Advanced Topics in Communication Networks

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich

Tue 27 Sep 2022

Last week on

Advanced Topics in Communication Networks

In this lecture, you'll learn

how to optimize the

- performance
- flexibility
- reliability

of network infrastructures

In this lecture, you'll learn

how to optimize the

- performance
- flexibility
- reliability

of network infrastructures

Why should *you* care, **as a user?**

best effort service

IP routers do their best to deliver packets to their destination without making any promises

For many applications and users
doing "its best" is just not good enough anymore

In this lecture, you'll learn

how to optimize the

- performance
- flexibility
- reliability

of network infrastructures

Why should *you* care, as a **network operator?**

IP networks carry always more critical services

Since the end of 2017,

All fixed-network services

(telephony, TV, and Internet)

run on IP technology

Source: https://www.swisscom.ch/en/business/enterprise/themen/work-smart/all-ip.html

IP networks carry always more traffic

always faster user speeds

x

always more users

x

always more content consumed

Techniques

Traffic Engineering

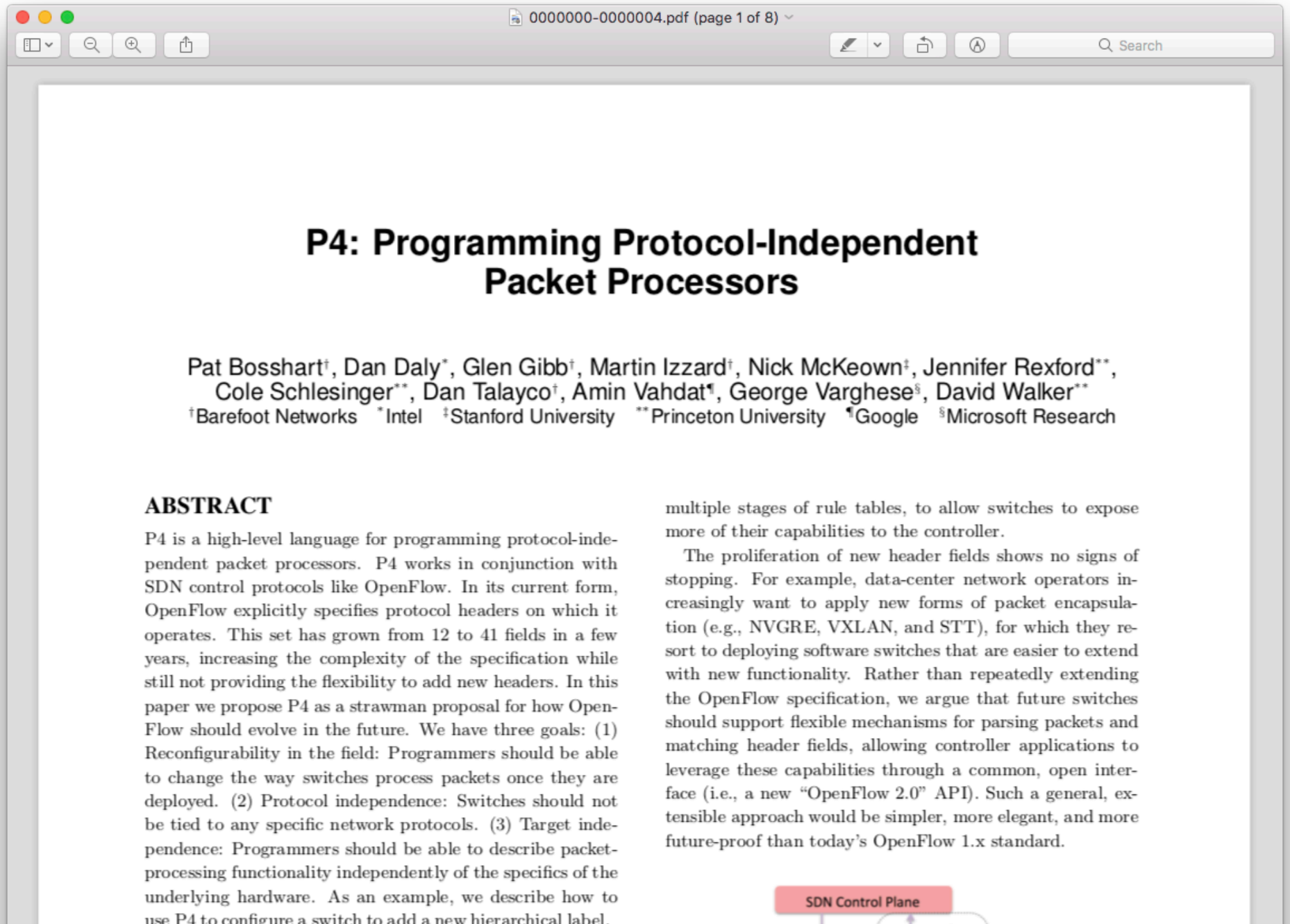
Load Balancing

Quality of Service

Fast Convergence

Besides learning about these techniques,

you'll learn how to *implement* them



0000000-0000004.pdf (page 1 of 8)

Search

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.

multiple stages of rule tables, to allow switches to expose more of their capabilities to the controller.

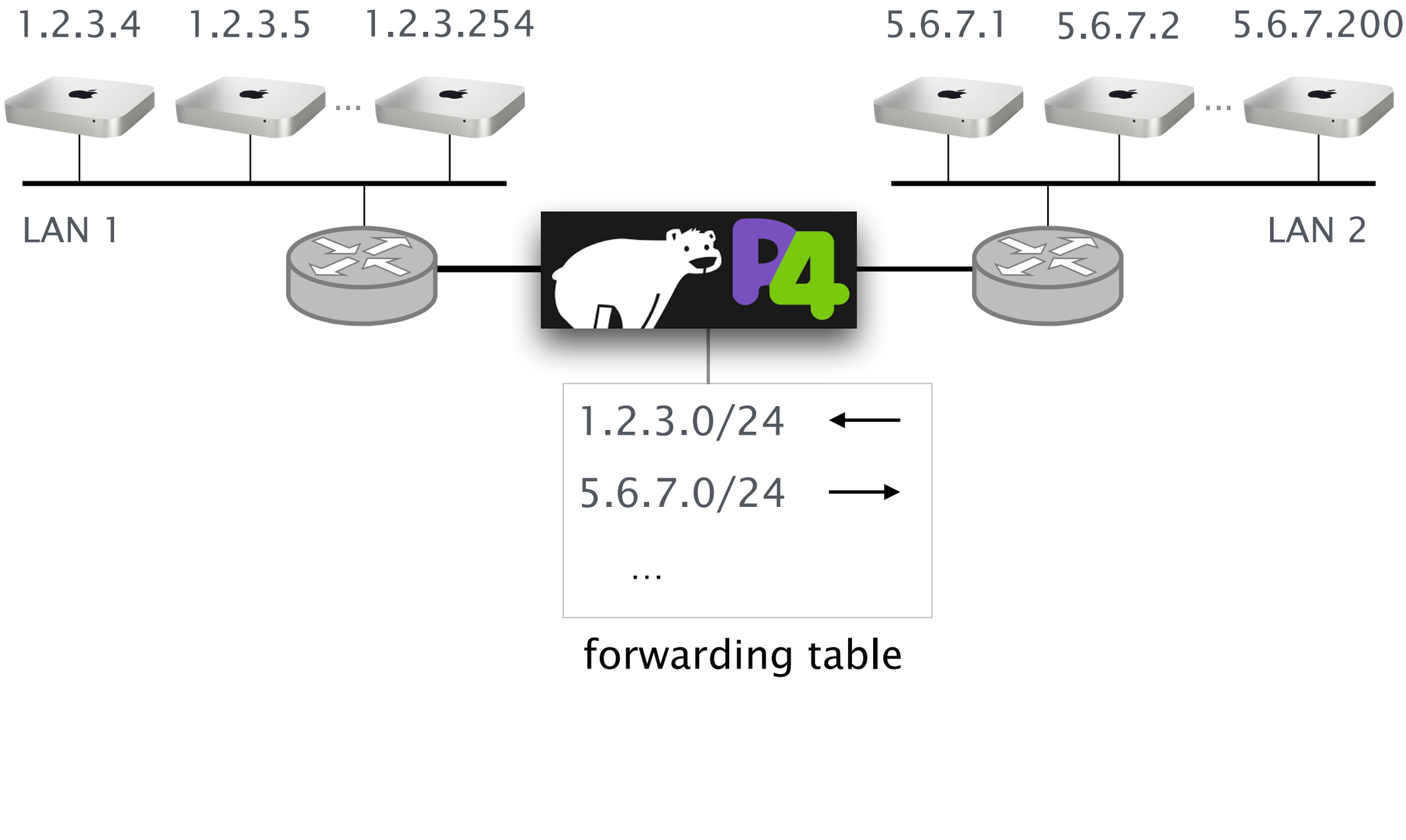
The proliferation of new header fields shows no signs of stopping. For example, data-center network operators increasingly want to apply new forms of packet encapsulation (e.g., NVGRE, VXLAN, and STT), for which they resort to deploying software switches that are easier to extend with new functionality. 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.

Your first

program

IP forwarding

in P4?



When forwarding an IP packet,

an IP router performs four actions:

- Lookup the next hop(s) to use
- Update the MAC addresses
- Decrement TTL
- Forward packets to output port(s)

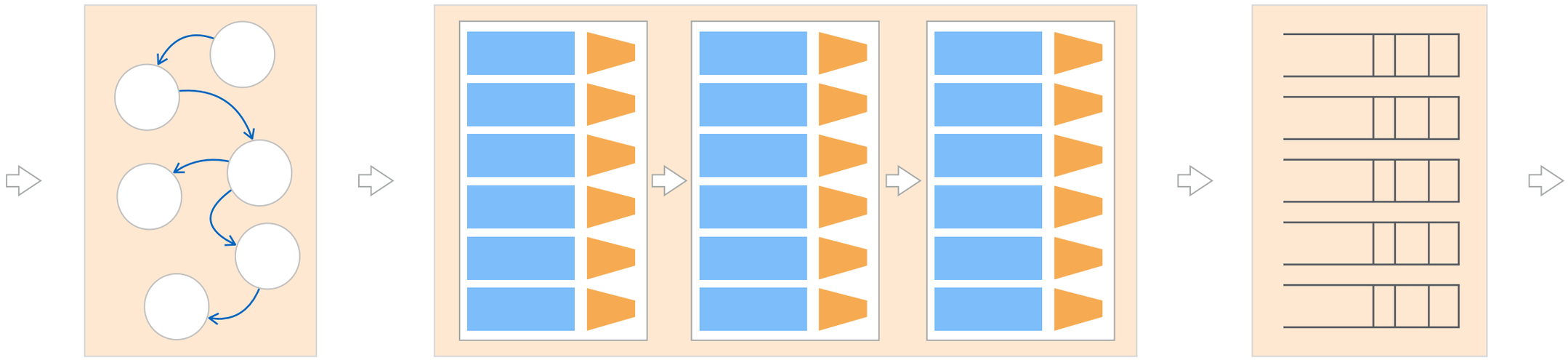
Each of them should be in implemented in P4

A P4 program consists of three basic parts

Parser

Match-Action Pipeline

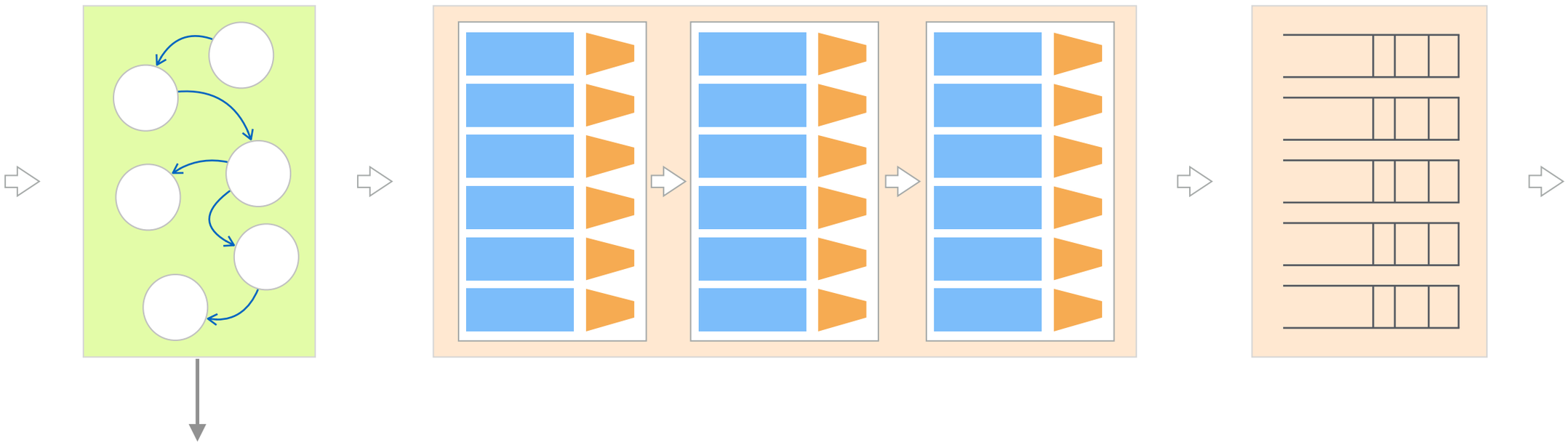
Deparser



Parser

Match-Action Pipeline

Deparser

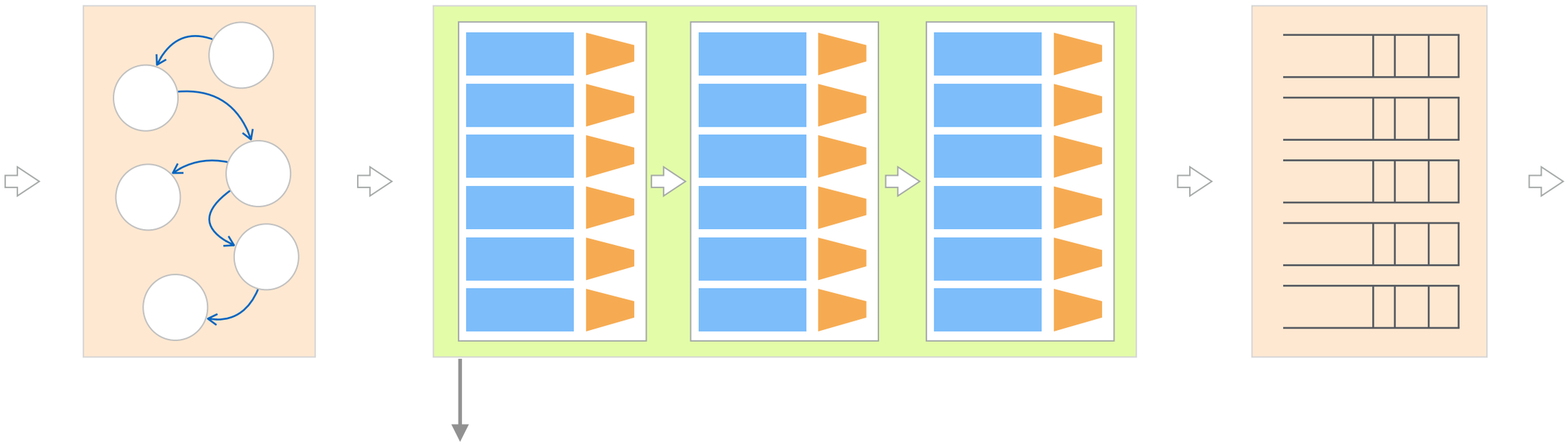


Specifies the headers that
should be extracted from the packet

Parser

Match-Action Pipeline

Deparser

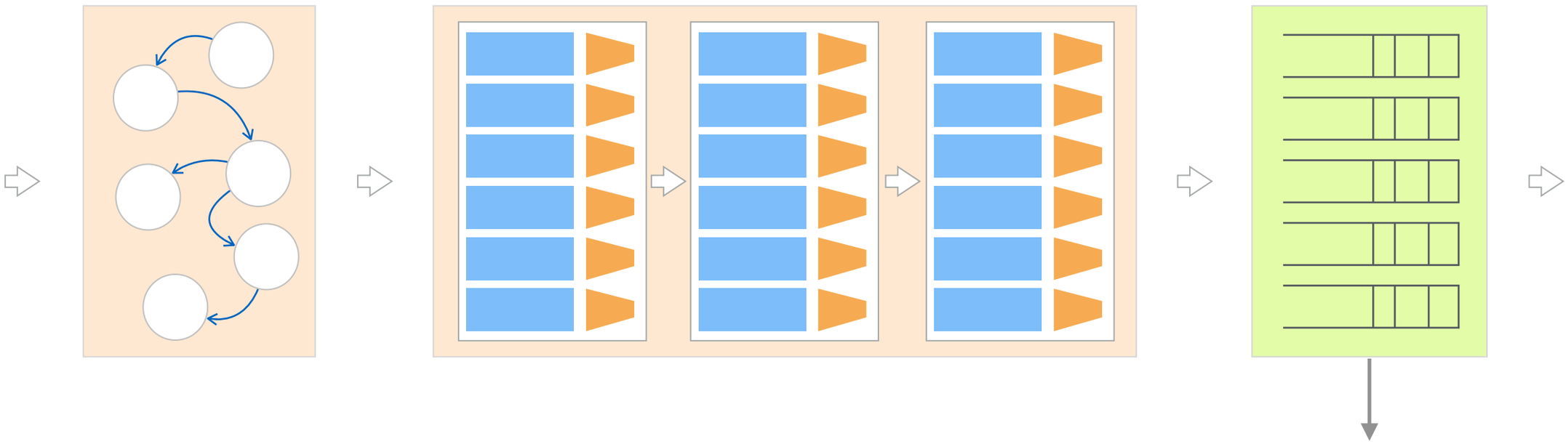


Specifies the processing logic

Parser

Match-Action Pipeline

Deparser



Specifies how
the output packet
will look on the wire

This week on

Advanced Topics in Communication Networks

We will dive into the P4 ecosystem and
look at more advanced language constructs

P4 environment	P4 language	stateful objects
What is needed to program in P4?	Deeper-dive into the language constructs	How do we maintain state in P4?

