P4 Tutorial Welcome

Noa Zilberman
University of Cambridge

Goals

- **Learn P4 Language**
  - Traditional applications
  - Novel applications
- **Learn P4 software tools**
  - P4 Compiler
  - BMv2
  - P4Runtime
- **Learn about P4 hardware targets**
  - mini-workshop featuring solutions by Barefoot, Netronome, Netcope and NetFPGA.
- **Networking (the other kind)**
- **Have fun!**

Introduction to Data Plane Programming

Language Basics

Break

Software Tools & P4 Runtime

Lunch

Group Programming Exercise

Break

Architectures Mini-Workshop

Reception

Instructors

Nate Foster
Cornell

Theo Jepsen
USI Lugano

Milad Sharif
Barefoot Networks

Robert Soulė
USI Lugano

Pietro Bressana
USI Lugano

Thank you

- **Instructors & TAs**
  - Nate Foster
  - Theo Jepsen
  - Milad Sharif
  - Robert Soulẻ
  - Pietro Bressana
- **Mini-Workshop speakers**
  - Pavel Benacek
  - Jaco Joubert
- **You!**

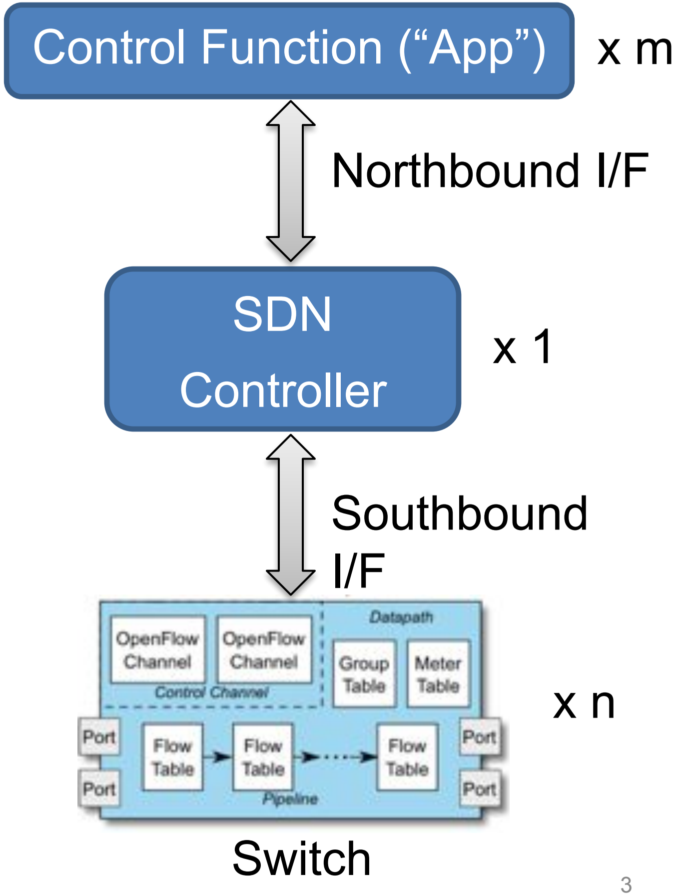
P4 Language Tutorial

What is Data Plane Programming?

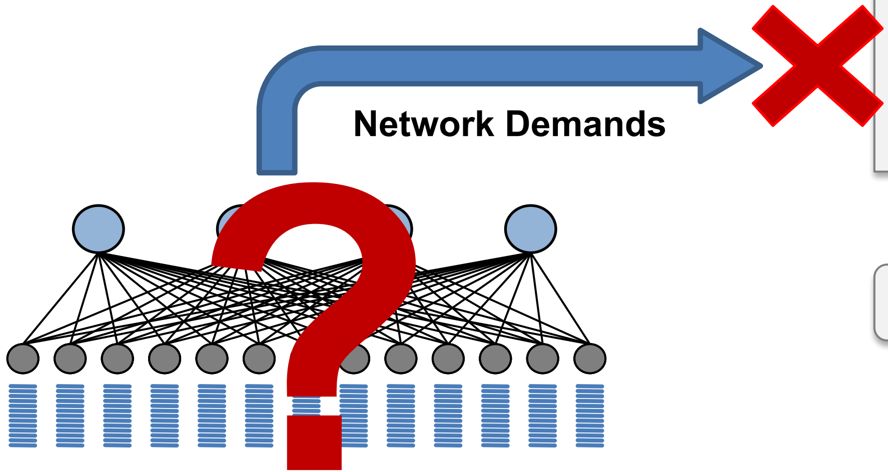
- **Why program the Data Plane?**

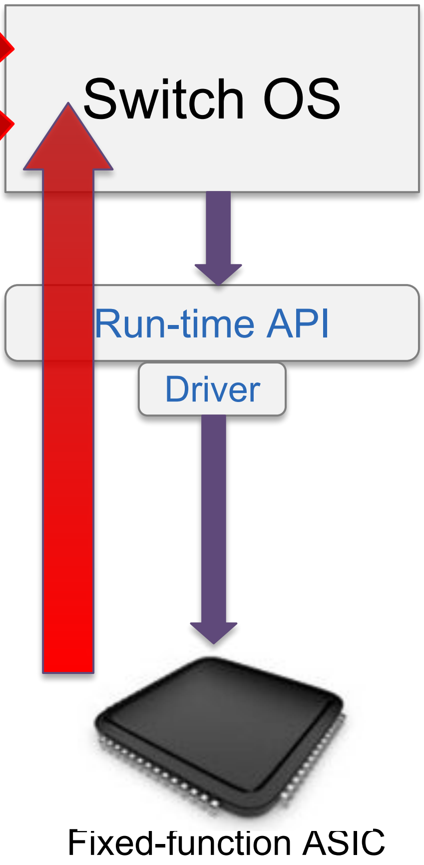
Software Defined Networking: Logically Centralized Control

- **Main contributions**
  - OpenFlow = standardized *model*
    - match/action abstraction
  - OpenFlow = standardized *protocol* to interact with switch
    - download flow table entries, query statistics, etc.
  - *Concept* of *logically* centralized control via a single entity ("SDN controller")
    - Simplifies control plane – e.g. compute optimal paths at one location (controller), vs. waiting for distributed routing algorithms to converge
- **Issues**
  - Data-plane protocol evolution requires changes to standards (12 → 40 OpenFlow match types)
  - Limited interoperability between vendors => southbound I/F differences handled at controller (OpenFlow / netconf / JSON / XML variants)



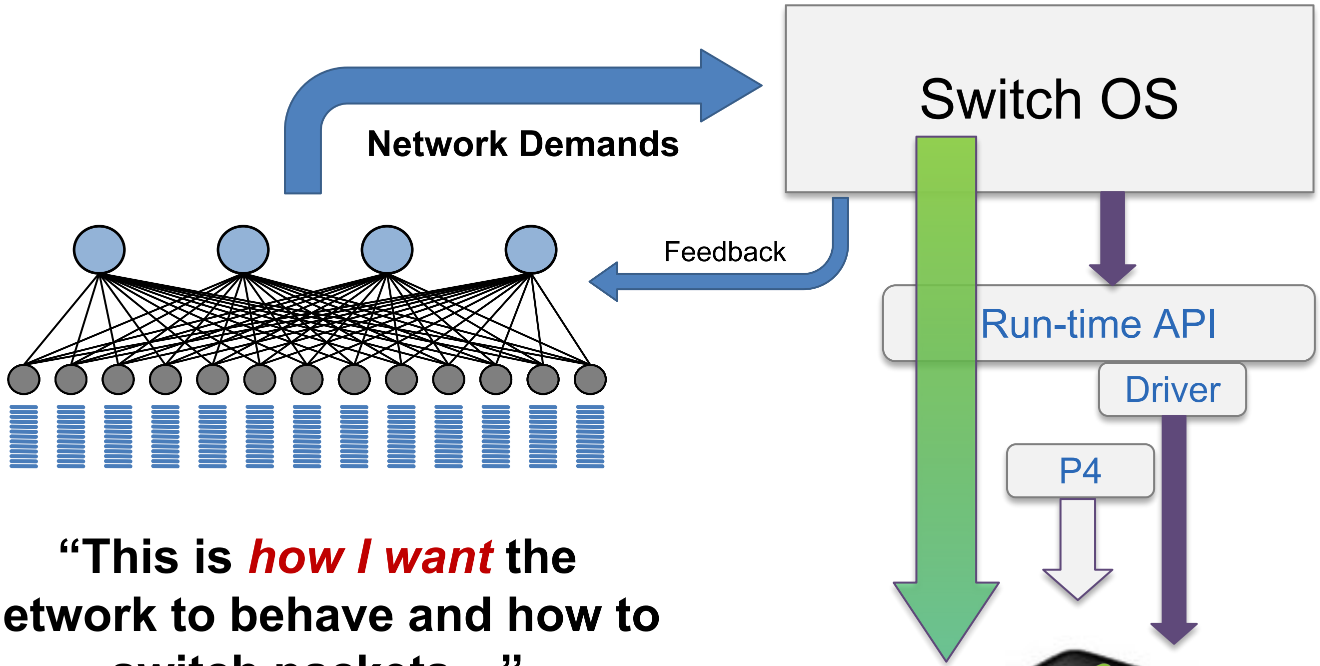
Status Quo: Bottom-up design





“This is *how I know* to process packets”
(i.e. the ASIC datasheet makes the rules)

A Better Approach: Top-down design



“This is *how I want* the network to behave and how to switch packets…”
(the user / controller makes the rules)

P4 Programmable Device

Benefits of Data Plane Programmability

- **New Features** – Add new protocols
- **Reduce complexity** – Remove unused protocols
- **Efficient use of resources** – flexible use of tables
- **Greater visibility** – New diagnostic techniques, telemetry, etc.
- **SW style development** – rapid design cycle, fast innovation, fix data plane bugs in the field
- **You keep your own ideas**

Think programming rather than protocols…

Programmable Network Devices

- **PISA: Flexible Match+Action ASICs**
  - Intel Flexpipe, Cisco Doppler, Cavium (Xpliant), Barefoot Tofino, …
- **NPU**
  - EZchip, Netronome, …
- **CPU**
  - Open Vswitch, eBPF, DPDK, VPP…
- **FPGA**
  - Xilinx, Altera, …

These devices let us tell them how to process packets.

What can you do with P4?

- **Layer 4 Load Balancer – SilkRoad[1]**
- **Low Latency Congestion Control – NDP[2]**
- **In-band Network Telemetry – INT[3]**
- **Fast In-Network cache for key-value stores – NetCache[4]**
- **Consensus at network speed – NetPaxos[5]**
- **Aggregation for MapReduce Applications [6]**
- **… and much more**

[1] Miao, Rui, et al. "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs." SIGCOMM, 2017.
[2] Handley, Mark, et al. "Re-architecting datacenter networks and stacks for low latency and high performance.” SIGCOMM, 2017.
[4] Kim, Changhoon, et al. "In-band network telemetry via programmable dataplanes.” SIGCOMM. 2015.
[3] Xin Jin et al. “NetCache: Balancing Key-Value Stores with Fast In-Network Caching.” To appear at SOSP 2017
[5] Dang, Huynh Tu, et al. "NetPaxos: Consensus at network speed.” SIGCOMM, 2015.
[6] Sapio, Amedeo, et al. "In-Network Computation is a Dumb Idea Whose Time Has Come." *Hot Topics in Networks*. ACM, 2017.

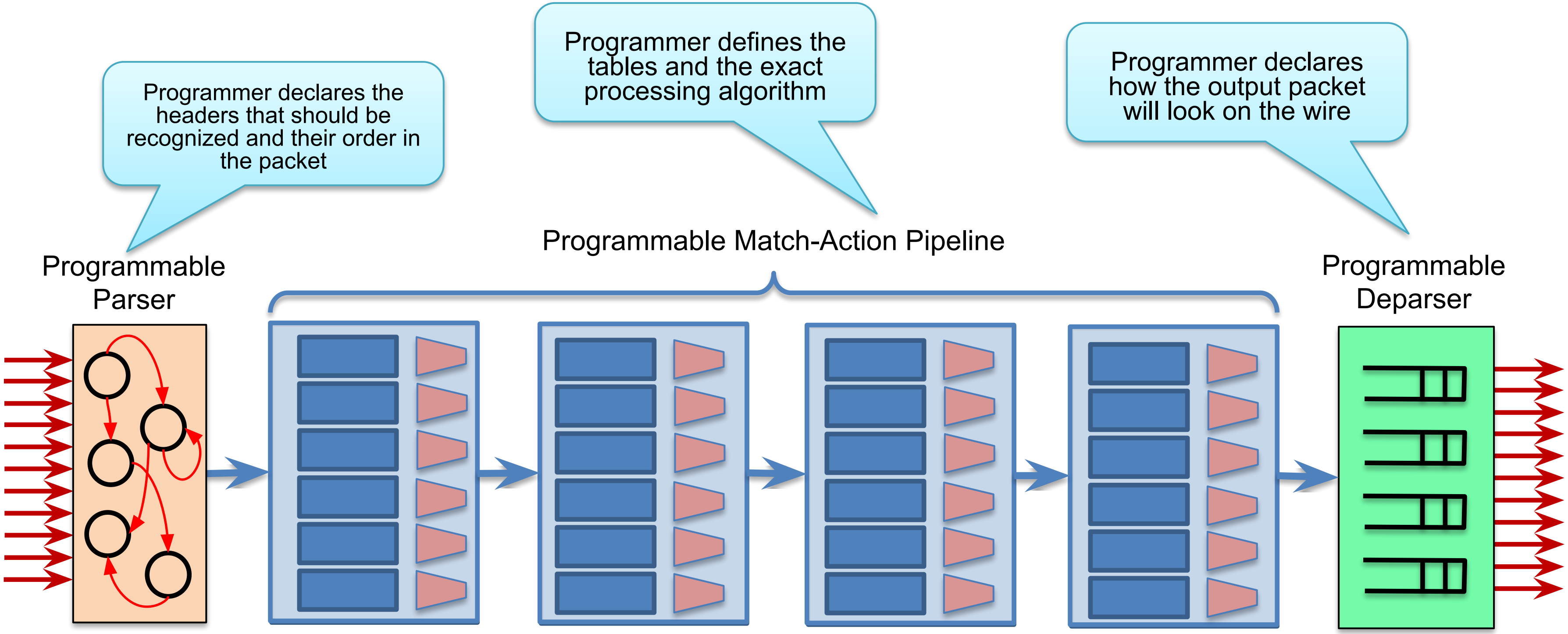
Brief History and Trivia

- May 2013: Initial idea and the name “P4”
- July 2014: First paper (SIGCOMM CCR)
- Aug 2014: First $P4_{14}$ Draft Specification (v0.9.8)
- Sep 2014: $P4_{14}$ Specification released (v1.0.0)
- Jan 2015: $P4_{14}$ v1.0.1
- Mar 2015: $P4_{14}$ v1.0.2
- Nov 2016: $P4_{14}$ v1.0.3
- May 2017: $P4_{14}$ v1.0.4

- Apr 2016: $P4_{16}$ – first commits
- Dec 2016: First $P4_{16}$ Draft Specification
- May 2017: $P4_{16}$ Specification released

