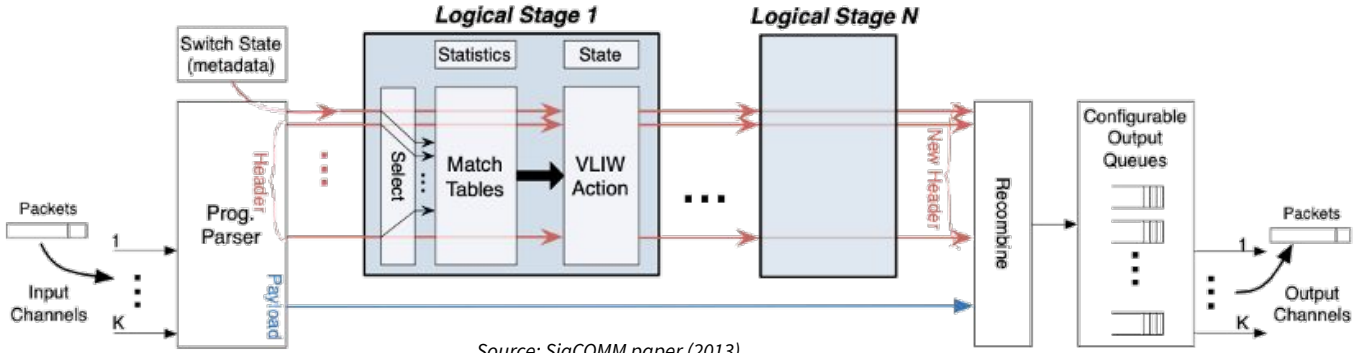
IP src	IP dst	Action
*	*	Forward to Port P2
*	131.114.52.*	Forward to Port P3
131.114.52.*	*	Drop

Table 2

PISA logical architecture

Protocol Independent Switch Architecture (PISA)

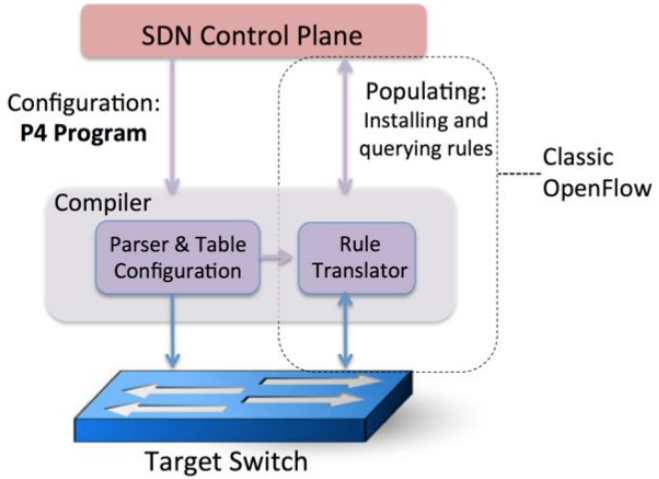
- Programmable *parser*
  - yields a **packet header vector**
    - header fields + metadata
- Programmable *match-action pipeline*
  - Packet header vectors flows through a sequence of logical match stages that run in series or in parallel
- Programmable *deparser*
  - Recompose the packet be serializing headers in the desired order



Source: SigCOMM paper (2013)

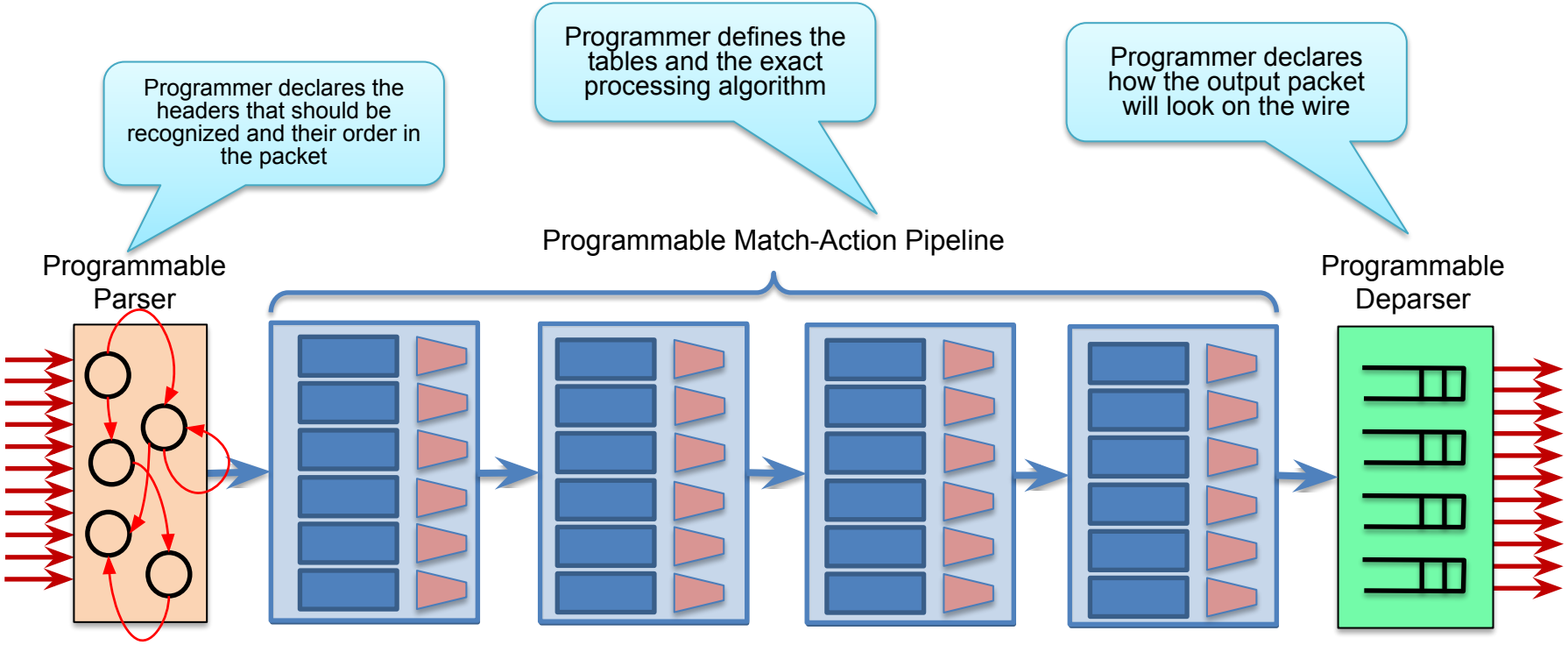
P4 Language design principles

- **Reconfigurability**
  - the controller should be able to reconfigure the switch on live
- **Protocol independence**
  - the controller defines arbitrary packet formats through
    - custom parser
    - custom MATs
- **Target independence**
  - low level details not exposed to the programmer
  - the P4 compiler takes care of translating a target independent program into a target dependent one
  - which targets? **Domain Specific Processors**
    - some switch ASIC → **PISA** (e.g., 12.8 Tbps Intel Tofino 2, etc.)
    - some NPUs
    - NetFPGAs
    - CPUs (sw switches)
- P4 used to program the switch, OF or any other south bound interface can be be used to populate the tables
- P4 language consortium https://p4.org/
- Specifications: $P4_{14}$ (2018) and $P4_{16}$ (2019)



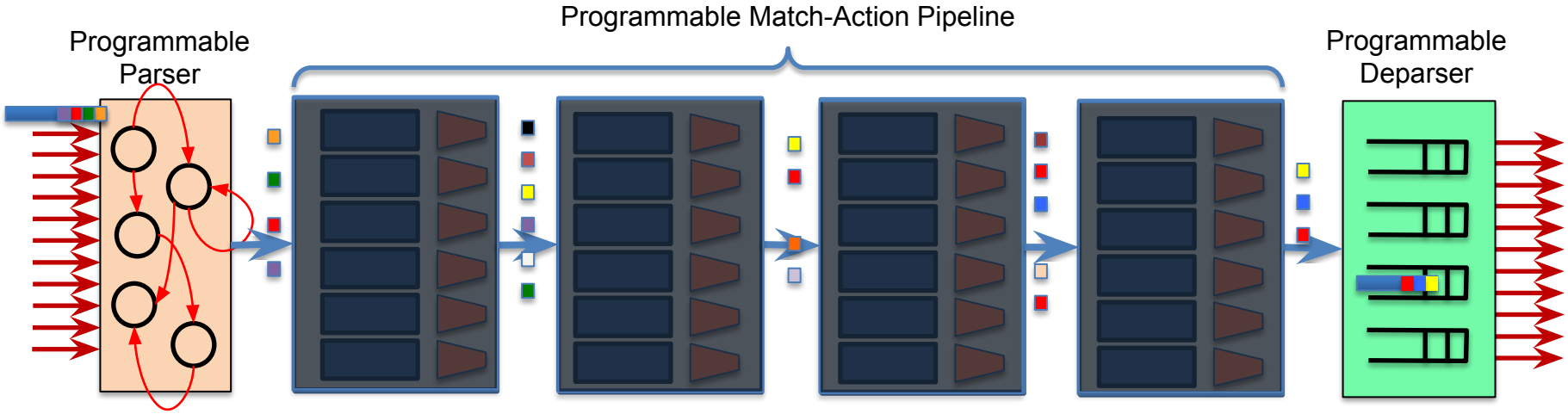
Source: CCR P4 paper (2014)

