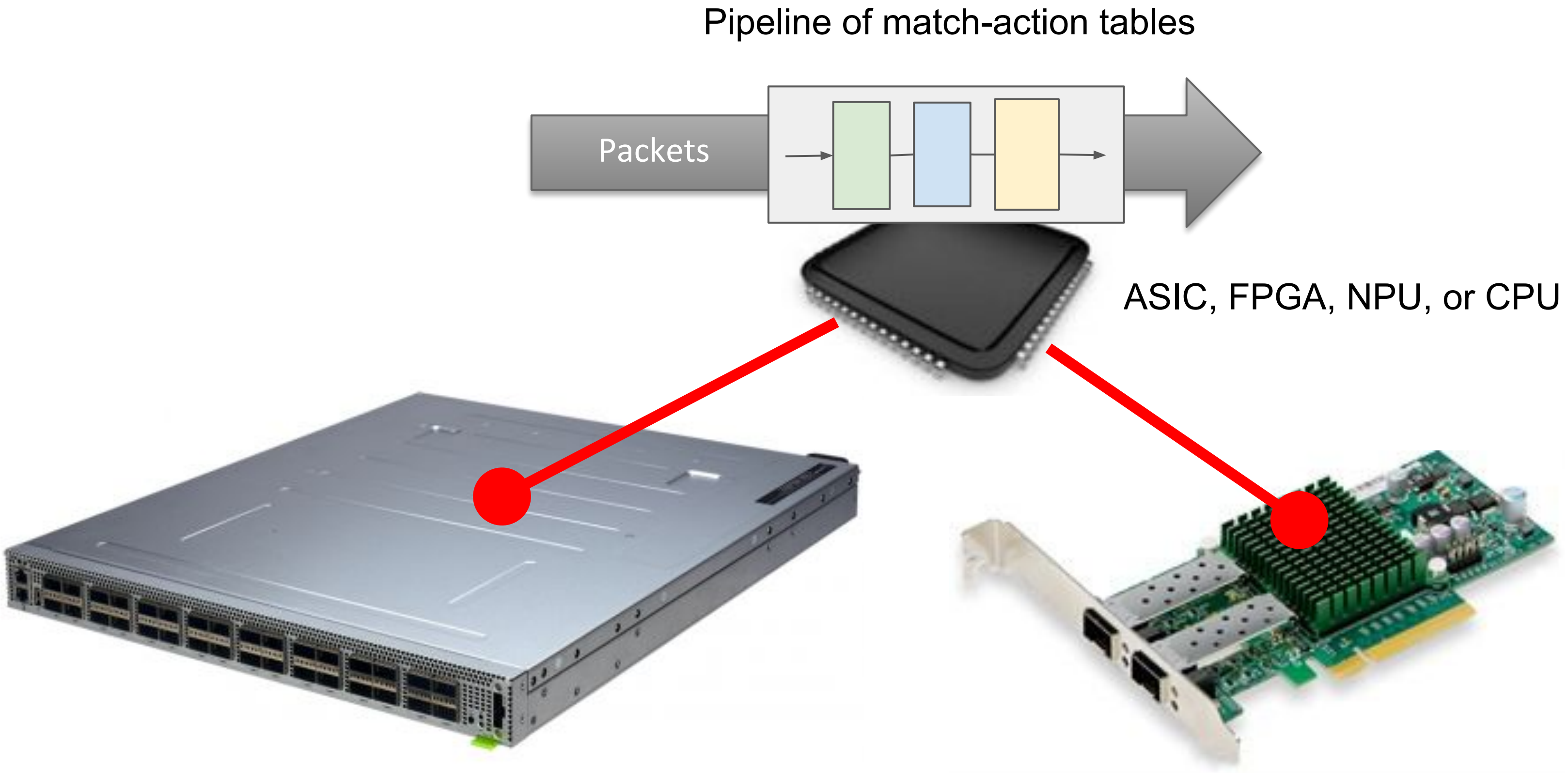
SESSION 1

P4 and P4Runtime basics

Overview

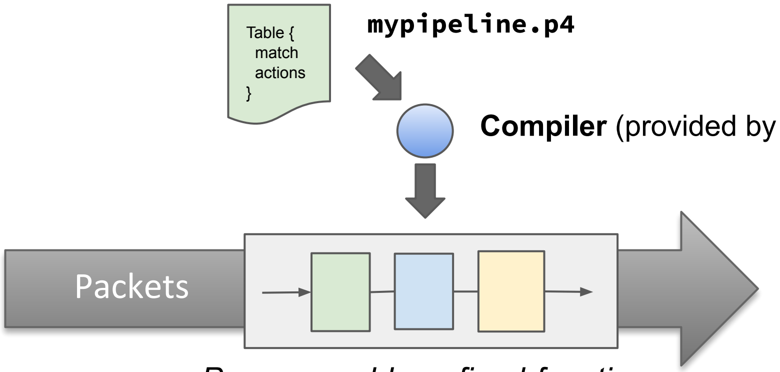
- **P4**
  - Data plane programming language
- **P4Runtime**
  - API for runtime control for P4-defined data planes
- **Hands-on lab (exercise 1)**

Data plane pipeline



P4 - Data Plane Programming Language

- **Domain-specific language to formally define the data plane pipeline**
  - Describe protocol headers, lookup tables, actions, counters, etc.
  - Can describe fast pipelines (e.g ASIC, FPGA) as well as a slower ones (e.g. SW switch)
- **Good for programmable switches, as well as fixed-function ones**
  - Defines “contract” between the control plane and data plane for runtime control



