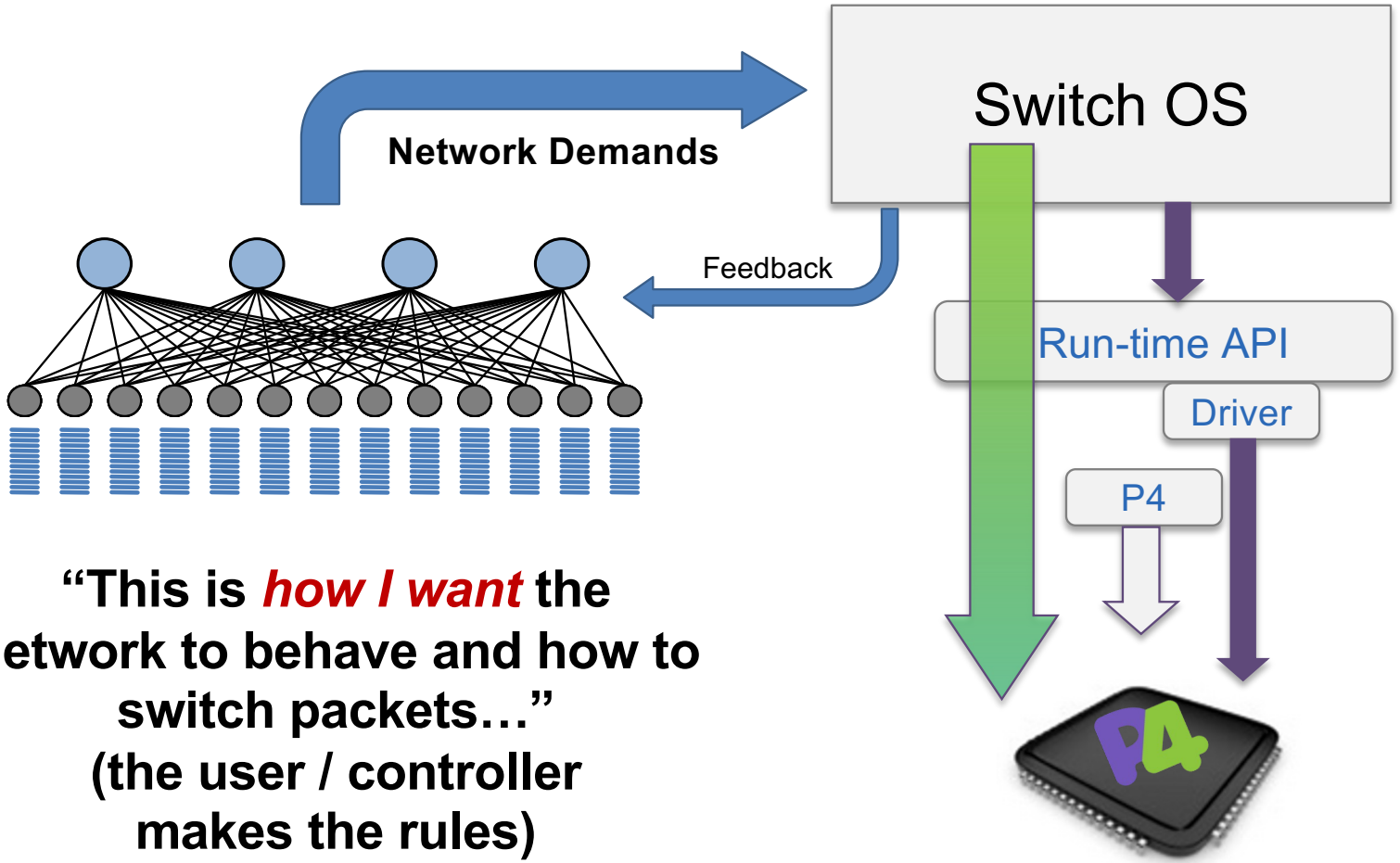
Introduction to Data Plane Programming

Advanced P4 Topics

Recap on P4

P4: Top-down design

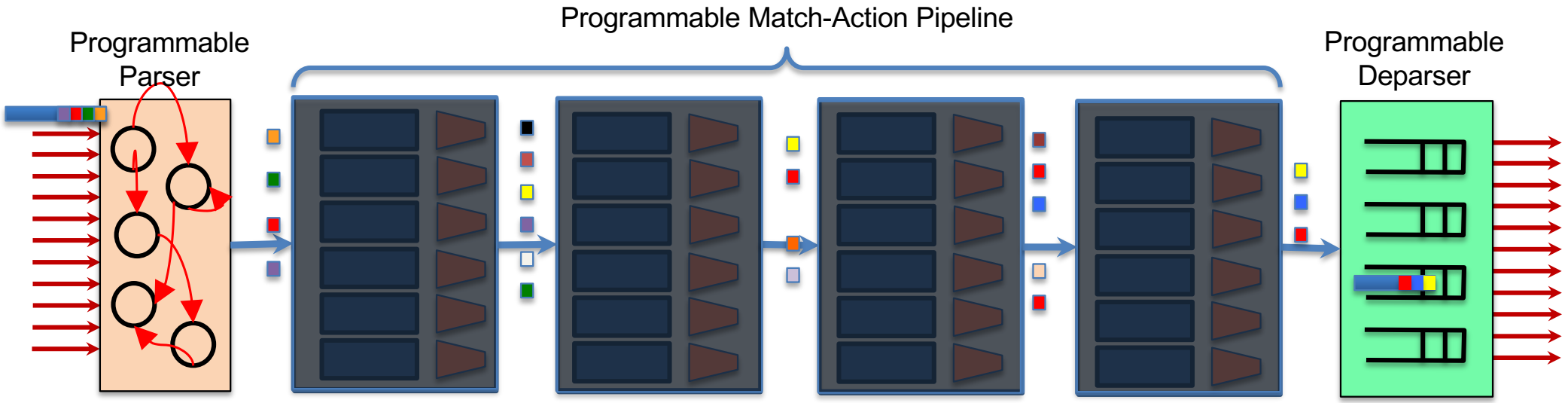


"This is *how I want* the network to behave and how to switch packets…"
(the user / controller makes the rules)

P4 Programmable Device

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

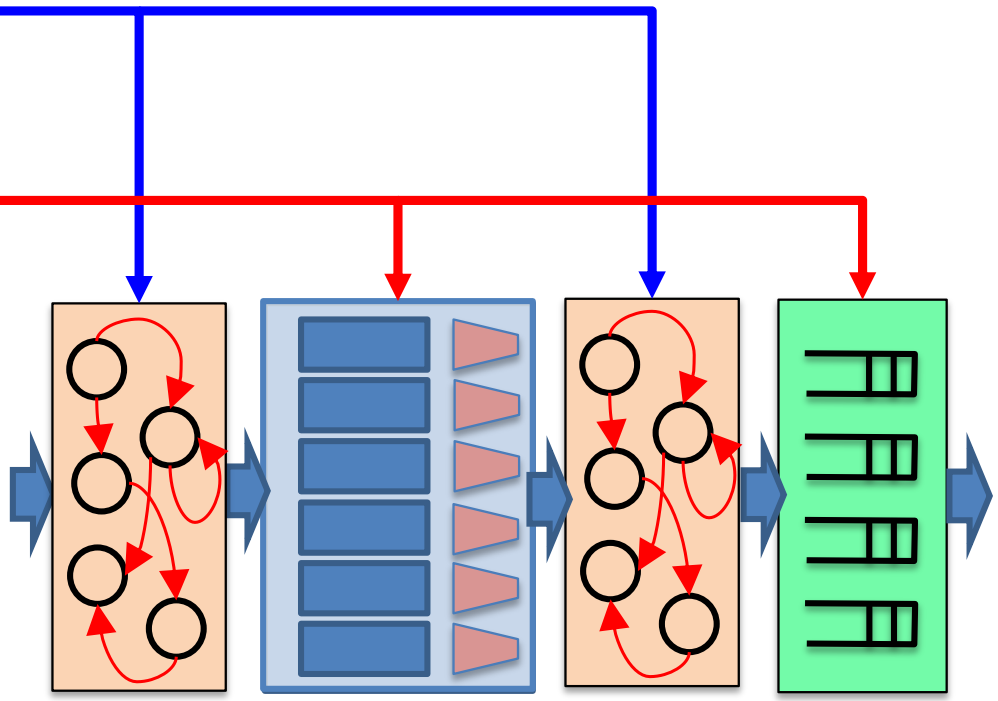
Controls — Tables, Actions, control flow statements