PISA: Protocol-Independent Switch Architecture



Source: p4.org

Packet Processing in PISA



Source: p4.org

P4 targets vs. architectures

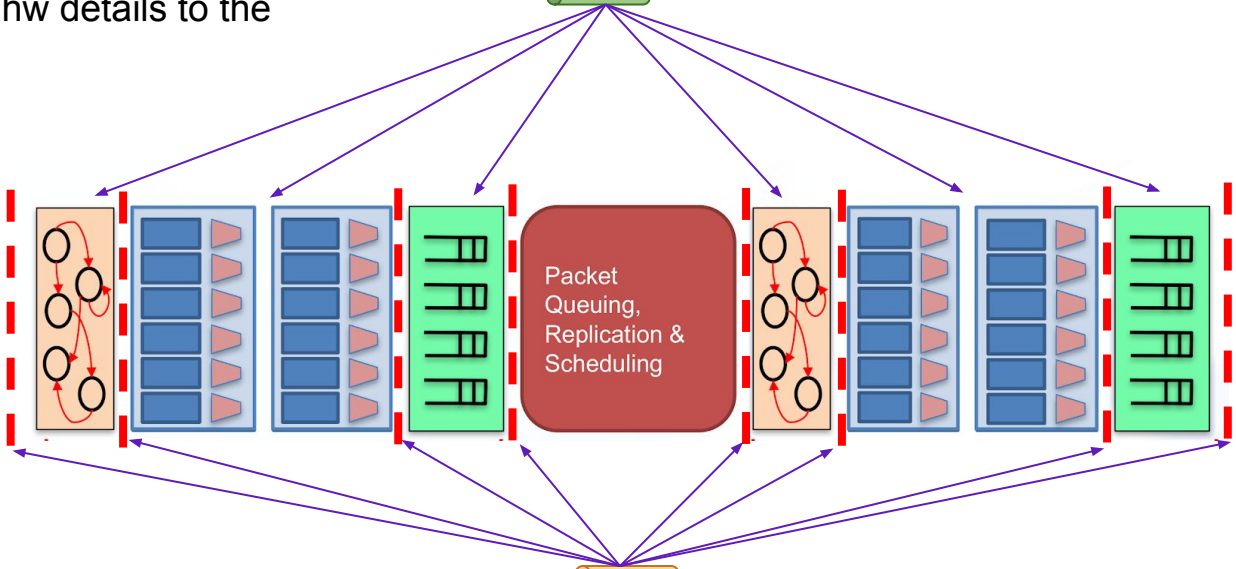
- $P4_{14}$ targeted PISA like switches
- $P4_{16}$ extends the support to multiple programmable devices (**targets**) through the notion of **architecture**
- P4 Architecture
  - is a programming model (a programming abstraction)
  - the architecture is what the programmer sees, a logical view of the processing
  - hide the underlying hw details to the programmer
  - provides an interface to program a target via some set of **P4-programmable components**, **externs**, **fixed components**
- Device vendors should provide compiler and architecture for their target

P4 architectures

The architecture is what the programmer sees, a logical view of the processing hide the underlying hw details to the programmer

`my_program.p4`

Written against a specific architecture
Defines the processing of each block



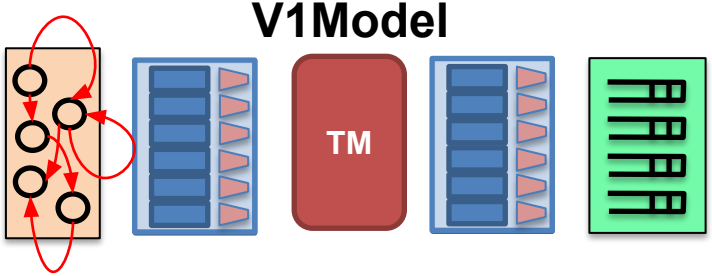
`architecture.p4`
Provided by switch vendor

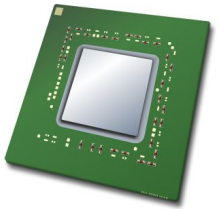
Defines which blocks are available, the interfaces of each block, and their capabilities

Source: Open Networking Foundation

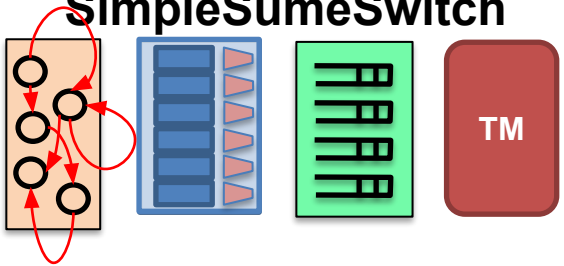
Architectures and targets

V1Model

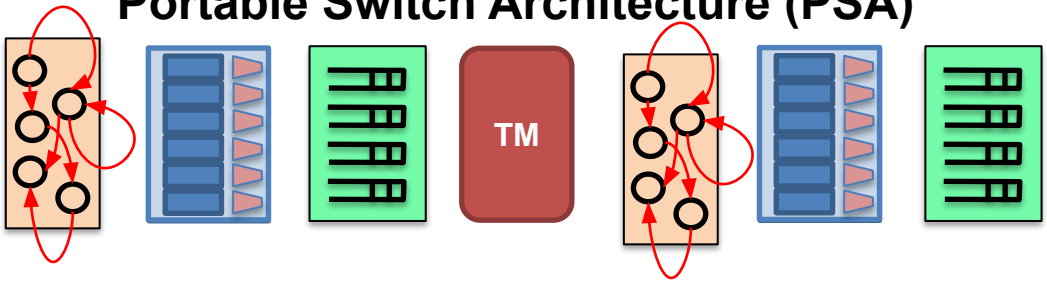




SimpleSumeSwitch



Portable Switch Architecture (PSA)