P4 environment

P4 language

stateful objects

What is needed to program in P4?

Quick historical recap

July

Initial paper

P4$_{14}$ v1.0.1

v1.0.2

v1.0.3

v1.0.4

December

P4$_{16}$ specification (draft)

P4$_{14}$ specification

September

P4$_{16}$ specification

May

P4$_{14}$ v1.0.1

v1.0.2

v1.0.3

v1.0.4

July

Initial paper

December

P4$_{16}$ specification (draft)

2014

2015

2016

2017

2018

P4$_{14}$ specification

September

P4$_{16}$ specification

focus of the lecture

$P4_{16}$ introduces the concept of an architecture

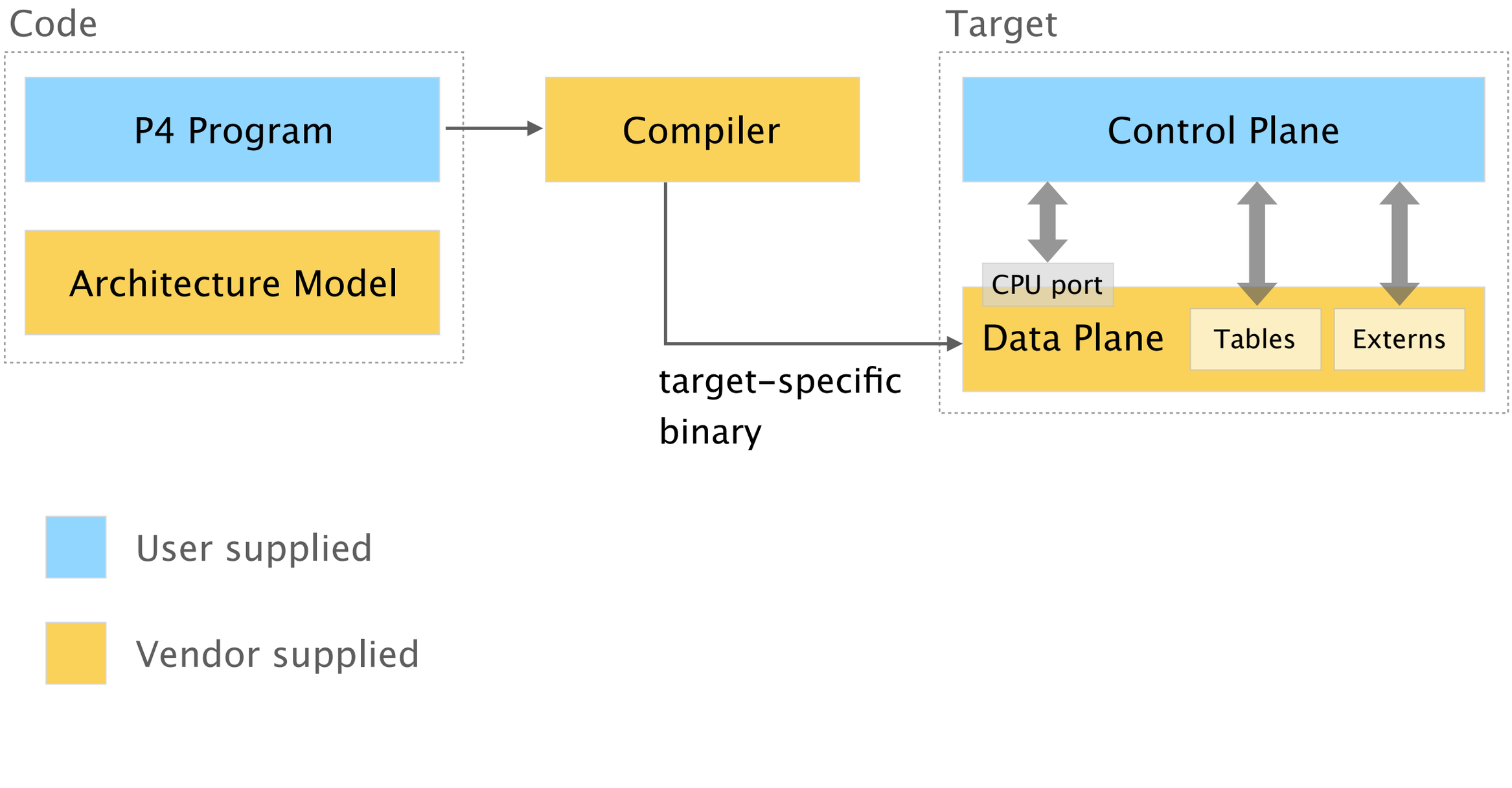
P4 Target

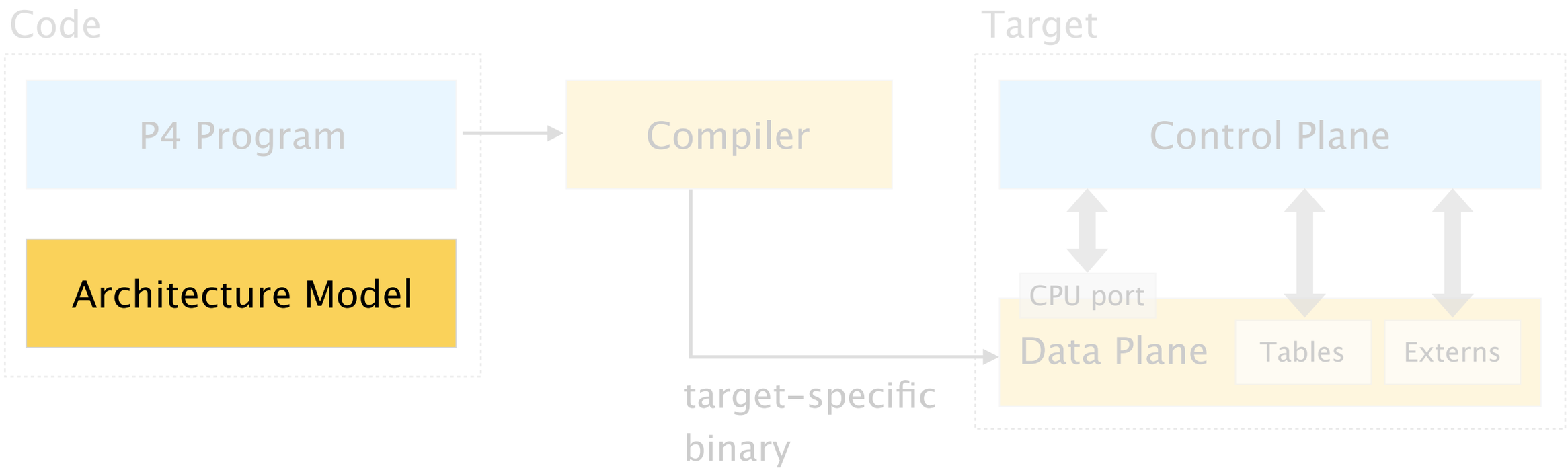
hardware or software entity
being programmed

P4 Architecture

a programming model
for a particular target

Programming a P4 target involves a few key elements





User supplied

Vendor supplied

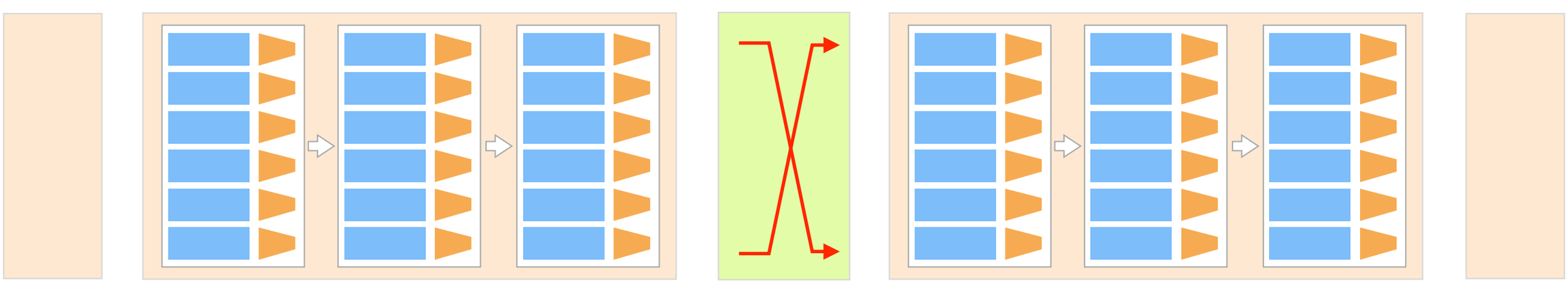
Protocol Independent Switch Architecture (PISA) for high-speed programmable packet forwarding

Ingress

Match-Action Pipeline

Egress

Match-Action Pipeline



Parser

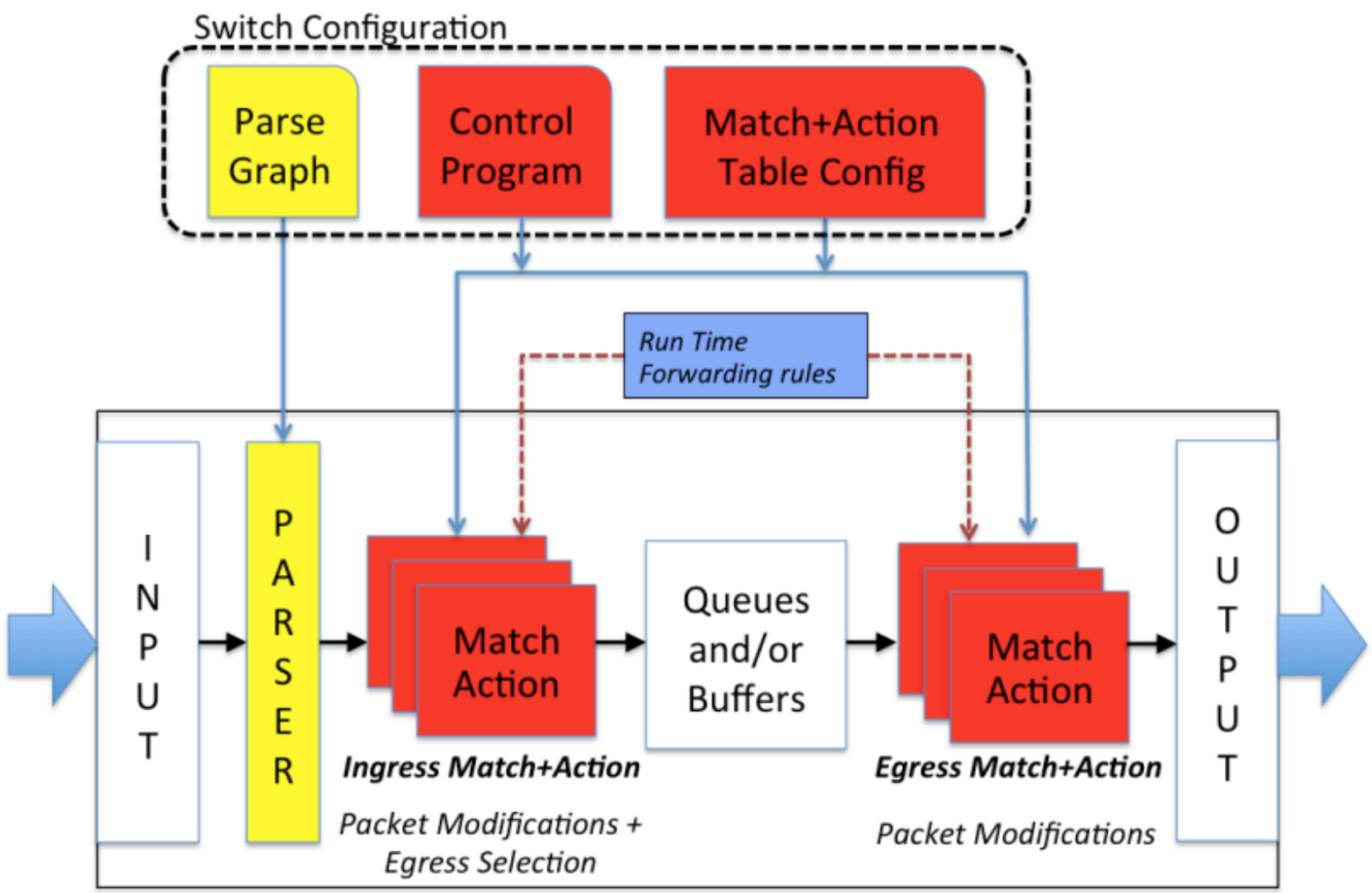
Switching logic

crossbar, shared buffers, …

Deparser

We'll rely on a simple P4$_{16}$ switch architecture (v1model) which is roughly equivalent to "PISA"

v1model/
simple switch



source https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf

Each architecture defines the metadata it supports, including both standard and intrinsic ones

v1model

```
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;
    error parser_error;
    bit<48> ingress_global_timestamp;
    bit<48> egress_global_timestamp;
    bit<32> lf_field_list;
    bit<16> mcast_grp;
    bit<32> resubmit_flag;
    bit<16> egress_rid;
    bit<1>  checksum_error;
    bit<32> recirculate_flag;
}
```

more info https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

Each architecture also defines a list of "externs",
i.e. blackbox functions whose interface is known

Most targets contain specialized components

which cannot be expressed in P4 (e.g. complex computations)

