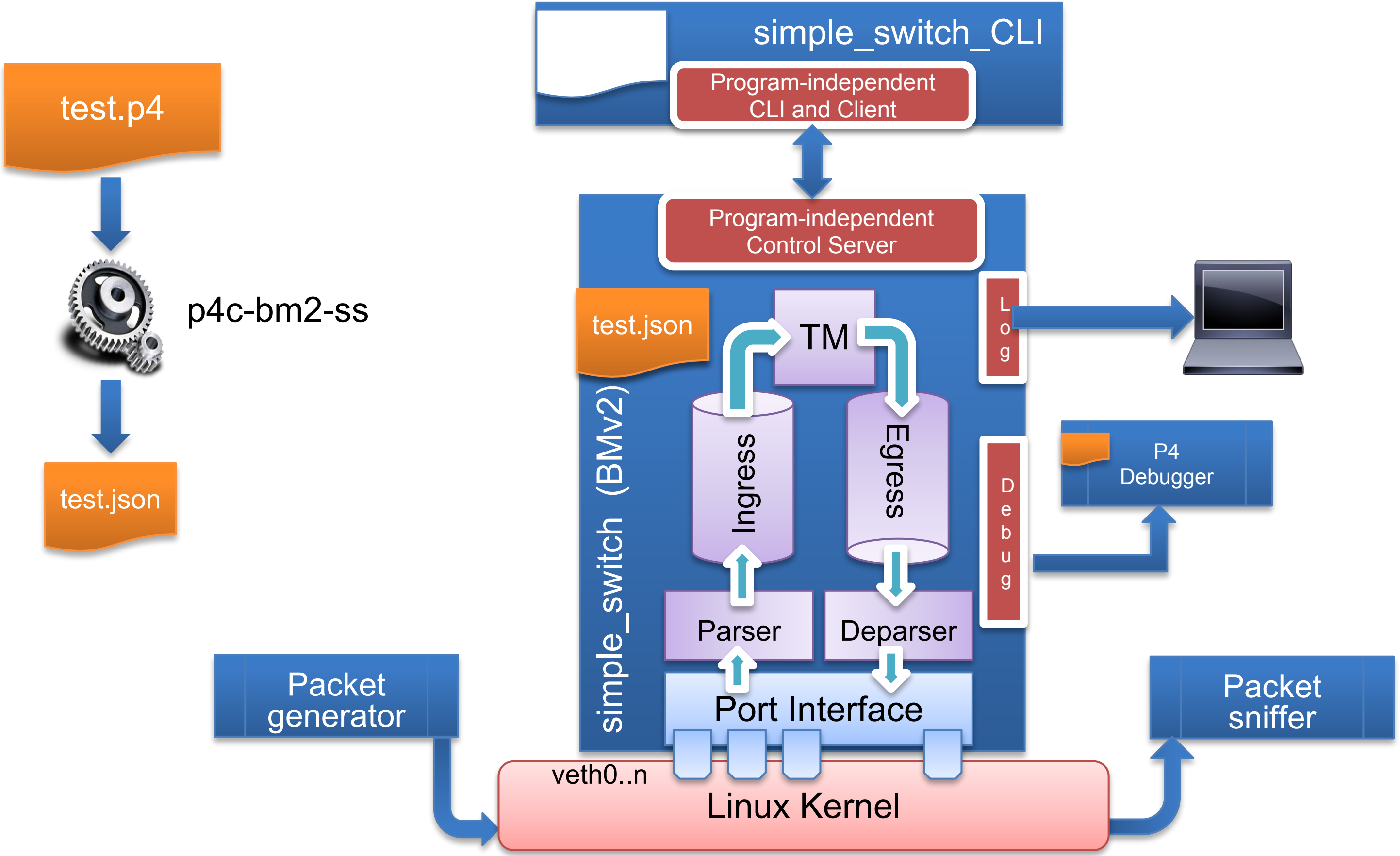
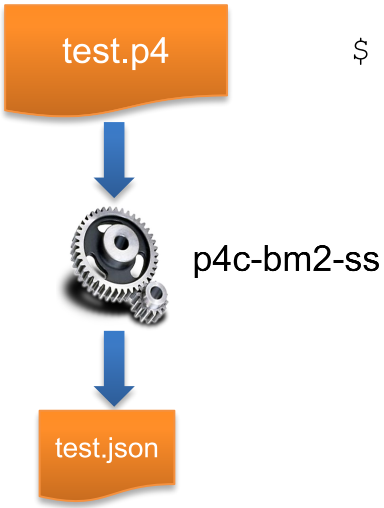
Lab 2: P4 Runtime