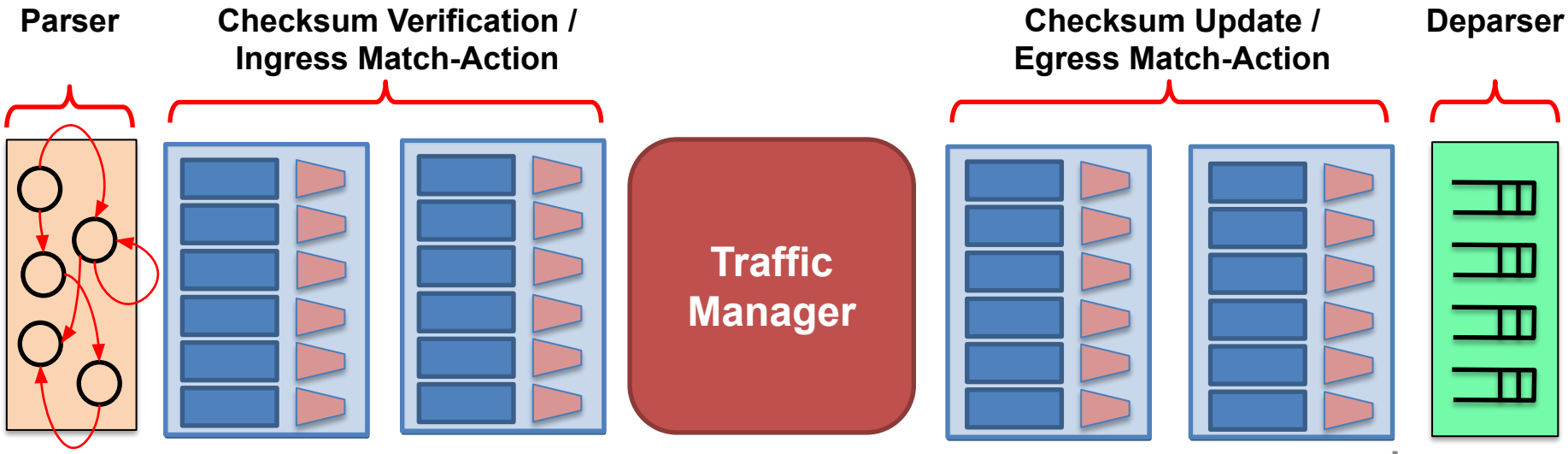


Anything

Source: p4.org

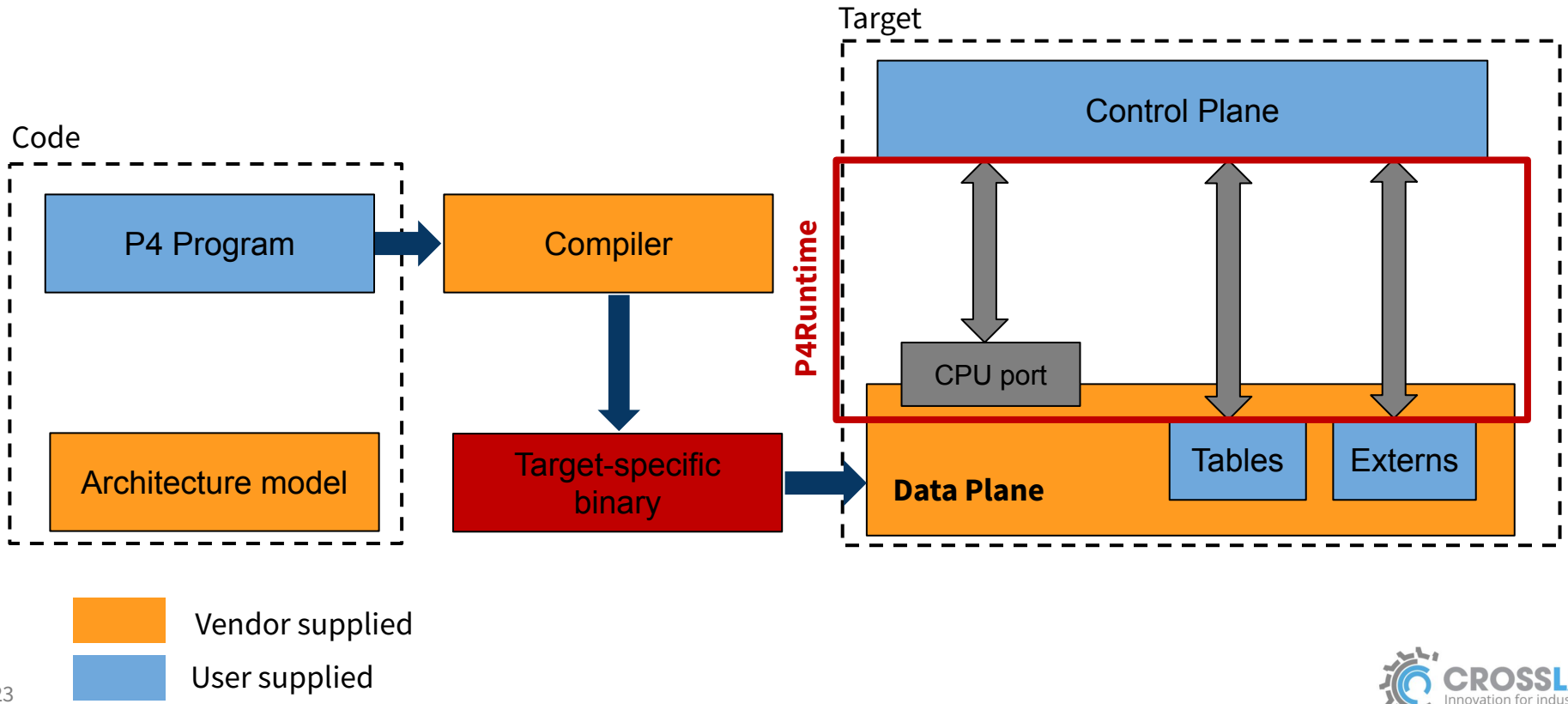
The v1Model architecture for the BMv2 soft switch

- Implemented in Bmv2's simple_switch target (not the only possible!!)
- Very similar to the original PISA architecture



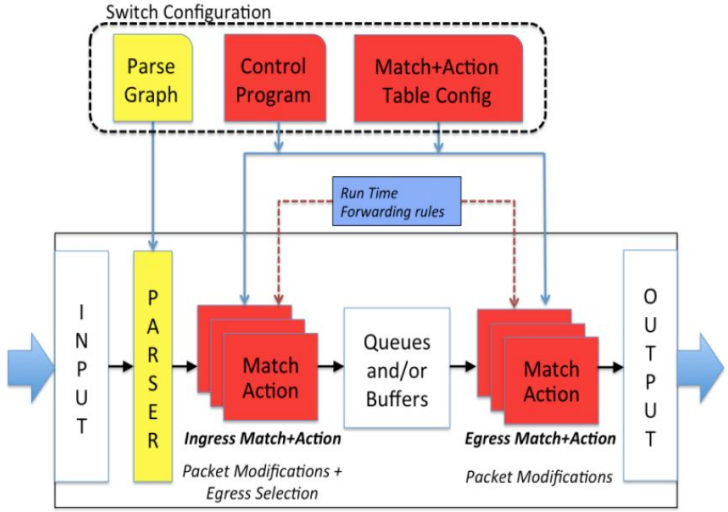
Source: p4.org

P4: programming a target



P4 Programming

- BMv2 Simple Switch
- v1model architecture
  - standard and intrinsic metadata
  - extern “specialized” functions
- P4 code will contain sections for:
  - Parsing
  - Control flow
    - ingress pipeline
    - extern pipeline
  - Deparsing

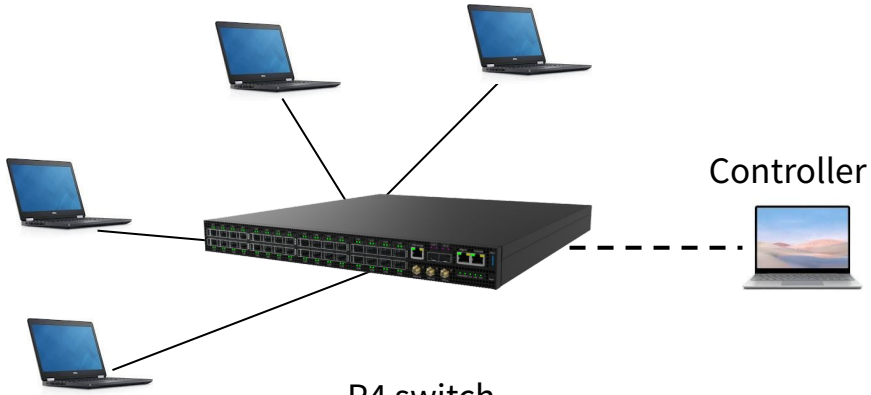


Source: p4.org

But we need first to setup the development environment...

A real running testbed?

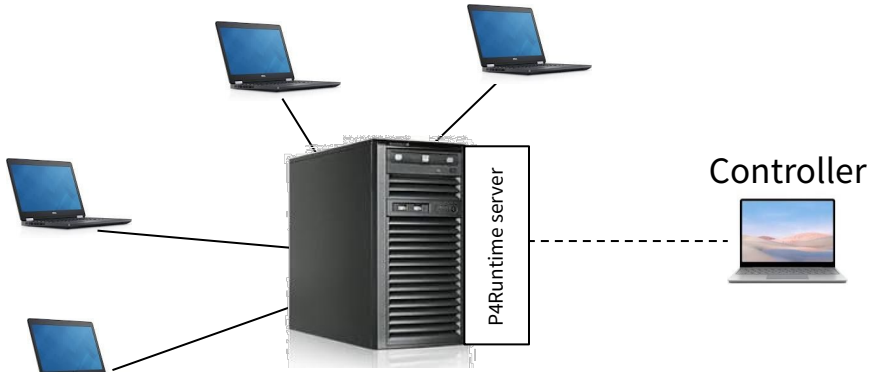
- Not very practical…
  - Need a real P4 switch and too many PCs…



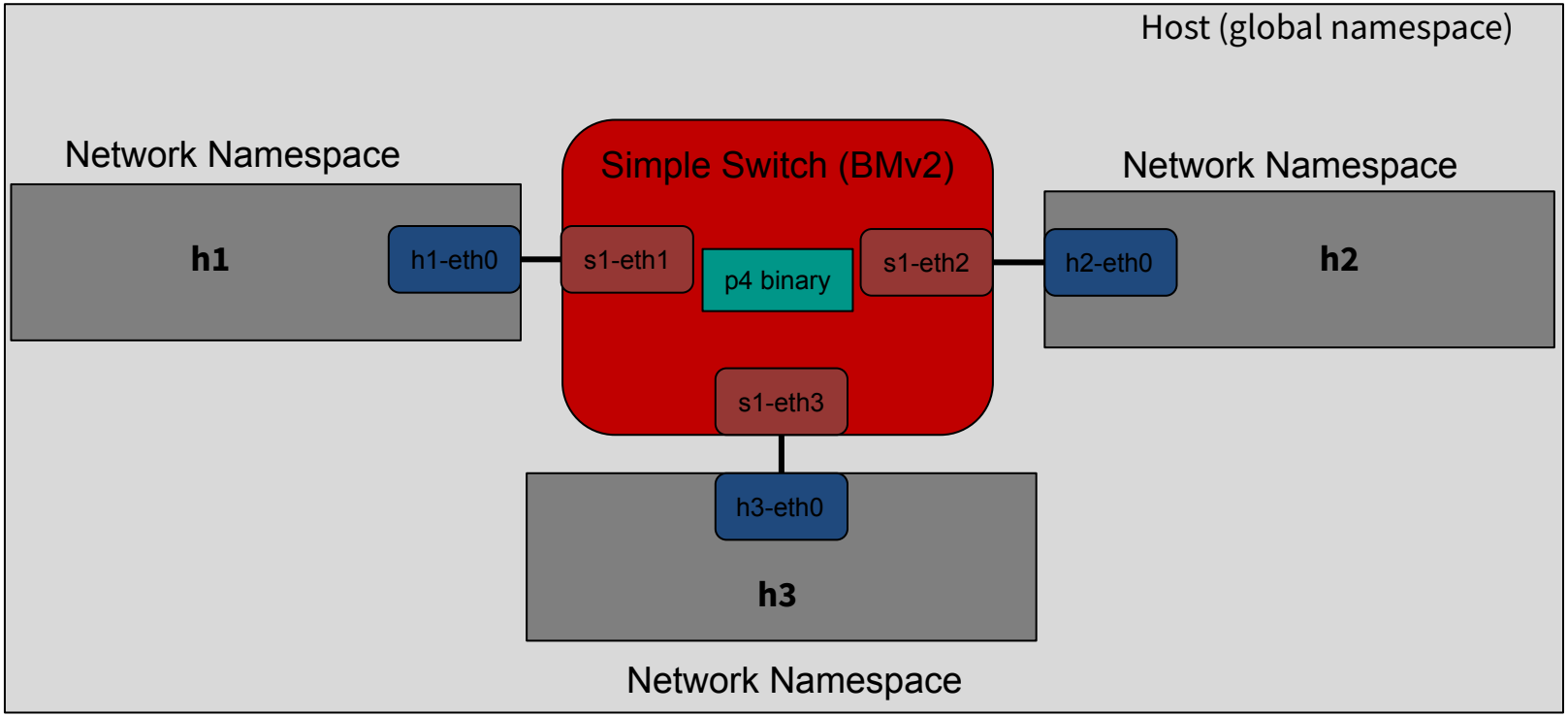
P4 switch
connected to hosts
and controller

A real running testbed?