Expressions — Basic operations and operators

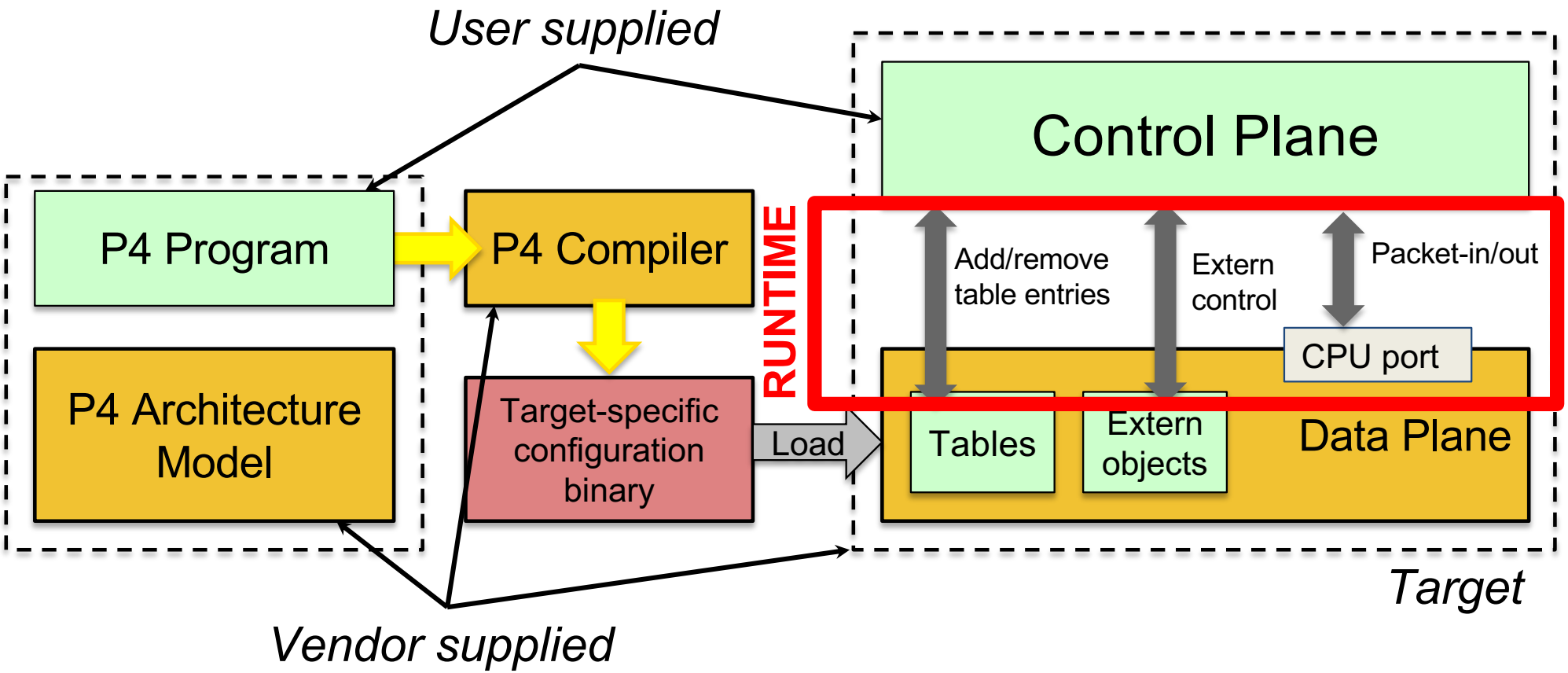
Data Types — Bistrings, headers, structures, arrays

Architecture Description — Programmable blocks and their interfaces

Extern Libraries — Support for specialized components



Programming a P4 Target

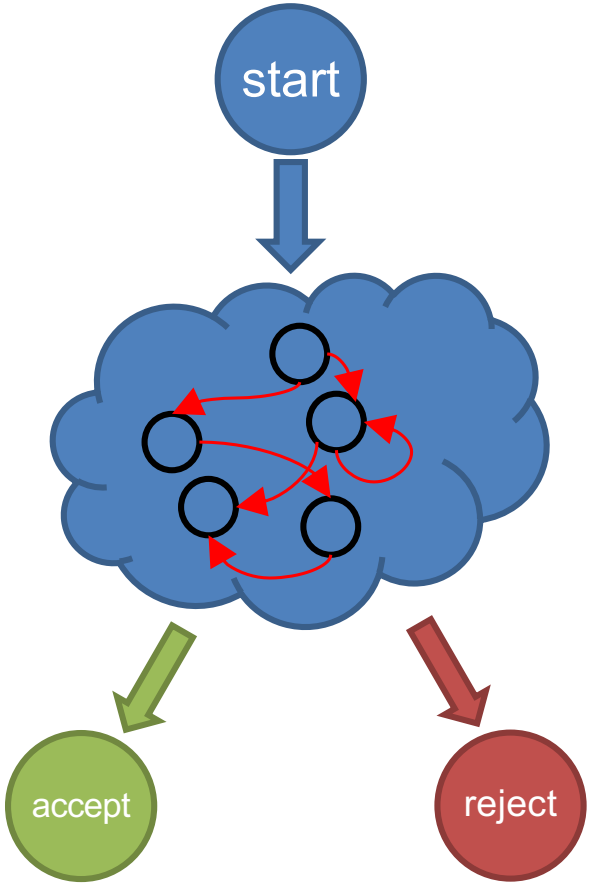


Existing approaches to runtime control

- **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

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)



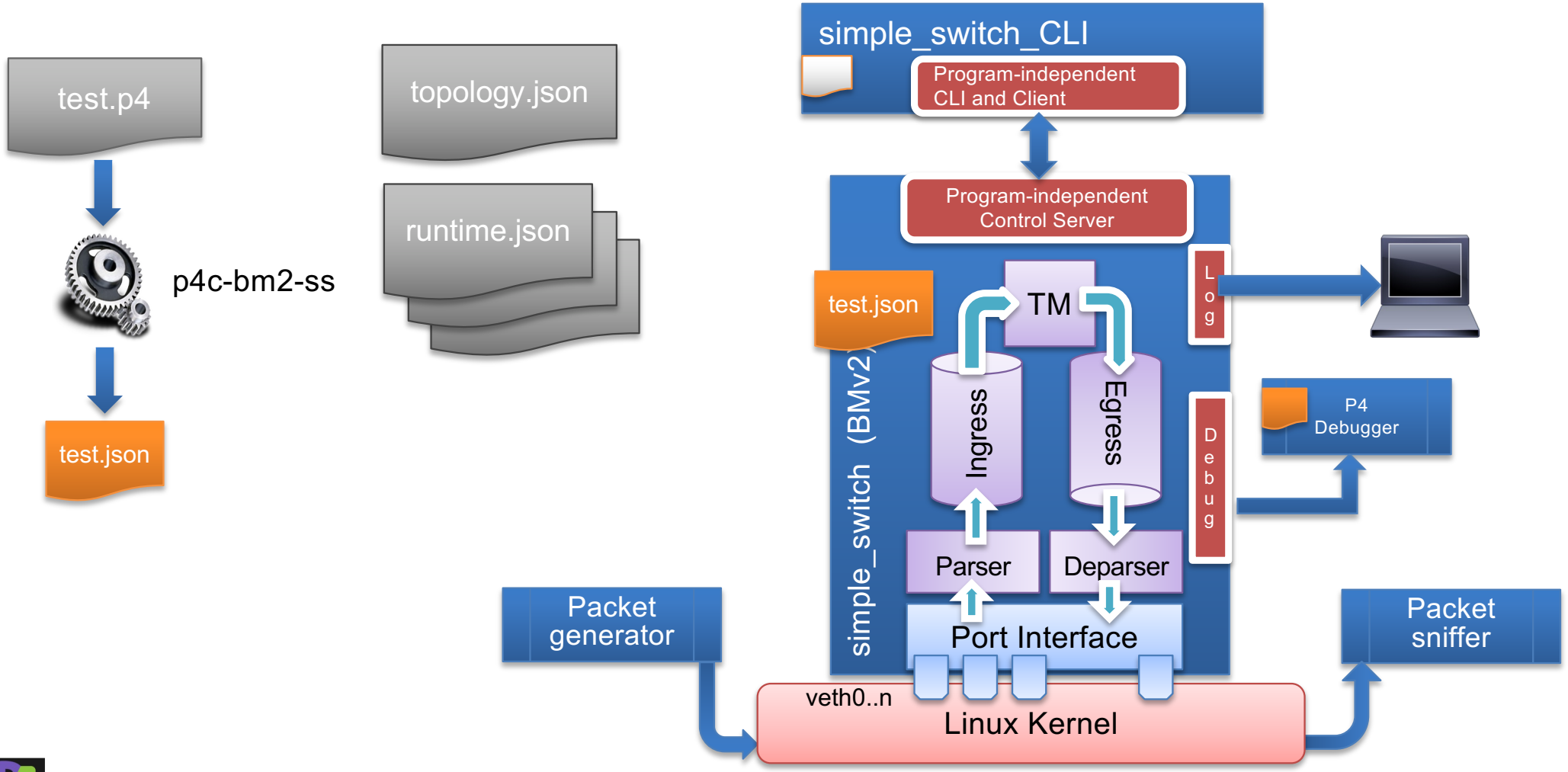
$P4_{16}$ Controls

- **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)**

$P4_{16}$ 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)

`Makefile`: under the hood



`Makefile`: under the hood (in pseudocode)

```
P4C_ARGS = --p4runtime-file $(basename $@).p4info
           --p4runtime-format text
RUN_SCRIPT = ../../utils/run_exercise.py
TOPO = topology.json
```

dirs:

```
    mkdir -p build pcaps logs
```

build: for each P4 program, generate BMv2 json file

```
    p4c-bm2-ss --p4v 16 $(P4C_ARGS) -o $@ $<
```

run: build, then ***[default target]***

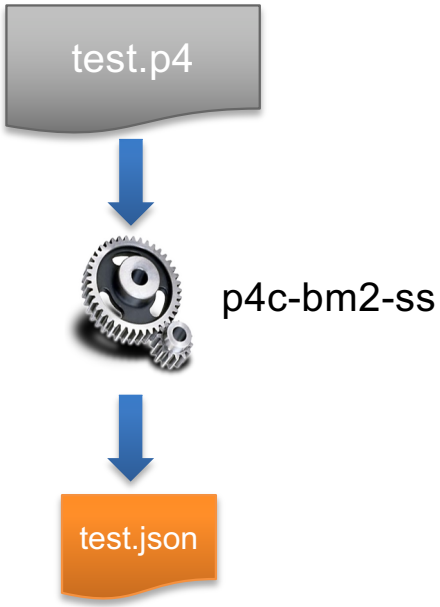
```
    sudo python $(RUN_SCRIPT) -t $(TOPO)
```

stop: `sudo mn -c`

clean: stop, then

```
    rm -f *.pcap
    rm -rf build pcaps logs
```

Step 1: P4 Program compilation [build phase]




```
$ p4c-bm2-ss -o test.json test.p4
```

alternatively, can also create a P4info message, which is a protobuf message which describes the data-model to be used by the control plane when generating P4 runtime requests