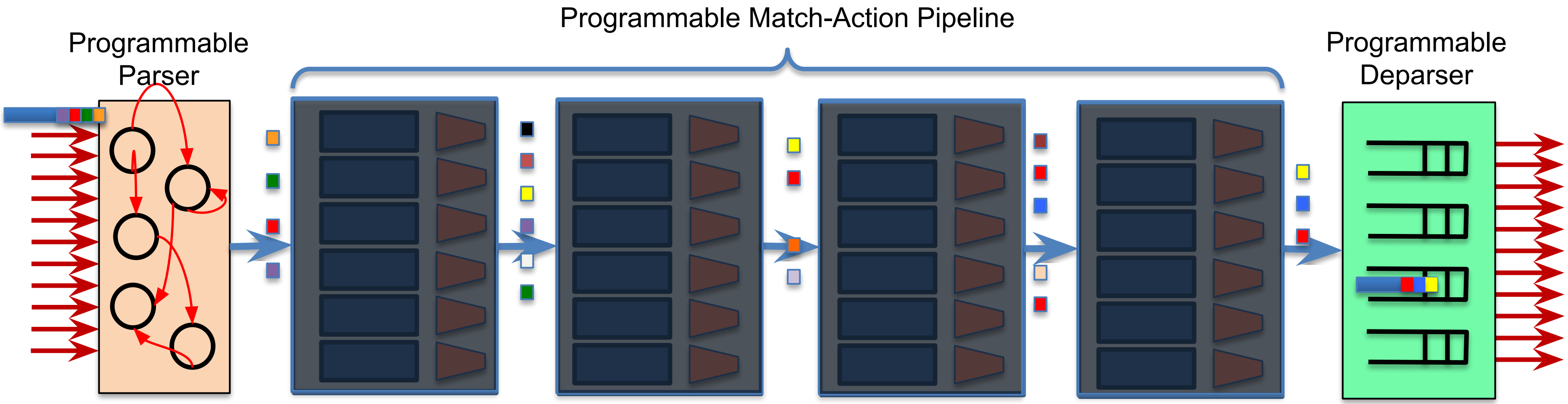
P4_16 Data Plane Model

PISA: Protocol-Independent Switch Architecture



PISA in Action

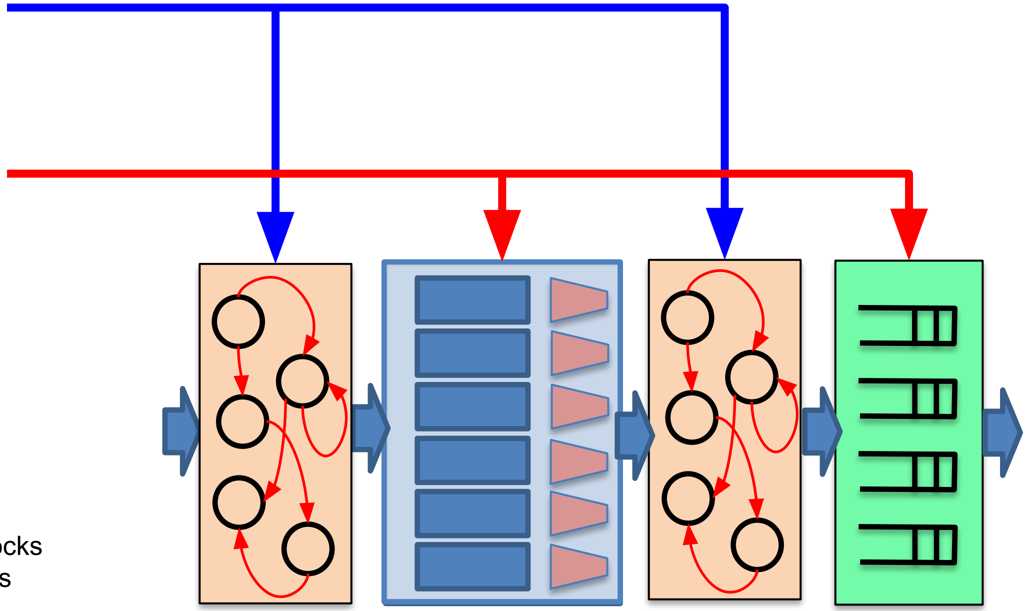
- Packet is parsed into individual headers (parsed representation)
- Headers and intermediate results can be used for matching and actions
- Headers can be modified, added or removed
- Packet is deparsed (serialized)



$P4_{16}$ Language Elements

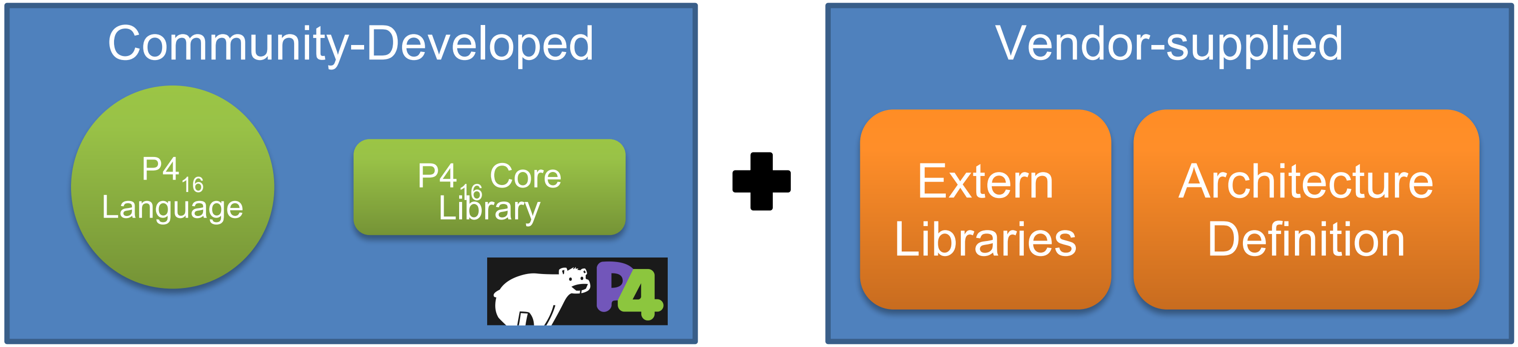
- **Parsers** — State machine, bitfield extraction
- **Controls** — Tables, Actions, control flow statements
- **Expressions** — Basic operations and operators
- **Data Types** — Bistrings, headers, structures, arrays

- **Architecture Description** — Programmable blocks and their interfaces
- **Extern Libraries** — Support for specialized components



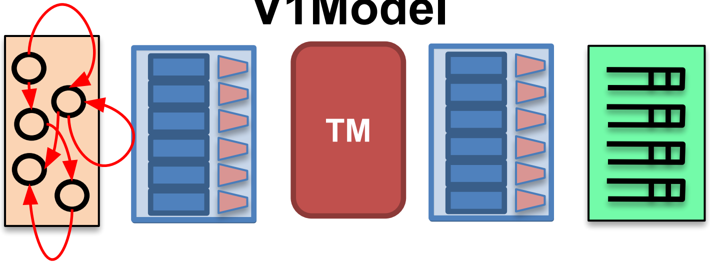
P4_16 Approach

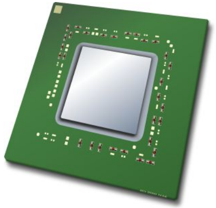
Term	Explanation
P4 Target	An embodiment of a specific hardware implementation
P4 Architecture	Provides an interface to program a target via some set of P4-programmable components, externs, fixed components



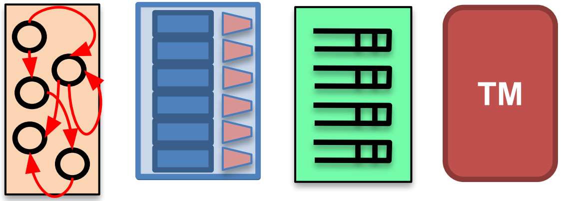
Example Architectures and Targets

V1Model

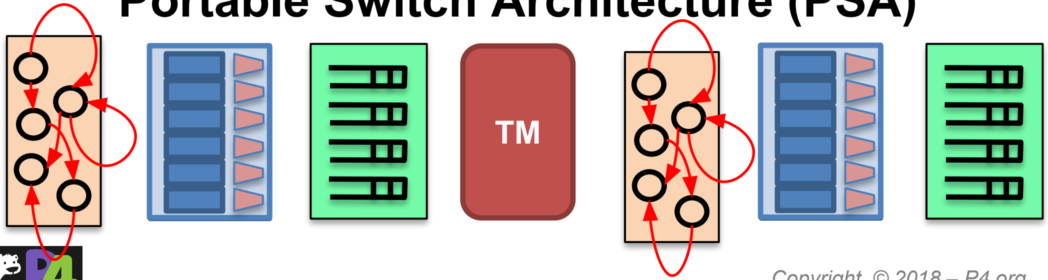




SimpleSumeSwitch

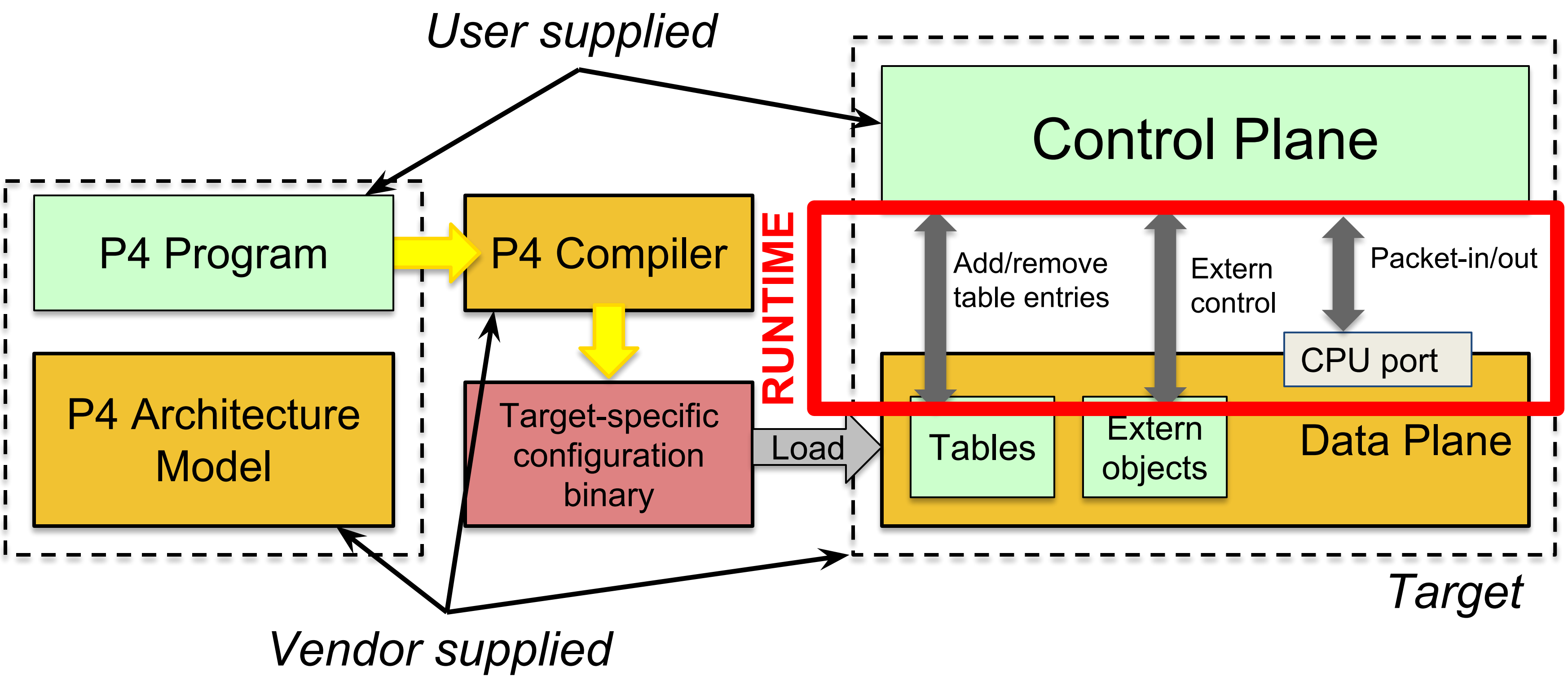


Portable Switch Architecture (PSA)



Anything

Programming a P4 Target



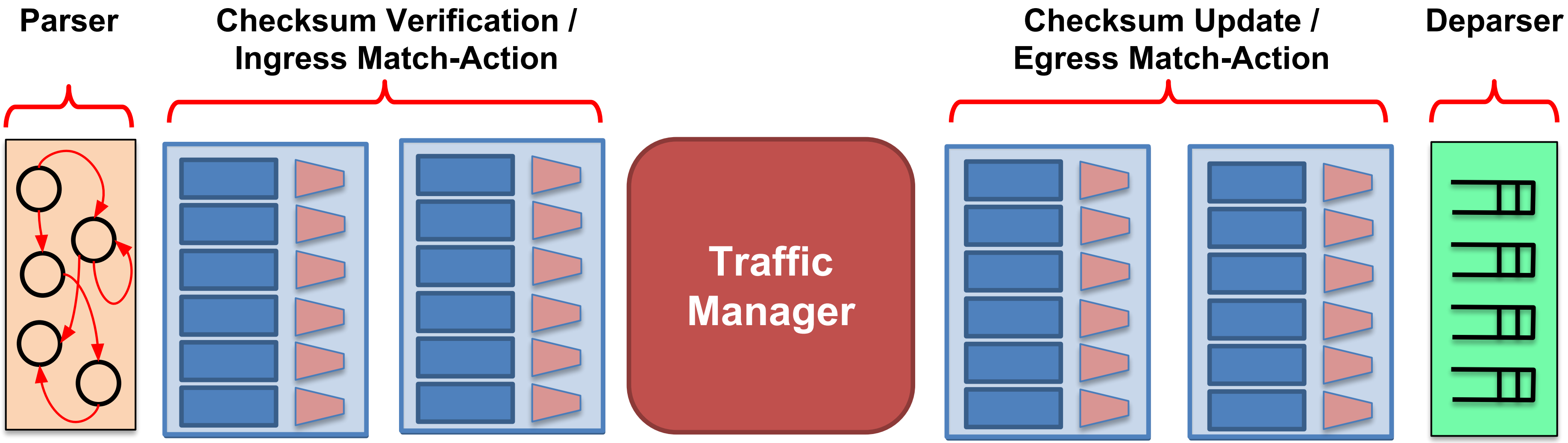
Lab 1: Basics

Before we start...

- **Install VM image (Look for instructor with USB sticks)**
- **Please make sure that your VM is up to date**
  - `$ cd ~/tutorials && git pull`
- **We'll be using several software tools pre-installed on the VM**
  - Bmv2: a P4 software switch
  - `p4c`: the reference P4 compiler
  - Mininet: a lightweight network emulation environment
- **Each directory contains a few scripts**
  - `$ make` : compiles P4 program, execute on Bmv2 in Mininet, populate tables
  - `*.py`: send and receive test packets in Mininet
- **Exercises**
  - Each example comes with an incomplete implementation; your job is to finish it!
  - Look for "TODOs" (or peek at the P4 code in `solution/` if you must)

V1Model Architecture

- **Implemented on top of Bmv2's `simple_switch` target**



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;
}
```

- **`ingress_port`** - the port on which the packet arrived
- **`egress_spec`** - the port to which the packet should be sent to
- **`egress_port`** - the port on which the packet is departing from (read only in egress pipeline)

$P4_{16}$ Program Template (V1Model)

```
#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;
```

P4$_{16}$ Hello World (V1Model)

```
#include <core.p4>
#include <v1model.p4>
struct metadata {}
struct headers {}

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

    state start { transition accept; }
}

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

control MyIngress(inout headers hdr,
   inout metadata meta,
   inout standard_metadata_t standard_metadata) {
apply {
        if (standard_metadata.ingress_port == 1) {
            standard_metadata.egress_spec = 2;
        } else if (standard_metadata.ingress_port == 2) {
            standard_metadata.egress_spec = 1;
        }
    }
}
```

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

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

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

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

P4$_{16}$ Hello World (V1Model)