At the same time, $P4_{16}$ should be target-independent

In $P4_{14}$ almost 1/3 of the constructs were target-dependent

Think of externs as Java interfaces

only the signature is known, not the implementation

v1model

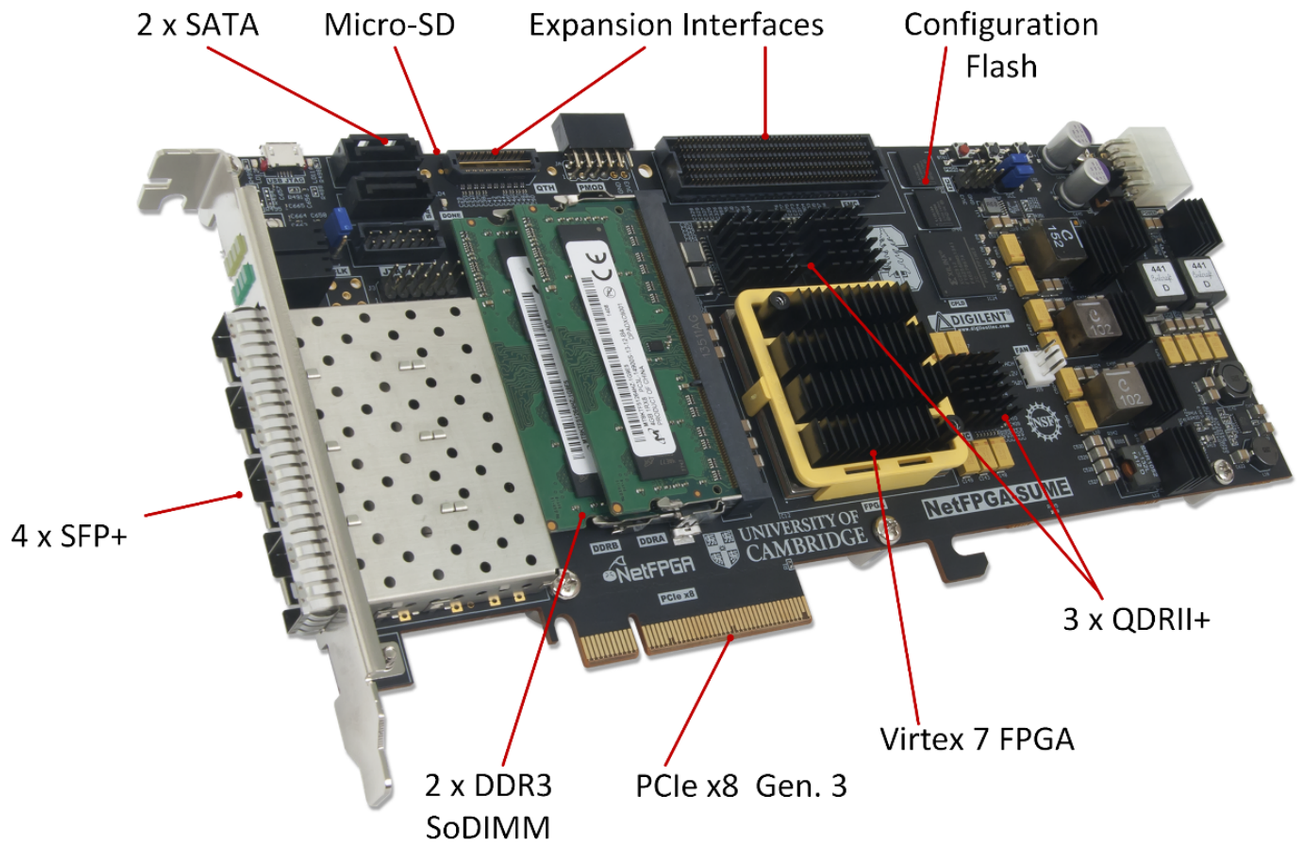
```
extern register<T> {
    register(bit<32> size);
    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}

extern void random<T>(out T result, in T lo, in T hi);

extern void hash<O, T, D, M>(out O result,
    in HashAlgorithm algo, in T base, in D data, in M max);

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

+ many others (see below)

more info https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

≠ architectures → ≠ metadata & ≠ externs

NetFPGA-SUME



more info http://isfpga.org/fpga2018/slides/FPGA-2018-P4-tutorial.pdf

Standard Metadata in SimpleSumeSwitch Architecture

```
/* standard sume switch metadata */
struct sume_metadata_t {
    bit<16> dma_q_size;
    bit<16> nf3_q_size;
    bit<16> nf2_q_size;
    bit<16> nf1_q_size;
    bit<16> nf0_q_size;
    bit<8> send_dig_to_cpu; // send digest_data to CPU
    bit<8> dst_port; // one-hot encoded
    bit<8> src_port; // one-hot encoded
    bit<16> pkt_len; // unsigned int
}
```

- `*_q_size` – size of each output queue, measured in terms of 32-byte words, when packet starts being processed by the P4 program
- `src_port`/`dst_port` – one-hot encoded, easy to do multicast
- `user_metadata`/`digest_data` – structs defined by the user

more info http://isfpga.org/fpga2018/slides/FPGA-2018-P4-tutorial.pdf

P4 environment

P4 language

stateful objects

Deeper dive into
the language constructs (*)

(*) full info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

Recap

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

Libraries

```
const bit<16> TYPE_IPV4 = 0x800;
typedef bit<32> ip4Addr_t;
header ipv4_t {…}
struct headers {…}
```

Declarations

```
parser MyParser(…) {
    state start {…}
    state parse_ethernet {…}
    state parse_ipv4 {…}
}
```

Parse packet headers

```
control MyIngress(…) {

    action ipv4_forward(…) {…}

    table ipv4_lpm {…}

    apply {
        if (…) {…}
    }
}
```

Control flow
to modify packet

```
control MyDeparser(…) {…}
```

Assemble
modified packet

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

“main()”

But first, the basics:

data types, operations, and statements

P4$_{16}$ is a statically-typed language with

base types and operators to derive composed ones

`bool`	Boolean value
`bit<W>`	Bit-string of width W
`int<W>`	Signed integer of width W
`varbit<W>`	Bit-string of dynamic length ≤W
`match_kind`	describes ways to match table keys
`error`	used to signal errors
`void`	no values, used in few restricted circumstances
`float`	not supported
`string`	not supported

P4$_{16}$ is a statically-typed language with base types and operators to derive composed ones

Header

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

```
Ethernet_h
  ethernetHeader;
```

corresponding declaration

Think of a header as a struct in C containing the different fields plus a hidden "validity" field

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

Parsing a packet using `extract()` fills in the fields of the header from a network packet

A successful `extract()` sets to true the validity bit of the extracted header

P4$_{16}$ is a statically-typed language with base types and operators to derive composed ones

Header

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

Header stack

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

Mpls_h[10] mpls;
```

Array of up to 10 MPLS headers

Header union

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

Either IPv4 or IPv6 header is present

only one alternative

P4$_{16}$ is a statically-typed language with
base types and operators to derive composed ones

Struct

Unordered collection
of named members

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

Tuple

Collection
of unnamed members

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

P4$_{16}$ is a statically-typed language with
base types and operators to derive composed ones

- enum `enum Priority {High, Low}`
- type specialization `typedef bit<48> macAddr_t;`
- extern …
- parser …
- control …
- package …

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

P4 operations are similar to C operations and vary depending on the types (unsigned/signed ints, …)

- arithmetic operations `+, -, *`
- logical operations `~, &, |, ^, >>, <<`
- non-standard operations `[m:l]` Bit-slicing

  `++` Bit concatenation

✘ *no* division and modulo (can be approximated)

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

Constants, variable declarations and instantiations are pretty much the same as in C too

Variable

```
bit<8> x = 123;
```

```
typedef bit<8> MyType;
MyType x;
x = 123;
```

Constant

```
const bit<8> X = 123;
```

```
typedef bit<8> MyType;
const MyType x = 123;
```

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

Variables have local scope and their values is not maintained across subsequent invocations

important

variables *cannot* be used to maintain state between different network packets

instead

to maintain state

you can only use two stateful constructs

- tables — modified by control plane
- extern objects — modified by control plane & data plane

more on this soon

P4 statements are pretty classical too

Restrictions apply depending on the statement location

`return` terminates the execution of the current control block, and returns to the caller

`exit` terminates the execution of the current control block, and of all the enclosing callers

Conditions

```
if (x==123) {…} else {…}
```

not in parsers

Switch

```
switch (t.apply().action_run) {
   action1: { ...}
   action2: { ...}
}
```

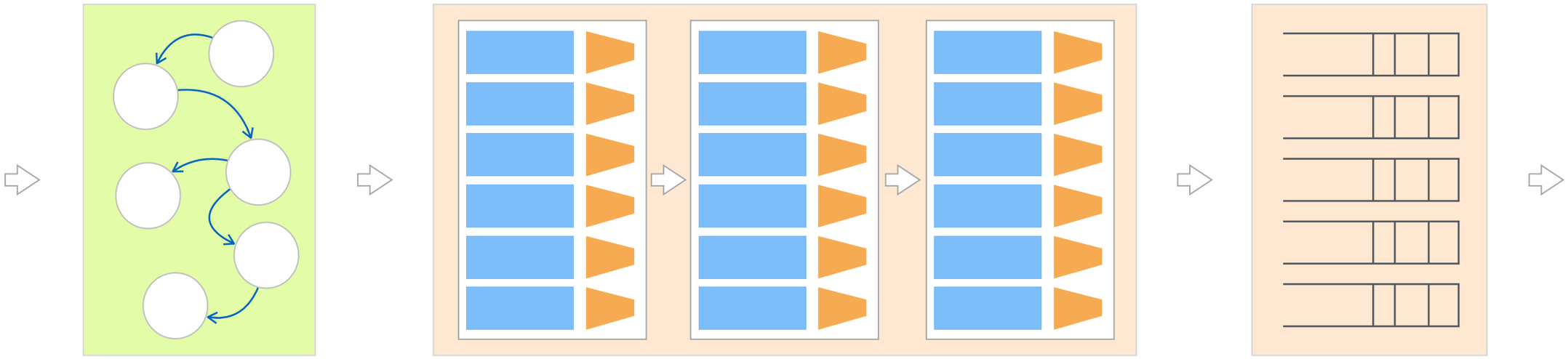
only in control blocks

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

Parser

Match-Action Pipeline

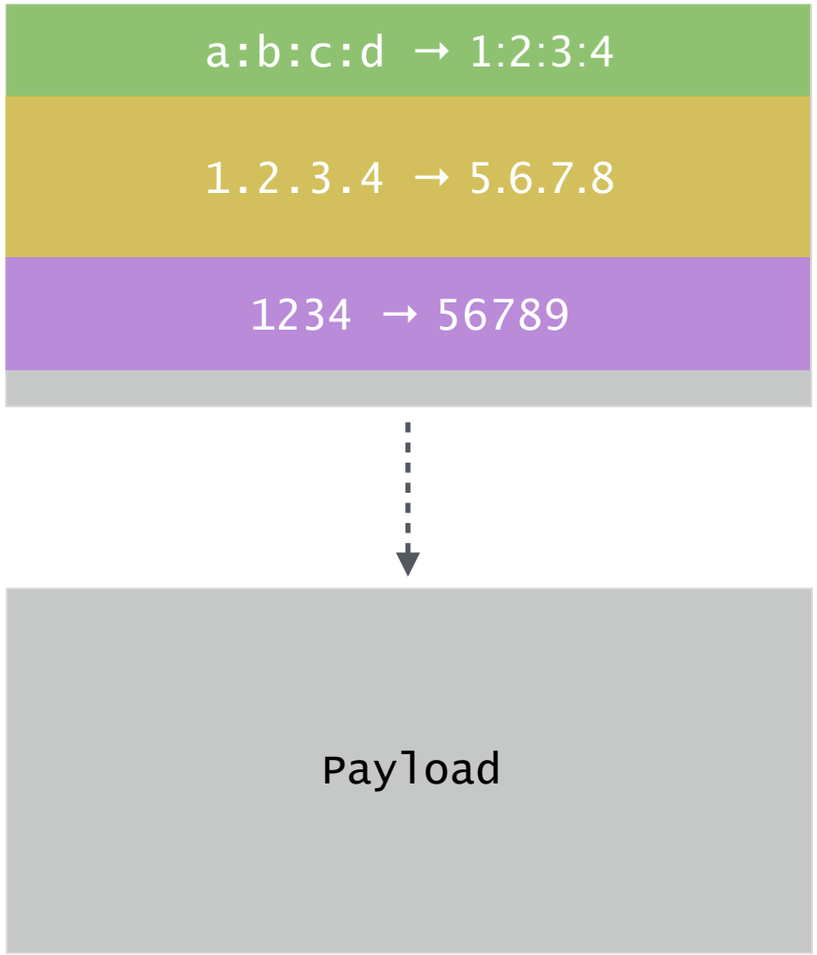
Deparser

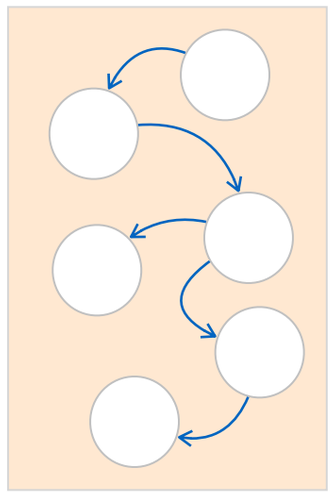


Recap

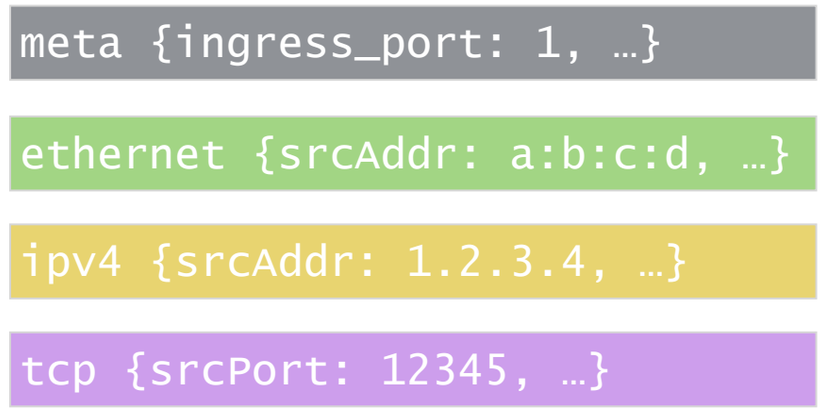
The parser uses a state machine to map packets into headers and metadata

Packet

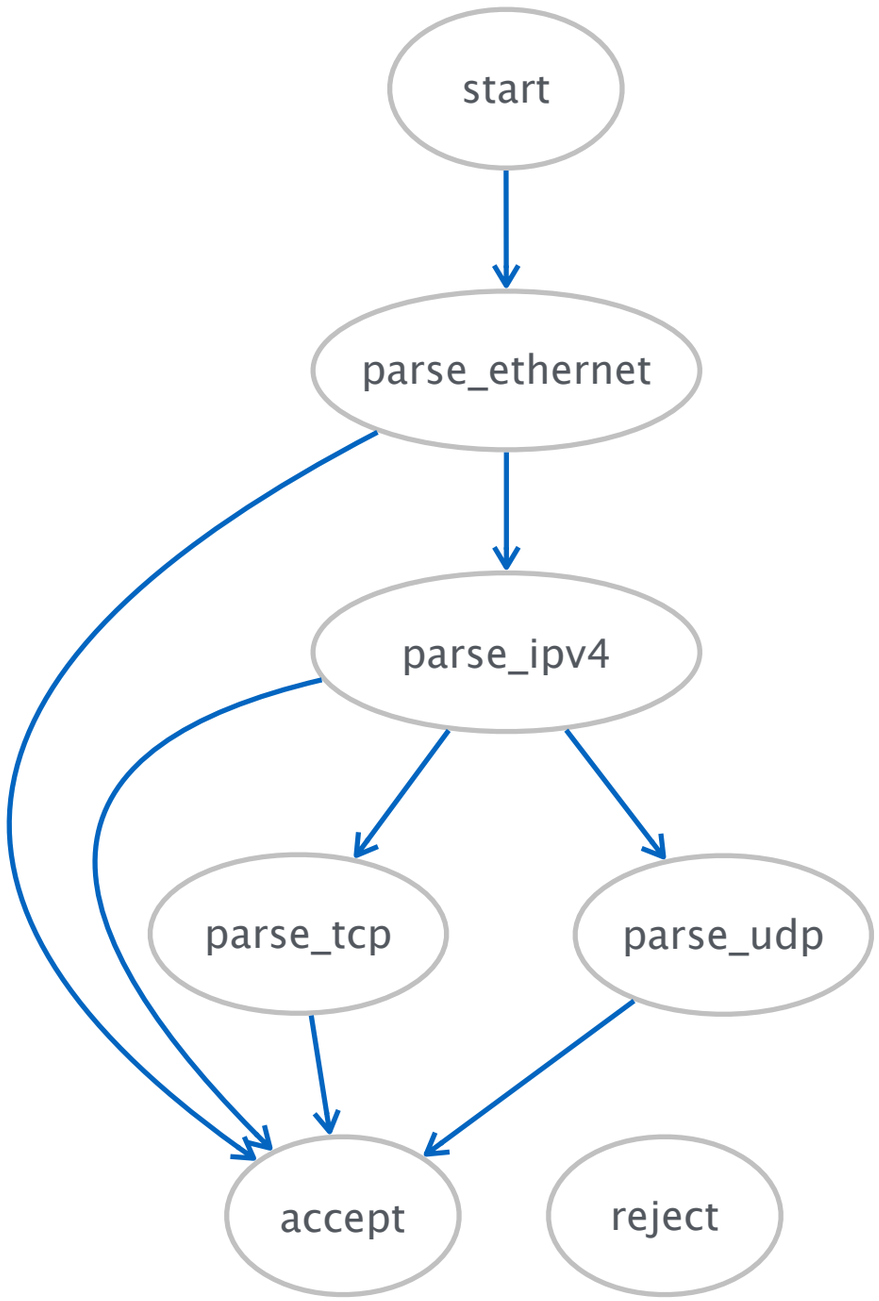




Headers and metadata



Recap




```
parser MyParser(…) {

 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 {
  packet.extract(hdr.ipv4);
  transition select(hdr.ipv4.protocol) {
   6: parse_tcp;
   17: parse_udp;
   default: accept;
  }
 }

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

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

}
```

