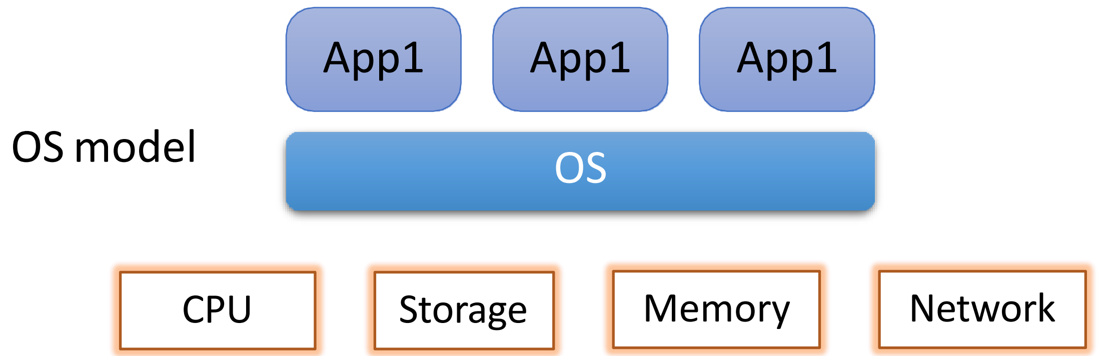
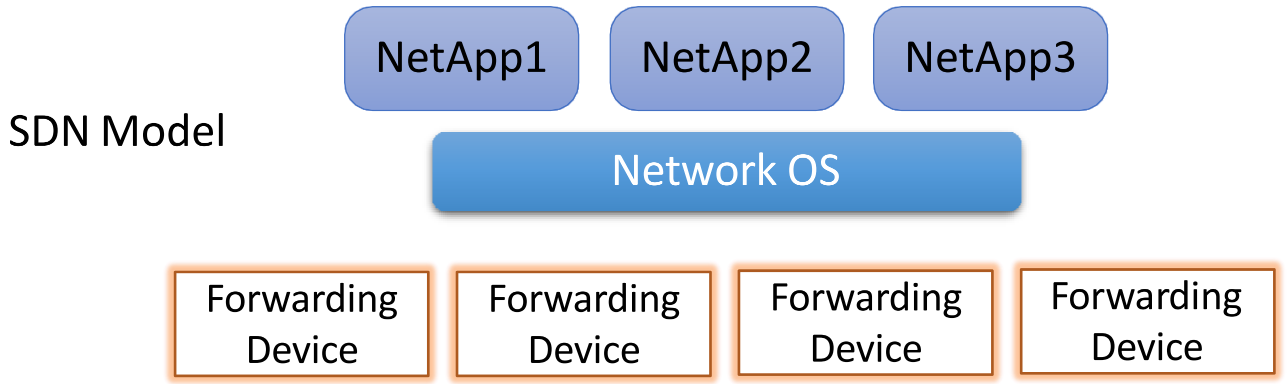
Lab 4
NETWORKING WITH RASPBERRY PI

- P4 programming introduction

Software-Defined Networks – Analogy to OS Model

- Operating System model in 3 layers
  - Provide flexibility to developers
  - OS manages the hardware resources such as CPU
  - OS makes it flexible for application developers to add/remove/customize applications without being tied to the underlying hardware
- Operating System(OS) -> Network OS (Also called SDN controller)
- SDN model
  - SDN Controller (NOS) interfaces to Network nodes (forwarding device) and provides a programmable interface to network applications
  - Forwarding receive packets, and does action according to set of rules defined in a table called Match-Action table
  - SDN Controller (NOS) populates these rules
  - Network developers can now develop Network Applications (NetApp) to make the rules





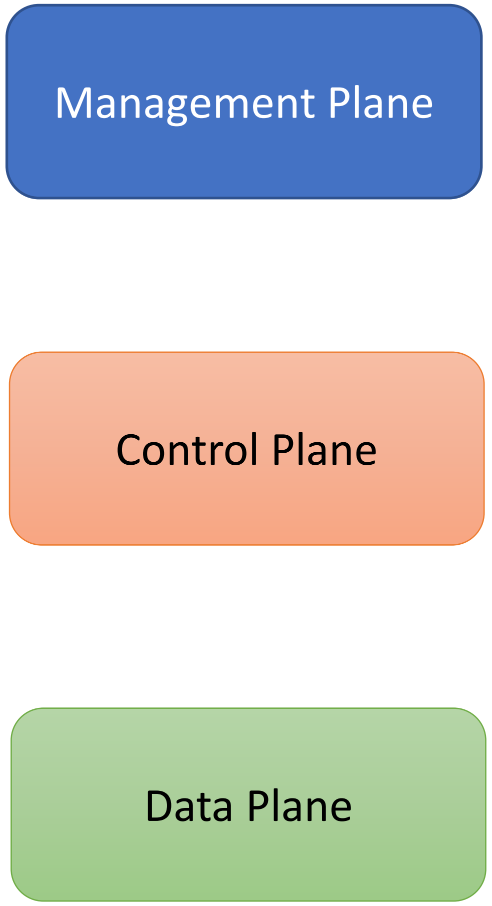
Networking architecture hierarchy

Interface (e.g. GUI) to control/monitor devices, configure Control Plane

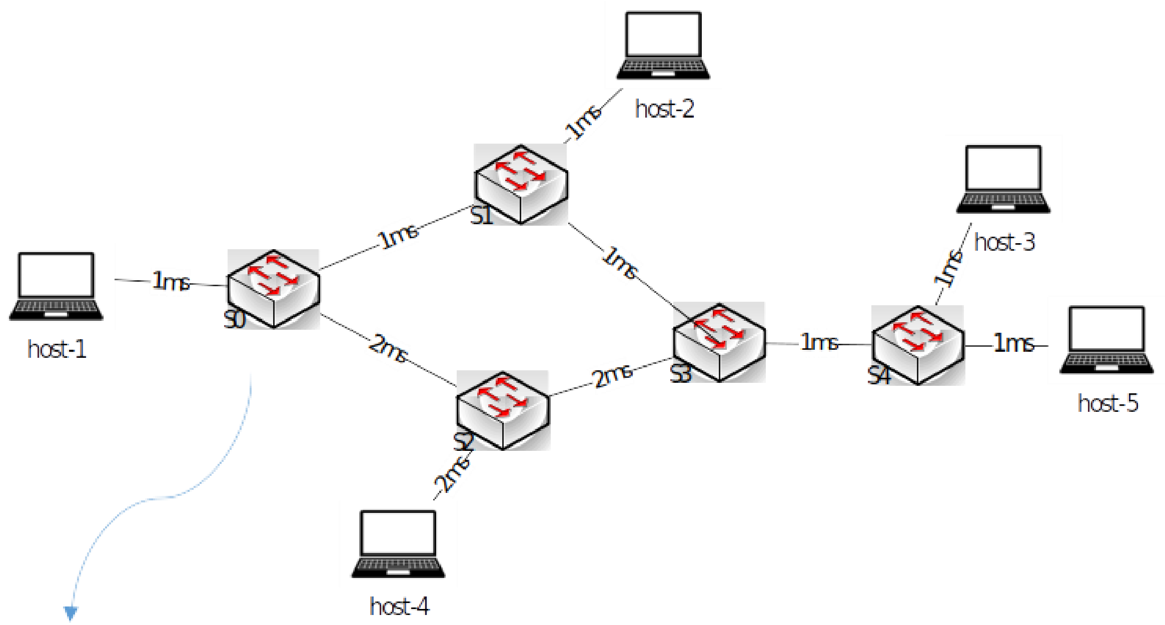
Functions to populate forwarding/routing tables and enable data plane functions

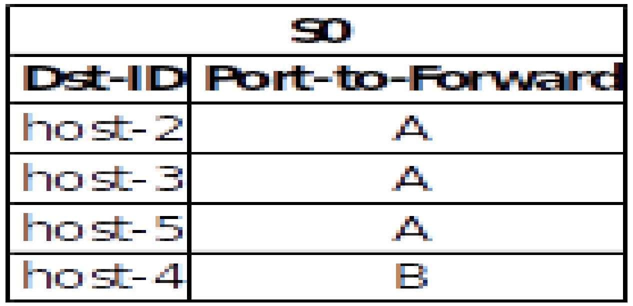
Code for handling switching/routing functions

- Very optimized assembly-language code for fast hardware (ASICs)



Traditional Networks: Switch/Router



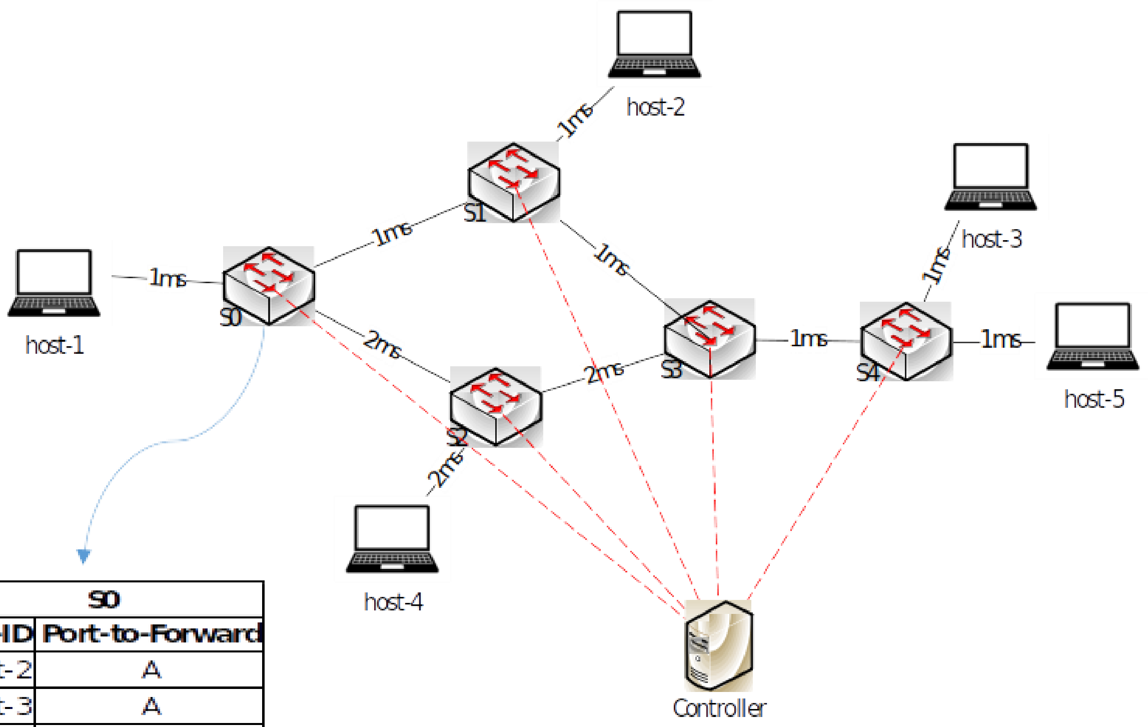


S0	
Dst-ID	Port-to-Forward
host-2	A
host-3	A
host-5	A
host-4	B

- In traditional networks switches/routers, perform two functions
  - Forwarding: It forwards the packets it receives. Switches looks up the forwarding/routing table to decide to which port it needs to route an incoming packet
  - Routing: Decides where to route the packets. i.e., populating the forwarding/routing table. Switches/routers runs decentralized algorithms to generate these tables.

