DVAD41 - Introduction to Data Plane Programming

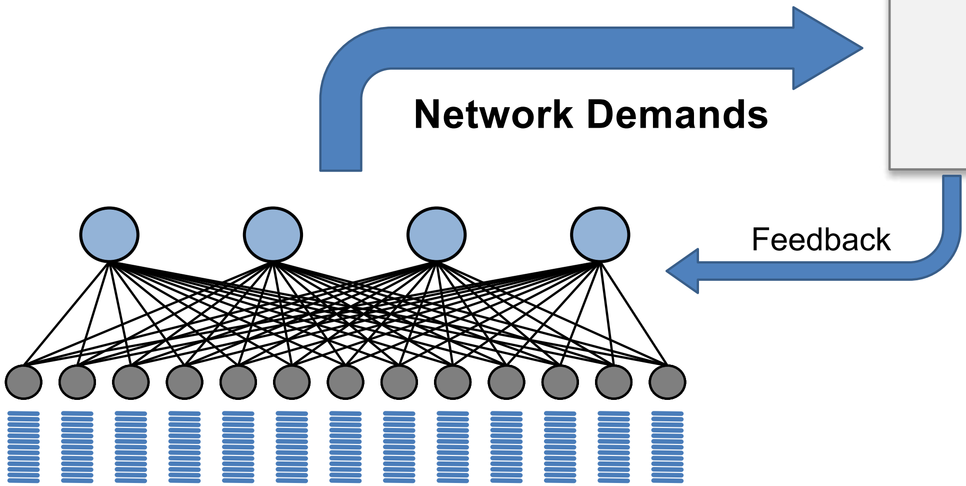
Webinar 3 - Tunneling

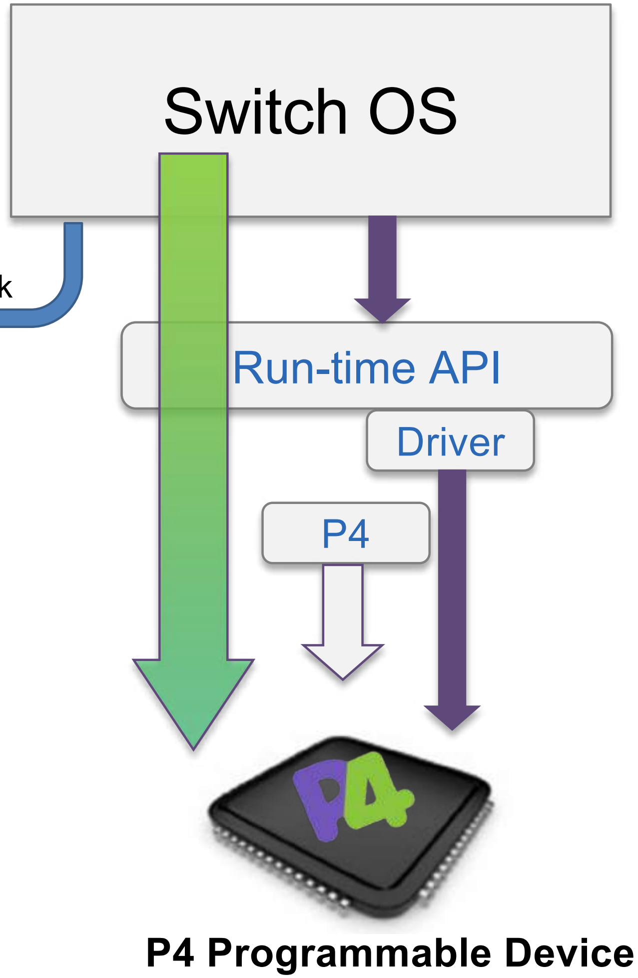
Exam Module 1

- **Proposal:**
  - available: 15th March EOD
  - Handin: 22nd march 2021

Recap on P4

P4: Top-down design

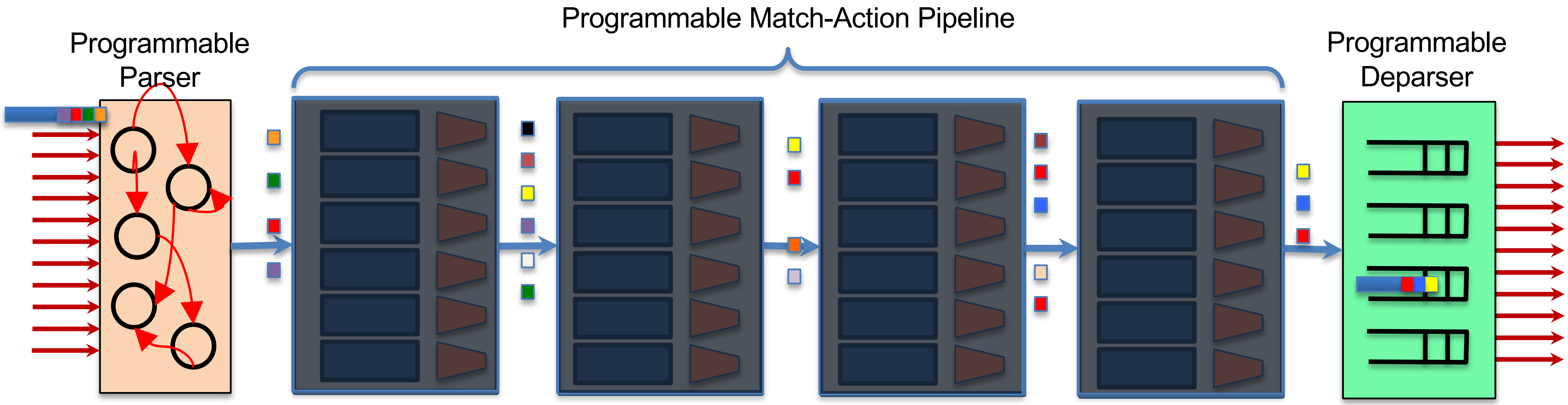




"This is *how I want* the network to behave and how to switch packets…"
(the user / controller makes the rules)