Programmable or fixed-function pipeline

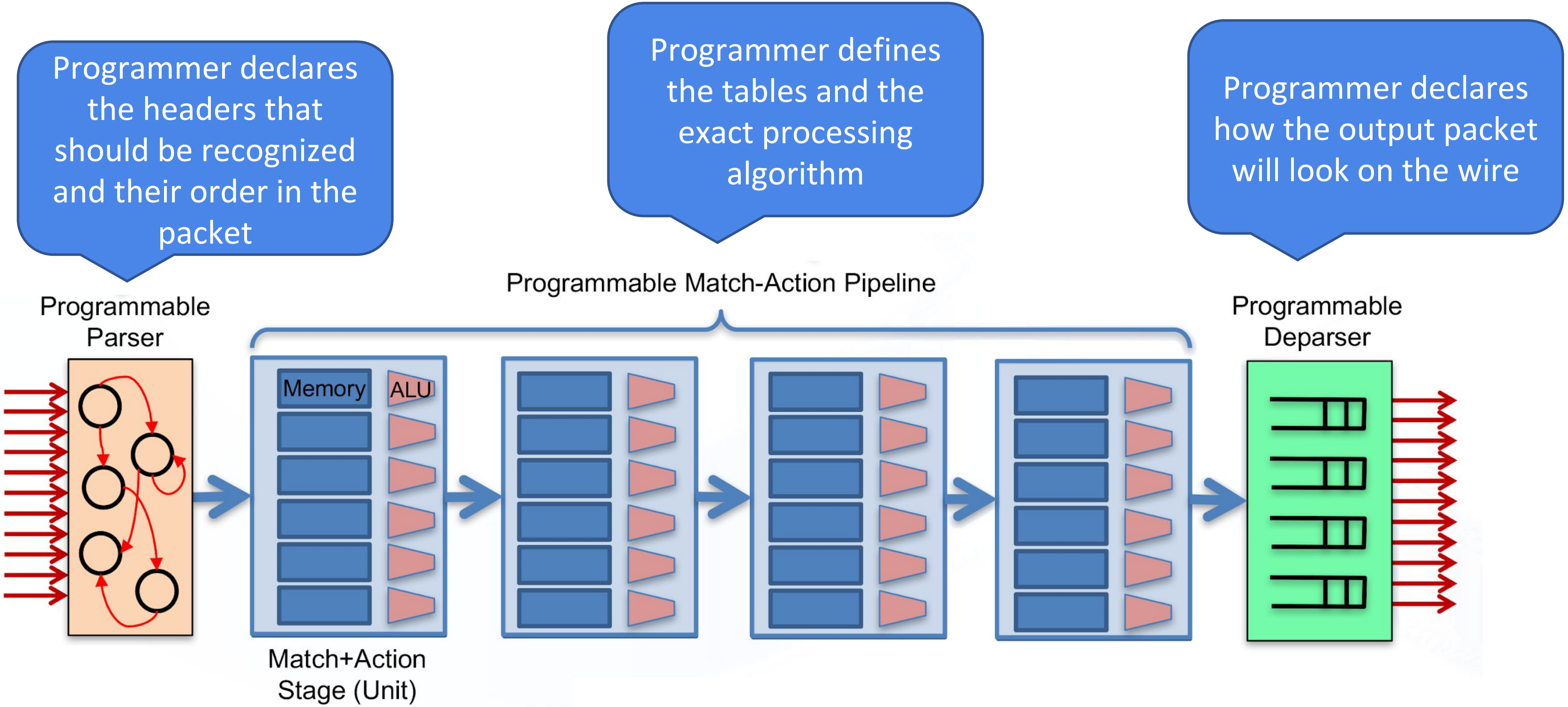
Evolution of the language

- **$P4_{14}$**
  - Original version of the language
  - Assumed specific device capabilities
  - Good only for a subset of programmable switch/targets

- **$P4_{16}$**
  - More mature and stable language definition
  - Does not assume device capabilities, which instead are defined by target manufacturer via external libraries/architecture definition
  - Good for many targets, e.g. switches and NICS, programmable or fixed-function
  - Focus of this tutorial

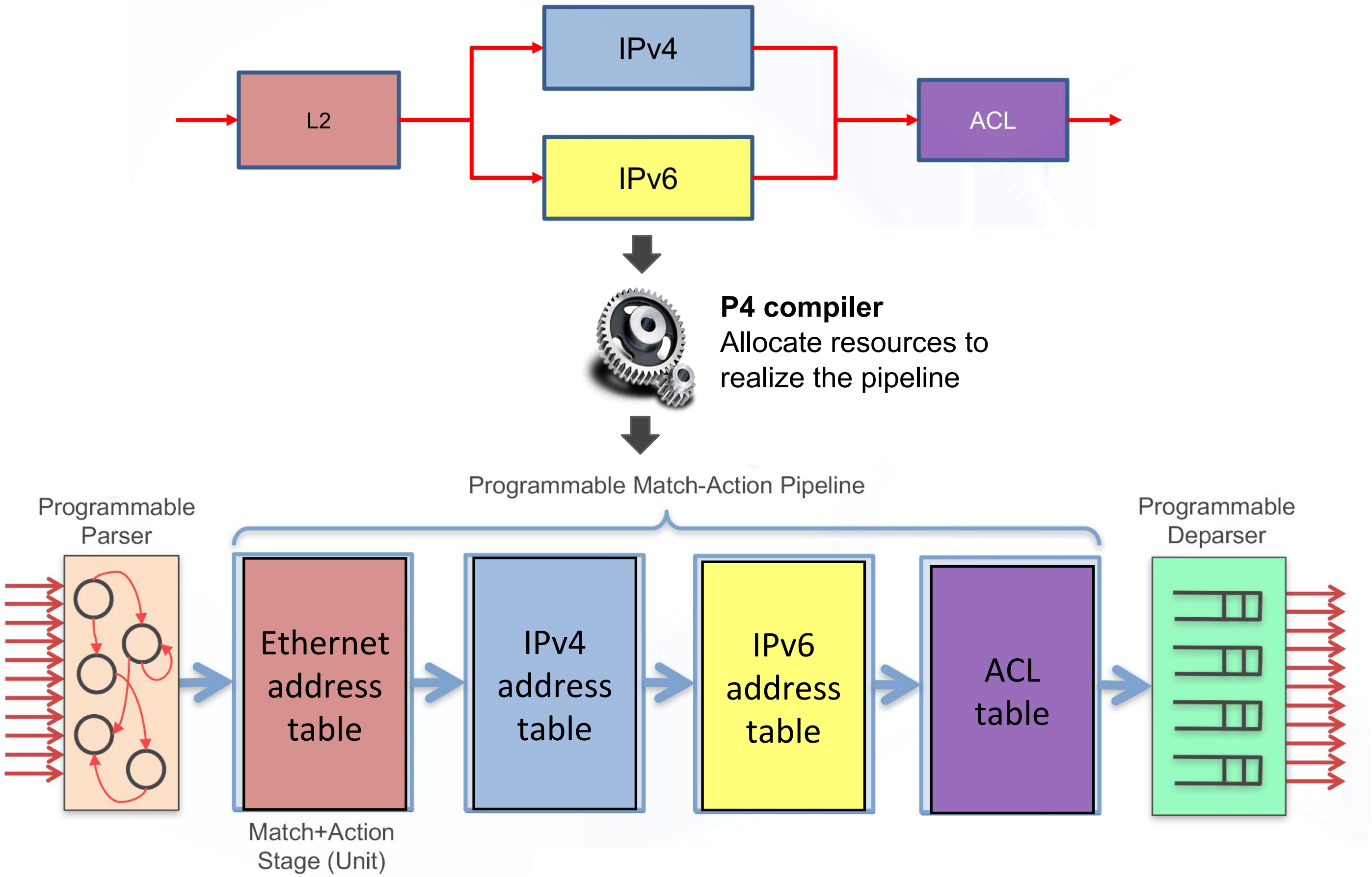
Architecture of a programmable switch

PISA: Protocol-Independent Switch Architecture



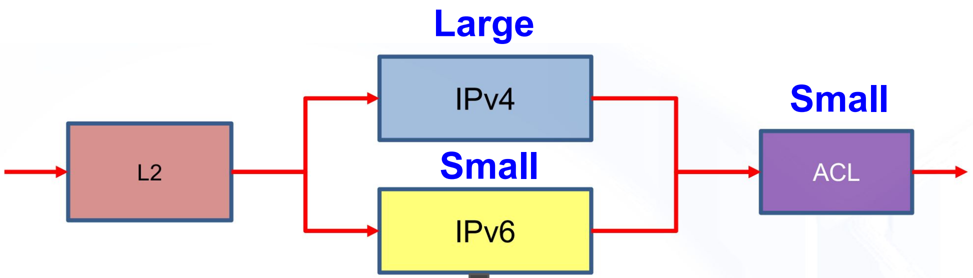
Slide courtesy P4.org

Compiling P4 on a programmable switch (PISA)