The last statement in a state is an (optional) transition, which transfers control to another state (inc. accept/reject)

```
state start {
 transition parse_ethernet;
}
```

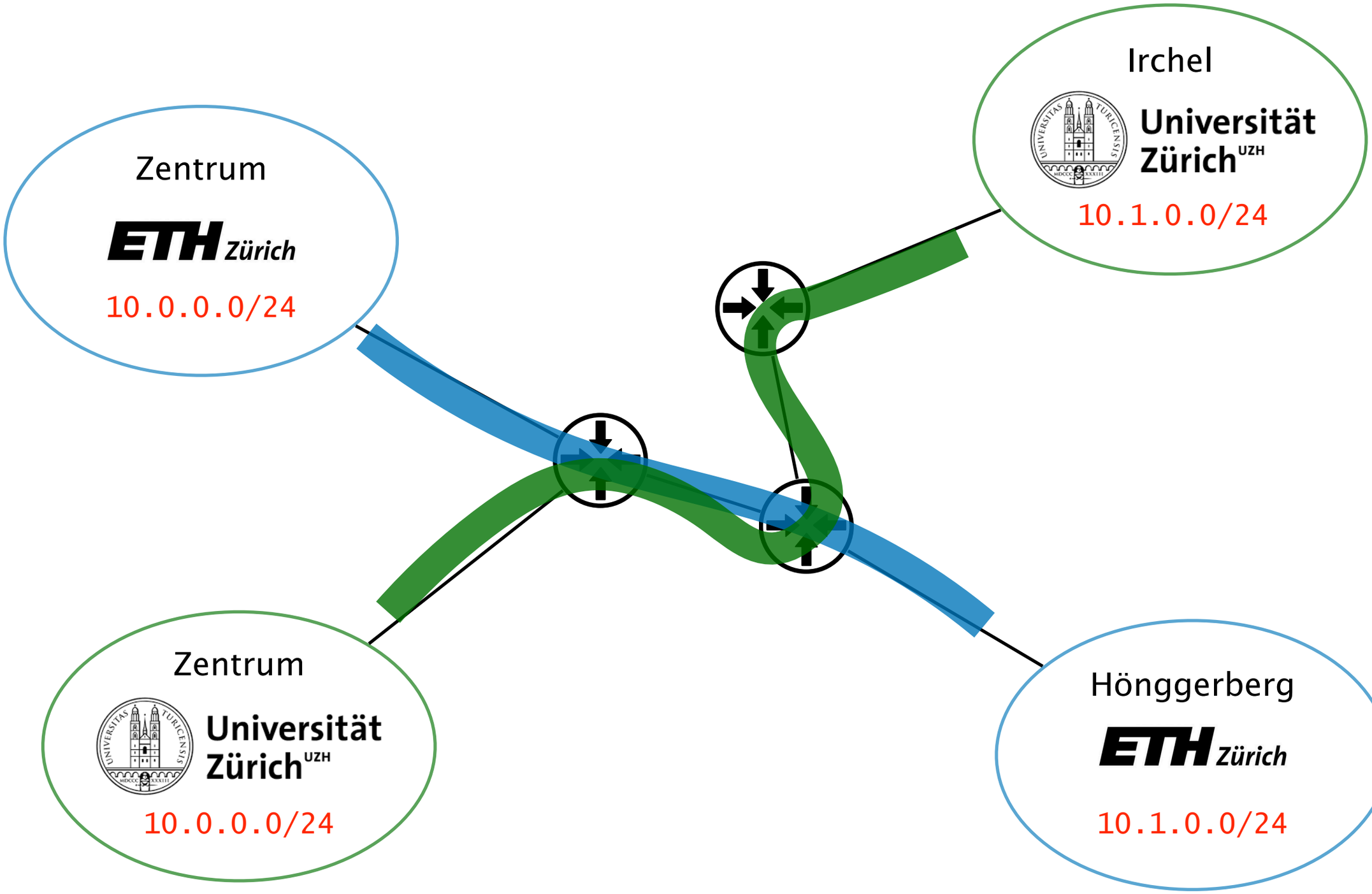
Go directly to parse_ethernet

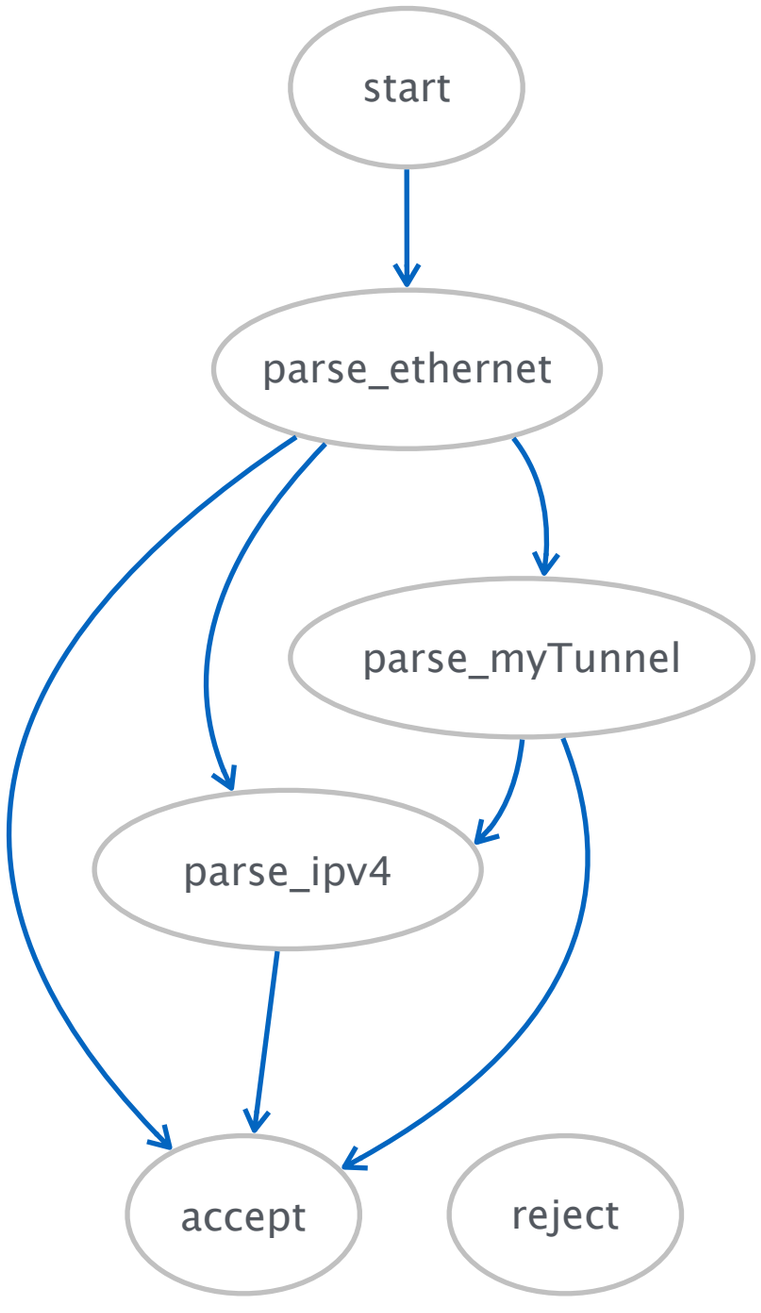
```
state parse_ethernet {
 packet.extract(hdr.ethernet);
 transition select(hdr.ethernet.etherType) {
  0x800: parse_ipv4;
  default: accept;
 }
}
```

Next state depends on etherType

Defining (and parsing) custom headers allow you to implement your own protocols

A simple example for tunneling






```
header myTunnel_t {
    bit<16> proto_id;
    bit<16> dst_id;
}

struct headers {
    ethernet_t   ethernet;
    myTunnel_t   myTunnel;
    ipv4_t       ipv4;
}

parser MyParser(…) {

 state start {…}

 state parse_ethernet {
  packet.extract(hdr.ethernet);
  transition select(hdr.ethernet.etherType) {
   0x1212: parse_myTunnel;
   0x800: parse_ipv4;
   default: accept;
  }
 }

 state parse_myTunnel {
  packet.extract(hdr.myTunnel);
  transition select(hdr.myTunnel.proto_id) {
   TYPE_IPV4: parse_ipv4;
   default: accept;
  }
 }

 state parse_ipv4 {…}
}
```

P4 parser supports both fixed and variable-width header extraction

```
header IPv4_no_options_h {
 ...
 bit<32>  srcAddr;
 bit<32>  dstAddr;
}

header IPv4_options_h {
 varbit<320> options;
}

...
parser MyParser(…) {
 ...
 state parse_ipv4 {
  packet.extract(headers.ipv4);
  transition select (headers.ipv4.ihl) {
   5: dispatch_on_protocol;
   default: parse_ipv4_options;
  }

 state parse_ipv4_options {
  packet.extract(headers.ipv4options,
       (bit<32>)(8w8*((bit<8>)headers.ipv4.ihl * 8w4 - 8w20)));
  transition dispatch_on_protocol;
 }
}
```

Fixed width fields

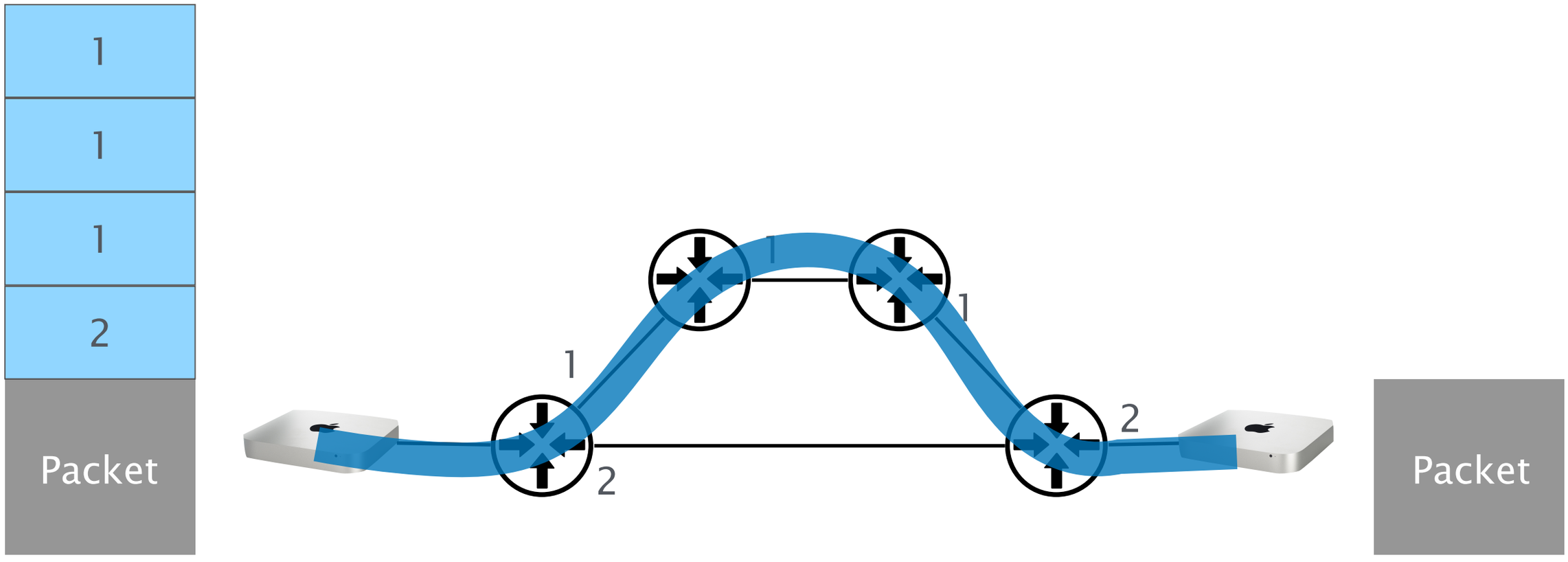
Variable width field

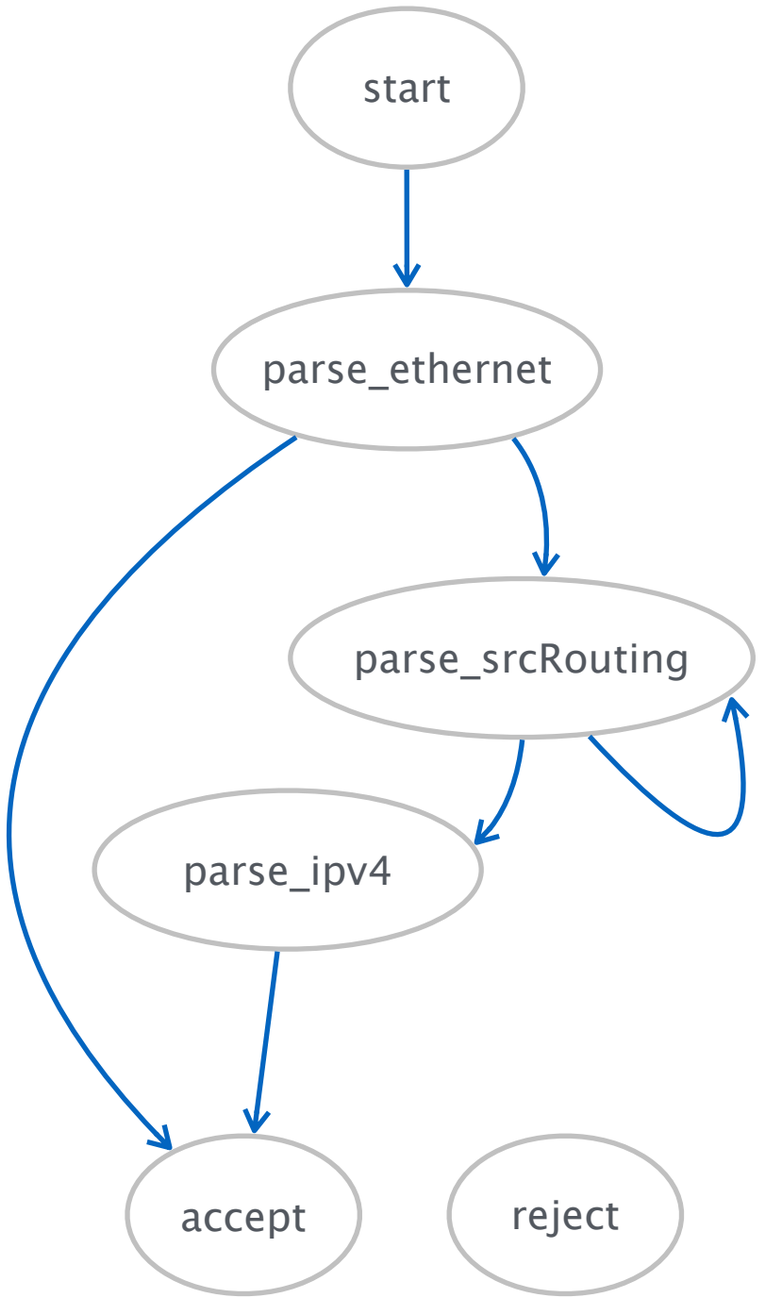
ihl determines length of options field

Parsing a header stack requires the parser to loop

the only “loops” that are possible in P4

Header stacks
for source routing






```
header srcRoute_t {
 bit<1>    bos;
 bit<15>   port;
}

struct headers {
 ethernet_t                ethernet;
 srcRoute_t[MAX_HOPS]      srcRoutes;
 ipv4_t                    ipv4;
}

parser MyParser(...) {
 state parse_ethernet {
  packet.extract(hdr.ethernet);
  transition select(hdr.ethernet.etherType) {
   TYPE_SRCROUTING: parse_srcRouting;
   default: accept;
  }
 }

 state parse_srcRouting {
  packet.extract(hdr.srcRoutes.next);
  transition select(hdr.srcRoutes.last.bos) {
   1: parse_ipv4;
   default: parse_srcRouting;
  }
 }
}
```

The parser contains more advanced concepts

check them out!