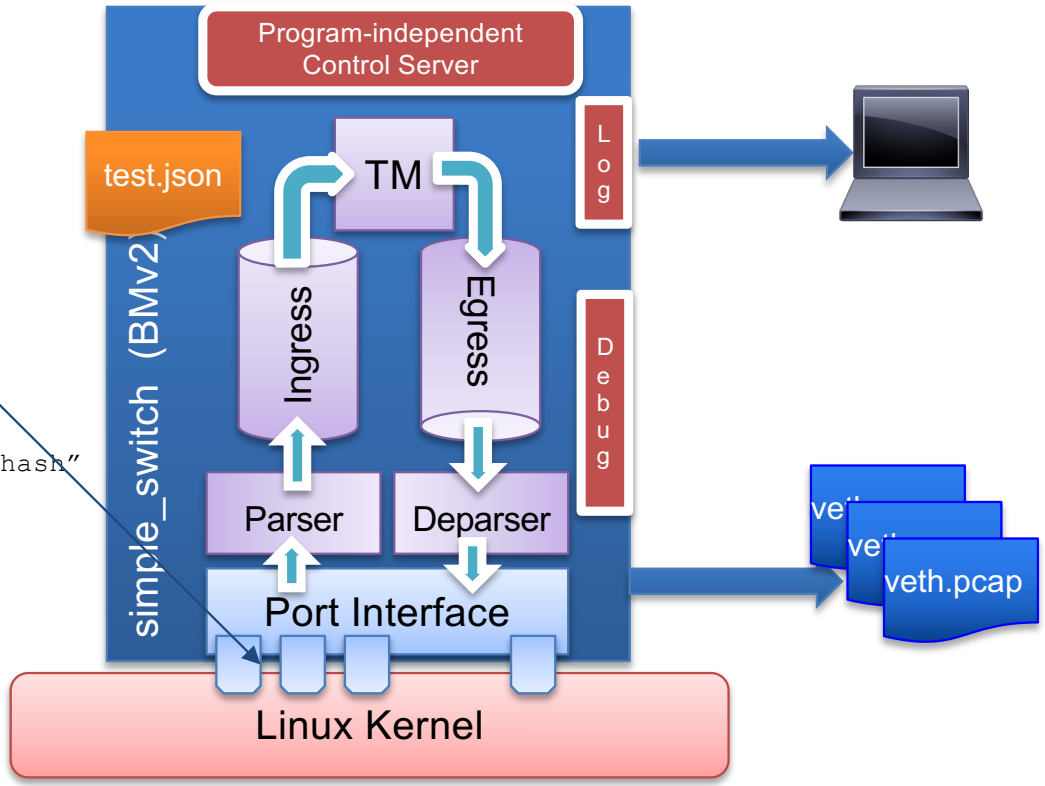
test.json is a JSON description of the forwarding pipeline as compiled from test.p4, which is required by the bmv2 simple_switch packet-processing binary

Step 2: Starting the model

```
$ sudo simple_switch --log-console --dump-packet-data 64 \
-i 0@veth0 -i 1@veth2 ... [--pcap]
test.json
```

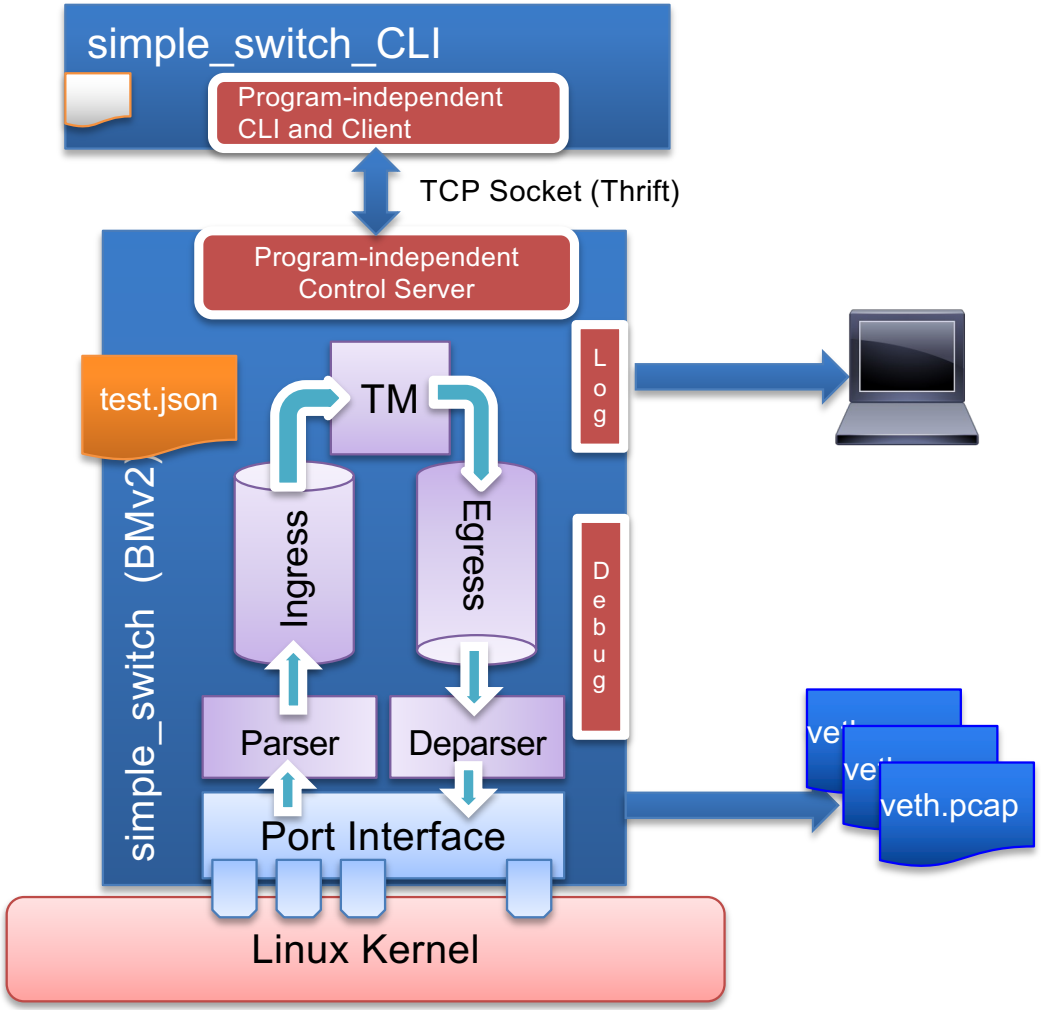
prepare veth interfaces first

```
# 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
```



Step 3: Starting the CLI

```
$ simple_switch_CLI
```



Step 3: Interacting with the Control Plane

- **P4 Program defined packet processing pipeline**
  - Rules within a table are entered by control plane at runtime → P4Runtime
  - When a rule matches a packet, its action is invoked with parameters supplied by the control plane as part of the rule.
- **For exercises - old**
  - When booting up Mininet instance, `make run` will install packet-processing rules in the tables of each switch (simple_switch_CLI).
  - These are defined in the `sX-commands.txt` files, where `X` corresponds to the switch number.
- **P4Runtime used to install control plane rules - new**
  - The content of files `sX-runtime.json` refer to specific names of tables, keys, and actions, as defined in the P4Info file `build/basic.p4info` after executing `make run`)

Step 4: Interacting with Switch using simple_switch_CLI

```
within  simple_switch_CLI
RuntimeCmd: show_tables
m_filter
m_table

RuntimeCmd: table_info m_table
m_table

RuntimeCmd: table_dump m_table
m_table:

RuntimeCmd: table_add m_table m_action 01:00:00:00:00:00&&&01:00:00:00:00:00 => 1 0
Adding entry to ternary match table m_table
[meta.meter_tag(exact, 32)] [ethernet.srcAddr(ternary, 48)]
[ethernet.srcAddr(ternary, 48)]
match key:
action:
runtime data:
SUCCESS
entry has been added with handle 1

RuntimeCmd: table_delete m_table 1

TERNARY-01:00:00:00:00:00 &&& 01:00:00:00:00:00 m_action
00:00:00:05
```

Value and mask for ternary matching. No spaces around "&&&"