Drawback – A network engineer has no control over which path the packets move

Software Defined Networks (SDN): Basics

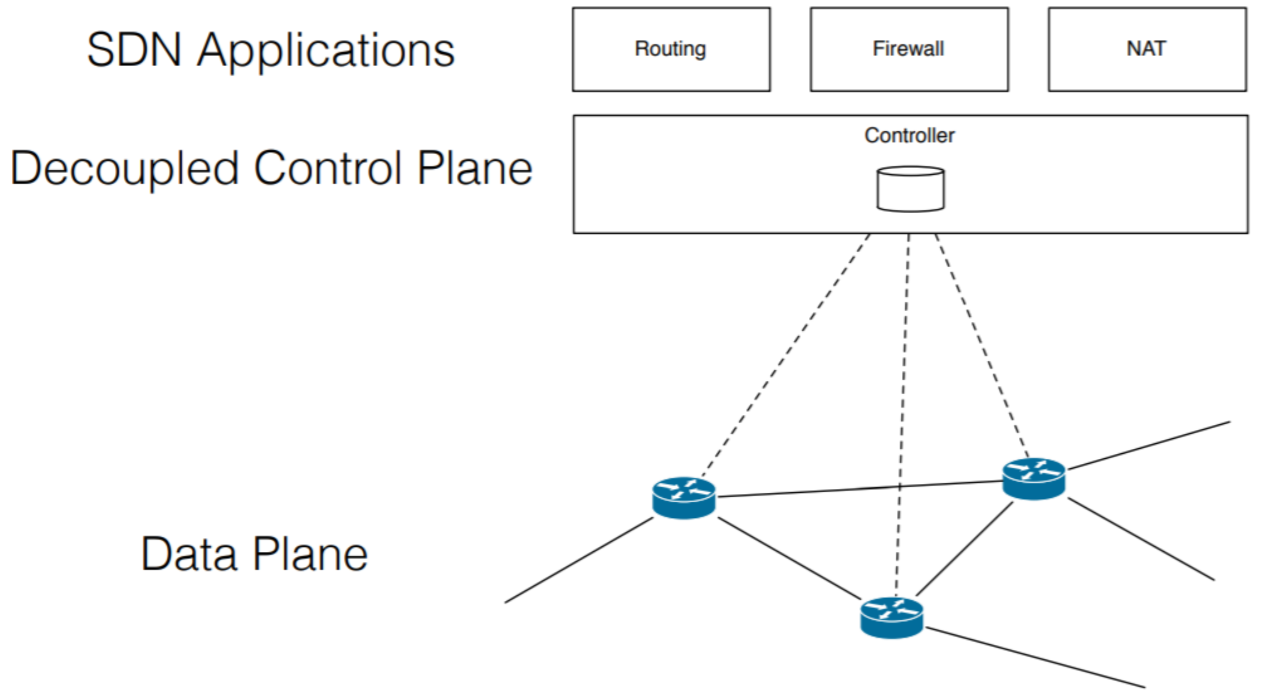


S0	
Dst-ID	Port-to-Forward
host-2	A
host-3	A
host-5	A
host-4	B

- Switches perform only one function
    - Forwarding: It forwards the packets it receives. Switches looks up the forwarding/routing table to decide to which port it needs to route an incoming packet

- Routing:
    - SDN controller generates routing tables for switches
    - Controller is connected to all switches in the network

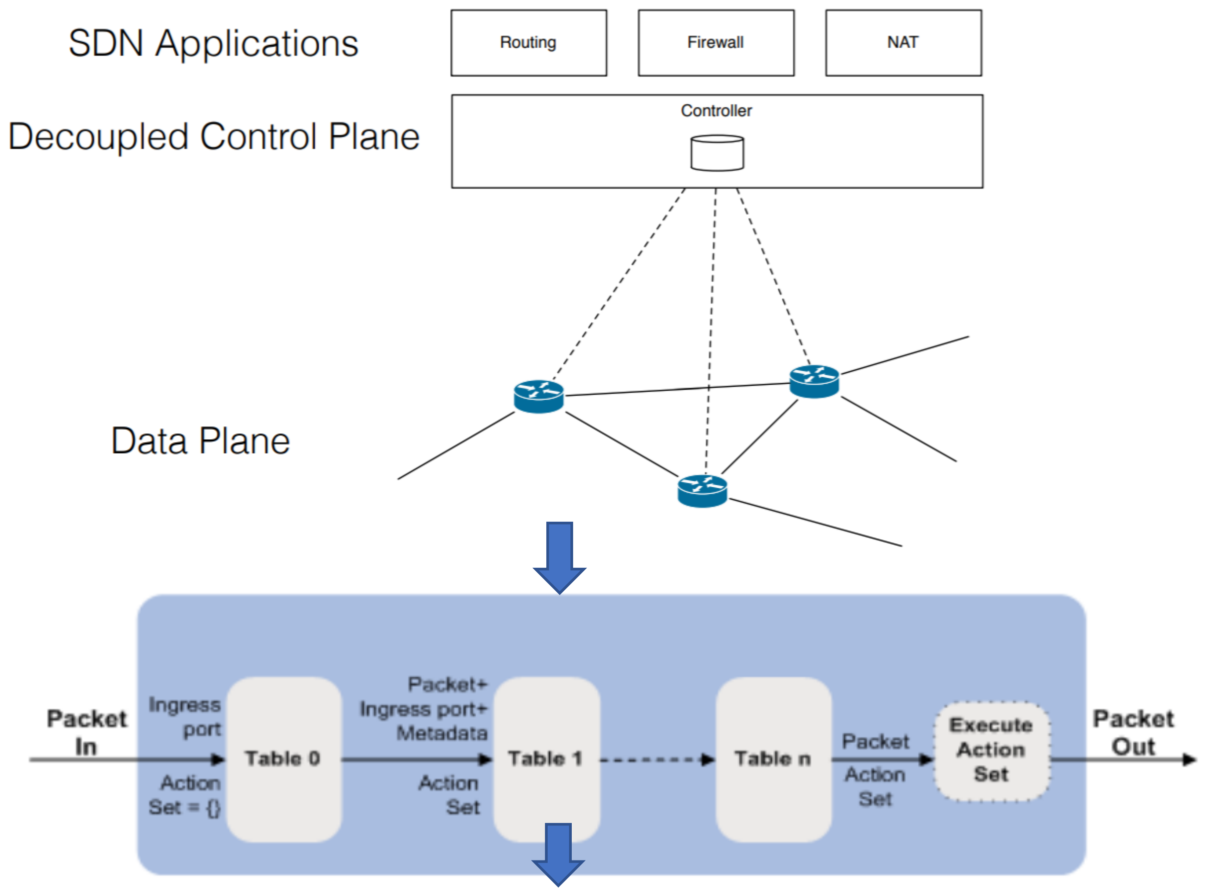
Advantage – Network engineer has a central control over the network



- Traditionally, planes were a logical construct

- SDN goals:
    - physically separates the control plane from data plane

SDN: Match-Action Table



- Match-Action table
  - A forwarding table
- Each switch can have more than one match-actions
- Each table can pick different **match-field**
- Each row of a table can pick different **actions**

Packet In ➡

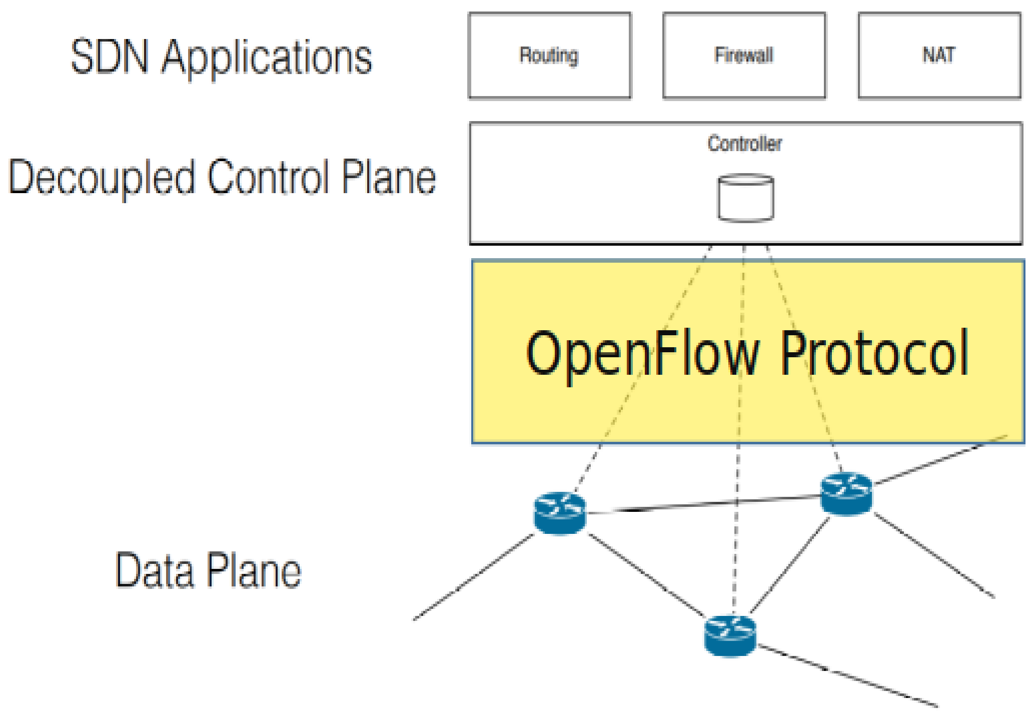
Match-Field=src-IP	Action	Action-Data
w.x.y.z1	Drop	
w.x.y.z2	Output	*Port-2*
w.x.y.z3	set TTL	*Value to Set*

… ➡

Match-Field=dst-IP	Action	Action-Data
w1.x.y.z1	Drop	
w1.x.y.z3	Output	*Port-2*

➡ Packet Out

SDN: Populating Match-Action Table

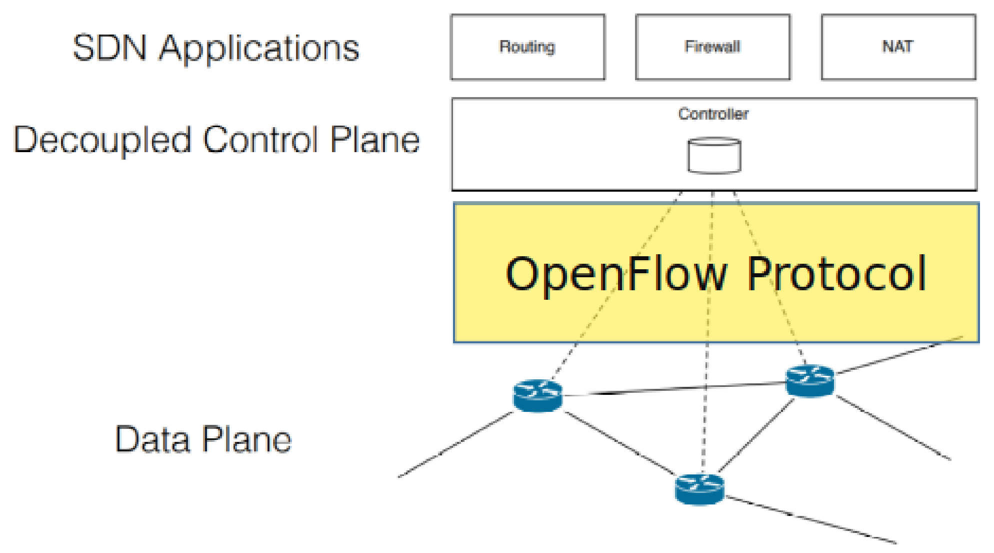


- Switch requests Controller to populate match-action tables
  - E.g. If a packet's field doesn't match any Match field value in the table, switch requests controller to update the table to add a match-action
- Controller than populates the switch's match-action table

- OpenFlow – An open-source standard protocol for controller-switch communication
- Other vendor-specific protocols exist too

- OpenFlow
  - Advantage – Interoperability between switches and controllers from different vendors
  - Disadvantage – Match-Action tables are fixed

SDN: Populating Match-Action tables

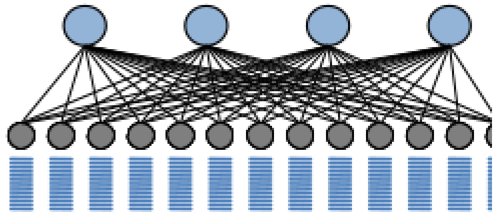


If an OpenFlow device does not support some match-action flow?

