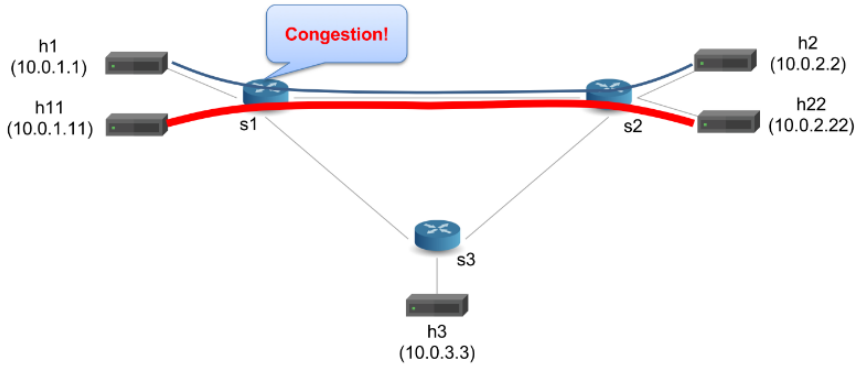
Router Engineering Project Report

- Abhinav Mahajan, IMT2020553, Abhinav.Mahajan@iiitb.ac.in
- Agastya Thoppur, IMT20202528, Agastya.Thoppur@iiitb.ac.in

Enhanced ECN (eECN)

- Our project is a non-exhaustive implementation of the following publication: - https://ieeexplore.ieee.org/document/9058340
- The GitHub repository with all the code is : - https://github.com/Abhinav-Mahajan10/Advanced-and-Faster-ECN
- The topology given below is considered for this project.



- h1 is sending packets to h2, and h11 is overloading the network with the data it is sending to h22. This causes congestion at s1.
- Traditional, Explicit Congestion Notification (ECN) is a congestion control algorithm used in IP networks. In which, s1 checks if its queue depth is greater than some predefined threshold. If so, it sets the ECN field of the packet coming from h1 to 3. This is only done if the ECN field of this packet is 1 or 2, which indicates that it is compatible with ECN. Then the onus is on the receiver to notify back to the sender that there is congestion.
- The problem with traditional ECN is that it takes at least the full RTT(round trip time) to tell h1 that it should reduce its congestion window size. It also generates an additional packet to notify h1, (the ECN echo packet) causing additional traffic in an already congested network.
- eECN speeds up this process. In case of congestion, instead of waiting for h2 to send h1 a packet with the ECN field set, it makes use of any ACK packets being sent from h2 to h1. The ECN-echo bit in an incoming ACK packet is set to 1 and

forwarded to h1. This speeds up the ECN process and also ensures that an additional packet does not have to be generated.

- Once h1 reduces its congestion window size, the next packet it sends will have the CWR bit set to 1. This notifies s1 that h1 has reduced its congestion window size. s1 clears this bit and forwards the packet to h2.
- This algorithm only works well with TCP networks and any other networks which supports end user to end user connections so that we can send acknowledgements and ECN echos.