P4 Software Tools

Makefile: under the hood

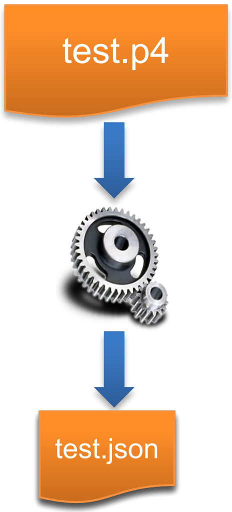


Step 1: P4 Program Compilation

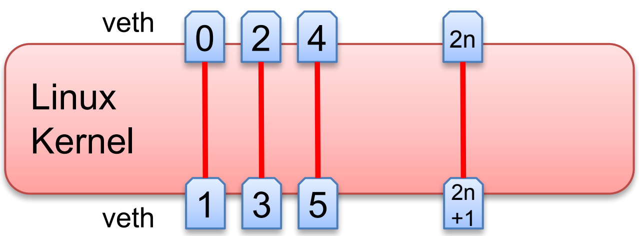


$ **p4c-bm2-ss** --p4v 16 -o test.json test.p4

Step 2: Preparing veth Interfaces

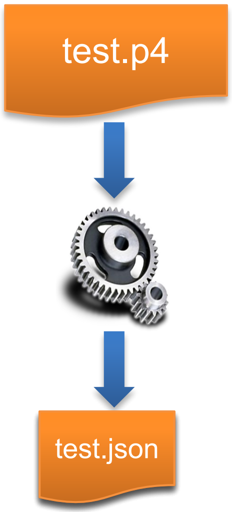


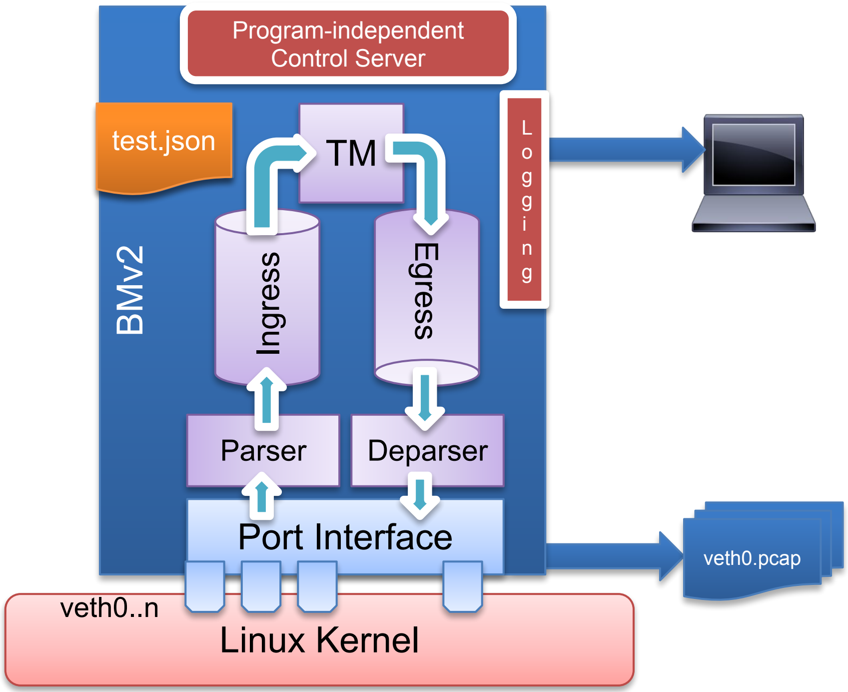

```
$ sudo ~/p4lang/tutorials/examples/veth_setup.sh

# 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 the model

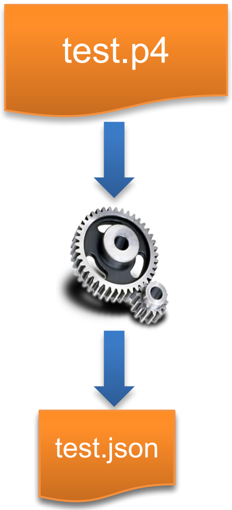
```
$ sudo simple_switch --log-console --dump-packet-data 64 \
       -i 0@veth0 -i 1@veth2 … [--pcap]                  \
       test.json
```

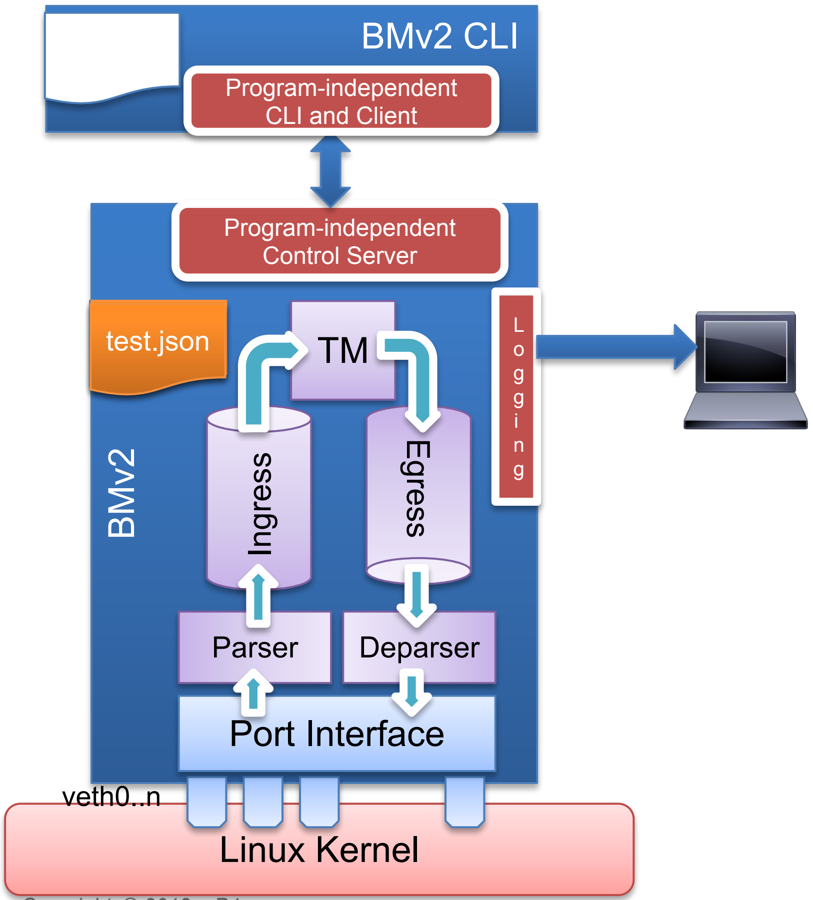




Step 4: Starting the CLI