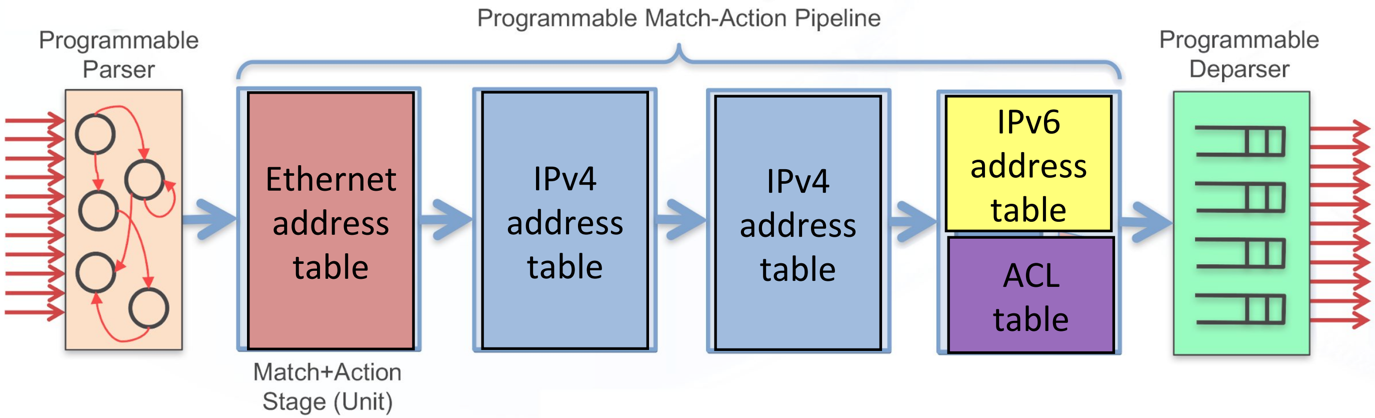


Slide courtesy P4.org

Compiling P4 on a programmable switch (PISA)