```
#include <core.p4>
#include <v1model.p4>
struct metadata {}
struct headers {}

parser MyParser(packet_in packet, out headers hdr,
   inout metadata meta,
   inout standard_metadata_t standard_metadata) {
    state start { transition accept; }
}

control MyIngress(inout headers hdr, inout metadata meta,
   inout standard_metadata_t standard_metadata) {
    action set_egress_spec(bit<9> port) {
        standard_metadata.egress_spec = port;
    }
    table forward {
        key = { standard_metadata.ingress_port: exact; }
        actions = {
            set_egress_spec;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }
    apply {   forward.apply();    }
}
```

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

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

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

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

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

Key	Action Name	Action Data
1	set_egress_spec	2
2	set_egress_spec	1

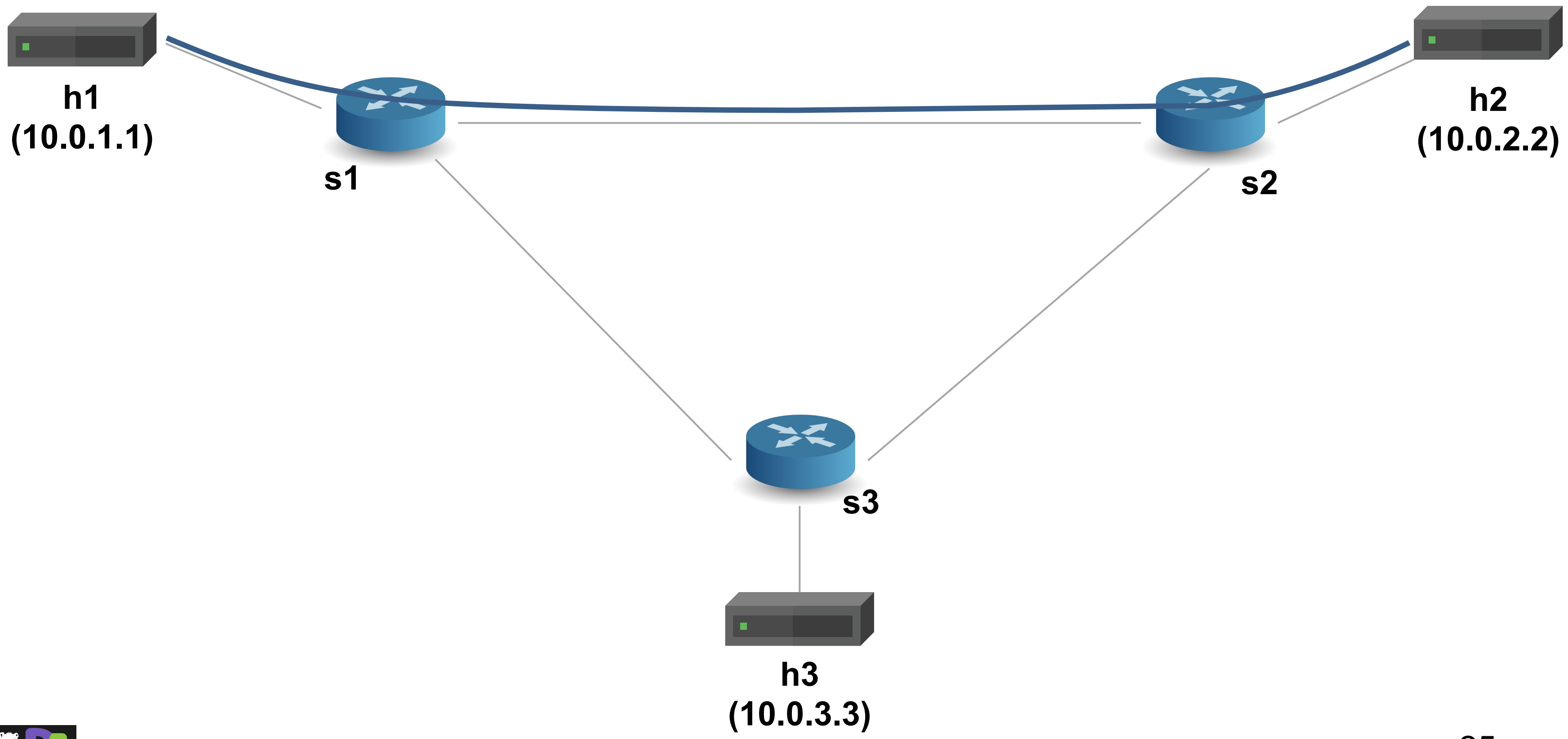
Running Example: Basic Forwarding

- **We'll use a simple application as a running example—a basic router—to illustrate the main features of $P4_{16}$**

- **Basic router functionality:**
  - Parse Ethernet and IPv4 headers from packet
  - Find destination in IPv4 routing table
  - Update source / destination MAC addresses
  - Decrement time-to-live (TTL) field
  - Set the egress port
  - Deparse headers back into a packet

- **We've written some starter code for you (`basic.p4`) and implemented a static control plane**

Basic Forwarding: Topology



P4$_{16}$ Types (Basic and Header Types)

```
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;
}
```

Basic Types

- **bit<n>**: Unsigned integer (bitstring) of size n
- **bit** is the same as **bit<1>**
- **int<n>**: Signed integer of size n (>=2)
- **varbit<n>**: Variable-length bitstring

Header Types: Ordered collection of members

- Can contain **bit<n>**, **int<n>**, and **varbit<n>**
- Byte-aligned
- Can be valid or invalid
- Provides several operations to test and set validity bit: **isValid()**, **setValid()**, and **setInvalid()**

Typedef: Alternative name for a type

P4$_{16}$ Types (Other Types)

```
/* Architecture */
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;
  ...
}

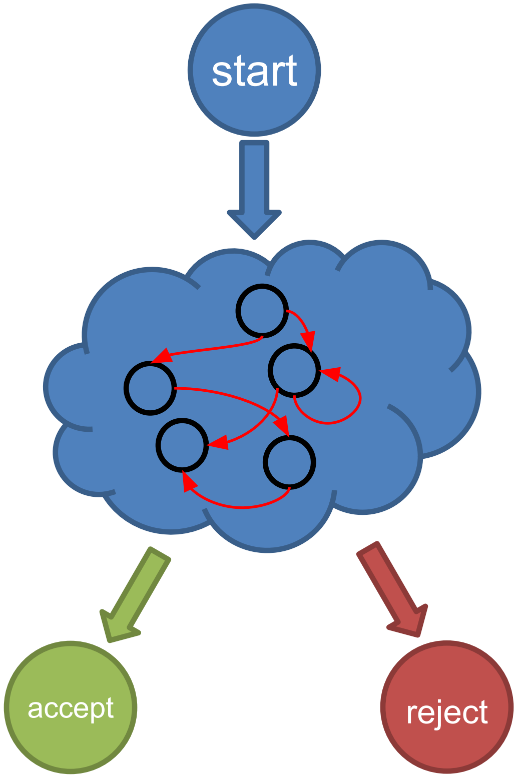
/* User program */
struct metadata {
  ...
}
struct headers {
  ethernet_t    ethernet;
  ipv4_t        ipv4;
}
```

Other useful types

- **Struct**: Unordered collection of members (with no alignment restrictions)
- **Header Stack:** array of headers
- **Header Union:** one of several headers

$P4_{16}$ Parsers

- Parsers are functions that map packets into headers and metadata, written in a state machine style
- Every parser has three predefined states
  - `start`
  - `accept`
  - `reject`
- Other states may be defined by the programmer
- In each state, execute zero or more statements, and then transition to another state (loops are OK)



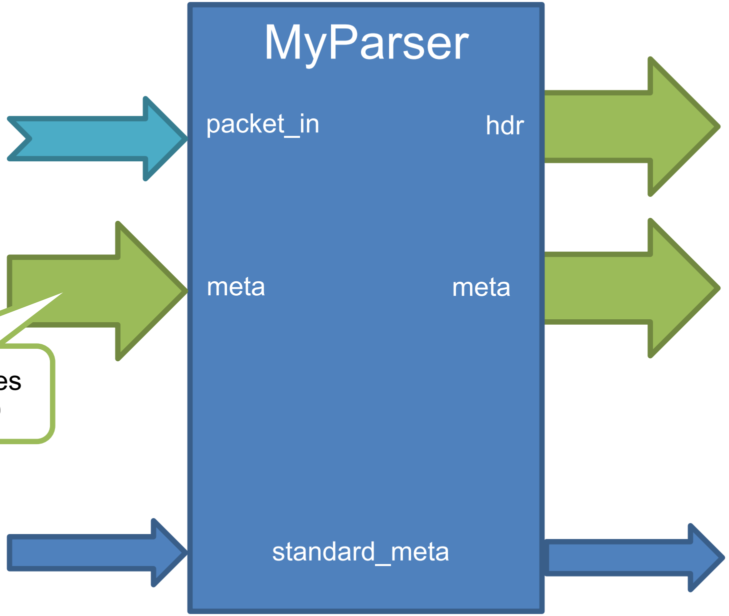
Parsers (V1Model)

```
/* From core.p4 */
extern packet_in {
  void extract<T>(out T hdr);
  void extract<T>(out T variableSizeHeader,
              in bit<32> variableFieldSizeInBits);
  T lookahead<T>();
  void advance(in bit<32> sizeInBits);
  bit<32> length();
}
/* User Program */
parser MyParser(packet_in packet,
          out headers hdr,
          inout metadata meta,
          inout standard_metadata_t std_meta) {

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

}
```

The platform Initializes User Metadata to 0



Select Statement

```
state start {
  transition parse_ethernet;
}

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

$P4_{16}$ has a `select` statement that can be used to branch in a parser

Similar to `case` statements in C or Java, but without "fall-through behavior"—i.e., `break` statements are not needed

In parsers it is often necessary to branch based on some of the bits just parsed

For example, etherType determines the format of the rest of the packet

Match patterns can either be literals or simple computations such as masks

Coding Break

P4$_{16}$ Controls

- **Similar to C functions (without loops)**
- **Can declare variables, create tables, instantiate externs, etc.**
- **Functionality specified by code in `apply` statement**
- **Represent all kinds of processing that are expressible as DAG:**
  - Match-Action Pipelines
  - Deparsers
  - Additional forms of packet processing (updating checksums)
- **Interfaces with other blocks are governed by user- and architecture-specified types (typically headers and metadata)**

Example: Reflector (V1Model)

```
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;
  }
}
```

Desired Behavior:

- **Swap source and destination MAC addresses**
- **Bounce the packet back out on the physical port that it came into the switch on**

Example: Simple Actions

```
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;
  }

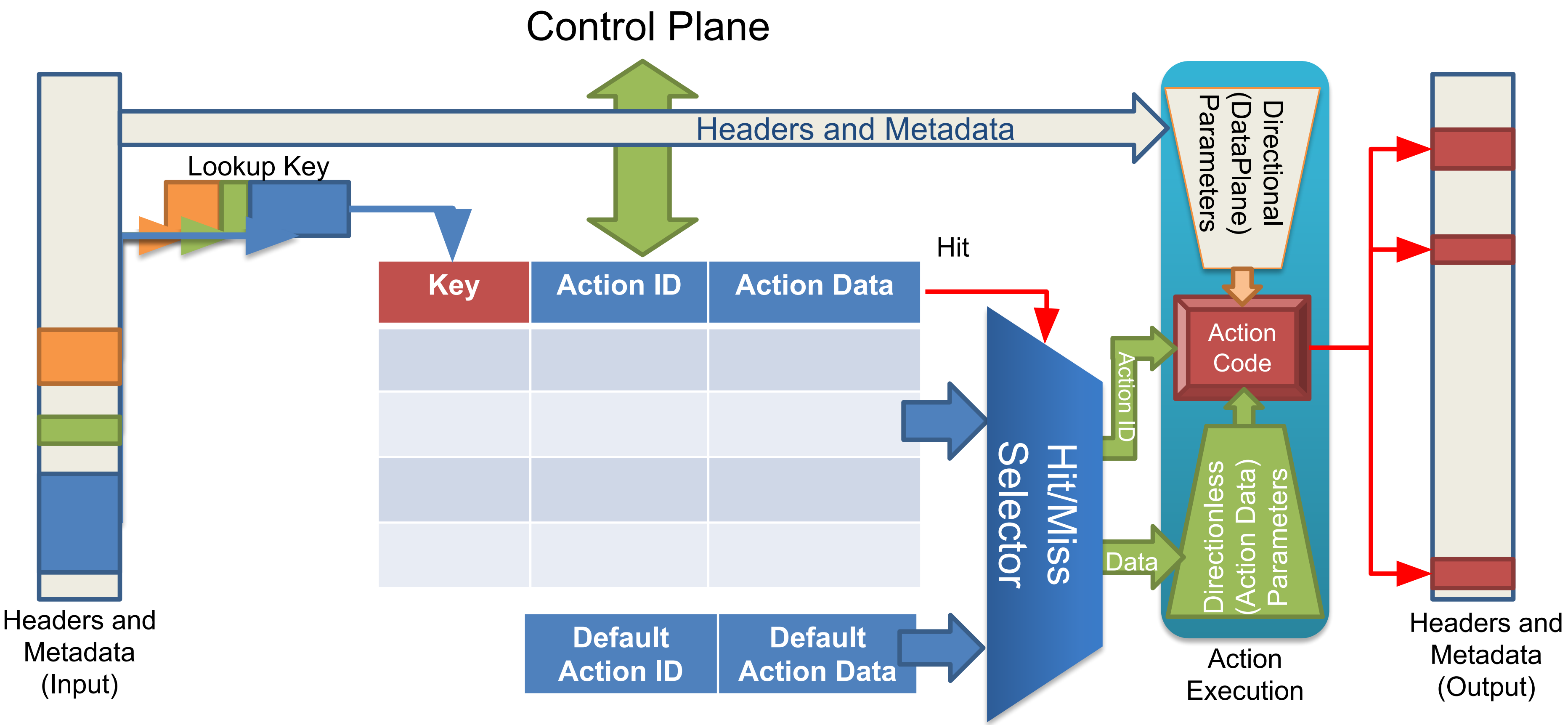
  apply {
    swap_mac(hdr.ethernet.srcAddr,
             hdr.ethernet.dstAddr);
    std_meta.egress_spec = std_meta.ingress_port;
  }
}
```

- **Very similar to C functions**
- **Can be declared inside a control or globally**
- **Parameters have type and direction**
- **Variables can be instantiated inside**
- **Many standard arithmetic and logical operations are supported**
  - +, -, *
  - ~, &, |, ^, >>, <<
  - ==, !=, >, >=, <, <=
  - No division/modulo
- **Non-standard operations:**
  - Bit-slicing: [m:l] (works as l-value too)
  - Bit Concatenation: ++

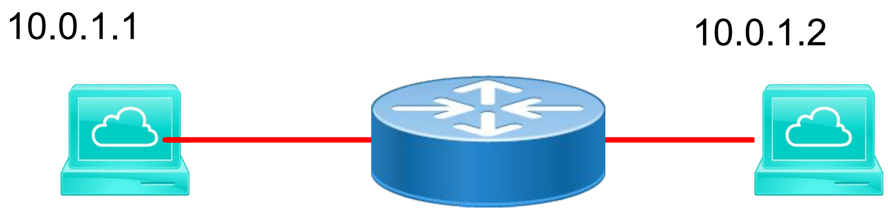
$P4_{16}$ Tables

- **The fundamental unit of a Match-Action Pipeline**
  - Specifies what data to match on and match kind
  - Specifies a list of *possible* actions
  - Optionally specifies a number of table **properties**
    - Size
    - Default action
    - Static entries
    - etc.
- **Each table contains one or more entries (rules)**
- **An entry contains:**
  - A specific key to match on
  - A **single** action that is executed when a packet matches the entry
  - Action data (possibly empty)

Tables: Match-Action Processing



Example: IPv4_LPM Table



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	

- **Data Plane (P4) Program**
  - Defines the format of the table
    - Key Fields
    - Actions
    - Action Data
  - Performs the lookup
  - Executes the chosen action
- **Control Plane (IP stack, Routing protocols)**
  - Populates table entries with specific information
    - Based on the configuration
    - Based on automatic discovery
    - Based on protocol calculations

IPv4_LPM Table

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

Match Kinds

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

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

/* Some other architecture */
match_kind {
    regexp,
    fuzzy
}
```

- **The type `match_kind` is special in P4**
- **The standard library (`core.p4`) defines three standard match kinds**
  - Exact match
  - Ternary match
  - LPM match
- **The architecture (`v1model.p4`) defines two additional match kinds:**
  - range
  - selector
- **Other architectures may define (and provide implementation for) additional match kinds**

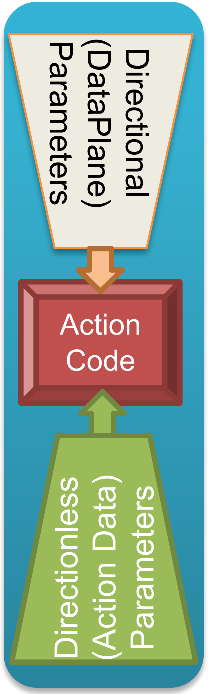
Defining Actions for L3 forwarding