```
$ simple_switch_CLI
```

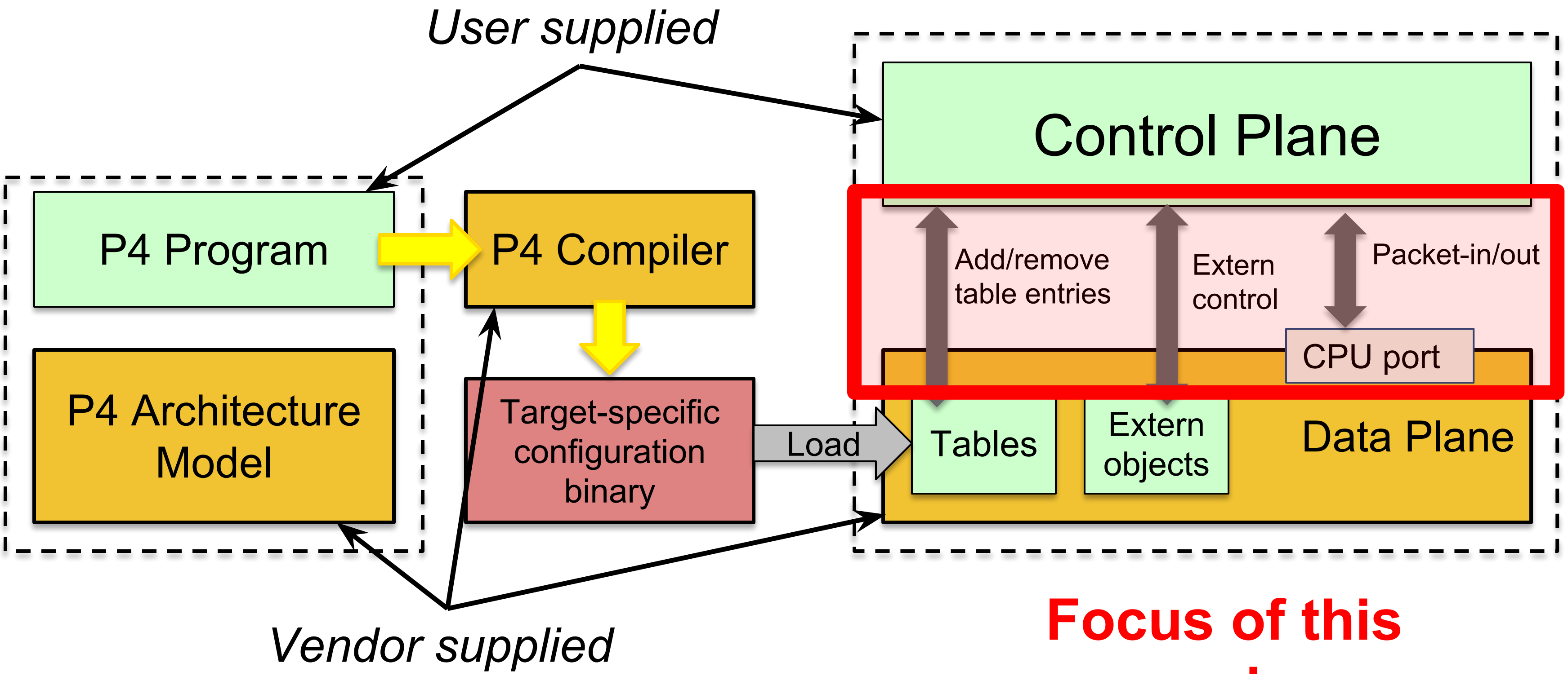




P4 Runtime

- API overview
- Workflow
- Exercise - Tunneling
- Example - ONOS

Runtime control of P4 data planes



Focus of this session

Existing approaches to runtime control

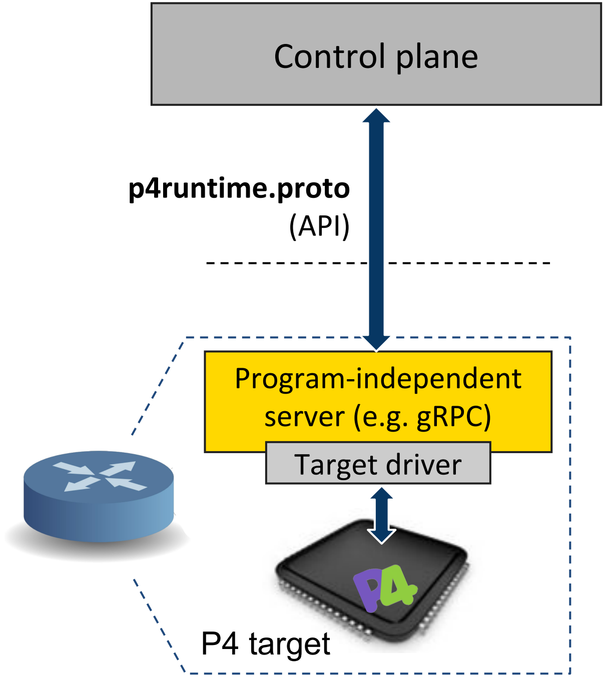
- **P4 compiler auto-generated runtime APIs**
  - Program-dependent -- hard to provision new P4 program without restarting the control plane!
- **BMv2 CLI**
  - Program-independent, but target-specific -- control plane not portable!
- **OpenFlow**
  - Target-independent, but protocol-dependent -- protocol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

