Getting familiar with P4

Himanshu Verma, Soumya Kanta Rana, Joydeep Pal, Deepak Choudhary, Nithish K. Gnani
Under the guidance of Prof. T.V. Prabhakar

A Tutorial session by Indian Institute of Science, Bangalore

p4.org

What is P4

- The 4 Ps: **P**rogramming **P**rotocol-independent **P**acket **P**rocessors.
- Open source, Domain-Specific programming language for network devices.
- Specifies how data plane devices (switches, routers, NICs, filters, etc.) process packets.
- Instead of having fixed-function ASICs as networking devices, which provide only limited configuration changes, P4 enables the developer to specify packet processing action completely.
- Programmable devices (eg: NetFPGA, Netronome Agilio, Tofino, etc.) come with the hardware and their specific P4 compiler.
- P4 enables the developers to implement network functions on existing protocols (essentially aiding Network Function Virtualization)
- Allows developers to test and implement new protocols quickly.

Traditional Switches

Traditional L2 Switch:

- Forwards packets based on Ethernet MAC address.
- It learns from the Source MAC address on the incoming port and redirects all packets with that address as a destination onto that port only.
- Hence reduces packet noise on the ports.

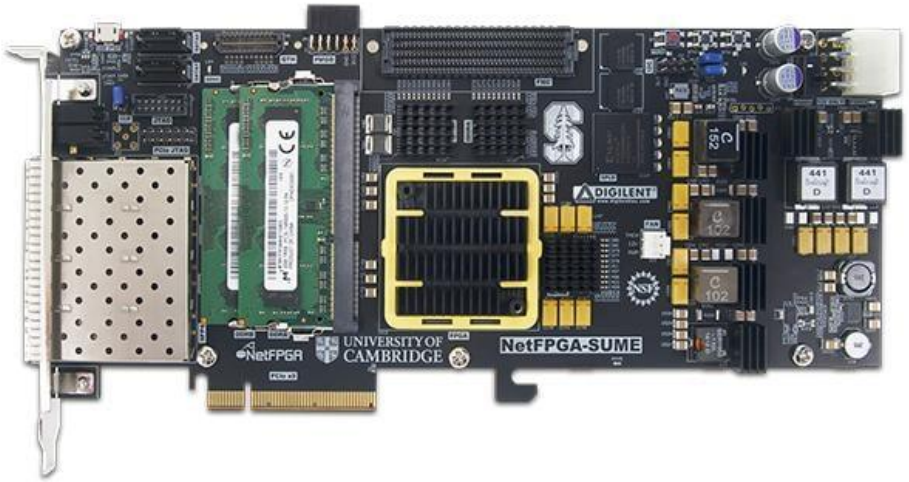
Router:

- Forwards packets based on IP addresses.
- Used to connect different sub-networks.
- Normally used to connect a local network to the internet.