P4 compiler
Allocate resources to
realize the pipeline

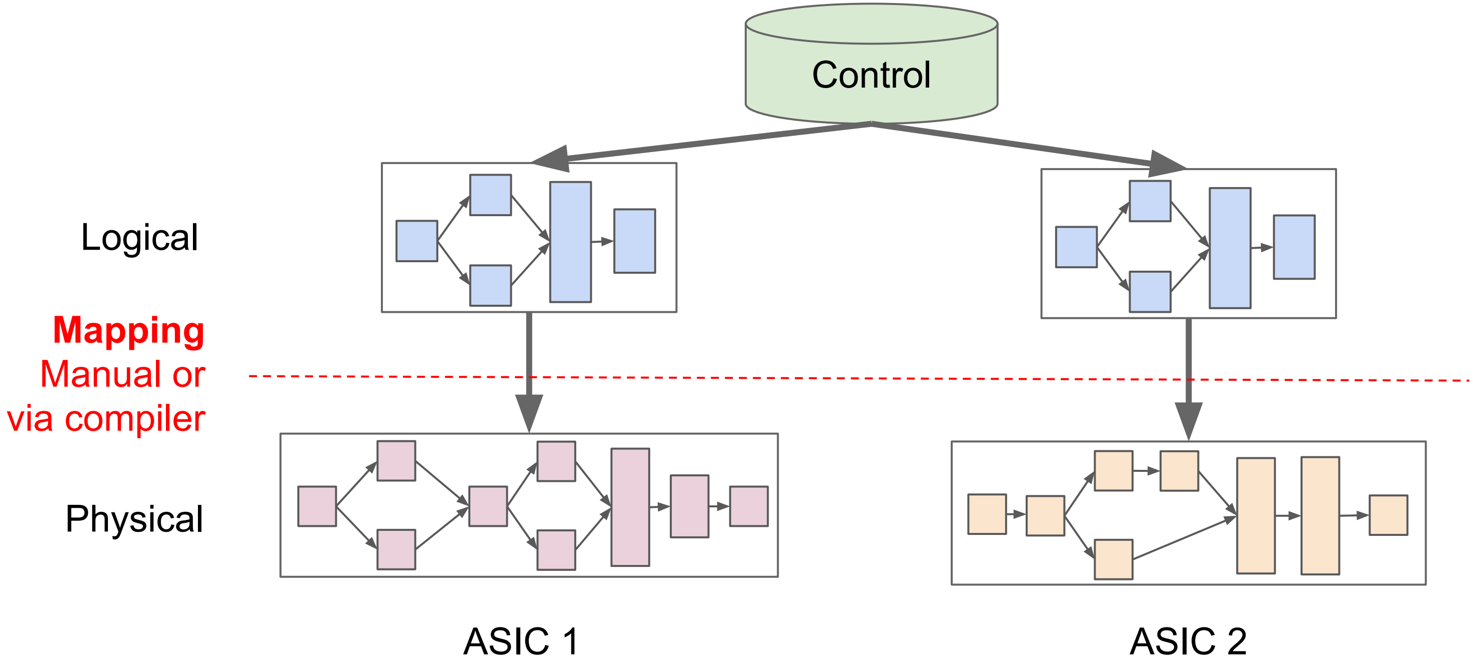


Slide courtesy P4.org

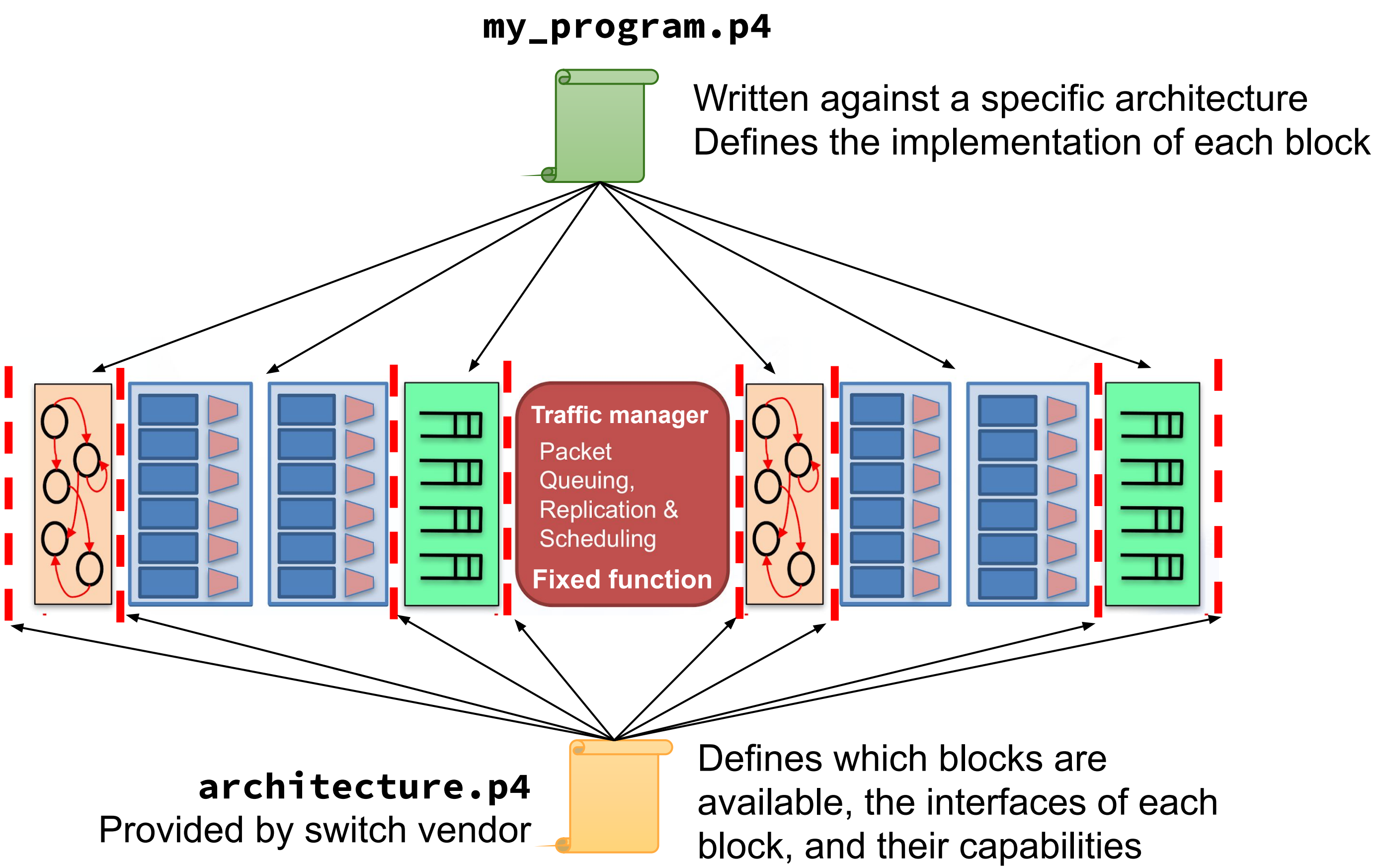
Role of P4 for fixed-function chips

Slide courtesy: Google

- P4 program tailored to apps / role - does not describe the hardware
- Switch maps program to fixed-function ASIC
- Enables portability of the control plane

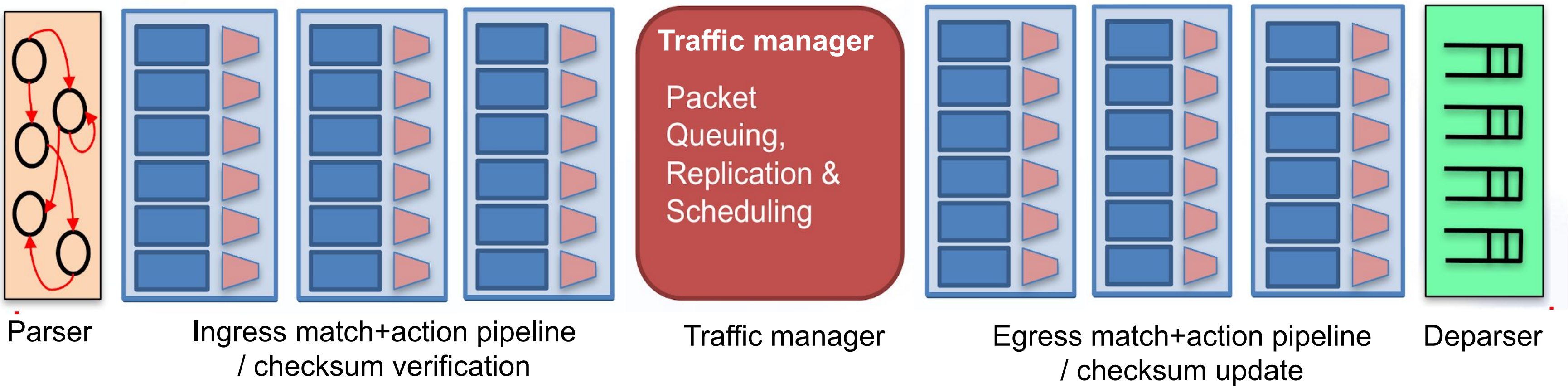


P4 architectures



V1Model P4 Switch Architecture (from P4_14)

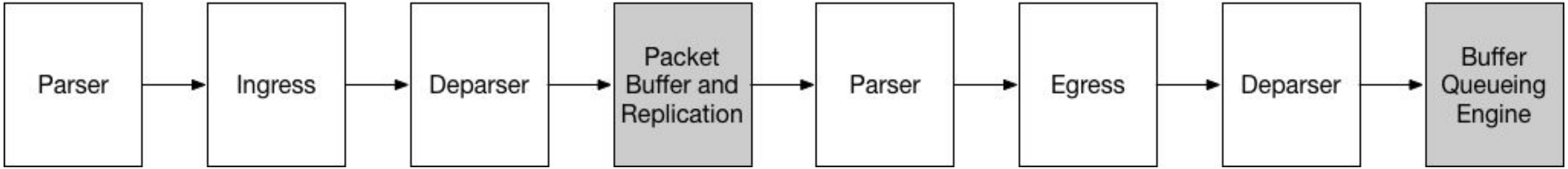
- **Parser/deparser** → **P4 programmable**
- **Checksum verification/update** → **P4 programmable**
- **Ingress Pipeline** → **P4 programmable**
- **Egress Pipeline** → **P4 programmable**
  - Match on egress port
- **Traffic Manager** → **Fixed function**



Slide courtesy P4.org

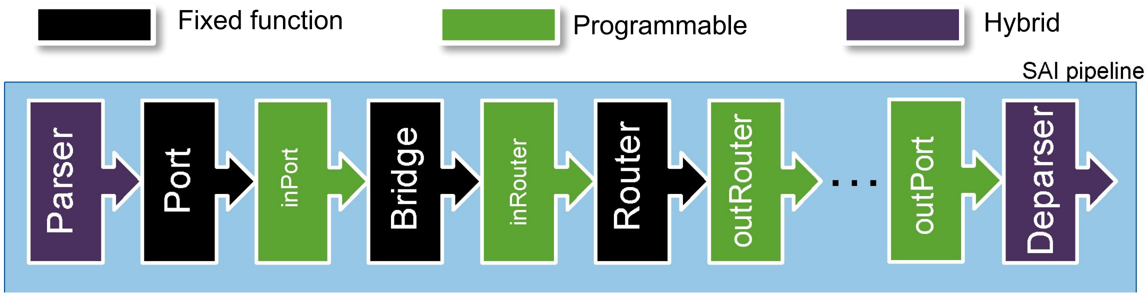
PSA - Portable Switch Architecture

- **Community-developed architecture (P4.org Arch WG)**
  - https://github.com/p4lang/p4-spec/tree/master/p4-16/psa
- **Describes common capabilities of a network switch**
- **6 programmable P4 blocks + 2 fixed-function blocks**
- **Defines capabilities beyond match+action tables**
  - Counters, meters, stateful registers, hash functions, etc.



Other P4 architectures

- **FlexSAI**
  - Hybrid programmable/fixed-function switch based on SAI
  - https://github.com/opencomputeproject/SAI/tree/master/flexsai/p4



- **Portable NIC Architecture (PNA)**
  - Work in progress by the P4.org Architecture WG
- **Proprietary architectures**
  - E.g., Tofino Native Architecture (TNA)

Preliminary takeaways

- **Can I implement/describe this or that function with P4?**
  - The P4 language aims at being flexible enough to express almost any behavior based on match-action tables
  - But, specific capabilities depend on the architecture
    - e.g. ternary match vs. longest-prefix match vs. exact match, ECMP-like action selectors, stateful memories, etc.
- **Can I execute my P4 program on a switch *X* from vendor *Y*?**
  - Yes, if vendor provides you with a P4 compiler for the specific arch

Architectures enable portability of P4 programs across different HW and SW targets