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

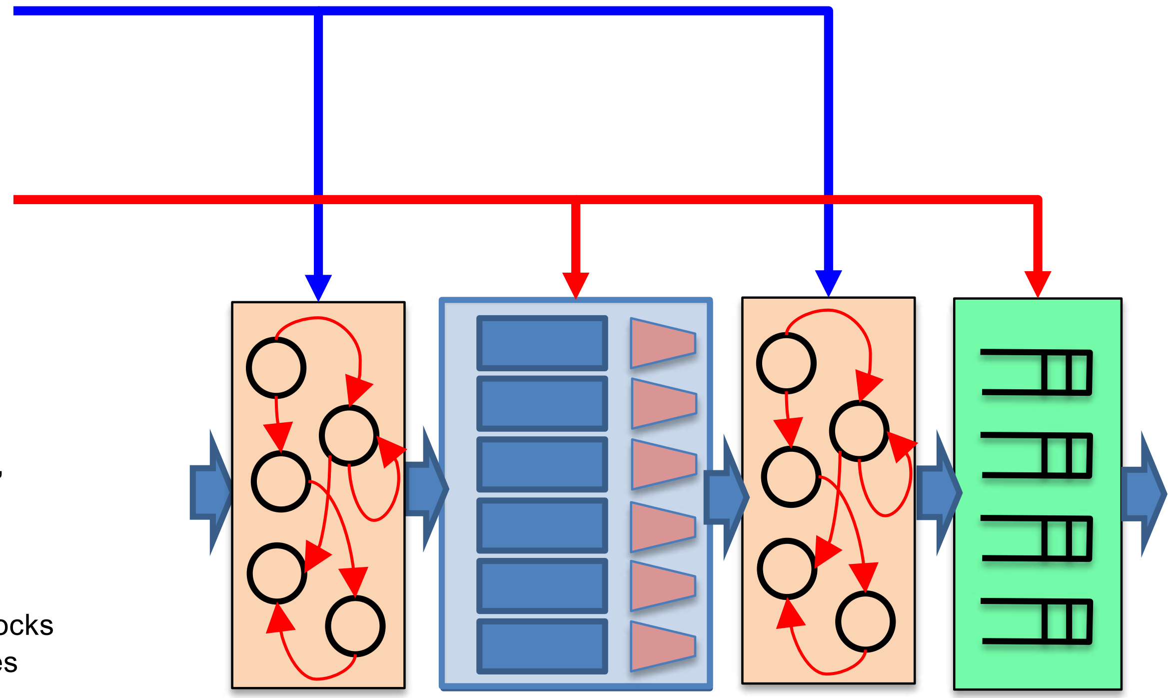
Controls — Tables, Actions, control flow statements

Expressions — Basic operations and operators

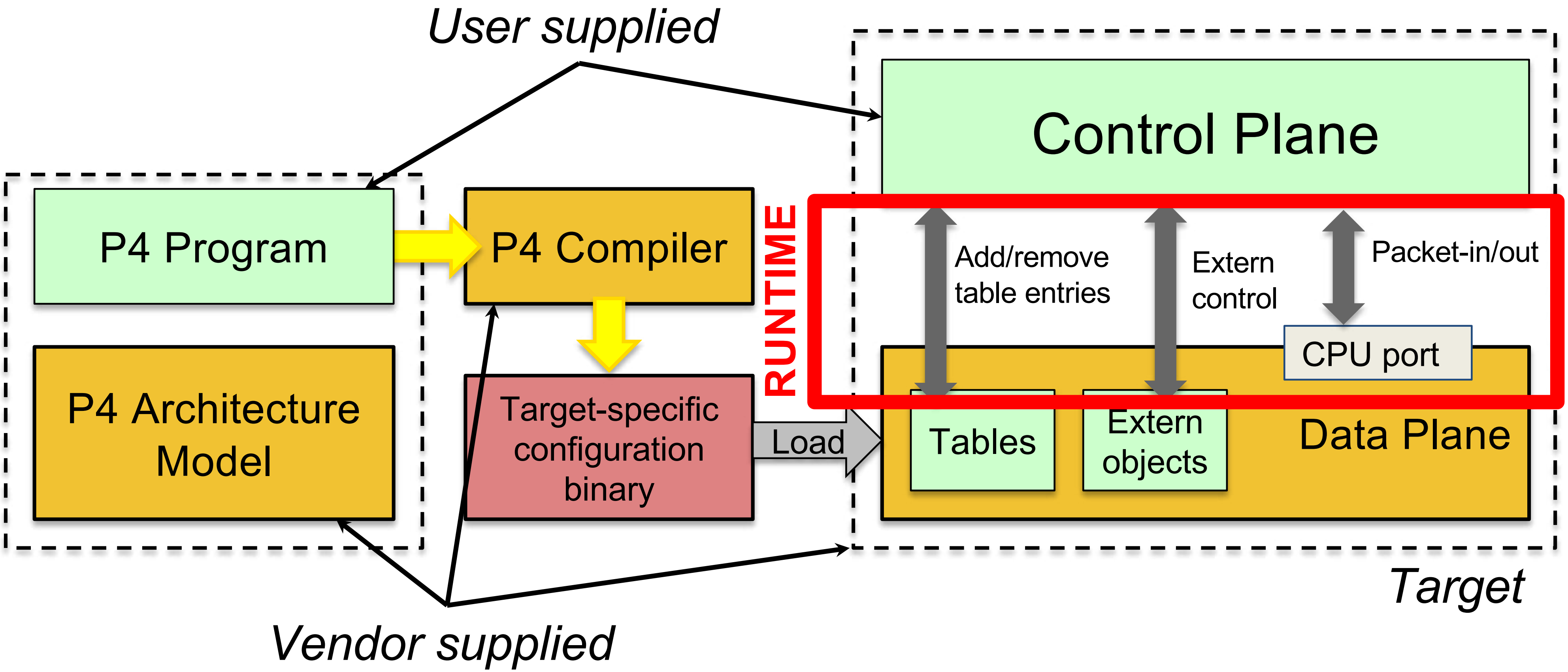
Data Types — Bistrings, headers, structures, arrays