entry priority

key => separates the key from the action data

All subsequent operations use the entry handle

Step 5: Run the traffic generator and sniffer

In some exercises, this is `send.py` **and** `receive.py`

In others, we use standard Linux programs, like `ping`

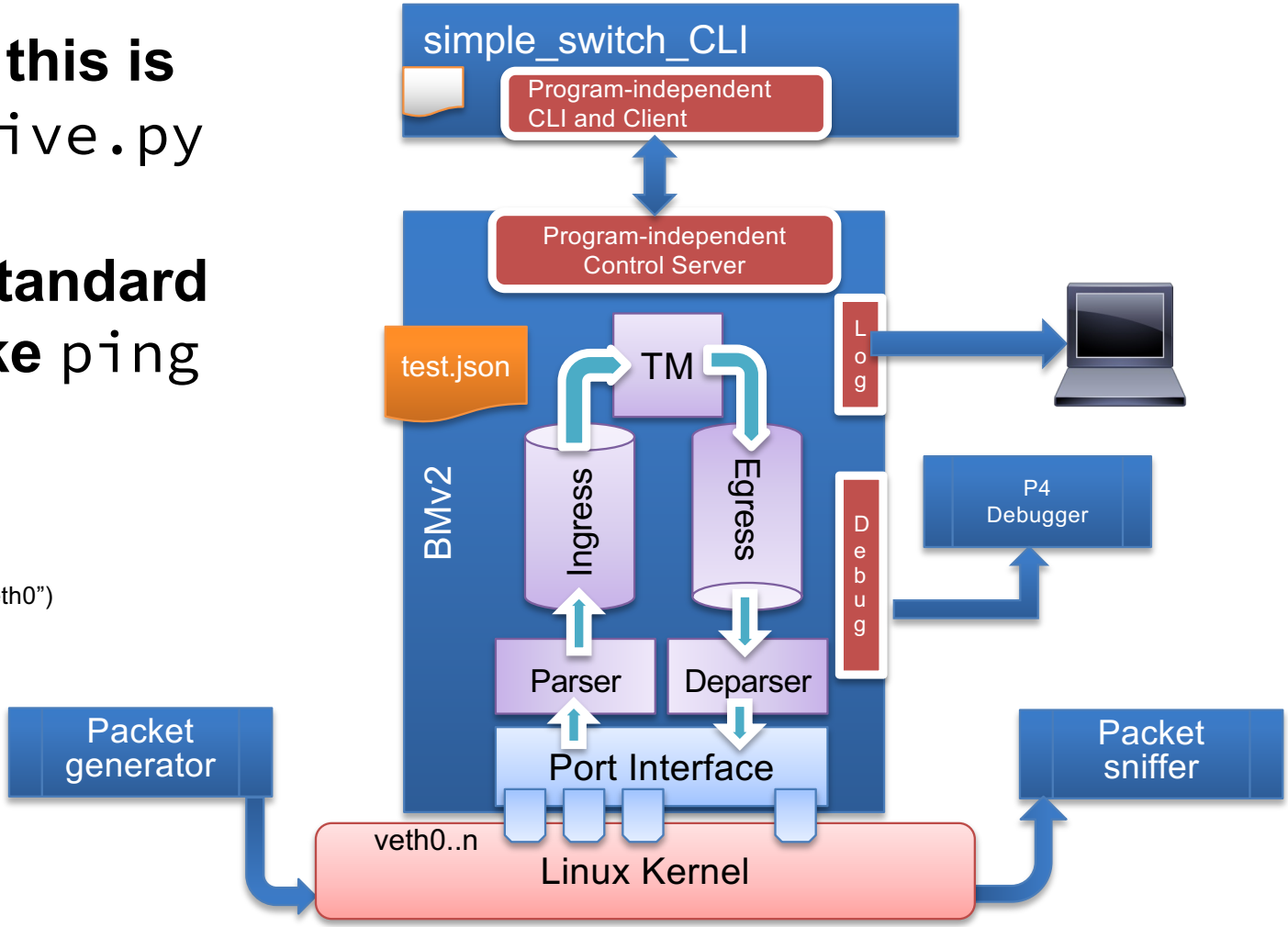
Can also use

scapy for sending

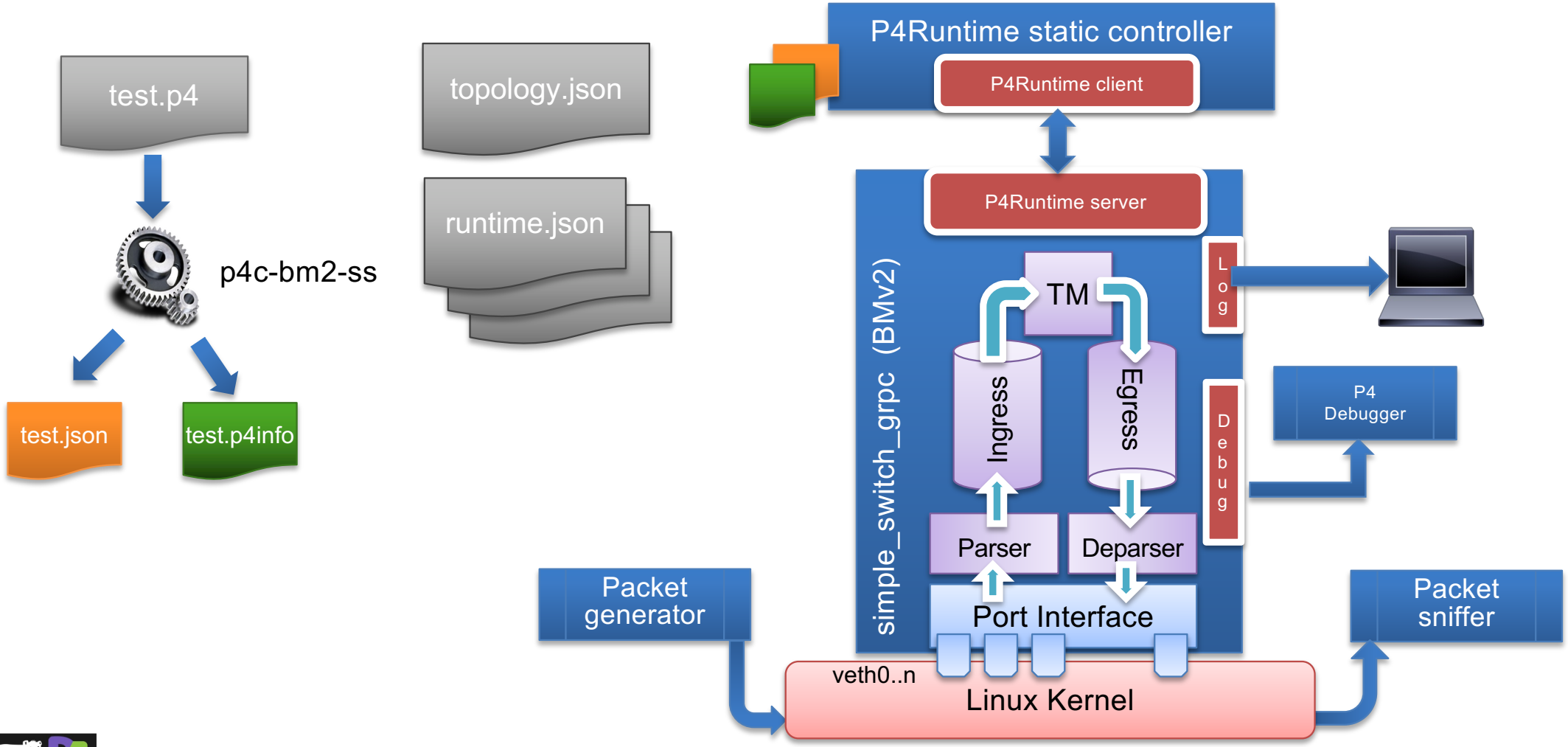
p = Ethernet()/IP()/UDP()/"Payload" sendp(p, iface="veth0")

scapy for sniffing

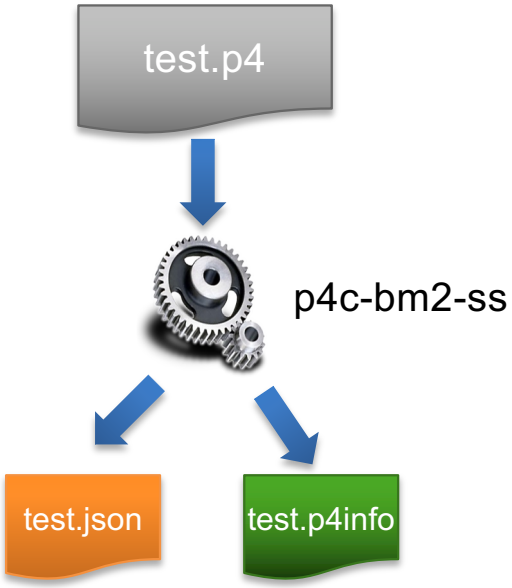
sniff(iface="veth9", prn=lambda x: x.show())



Alternative: P4 Runtime



Step 1: P4 Program compilation [build phase]



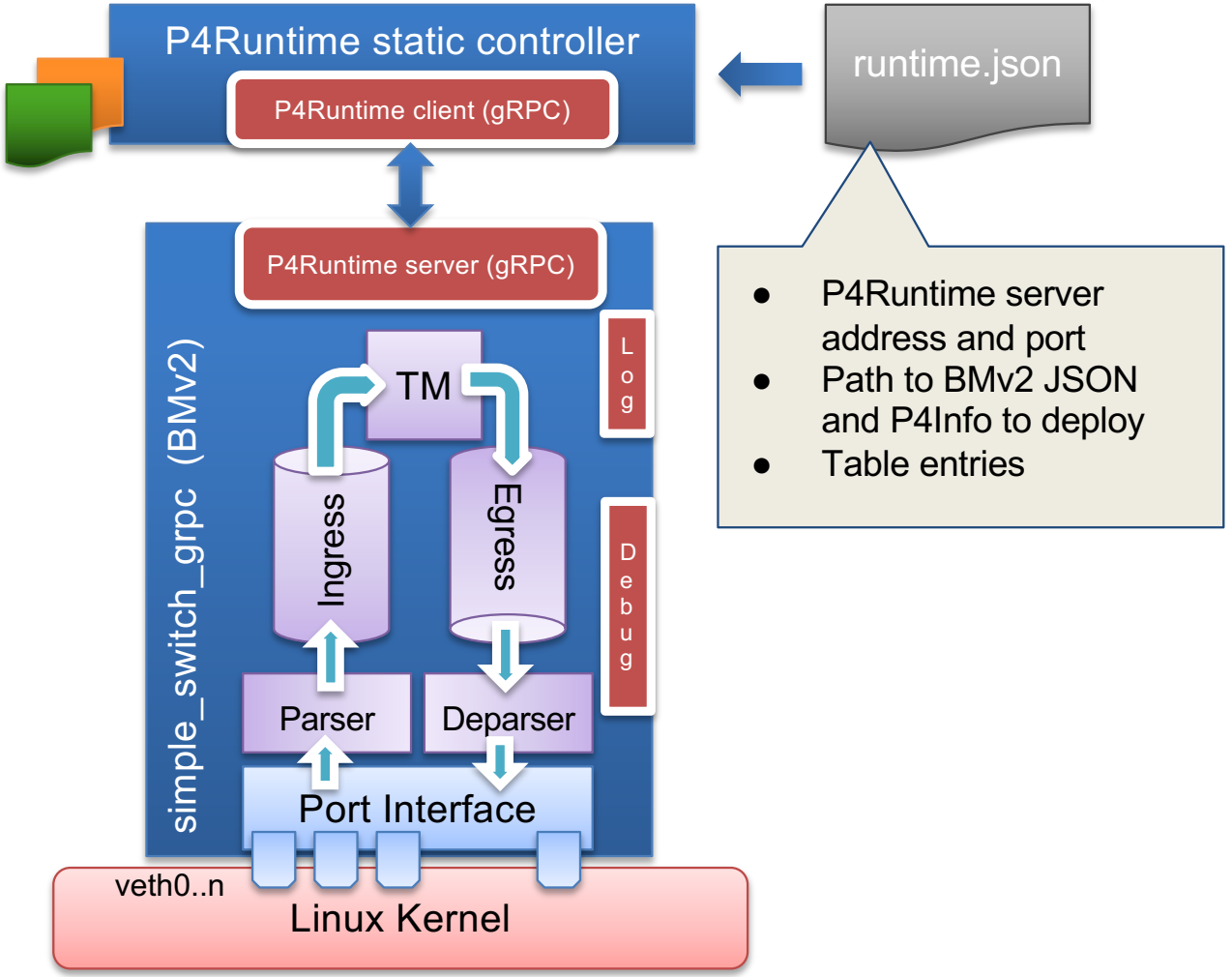

```
$ p4c-bm2-ss --p4v 16 \
    -o test.json \
    --p4runtime-file test.p4info \
    --p4runtime-format text \
    test.p4
```