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**
- **Protobuf-based API definition**
  - p4runtime.proto
  - gRPC as a possible RPC 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



More details on the P4Runtime API

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

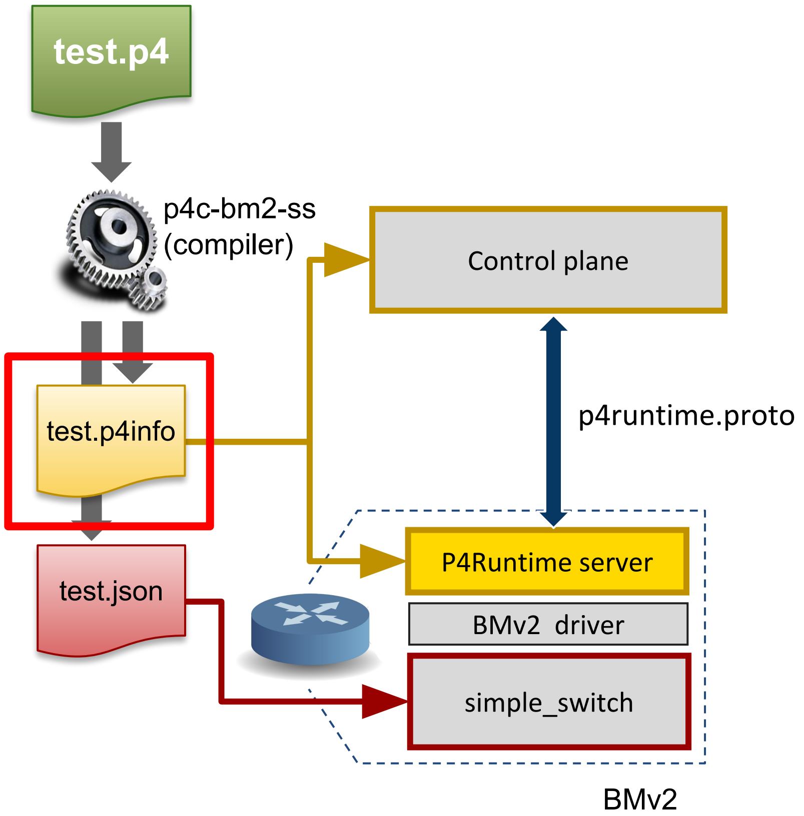
To add a table entry, the control plane needs to know:

- **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/PI/blob/master/proto/p4/config/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 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

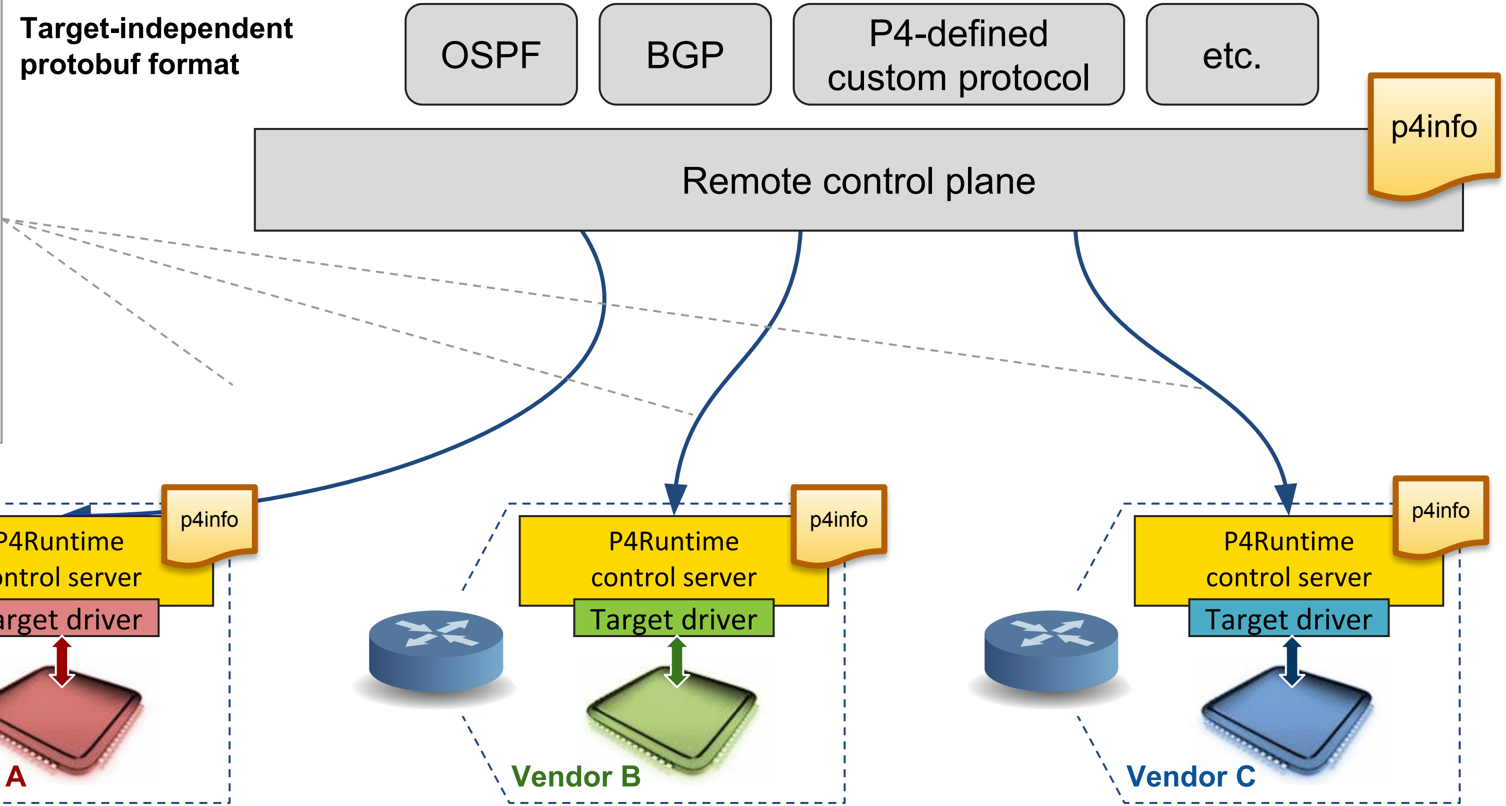
Protobuf message