Hardware Support for P4

Netronome Agilio CX 2x10GbE

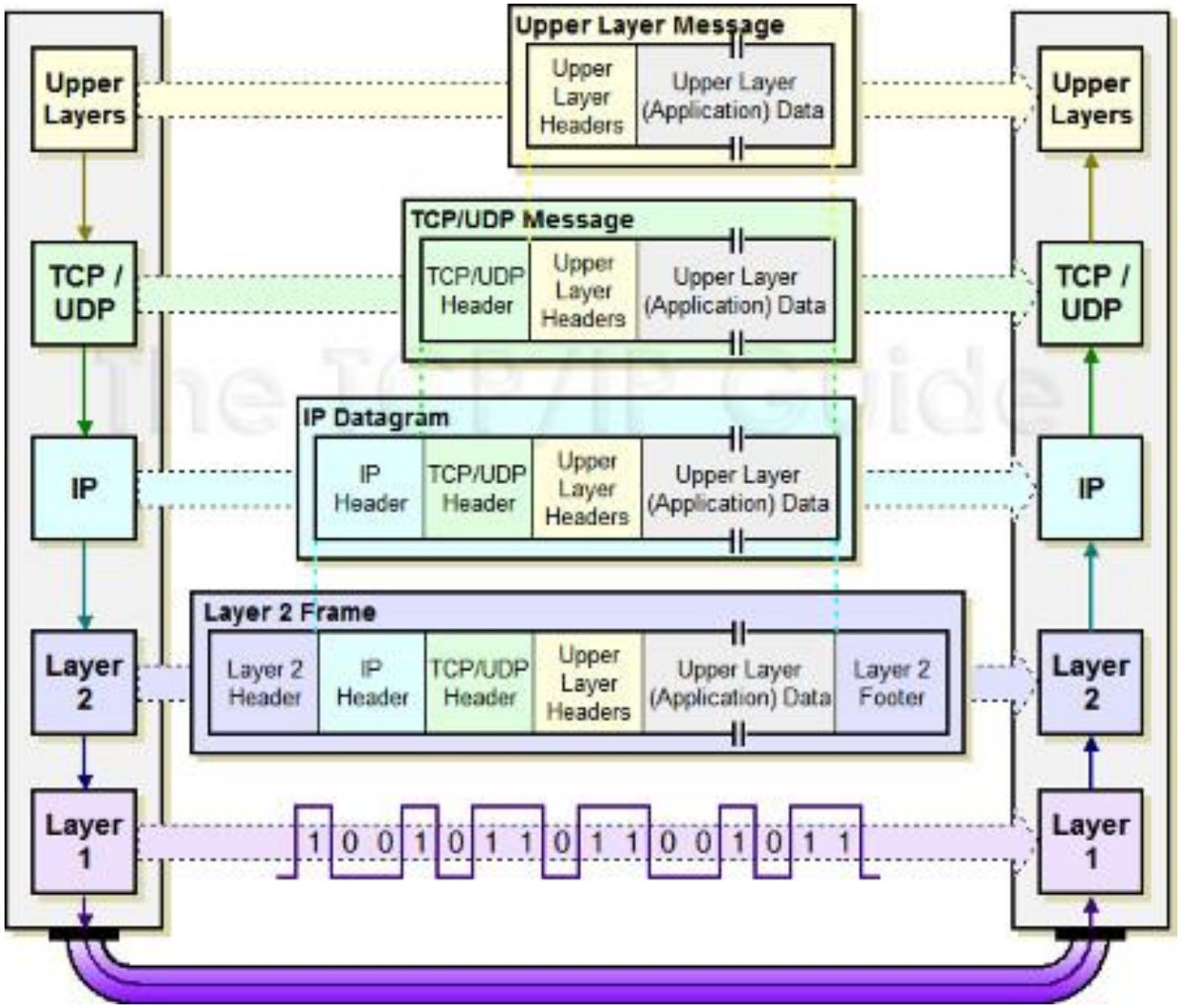
- Smart NIC with 2 ports capable of 10 Gbps each.
- NFP 4000 (Network Flow Processor) consisting of Arm11 Core, 48 packet processor cores, 60 flow processor cores, accelerators.
- 2GB DDR3 onboard memory.



NetFPGA Sume

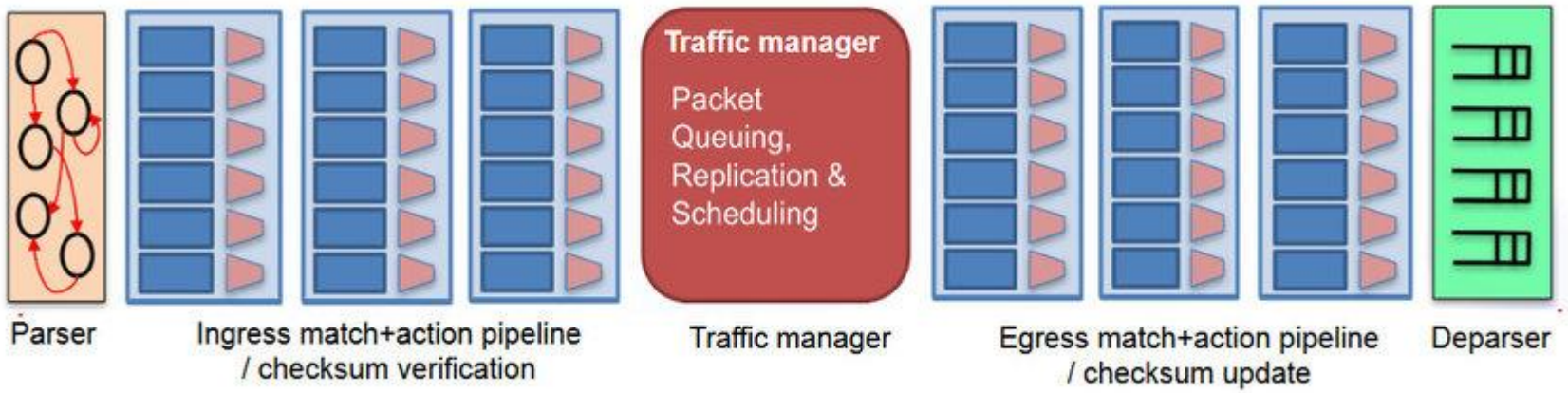
- Development Platform with Four SFP+ 10Gb/s ports
- Xilinx Virtex 7 FPGA at the core.
- Two 4GB DDR3 SODIMM
- Three x36 72Mbits QDR II SRAM

Network Layers



Credits: https://www.linkedin.com/pulse/what-tcpip-stack-phillip-zito/

Working of a Programmable Switch



V1model switch
Credits: p4.org

- **Parser**:
  - It extracts Headers of different layers from the Packet.
  - The type of header to be extracted depends on the protocol used at that layer. Hence a state machine logic is also written here.
- **Match Action Pipeline**: They use particular fields of the packet and different action is taken based on them which can modify the packet data, or its flow.
- **Deparser**: It serialises the packet back to be sent out over the Egress port.