```
/* core.p4 */
action NoAction() {
}

/* basic.p4 */
action drop() {
  mark_to_drop();
}

/* basic.p4 */
action ipv4_forward(macAddr_t dstAddr,
                    bit<9> port) {
  ...
}
```

- **Actions can have two different types of parameters**
  - Directional (from the Data Plane)
  - Directionless (from the Control Plane)
- **Actions that are called directly:**
  - Only use directional parameters
- **Actions used in tables:**
  - Typically use directionless parameters
  - May sometimes use directional parameters too



Action Execution

Applying Tables in Controls

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
  table ipv4_lpm {
    ...
  }
  apply {
     ...
     ipv4_lpm.apply();
     ...
  }
}
```

$P4_{16}$ Deparsing

```
/* From core.p4 */
extern packet_out {
  void emit<T>(in T hdr);
}

/* User Program */
control DeparserImpl(packet_out packet,
                     in headers hdr) {
  apply {
    ...
    packet.emit(hdr.ethernet);
    ...
  }
}
```

- **Assembles the headers back into a well-formed packet**
- **Expressed as a control function**
  - No need for another construct!
- **packet_out extern is defined in core.p4:** emit(hdr): serializes header if it is valid
- **Advantages:**
  - Makes deparsing explicit...
    ...but decouples from parsing

Coding Break

Basic Tunneling

- Add support for basic tunneling to the basic IP router

- Define a new header type (`myTunnel`) to encapsulate the IP packet

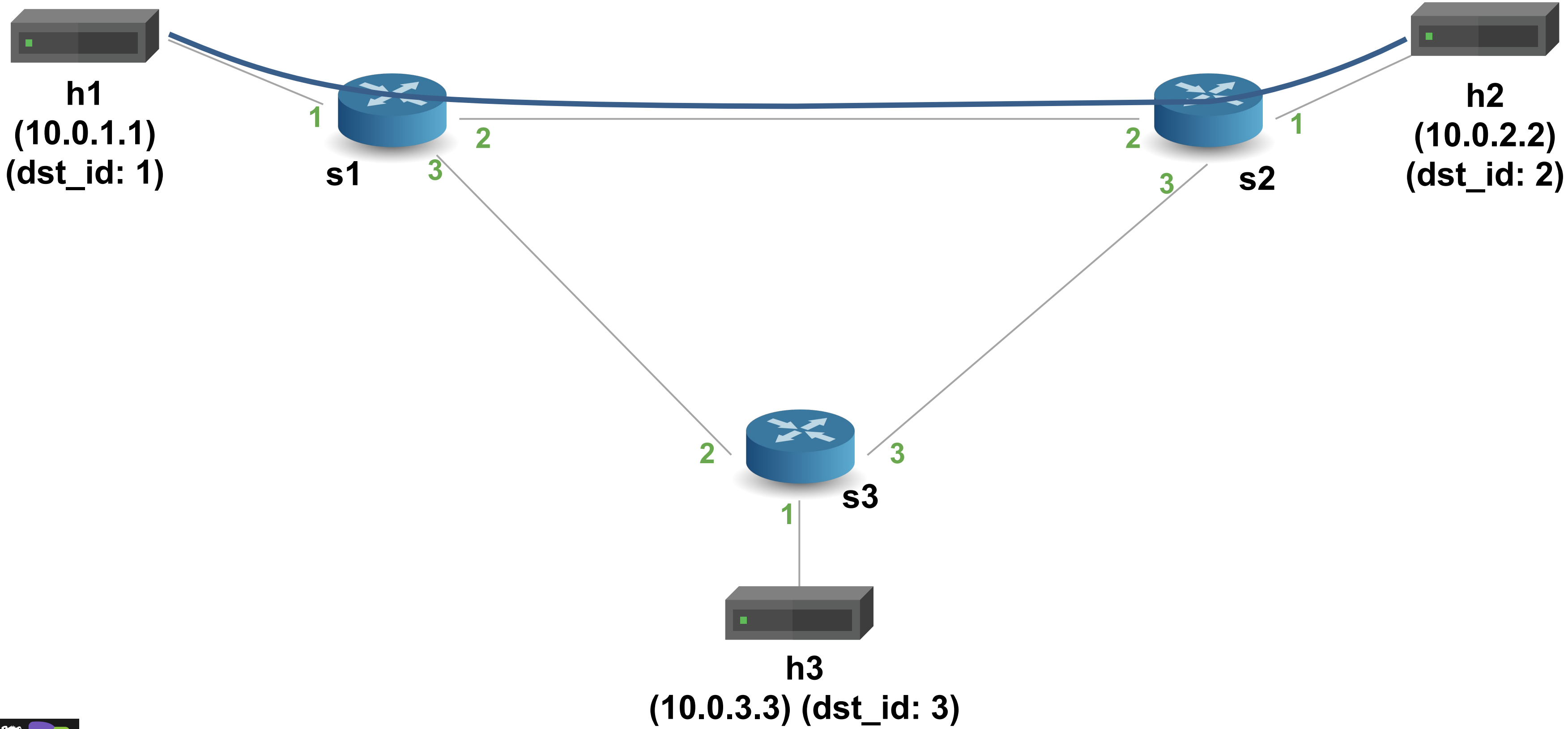
- `myTunnel` header includes:
    - `proto_id` : type of packet being encapsulated
    - `dst_id` : ID of destination host

- Modify the switch to perform routing using the `myTunnel` header

Basic Tunneling TODO List

- Define `myTunnel_t` header type and add to `headers` struct
- Update parser
- Define `myTunnel_forward` action
- Define `myTunnel_exact` table
- Update table application logic in `MyIngress` `apply` statement
- Update deparser
- Adding forwarding rules

Basic Forwarding: Topology



Coding Break

FAQs

- **Can I apply a table multiple times in my P4 Program?**
  - No (except via resubmit / recirculate)
- **Can I modify table entries from my P4 Program?**
  - No (except for direct counters)
- **What happens upon reaching the `reject` state of the parser?**
  - Architecture dependent
- **How much of the packet can I parse?**
  - Architecture dependent

Fin!

Debugging

```
control MyIngress(...) {
  table debug {
    key = {
      std_meta.egress_spec : exact;
    }
    actions = { }
  }
  apply {
     ...
     debug.apply();
  }
}
```

- **Bmv2 maintains logs that keep track of how packets are processed in detail**
  - `/tmp/p4s.s1.log`
  - `/tmp/p4s.s2.log`
  - `/tmp/p4s.s3.log`

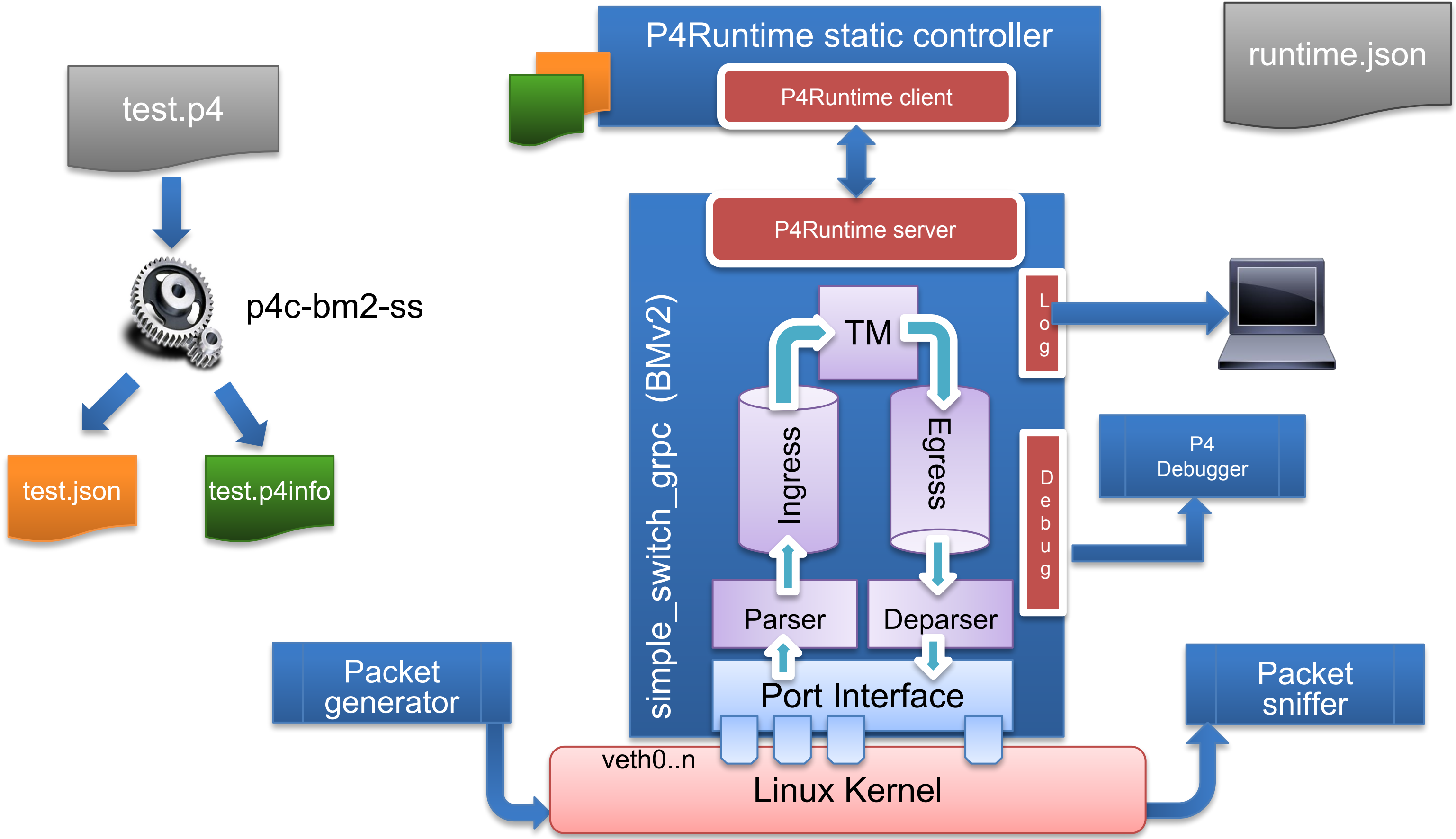
- **Can manually add information to the logs by using a dummy debug table that reads headers and metadata of interest**

  - ```
    [15:16:48.145] [bmv2] [D]
    [thread 4090] [96.0] [cxt 0]
    Looking up key:
    * std_meta.egress_spec : 2
    ```
    ```

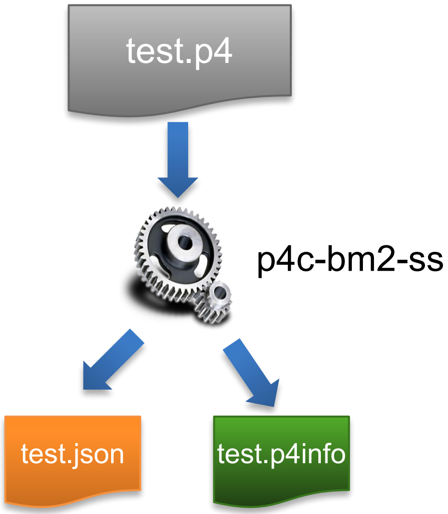
Lab 2: P4Runtime

P4 Software Tools

Makefile: under the hood

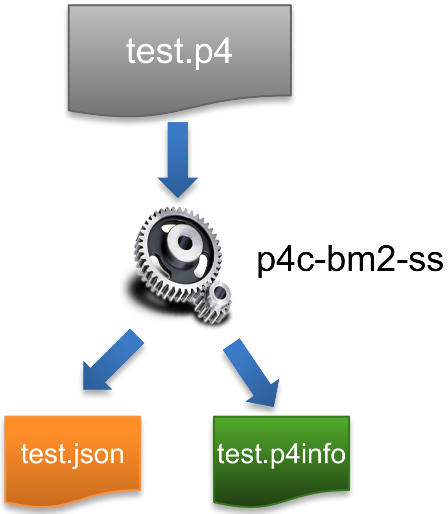


Step 1: P4 Program compilation



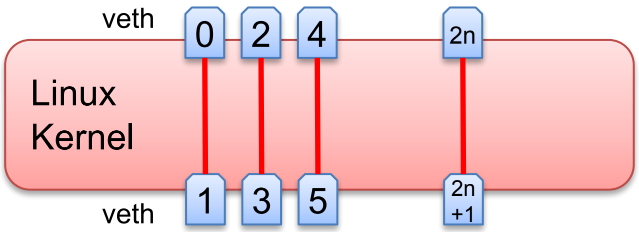

```
$ p4c-bm2-ss --p4v 16 \
    -o test.json \
    --p4runtime-file test.p4info \
    --p4runtime-format text \
    test.p4
```

Step 2: Preparing veth Interfaces

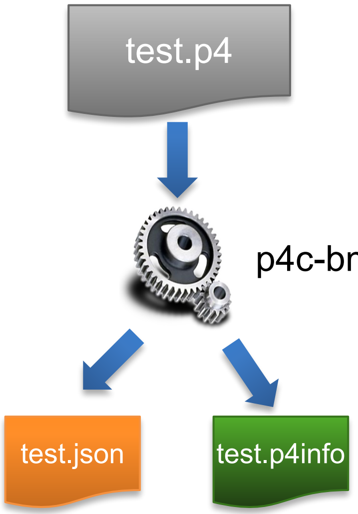