Alternative,using runtime server

Step 2: `run_exercise.py` [run phase]

a. **Create network based on topology.json**
b. **Start `simple_switch_grpc` instance for each switch**
c. **Use P4Runtime to push the P4 program (P4Info and BMv2 JSON)**
d. **Add the static rules defined in `runtime.json`**

Alternative, using runtime server

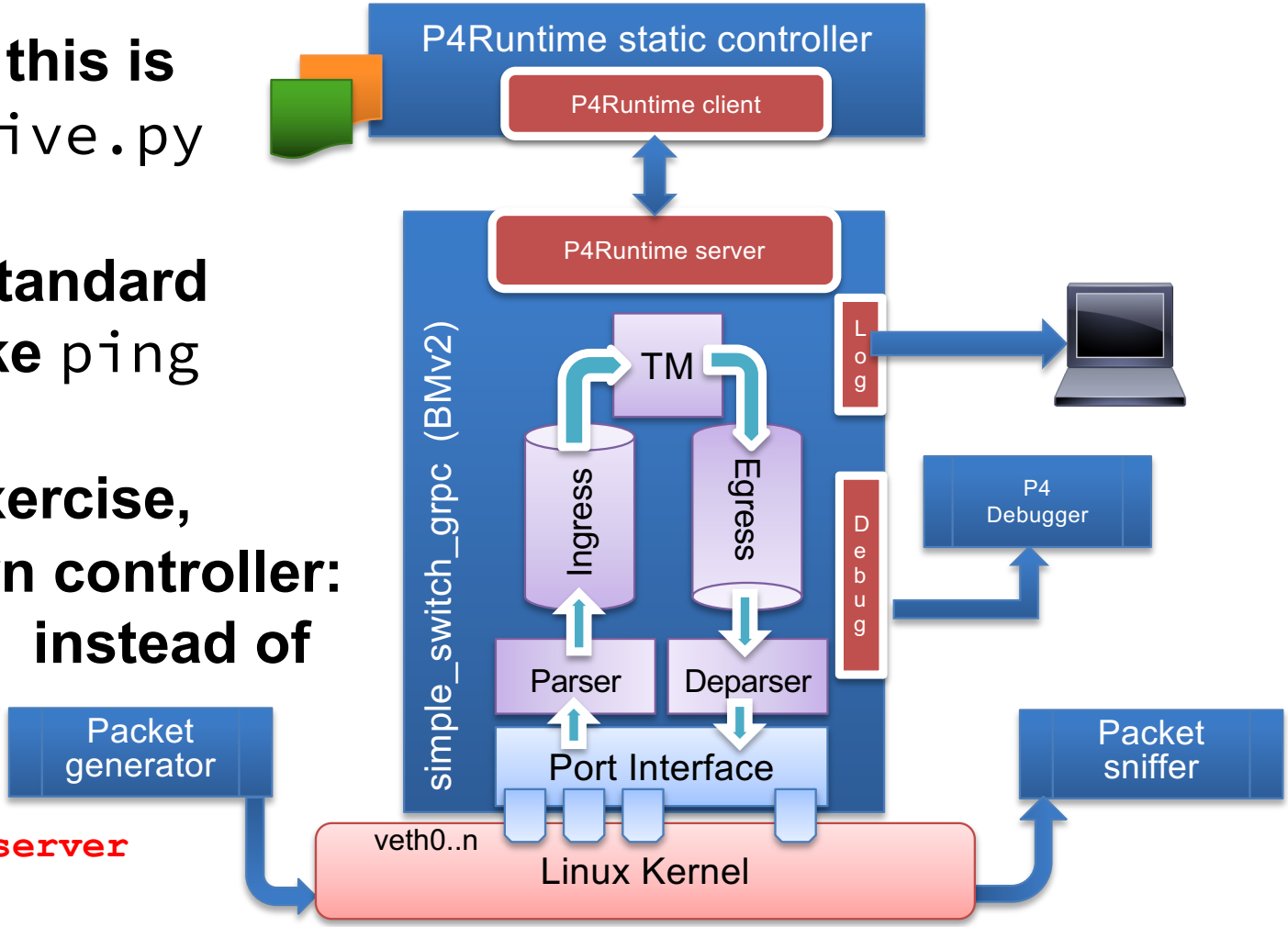


Step 3: Run the traffic generator and sniffer

In some exercises, this is `send.py` **and** `receive.py`

In others, we use standard Linux programs, like `ping`

In the p4runtime exercise, we also run our own controller: `mycontroller.py` **instead of the static one**



`Alternative,using runtime server`

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), need to do this via control plane
  - alternatively, can use registers
- **What happens upon reaching the `reject` state of the parser?**
  - Architecture dependent
- **How much of the packet can I parse?**
  - Architecture dependent

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**

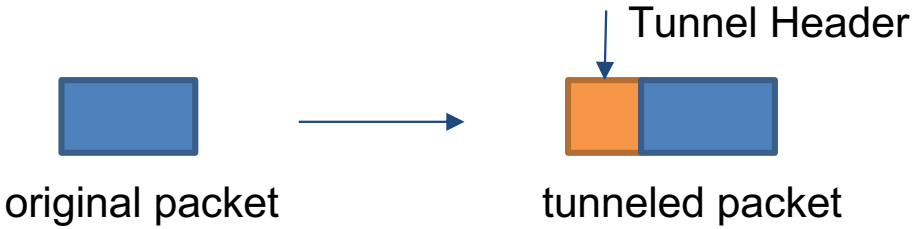
```
[15:16:48.145] [bmv2] [D]
[thread 4090] [96.0] [cxt 0]
Looking up key:
* std_meta.egress_spec : 2
```

Exercise 2: Tunneling

basic_tunnel

Basic Tunneling

- **Tunneling main feature for**
  - Data Center networks
  - Mobile Core Networks (e.g. Evolved Packet Core - EPC)
  - Network Virtualization (e.g. VXLAN, GRE, ...)
  - Mobility Management (e.g. Mobile IP)
  - Overlay Routing
  - ...
- **How can we implement tunneling?**
  - encapsulate a packet into another one by prepending a new header



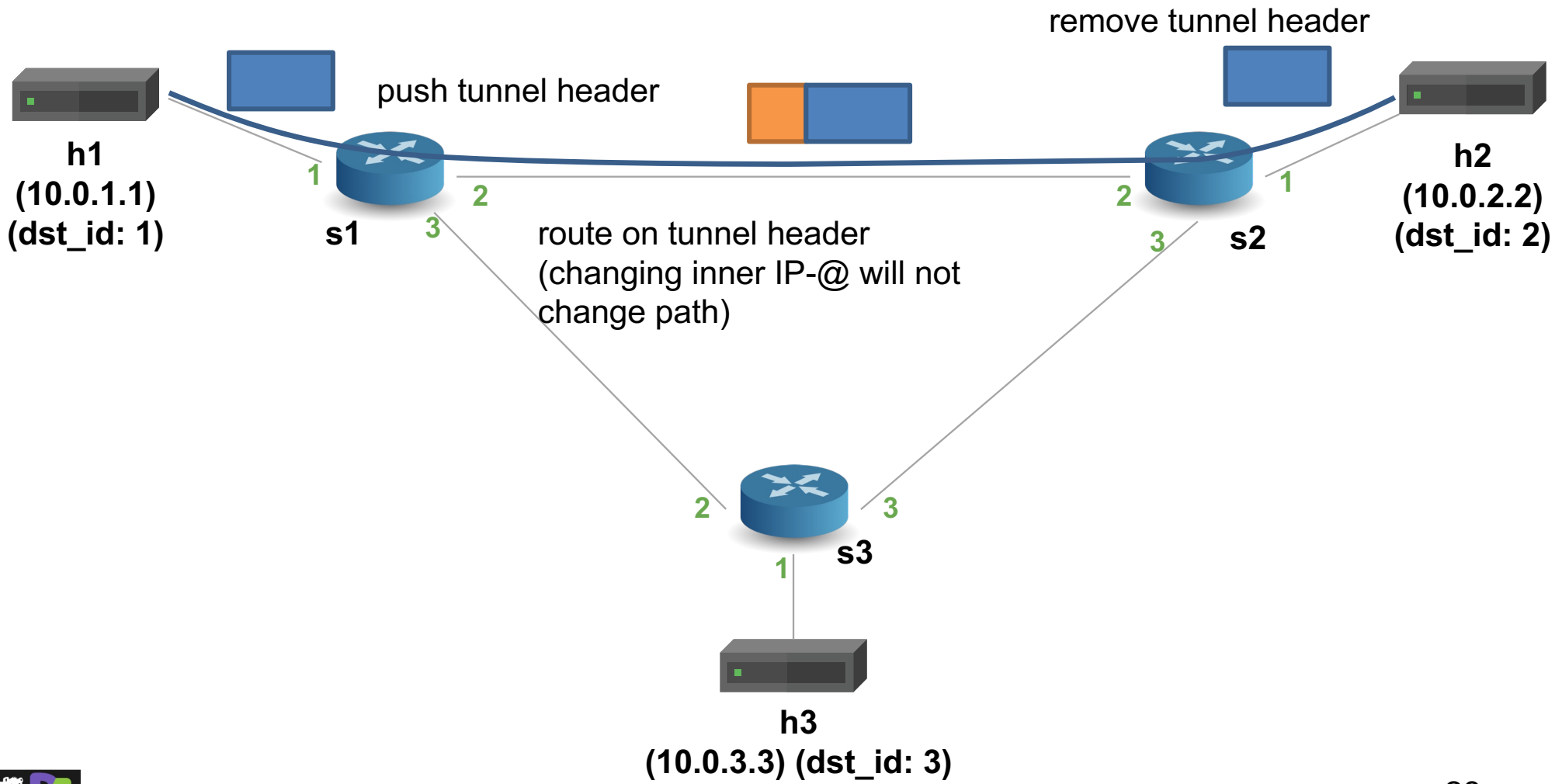
Basic Tunneling

- ToDo: Add support for basic tunneling to the basic IP router in P4

- 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 do routing using the `myTunnel` header