Architecture Description — Programmable blocks and their interfaces

Extern Libraries — Support for specialized components

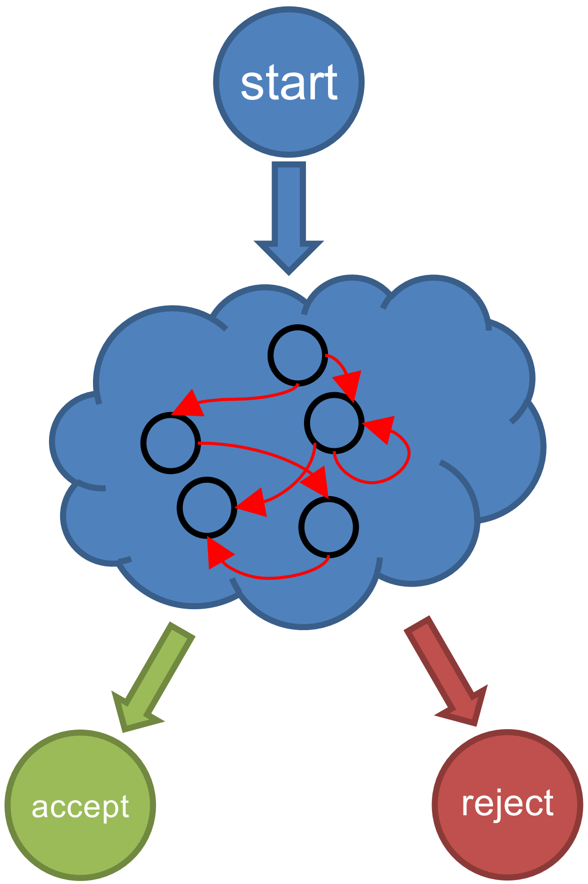


Programming a P4 Target



$P4_{16}$ 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)**



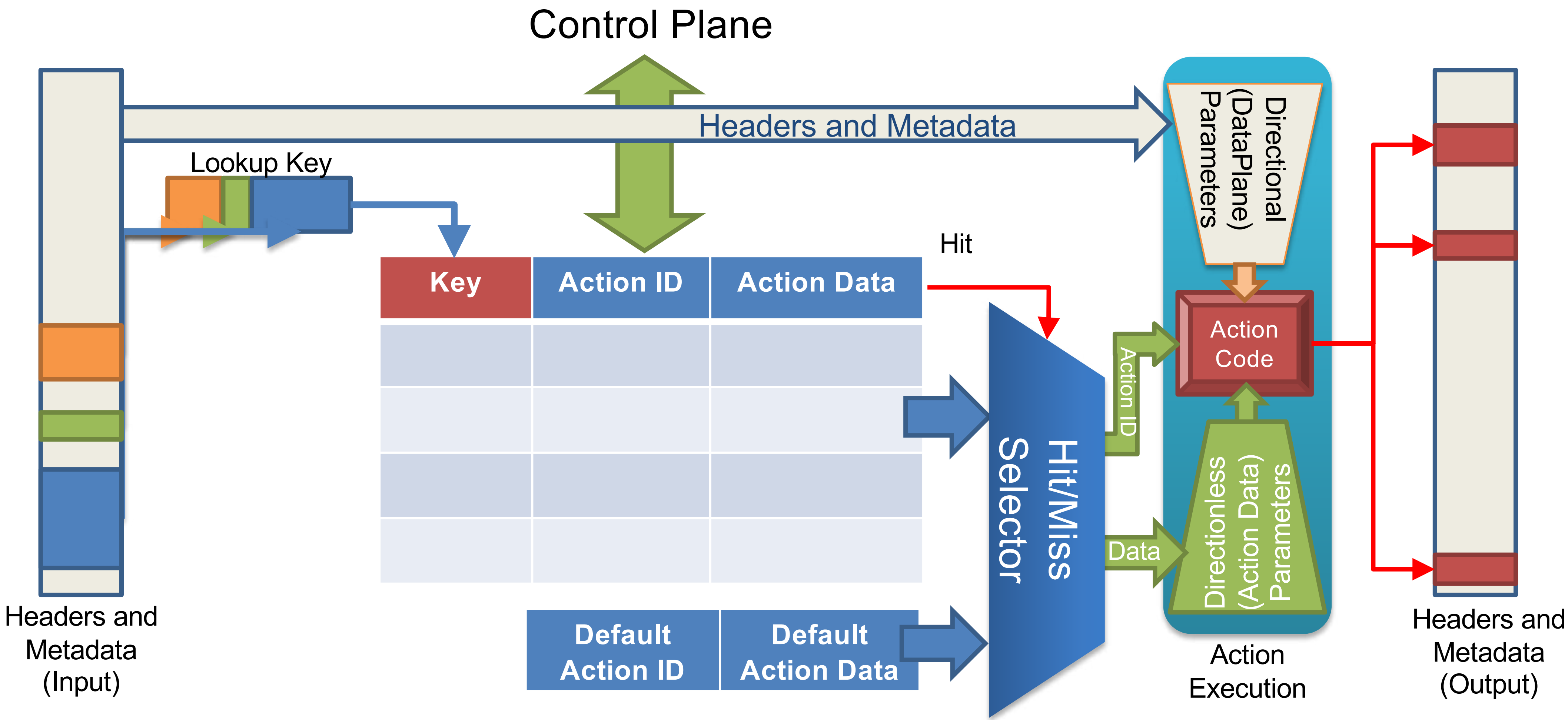
P4$_{16}$ Controls

- **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)