- Step 1: Update OpenFlow function
- Step-2: Redesign the controller for this new function
- Step-3: Redesign the switch for this new function(design of ASIC takes years)
- Time Consuming. What’s the Solution ? **P4**

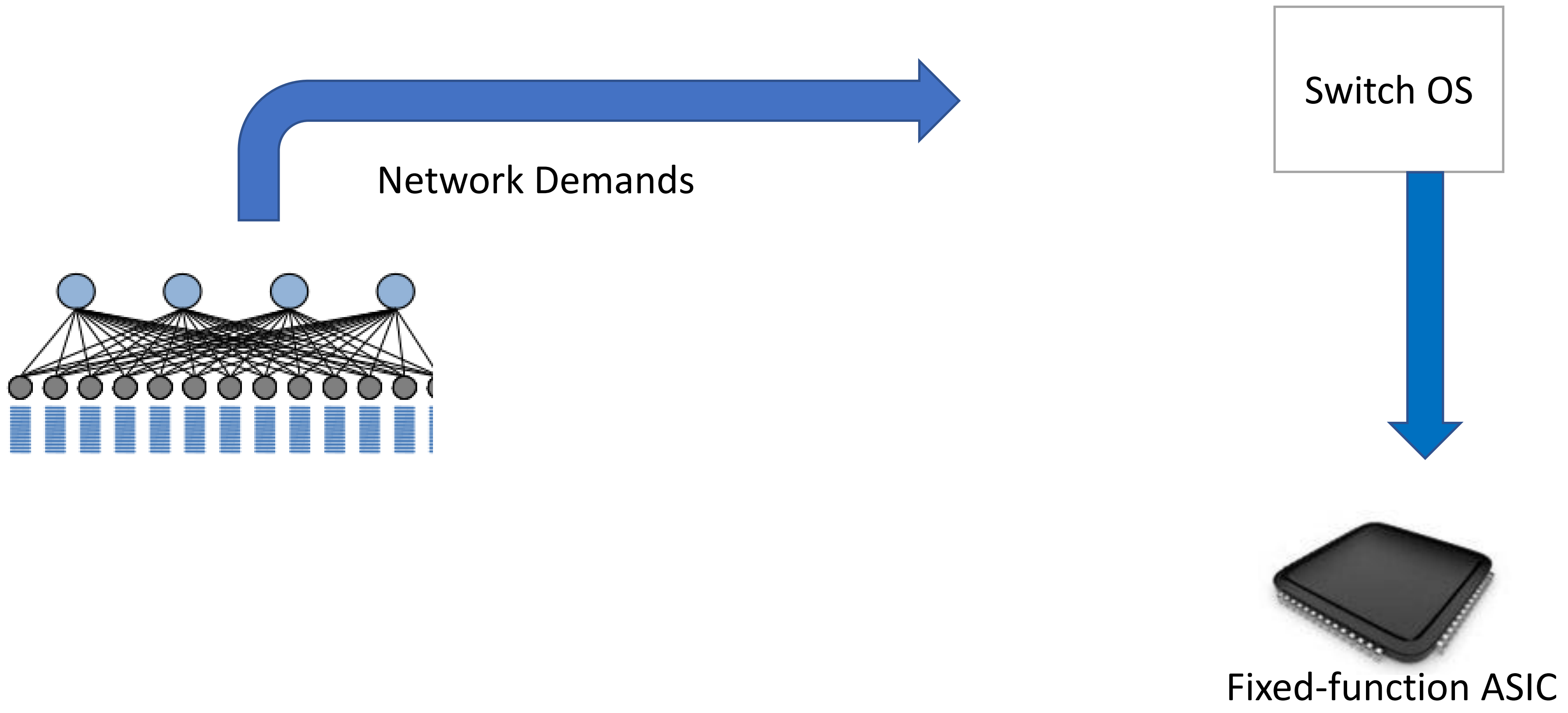
Status quo: Bottom-up design

Network Demands

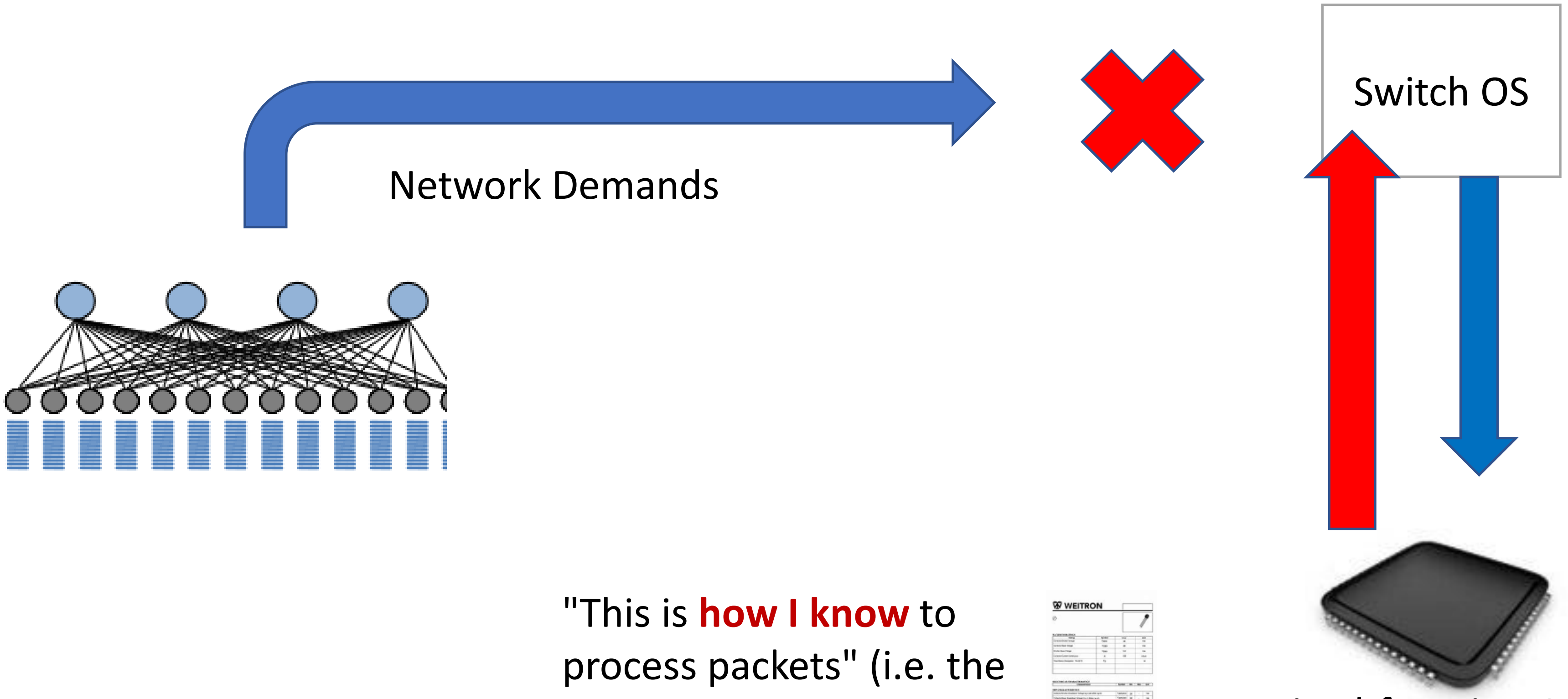


Switch OS

Status quo: Bottom-up design



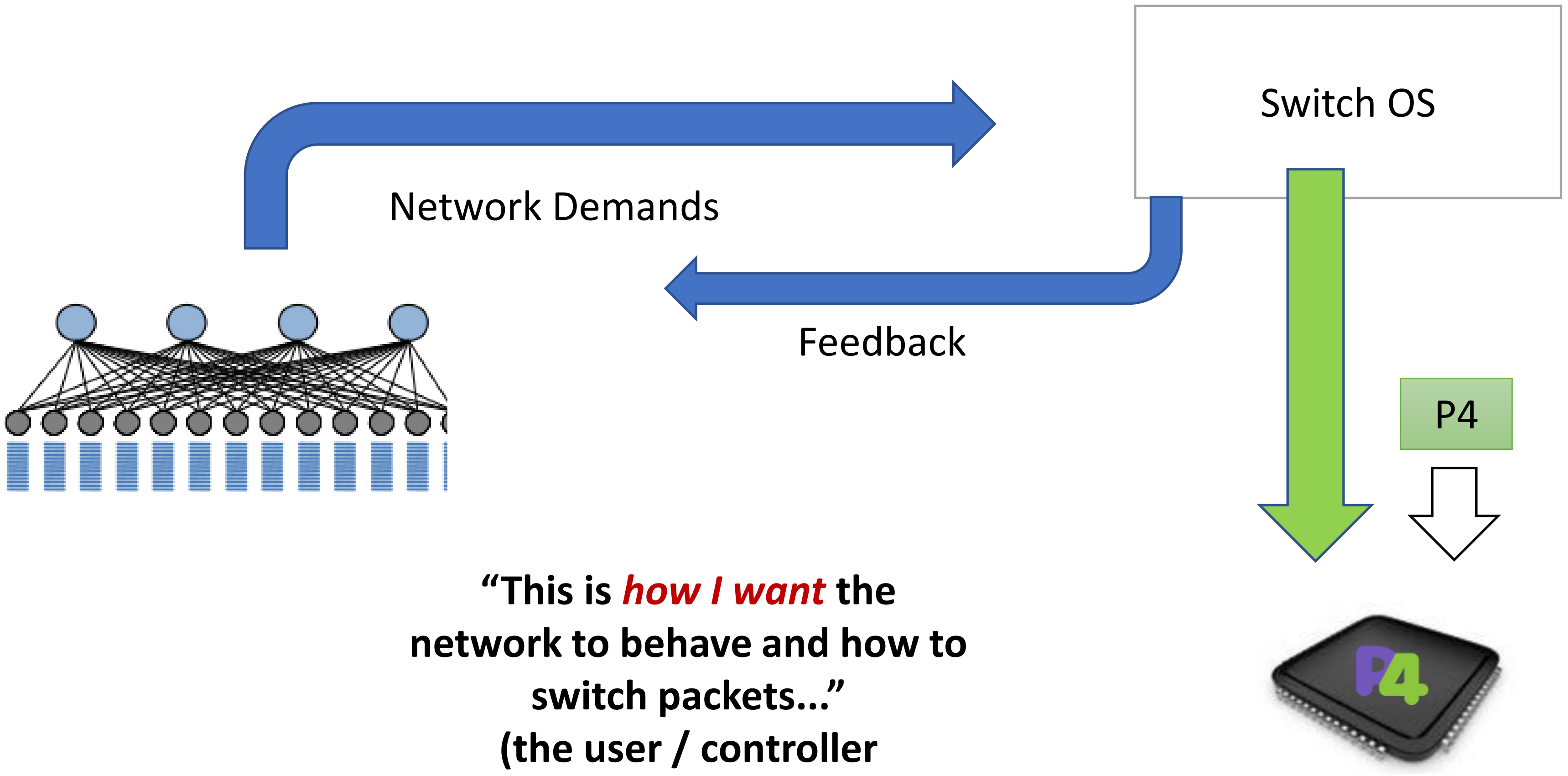
Status quo: Bottom-up design



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

Vendors rolled out networking hardware with fixed functions (Bottoms-up approach)

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

- Trend has been to decouple virtual network from the physical network

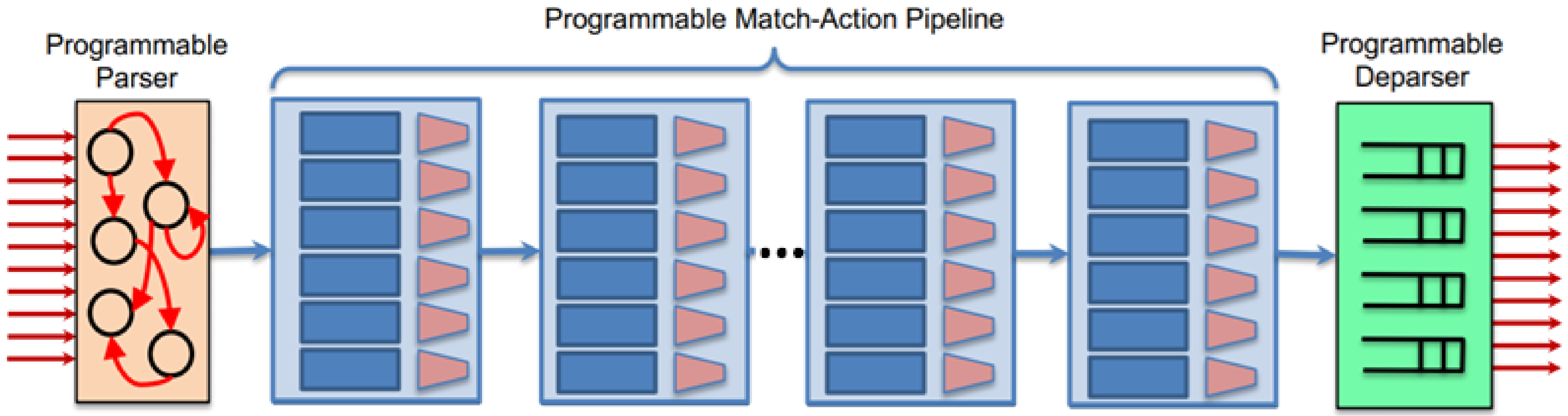
- Top-down design approach
    - Application developers and network engineers use programming language to program a reconfigurable hardware to implement and test new functions quickly (minutes vs months/years)

Introduction to P4

- P4 is language
  - Used to program **P**rogramming **P**rotocol-Independent **P**acket **P**rocessors
  - Also called P4Switch – A switch hardware that can be programmed using P4 language
- P4-Runtime - The controller-P4Switch communication protocol