Our Playground: Mininet

- Mininet creates 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.

How does a P4 code look like? [Data Plane]

```
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_HaP = 0x1234;

/*************************************************************************
*********************** H E A D E R S  ***********************************
*************************************************************************/

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> HaPAddr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header HaP_t {
    bit<8>    ttl;
    HaPAddr_t srcAddr;
    HaPAddr_t dstAddr;
}

struct metadata {
    /* empty */
}

struct headers {
    ethernet_t   ethernet;
    HaP_t        HaP;
}
```

```
/*************************************************************************
*********************** 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_HaP: parse_HaP;
            default: accept;
        }
    }

    state parse_HaP {
        packet.extract(hdr.HaP);
        transition accept;
    }

}
```

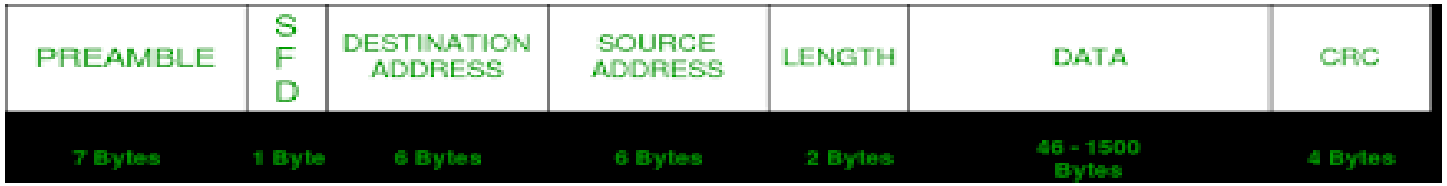
```
/*************************************************************************
**************  I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

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

    action HaP_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.HaP.ttl = hdr.HaP.ttl - 1; //reduce TTL by 1
    }

    table HaP_lpm {
        key = {
            hdr.HaP.dstAddr: lpm;
        }
        actions = {
            HaP_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
            HaP_lpm.apply();
    }
}

/*************************************************************************
***************  E G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

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

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

control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.HaP);
    }
}
```

Control Plane

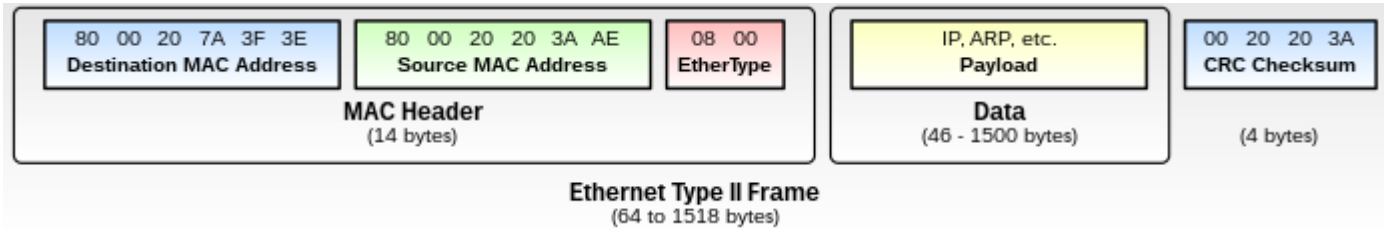
- JSON files are used to define the runtime behaviour of Programmable switches.
- Even though all switches run the same P4 program in a Mininet environment, the JSON files control the Match-Action output and hence can control the flow.
- Topology for the Virtual environment (Mininet) is also defined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

Ethernet Protocol



Credits: geeksforgeeks

- Ethernet has a Destination and Source MAC address, each of 48 bits.
- EtherType: It specifies what kind of higher layer protocol is present in the Ethernet Payload.
- Checksum (4 Bytes) is not our concern; it is handled by the NIC.



Credits: Wikipedia

- Moreover, an Ethernet frame consists of a Preamble (7Bytes) and SFD (Start of Frame Delimiter, 1 Byte). These are also handled by NIC hardware.
- Hence only the MAC Header (14 Bytes) needs to be crafted and parsed by us, since the other bytes are stripped off before being passed over.