Tables: Match-Action Processing



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

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

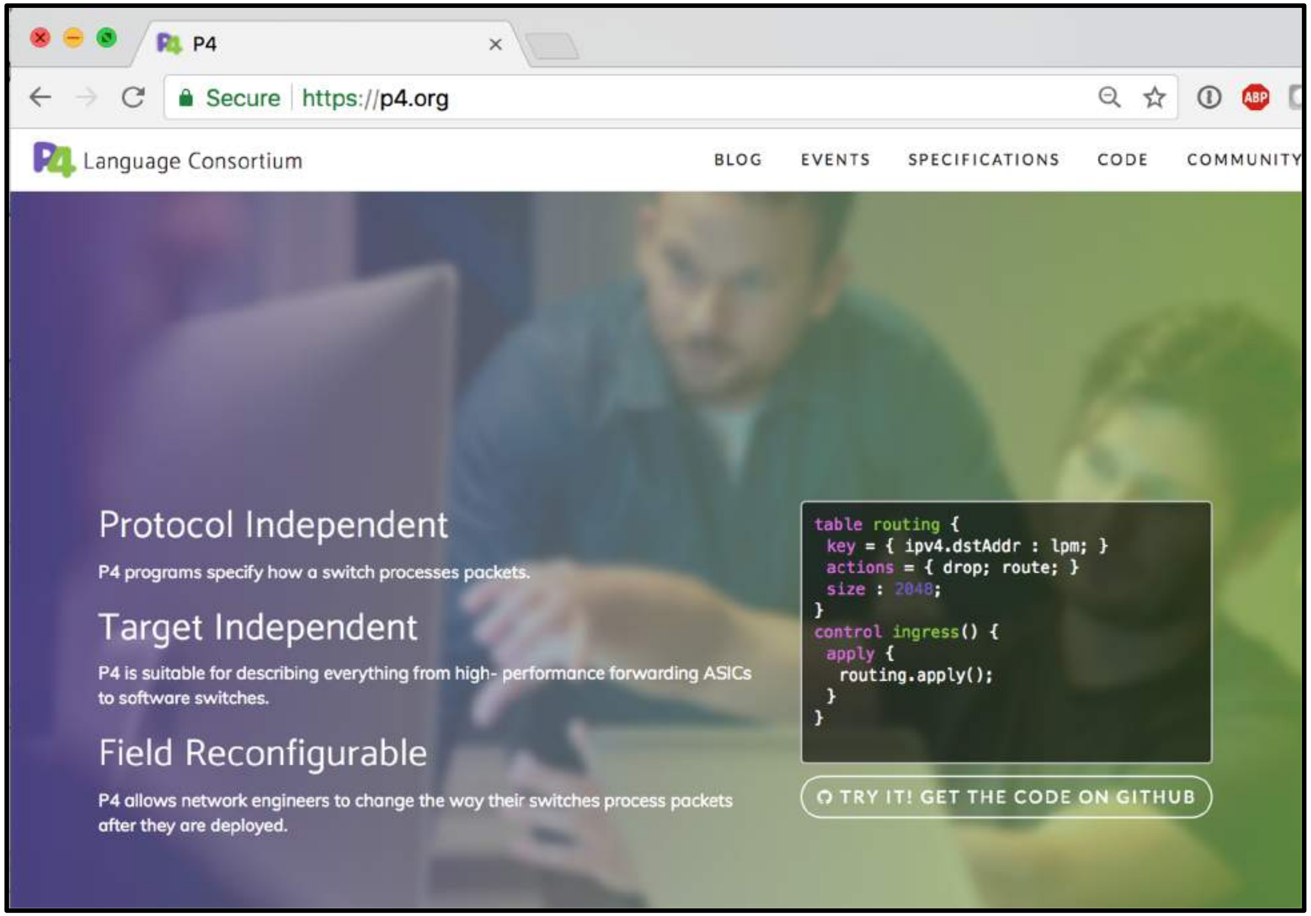
- **The P4 ”world view”**
  - Protocol-Independent Packet Processing
  - Language/Architecture Separation
  - If you can interface with it, it can be used
- **Key data types**
- **Constructs for packet parsing**
  - State machine-style programming
- **Constructs for packet processing**
  - Actions, tables 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
  - external functions
  - registers, meters, markers
- **Other architectures**
- **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**