- Open-Source
- High-level Domain-Specific language – to program network devices
- Enables data plane programming - Specifies how data plane devices process packets
- Data-plane devices – Switches, Routers, NICs

P4-Switch

- P4-Switch Architecture
  - Different Architectures are Proposed for P4-Switch
  - PISA: Protocol-Independent Switch Architecture is one variant

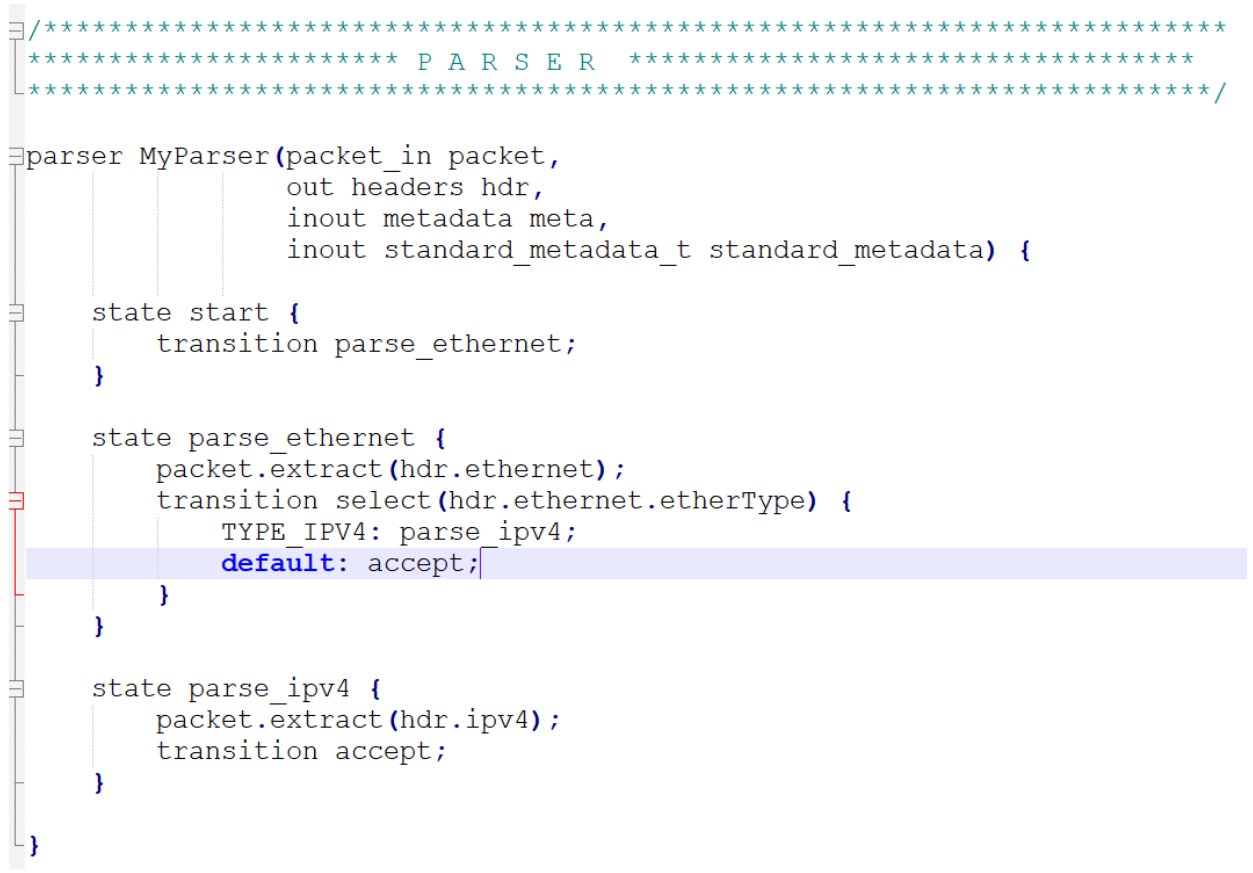


P4 Parser

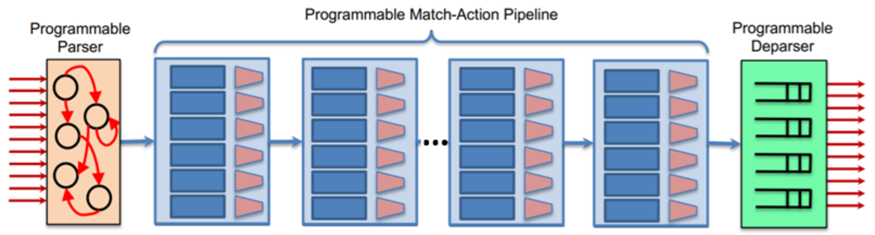
- Parser Block
  - Reads a packet and extracts packet headers
    - E.g. Source and Destination MAC address, Source and Destination IP, Checksum
    - These headers are used as Match Fields in a Match-Action table

- P4 Programmable Parser ?
  - Allows programmers to define headers and their locations in the packet using P4
  - Allows programmers to define the headers to be extracted during run-time

P4 Parser Code

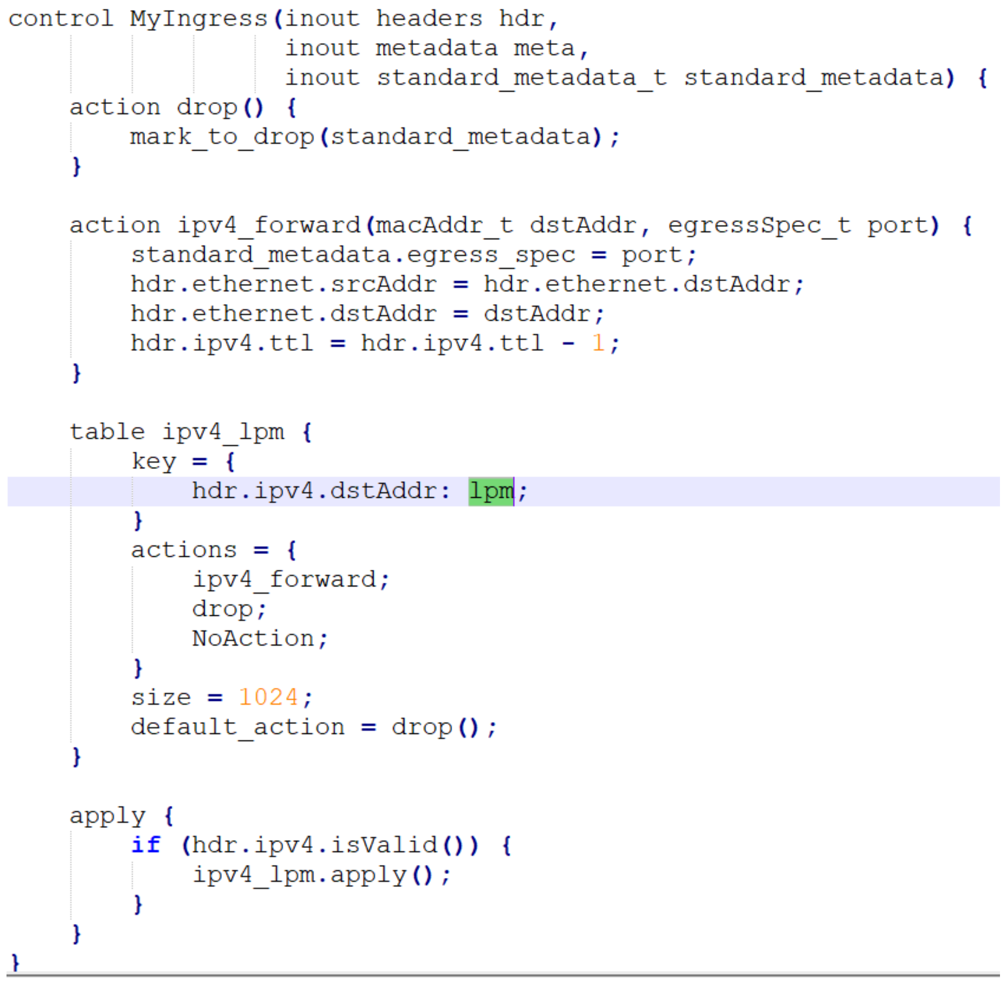


```
/**********************************************************************
********************** P A R S E R  *********************************
*********************************************************************/

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

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }

}
```