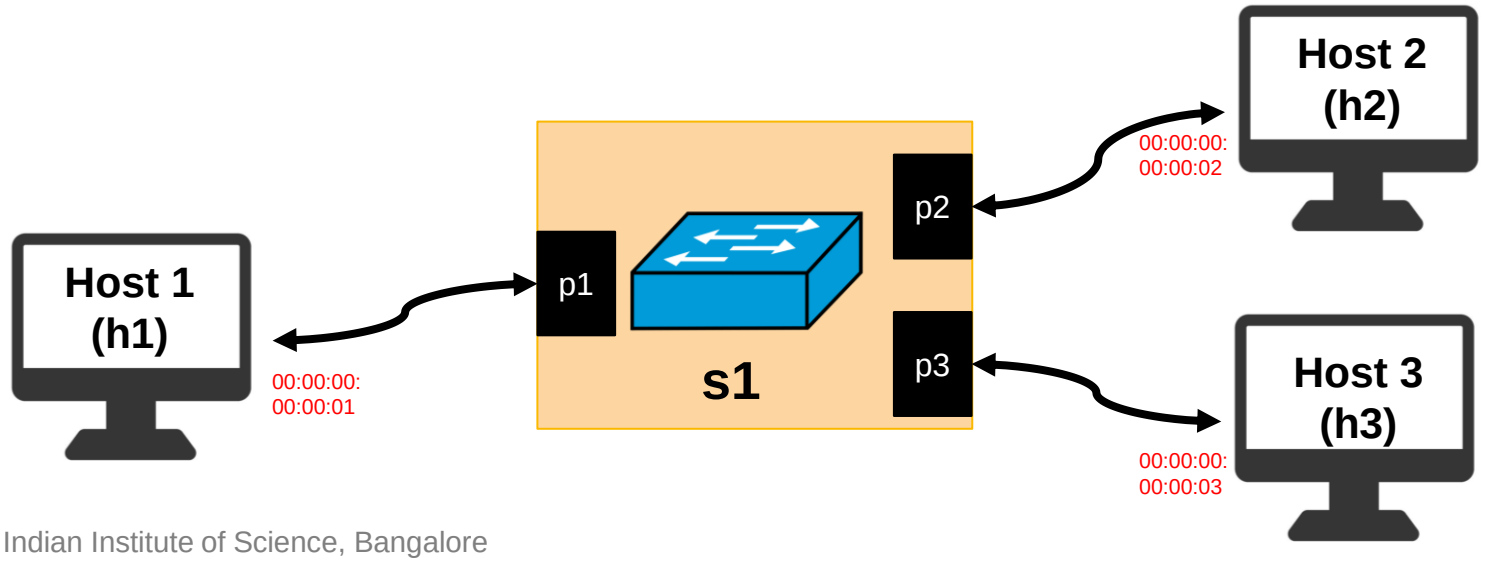
Tutorial 1: Making a Ethernet Switch (pre-learned)

- In this Tutorial, we will create an Ethernet switch.
- The purpose of the switch is to Parse an Ethernet Frame and get the Destination MAC address from the Header.
- It has to match the Destination MAC address present in the packet with some pre-learned MAC addresses.
- The packet must go to the Port which is connected to the device with that MAC address only and not to other ports.
- In the real world, the switches can learn about which port is connected to what MAC address by looking at the Source MAC of incoming packets, but we will not show that for the purpose of simplification.

Network Topology: 3 Hosts connected to 1 Switch

topology.json

```
{
    "hosts": {
        "h1": {"mac": "00:00:00:00:00:01"},
        "h2": {"mac": "00:00:00:00:00:02"},
        "h3": {"mac": "00:00:00:00:00:03"}

    },
    "switches": {
        "s1": { "runtime_json" : "s1-runtime.json" }

    },
    "links": [
        ["h1", "s1-p1"], ["s1-p2", "h2"], ["s1-p3", "h3"]
    ]
}
```