Exercise 1: Recap

- **Mentimeter: www.menti.com and enter 204959**

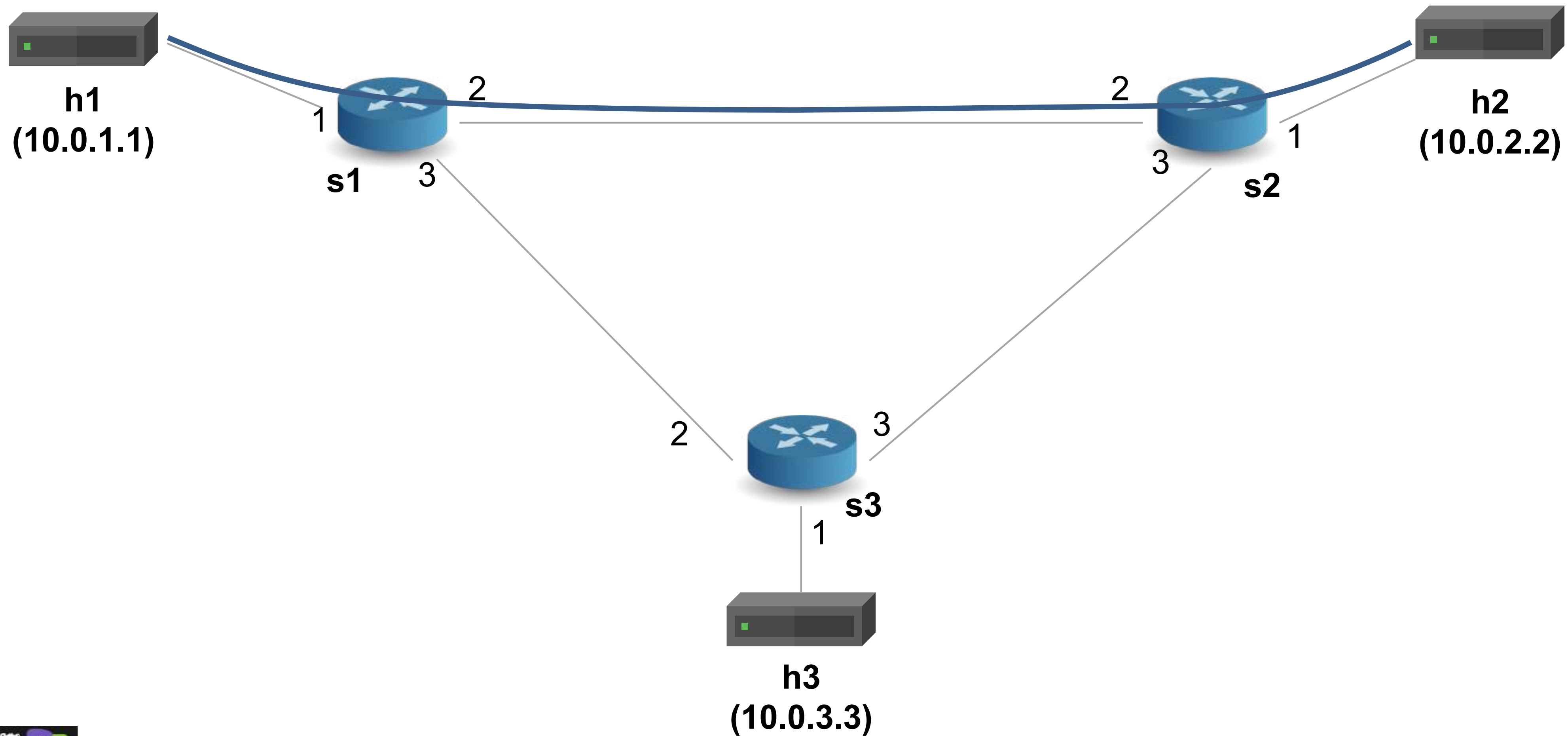
Running Example: Basic Forwarding

- **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**

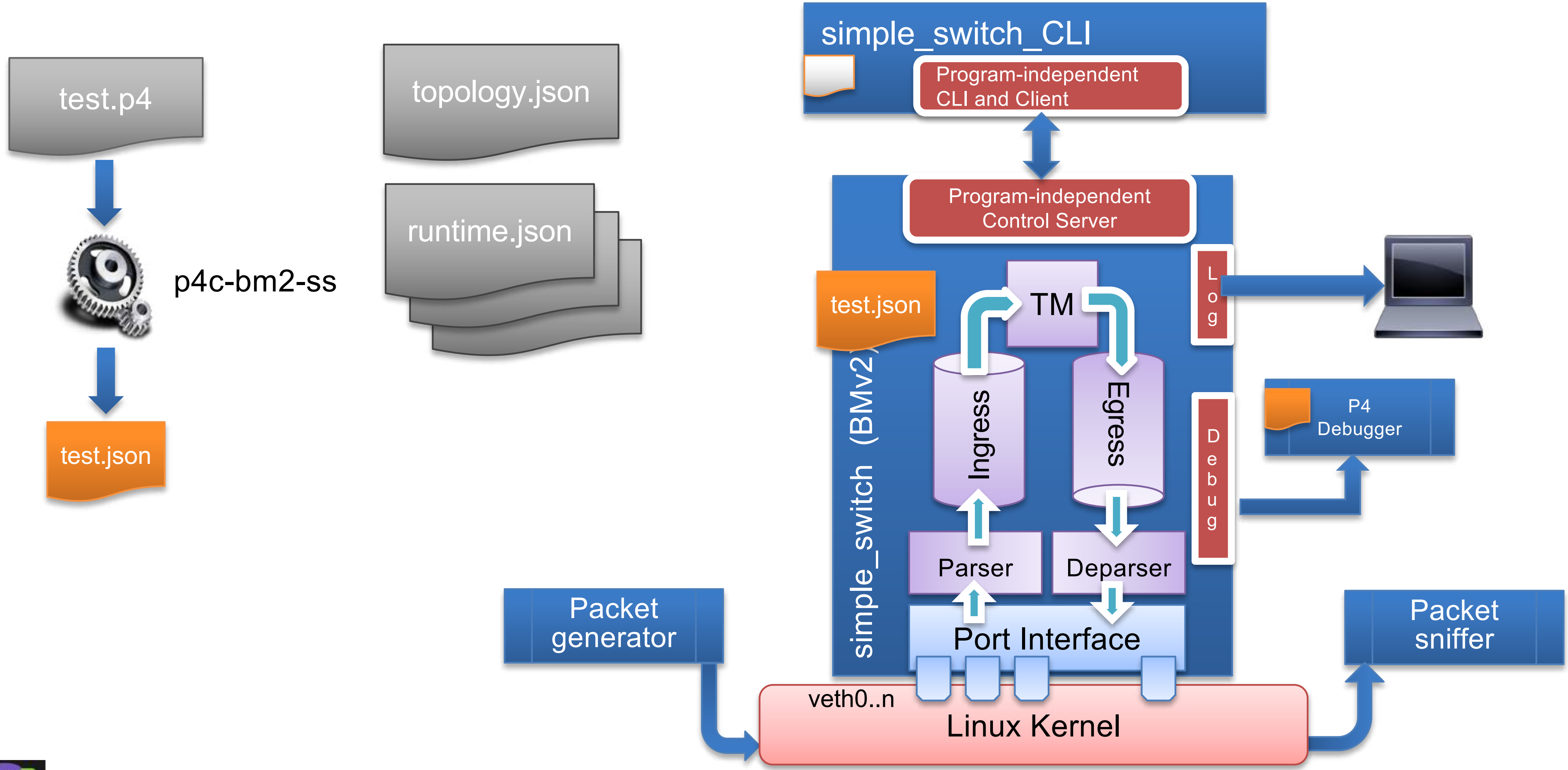
Basic Forwarding: Topology



Coding Homework

- **Complete the basic.p4**
  - The Ethernet and IPv4 headers have already been defined and added into the headers struct, but the parser block is empty so you must fill this in.
    - Begin by defining the states we want to use.
    - Parsers must always start in the start state.
  - Then define a state for the ethernet header as well as a state for the IPv4 header.
    - Packets will always begin with the Ethernet header so transition to the parse_ethernet state from the start state.