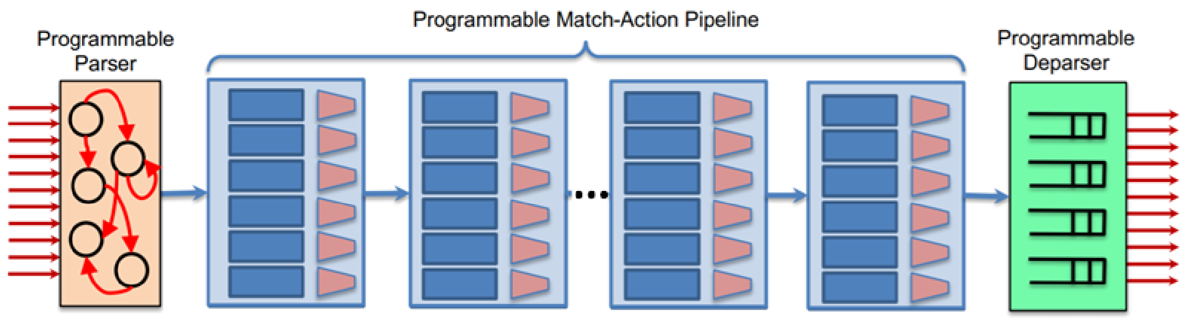


- Extract headers
- Decide whether to accept or drop packets
    - Accepted packets are forwarded to the match-action pipeline

P4 Match Actions



```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action drop() {
        mark_to_drop(standard_metadata);
    }

    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

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

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}
```



What does match-actions do ?

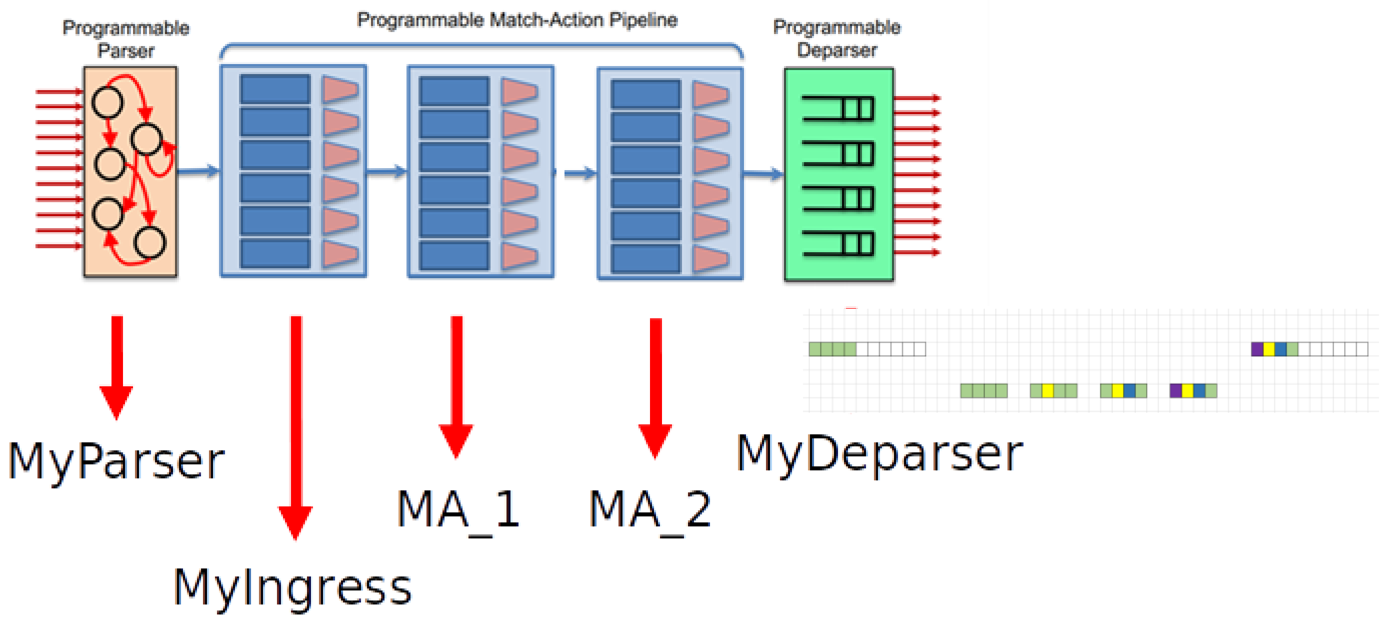
- It matches the headers of a packet with a match field
- If the match is success, it applies an action
    - An action is a procedure that updates the packet headers
    - An action can also drop the packet

What does a deparser do?

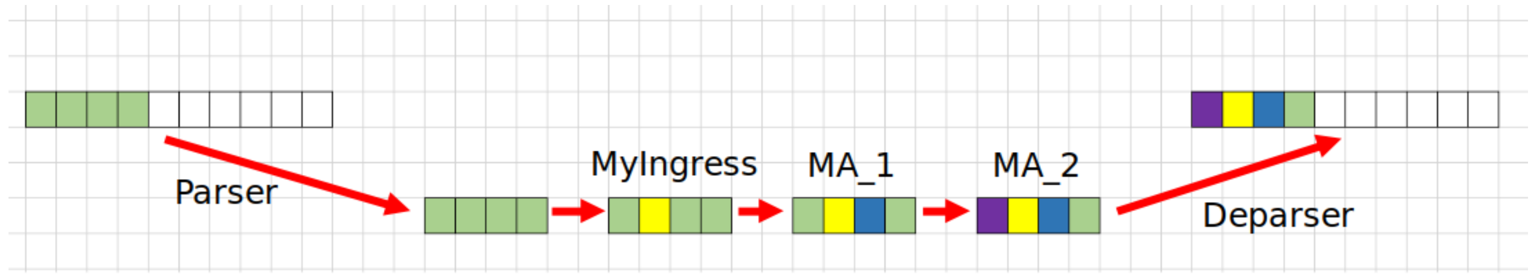
- It replaces the header of the original packet with the modified header in the order you want and transmits the packet out

Match-Field=dstAddr	**Action**	**Action-Data**
w1.x.y.z1	Drop	
w1.x.y.z3	ipv4_forward	*dstAddrX, portX*
w1.x.y.z4	ipv4_forward	*dstAddrY, portY*

Linking the Blocks Together

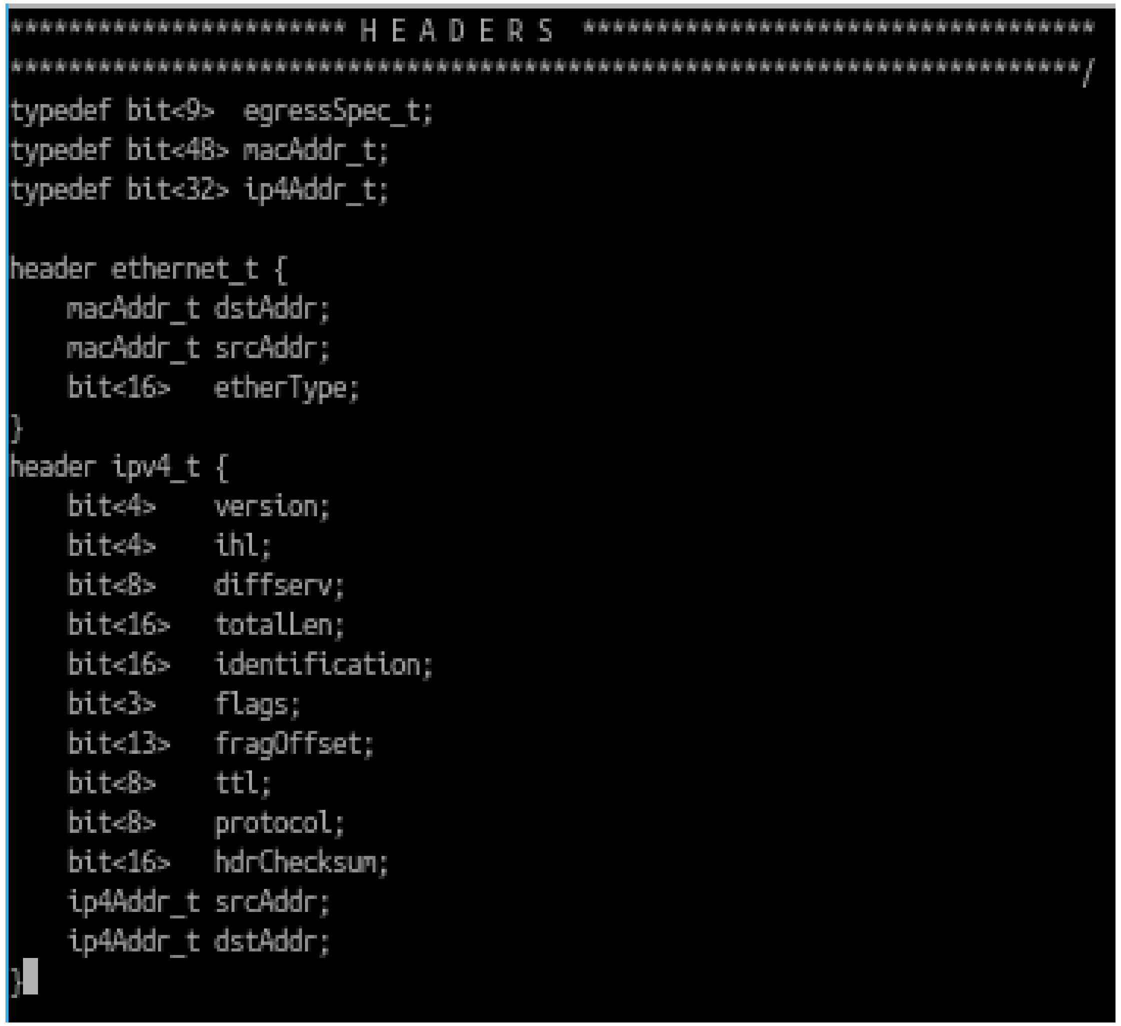


```
V1Switch
(
MyParser(),
MyIngress(),
MA_1(),
MA_2(),
MyDeparser()
) main;
```