```
$ sudo ~/p4lang/tutorials/examples/veth_setup.sh

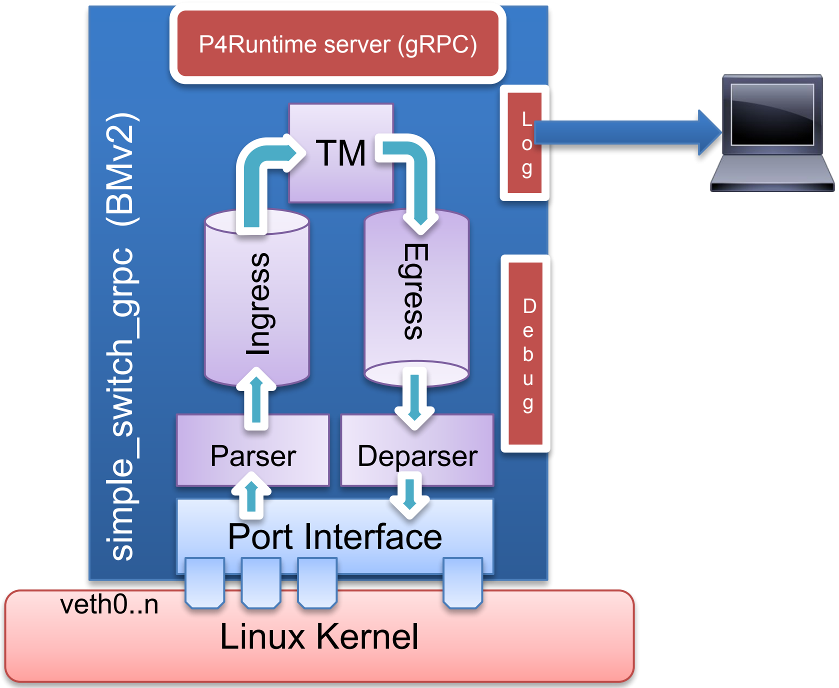
# ip link add name veth0 type veth peer name veth1
# for iface in "veth0 veth1"; do
      ip link set dev ${iface} up
      sysctl net.ipv6.conf.${iface}.disable_ipv6=1
      TOE_OPTIONS="rx tx sg tso ufo gso gro lro rxvlan txvlan rxhash"
      for TOE_OPTION in $TOE_OPTIONS; do
          /sbin/ethtool --offload $intf "$TOE_OPTION"
      done
  done
```



Step 3: Starting BMv2

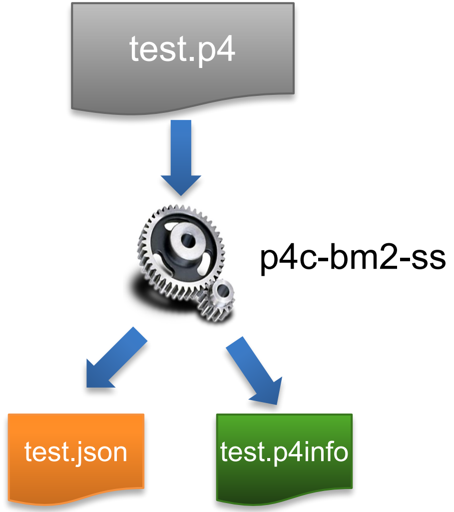


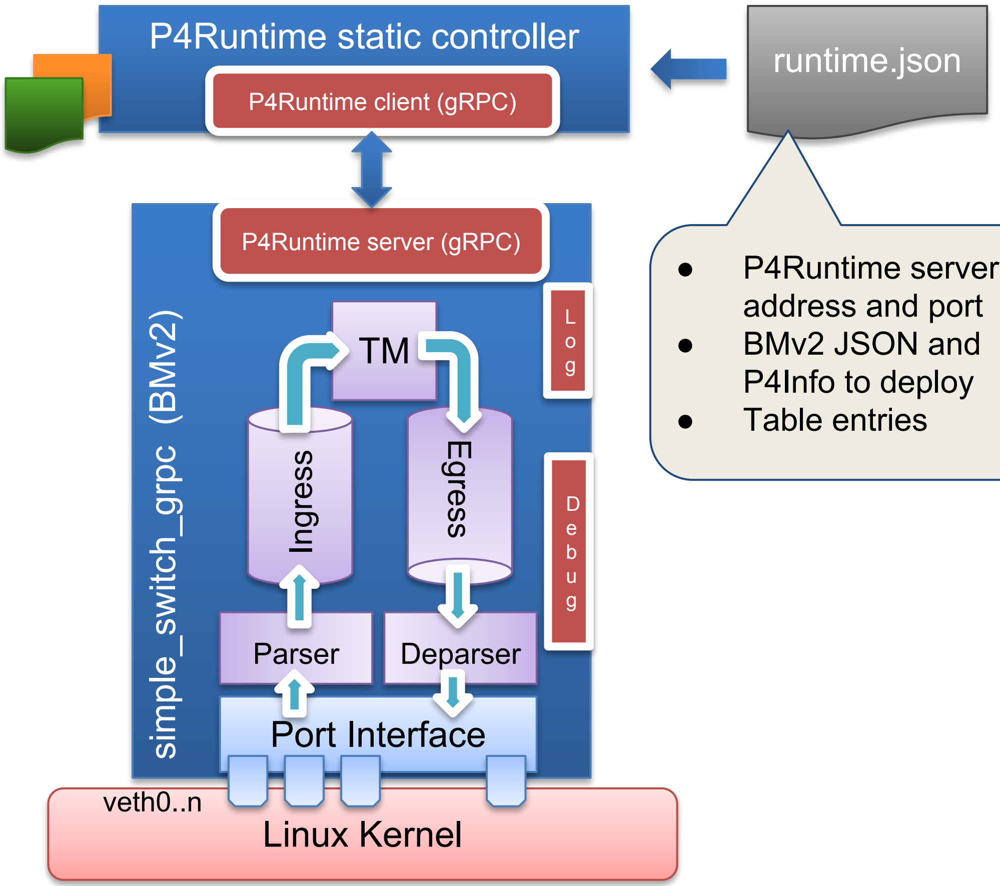

```
$ sudo simple_switch_grpc --log-console --dump-packet-data 64 \
       -i 0@veth0 -i 1@veth2 … [--pcap]  --no-p4              \
       -- --grpc-server-addr 0.0.0.0:50051 --cpu-port 255     \
       test.json
```



Step 4: Starting P4Runtime static controller

```
$ python $(RUN_SCRIPT) -t
$(TOPO) $(run_args)
```



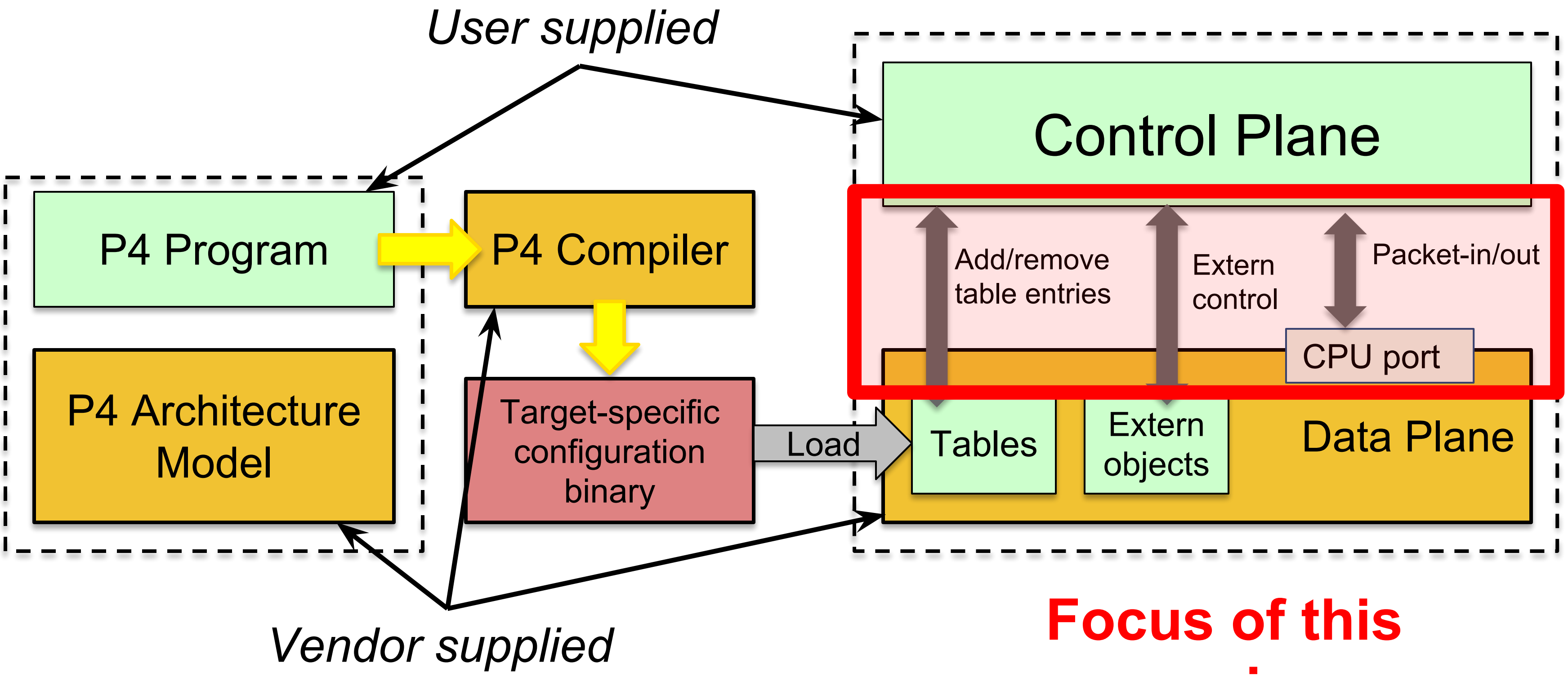


- P4Runtime server address and port
- BMv2 JSON and P4Info to deploy
- Table entries

P4Runtime

- API overview
- Workflow
- Exercise - Tunneling

Runtime control of P4 data planes

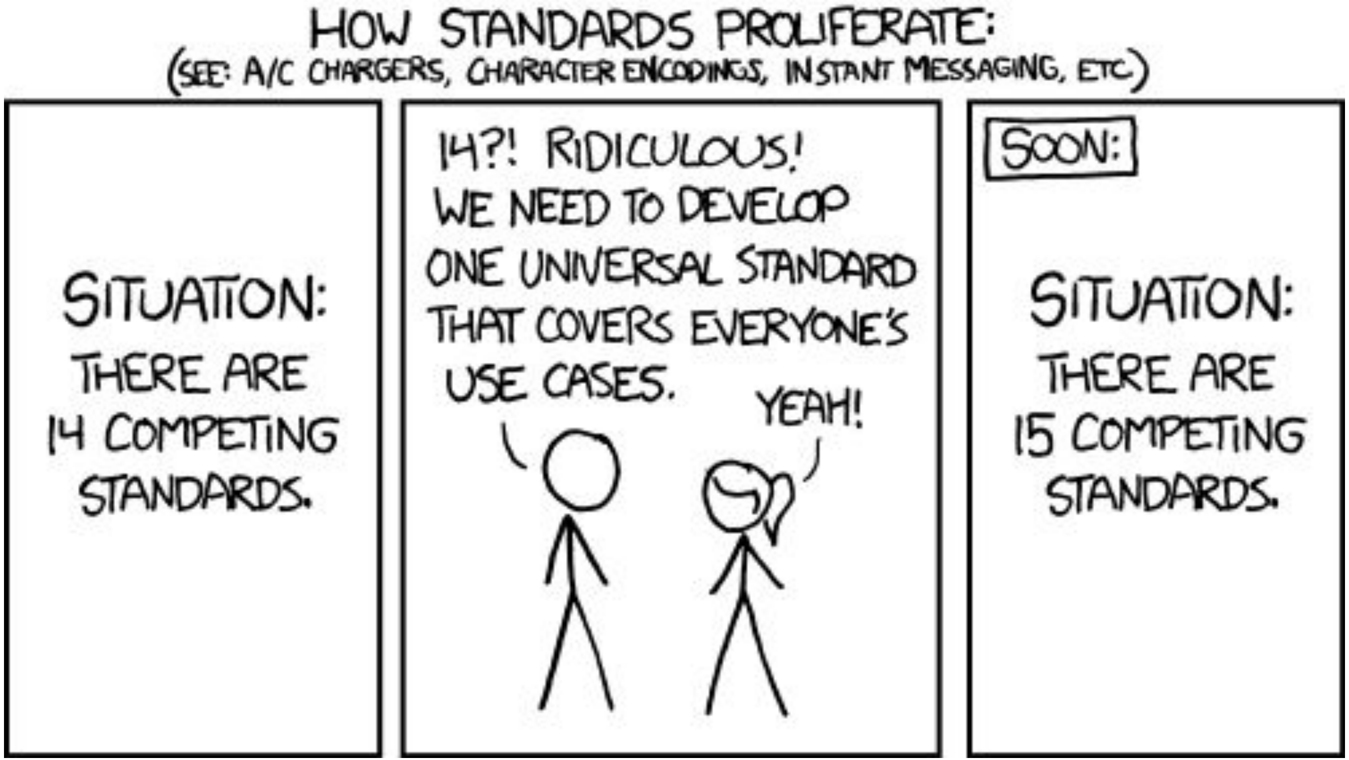


Focus of this session

Existing approaches to runtime control

- **P4 compiler auto-generated runtime APIs**
    - Program-dependent -- hard to provision new P4 program without restarting the control plane!
- **BMv2 CLI**
    - Program-independent, but target-specific -- control plane not portable!
- **OpenFlow**
    - Target-independent, but protocol-dependent -- protocol headers and actions baked in the specification!
- **OCP Switch Abstraction Interface (SAI)**
    - Target-independent, but protocol-dependent

Why do we need another data plane control API?

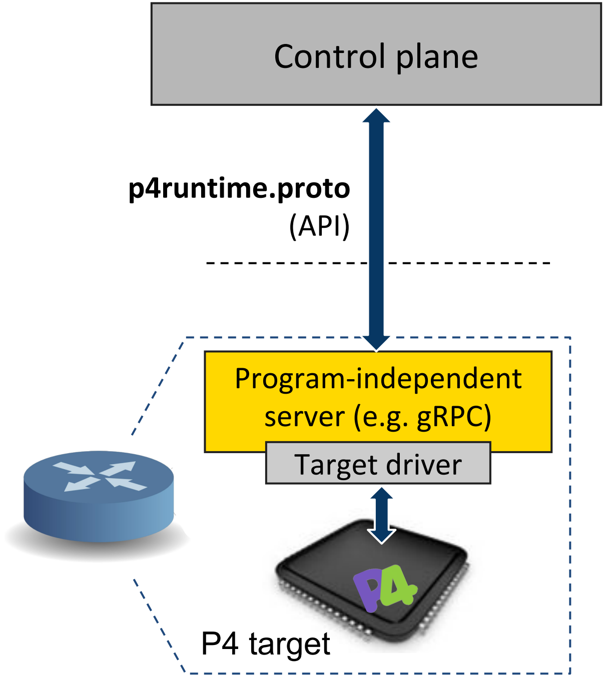


Properties of a runtime control API

API	Target-independent	Protocol-independent
P4 compiler auto-generated	✔	✘
BMv2 CLI	✘	✔
OpenFlow	✔	✘
SAI	✔	✘
P4Runtime	✔	✔

What is P4Runtime?

- **Framework for runtime control of P4 targets**
  - Open-source API + server implementation
    https://github.com/p4lang/PI
  - Initial contribution by Google and Barefoot
- **Work-in-progress by the p4.org API WG**
  - Draft of version 1.0 available
- **Protobuf-based API definition**
  - p4runtime.proto
  - gRPC transport
- **P4 program-independent**
  - API doesn't change with the P4 program
- **Enables field-reconfigurability**
  - Ability to push new P4 program without recompiling the software stack of target switches



Protocol Buffers (protobuf) Basics

- Language for describing data for serialization in a structured way
- Common binary wire-format
- Language-neutral
  - Code generators for: *Action Script, C, C++, C#, Clojure, Lisp, D, Dart, Erlang, Go, Haskell, Java, Javascript, Lua, Objective C, OCaml, Perl, PHP, Python, Ruby, Rust, Scala, Swift, Visual Basic, ...*
- Platform-neutral
- Extensible and backwards compatible
- Strongly typed

```
syntax = "proto3";

message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;

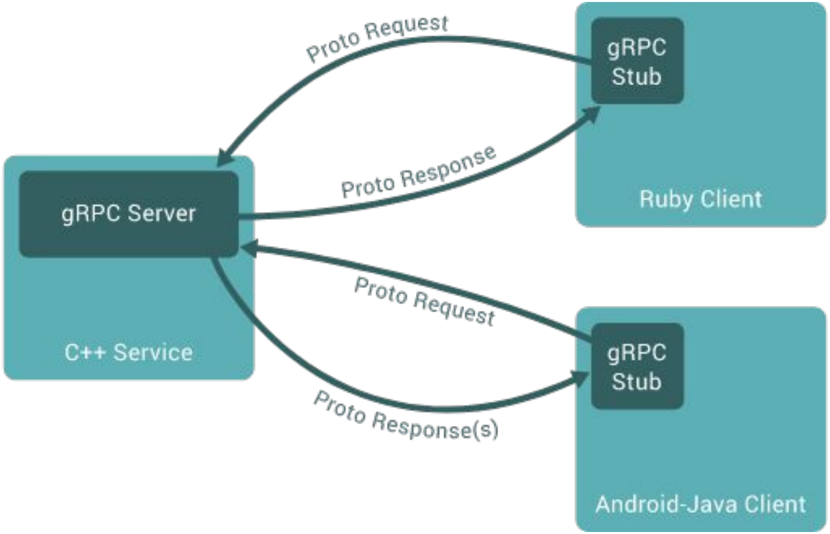
  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber phone = 4;
}
```

gRPC Basics

- Use Protocol Buffers to define service API and messages
- Automatically generate native stubs in:
  - C / C++
  - C#
  - Dart
  - Go
  - Java
  - Node.js
  - PHP
  - Python
  - Ruby



- Transport over HTTP/2.0 and TLS
  - Efficient single TCP connection implementation that supports bidirectional streaming

gRPC Service Example

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

More details here: https://grpc.io/docs/guides/

P4Runtime Service

Enables a local or remote entity to arbitrate mastership, load the pipeline/program, send/receive packets, and read and write forwarding table entries, counters, and other P4 entities.