- Again, not very practical…
  - Have not many laptops…



P4 (virtual) running environment



What we need

- A Linux machine with:
  - The **BMv2** software switch
  - **p4c**: the reference P4 compiler
  - **Mininet**: a lightweight network emulation environment
- As lot of dependencies need to be installed, I suggest downloading a VM with everything in it:
  - Tutorial from P4 official page:
    - **https://github.com/p4lang/tutorials**
    - lots of exercises in there
  - github repository of examples presented today
    - **https://github.com/grp-xx/5G-SummerSchool-2022**

p4 program structure

```
#include <core.p4>
#include <v1model.p4>
```

Libraries

```
struct metadata {
}
struct headers {
}
```

Declarations

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start{
        transition accept;
    }
}
```

Parser

```
control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
    apply {  }
}
```

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action forward(bit<9> egress_port){
        standard_metadata.egress_spec = egress_port;
    }

    table repeater {...}

    apply {
        repeater.apply();
    }
}
```

Control for Ingress Pipeline

Egress Pipeline

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {  }
}
```

```
control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
    apply { }
}
```

Packet re-assembly

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {

    /* Deparser not needed */

    }
}
```

```
V1Switch(
    MyParser(),
    MyVerifyChecksum(),
    MyIngress(),
    MyEgress(),
    MyComputeChecksum(),
    MyDeparser()
) main;
```

main

v1Model standard_metadata

```
struct standard_metadata_t {
   bit<9>  ingress_port;
   bit<9>  egress_spec;
   bit<9>  egress_port;
   bit<32> clone_spec;
   bit<32> instance_type;
   bit<1>  drop;
   bit<16> recirculate_port;
   bit<32> packet_length;
   bit<32> enq_timestamp;
   bit<19> enq_qdepth;
   bit<32> deq_timedelta;
   bit<19> deq_qdepth;
   bit<48> ingress_global_timestamp;
   bit<32> lf_field_list;
   bit<16> mcast_grp;
   bit<1>  resubmit_flag;
   bit<16> egress_rid;
   bit<1>  checksum_error;
}
```

The arrival port of the packet

The port the packet should be sent to

The departure port (can be read in the egress pipeline only)

P4 Types

- Statically typed language with
  - base types
  - composed types

```
bool
bit<W>
int<W>
varbit<W>
enum
typedef bit<48> macAddr_t;
...
no float, no strings!
```

```
header Ethernet_h {
 bit<48> dstAddr;
 bit<48> srcAddr;
 bit<16> etherType;
}
Ethernet_h etherHeader;
```

```
header Mpls_h {
 bit<20> label;
 bit<3>  tc;
 bit     bos;
 bit<8>  ttl;
}
Mpls_h[10] mpls;
```

```
header_union IP_h {
 IPv4_h v4;
 IPv6_h v6;
}
IP_h ipHeader;
```

```
struct standard_metadata_t {
  bit<9> ingress_port;
  bit<9> egress_spec;
  bit<9> egress_port;
... }
```

```
tuple<bit<32>, bool> x;
x = { 10, false };
```

P4 operations

- Arithmetic
  - +, -, * **/ * no division!!! */**
- Logical
  - ~, &, |, ^, <<, >>
- Bit-slicing [m:l]
- Bit concatenation ++
- **No modulo operation!!!**

P4 Variables and constants

- Variables
  - Have local scope
  - not maintained upon the next invocation
  - **cannot be used to save states**!
    - use tables and extern for that

```
bit<8> x = 10;

typedef bit<16> TcpPort;
TcpPort s_port
s_port = 10000;
```

- Constants

```
const bit<8> x = 10;

typedef bit<16> TcpPort;
const TcpPort s_port = 10000;
```

P4 main statements

- `return`
  - terminate action or control in which it is contained
- `exit`
  - terminate of all blocks in execution
- `conditions`
  - cannot be used in parsers

```
if (x==100) {...} else {...}
```

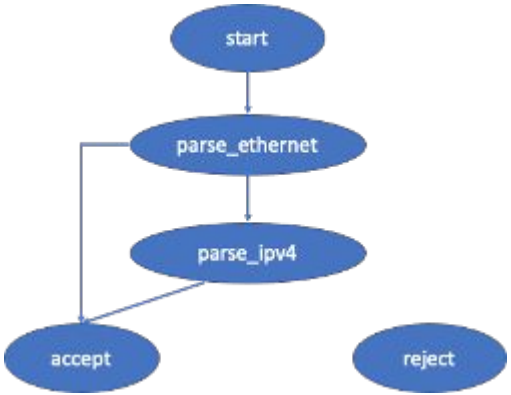
- `switch`
  - can be used in control blocks only

```
switch (hdr.ethernet.etherType) {
   0x86dd: { /* body omitted */ }
   0x0800: { /* body omitted */ }
   0x0802: { /* body omitted */ }
   0xcafe: { /* body omitted */ }
   default: { /* body omitted */ }
}
```

- **Forget about loops in P4!!!**

P4 parser

- Finite State Machine
  - states
    - predefined **`start, accept, reject`**
    - + user defined ones
  - transitions
  - loops are allowed (e.g. tunneling)
  - more advanced methods: `verify, lookahead, ...`




```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t std_meta) {

    state start {
      transition parse_ethernet;
    }

    state parse_ethernet {
      packet.extract(hdr.ethernet);
      transition select(hdr.ethernet.etherType)
    {
        0x800: parse_ipv4;
        default: accept;
      }
    }

    state parse_ipv4 {
    ...
    }
}
```

P4 control

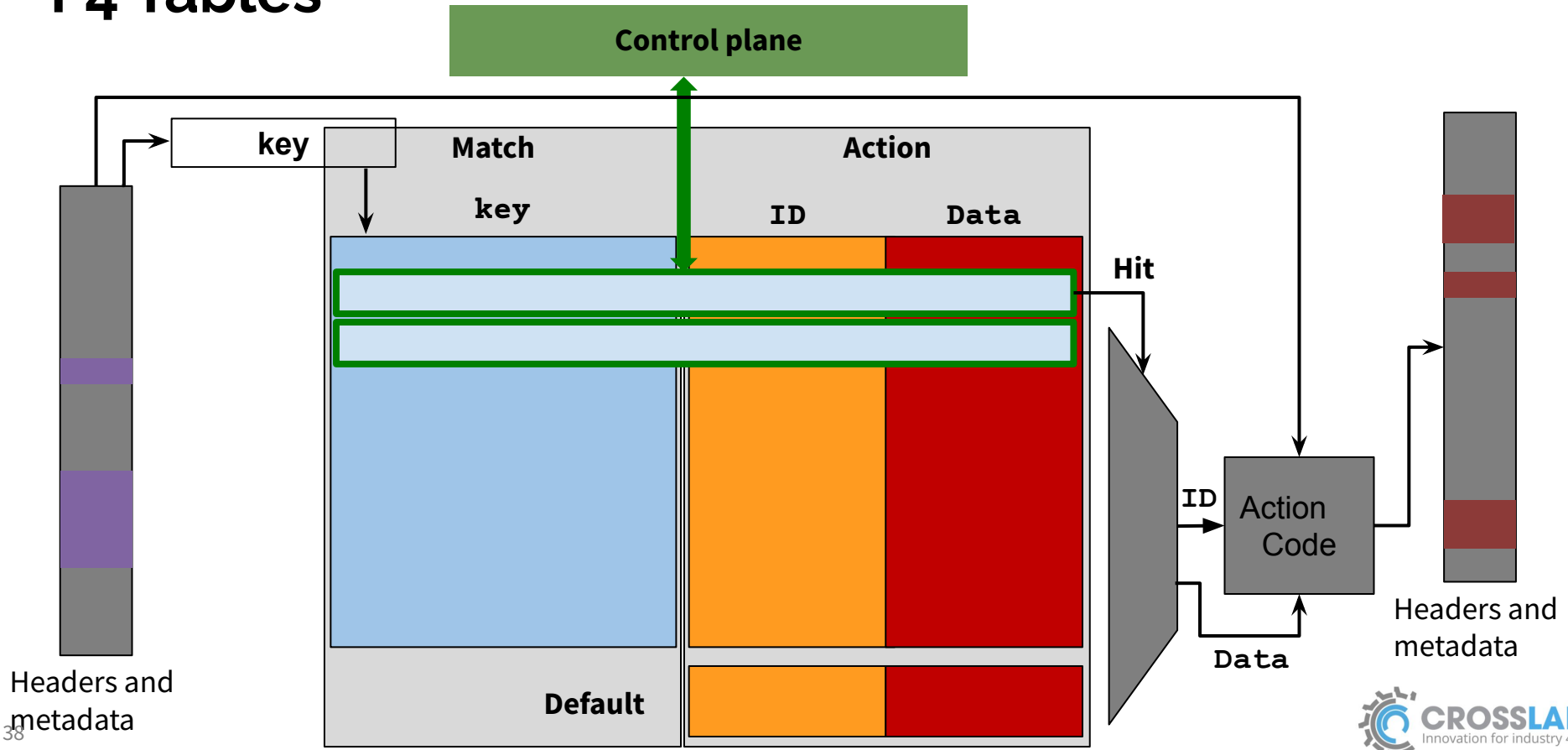
- Control blocks implements:
  - match actions pipelines
  - deparser
  - additional packet processing (e.g. checksum update)
- Control blocks contain:
  - Tables
    - match a key and return an action (and data)
  - Actions
    - pretty much like C functions, for software re-use
    - no return value, though
    - parameters have **directions**
      - `in` ReadOnly in the action
      - `out` to be written in the action
      - `inout` ReadWrite
  - Control flow
    - blocks of imperative code (without loops!)
    - may contain advanced concepts such as cloning packets, sending packets to control plane, etc.