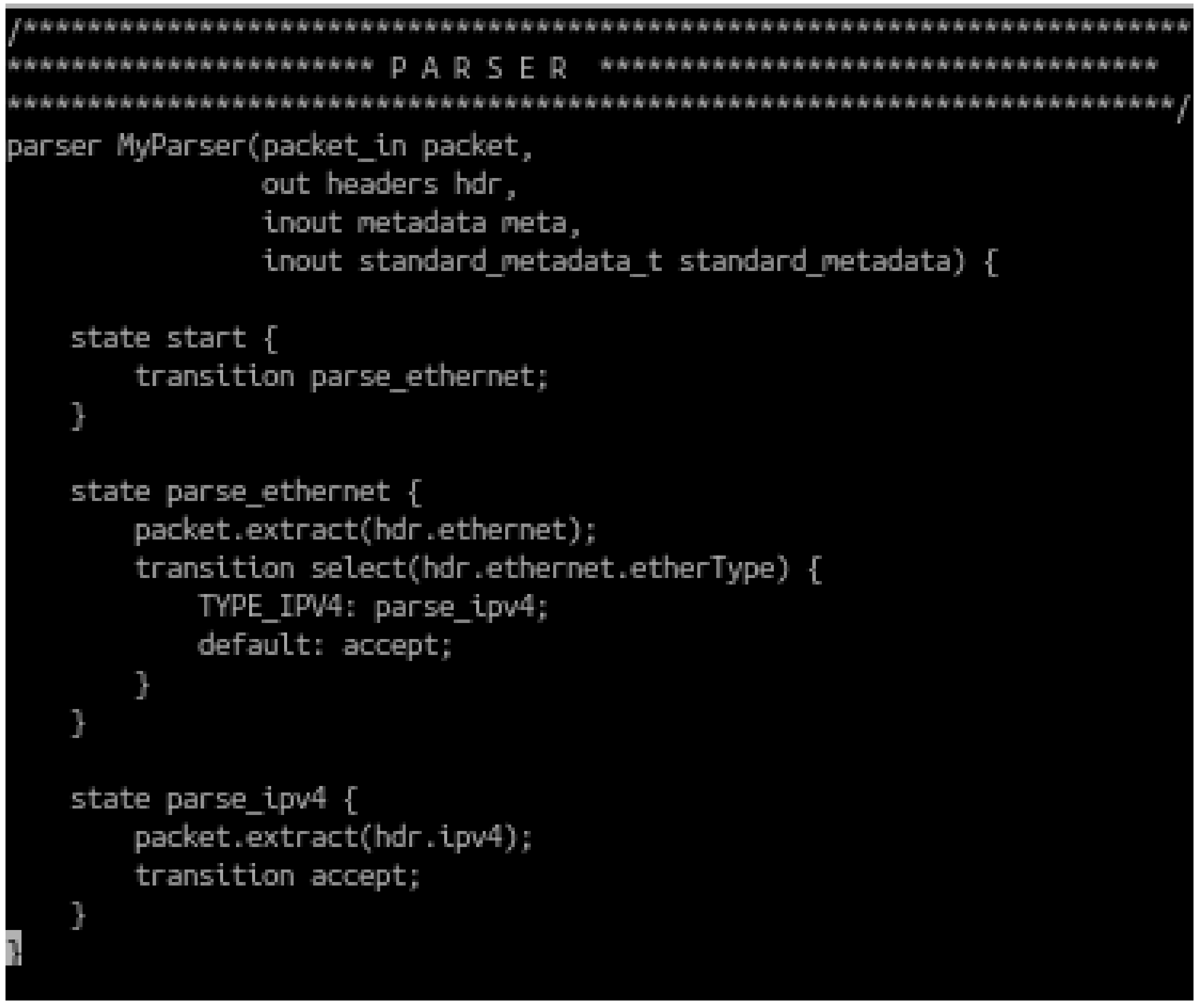


- P4 on Mininet to create a realistic virtual network, running real kernel, switch and application code on a single machine.
- We can use it to create a custom topology, having Hosts and Switches.
- Topology can be defined as a JSON file.
- The P4 code for the Switch can be run and tested. All switches run the same P4 code.
- The Match-Action Tables for different switches can be written as separate JSON files.
- We can see the terminal of different Hosts in the same virtual environment.
- Packets can be crafted using Scapy (in a Python script) and can be seen at the destination hosts.

P4 Data Plane

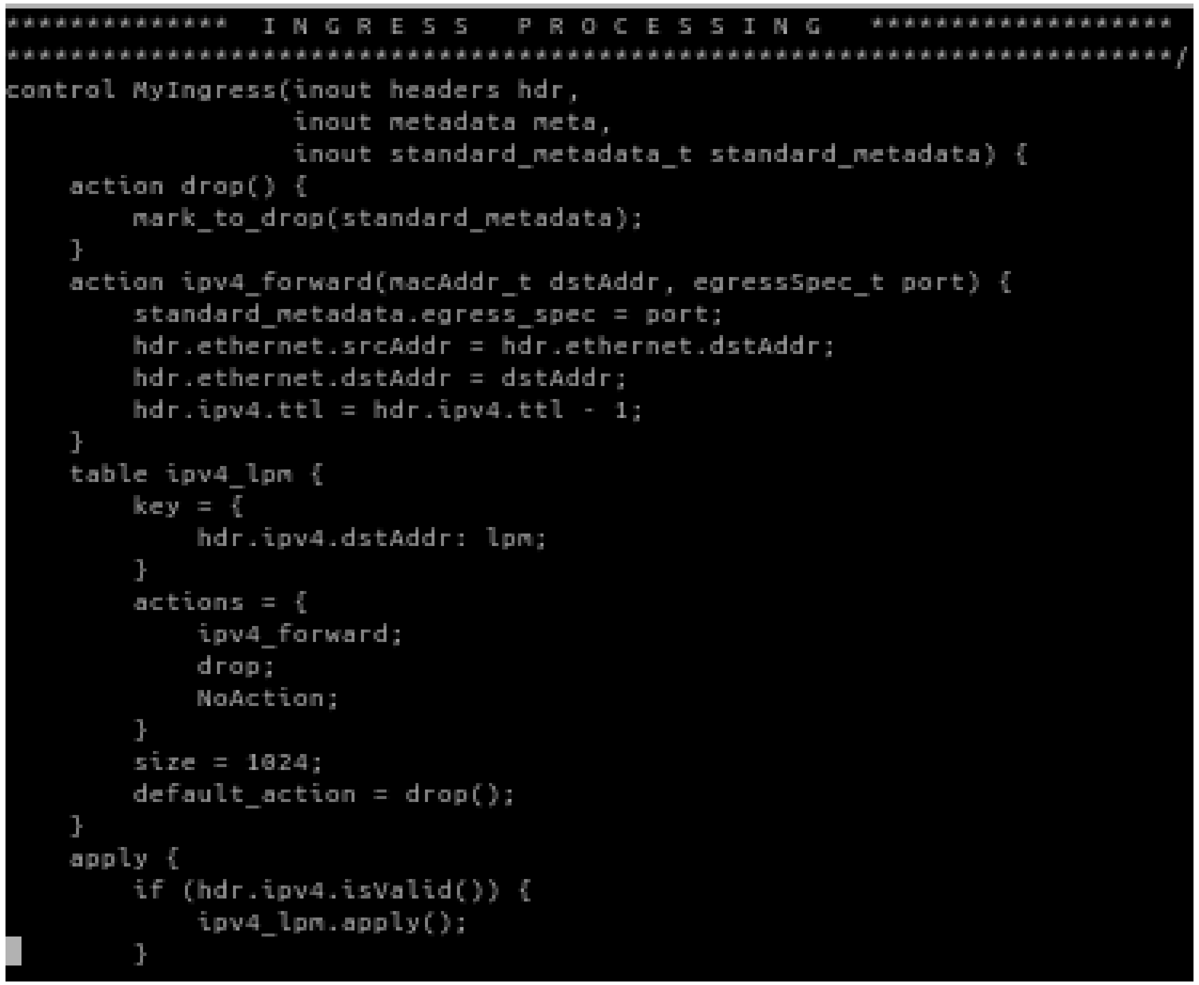


```
*********************** H E A D E R S ***********************************
*************************************************************************/
typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

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;
}
```

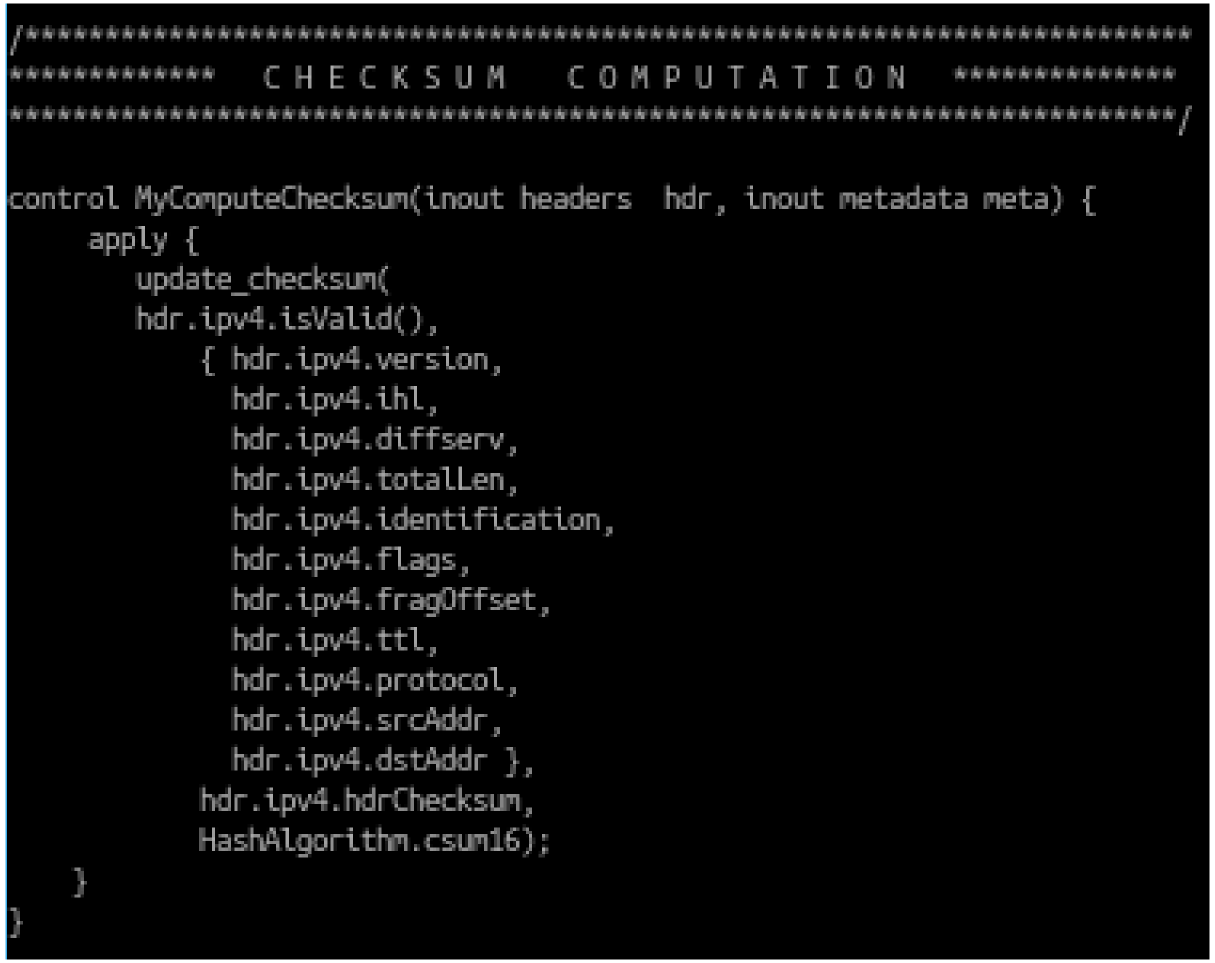


```
/*************************************************************************
*********************** P A R S E R ***********************************
*************************************************************************/
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
```