```
service P4Runtime {
 rpc Write(WriteRequest) returns (WriteResponse) {}
 rpc Read(ReadRequest) returns (stream ReadResponse) {}
 rpc SetForwardingPipelineConfig(SetForwardingPipelineConfigRequest)
     returns (SetForwardingPipelineConfigResponse) {}
 rpc GetForwardingPipelineConfig(GetForwardingPipelineConfigRequest)
     returns (GetForwardingPipelineConfigResponse) {}
 rpc StreamChannel(stream StreamMessageRequest)
     returns (stream StreamMessageResponse) {}
}
```

P4Runtime Service

P4Runtime Protobuf Definition:

https://github.com/p4lang/p4runtime/blob/master/proto/p4/v1/p4runtime.proto

Service Specification:

Working draft of version 1.0 is available now

https://p4.org/p4-spec/docs/P4Runtime-v1.0.0.pdf

P4Runtime `Write` Request

```
message WriteRequest {
 uint64 device_id = 1;
 uint64 role_id = 2;
 Uint128 election_id = 3;
 repeated Update updates = 4;
}
```

```
message Update {
 enum Type {
   UNSPECIFIED = 0;
   INSERT = 1;
   MODIFY = 2;
   DELETE = 3;
 }
 Type type = 1;
 Entity entity = 2;
}
```

```
message Entity {
 oneof entity {
   ExternEntry extern_entry = 1;
   TableEntry table_entry = 2;
   ActionProfileMember
         action_profile_member = 3;
   ActionProfileGroup
         action_profile_group = 4;
   MeterEntry meter_entry = 5;
   DirectMeterEntry direct_meter_entry = 6;
   CounterEntry counter_entry = 7;
   DirectCounterEntry direct_counter_entry = 8;
   PacketReplicationEngineEntry
         packet_replication_engine_entry = 9;
   ValueSetEntry value_set_entry = 10;
   RegisterEntry register_entry = 11;
 }
}
```

P4Runtime Table Entry

p4runtime.proto simplified excerpts:

```
message TableEntry {
  uint32 table_id;
  repeated FieldMatch match;
  Action action;
  int32 priority;
  ...
}
```

```
message Action {
  uint32 action_id;
  message Param {
    uint32 param_id;
    bytes value;
  }
  repeated Param params;
}
```

```
message FieldMatch {
  uint32 field_id;
  message Exact {
    bytes value;
  }
  message Ternary {
    bytes value;
    bytes mask;
  }
  ...
  oneof field_match_type {
    Exact exact;
    Ternary ternary;
    ...
  }
}
```

Full protobuf definition:
https://github.com/p4lang/PI/blob/master/proto/p4/p4runtime.proto

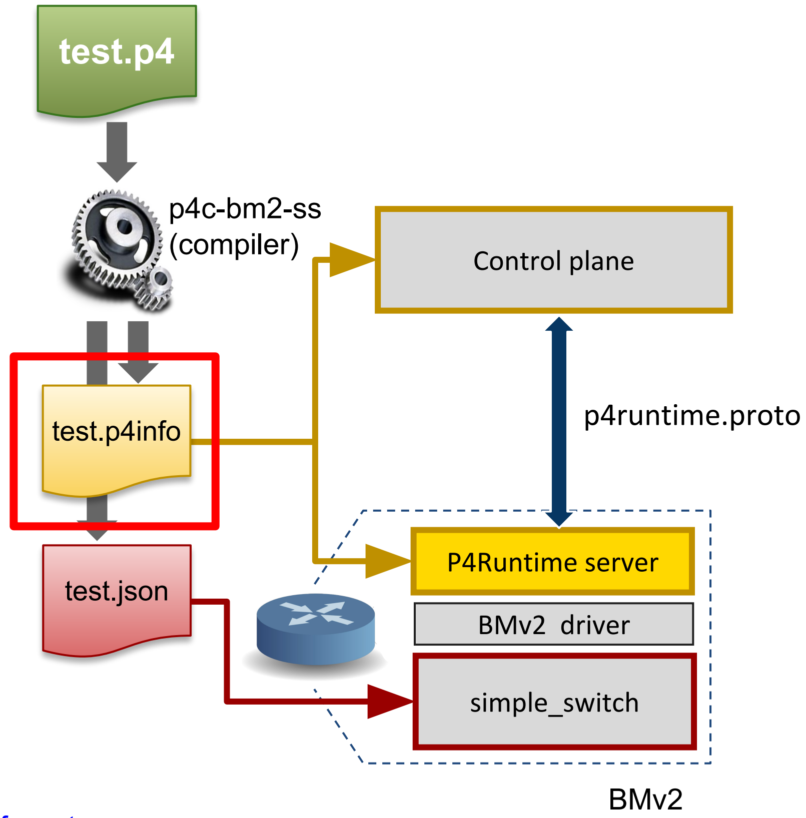
To add a table entry, the control plane needs to know:

- **IDs of P4 entities**
  - Tables, field matches, actions, params, etc.
- **Field matches for the particular table**
  - Match type, bitwidth, etc.
- **Parameters for the particular action**
- **Other P4 program attributes**

P4Runtime workflow

P4Info

- **Captures P4 program attributes needed at runtime**
  - IDs for tables, actions, params, etc.
  - Table structure, action parameters, etc.
- **Protobuf-based format**
- **Target-independent compiler output**
  - Same P4Info for BMv2, 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) {
   /* Action implementation */
}

...

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

P4 compiler

basic_router.p4info

```
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 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)
```

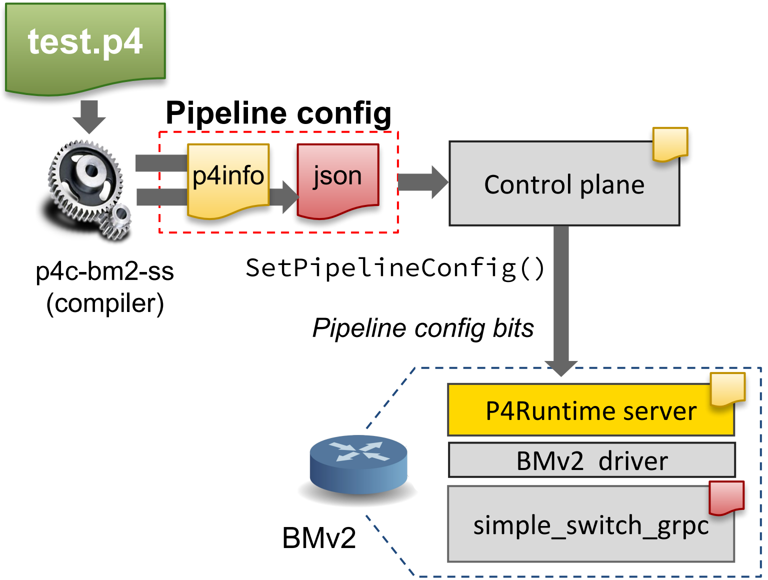
Control plane generates

Protobuf message

```
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 StreamChannel

```
message StreamMessageRequest {
 oneof update {
   MasterArbitrationUpdate
          arbitration = 1;
   PacketOut packet = 2;
   DigestListAck digest_ack = 3;
 }
}
```

```
// Packet sent from the controller to the switch.
message PacketOut {
 bytes payload = 1;
 // This will be based on P4 header annotated as
 // @controller_header("packet_out").
 // At most one P4 header can have this annotation.
 repeated PacketMetadata metadata = 2;
}
```

```
message StreamMessageResponse {
 oneof update {
   MasterArbitrationUpdate
          arbitration = 1;
   PacketIn packet = 2;
   DigestList digest = 3;
 }
}
```

```
// Packet sent from the switch to the controller.
message PacketIn {
 bytes payload = 1;
 // This will be based on P4 header annotated as
 // @controller_header("packet_in").
 // At most one P4 header can have this annotation.
 repeated PacketMetadata metadata = 2;
}
```

P4Runtime Common Parameters

- `device_id`
  - Specifies the specific forwarding chip or software bridge
  - **Set to 0 for single chip platforms**
- `role_id`
  - Corresponds to a role with specific capabilities (i.e. what operations, P4 entities, behaviors, etc. are in the scope of a given role)
  - Role definition is currently agreed upon between control and data planes offline
  - **Default role_id (0) has full pipeline access**
- `election_id`
  - P4Runtime supports mastership on a per-role basis
  - Client with the highest election ID is referred to as the "master", while all other clients are referred to as "slaves"
  - **Set to 0 for single instance controllers**

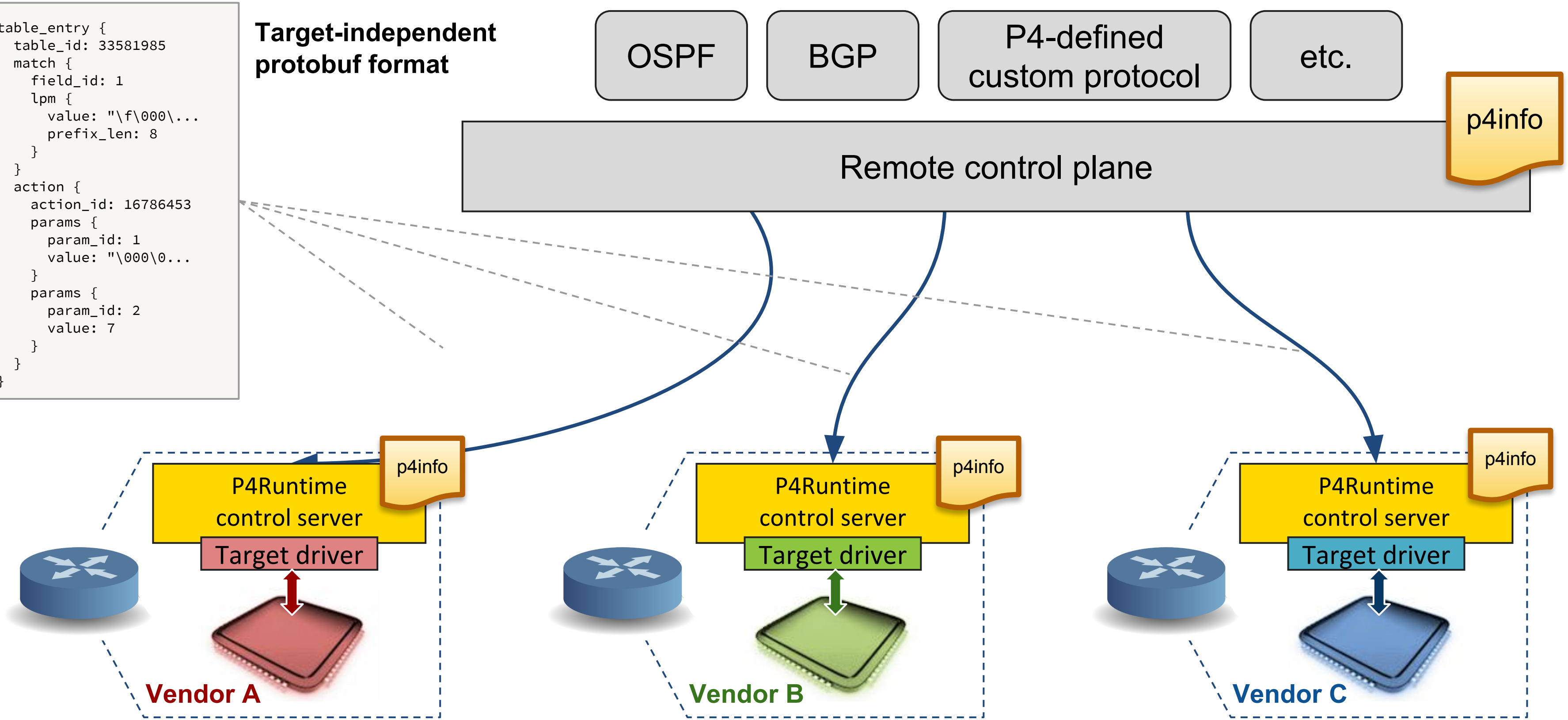
Mastership Arbitration

- **Upon connecting to the device, the client (e.g. controller) needs to open a `StreamChannel`**
- **The client must advertise its `role_id` and `election_id` using a `MasterArbitrationUpdate` message**
  - If `role_id` is not set, it implies the default role and will be granted full pipeline access
  - The `election_id` is opaque to the server and determined by the control plane (can be omitted for single-instance control plane)
- **The switch marks the client for each role with the highest `election_id` as master**
- **Master can:**
  - Perform `Write` requests
  - Receive `PacketIn` messages
  - Send `PacketOut` messages

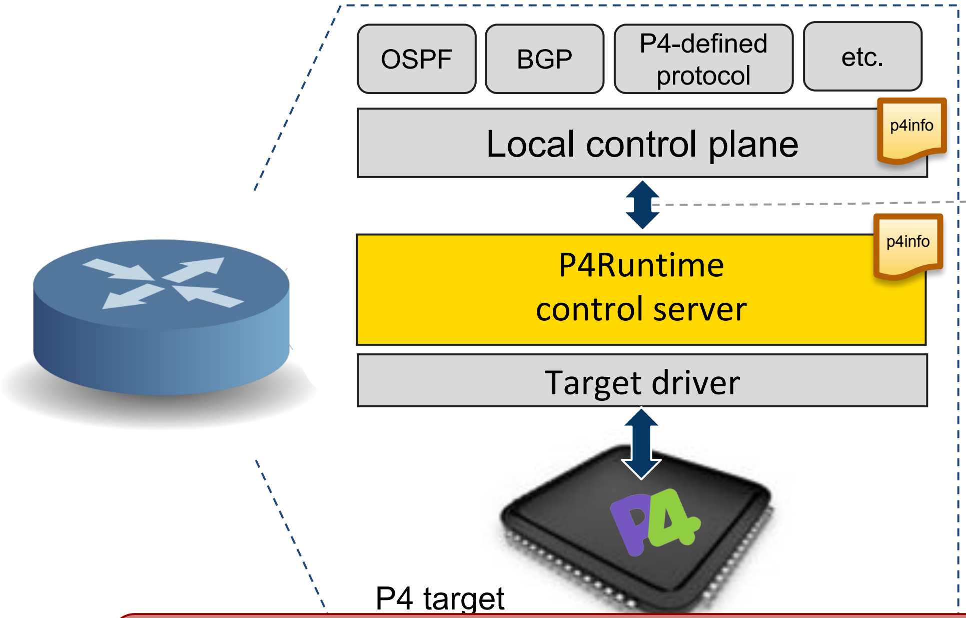
Remote control

```
table_entry {
  table_id: 33581985
  match {
    field_id: 1
    lpm {
      value: "\f\000\...
      prefix_len: 8
    }
  }
  action {
    action_id: 16786453
    params {
      param_id: 1
      value: "\000\0...
    }
    params {
      param_id: 2
      value: 7
    }
  }
}
```

Target-independent protobuf format



Local control



Same target-independent protobuf format

```
table_entry {
  table_id: 33581985
  match {
    field_id: 1
    lpm {
      value: "\f\000\...
      prefix_len: 8
    }
  }
  action {
    action_id: 16786453
    params {
      param_id: 1
      value: "\000\0...
    }
    params {
      param_id: 2
      value: 7
    }
  }
}
```