P4 program template (V1Model architecture)


```
#include <core.p4>
#include <v1model.p4>
/* HEADERS */
struct metadata { ... }
struct headers {
  ethernet_t    ethernet;
  ipv4_t        ipv4;
}
/* PARSER */
parser MyParser(packet_in packet,
         out headers hdr,
         inout metadata meta,
         inout standard_metadata_t smeta) {
  ...
}
/* CHECKSUM VERIFICATION */
control MyVerifyChecksum(in headers hdr,
                         inout metadata meta) {
   ...
}
/* INGRESS PROCESSING */
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t std_meta) {
  ...
}
```

```
/* EGRESS PROCESSING */
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t std_meta) {
  ...
}
/* CHECKSUM UPDATE */
control MyComputeChecksum(inout headers hdr,
                          inout metadata meta) {
  ...
}
/* DEPARSER */
control MyDeparser(inout headers hdr,
                   inout metadata meta) {
  ...
}
/* SWITCH */
V1Switch(
  MyParser(),
  MyVerifyChecksum(),
  MyIngress(),
  MyEgress(),
  MyComputeChecksum(),
  MyDeparser()
) main;
```

Slide courtesy P4.org

P4 program example: simple_router.p4

```
header ethernet_t {
  bit<48> dst_addr;
  bit<48> src_addr;
  bit<16> eth_type;
}

header ipv4_t {
  bit<4>  version;
  bit<4>  ihl;
  bit<8>  diffserv;
  …
}

parser parser_impl(packet_in pkt, out headers_t hdr) {
  /* Parser state machine to extract header fields */
}
```

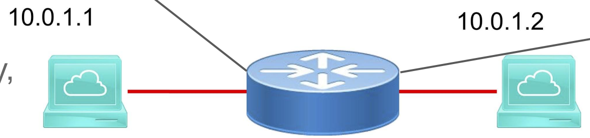
Ingress pipeline implementation:

```
action set_next_hop(bit<48> dst_addr) {
  ethernet.dst_addr = dst_addr;
  ipv4.ttl = ipv4.ttl - 1;
}
...
table ipv4_routing_table {
  key = { ipv4.dst_addr : LPM; // longest-prefix match }
  actions = { set_next_hop(); drop(); }
  size = 4096; // table entries
}
...
apply {
  if (ipv4.isValid()) {
    ipv4_routing_table.apply();
  }
}
...
```

Simple router example

- **Data plane (P4) program**
  - Defines the match-action tables
  - Performs the lookup
  - Executes the chosen action
- **Control plane**
  - Populates table entries with specific information
    - Based on configuration, automatic discovery, protocol calculations

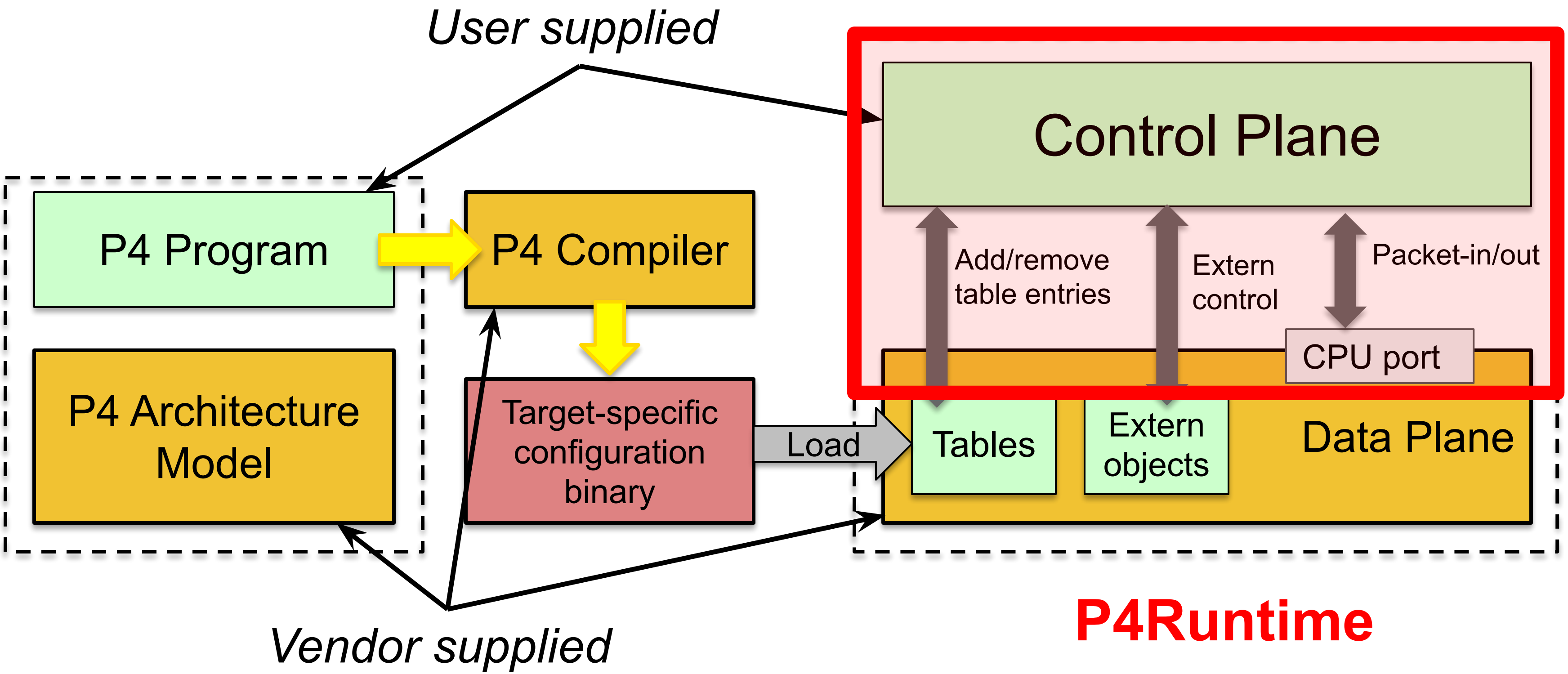
```
action ipv4_forward(bit<48> dst_addr, bit<9> port) {
  ethernet.dst_addr = dst_addr;
  standard_metadata.egress_spec = port;
  ipv4.ttl = ipv4.ttl - 1;
}
table ipv4_routing_table {
 key = {
        ipv4.dst_addr : LPM; // longest-prefix match
 }
 actions = {
        ipv4_forward();
        drop();
  }
}
```



Control plane populates table entries

Key	Action	Action Data
10.0.1.1/32	ipv4_forward	dstAddr=00:00:00:00:01:01 port=1
10.0.1.2/32	drop	
*`	NoAction	

P4 workflow summary



Slide courtesy P4.org

P4Runtime
Runtime control API for P4-defined data planes

P4Runtime v1.0

- **Released on Jan 2019**
- **Open source specification**
  - Started by Google and Barefoot in mid-2016
  - Contributions by many industry professionals
  - Use GitHub issues / PR for discussions
- **Based on continuous implementation feedbacks from Google and ONF**
  - First ONF demo in Oct 2017

P4Runtime Specification

version 1.0.0

The P4.org API Working Group

2019-01-29

Abstract

P4 is a language for programming the data plane of network devices. The P4Runtime API is a control plane specification for controlling the data plane elements of a device defined or described by a P4 program. This document provides a precise definition of the P4Runtime API. The target audience for this document includes developers who want to write controller applications for P4 devices or switches.