P4 Data Plane



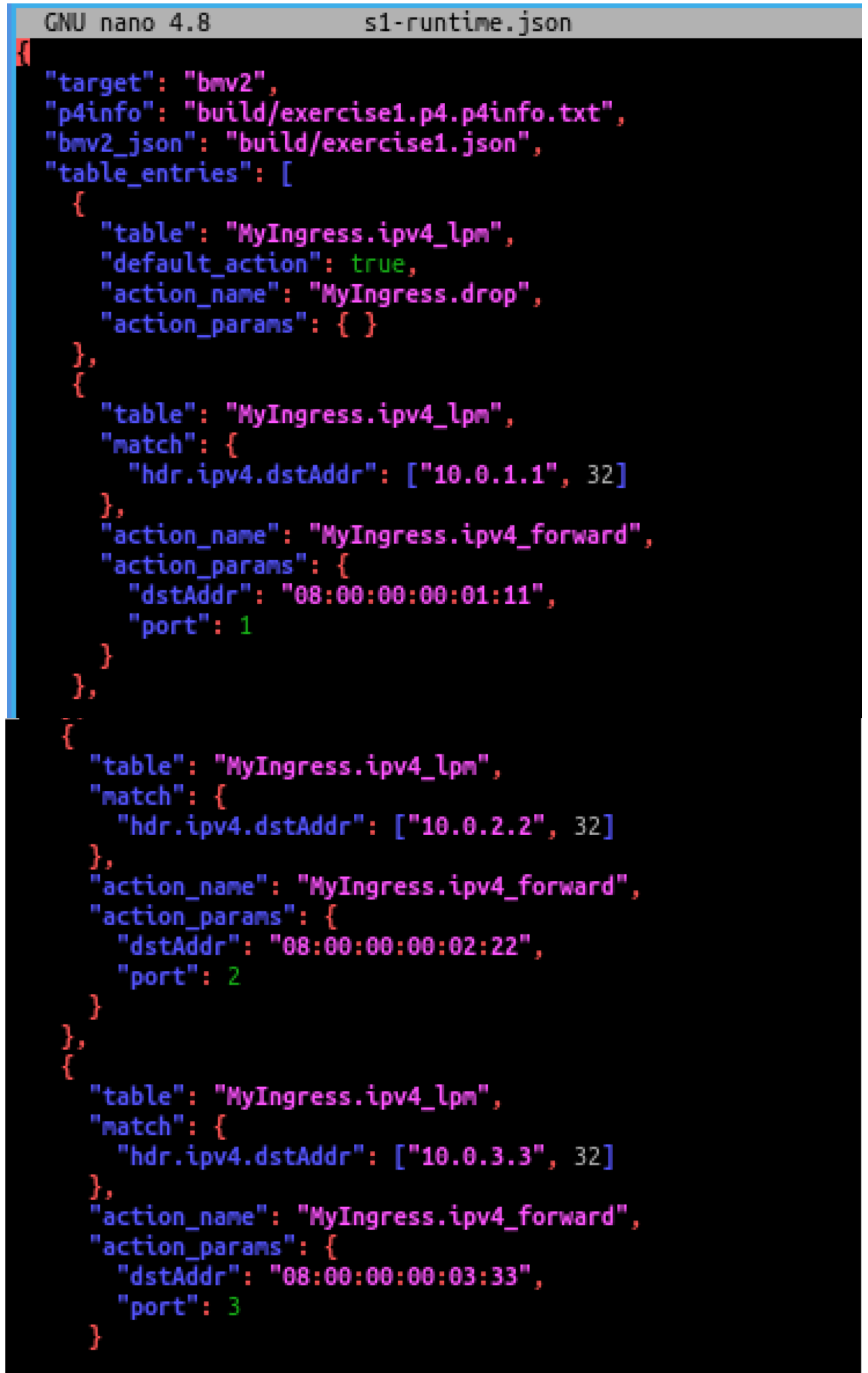
```
**************  I N G R E S S   P R O C E S S I N G   ******************
*********************************************************************/
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action drop() {
        mark_to_drop(standard_metadata);
    }
    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }
    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }
    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
```



```
/*************************************************************************
*************   C H E C K S U M    C O M P U T A T I O N   **************
*************************************************************************/

control MyComputeChecksum(inout headers  hdr, inout metadata meta) {
     apply {
        update_checksum(
        hdr.ipv4.isValid(),
            { hdr.ipv4.version,
              hdr.ipv4.ihl,
              hdr.ipv4.diffserv,
              hdr.ipv4.totalLen,
              hdr.ipv4.identification,
              hdr.ipv4.flags,
              hdr.ipv4.fragOffset,
              hdr.ipv4.ttl,
              hdr.ipv4.protocol,
              hdr.ipv4.srcAddr,
              hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
}
```

P4 Control Plane

- JSON files are used to define the runtime behaviour of Programmable switches.
- Even though all switches run the same P4 program in a Mininet environment, the JSON files control the Match-Action output and hence can control the flow.
- Topology for the Virtual environment (Mininet) is also defined using a JSON file. In the real world, topologies are defined by actual physical connections.



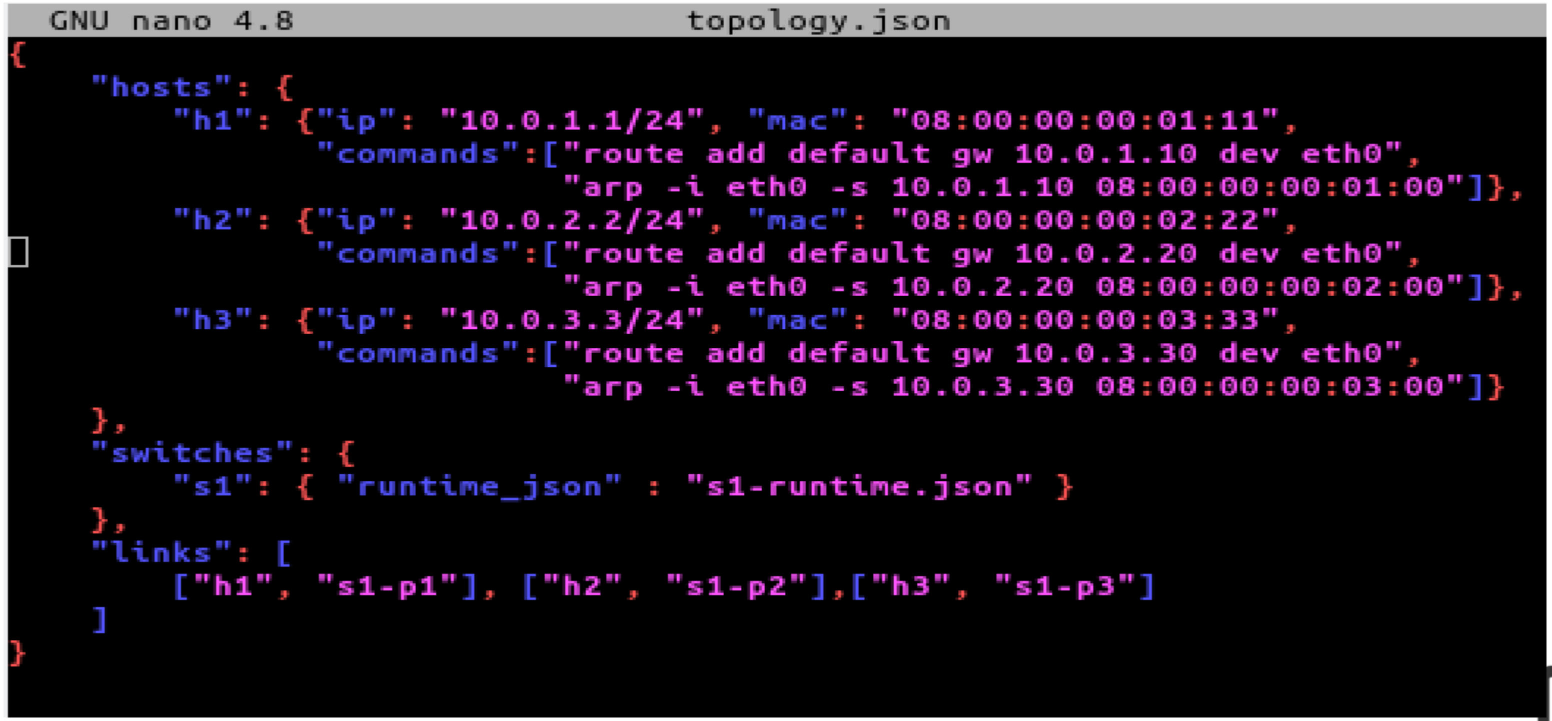
```
  GNU nano 4.8                  s1-runtime.json
{
 "target": "bmv2",
 "p4info": "build/exercise1.p4.p4info.txt",
 "bmv2_json": "build/exercise1.json",
 "table_entries": [
  {
    "table": "MyIngress.ipv4_lpm",
    "default_action": true,
    "action_name": "MyIngress.drop",
    "action_params": { }
  },
  {
    "table": "MyIngress.ipv4_lpm",
    "match": {
      "hdr.ipv4.dstAddr": ["10.0.1.1", 32]
    },
    "action_name": "MyIngress.ipv4_forward",
    "action_params": {
      "dstAddr": "08:00:00:00:01:11",
      "port": 1
    }
  },
  {
    "table": "MyIngress.ipv4_lpm",
    "match": {
      "hdr.ipv4.dstAddr": ["10.0.2.2", 32]
    },
    "action_name": "MyIngress.ipv4_forward",
    "action_params": {
      "dstAddr": "08:00:00:00:02:22",
      "port": 2
    }
  },
  {
    "table": "MyIngress.ipv4_lpm",
    "match": {
      "hdr.ipv4.dstAddr": ["10.0.3.3", 32]
    },
    "action_name": "MyIngress.ipv4_forward",
    "action_params": {
      "dstAddr": "08:00:00:00:03:33",
      "port": 3
    }
```