P4 action and control flow

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t std_meta) {
  /* Declarations region */
  bit<48> tmp;

  apply {
    /* Control Flow */
    tmp = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = hdr.ethernet.srcAddr;
    hdr.ethernet.srcAddr = tmp;
    std_meta.egress_spec = std_meta.ingress_port;
  }
}
```

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t std_meta) {

  action swap_mac(inout bit<48> src,
                  inout bit<48> dst) {
    bit<48> tmp = src;
    src = dst;
    dst = tmp;
  }

  apply {
    swap_mac(hdr.ethernet.srcAddr,
             hdr.ethernet.dstAddr);
    std_meta.egress_spec = std_meta.ingress_port;
  }
}
```

P4 Tables


Control plane
key
Match
key
Action
ID
Data
Hit
ID
Action
Code
Data
Default
Headers and
metadata
Headers and
metadata

P4 tables in the code...

```
table ipv4_lpm {
    key = {
      hdr.ipv4.dstAddr: lpm;
    }
    actions = {
      ipv4_forward;
      drop;
      NoAction;
    }

    size = 1024;

    default_action = NoAction();
}
```

Table Name

Match type

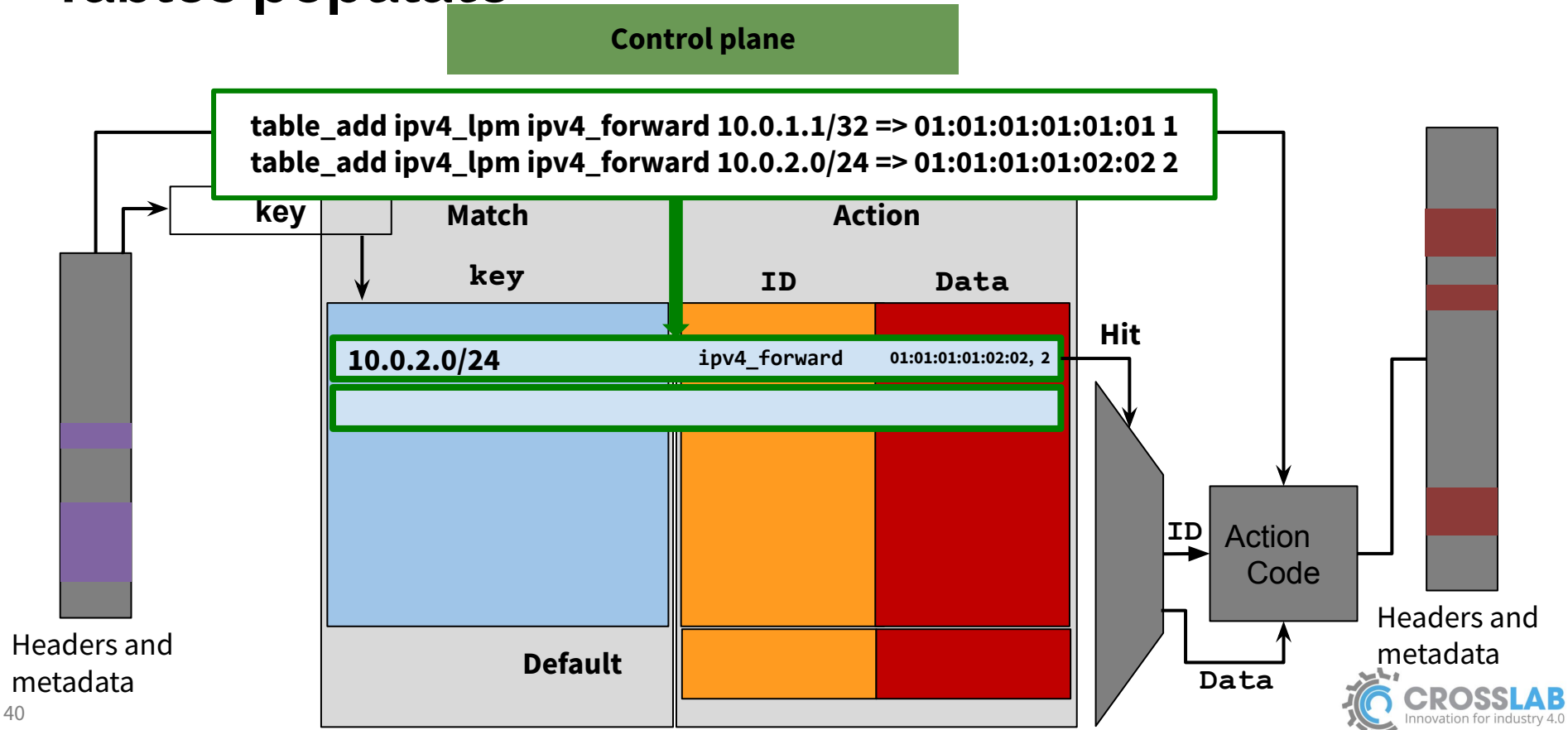
Possible Actions

Max # of table entries

Default action

```
/* core.p4 */
match_kind {
    exact,
    ternary,
    lpm
}

/* v1model.p4 */
match_kind {
    range,
    selector
}

/* Some other architecture */
match_kind {
    ...
}
```

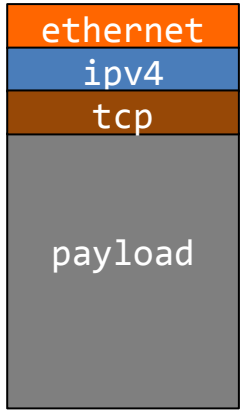
Tables populate



P4 Deparser

- Serializes headers in the desired order and rebuild the packet

```
control DeparserImpl(packet_out packet,
                     in headers hdr) {
  apply {
    ...
    packet.emit(hdr.ethernet);
    packet.emit(hdr.ipv4);
    packet.emit(hdr.tcp);
    ...
  }
}
```



Checksum - Validation and update (1)

- Extern defined in the v1model

```
extern void verify_checksum<T, O>( in bool condition,
                                   in T data,
                                   inout O checksum,
                                   HashAlgorithm algo
                                 );
extern void update_checksum<T, O>( in bool condition,
                                   in T data,
                                   inout O checksum,
                                   HashAlgorithm algo
                                 );
```

Checksum - Validation and update (2)

```
control MyComputeChecksum(...) {
   apply {
     update_checksum(
     hdr.ipv4.isValid(),
     {     hdr.ipv4.version,
           hdr.ipv4.ihl,
           hdr.ipv4.diffserv,
           hdr.ipv4.totalLen,
           hdr.ipv4.identification,
           hdr.ipv4.flags,
           hdr.ipv4.fragOffset,
           hdr.ipv4.ttl,
           hdr.ipv4.protocol,
           hdr.ipv4.srcAddr,
           hdr.ipv4.dstAddr },
           hdr.ipv4.hdrChecksum,
      HashAlgorithm.csum16);
    }
}
```

