# EEE4121F Module B Lab 2

By Shahil Poonsamy, PNSSHA003
26 May 2021

## Introduction

The purpose of this lab was to simulate a simple router in the data plane using P4 – a programming language used to program the operations of routers. A simple, traditional forwarding network was created in this lab.

## Topology and implementation

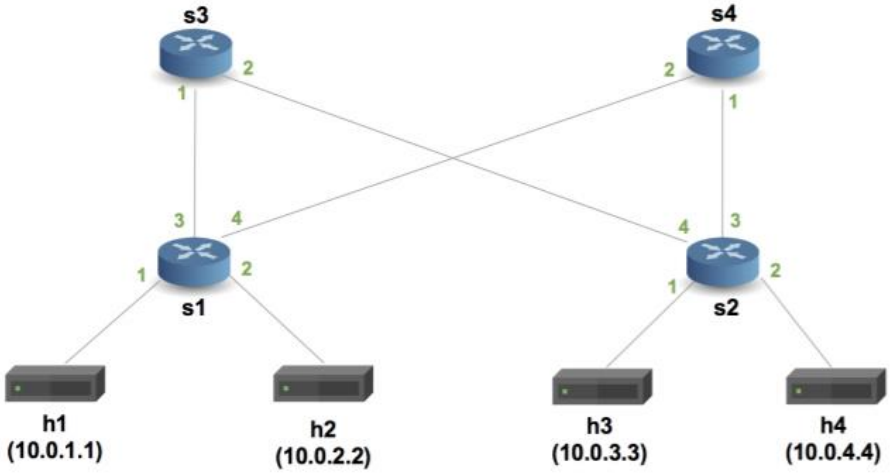


*Figure 1: Network Topology*

This was the topology that was simulated for the lab. Skeleton code was provided by UCT which already involved the creation and interfaces of the switches in this topology. What needed to be implemented were parsing, control, and deparsing logic in the basic p4 file.

These were implemented, respectively, as follows:

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

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

    /* Start state */
    state start {
        transition parse_ethernet;
    }

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

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

}
```

*Figure 2: Parser code implementation*

```
 action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
     /* set egress port for the next hop */
     standard_metadata.egress_spec = port;
     /* update the ethernet destination address with the address of the next hop */
     hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
     /* update the ethernet source address with the address of the switch */
     hdr.ethernet.dstAddr = dstAddr;
     /* decrement the TTL */
     hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
 }
```

*Figure 3: Forwarding action block*

```
apply {
    /* ipv4_lpm should be applied only when IPv4 header is valid */
    if (hdr.ipv4.isValid()) { ipv4_lpm.apply(); }
}
```

*Figure 4: Table application*

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

control MyDeparser(packet_out packet, in headers hdr) {
    /* ethernet then ipv4 */
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}
```

*Figure 5: Deparser logic*

## Ping test result:

To test the code, a simple pingall command was sent through the mininet CLI. No packets were dropped, which means all routers were reachable and so the forwarding operation was successfully created as expected.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
```

## Handling ARP Requests

To process and handle ARP Requests, an additional ARP header and parsing logic would need to be implemented in the code. The header fields were added as:

```
header arp_t {
    bit<16>   hwType;
    bit<16>   protoType;
    bit<8>    hwAddrLen;
    bit<16>   opcode;
    macAddr_t hwSrcAddr;
    ip4Addr_t protoSrcAddr;
    macAddr_t hwDstAddr;
    ip4Addr_t protoDstAddr;
}
```

while the parsing logic was implemented as:

```
 /* Ethernet */
 state parse_ethernet {
     packet.extract(hdr.ethernet);
     transition select(hdr.ethernet.etherType) {
         TYPE_IPV4: parse_ipv4;
         TYPE_ARP: parse_arp;
         default: accept;
     }
/* ARP */
state parse_arp {
    packet.extract(hdr.arp);
    transition accept;
}
```

Control logic would then need to be set to gather the hardware information (specifically the MAC address) and send an ARP reply:

