

INT based Packet waiting time in the Buffer and tracing the path of the packet

A SDN COURSE PROJECT REPORT

Submitted by

Atilli Sanjeet (18100011)

Abhishek Pragada(18101031)

Sayyed Bashar Ali (18100051)

Under the guidance of

Dr. VENKANNA U.

(Assistant Professor, IIITNR)



**Dr. Shyama Prasad Mukherjee International Institute of
Information Technology, Naya Raipur**

APRIL 2021

Abstract

Internet traffic maintenance has become quite a necessity these days for data centers. With the advancement of networks we developed the state of the art network management utility system to calculate packet waiting time in a sophisticated Software Defined Network which enables dynamic, programmatically efficient network configuration in order to improve performance and monitoring, making it more like cloud computing than traditional network management. The focus of this paper is network monitoring with the help of INT. Inband Network Telemetry (INT) is a framework that is designed to monitor, collect, and report flows and network states, by the data plane, without requiring intervention or work by the control plane. INT sources embed the monitored INT flow information in normal data packets, and all the downstream devices add the same information with their details that are based on the source information received. Inband Network Telemetry is used to achieve per-packet network visibility with low overheads. Thus with the help of INT we calculated the time the packets remains in the ingress pipeline till it's exit in the egress pipeline thus we used the metadata information to calculate the buffer time and secondly used switch IDs to track the packet in the network and track it's route through which the packet transit through switches. Implementation is done with the help of digest packet which holds value of source address, destination address, timestamp.

Keywords: data centers, INT, flow info, network states, cloud computing

1. Introduction

It is well known how much monitoring a system matters even after it's deployment. This paper discusses the monitoring of a system and calculating a packet takes time in a SDN environment inside the buffer i.e in ingress and egress. SDN's primary motive is to make programmability of intermediate devices. Till date old techniques like port based classification and deep packet inspection were used in the traditional network for monitoring the network. But the SDN implementation of INT has reduced the task of monitoring to a huge extent. At INT, source flows to be monitored are filtered using the flow watchlist. Various pieces of information on the filtered flows are extracted, based on the collect parameters enabled, such as ingress or egress port ID, ingress or egress timestamp, switch ID, and queue occupancy.

The data packets are encapsulated with the INT information and sent toward the destination. The telemetry INT reports are then sent to the monitor-collector, based on whether the events are flow events, packet-drop events, or queue-congestion events. To achieve this we have used an INT packet which captures switch ID, source address, destination address. With the help of digest we display the data in the other window. The unique ID of a switch. Switch IDs must be unique within an INT domain. The switch ID is generated automatically from bytes [0][1][4][5] of the VDC MAC address.

With knowing timestamps for packets, network administrators can analyze better for any congestion in the network. This will ultimately increase the QoS of the network. With each packet details tasks like bandwidth allocation, packet tracing becomes very easy. With the help of the P4 program in the mininet environment the program is developed. To develop the P4 program we have used INT headers, a packet header that carries INT information, INT Packet — A packet containing an INT Header, INT Instruction — Instructions that are embedded in the INT header, indicating which INT Metadata (defined below) to collect at each INT switch. The collected data is written into the INT Header and INT Metadata — Information that an INT Source inserts into the INT Header. Examples of metadata are described in INT Collection Parameters. We have used the Local flow reports which is a general telemetry report, which is generated from flow events. Sent from the source or sink for host-to-host data flows matching the watchlist.

The results of the statistics stored in a text file which is exported using `digest_messages.py` file. In this project we show how to use the `bmw2` digest extern to send information to the control plane using an out of band channel. A message digest is a fixed size numeric representation of the contents of a message, computed by a hash function. A message digest can be encrypted, forming a digital signature. Messages are inherently variable in size. A message digest is a fixed size numeric representation of the contents of a message.

2. Research problem

Telemetry is an automated process for remotely collecting and processing network information. Network telemetry has been widely considered as an ideal means to gain sufficient network visibility with better scalability, accuracy, coverage, and performance than traditional network measurement technologies. Compared to traditional network measurement, software-defined measurement has great advantages in openness, transparency, and programmability. INT is defined as a framework designed to allow the collection and reporting of network state, by the data plane, without requiring intervention or work by the control plane. The specification for INT can be found [here](#). The key is to insert critical metadata, which can be extracted and interpreted later, in-band without affecting network performance.