```
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"
    }
  }
}
```

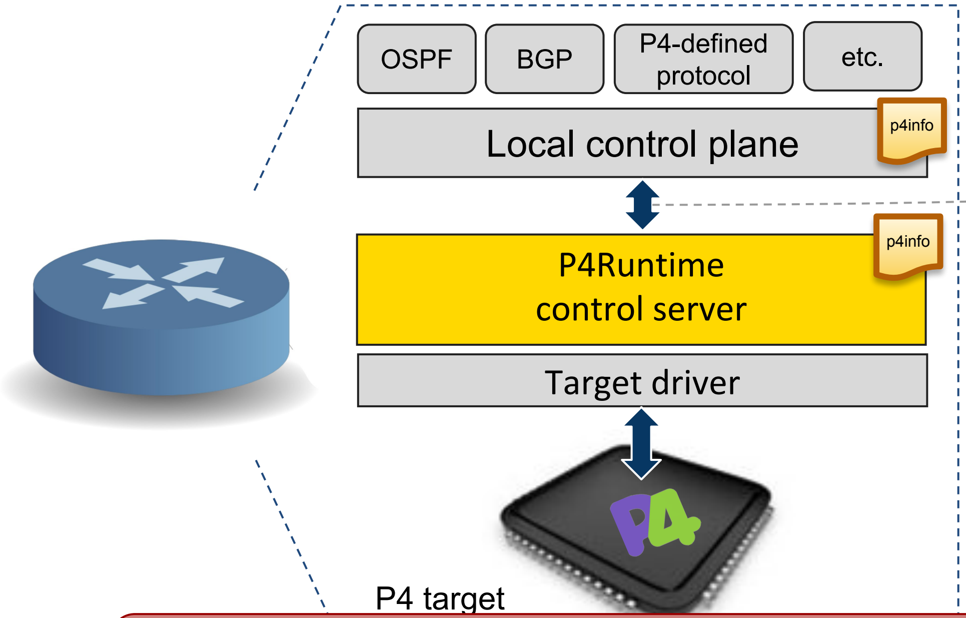
Remote control