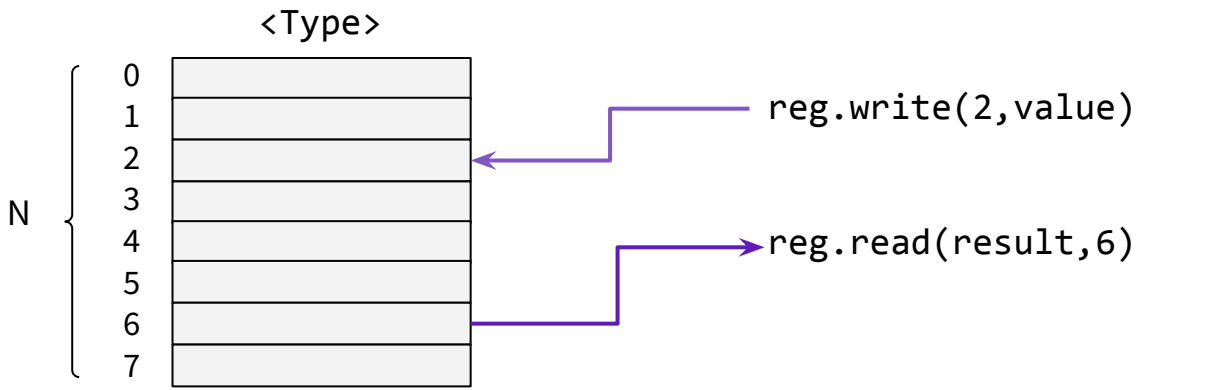
P4 stateful programming

- Stateless objects
  - variables
  - headers
- Stateful objects
  - tables
    - can only be populated by the control plane!
  - Registers
  - Counters
  - Meters

  } *externs*
    - used to limit traffic rate

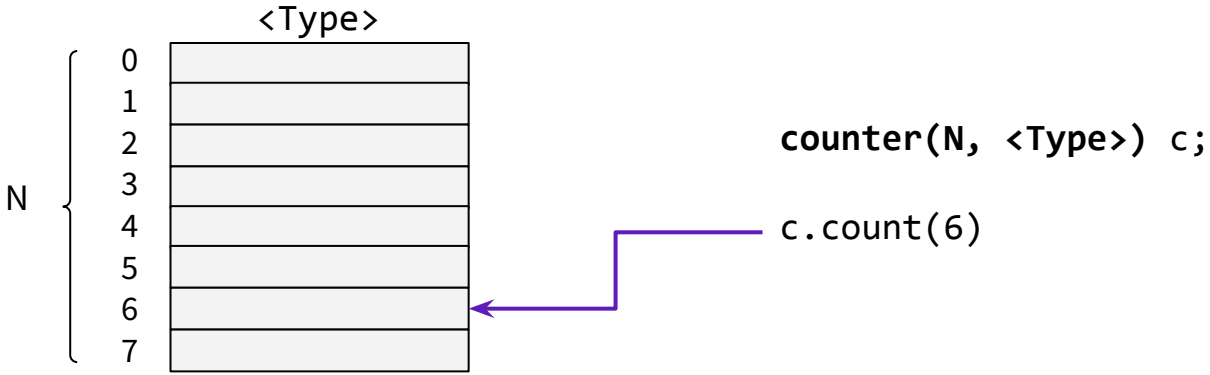
Registers

- Good to store arbitrary data (of type `<Type>`)
- **`read`** and **`write`** methods available
- assigned in arrays of length N



Counters

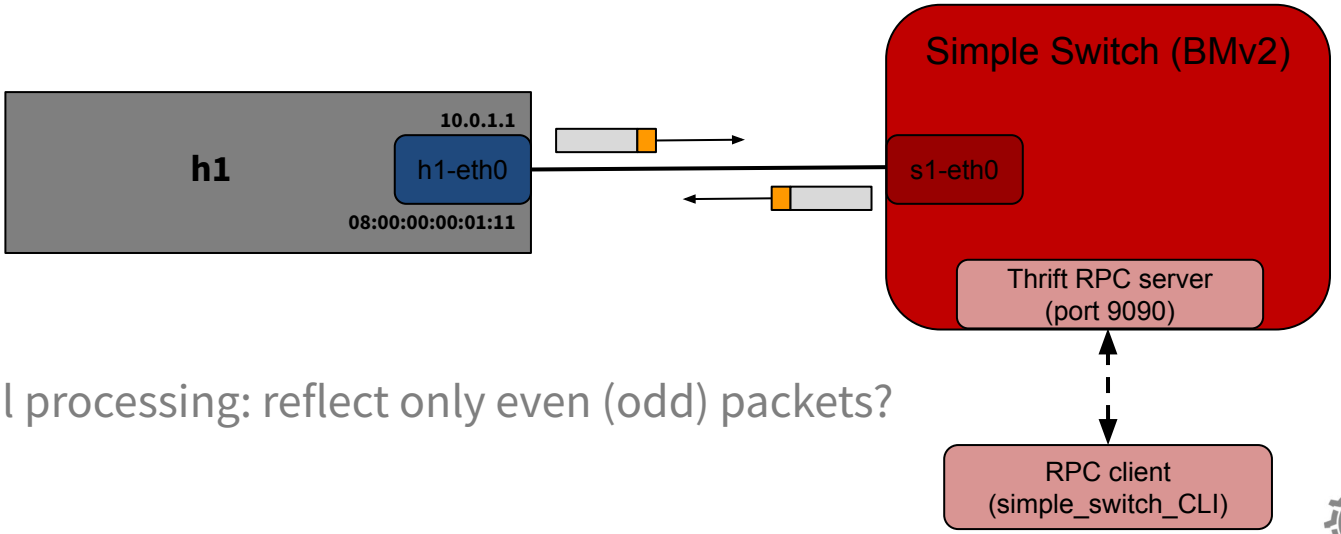
- Good for... counting ;-)
- Three types only:
  - `packets, bytes, packets_and_bytes`
- Only `count` method available (read available from the control plane)
- Assigned in arrays of length N



- there are also *direct counters*: special type of counters attached to tables

P4 Hello World: packet reflector

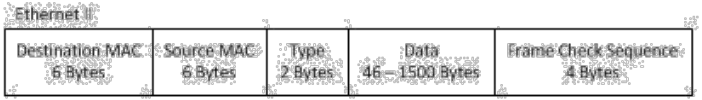
- Not easy to define a hello world for packet processing…
  - can’t print a string on video ;-)
- Packet reflector
  - the switch receives the ethernet frame and sends it back to the host (... swaps mac addresses)



- Stateful processing: reflect only even (odd) packets?

Packet reflector implementation (1)

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

/*** H E A D E R S ****/

typedef bit<48> macAddr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16> etherType;
}

struct metadata {
    /* empty */
}

struct headers {
       ethernet;
}
```




```
/*** P A R S E R ****/

parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

      state start{
         packet.extract(hdr.ethernet);
         transition accept;
      }
}
```

Packet reflector implementation (2)

```
/*** I N G R E S S   P R O C E S S I N G ****/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {


    action swap_macs(inout macAddr_t src, inout macAddr_t dst) {
        macAddr_t tmp = src;
        src = dst;
        dst = tmp;
    }


    apply {
       // Swap MAC addresses.
       swap_macs(hdr.ethernet.srcAddr, hdr.ethernet.dstAddr);

       //Set Output port == Input port
       standard_metadata.egress_spec = standard_metadata.ingress_port;
    }
}
```

```
/*** D E P A R S E R ****/

control MyDeparser(packet_out packet,
                   in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
    }
}
```

```
/*** S W I T C H  ****/

V1Switch(
       MyParser(),
       MyVerifyChecksum(),
       MyIngress(),
       MyEgress(),
       MyComputeChecksum(),
       MyDeparser()
) main;
```

Reflect one out of two? Count reflected frames?

```
/*** I N G R E S S   P R O C E S S I N G ****/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action swap_macs(...){  }

    register<bit<1>>(8) reg;
    apply {
        bit<1> flag;
        bit<32> input_port = (bit<32>) standard_metadata.ingress_port;
        reg.read(flag,input_port);
        reg.write(input_port,flag+1);

        if (flag == 1) {
            mark_to_drop(standard_metadata);
        }
        else {
             swap_macs(hdr.ethernet.srcAddr, hdr.ethernet.dstAddr);
            standard_metadata.egress_spec = standard_metadata.ingress_port;
        }
     }

}
```

```
/*** E G R E S S ****/

control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata)
{

    // Counting the number of reflected frames
    counter(8,CounterType.packets) egress_port_counter;

    apply {
        egress_port_counter.count(
(bit<32>)standard_metadata.egress_port);
    }

}
```

- How do I access **registers** and **counters** from the **control plane**?