    - In this state, first extract the ethernet header and then branch based on the etherType field using the select statement that we saw earlier.
    - If the etherType field is equal to the IPV4_TYPE defined above, then transition to the parse_ipv4 state, otherwise the packet does not contain an IPv4 header so we are done
    - In the parse_IPv4 state, simply extract the IPv4 header and then you are done.

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]***

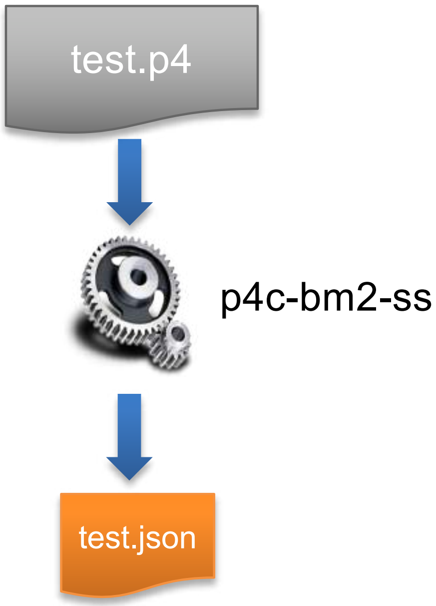
```
    sudo python $(RUN_SCRIPT) -t $(TOPO)
```

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

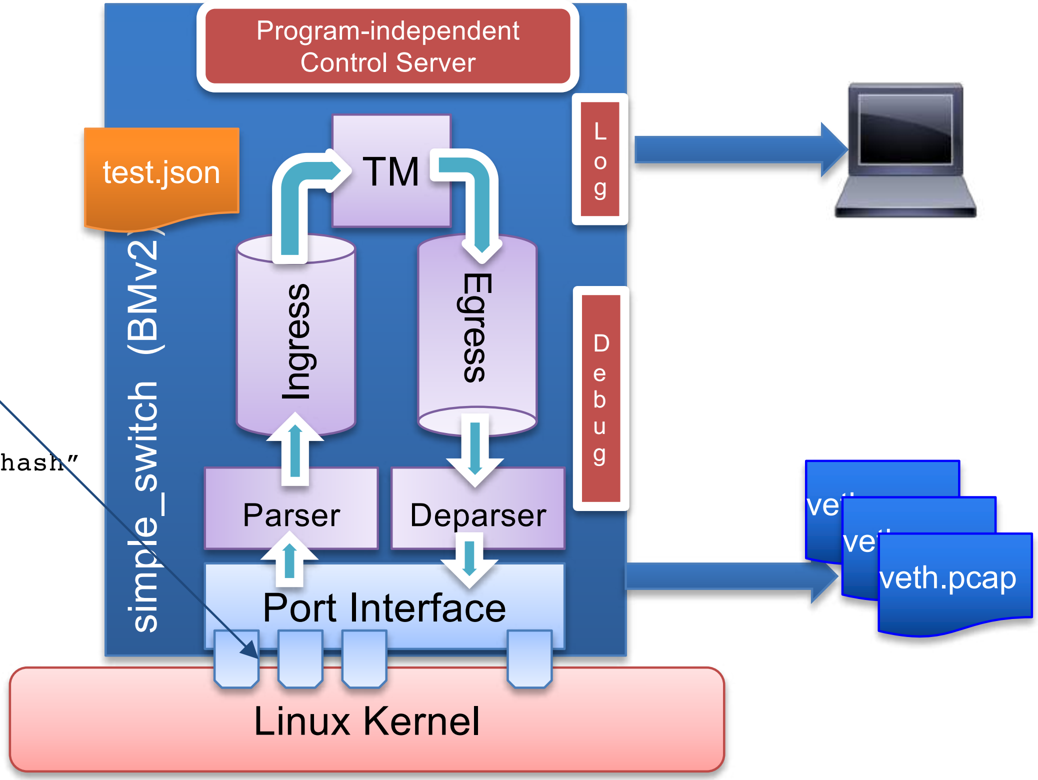
```
$ sudo simple_switch --log-console --dump-packet-data 64 \
–i 0@veth0 -i 1@veth2 ... [--pcap]
test.json
```