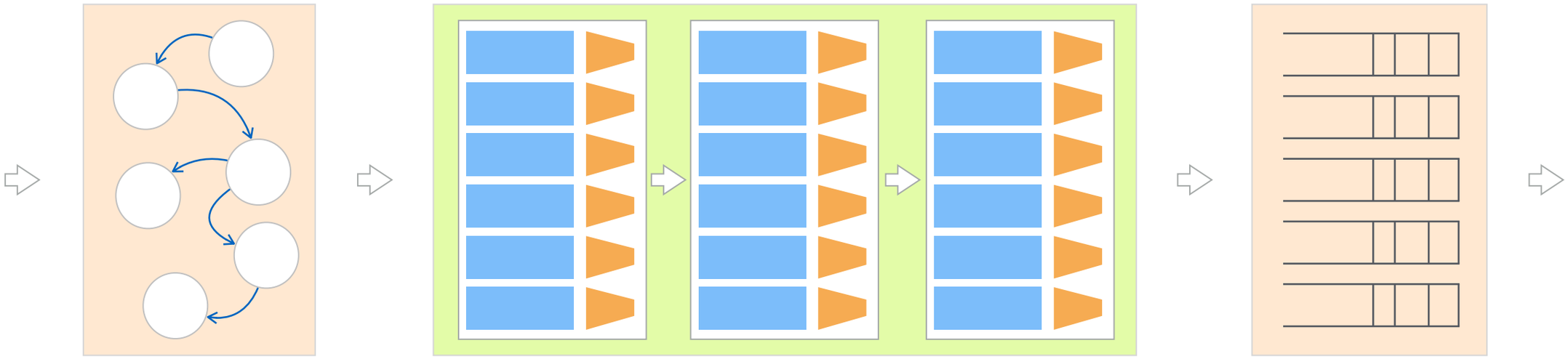
- verify — error handling in the parser
- lookahead — access bits that are not parsed yet
- sub-parsers — like subroutines

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

Parser

Match-Action Pipeline

Deparser



Recap

Control

Tables

match a key and return an action

Actions

similar to functions in C

Control flow

similar to C but without loops

Control

Tables

match a key and return an action

Actions

similar to functions in C

Control flow

similar to C but without loops

Tables can match on one or multiple keys in different ways

```
table Fwd {
    key = {
        hdr.ipv4.dstAddr : ternary;
        hdr.ipv4.version : exact;
    }
    ...
}
```

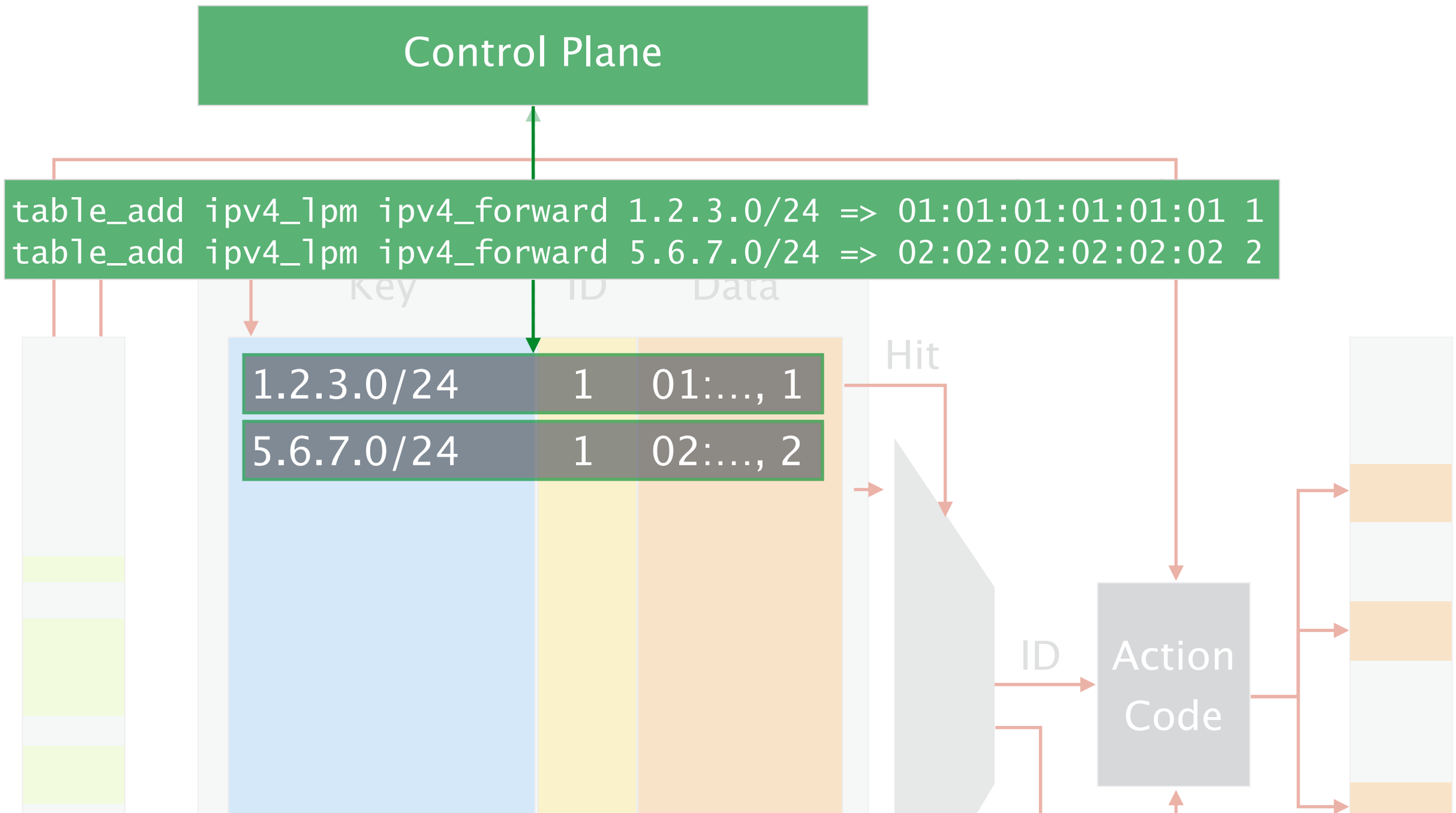
Fields to match

Match kind

Match types are specified in the P4 core library and in the architectures

`exact`	exact comparison `0x01020304`	core.p4
`ternary`	compare with mask `0x01020304 & 0x0F0F0F0F`	
`lpm`	longest prefix match `0x01020304/24`	
`range`	check if in range `0x01020304 - 0x010203FF`	v1model.p4
`...`		
`...`		other architecture

Table entries are added through the control plane



Control

Tables

match a key and return an action

Actions

similar to functions in C

Control flow

similar to C but without loops

Actions are blocks of statements that possibly modify the packets

Actions usually take directional parameters indicating how the corresponding value is treated within the block

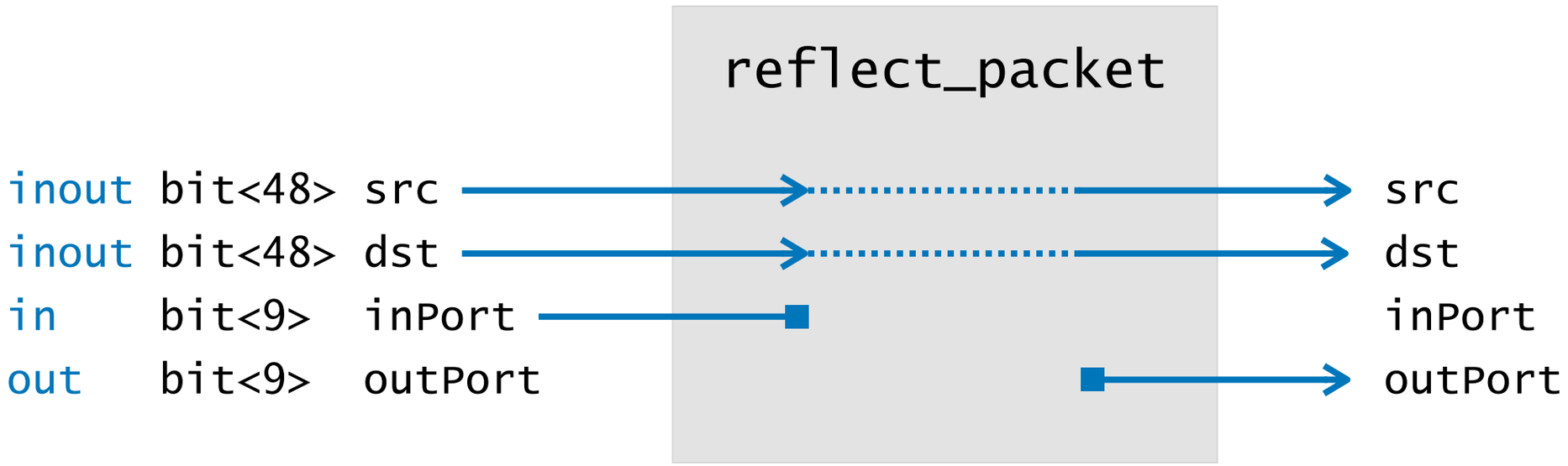
Directions can be of three types

`in`	read-only inside the action like parameters to a function
`out`	uninitialized, write inside the action like return values
`inout`	combination of `in` and `out` like “call by reference”

Let's reconsider a known example

```
action reflect_packet( inout bit<48> src,
                       inout bit<48> dst,
                       in bit<9> inPort,
                       out bit<9> outPort
                     ) {
 bit<48> tmp = src;
 src = dst;
 dst = tmp;
 outPort = inPort;
}


reflect_packet( hdr.ethernet.srcAddr,
                hdr.ethernet.dstAddr,
                standard_metadata.ingress_port,
                standard_metadata.egress_spec
              );
```

Parameter
with direction


reflect_packet
inout bit<48> src
inout bit<48> dst
in bit<9> inPort
out bit<9> outPort
src
dst
inPort
outPort

Actions parameters resulting from a table lookup do not take a direction as they come from the control plane

Parameter
without direction

```
action set_egress_port(bit<9> port) {
 standard_metadata.egress_spec = port;
}
```

Control

Tables

match a key and return an action

Actions

similar to functions in C

Control flow

similar to C but without loops

Interacting with tables from the control flow

- Applying a table

```
ipv4_lpm.apply()
```

- Checking if there was a hit

```
if (ipv4_lpm.apply().hit) {...}
else {...}
```

- Check which action was executed

```
switch (ipv4_lpm.apply().action_run) {
    ipv4_forward: { ... }
}
```

Validating and computing checksums

```
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
                                 );
```

v1model.p4

Re-computing checksums

(e.g. after modifying the IP header)

```
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);
 }
}
```

pre-condition

fields list

checksum field

algorithm

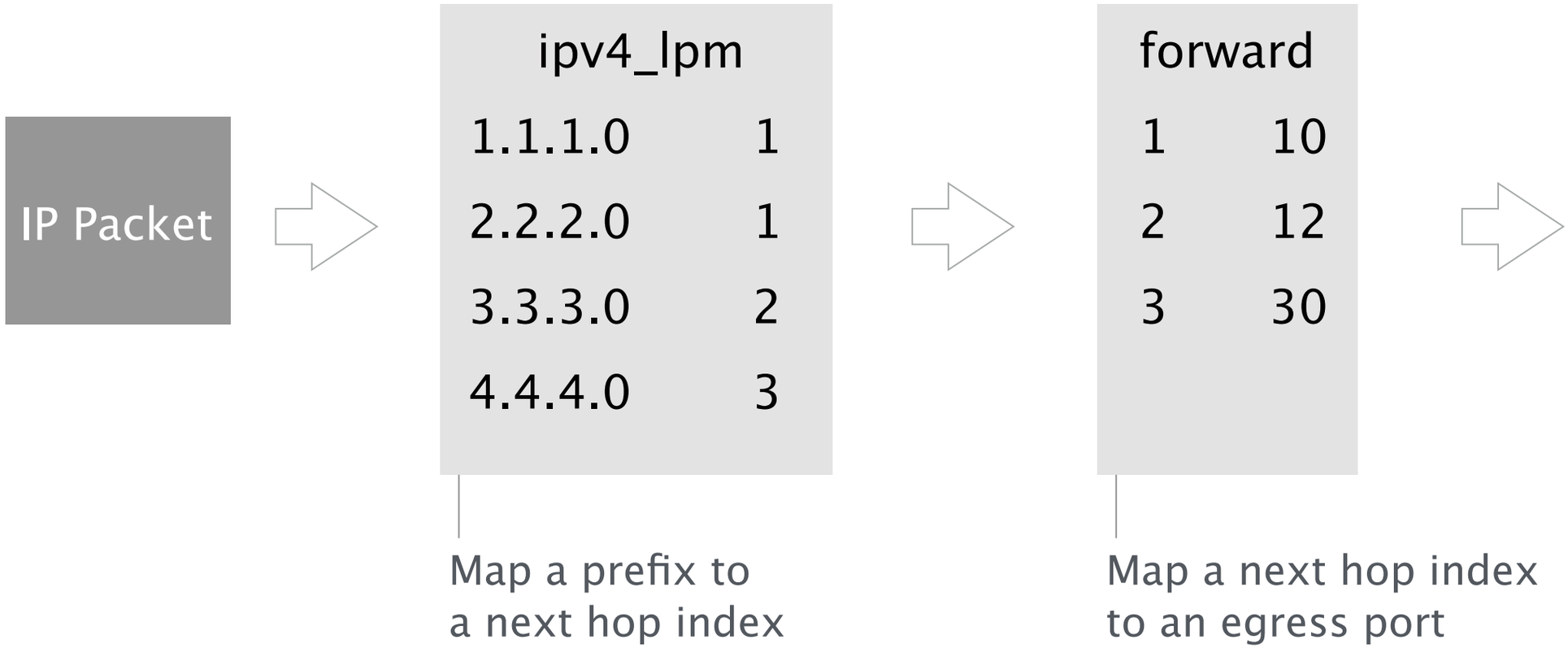
Control flows contain more advanced concepts

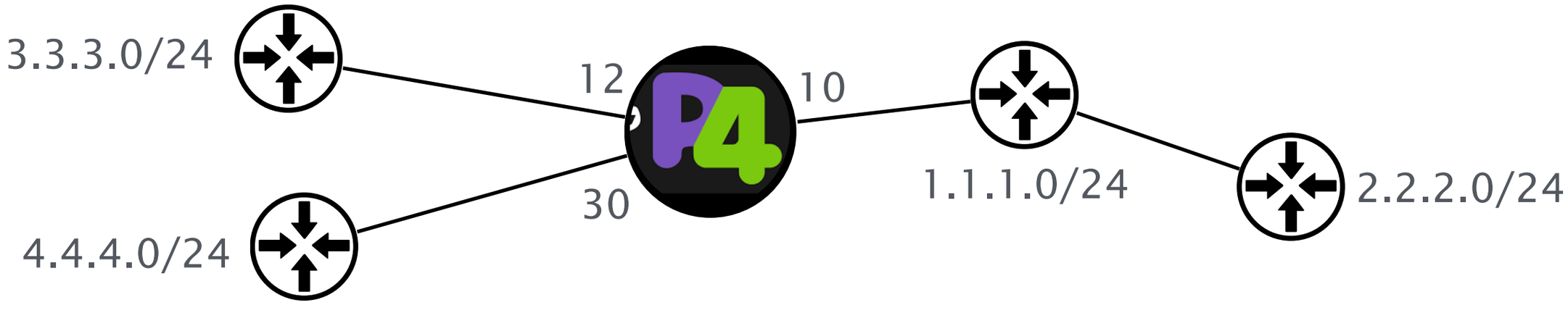
check them out!