The P4Runtime API can be used equally well by a remote or local control plane

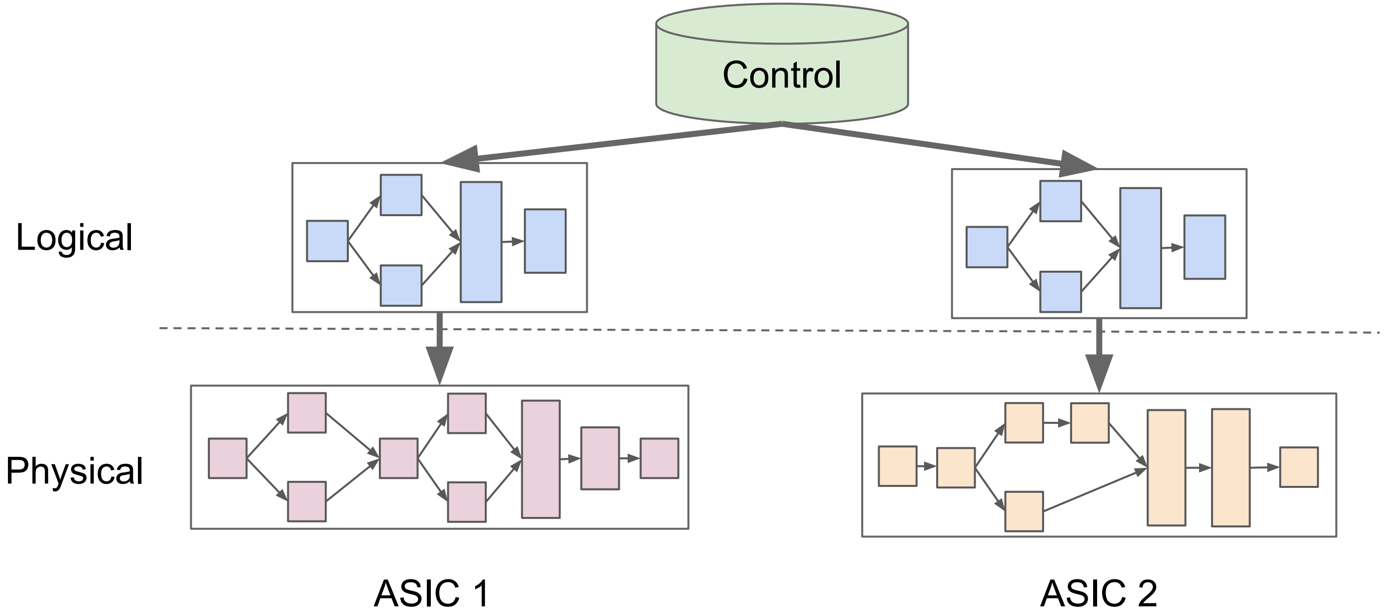
P4Runtime on Fixed-Function Chips

TODO pull some stratum slides

Slide from Google

P4 Program as Fixed-Function Chip Abstraction

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



P4Runtime API recap

Things we covered:

- **P4Info**
- **Table entries**
- **Set pipeline config**

What we didn't cover:

- **How to control other P4 entities**
  - Externs, counters, meters
- **Packet-in/out support**
- **Controller replication**
  - Via master-slave arbitration
- **Batched reads/writes**
- **Switch configuration**
  - Outside the P4Runtime scope
  - Achieved with other mechanisms
    - e.g., OpenConfig and gNMI

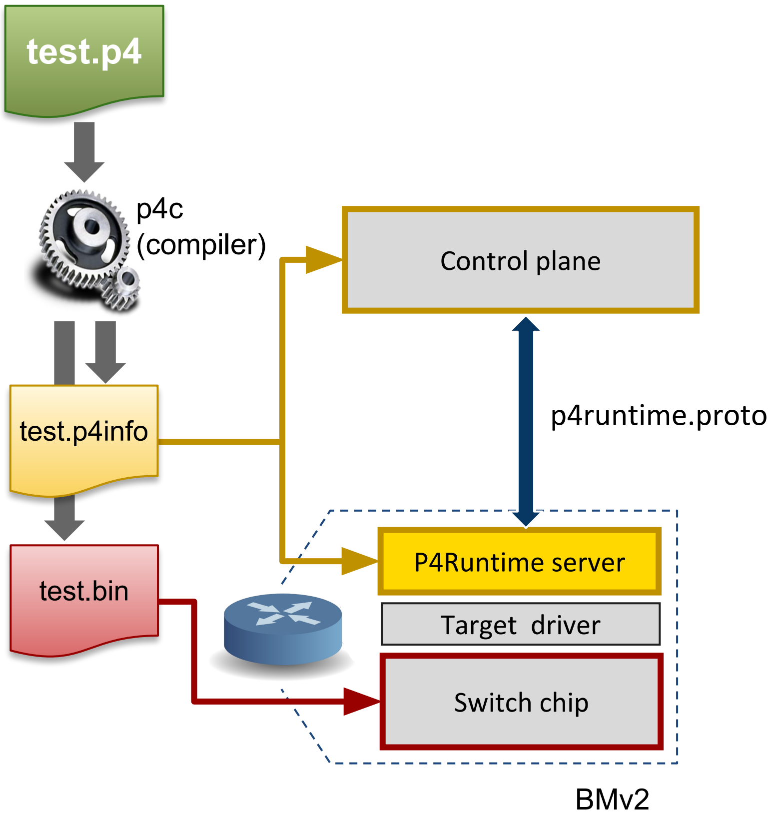
Work-in-progress by the p4.org API WG
Expect API changes in the future

P4Runtime workflow

P4 compiler generates 2 files:

1. Target-specific binaries
   - Used to configure switch pipeline
     (e.g. binary config for ASIC, bitstream for FPGA, etc.)
2. P4Info file
   - Captures P4 program attributes needed to runtime control
     - 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/PI/blob/master/proto/p4/config/p4info.proto



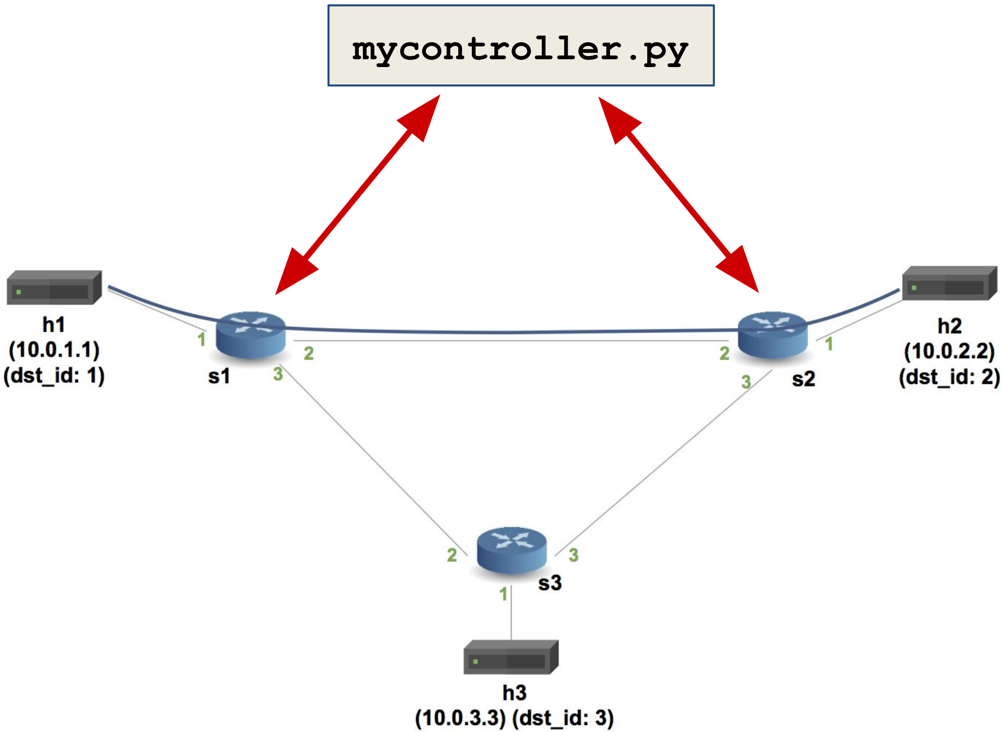
Slide courtesy P4.org

P4Runtime exercise

Exercise Overview

Controller's responsibilities:

1. **Establish a gRPC connection to the switches for the P4Runtime service**
2. **Push the P4 program to each switch**
3. **Write the tunnel forwarding rules:**
   a. `myTunnel_ingress` rule to encapsulate packets on the ingress switch
   b. `myTunnel_forward` rule to forward packets on the ingress switch
   c. `myTunnel_egress` rule to decapsulate and forward packets on the egress switch
4. **Read the tunnel ingress and egress counters every 2 seconds**



Getting started

The source code has already been downloaded on your VM:
`~/tutorials/exercises/p4runtime`

You should start by reading the `README.md`

In this exercise, you will need to complete the implementation of `writeTunnelRules` **in** `mycontroller.py`

You will need two Terminal windows: one for your dataplane network (Mininet) that you will start using `make`**, and the other is for your controller program.**

To find the source code:
https://github.com/p4lang/tutorials/

README.md

Implementing a Control Plane using P4 Runtime

Introduction

In this exercise, we will be using P4 Runtime to send flow entries to the switch instead of using the switch's CLI. We will be building on the same P4 program that you used in the basic_tunnel exercise. The P4 program has been renamed to `advanced_tunnel.py` and has been augmented with two counters (`ingressTunnelCounter` , `egressTunnelCounter`) and two new actions (`myTunnel_ingress` , `myTunnel_egress`).

You will use the starter program, `mycontroller.py` , and a few helper libraries in the `p4runtime_lib` directory to create the table entries necessary to tunnel traffic between host 1 and 2.

> **Spoiler alert:** There is a reference solution in the `solution` sub-directory. Feel free to compare your implementation to the reference.

Step 1: Run the (incomplete) starter code

The starter code for this assignment is in a file called `mycontroller.py` , and it will install only some of the rules that you need to tunnel traffic between two hosts.

Let's first compile the new P4 program, start the network, use `mycontroller.py` to install a few rules, and look at the `ingressTunnelCounter` to see that things are working as expected.

1. In your shell, run:

```
make
```

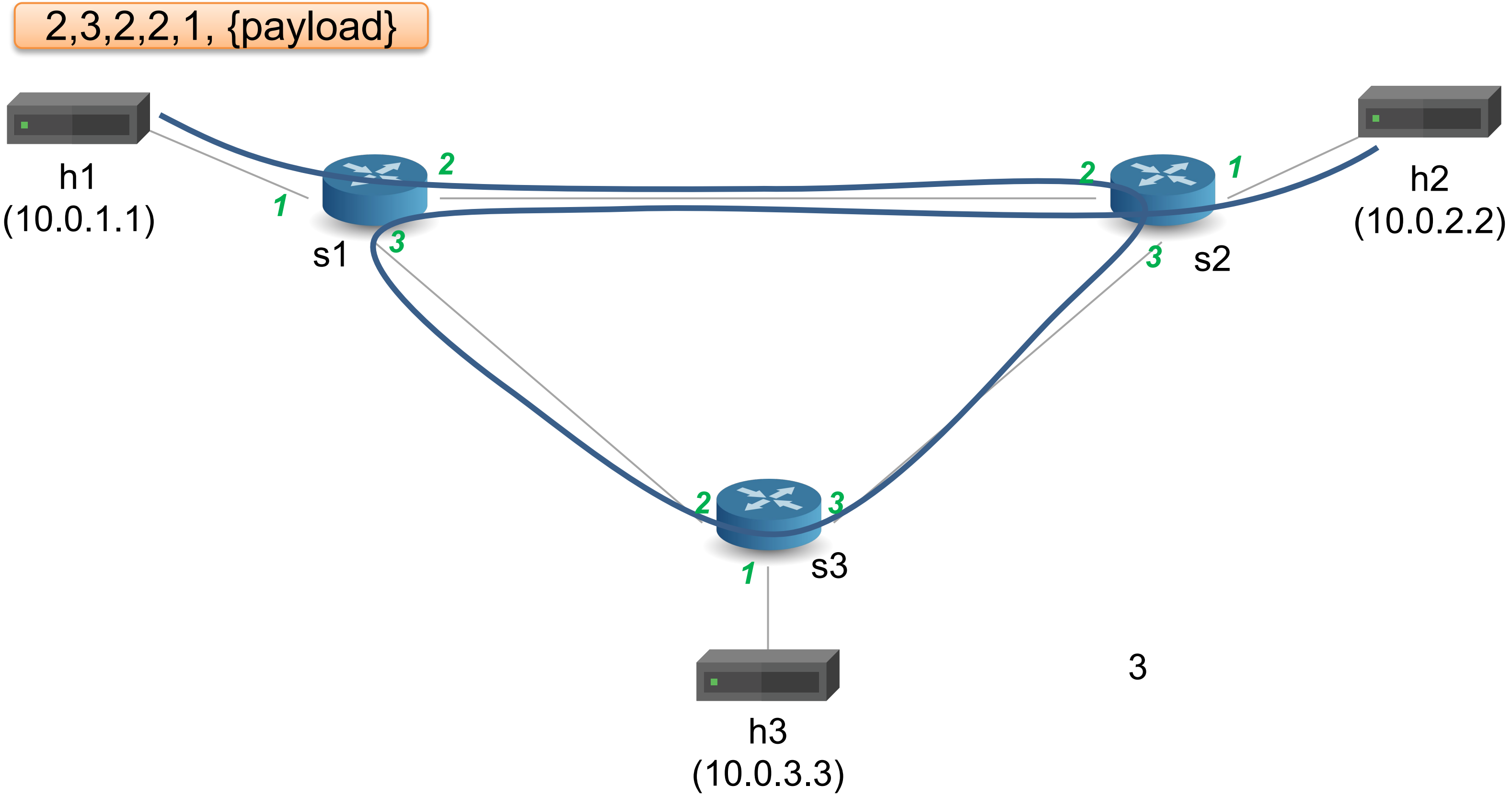
Lab 3: Advanced P4 Features

Plan

In this afternoon lab session, we'll explore some of P4's advanced features and see how to implement functionality commonly found in modern data planes.

- ***Source Routing:*** **implement simple form of source routing using P4's header stacks**
- ***Load Balancing:*** **implement ECMP-like functionality using P4's `extern` functions**
- ***Network Telemetry:*** **implement fine-grained telemetry directly in the data plane**