```
simple_switch_CLI --thrift-port 9090
> register_read...
> counter_read...
```

And finally using tables…

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t
                                  standard_metadata) {

    register<bit<1>>(8) reg;
    bit<1> flag;

    action swap_macs(inout macAddr_t src, inout macAddr_t dst)
    {
        macAddr_t tmp = src;
        src = dst;
        dst = tmp;
    }

    action frame_reflect() {
        swap_macs();
        standard_metadata.egress_spec =
                        standard_metadata.ingress_port;
    }

    action drop() {
        mark_to_drop(standard_metadata);
    }
```

```
        table odd_even {
        key = {
            flag: exact;
        }

        actions = {
            frame_reflect;
            drop;
        }

        size = 8;
        default_action = drop();
       }

    apply {
        bit<32> input_port = (bit<32>)
                            standard_metadata.ingress_port;
        reg.read(flag,input_port);
        reg.write(input_port,flag+1);
        odd_even.apply();
     }
}
```

- To populate the table…

```
simple_switch_CLI --thrift-port 9090
> table_add odd_even frame_reflect 0 =>
```

Add your own header…

A simple (and useless) example: add the input port number after the ethernet header

- In the declaration section

```
header extra_t {
    bit<16> in_port;
}

struct headers {
    extra_t extra;
}
```

- In the MyIngress control block

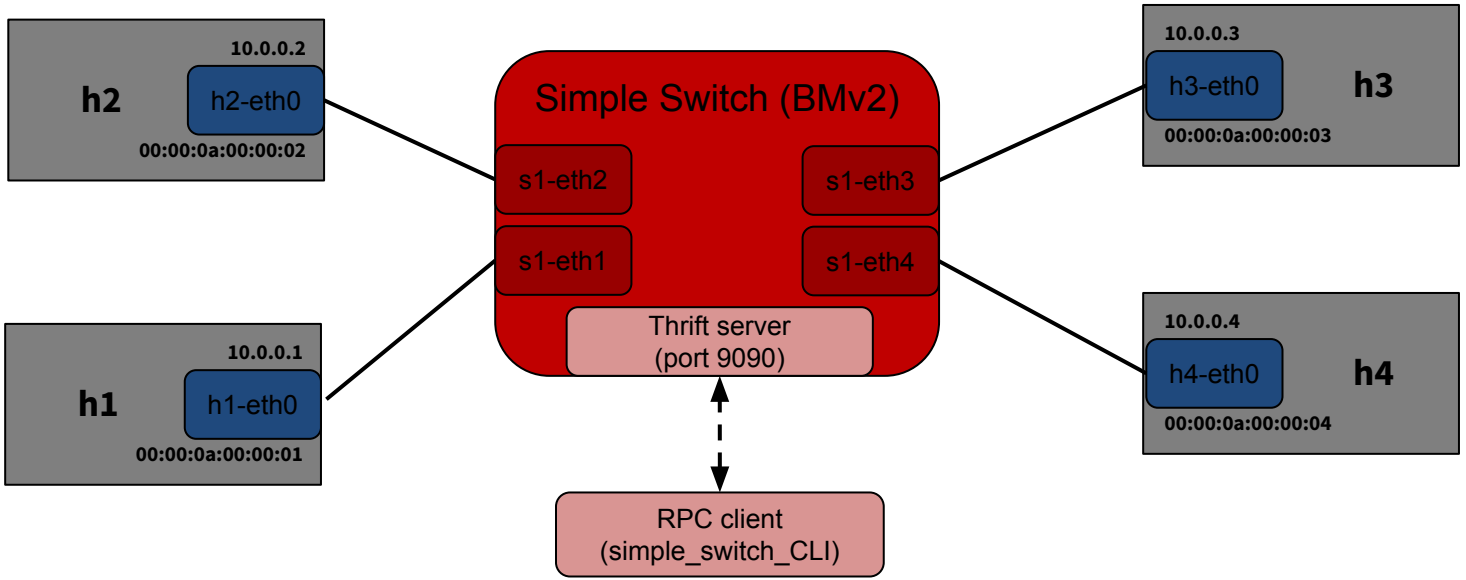
```
  hdr.extra.setValid();
   hdr.extra.in_port = (bit<16>) std_meta.ingress_port;
```

- In the deparser

```
control MyDeparser(packet_out packet, in headers hdr) {

 apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.extra);
        }
}
```

L2 Switching

- Switching table provided through the control plane
- Table populated through the simple_switch_CLI (gRPC client)



L2 Switching - Solution

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action drop() {
        mark_to_drop(standard_metadata);
    }

    action eth_forward(bit<9> out_port) {
        standard_metadata.egress_spec = out_port;
    }
```

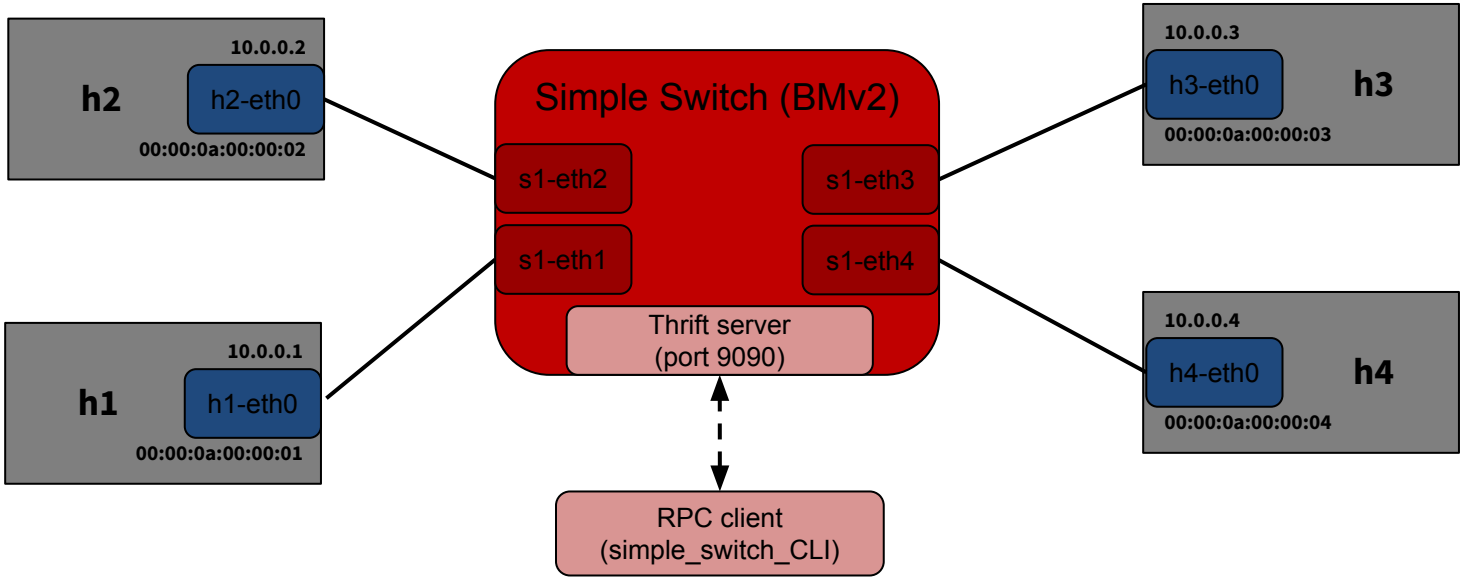
```
table out_iface {
        key = {
            hdr.ethernet.dstAddr: exact;
        }
        actions = {
            drop;
            eth_forward;
        }
        size = 8;
        default_action = drop();
    }

    apply {
        out_iface.apply();
    }
}
```

- To populate the table...

```
simple_switch_CLI --thrift-port 9090
> table_add out_iface eth_forward 00:00:0a:00:00:01 => 1
> ...
> table_add out_iface eth_forward 00:00:0a:00:00:04 => 4
```

L2 Switching with Access Control List (ACL)

- Switching table provided through the control plane
- Table populated through the simple_switch_CLI (gRPC client)
- Allow forwarding only from mac address included in ACL



L2 Switching with ACL - Solution

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {

    action block() {
        mark_to_drop(standard_metadata);
    }
    table acl {
    key = {
            hdr.ethernet.srcAddr: exact;
    }
    actions = {
               block;
               NoAction;
    }
    default_action = block();
    size = 8;
    }
    apply {
          acl.apply();
    }
}
```

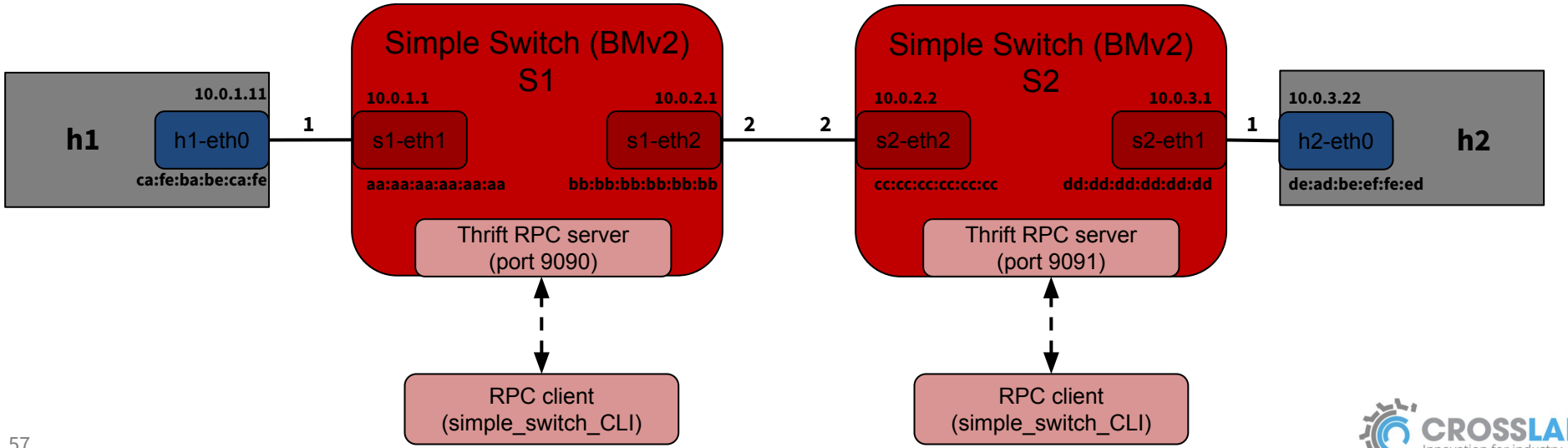
```
table out_iface {
        key = {
            hdr.ethernet.dstAddr: exact;
        }
        actions = {
            drop;
            eth_forward;
        }
        size = 8;
        default_action = drop();
    }

    apply {
        out_iface.apply();
    }
}
```