- cloning packets — create a clone of a packet
- sending packets to control plane — using dedicated Ethernet port, or target-specific mechanisms (e.g. digests)
- recirculating — send packet through pipeline multiple times

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

Example: L3 forwarding with multiple tables





Example 1: L3 forwarding with multiple tables

```
action set_nhop_index(bit<8> index) {
 meta.nhop_index = index;
}

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

```
table forward {
 key = {
  meta.nhop_index: exact;
 }
 actions = {
  _forward;
  NoAction;
 }

 size = 64;
 default_action = NoAction();
}
```

Applying multiple tables in sequence

and checking whether there was a hit

```
control MyIngress(...) {
 action drop() {...}
 action set_nhop_index(...}
 action _forward(...}
 table ipv4_lpm {...}
 table forward {...}
 apply {
  if (hdr.ipv4.isValid()){
   if (ipv4_lpm.apply().hit) {
    forward.apply();
   }
  }
 }
}
```

Check if IPv4 packet

Apply `ipv4_lpm` table and check if there was a hit

apply `forward` table

P4 environment

P4 language

stateful objects

How do we maintain state in P4?

Stateful objects in P4

- Table — managed by the control plane
- Register — store arbitrary data
- Counter — count events
- Meter — rate-limiting
- … — …

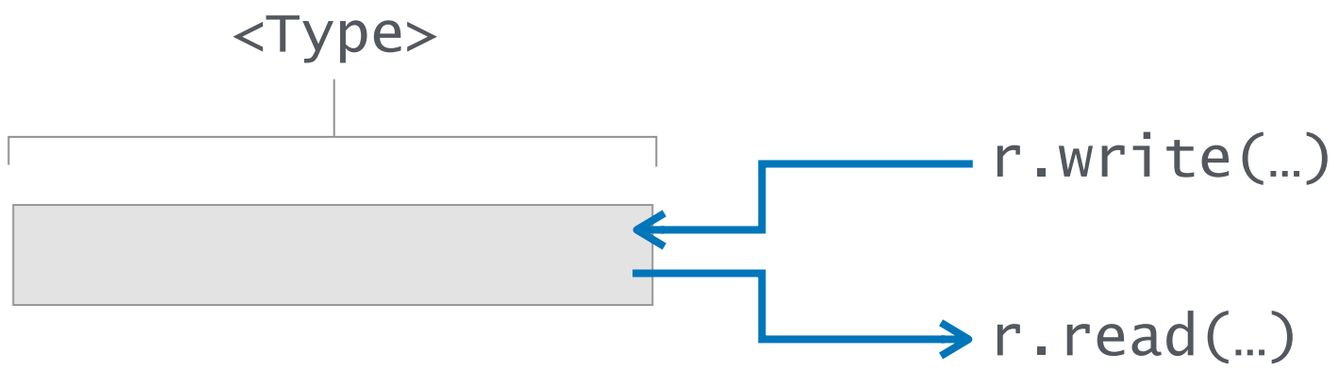
externs in v1model

Stateful objects in P4

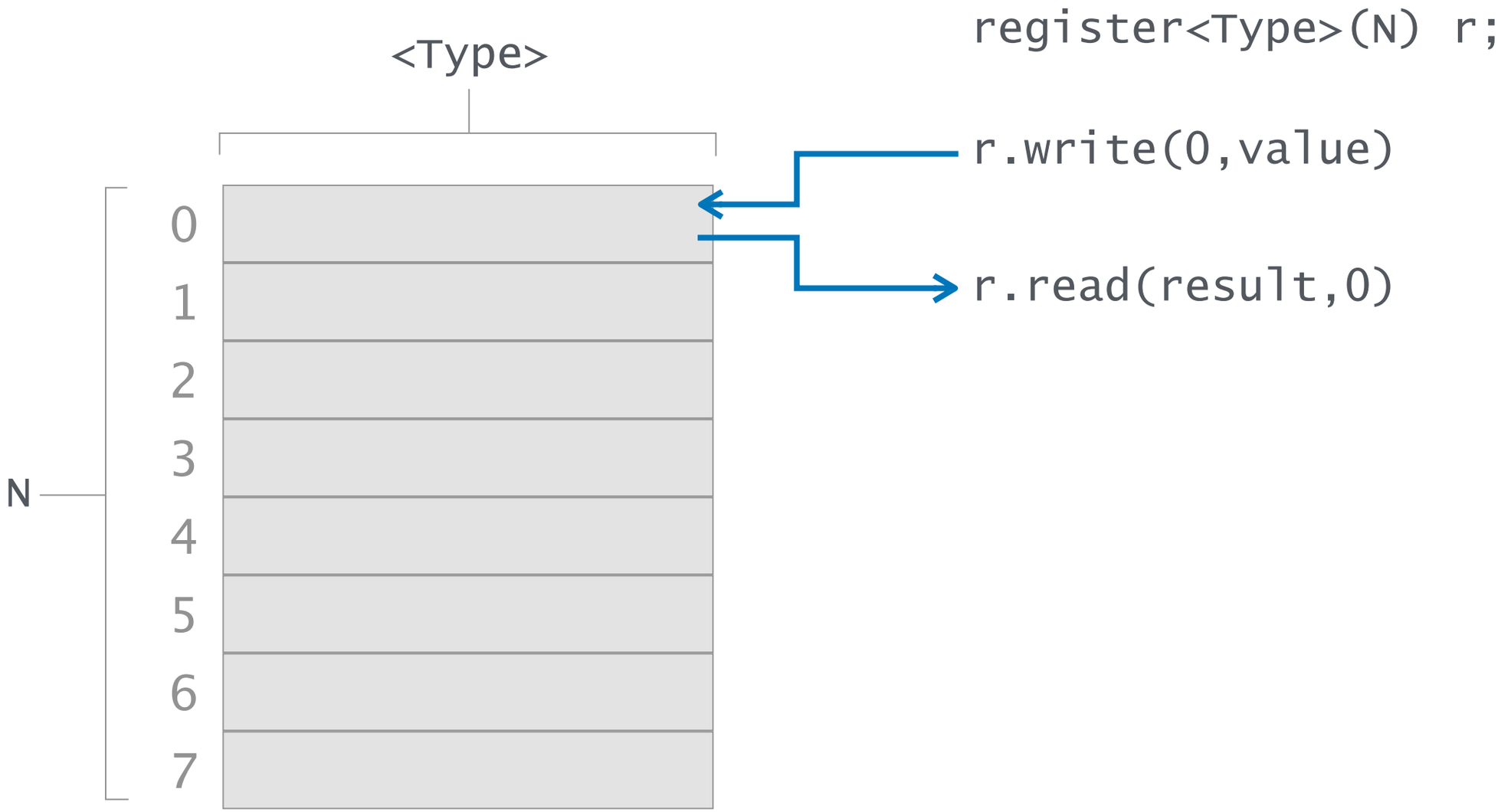
- Table — managed by the control plane
- Register — store arbitrary data
- Counter — count events
- Meter — rate-limiting
- … — …

externs in v1model

Registers are useful for storing
(small amounts of) arbitrary data



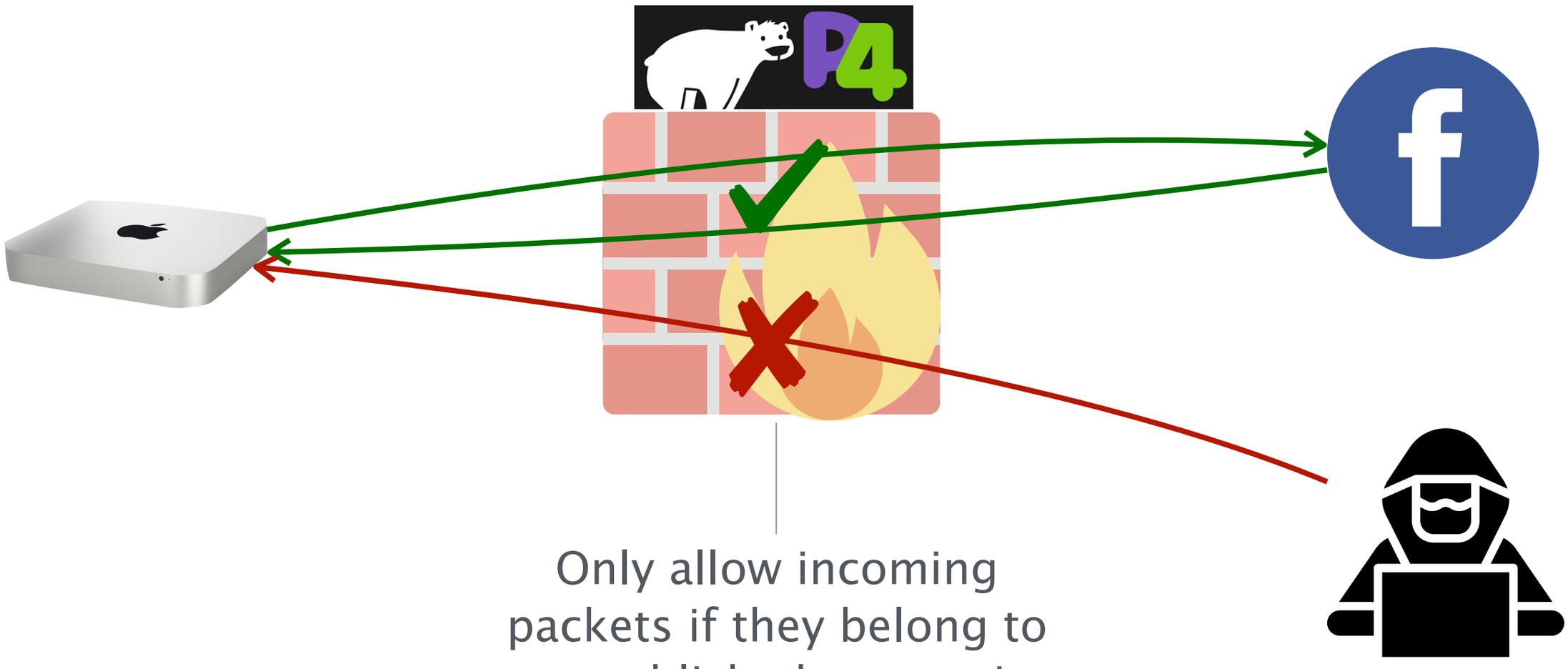
Registers are assigned in arrays



Example: Calculating inter packet gap

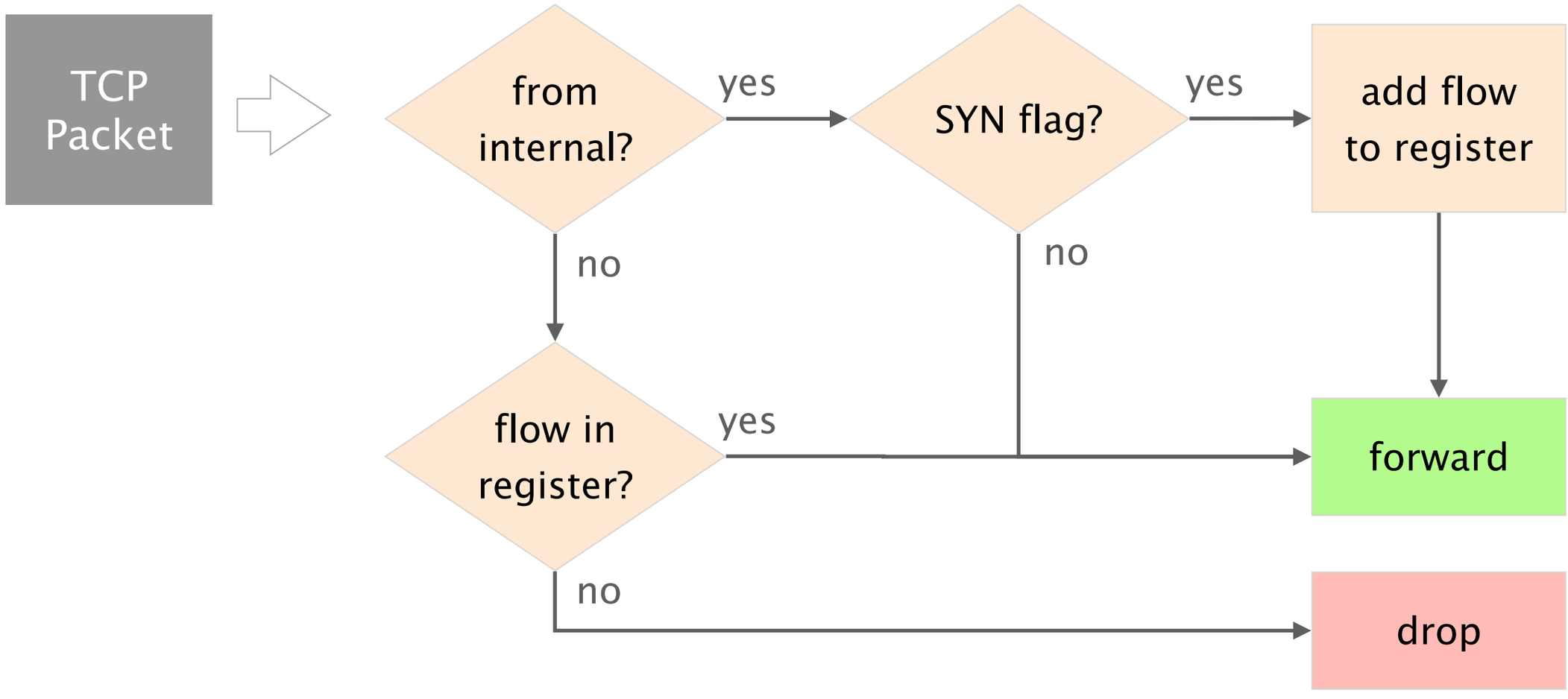
```
register<bit<48>>(16384) last_seen;

action get_inter_packet_gap(out bit<48> interval, bit<32> flow_id)
{
 bit<48> last_pkt_ts;

 /* Get the time the previous packet was seen */
 last_seen.read(last_pkt_ts, flow_id);

 /* Calculate the time interval */
 interval = standard_metadata.ingress_global_timestamp - last_pkt_ts;

 /* Update the register with the new timestamp */
 last_seen.write(flow_id, standard_metadata.ingress_global_timestamp);

 ...
}
```

Example: Stateful firewall



Only allow incoming packets if they belong to an established connection

Example: Stateful firewall



Example: Stateful firewall

```
control MyIngress(...) {
 register<bit<1>>(4096) known_flows;
 ...
 apply {
  meta.flow_id = ... // hash(5-tuple)
  if (hdr.ipv4.isValid()){
   if (hdr.tcp.isValid()){
    if (standard_metadata.ingress_port == 1){
     if (hdr.tcp.syn == 1){
      known_flows.write(meta.flow_id, 1);
     }
    }