Basic Forwarding: Topology



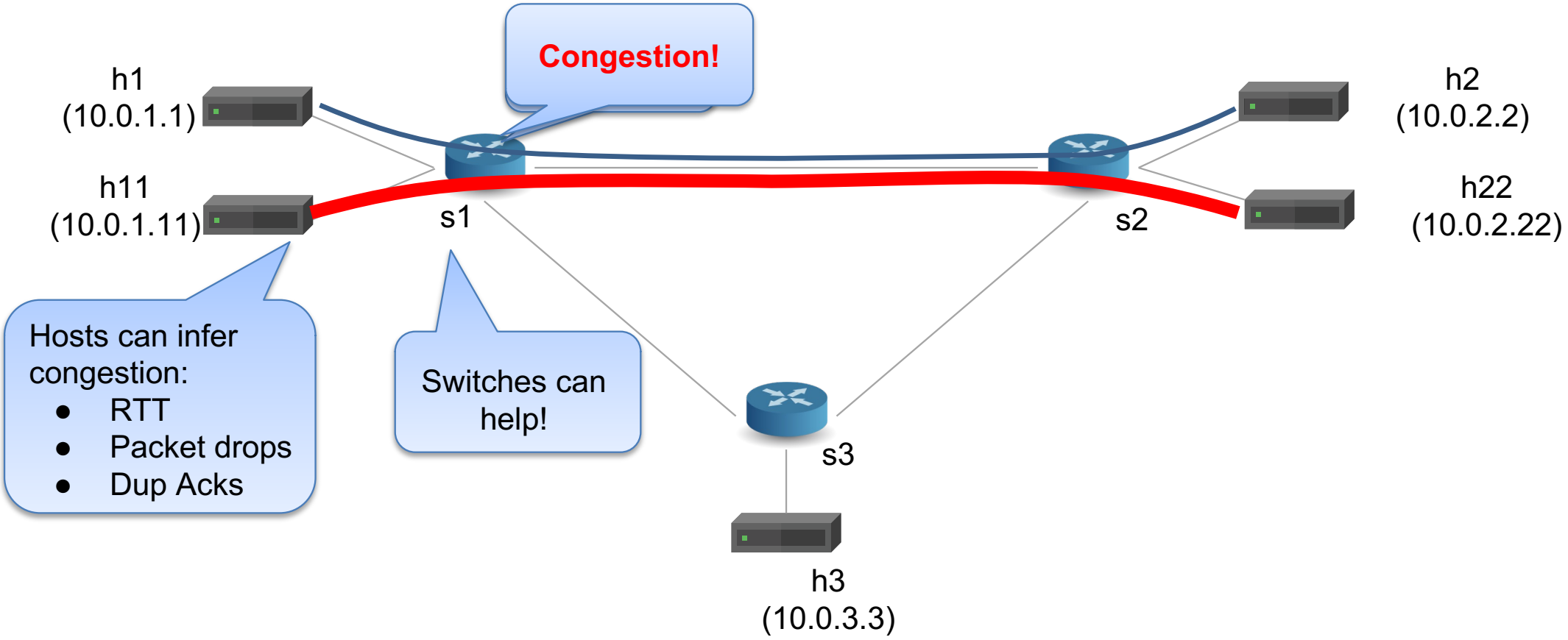
Basic Tunneling TODO List

- **Define `myTunnel_t` header type and add to `headers` struct**
- **Update parser based on ethertype (0x1212: tunnel)**
- **Define `myTunnel_forward` action**
- **Define `myTunnel_exact` table**
- **Update table application logic in `MyIngress` `apply` statement**
- **Update deparser**
- **Adding forwarding rules**
  - myTunnel_ingress rule to encapsulate packets on the ingress switch
  - myTunnel_forward rule to forward packets on the ingress switch
  - myTunnel_egress rule to decapsulate and forward packets on the egress switch
- **Read the tunnel ingress and egress counters every 2 seconds**