Contents

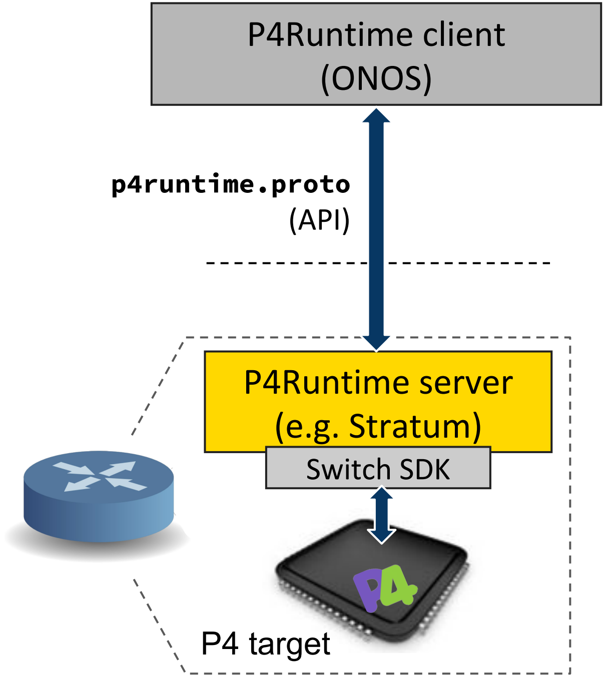
1. Introduction and Scope . . . 4
   1.1. P4 Language Version Applicability . . . 4
   1.2. In Scope . . . 5
   1.3. Not In Scope . . . 5
2. Terms and Definitions . . . 5
3. Reference Architecture . . . 7
   3.1. Idealized Workflow . . . 8
   3.2. P4 as a Behavioral Description Language . . . 8
   3.3. Alternative Workflows . . . 9
      3.3.1. P4 Source Available, Compiled into P4Info but not Compiled into P4 Device Config . . . 9
      3.3.2. No P4 Source Available, P4Info Available . . . 9
      3.3.3. Partial P4Info and P4 Source are Available . . . 9
      3.3.4. P4Info Role-Based Subsets . . . 10
4. Controller Use-cases . . . 10
   4.1. Single Embedded Controller . . . 10
   4.2. Single Remote Controller . . . 10
   4.3. Embedded + Single Remote Controller . . . 11
   4.4. Embedded + Two Remote Controllers . . . 11
   4.5. Embedded Controller + Two High-Availability Remote Controllers . . . 11
5. Master-Slave Arbitration and Controller Replication . . . 13
   5.1. Default Role . . . 15
   5.2. Role Config . . . 15
   5.3. Rules for Handling MasterArbitrationUpdate Messages Received from Controllers . . . 16
   5.4. Mastership Change . . . 17
6. The P4Info Message . . . 17

1

https://p4.org/p4-spec/
https://github.com/p4lang/p4runtime

P4Runtime overview

- **Protobuf-based API definition**
  - Efficient wire format
  - Automatically generate code to serialize/deserialize messages for many languages
- **gRPC-based transport**
  - Automatically generate high-performance client/server code in many languages
  - Pluggable authentication and security
  - Bi-directional stream channels
- **P4-program independent**
  - Allow pushing new P4 programs to reconfigure the pipeline at runtime
- **Equally good for remote or local control plane**
  - With or without gRPC



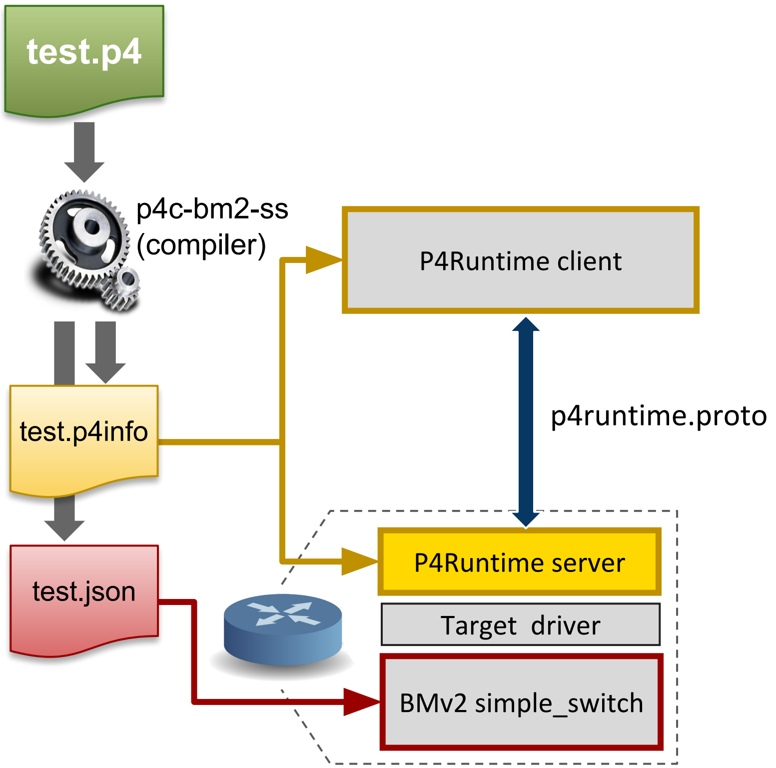
P4Runtime main features

- **Batched read/writes**
  - Table entries, action groups, counters, registers, etc.
- **Master-slave arbitration**
  - For control plane high-availability and fault-tolerance
- **Multiple master controllers via role partitioning**
  - E.g. local control plane for L2, remote one for L3
- **Flexible and efficient packet I/O**
  - OpenFlow-like packet-in/out with arbitrary metadata
  - Digests, i.e. batched notification to controller with subset of packet headers
- **Designed around P4 PSA architecture**
  - But can be extended to others via Protobuf “Any” messages
  - Works well with V1Model

P4 compiler workflow

P4 compiler generates 2 outputs:

1. **Target-specific binaries**
   - Used to realize switch pipeline
     (e.g. binary config for ASIC, BMv2 JSON, etc.)
2. **P4Info file**
   - "Schema" of pipeline for runtime control
     - Captures P4 program attributes such as tables, actions, parameters, etc.
   - Protobuf-based format
   - Target-independent compiler output
     - Same P4Info for SW switch, ASIC, etc.



Full P4Info protobuf specification:
https://github.com/p4lang/p4runtime/blob/master/proto/p4/config/v1/p4info.proto

P4Info example

basic_router.p4

```
...

action ipv4_forward(bit<48> dstAddr,
                    bit<9> port) {
   eth.dstAddr = dstAddr;
   metadata.egress_spec = port;
   ipv4.ttl = ipv4.ttl - 1;
}

...

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

P4 compiler

basic_router.p4info

Protobuf message text format

```
actions {
  id: 16786453
  name: "ipv4_forward"
  params {
    id: 1
    name: "dstAddr"
    bitwidth: 48
    ...
    id: 2
    name: "port"
    bitwidth: 9
  }
}
...
tables {
  id: 33581985
  name: "ipv4_lpm"
  match_fields {
    id: 1
    name: "hdr.ipv4.dstAddr"
    bitwidth: 32
    match_type: LPM
  }
  action_ref_id: 16786453
}
```

P4Runtime table entry WriteRequest example

basic_router.p4

```
action ipv4_forward(bit<48> dstAddr,
                    bit<9>  port) {
   /* Action implementation */
}
table ipv4_lpm {
   key = {
       hdr.ipv4.dstAddr: lpm;
   }
   actions = {
       ipv4_forward;
       ...
   }
   ...
}
```

Logical view of table entry