    if (standard_metadata.ingress_port == 2){
     known_flows.read(meta.flow_is_known, meta.flow_id);
     if (meta.flow_is_known != 1){
      drop(); return;
     }
    }
   }
   ipv4_lpm.apply();
  }
 }
}
```

register to memorize established connections

add to register if it it is a SYN packet from internal

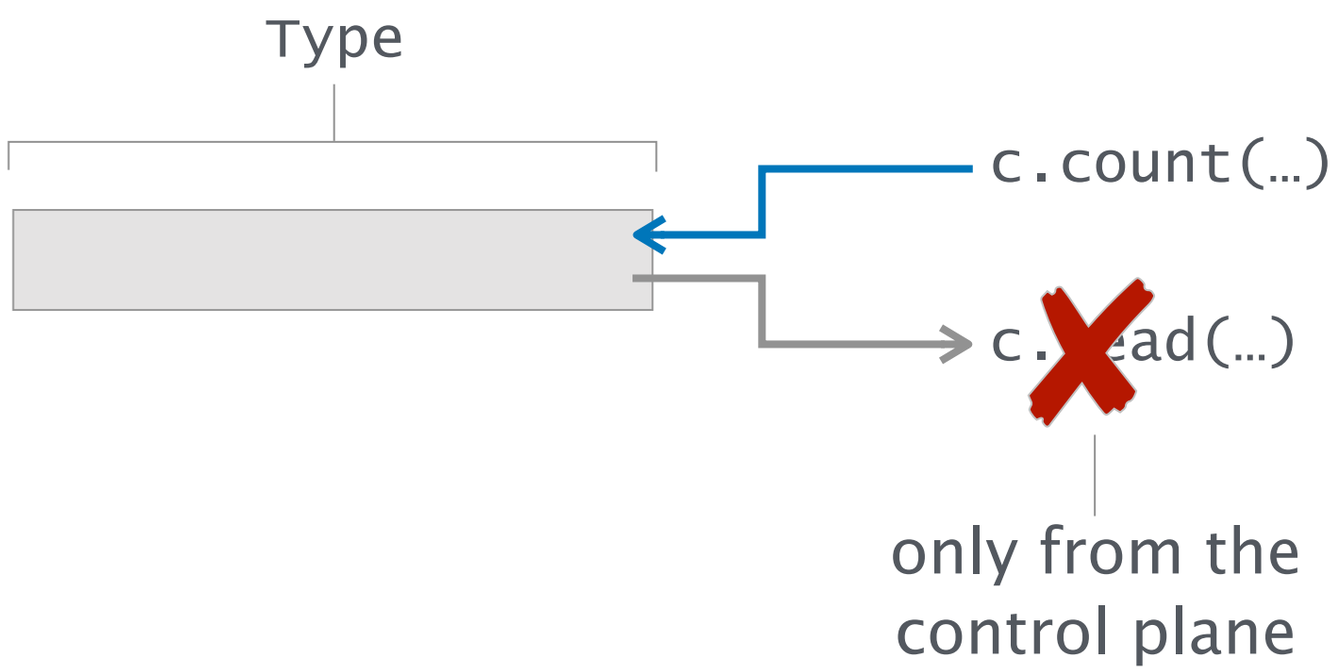
drop if the packet does not belong to a known flow and comes from outside

Stateful objects in P4

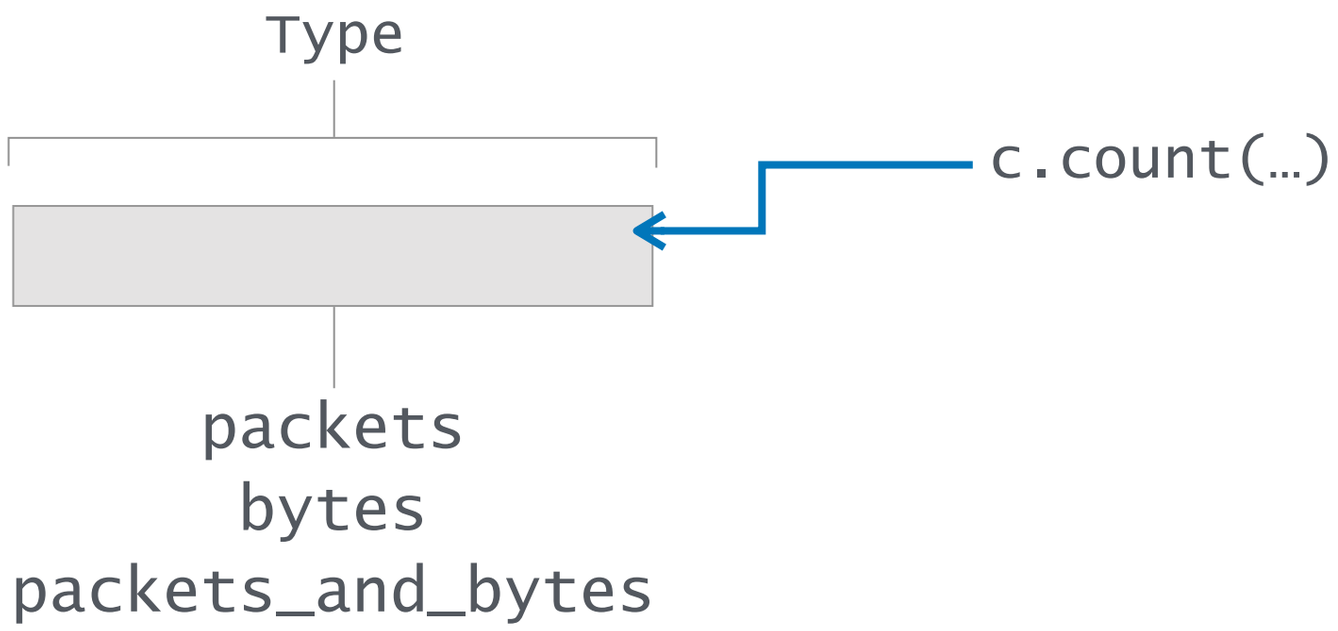
- Table — managed by the control plane
- Register — store arbitrary data
- Counter — count events
- Meter — rate-limiting
- … — …

externs in v1model

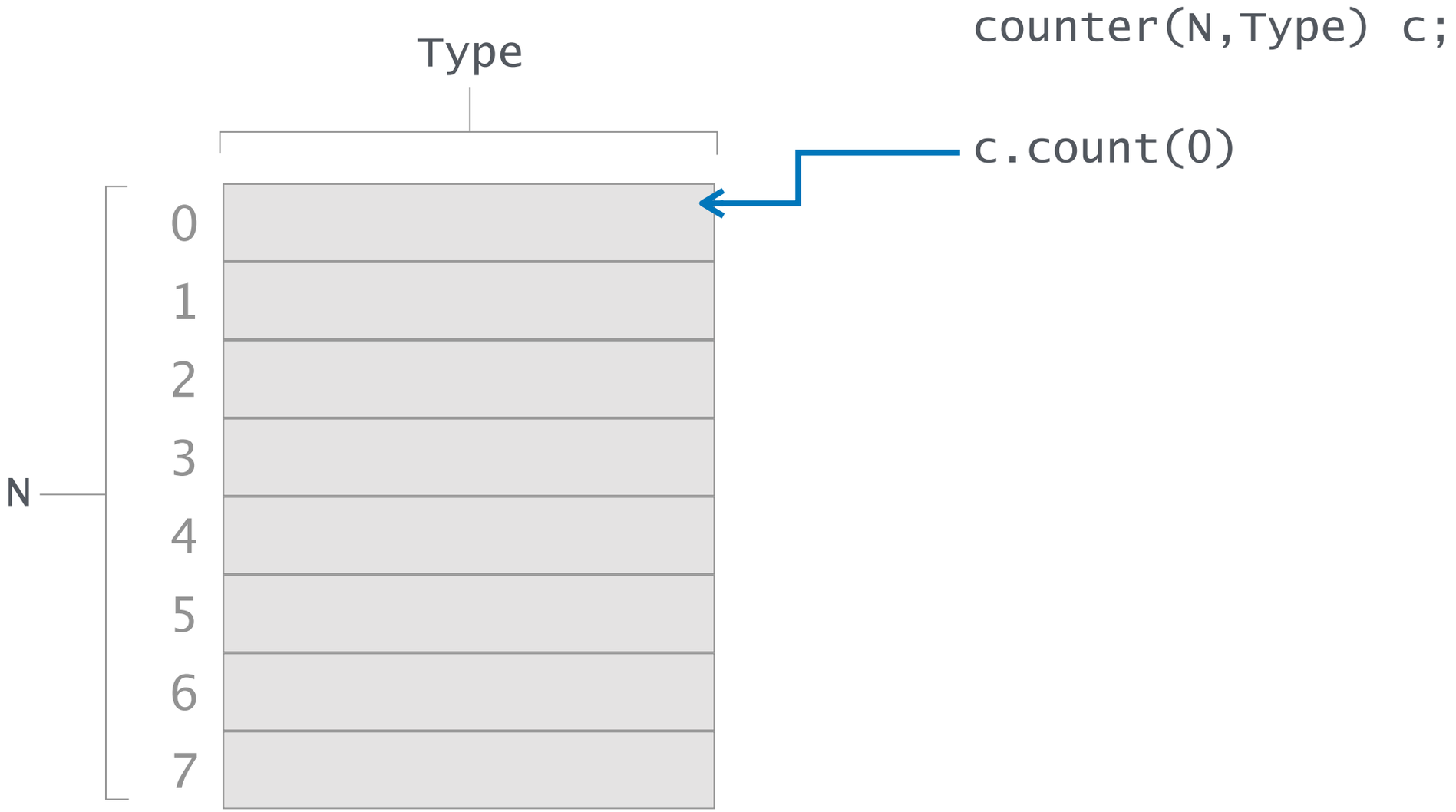
Counters are useful for… counting



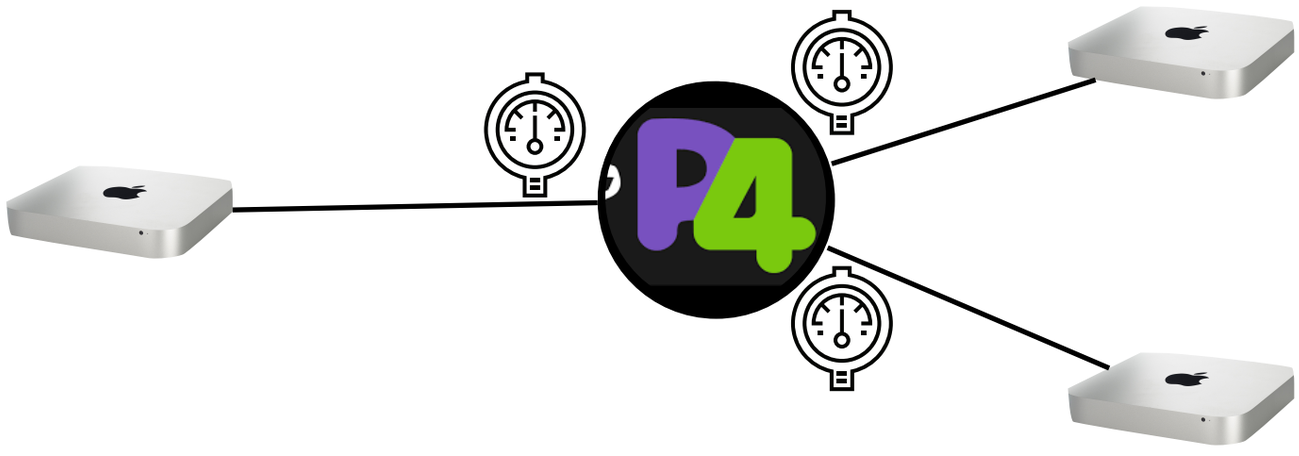
Counters can be of three different types



Like registers, counters are assigned in arrays



Example: Counting packets and bytes arriving at each port




```
control MyIngress(...) {

    counter(512, CounterType.packets_and_bytes) port_counter;

    apply {
        port_counter.count((bit<32>)standard_metadata.ingress_port);
    }
}
```

use the ingress port as counter index

Example: Reading the counter values from the control plane

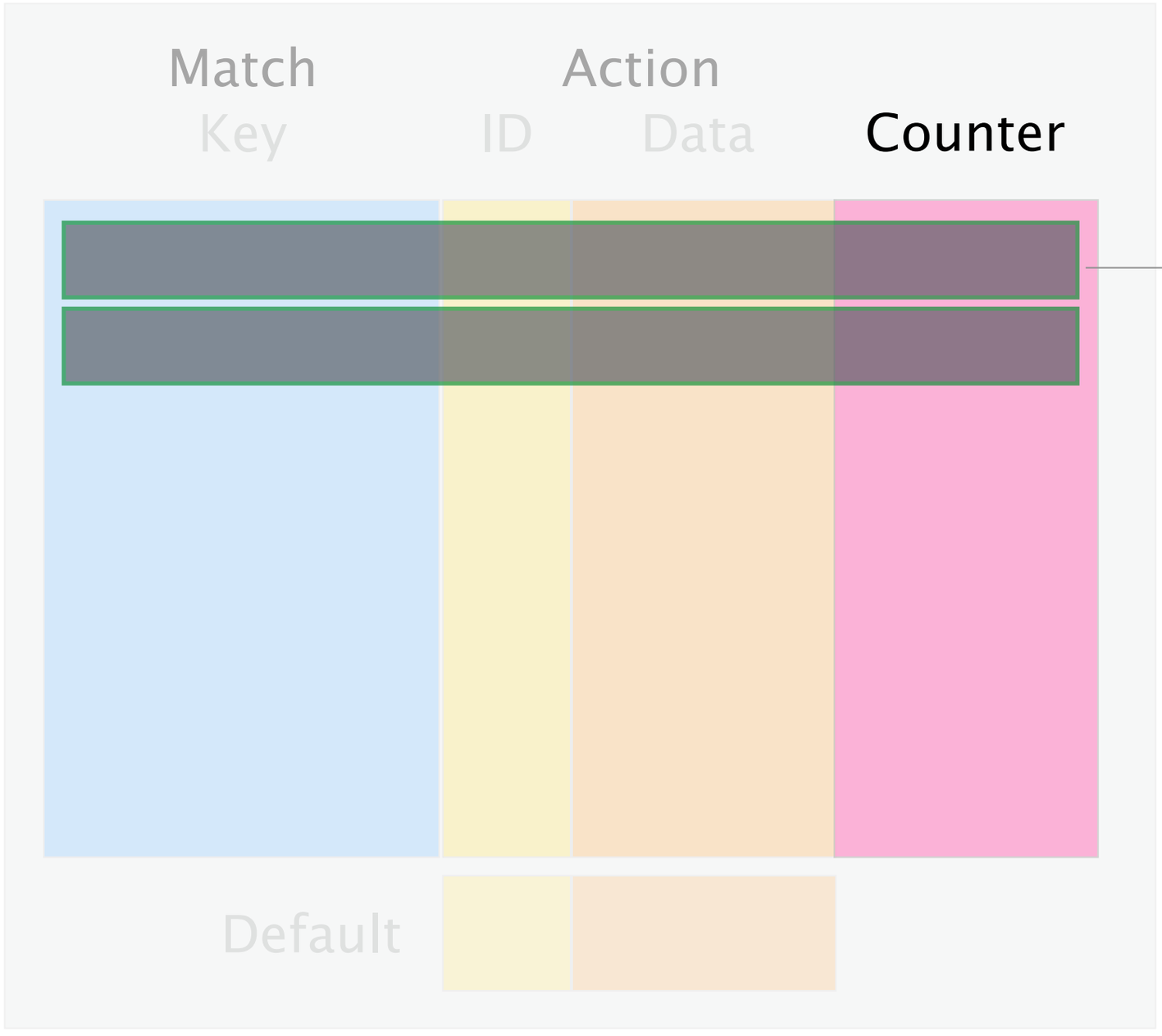
Control Plane

```
RuntimeCmd: counter_read MyIngress.port_counter 1
MyIngress.port_counter[1]=  BmCounterValue(packets=13, bytes=1150)
```

```
control MyIngress(...) {

    counter(512, CounterType.packets_and_bytes) port_counter;

    apply {
        port_counter.count((bit<32>)standard_metadata.ingress_port);
    }
}
```

use the ingress port as counter index

Direct counters are a special kind of counters that are attached to tables



Each entry has a counter cell that counts when the entry matches

Example: Counting packets and bytes arriving at each port *using a direct counter*

```
control MyIngress(...) {

 direct_counter(CounterType.packets_and_bytes) direct_port_counter;

 table count_table {
  key = {
   standard_metadata.ingress_port: exact;
  }
  actions = {
   NoAction;
  }
  default_action = NoAction;
  counters = direct_port_counter;
  size = 512;
 }

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

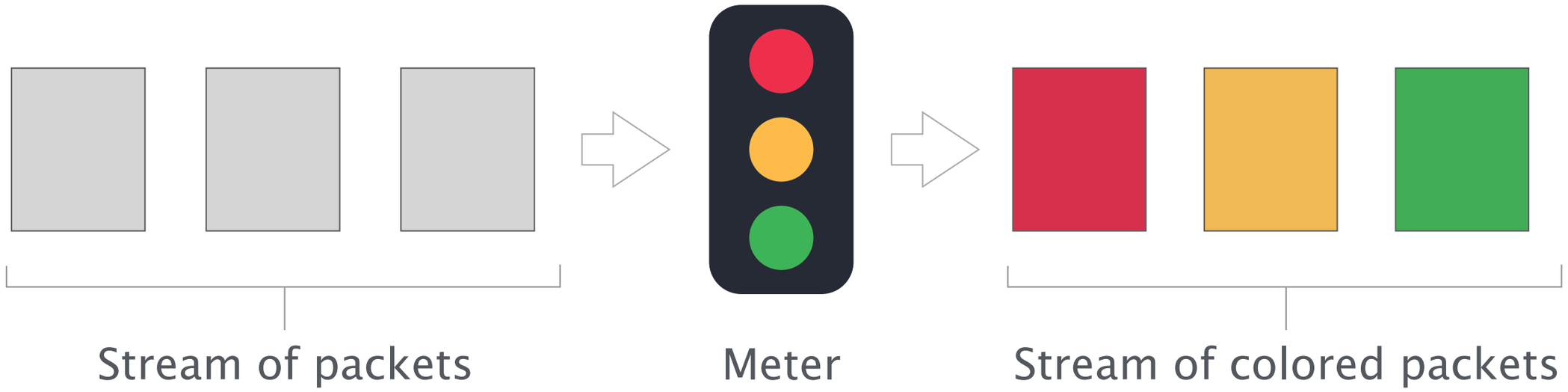
attach counter to table

Stateful objects in P4

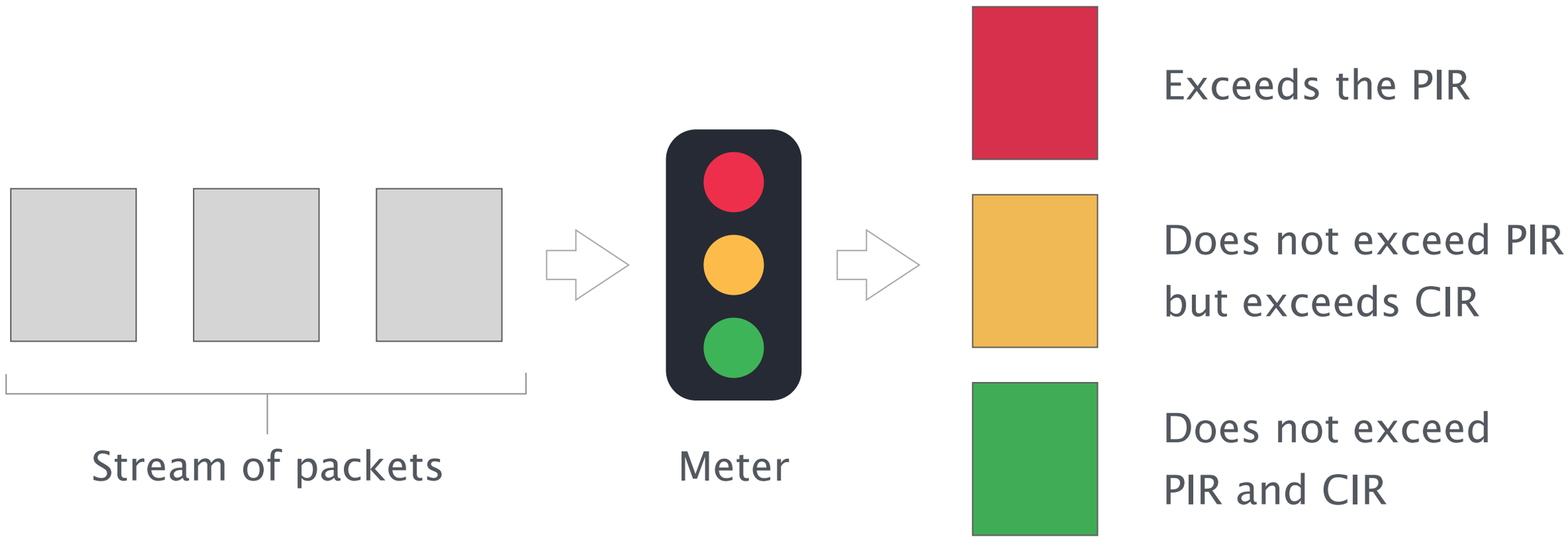
- Table — managed by the control plane
- Register — store arbitrary data
- Counter — count events
- Meter — rate-limiting
- … — …