eECN Algorithm/Pseudocode

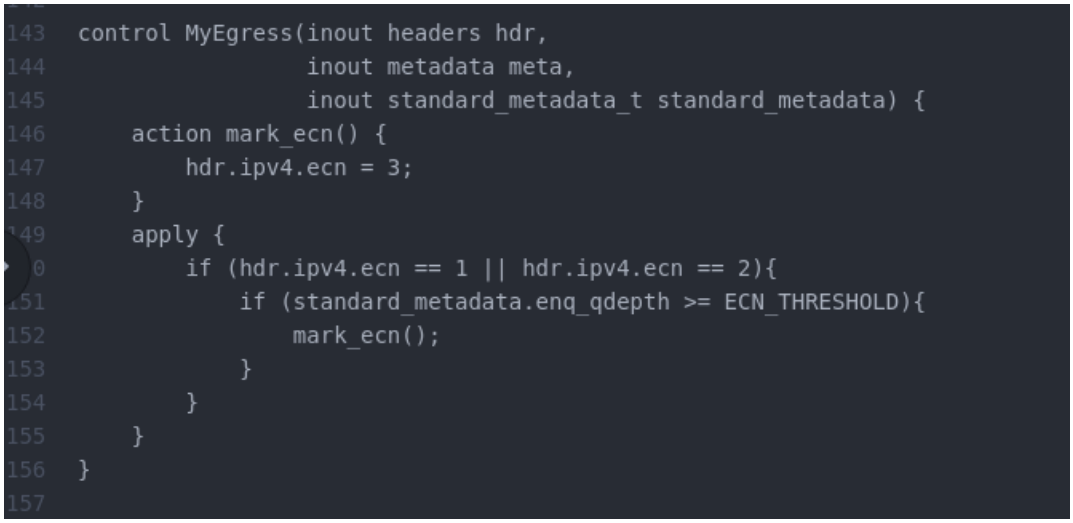
```
If ipv4.ecn==1 or ipv4.ecn==2
        If queue_depth>threshold
                ipv4.ecn=3
                Store port_no in port_no_register
                Store source_ip in source_ip_register

If ACK_packet and dest_ip==source_ip_register and dest_port==port_no_register
        tcp.ecn_echo=1
        ipv4.ecn=0

If ACK_packet and source_ip==source_ip_register and tcp.cwr==1
        tcp.cwr=0
```

Our ECN code

- Since eECN only works on TCP networks, we defined a TCP header in the ecn.p4 file.

```
header tcp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<32> segNo;
    bit<32> ackNo;
    bit<4> dataOffset;
    bit<3> res;
    bit<1> ns;
    bit<1> cwr;
    bit<1> ece;
    bit<6> ctrl;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}
```

- The original ECN code in the p4lang tutorial github repository looked like this-



```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    action mark_ecn() {
        hdr.ipv4.ecn = 3;
    }
    apply {
        if (hdr.ipv4.ecn == 1 || hdr.ipv4.ecn == 2){
            if (standard_metadata.enq_qdepth >= ECN_THRESHOLD){
                mark_ecn();
            }
        }
    }
}
```

- Screenshots of the MyEgress function in our eECN code are given below. The below code snippet checks if the network is congested and stores the IP

address and port number of the host causing congestion in p4 registers.

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    register<bit<32>>(1) source_ip_register;
    register<bit<16>>(1) source_port_register;
    action mark_ecn() {
        hdr.ipv4.ecn = 3;
    }
    apply {
        if (hdr.ipv4.ecn == 1 || hdr.ipv4.ecn == 2){
            if (standard_metadata.enq_qdepth >= ECN_THRESHOLD){
                mark_ecn();
                source_ip_register.write(0, hdr.ipv4.srcAddr);
                source_port_register.write(0, hdr.tcp.srcPort);
            }
        }
        bit<32> source_ip_register_value;
        bit<16> source_port_register_value;
```

- The below code snippet checks if the incoming packet is an ACK packet. It also checks if the ACK packet's destination is the host that is causing congestion. If so, it sets the echo bit to 1 and resets the ECN bit. It then checks the CWR bit of an ACK packet coming from the sending host. If the CWR bit is set, the congestion window size has been reduced, and the CWR bit is reset.

```
                source_port_register.write(0, hdr.tcp.srcPort);
            }
        }
        bit<32> source_ip_register_value;
        bit<16> source_port_register_value;

        source_ip_register.read(source_ip_register_value, 0);
        source_port_register.read(source_port_register_value, 0);

        if(hdr.tcp.ctrl & 16 != 0 && hdr.ipv4.dstAddr == source_ip_register_value
            && hdr.tcp.dstPort == source_port_register_value && hdr.ipv4.ecn == 3)
        {
            hdr.tcp.ece = 1;
            hdr.ipv4.ecn = 0;
        }
        if(hdr.tcp.ctrl & 16 != 0 && hdr.tcp.cwr == 1 &&
          hdr.ipv4.srcAddr == source_ip_register_value)
        {
            hdr.tcp.cwr = 0;
        }
    }
}
```

Changes made to the send.py and receive.py files

- This part of the project was particularly challenging. We had to make minimal changes to our environment, topology building files so that we can run our

simulations on both a UDP and TCP environment. However, in the process, we learnt how to use the Scapy library in python, which makes and generates packets and takes care of the intricacies of UDP and TCP modes of transmissions with minimal changes to the code. Below I will attach the corresponding receive.py and send.py files for TCP vs UDP.

- send.py differences: -