prepare veth interfaces first

```
# ip link add name veth0 type veth peer name veth1
# for iface in “veth0 veth1”; do

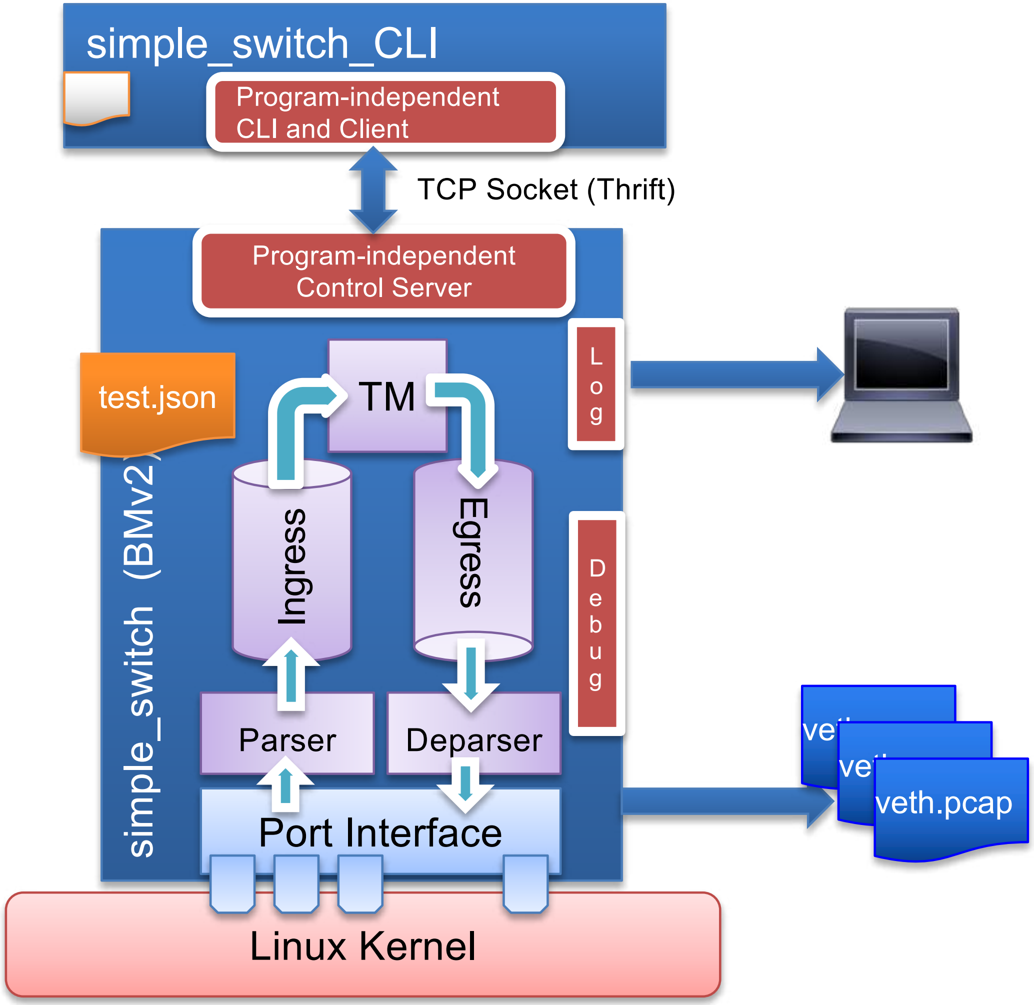
    ip link set dev ${iface} up
     sysctl net.ipv6.conf.${iface}.disable_ipv6=1
     TOE_OPTIONS="rx tx sg tso ufo gso gro lro rxvlan txvlan rxhash”
     for TOE_OPTION in $TOE_OPTIONS; do

        /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**
  - 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.**
  - 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`

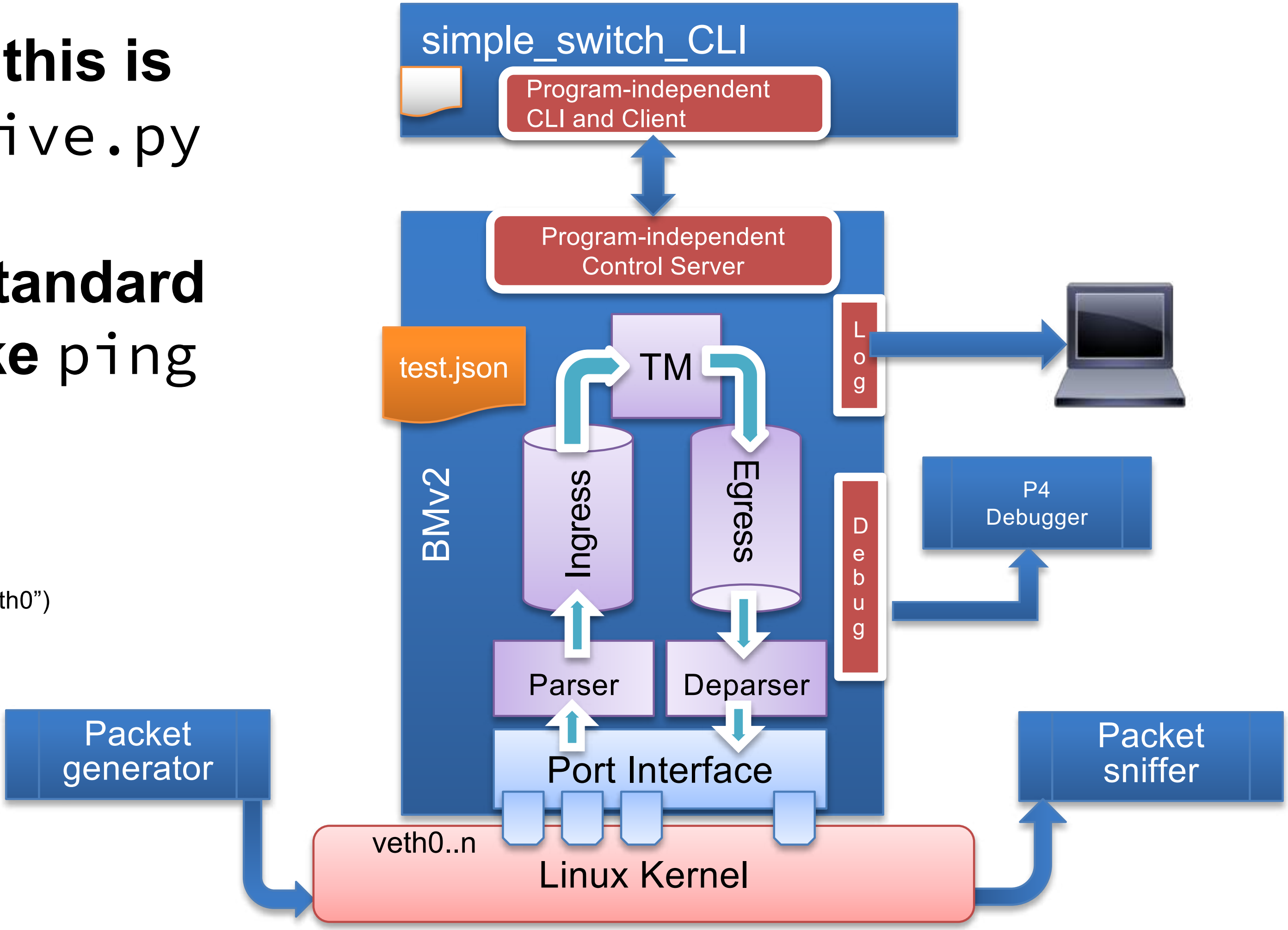
Can also use

scapy for sending

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

scapy for sniffing

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



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**

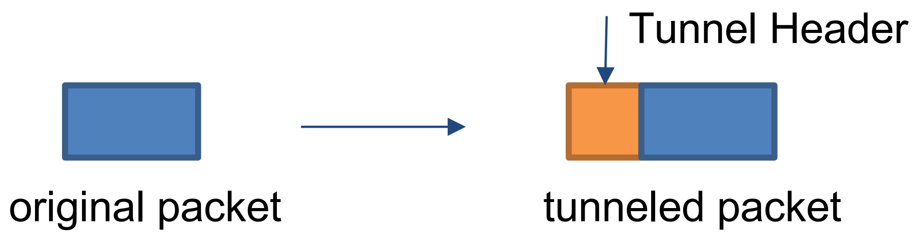
- ```
  [15:16:48.145] [bmv2] [D]
  [thread 4090] [96.0] [cxt 0]
  Looking up key:
  * std_meta.egress_spec : 2
  ```
  ```