```
hdr.ipv4.dstAddr=10.0.1.1/32
     -> ipv4_forward(00:00:00:00:00:10, 7)
```

Control plane generates

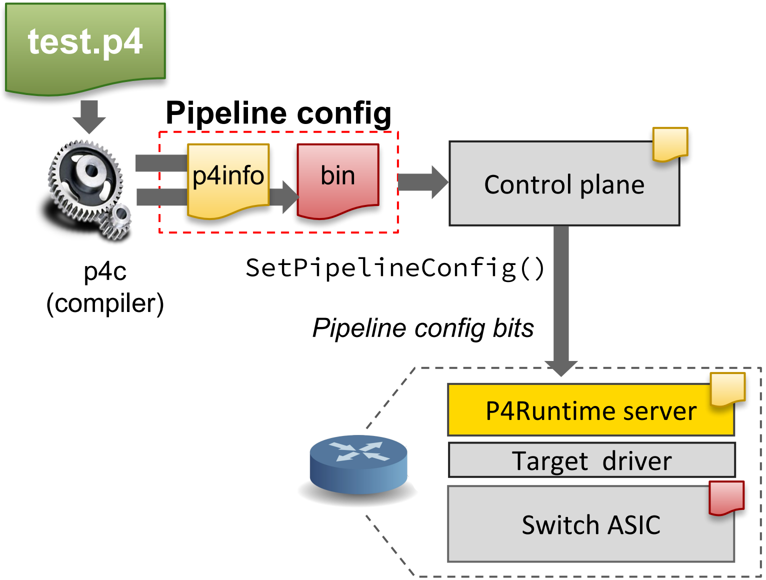
WriteRequest message

Protobuf message text format

```
device_id: 1
election_id { … }
updates {
  type: INSERT
  entity {
    table_entry {
      table_id: 33581985
      match {
        field_id: 1
        lpm {
          value: "\n\000\001\001"
          prefix_len: 32
        }
      }
      action {
        action_id: 16786453
        params {
          param_id: 1
          value: "\000\000\000\000\000\n"
        }
        params {
          param_id: 2
          value: "\000\007"
…
```

P4Runtime SetPipelineConfig

```
message SetForwardingPipelineConfigRequest {
 enum Action {
   UNSPECIFIED = 0;
   VERIFY = 1;
   VERIFY_AND_SAVE = 2;
   VERIFY_AND_COMMIT = 3;
   COMMIT = 4;
   RECONCILE_AND_COMMIT = 5;
 }
 uint64 device_id = 1;
 uint64 role_id = 2;
 Uint128 election_id = 3;
 Action action = 4;
 ForwardingPipelineConfig config = 5;
}
```