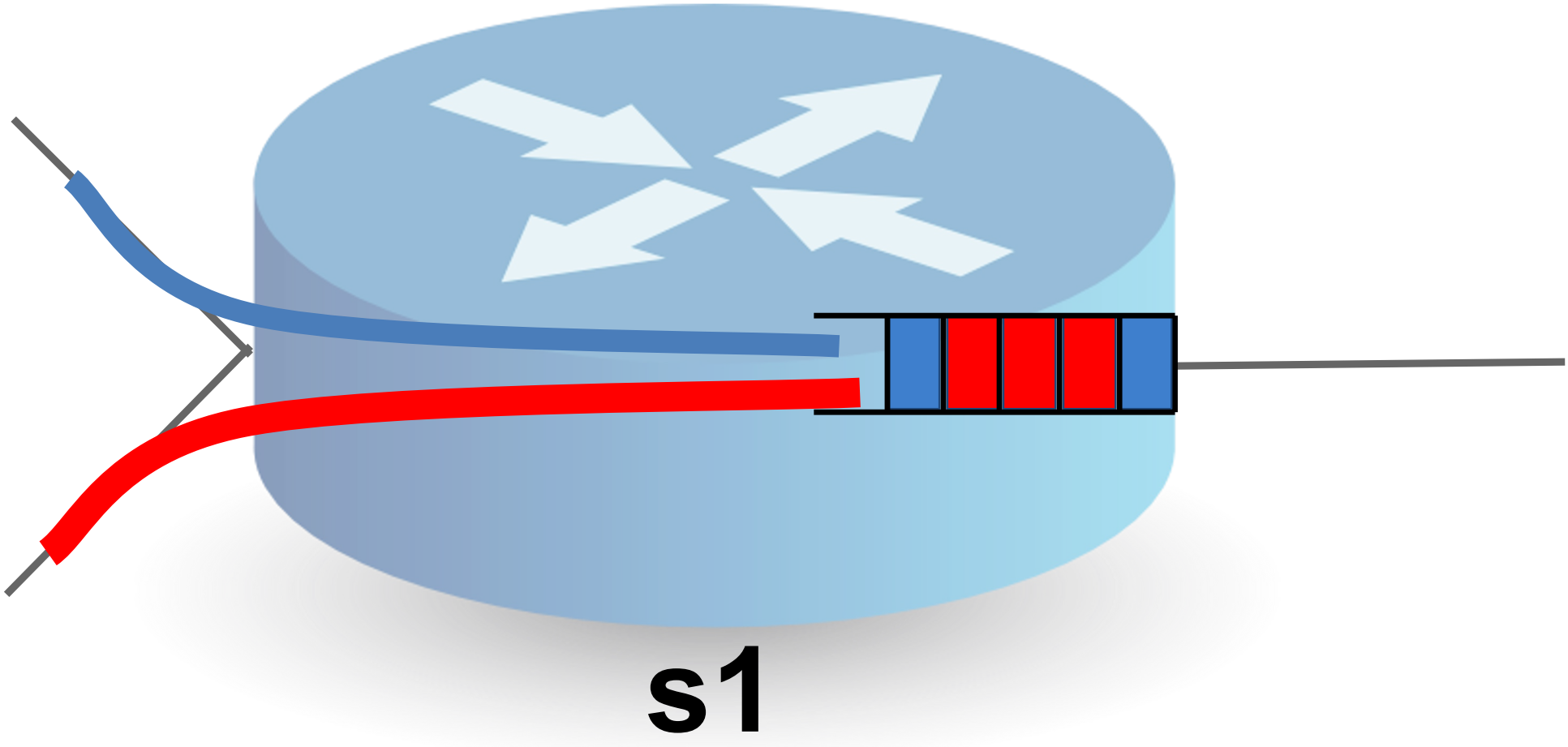
Exercise 3: Monitoring & Debugging

Explicit Congestion Notification - ECN

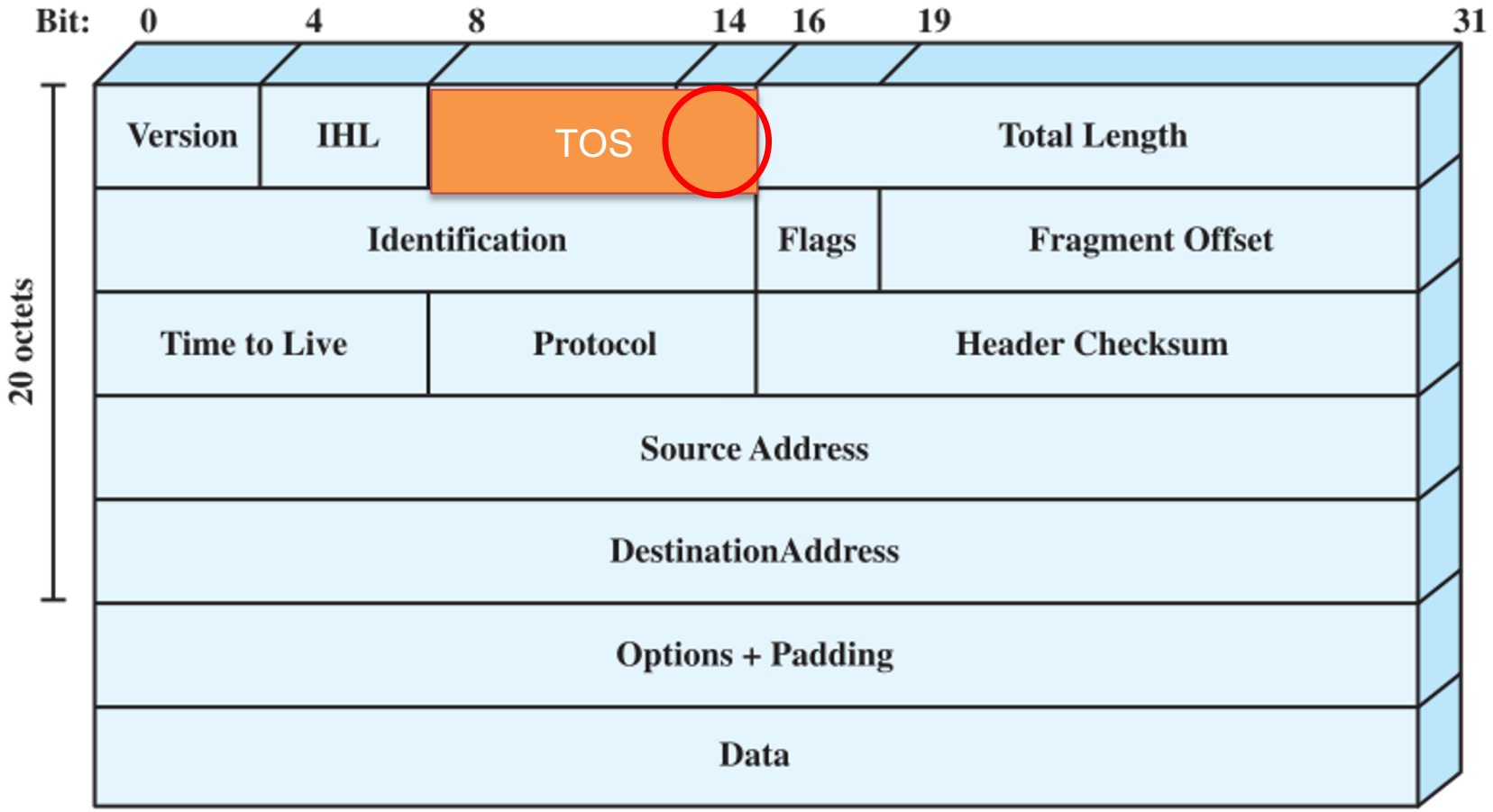
Monitoring & Debugging



Monitoring & Debugging



Explicit Congestion Notification



Explicit Congestion Notification

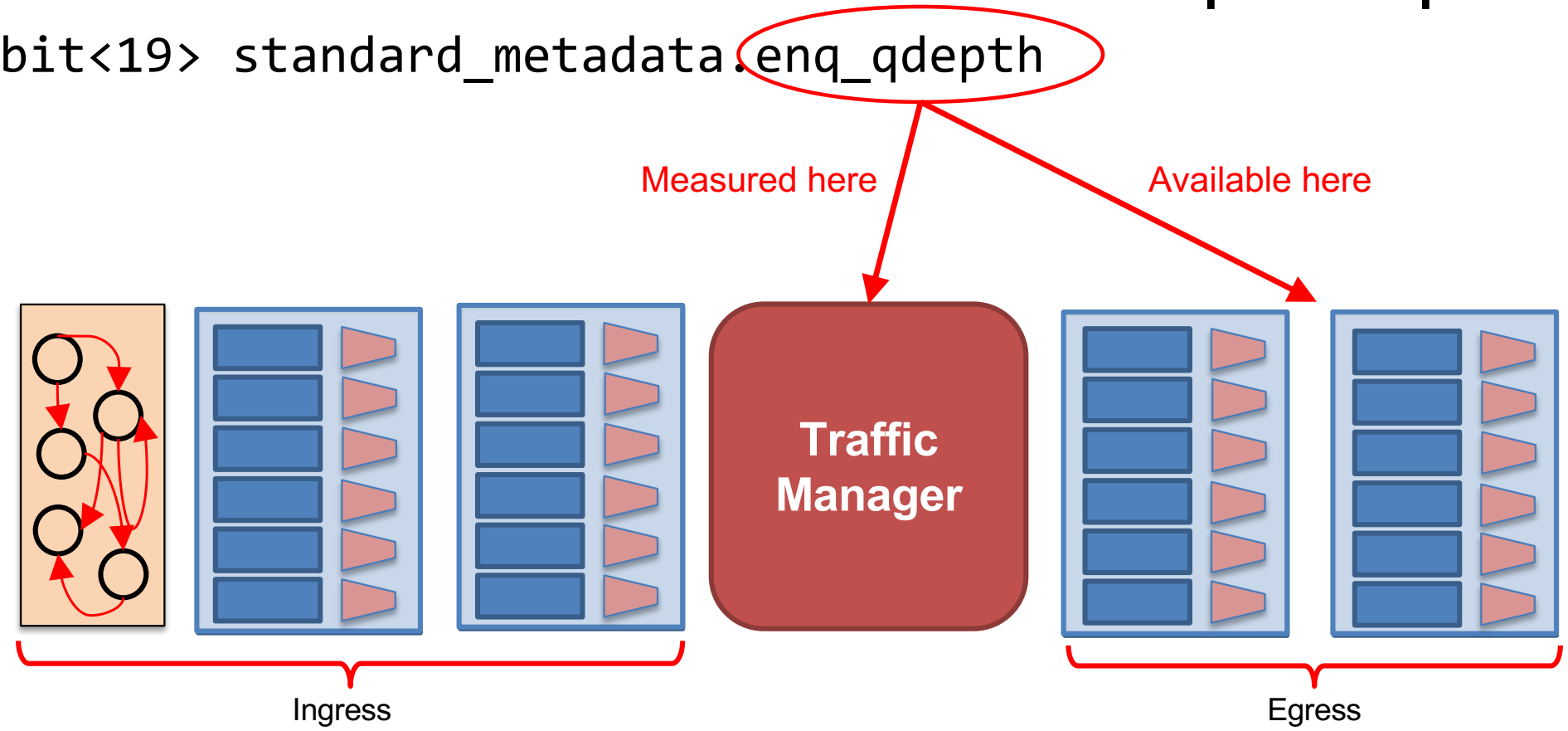
- **Explicit Congestion Notification**
  - `00`: Non ECN-Capable Transport, Non-ECT
  - `10`: ECN Capable Transport, ECT(0)
  - `01`: ECN Capable Transport, ECT(1)
  - `11`: Congestion Encountered, CE

- **For packets originating from ECT, ECN-capable switches set the CE bit upon congestion**
  - E.g., observed queue depth > threshold
  - more details: IETF RFC 3168
    - https://tools.ietf.org/html/rfc3168

Explicit Congestion Notification in P4

- **The standard data for the V1Model includes the queue depth:**

```
bit<19> standard_metadata.enq_qdepth
```



ECN marking

- **ToDo: Add support for ECN marking to the basic IP router**

- **Desired behavior:**
  - If an end-host supports ECN, it puts the value of 1 or 2 in the ipv4.ecn field.
  - For such packets, each switch may change the value to 3 if the queue size is larger than a threshold.
  - The receiver copies the value to sender, and the sender can lower the rate.

ECN marking in P4

- **modify `ipv4_t` to split TOS into DiffServ and ECN**
- **update checksum accordingly**
- **In egress, compare queue length with ECN_THRESHOLD**
  - if queue is larger, set ECN bits to 3 (bin 11) (congestion encountered)
  - do this only if end-host supports by having set original ECN to 1 or 2
- **Define an action to drop a packet which calls mark_to_drop();**
- **Define an egress control block that checks the ECN and `standard_metadata.enq_qdepth` and sets the `ipv4.ecn` accordingly**
- **test your solution by redirecting the receive.py to a log file to check the TOS**

Advanced P4 Constructs

Data Types, Externs, Registers, Meters, etc.

Different Data Types in P4

`bool`	Boolean values
`bit<W>`	Bit-strings of width W
`int<W>`	Signed integer of width W
`varbit<W>`	Bit-string with dynamic length (max W)
~~`float`~~	no support
~~`string`~~	no support

Operators to define composed types in P4

Header

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

We have shown already struct

Header Union

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

either IPv4 or IPv6 header

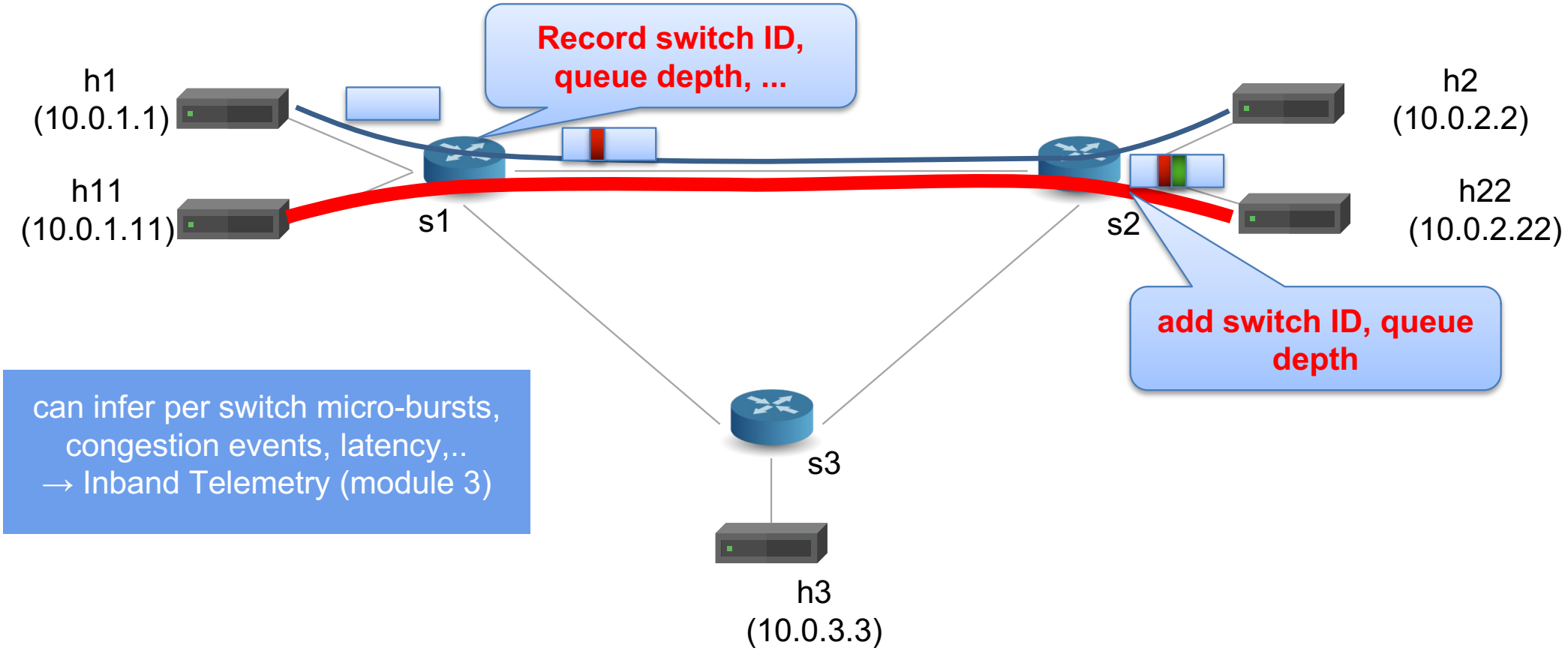
Header Stack

```
header mpls_h {
        bit<20>         label;
        bit<3>          tc;
        bit             BoS;
        bit<8>          ttl;
}
```

mpls_h[8] mpls;