```
addr = socket.gethostbyname(sys.argv[1])
iface = get_if()

pkt = Ether(src=get_if_hwaddr(iface)) / IP(dst=addr, tos=1) /
        TCP(sport=1234, dport=4321, flags="S") / sys.argv[2]
pkt.show2()
try:
  for i in range(int(sys.argv[3])):
    sendp(pkt, iface=iface)
    pkt.show2()
    sleep(1)
except KeyboardInterrupt:
    raise
```

```
if len(sys.argv)<4:
    print('pass 2 arguments: <destination> "<message>" <duration>')
    exit(1)
addr = socket.gethostbyname(sys.argv[1])
iface = get_if()

pkt = Ether(src=get_if_hwaddr(iface), dst="ff:ff:ff:ff:ff:ff") /
    IP(dst=addr, tos=1) / UDP(dport=4321, sport=1234) / sys.argv[2]
pkt.show2()
#hexdump(pkt)
try:
  for i in range(int(sys.argv[3])):
    sendp(pkt, iface=iface)
    sleep(1)
except KeyboardInterrupt:
    raise


if __name__ == '__main__':
    main()
```

receive.py differences: -

```
def handle_pkt(pkt):
    print("got a packet")
    pkt.show2()
#    hexdump(pkt)
    sys.stdout.flush()


def main():
    ifaces = [i for i in os.listdir('/sys/class/net/') if 'eth' in i]
    iface = ifaces[0]
    print("sniffing on %s" % iface)
    sys.stdout.flush()
    sniff(filter="tcp", iface = iface,
          prn = lambda x: handle_pkt(x))
    # sniff(filter="udp and port 4321", iface = iface,
    #       prn = lambda x: handle_pkt(x))

if __name__ == '__main__':
    main()
```

```
def handle_pkt(pkt):
    print("got a packet")
    pkt.show2()
#    hexdump(pkt)
    sys.stdout.flush()


def main():
    iface = 'eth0'
    print("sniffing on %s" % iface)
    sys.stdout.flush()
    sniff(filter="udp and port 4321", iface = iface,
          prn = lambda x: handle_pkt(x))

if __name__ == '__main__':
    main()
```

Testing our code

- h1 sent packets to h2 for 50 seconds. h11 sent heavy load packets to h22 for 45 for UDP and 15 seconds for TCP. The reason for that is, in AIMD TCP due to the heavy congestion posed by h11 to h22, there might be huge congestion and there might be packet drop if we prolong traffic from h11 to h22. Screenshots of the TOS field (equivalent to the ECN bit that indicates congestion) for 4 cases are given below.
- Case 1 and 2: - Traditional ECN with UDP(on the left), Congestion occurs as soon as h11 starts sending packets. The ECN bit is marked, and congestion will only stop when h2 sends a packet back to h1 with the ECN bit marked. eECN with UDP (on the

right) The results of this are similar since h2 is not sending any packets back to h1.

```
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
```

```
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x1
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
tos            = 0x3
```

- Case 3 and 4: - Traditional ECN with TCP(on the left), The TOS field is eventually set back to 1 from 3 (After three packet transmissions when the network is congested). This is because the ECN Echo bit is set to 1 on the ACK packet coming from h2 to h1. eECN with TCP(on the right) The TOS field is set back to 1 from 3 after just one packet transmission during network congestion. This is a great result! We are able to detect congestion and act on it much faster!

```
tos            = 0x1
tos            = 0x0
tos            = 0x1
tos            = 0x0
tos            = 0x3
tos            = 0x0
tos            = 0x3
tos            = 0x0
tos            = 0x3
tos            = 0x0
tos            = 0x1
tos            = 0x0
tos            = 0x1
tos            = 0x0
tos            = 0x1
tos            = 0x0
tos            = 0x1
tos            = 0x0
```

```
tos            = 0x1
tos            = 0x0
tos            = 0x1
tos            = 0x0
tos            = 0x3
tos            = 0x0
tos            = 0x1
tos            = 0x0
tos            = 0x1
tos            = 0x0
```