```
message ForwardingPipelineConfig {
 config.P4Info p4info = 1;
 // Target-specific P4 configuration.
 bytes p4_device_config = 2;
}
```

P4Runtime summary

- **P4Runtime is an improvement over previous data plane APIs**
  - Realize the vision of OpenFlow 2.0
  - Provides protocol and pipeline-independence
  - Protocols supported and pipeline are formally specified using P4
- **Based on protobuf and gRPC**
  - Makes it easy to implement a P4Runtime client/server by auto-generating code for different languages
- **P4Info as a contract between control and data plane**
  - Generated by P4 compiler
  - Needed by the control plane to format the body of P4Runtime messages (e.g. to add table entry)

Exercise 1 overview

Exercise 1: Steps

1. Look at given P4 program
2. Answer questions about the implementation
3. Compile it for BMv2, obtain bmv2.json and p4info.txt
4. Start stratum_bmv2 in Mininet
5. Use P4Runtime Shell to push pipeline config and write table entries in the bridging table
6. Test connectivity via ping

Exercise 1: Tools

Docker container 1: `opennetworking/mn-stratum`

- Provides Mininet with stratum_bmv2
- Allow execution of custom topology scripts (2x2 fabric in our case)

Docker container 2: `opennetworking/p4c`

- Containerized version of the open source P4_16 compiler

Docker container 3: `p4lang/p4runtime-sh`

- Interactive P4Runtime Shell (based on IPython)

Docker container 4: `onosproject/onos:2.2.0`

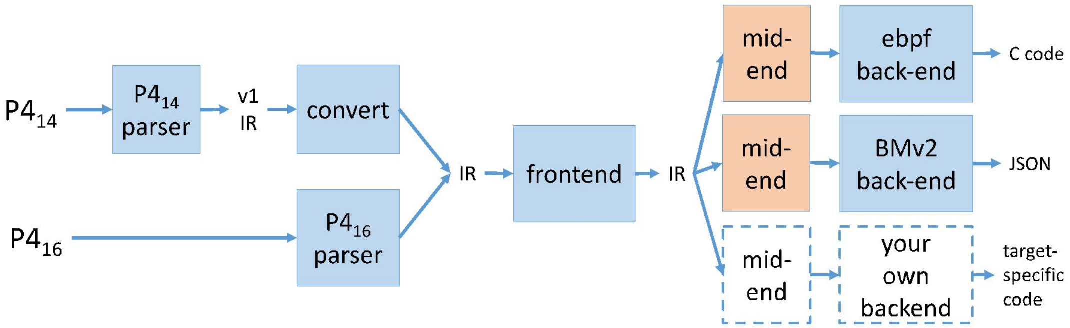
- ONOS, not used in this exercise
- We'll leave it running to use in next exercises

Starter P4 program

- **Goal: build an IPv6-based leaf-spine data center fabric**
- **Each switch acts as a (simplified) IPv6 router:**
  - L2 bridging for hosts in the same subnet
    - Forward based on MAC dest with host learning
  - IPv6 routing for hosts in different subnets
  - ECMP to load balance traffic across multiple spines
  - Controller packet-in/packet-out
    - For link and host discovery
- **Same P4 code used for leaves and spines (p4src/main.p4)**
  - Well commented, easy to understand even with little or no P4 experience

p4c

- **Open-source frontend compiler**
  - https://github.com/p4lang/p4c
- **Generates P4Info**
- **Support multiple backends (vendor-supplied)**
  - Generate code for ASICs, NICs, FPGAs, software switches and other targets
- **Some backends are open-source (BMv2, eBPF)**

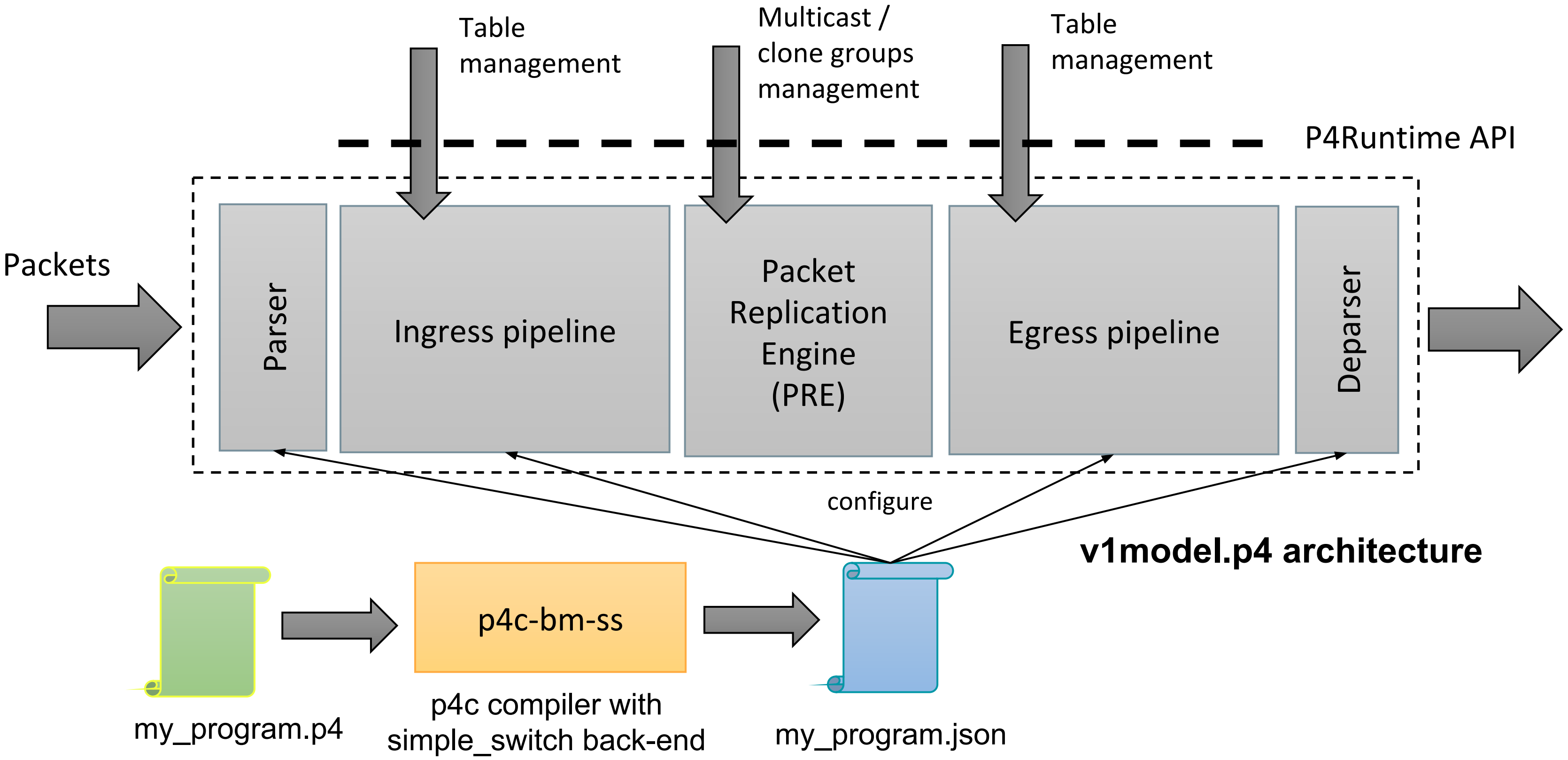


Slide courtesy: M. Budiu, C. Doss. The architecture of the P4_16 compiler. P4 Workshop 2017

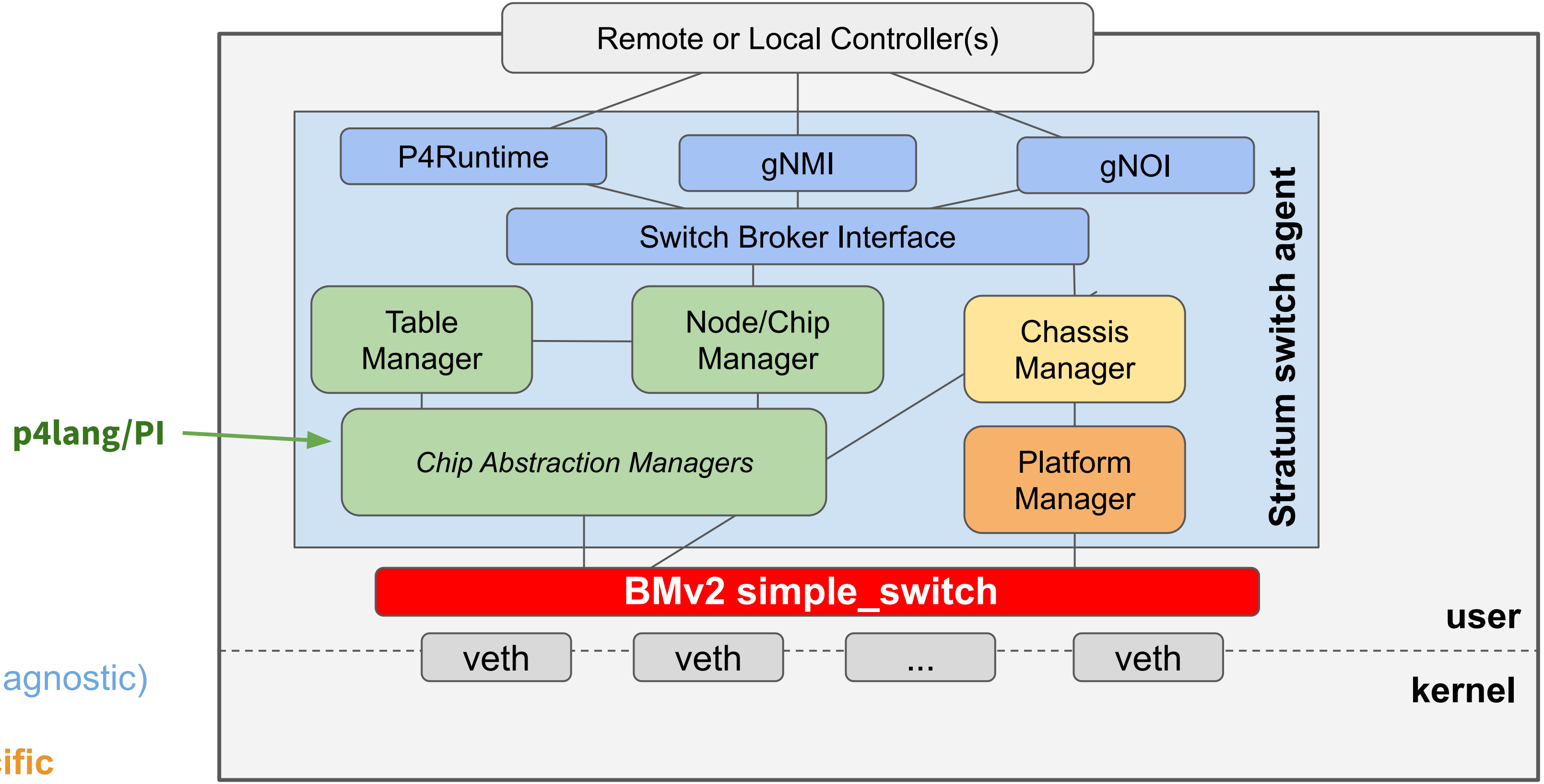
BMv2 – Reference P4 software switch

- **Open-source user-space implementation**
  - https://github.com/p4lang/behavioral-model
- **BMv2 = Behavioral-Model version 2**
- **Aimed at being 100% conformant to the P4 specification**
  - Performance is non-goal, i.e. low throughput
- **Architecture-independent**
  - Mostly generic code which can be used to implement any P4 architecture
- **We use the "simple_switch" with Stratum support**
  - Implementation of V1Model architecture with Stratum APIs over gRPC

BMv2’s simple_switch target



stratum_bmv2



Common (HW agnostic)
Chip specific
Platform specific
Chip and Platform specific

Exercise 1: Get Started

These slides:
`http://bit.ly/ngsdn-tutorial-slides`

Open lab README on GitHub:

http://bit.ly/ngsdn-tutorial-lab

Or open in text editor:

`~/ngsdn-tutorial/README.md`

`~/ngsdn-tutorial/EXERCISE-1.md`

Before starting!
Update tutorial repo
(requires Internet access)

```
cd ~/ngsdn-tutorial
git pull origin master
make pull-deps
```

P4 language cheat sheet:
`http://bit.ly/p4-cs`

You can work on your own using the instructions.
You have time until 11.15 - coffee and snacks are outside.