- To populate the table...

```
simple_switch_CLI --thrift-port 9090
> table_add acl NoAction 00:00:0a:00:00:02 =>
> table_add acl NoAction 00:00:0a:00:00:03 =>
```

L3 Forwarding

- Forwarding table provided via the control plane
- Table populated through the simple_switch_CLI (gRPC client)



L3-forwarding - Headers

```
const bit<16> TYPE_IPV4 = 0x800;

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
```

```
struct metadata {
    /* empty */
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}
```

L3-forwarding - Parser

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {

        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType){
            TYPE_IPV4: ipv4;  //TYPE_PIV4 = 0x0800
            default: accept;
        }
    }

    state ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }

}
```

L3-forwarding - Control flow

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action drop() {
        mark_to_drop(standard_metadata);
    }

    action ipv4_forward(macAddr_t d_mac, egressSpec_t port){

        hdr.ethernet.dstAddr = d_mac;

        standard_metadata.egress_spec = port;

        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }
```

```
table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }


    apply {
        if ( hdr.ipv4.isValid() ) {
            ipv4_lpm.apply();
        }
    }
}
```

L3-forwarding - Egress processing

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {

    action drop() {
        mark_to_drop(standard_metadata);
    }

    action set_smac(macAddr_t mac) {
        hdr.ethernet.srcAddr = mac;
    }

    table s_mac {
        key = {
            standard_metadata.egress_port: exact;
        }
        actions = {
            set_smac;
            drop;
            NoAction;
        }
        size = 16;
        default_action = NoAction();
    }

    apply {
        s_mac.apply();
    }
}
```

L3 forwarding - Checksum and deparser

```
control MyComputeChecksum(inout headers hdr, inout metadata meta)
{
     apply {
          update_checksum(
          hdr.ipv4.isValid(),
            { hdr.ipv4.version,
             hdr.ipv4.ihl,
               hdr.ipv4.diffserv,
               hdr.ipv4.totalLen,
               hdr.ipv4.identification,
               hdr.ipv4.flags,
               hdr.ipv4.fragOffset,
               hdr.ipv4.ttl,
               hdr.ipv4.protocol,
               hdr.ipv4.srcAddr,
               hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
}
```

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {

        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);

    }
}
```

```
//switch architecture
V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

L3 Forwarding: populating the tables

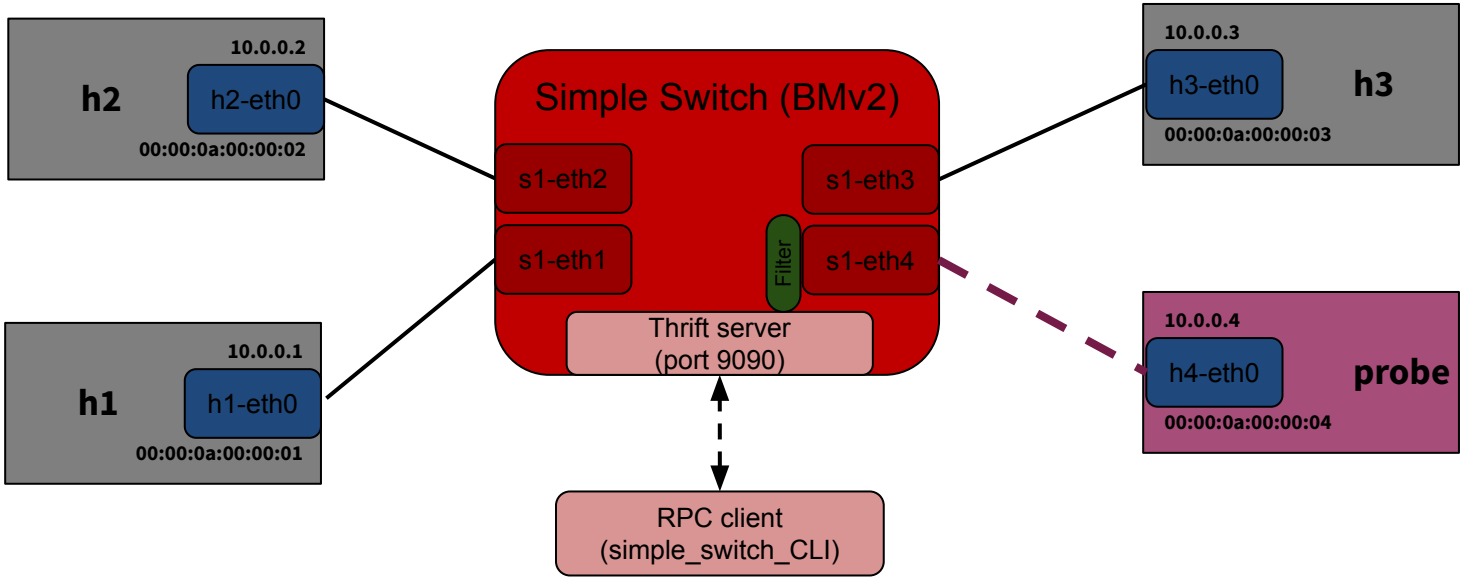
```
> simple_switch_CLI --thrift-port 9090 < s1-commands.txt

> simple_switch_CLI --thrift-port 9091 < s2-commands.txt
```

```
# s1-commands.txt

table_clear ipv4_lpm
table_add ipv4_lpm ipv4_forward 10.0.1.11/32 => ca:fe:ba:be:ca:fe 1
table_add ipv4_lpm ipv4_forward 10.0.3.0/24  => cc:cc:cc:cc:cc:cc 2

table_clear s_mac
table_add s_mac set_smac 1 => aa:aa:aa:aa:aa:aa
table_add s_mac set_smac 2 => bb:bb:bb:bb:bb:bb
```

```
# s2-commands.txt

table_clear ipv4_lpm
table_add ipv4_lpm ipv4_forward 10.0.1.0/24  => bb:bb:bb:bb:bb:bb 2
table_add ipv4_lpm ipv4_forward 10.0.3.22/32 => de:ad:be:ef:fe:ed 1

table_clear s_mac
table_add s_mac set_smac 1 => dd:dd:dd:dd:dd:dd
table_add s_mac set_smac 2 => cc:cc:cc:cc:cc:cc
```

IP traffic probe and filter

- The switch normally forward frames among h1, h2 and h3
- in addition, it “mirrors” (clone) every IP packet to the probe
  - Traffic to the probe is further filtered by *protocol* (in the IP header)



IP probe - Headers

```
const bit<16> TYPE_IPV4 = 0x800;

typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
```

```
struct metadata {
    /* empty */
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}
```

IP probe - Parser

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {

        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType){
            TYPE_IPV4: ipv4;  //TYPE_PIV4 = 0x0800
            default: accept;
        }
    }

    state ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }

}
```

IP probe - Control flow

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action drop() {
        mark_to_drop(standard_metadata);
    }
    action eth_forward(bit<9> out_port) {
        standard_metadata.egress_spec = out_port;
    }

    table out_iface {
        key = {
            hdr.ethernet.dstAddr: exact;
        }
        actions = {
            drop;
            eth_forward;
        }
        size = 8;
        default_action = drop();
    }
    apply {
        out_iface.apply();
    }
}
```

```
action pkt_clone(bit<32> mirror_id) {
        clone(CloneType.I2E, mirror_id);
}

table monitor {
        key = {
            hdr.ipv4.protocol: exact;
        }
        actions = {
            NoAction;
            pkt_clone;
        }
        size = 8;
        default_action = NoAction();
 }


    apply {
        if ( hdr.ipv4.isValid() ) {
            monitor.apply();
        }

       out_iface.apply();
     }
```

IP probe: populating the tables

```
> simple_switch_CLI --thrift-port 9090 < s1-commands.txt
```

```
# s1-commands.txt

table_add out_iface eth_forward 00:00:0a:00:00:01 => 1
table_add out_iface eth_forward 00:00:0a:00:00:02 => 2
table_add out_iface eth_forward 00:00:0a:00:00:03 => 3

mirroring_add 100 4
table_add monitor pkt_clone 1 => 100
table_add monitor pkt_clone 6 => 100
```

Part II

Giuseppe Lettieri

End host data plane programmability

eBPF and XDP

We are going to see a few practical examples.

If you want to replicate the examples on your own:

- If you already have Linux, download the 5gss-xpd.tar.gz tarball
- Or, download the VM (VirtualBox): https://calcolatori.iet.unipi.it/xdp.ova

The tarball and VM also contain an handout (notes.pdf) for this part of the lecture.