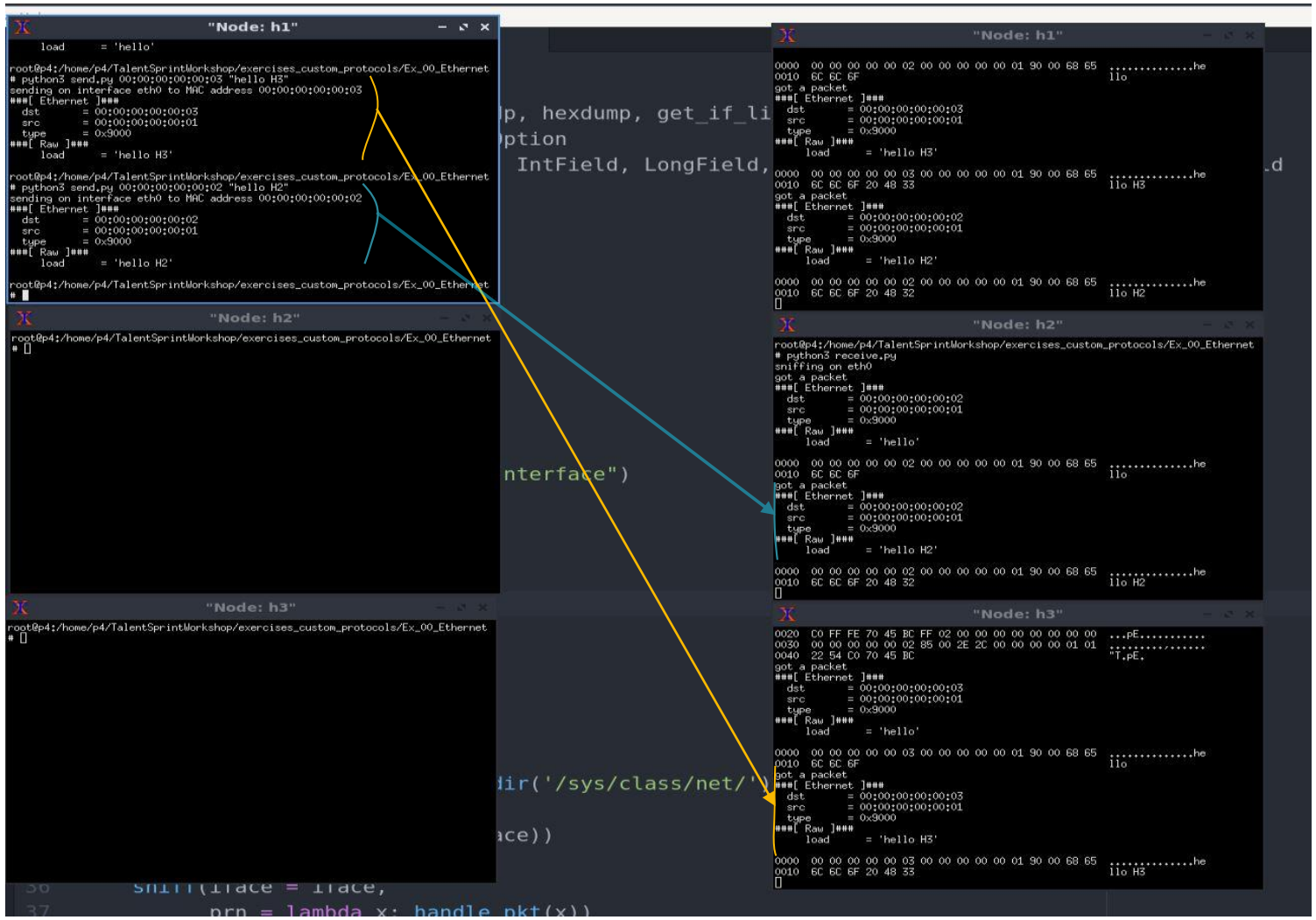


Running Tutorial 1:

- Go to the current directory containing the P4 code, JSON and Python scripts.
- Open Terminal in the current directory
- Type “**make run**”
- This will generate the topology, compile the P4 code and load that onto the switch.
- Now we need to access the Terminals of the Virtual Hosts that we have created. To do that, type “**xterm h1 h1 h2 h2 h3 h3**”.
- This will open 6 terminals. In 3 terminals (one of each host), we need to run the receive.py script. To do that type “**python3 receive.py**”. Now these will display any packets they receive.
- Now, we need to send packets from the hosts. To do that, type “**python3 send.py 00:00:00:00:00:0x “<message payload>”**”, where x can be 1,2 or 3 depending on where we want to send the packet, and we can type in any message which can go as payload. We can do this in any of the other 3 unused (out of 6) host terminals open.
- To exit the mininet terminal, type “**exit**”
- Run “**make clean**” before every retrial

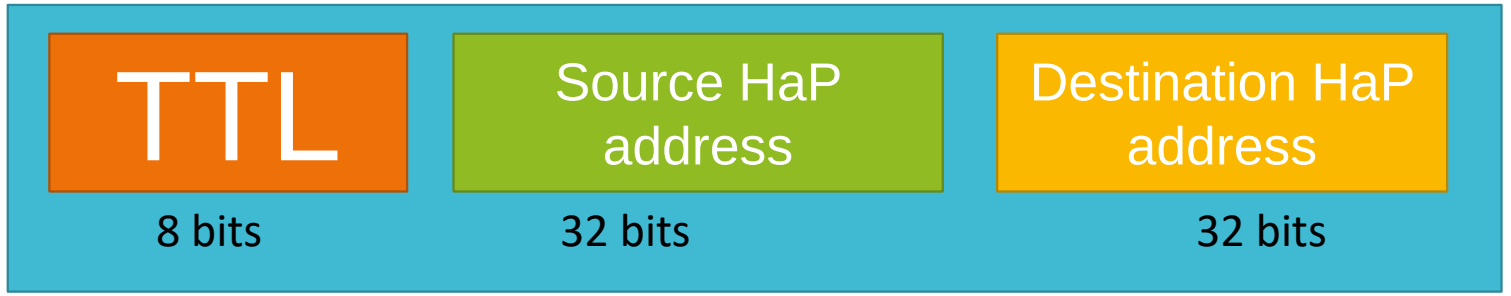
Results

- We observe that packets go to the Host with the destined MAC address only.