Array of up to 8 MPLS headers

Example Use of Header Stacks - MultiRoute Inspect



can infer per switch micro-bursts, congestion events, latency,..
→ Inband Telemetry (module 3)

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 operations:**
  - `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)`

State Management in P4

- **Stateless Objects**
  - Variables (metadata), headers,..do not maintain state across packets
- **Stateful Objects**
  - Tables
  - Externs in P4-14: Counters, Meters, ...keep state across different packets

Object	Data Plane Interface		Control Plane Can	
	Read State	Modify/Write State	Read	Modify/Write
Table	apply()	---	Yes	Yes
Parser Value Set	get()	---	Yes	Yes
Counter	---	count()	Yes	Yes*
Meter	execute ()		Configuration Only	Configuration Only
Register	read()	write()	Yes	Yes

WriteRequest with the MODIFY update type

P4 Registers

- Store arbitrary data (single values or arrays of N entries)
- Definition:

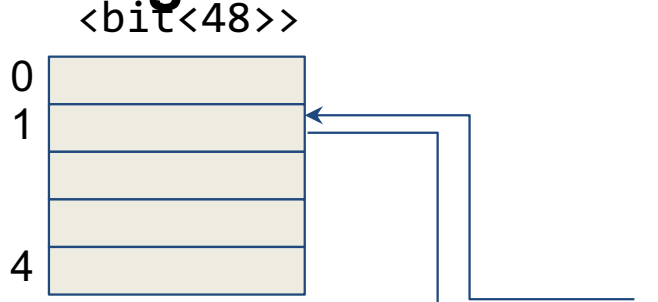
```
register<Type>(N) reg;
```

- writing:

```
reg.write(n, val) //0 <= n <N
```

- reading into result variable:

```
reg.read(result,n)
```




```
register <bit<48>>(5) hello;

hello.write(1,0xff);
hello.read(res, 1);
```

Example: Inter packet gap detection

```
register<bit<32>>(8192) flowlet_lasttimeseen;

action flowlet_gap(out bit<32> delta, bit<32> flow_id)
{
    bit<32> last_pkt_seen;

    /* Get the time the previous packet was seen for same flow */
    flowlet_lasttimeseen.read(last_pkt_seen, flow_id);

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

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

Caveat: concurrent read and write needs to be synchronized if required

What is this code doing?

P4 Counters

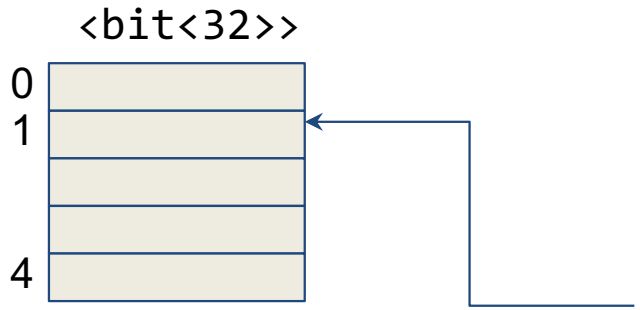
- used to count packets, bytes or both, formed in arrays
- Definition:

```
counter(N, Type) my_count;
```

- Updating:

```
my_count.count(n)
```

```
enum CounterType {
    packets,
    bytes,
    packets_and_bytes
}
```




```
counter (5,CounterType.packets_and_bytes) hello;
```

```
hello.count(1); //increases counter at position 1
with current packet size information
```

Example: Count incoming packet and bytes per port

```
control MyIngress(...) {

    counter(64, CounterType.packets_and_bytes) c;

    apply { //ingress port number as index
        c.count((bit<32>)standard_metadata.ingress_port);
    }
}
```

Interaction from Control Plane

```
RuntimeCmd: counter_read MyIngress.c 1   //will then return
MyIngress.c[1] = BmCounterValue(packets=1, bytes=658)

//Note: cannot access counter information from within data plane
```

Example: counter use

```
control MyIngress(...) {
     counter(64, CounterType.packets) c;

  action tally() {
    c.count((bit<32>) standard_metadata.ingress_port);  }

  table monitor {
    key = {
      hdr.ipv4.srcAddr: lpm;   }
    actions = { tally; NoAction; } }

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

Question: What this P4 code does?

```
{
  "table": "MyIngress.monitor",
  "match": {
    "hdr.ipv4.srcAddr": ["10.0.1.1", 32]
  },
  "action_name": "MyIngress.tally",
  "action_params": { }
}
```

Example: using direct counters

```
control MyIngress(...) {
    direct_counter(CounterType.packets) c;

  action tally() {
    c.count();  }

  table monitor {
    key = {
      hdr.ipv4.srcAddr: lpm;   }
    actions = { tally; NoAction; }
    counters = c;
    size =1024}
  apply {
    ...
    if(hdr.ipv4.isValid()) {
      ...
      monitor.apply();
    }
```

- Direct counters are attached to tables
- Each table entry has a counter that counts upon match

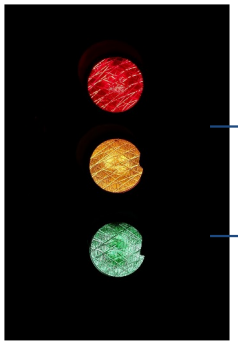
Question: What this P4 code does?

```
{
  "table": "MyIngress.monitor",
  "match": {
    "hdr.ipv4.srcAddr": ["10.0.1.1", 32]
  },
  "action_name": "MyIngress.tally",
  "action_params": { }
}
```

P4 Meters

- **used to measure packet rates, can be formed in arrays**
- **Definition:** `meter(N, Type) my_meter;`
- **Applying:** `my_meter.execute(n)`

```
enum MeterType {
    packets,
    bytes
}
enum MeterColor {
    RED, GREEN, YELLOW }
```

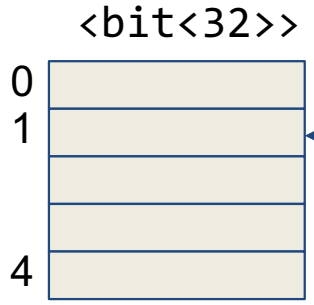


more info https://tools.ietf.org/html/rfc2698

exceeds Peak Information Rate (PIR) per sec.

exceeds Committed Information Rate (CIR) but still below PIR per sec.

does not exce




```
meter (5,MeterType.packets) hello;
```

```
hello.execute(1);
```

Rates set by control plane per instance

```
meter_set_rates hello 1 0.0001:1 0.0005:1
.0001=> allow 100packets/sec , if the each
packet size is equal to 1000 bytes, then the
obtained throughput can be 100 packets/sec *
1000 (bytes) *8 = 800 kbps
```

Example: Rate-limitig with meters

```
control MyIngress(...){
 meter(16384, MeterType.packets) acl_meter;

action color_my_packets(bit<32> index) {
       acl_meter.execute_meter((bit<32>)index, meta.meta_tag);
}

table m_read {
       key = { hdr.ethernet.srcAddr: exact; }
       actions = { color_my_packets; NoAction; }
       …
       }

table m_filter {
       key = { meta.meta_tag: exact; }
       actions = { drop; NoAction; }
       ...
}
apply {
       m_read.apply();
       m_filter.apply();
}
}
```

- **can also use direct meters**
- **direct meters are attached to table entry**

```
// This is v1 model code
// https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4
// meta.meta_tag now holds the color
```

depending on the meter tag, treat packets differently

Question: What this P4 code does?

Example: Rate-limitig with direct meters

```
control MyIngress(...){
 direct_meter(16384, MeterType.packets) acl_meter;

action color_my_packets(bit<32> index) {
       acl_meter.read(meta.meta_tag);
}

table m_read {
       key = { hdr.ethernet.srcAddr: exact; }
       actions = { color_my_packets; NoAction; }
       meters = acl_meter;
       …
       }

table m_filter {
       key = { meta.meta_tag: exact; }
       actions = { drop; NoAction; }
       ...
}
apply {
       m_read.apply();
       m_filter.apply();
}
}
```

```
// This is v1 model code
// https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4
// meta.meta_tag now holds the color
```

```
@ControlPlaneAPI
  {
    reset(in MeterColor_t color);
    setParams(in S index, in MeterConfig config);
    getParams(in S index, out MeterConfig config);
  }
  */
```

attach acl_meter to table m_read

Summary

- Several Stateful constructs to record and update per flow/packet state
- Useful for many things,
  - e.g. congestion tracking, stateful forwarding,....
- Will see some of these concepts applied in next module
  - e.g. congestion aware load-balancing

Summary - P4

- **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

Summary - P4 things we covered

- **The P4 world**
  - Protocol Independent Packet Processing
  - Language/Architecture separation
  - If you can interface with it, you can use it
- **Key data types**
- **Constructs for packet parsing**
  - state-machine type
- **Constructs for packet processing**
  - Actions, tables, controls
- **Packet Deparsing**
- **Architectures and Programs**

Summary - P4 things we did not cover

- **Enforcing Modularity**
  - Instantiating and invoking parsers or controls
- **Variable Length field processing**
  - parsing and deparsing of TLVs
- **Architecture definition constructs**
  - How to create such template definitions
- **Advanced features**
  - learning, multicast, cloning, resubmitting
  - header unions
- **Control Plane Interface**

Next

- **Datacenter Load-balancing**
  - What is a datacenter
  - What is data center networking
  - Traffic characteristics in a data center
  - Load-balancing techniques for data center networking
  - P4 based Load-balancing