```
action send_arp_reply() {
    standard_metadata.egress_spec = standard_metadata.ingress_port;
    hdr.ethernet.dstAddr    = hdr.arp.hwSrcAddr;
    hdr.ethernet.srcAddr    = meta.mac_da;
    hdr.arp.oper            = ARP_OPER_REPLY;
    hdr.arp.hwDstAddr       = hdr.arp.hwSrcAddr;
    hdr.arp.protoDstAddr    = hdr.arp.protoSrcAddr;
    hdr.arp.hwSrcAddr       = meta.mac_da;
    hdr.arp.protoSrcAddr    = meta.dst_ipv4;
}
```

Finally, this would need to be deparsed the same as how the IPv4 packets are.

## Traceroute Support

To enhance the program with traceroute support, it needs to be able to detect and respond to traceroute packets with an ICMP “Time Exceeded” message, sent to the original sender.

This was implemented to be able to respond to TCP probes, and a lot of the code was derived from (https://github.com/nsg-ethz/p4-learning/blob/master/exercises/09-Traceroutable/solution/p4src/traceroutable.p4)

The first addition to the program is adding the icmp and tcp headers to the program:

```
/* TRACEROUTE */
header icmp_t {
    bit<8> type;
    bit<8> code;
    bit<16> checksum;
    bit<32> unused;
}

header tcp_t{
    bit<16> srcPort;
    bit<16> dstPort;
    bit<32> seqNo;
    bit<32> ackNo;
    bit<4>  dataOffset;
    bit<4>  res;
    bit<1>  cwr;
    bit<1>  ece;
    bit<1>  urg;
    bit<1>  ack;
    bit<1>  psh;
    bit<1>  rst;
    bit<1>  syn;
    bit<1>  fin;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
```

Next, parser and deparser logic was implemented:

```
/* ipv4 */
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    /* TRACEROUTE */
    transition select(hdr.ipv4.protocol){
        6 : parse_tcp;
        default : accept;
    }
}
```

```
/* TRACEROUTE */
state parse_tcp {
    packet.extract(hdr.tcp);
    transition accept;
}
```

After this, control logic was implemented with an ingress icmp table:

```
/* TRACEROUTE */
table icmp_ingress_port {
    key = {
        standard_metadata.ingress_port: exact;
    }
    actions = {
        set_src_icmp_ip;
        NoAction;
    }
    size=64;
    default_action=NoAction;
}

/* TRACEROUTE */
action set_src_icmp_ip (bit<32> src_ip){
    hdr.ipv4_icmp.srcAddr = src_ip;
}
```

And the rules and actions were applied at the end of the control block:

```
apply {
    /* make up a mac address */
    meta.my_mac = 0x000102030405;
    /* ipv4_lpm should be applied only when IPv4 header is valid */
    if (hdr.ipv4.isValid() && hdr.ipv4.ttl > 1) { ipv4_lpm.apply(); }
    /* ARP */
    else if (hdr.arp.isValid()) { arp_forward.apply(); }
    /* TRACEROUTE */
    else if (hdr.ipv4.isValid() && hdr.tcp.isValid() && hdr.ipv4.ttl == 1){

        hdr.ipv4_icmp.setValid();
        hdr.icmp.setValid();
        standard_metadata.egress_spec = standard_metadata.ingress_port;
        bit<48> tmp_mac = hdr.ethernet.srcAddr;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = tmp_mac;
        hdr.ipv4_icmp = hdr.ipv4;
        hdr.ipv4_icmp.dstAddr = hdr.ipv4.srcAddr;
        icmp_ingress_port.apply();

        hdr.ipv4_icmp.protocol = IP_ICMP_PROTO;
        hdr.ipv4_icmp.ttl = 64;
        hdr.ipv4_icmp.totalLen = 56;
        hdr.icmp.type = ICMP_TTL_EXPIRED;
        hdr.icmp.code = 0;
        truncate((bit<32>)70);

    }
}
```

## Router replacement:

This program will suffice as a very basic router, but only in situations where ipv4 packets or tcp probes are being sent, it will not be able to handle the processing of IPv6 packets. On top of that, the ARP requests will only work for IPv4, a separate protocol for IPv6 will need to be implemented here.

## Notes to Tutors:

To run this program, simply navigate to the directory and type “make”. Upon closing, exit the mininet CLI and type “make stop” followed by “make clean” to clean the directory. The instructions call for a bash script to run the program. This was not implemented simply because of how much more efficient it is to type “make”.

For implementing the extra-credit sections, additions in the code have been marked with either “ ARP “ or “ TRACEROUTE “ preceding the relevant code sections.