Source Routing



Source Routing: Packet Format

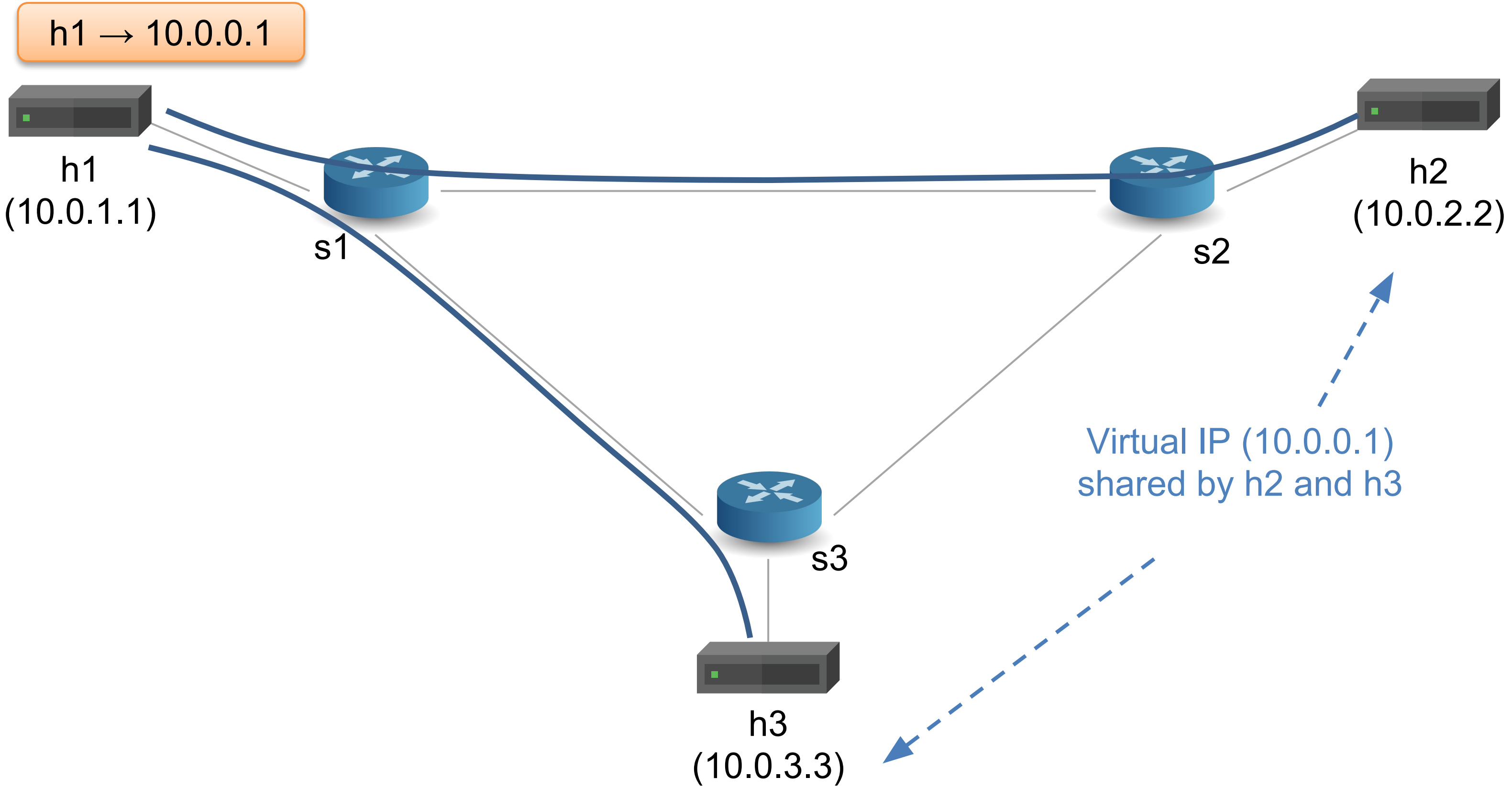
```
#define MAX_HOPS 9

const bit<16> TYPE_IPV4 = 0x800;
const bit<16> TYPE_SRCROUTING = 0x1234;
header srcRoute_t {
  bit<1>    bos;
  bit<15>   port;
}

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

- **Parse source routes only if etherType is 0x1234**
- **The special value bos == 1 indicates the "bottom of stack"**
- **Forward packets using source routes, and also decrement IPv4 TTL**
- **Drop the packet if source routes are not valid**
- **Hint: Use the next, pop_front primitives packet.extract(hdr.srcRoutes.next) hdr.srcRoutes.pop_front(1)**

Simple Load Balancing



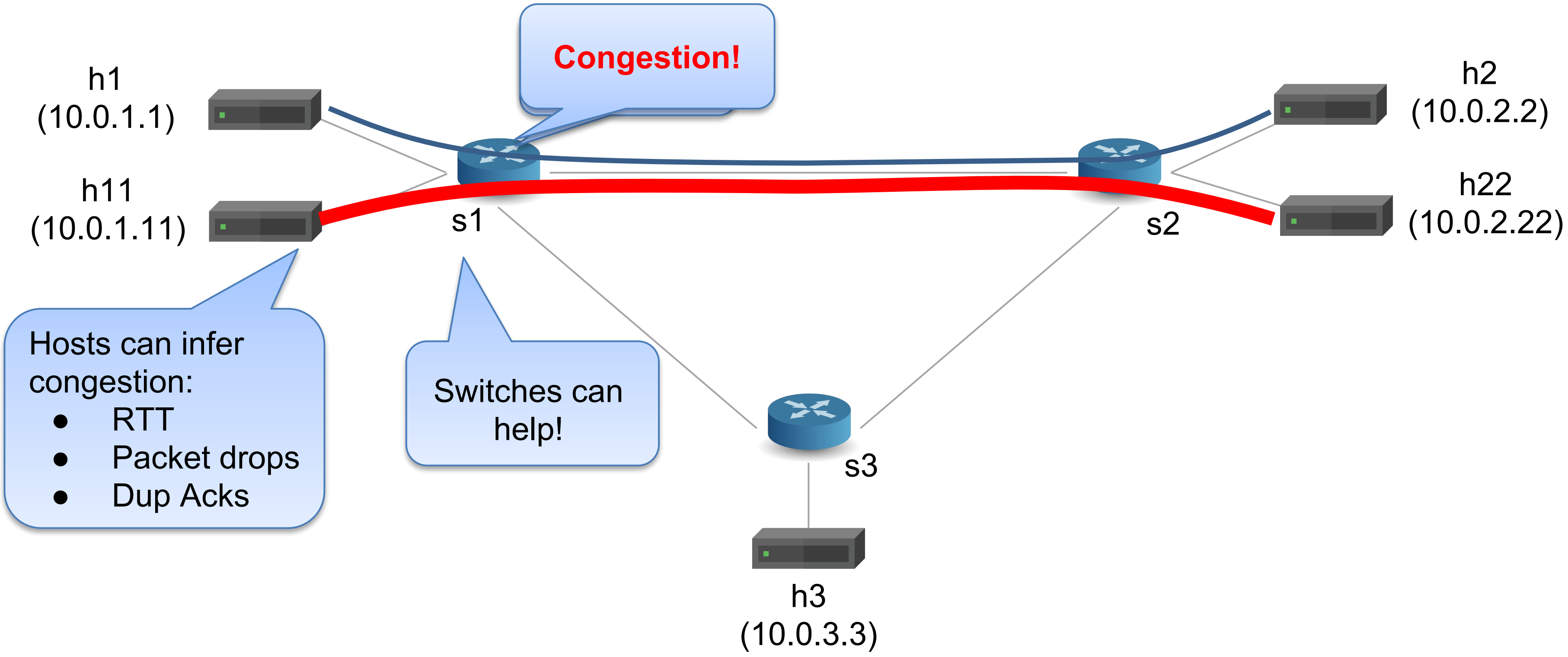
Hashing (V1Model)

```
enum HashAlgorithm {
  csum16,
  xor16,
  crc32,
  crc32_custom,
  crc16,
  crc16_custom,
  random,
  identity
}
extern void hash<O, T, D, M>(
  out O result,
  in HashAlgorithm algo,
  in T base,
  in D data,
  in M max);
```

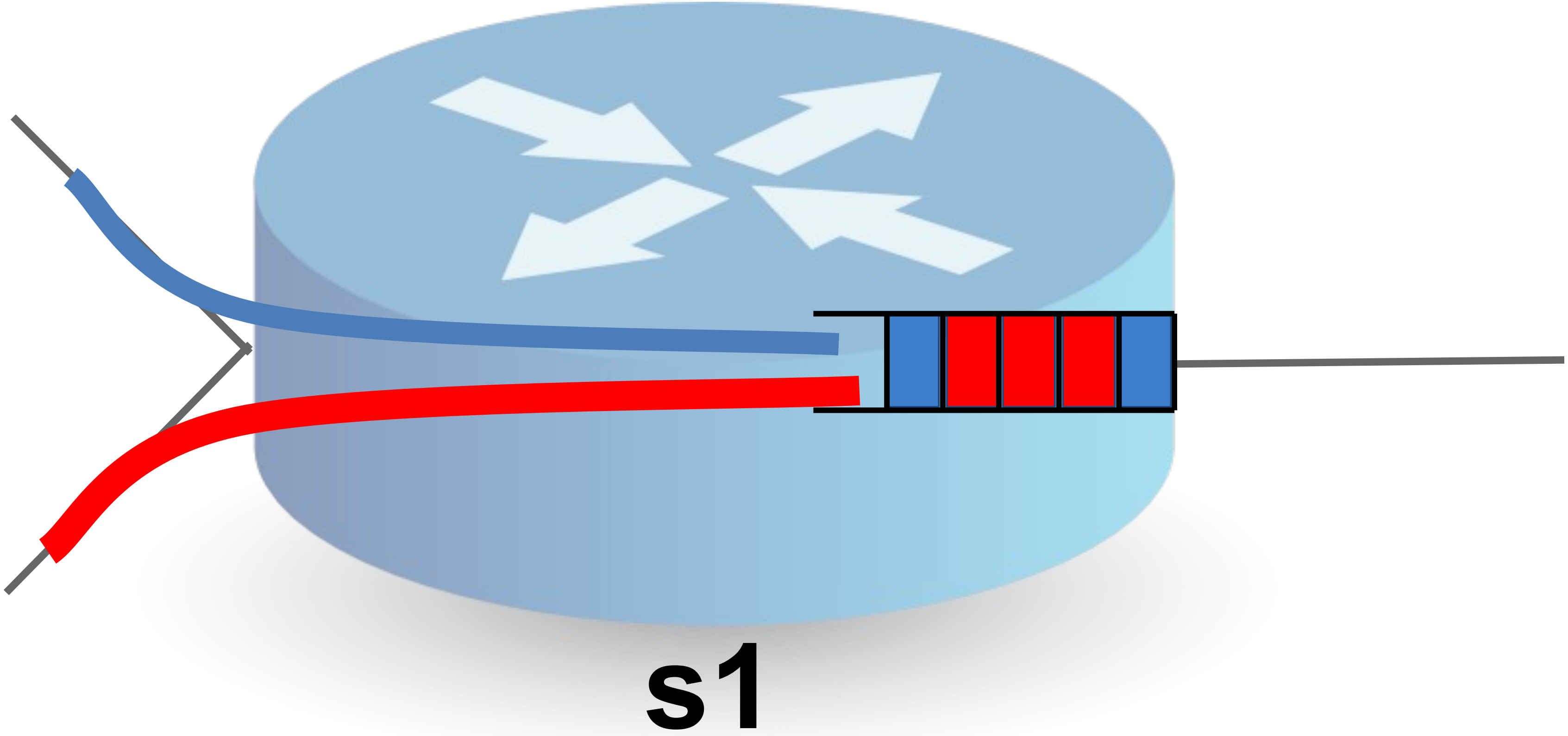
Computes the hash of `data` (using `algo`) modulo `max` and adds it to `base`

Uses type variables (like C++ templates / Java Generics) to allow hashing primitive to be used with many different types.

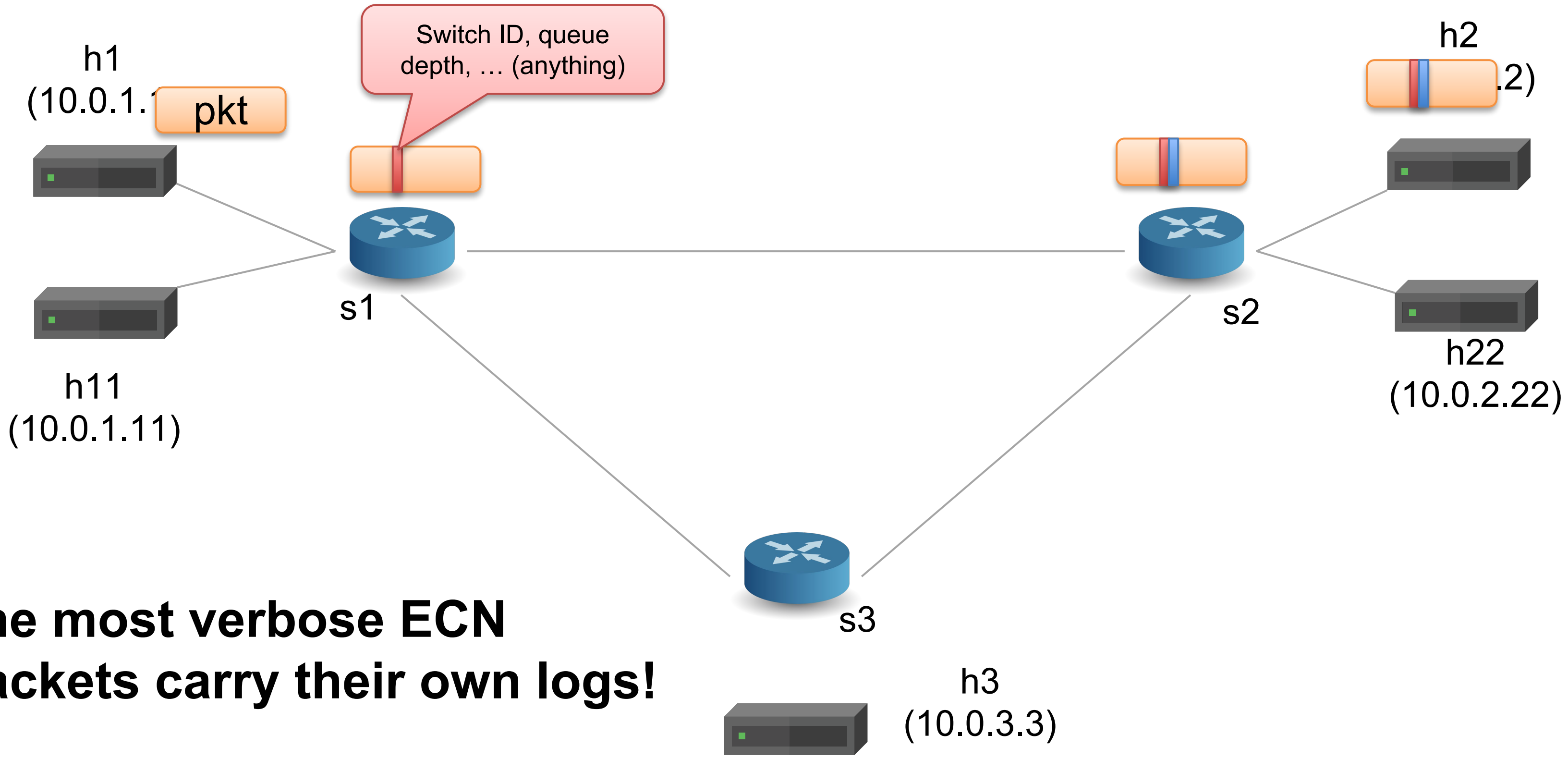
Monitoring & Debugging



Monitoring & Debugging



Multi-Route Inspection



The most verbose ECN

Packets carry their own logs!

Multi-Route Inspect: Packet Format

```
header mri_t {
  bit<16>  count;
}
```

```
header switch_t {
  switchID_t  swid;
  qdepth_t    qdepth;
}
```

```
struct headers {
  ethernet_t           ethernet;
  ipv4_t               ipv4;
  ipv4_option_t        ipv4_option;
  mri_t                mri;
  switch_t[MAX_HOPS]   swtraces;
}
```

- **Header validity opertions:**
  - `hdr.setValid()`: add_header
  - `hdr.setInvalid()`: remove_header
  - `hdr.isValid()`:test validity

- **Header Stacks**
  - `hdr[CNT] stk;`
- **Header Stacks in Parsers**
  - `stk.next`
  - `stk.last`
  - `stk.lastIndex`
- **Header Stacks in Controls**
  - `stk[i]`
  - `stk.size`
  - `stk.push_front(int count)`
  - `stk.pop_front(int count)`

Header verification

```
/* Standard errors, defined in core.p4 */
error {
   NoError,              // no error
   PacketTooShort,       // not enough bits in packet for extract
   NoMatch,              // match expression has no matches
   StackOutOfBounds,     // reference to invalid element of a header stack
   OverwritingHeader,    // one header is extracted twice
   HeaderTooShort,       // extracting too many bits in a varbit field
   ParserTimeout         // parser execution time limit exceeded
}
/* Additional error added by the programmer */
error { IPv4BadHeader }
...
state parse_ipv4 {
  packet.extract(hdr.ipv4);
  verify(hdr.ipv4.version == 4, error.IPv4BadHeader);
  transition accept;
}
```

Wrapping up & Next Steps

Why $P4_{16}$?

- **Clearly defined semantics**
  - You can describe what your data plane program is doing
- **Expressive**
  - Supports a wide range of architectures through standard methodology
- **High-level, Target-independent**
  - Uses conventional constructs
  - Compiler manages the resources and deals with the hardware
- **Type-safe**
  - Enforces good software design practices and eliminates “stupid” bugs
- **Agility**
  - High-speed networking devices become as flexible as any software
- **Insight**
  - Freely mixing packet headers and intermediate results

Things we covered

- **The P4 ”world view”**
  - Protocol-Independent Packet Processing
  - Language/Architecture Separation
  - If you can interface with it, it can be used
- **Key data types**
- **Constructs for packet parsing**
  - State machine-style programming
- **Constructs for packet processing**
  - Actions, tables and controls
- **Packet deparsing**
- **Architectures & Programs**

Things we didn't cover

- **Mechanisms for modularity**
  - Instantiating and invoking parsers or controls
- **Details of variable-length field processing**
  - Parsing and deparsing of options and TLVs
- **Architecture definition constructs**
  - How these "templated" definitions are created
- **Advanced features**
  - How to do learning, multicast, cloning, resubmitting
  - Header unions
- **Control plane interface**

The P4 Language Consortium

- **Consortium of academic and industry members**
- **Open source, evolving, domain-specific language**
- **Permissive Apache license, code on GitHub today**
- **Membership is free: contributions are welcome**
- **Independent, set up as a California nonprofit**