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



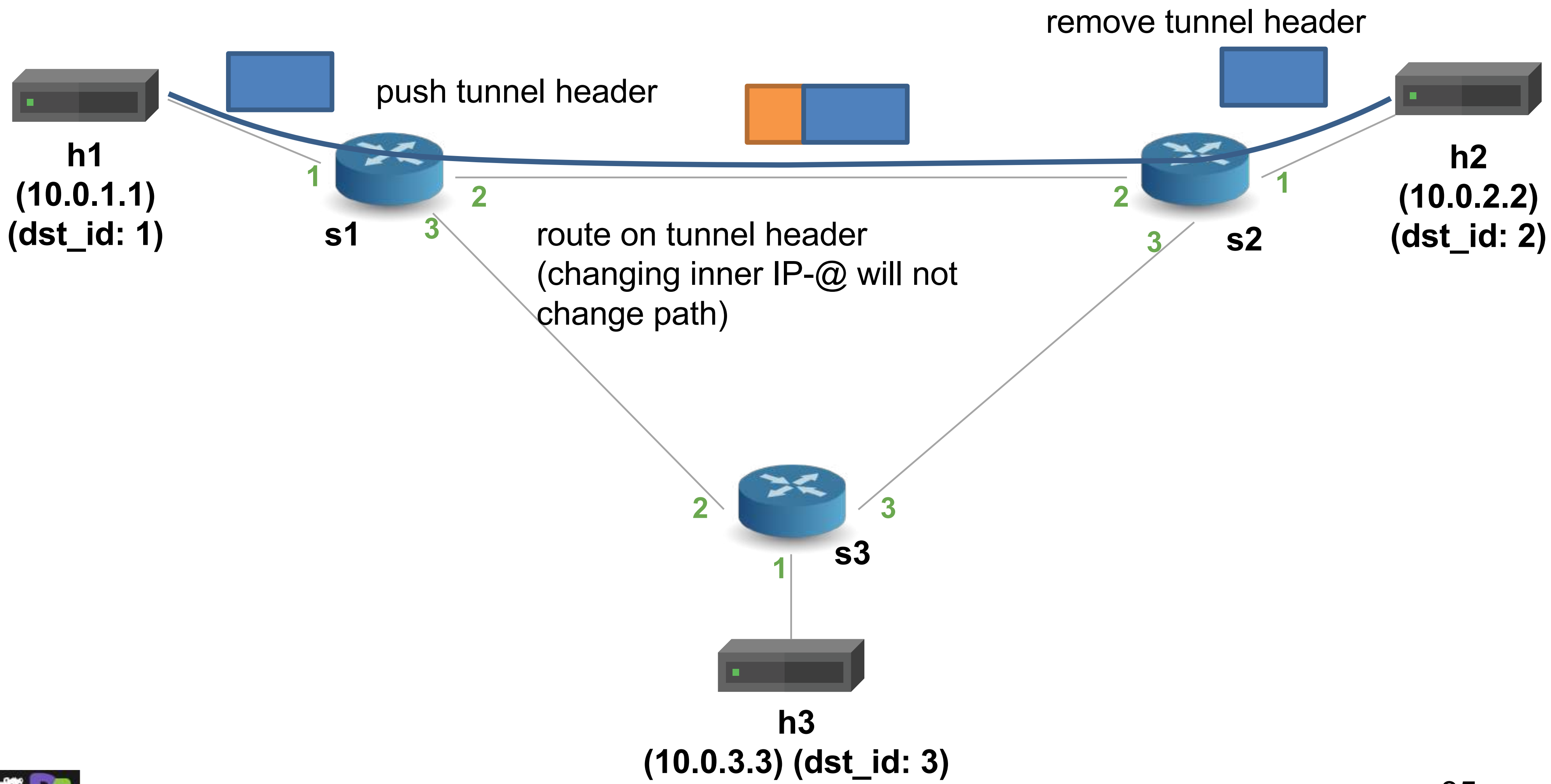
Basic Tunneling

- 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**