Tutorial 2: Defining our own Internet Protocol simpler than IP (*HaP*) ☻

- We need an Internet Protocol to be able to connect different networks.
- The Actual IP has a lot of fields in the Header.
- For the sake of simplicity and to touch upon how P4 is protocol independent and hence can help us define custom protocol, we will be creating our own simplified version of IP called HaP.
- HaP header has only 3 fields: Source HaP address, Destination HaP address and another field called TTL (Time To Live).



Defining our custom HaP layer in P4 and Scapy

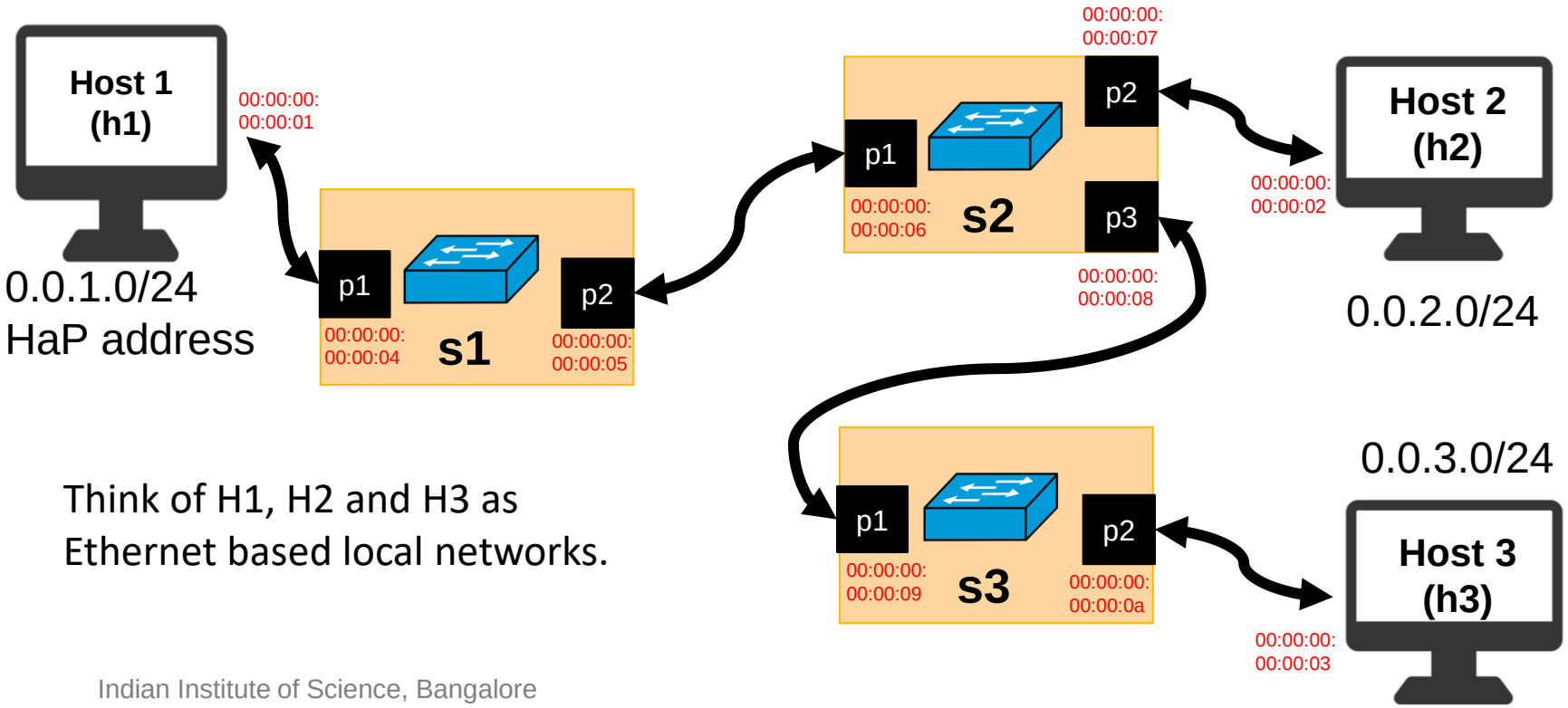
```
typedef bit<32> HaPAddr_t;
```

```
header HaP_t {
    bit<8>    ttl;
    HaPAddr_t srcAddr;
    HaPAddr_t dstAddr;
}
```

```
from scapy.all import *
import sys, os

TYPE_HaP = 0x1234


class HaP(Packet):
    name = "HaP_Header"
    fields_desc = [BitField("ttl", 0, 8),BitField("srcHaP", 0, 32),BitField("dstHaP", 0, 32)]
    def mysummary(self):
        return self.sprintf(" ttl=%ttl% \n srcHaP=%srcHaP% \n dstHaP=%dstHaP% \n")


bind_layers(Ether, HaP, type=TYPE_HaP)
```