externs in v1model

Meters



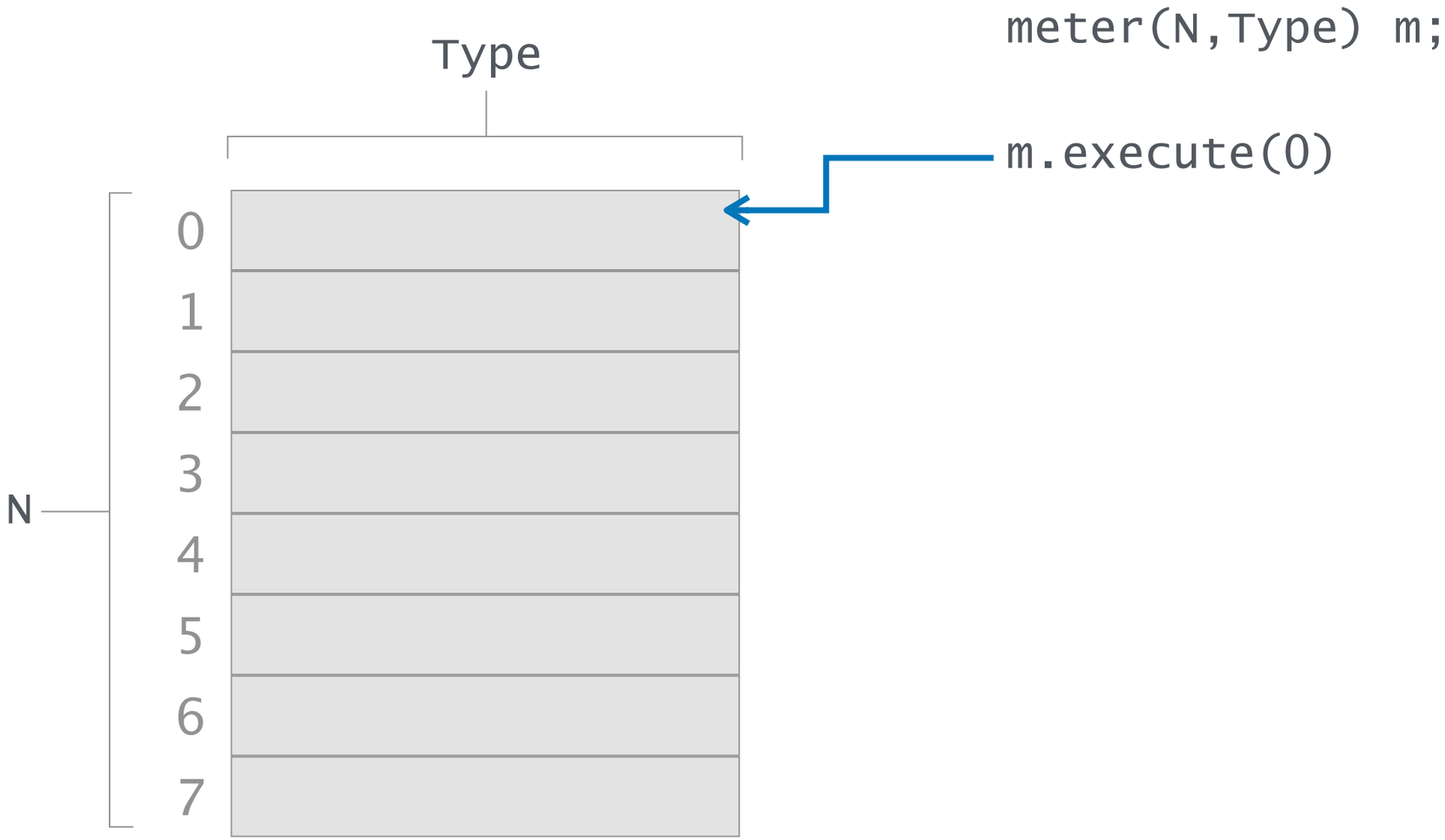
Meters



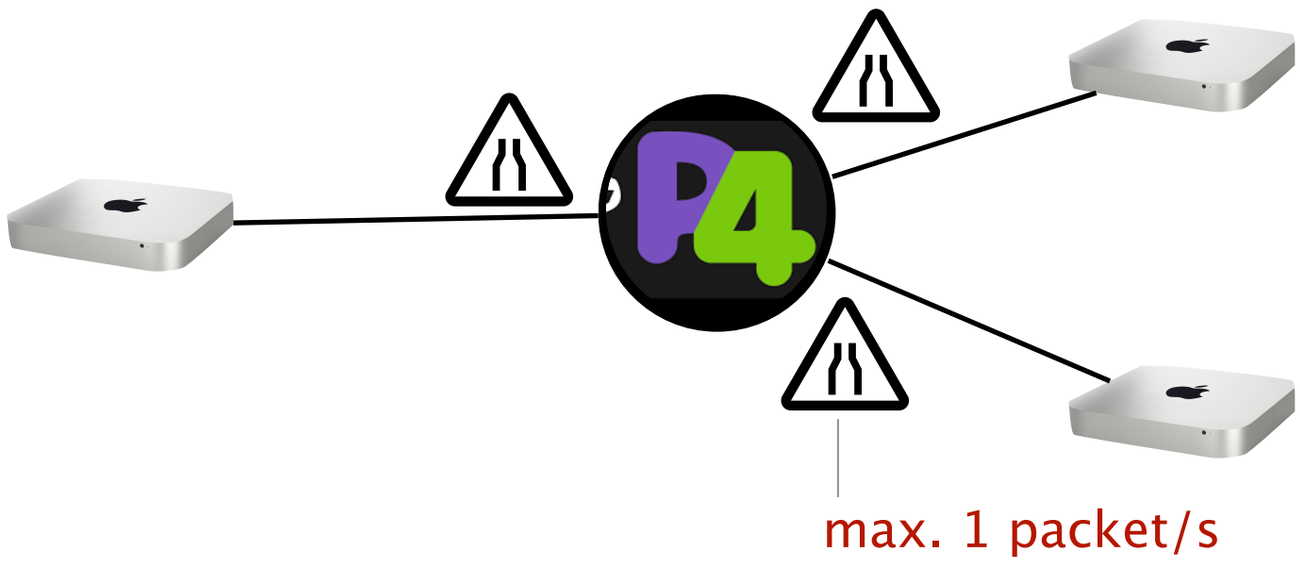
Parameters:	PIR	Peak Information Rate	[bytes/s] or [packets/s]
	CIR	Committed Information Rate	[bytes/s] or [packets/s]

more info https://tools.ietf.org/html/rfc2698

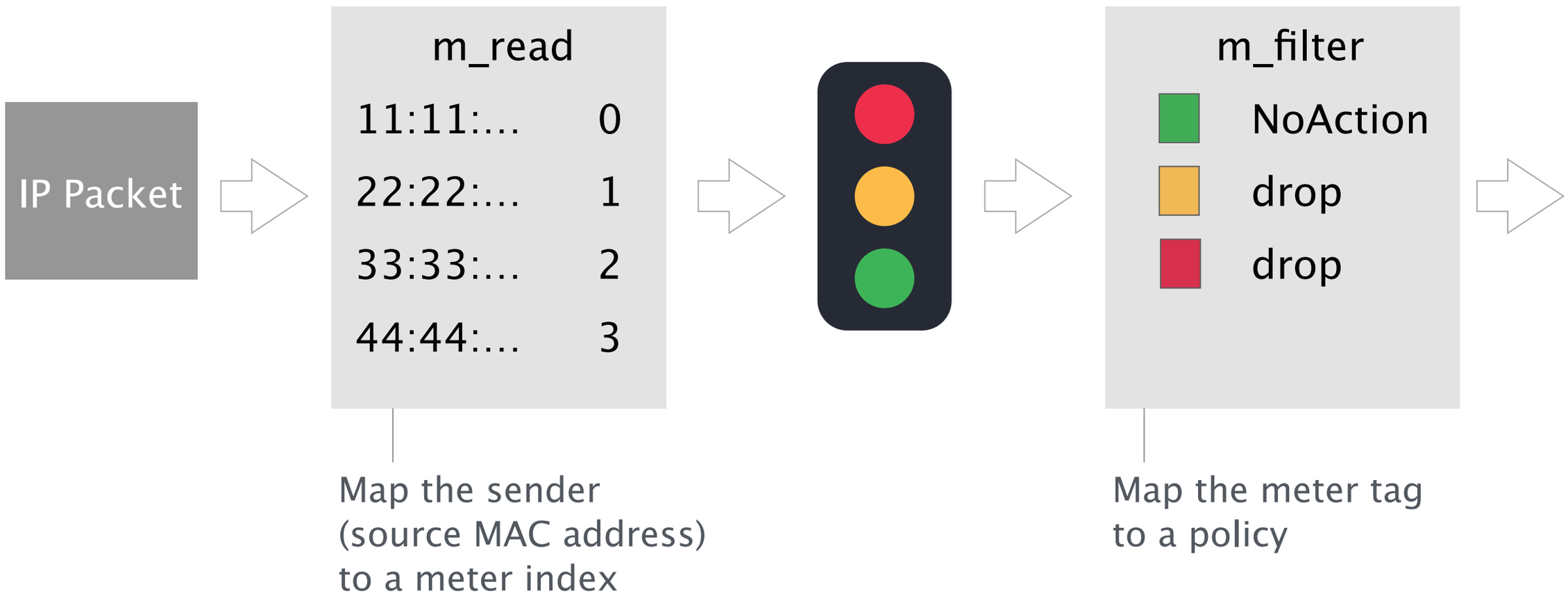
Like registers and counters,
meters are assigned in arrays



Example: Using a meter for rate-limiting



Example: Using a meter for rate-limiting



Example: Using a meter for rate-limiting

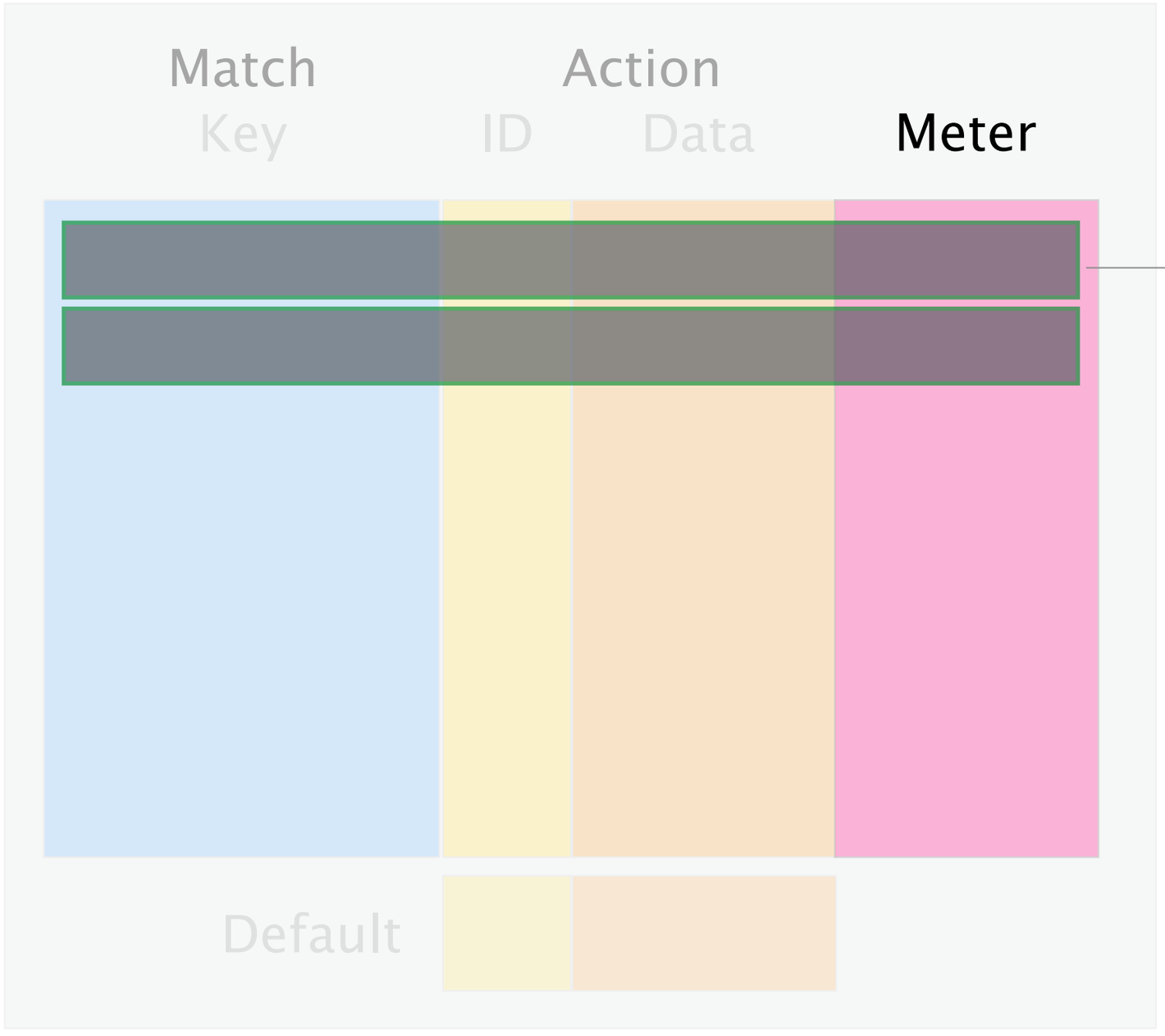
```
control MyIngress(...) {
 meter(32w16384, MeterType.packets) my_meter;

 action m_action(bit<32> meter_index) {
  my_meter.execute_meter<bit<32>>(meter_index, meta.meter_tag);
 }

 table m_read {
  key = { hdr.ethernet.srcAddr: exact; }
  actions = { m_action; NoAction; }
  ...
 }
 table m_filter {
  key = { meta.meter_tag: exact; }
  actions = { drop; NoAction; }
  ...
 }

 apply {
  m_read.apply();
  m_filter.apply();
 }
}
```

packet meter

execute meter

handle packets depending on meter tag

Direct meters are a special kind of meters that are attached to tables



Each entry has a meter cell that is executed when the entry matches

Example: Using a meter for rate-limiting

```
control MyIngress(...) {
 direct_meter<bit<32>>(MeterType.packets) my_meter;

 action m_action(bit<32> meter_index) {
  my_meter.read(meta.meter_tag);
 }

 table m_read {
  key = { hdr.ethernet.srcAddr: exact; }
  actions = { m_action; NoAction; }
  meters = my_meter;
  ...
 }
 table m_filter { ... }

 apply {
  m_read.apply();
  m_filter.apply();
 }
}
```

direct meter

read meter

attach meter to table

Summary

	Data plane interface		Control plane interface	
Object	read	modify/write	read	modify/write
Table	`apply()`	—	yes	yes
Register	`read()`	`write()`	yes	yes
Counter	—	`count()`	yes	reset
Meter	`execute()`		configuration only	

Advanced Topics in Communication Networks

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich

Tue 27 Sep 2022