Tutorial 1: 3 hosts and 1 ethernet switch

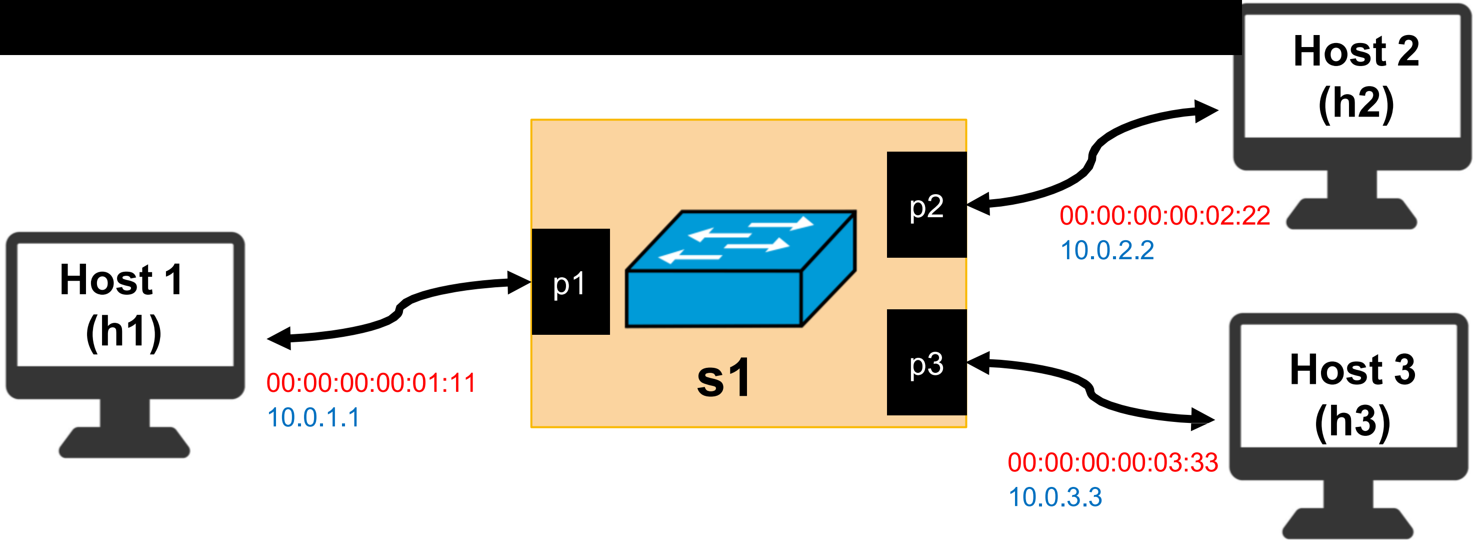
- In this Tutorial, we will create an Ethernet switch s1 connected to 3 hosts h1, h2 and h3
- P4 program that implements basic forwarding, for IPv4
- With IPv4 forwarding, the switch performs the following actions for every packet:
  - (i) update the source and destination MAC addresses
  - (ii) decrement the time-to-live (TTL) in the IP header
  - (iii) forward the packet out the appropriate port.

  It has to match the Destination MAC address present in the packet with some pre-learned MAC addresses.
- Switch will have a single table, which the control plane will populate with static rules. Each rule will map an IP address to the MAC address and output port for the next hop.

Tutorial 1: 3 hosts and 1 ethernet switch



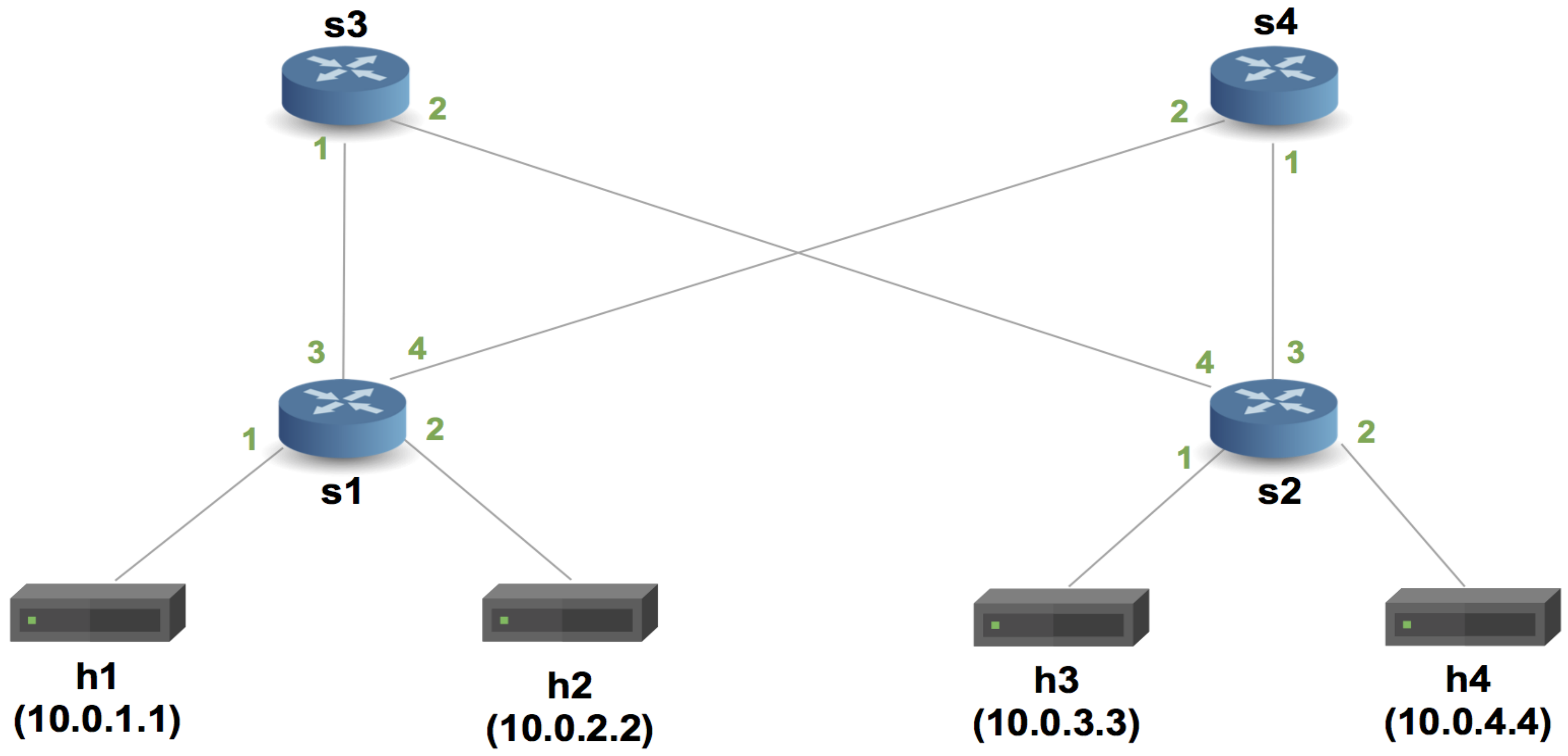
```
  GNU nano 4.8                          topology.json
{
    "hosts": {
        "h1": {"ip": "10.0.1.1/24", "mac": "08:00:00:00:01:11",
               "commands":["route add default gw 10.0.1.10 dev eth0",
                           "arp -i eth0 -s 10.0.1.10 08:00:00:00:01:00"]},
        "h2": {"ip": "10.0.2.2/24", "mac": "08:00:00:00:02:22",
               "commands":["route add default gw 10.0.2.20 dev eth0",
                           "arp -i eth0 -s 10.0.2.20 08:00:00:00:02:00"]},
        "h3": {"ip": "10.0.3.3/24", "mac": "08:00:00:00:03:33",
               "commands":["route add default gw 10.0.3.30 dev eth0",
                           "arp -i eth0 -s 10.0.3.30 08:00:00:00:03:00"]}
    },
    "switches": {
        "s1": { "runtime_json" : "s1-runtime.json" }
    },
    "links": [
        ["h1", "s1-p1"], ["h2", "s1-p2"],["h3", "s1-p3"]
    ]
}
```



Running Tutorial 1: 3 hosts and 1 ethernet switch

- Go 'exercise1’ directory containing the P4 code, JSON and Python scripts.
- Open Terminal in the current directory
- Type “**make run**”
  - this will compile ‘**exercise1.p4**’
  - generate the topology, configure the switch, table entries
  - configure all hosts with the ‘**topology.json**’
- Mininet command prompt will show up , try the commands
  - mininet> h1 ping h2
  - mininet> pingall
  - Xterm h1 etc.
- Type ‘exit’ to leave each xterm and the Mininet command line
- Type ‘make stop’
- Then type ‘make clean’

Implementing packet forwarding for IPV4



Reference : https://github.com/p4lang/tutorials/tree/master/exercises/basic

Implementing packet forwarding for IPV4

IPv4 forwarding Switch will perform the following actions for every packet:

  - update the source and destination MAC addresses
  - decrement the time-to-live (TTL) in the IP header
  - forward the packet out the appropriate port

- Switch will have a single table, and the control plane will populate the table
- Rule is to map an IP address to the MAC address and to output port for the next hop
- Understand the control plane rules, and the data plane logic of the P4 program

Implementing packet forwarding for IPV4

Steps to compile the Program

- Go to the directory containing the P4 code, JSON and Python scripts.
- Open a terminal in the current directory
- Type “make run”
  - this will compile your ‘<filename>.p4’
  - generate the topology, configure the switch, table entries
  - configure all hosts with JSON file
- The Mininet command prompt will show up , try the commands
  - mininet> h1 ping h2
  - mininet> pingall
  - Xterm h1 etc.
- Type ‘exit’ to leave each xterm and the Mininet command line
- Type ‘make stop’
- Then type ‘make clean’

Lab4 Assignment

Q1- Complete a small function (P4_Tutorial 1) in P4 language, learn the syntax of a P4 program, Main features of P4

- Prerequisites:
  - Virtual Machine (VM) with required softwares, reference switch, p4 compilers and dependencies
    - VM Image - https://github.com/p4lang/tutorials

Q2 – In the figure shown on slide 27, the data plane for switch S1 has been provided in the Lab4 (exercise2 folder). Write the data plane programs for switches S2, S3, and S4, and simulate the P4 program

Submission details:

- Create a latex pdf or word report with the code
- Steps to run the code
- Explain what the code is doing

References for further reading

- To understand importance of SDN for future networks
- Introduction to SDN –
  - Gives a good introduction to SDN
  - https://www.youtube.com/watch?v=DiChnu_PAzA&ab_channel=DavidMahler

- Programming the Network Data Plane in P4 -

  Timeline of how P4 came into being, detailed structure of a P4 program and how a compiler loads the same P4 program to different target architectures
  - https://www.youtube.com/watch?v=e1x-XyiBits&ab_channel=GENIEducation

- P4 Runtime: Putting the Control Plane in Charge of the Forwarding Plane –(First 40 minutes)
  - Why OpenFlow was needed, its limitations and how P4Runtime builds over it to act as a betteí alteínative
  - https://www.youtube.com/watch?v=oEV_qY4rOWg&ab_channel=OpenNetworkingFoundation