Network Topology: Multi-hop packet routing

```
{
    "hosts": {
        "h1": {"ip": "0.0.1.0/24", "mac": "00:00:00:00:00:01",
               },
        "h2": {"ip": "0.0.2.0/24", "mac": "00:00:00:00:00:02"
            },
        "h3": {"ip": "0.0.3.0/24", "mac": "00:00:00:00:00:03",
            }

    },
    "switches": {
        "s1": { "runtime_json" : "s1-runtime.json" },
        "s2": { "runtime_json" : "s2-runtime.json" },
        "s3": { "runtime_json" : "s3-runtime.json" }
    },
    "links": [
        ["h1", "s1-p1"], ["s1-p2", "s2-p1"], ["h2", "s2-p2"], ["s2-p3", "s3-p1"], ["s3-p2", "h3"]
    ]
}
```



Think of H1, H2 and H3 as Ethernet based local networks.

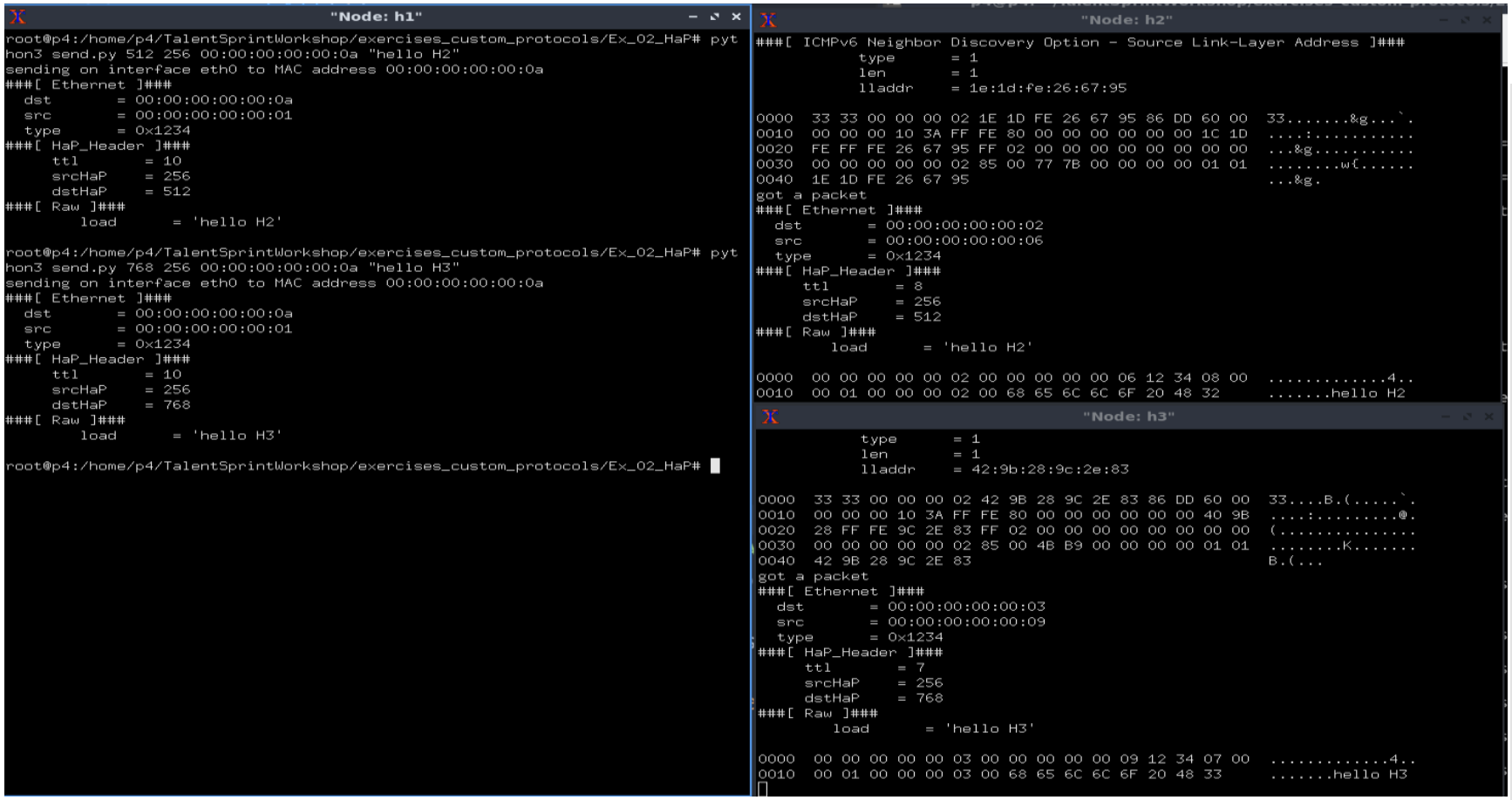
Running Tutorial 2:

- Go to the current directory containing the P4 code, JSON and Python scripts.
- Open Terminal in the current directory
- Type “**make run**”
- This will generate the topology, compile the P4 code and load that onto the switch.
- Now we need to access the Terminals of the Virtual Hosts that we have created. To do that, type “**xterm h1 h1 h2 h2 h3 h3**”.
- This will open 6 terminals. In 3 terminals (one of each host), we need to run the receive.py script. To do that type “**python3 receive.py**”. Now these will display any packets they receive.
- Now, we need to send packets from the hosts. To do that, type “**python3 send.py <destination_HaP> <source_HaP> <dest_port_MAC> “<message payload>””**, the details are in the next slide.
- To exit the mininet terminal, type “**exit**”
- Run “**make clean**” before every retrial

Values to be put for running the send.py script from terminal

Host Name	HaP Address (use accordingly for source and destination)	Packet should be directed to which Ethernet MAC address for first hop
H1	0.0.1.0 (Type 256 in Terminal)	**00:00:00:00:00:04**
H2	0.0.2.0 (Type 512 in Terminal)	**00:00:00:00:00:07**
H3	0.0.3.0 (Type 768 in Terminal)	**00:00:00:00:00:0a**

Results

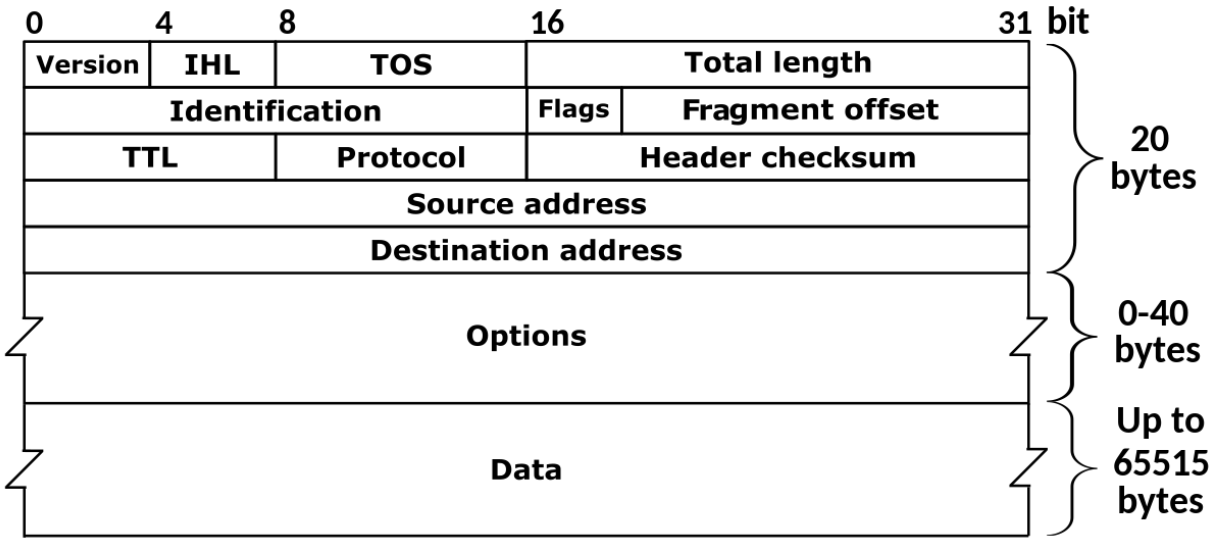


- Packet is routed to the correct host as seen above.
- As per the number of hops, the ttl value is reduced.
- Hence our custom HaP protocol works!

Things to Ponder:

- Need for IP layer?
  - The MAC layer in between can be different, not necessarily Ethernet. Hence their addressing will be non-compliant with each other.
  - OSI's objective is to connect systems with different kinds of hardware.
  - This video by Ben Eater explains it perfectly: https://youtu.be/rPoalUa4m8E
- Need for TTL?
  - Time To Live helps to drop packets which may be stuck in a circular loop.
  - But does our code use TTL for dropping?
  - If not, how can we do that?

The actual IP Header



IPv4 Packet showing the IPv4 Header
Credits: Wikipedia

Defining it in P4:

```
typedef bit<32> ip4Addr_t;
```

```
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;
}
```

Thank You

- Resources:
  - Ben Eater's Networking basics playlist: https://youtube.com/playlist?list=PLowKtXNTBypH19whXTVoG3oKSuOcw_XeW
  - Please download Oracle VM Virtualbox by following this link: https://forum.p4.org/t/installing-open-source-p4-development-tools/84
  - P4 Tutorials: https://github.com/p4lang/tutorials
  - Our GitHub repository: https://github.com/ihimu/P4_Exercises_CustomProtocols (Will be useful to get source codes demonstrated in the workshop).