```
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

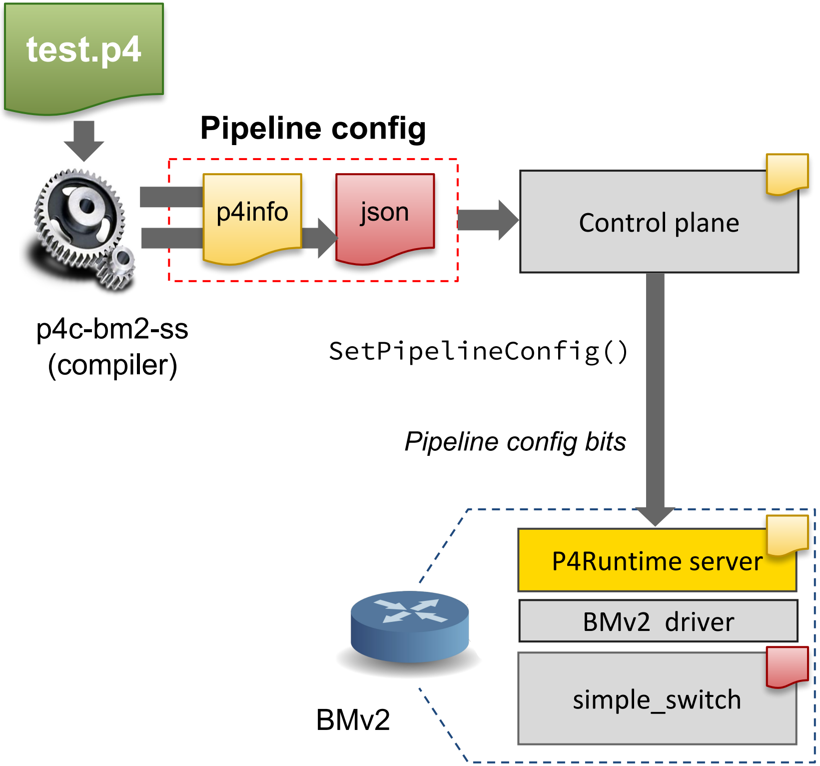
```
table_entry {
  table_id: 33581985
  match {
    field_id: 1
    lpm {
      value: "\f\000\...
      prefix_len: 8
    }
  }
  action {
    action_id: 16786453
    params {
      param_id: 1
      value: "\000\0...
    }
    params {
      param_id: 2
      value: 7
    }
  }
}
```

The P4 Runtime API can be used equally well by a remote or local control plane

Set Pipeline Config

p4runtime.proto simplified excerpt

```
message ForwardingPipelineConfig {
  P4Info p4info;
  /* Target-specific P4 configuration.
     e.g JSON bits for BMv2 */
  bytes p4_device_config;
  ...
}
```



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 P4 Runtime scope
  - Achieved with other mechanisms
    - e.g., OpenConfig and gNMI

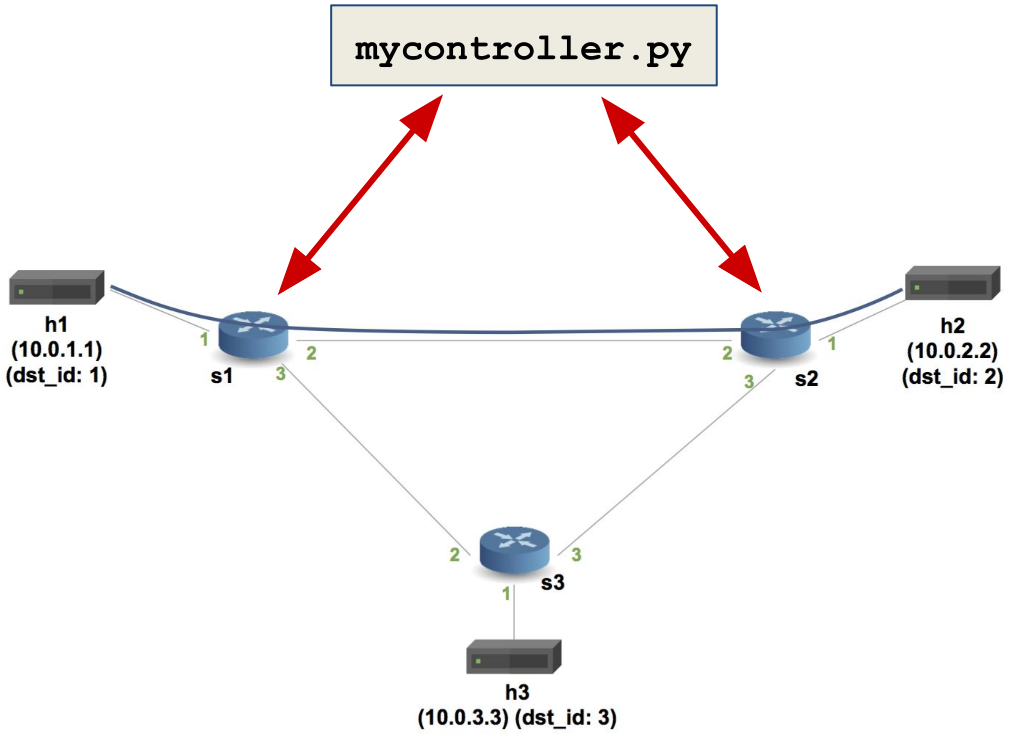
Work-in-progress by the p4.org API WG
Expect API changes in the future

P4 Runtime 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/P4D2_2018_East/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.

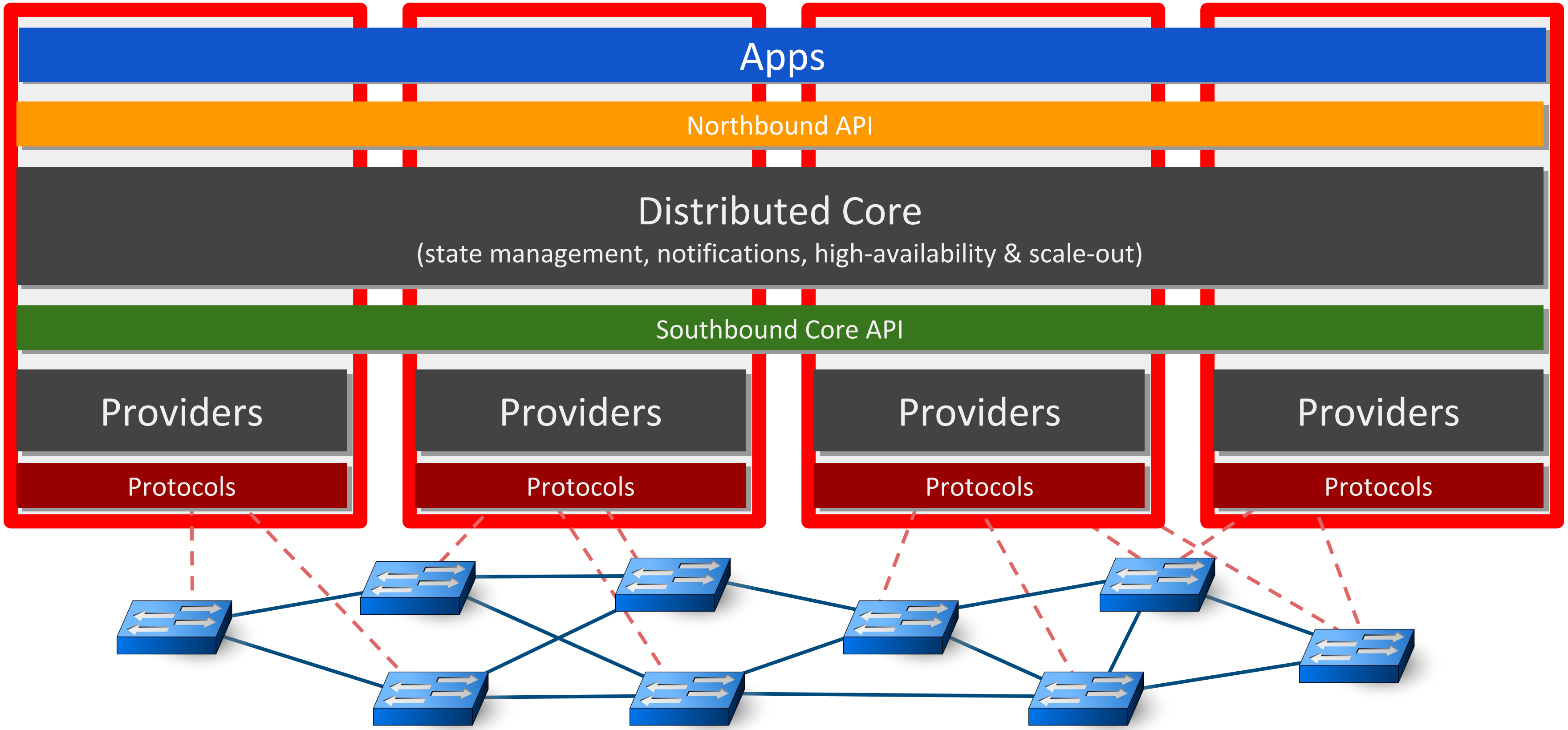
1. In your shell, run:

```
make
```

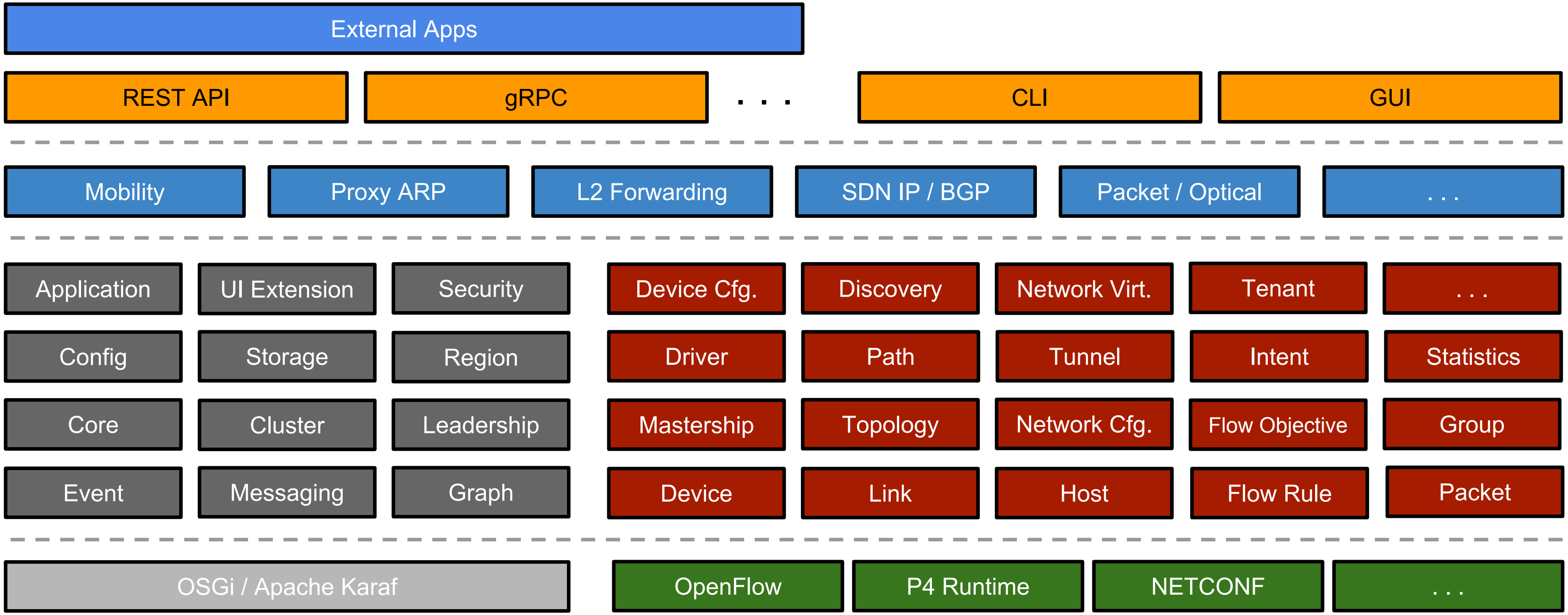
Example: ONOS

Open Network Operating System (ONOS) is an open source SDN network operating system, originally created by ON.Lab and currently hosted by the Linux Foundation.

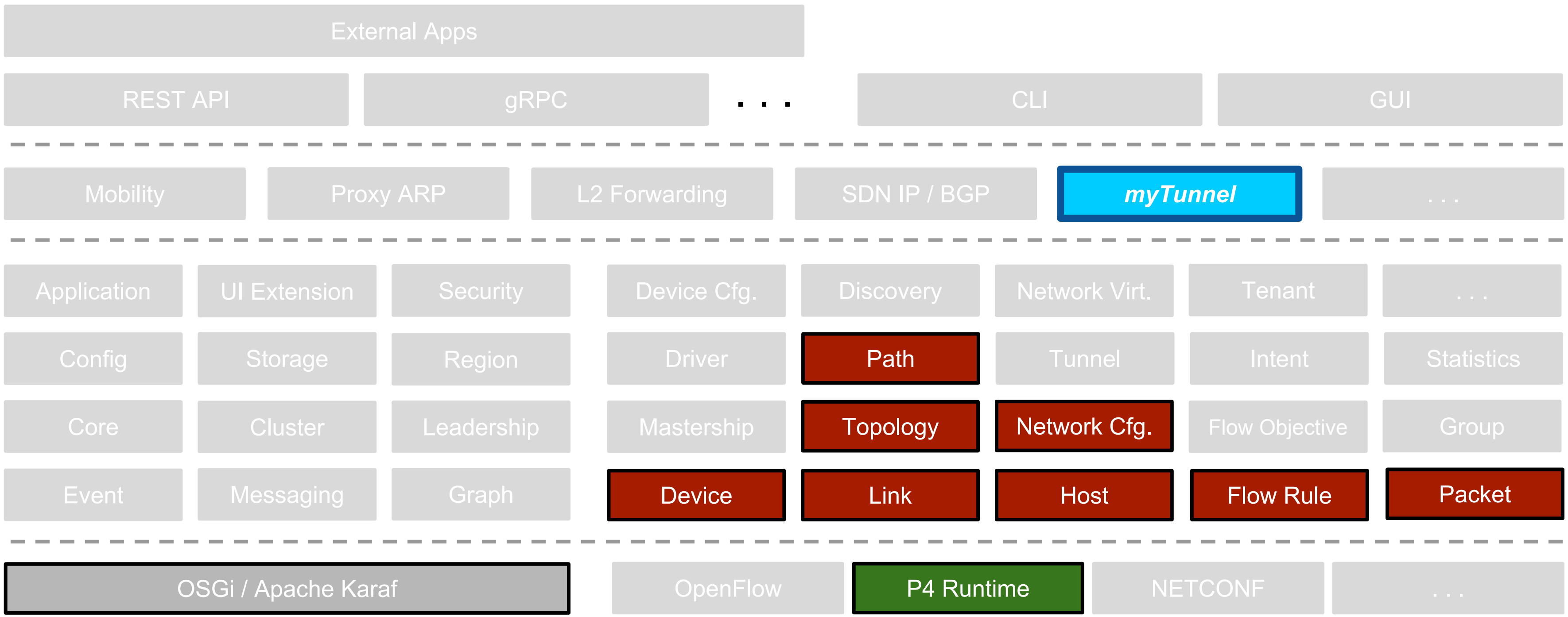
ONOS – SDN Controller



ONOS Core Subsystems



ONOS Core Subsystems



P4 Runtime takeaways

- **Program-independent API**
  - API doesn't change with the P4 program
  - No need to restart the control-plane with a different P4 program
- **Device does not need to be fully programmable**
  - Can be used on fixed-function devices
  - Provided that their behavior can be expressed in P4
- **Protobuf-based format**
  - Well-supported serialization format in many languages
  - Supported by many RPC frameworks, e.g., gRPC
    - Auto-generate client/server code for different languages
    - No need to define common RPC features (e.g., authentication)

P4 support in ONOS

PD = protocol-dependent
PI = protocol-independent