3. Major contributions

- Implemented Timestamp in P4 for calculating the arrival time of the packets.
- Calculated the packet's waiting time in the buffer.
- Used digest to send the collected data on to the controller.

4. Implementations

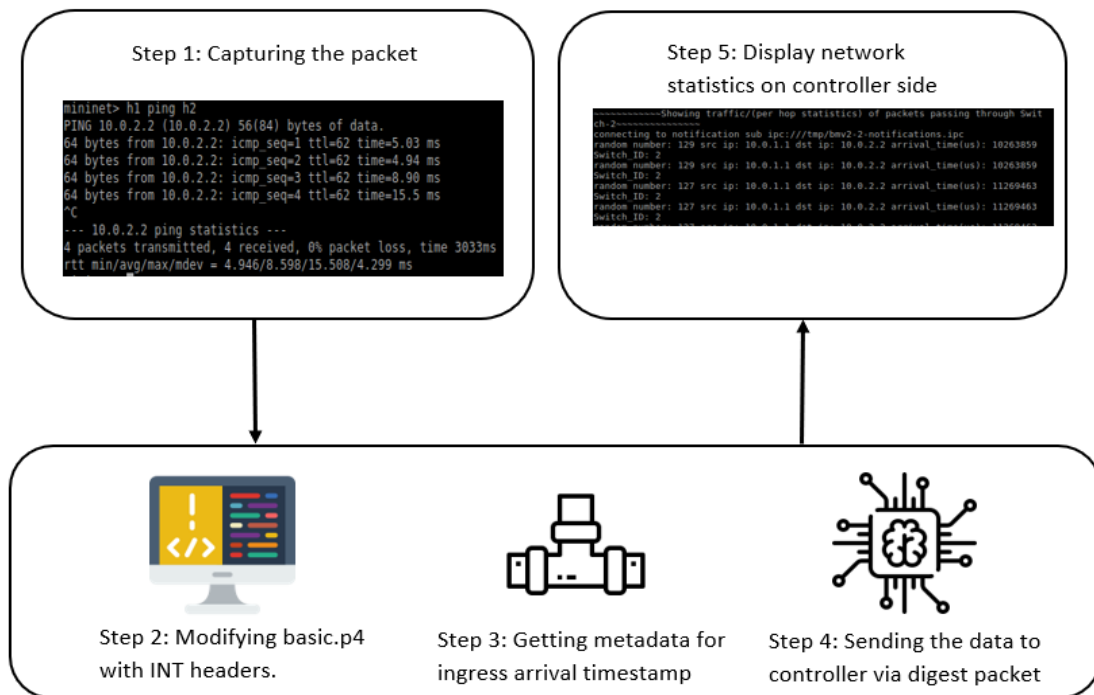


Fig (1) : Conceptual Diagram

The following project has used the mininet environment for running the p4 programs. Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine. We used the ping command to send packets to the receiver. Ping command works by sending out packets and waiting for replies over the network from another computer or network device. This is useful to test the ability of the source computer to reach a specified destination computer.

Fig (1) signifies the block diagram of our work

Firstly we designed a network topology using 3 switches S1,S2,S3 and 4 hosts H1,H2,H3,H4 as shown in Fig (2).

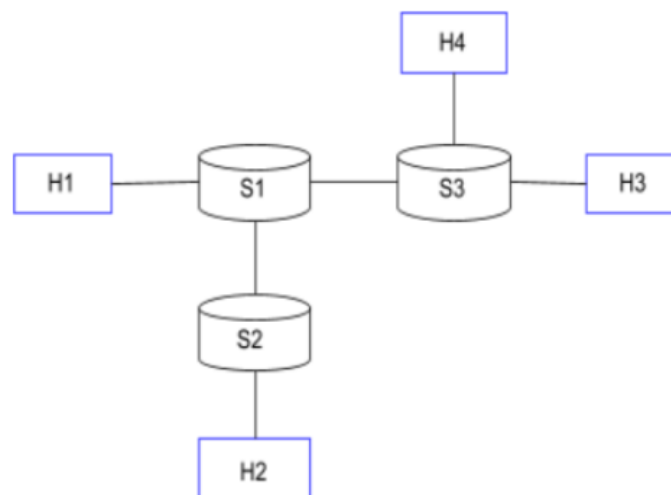


Fig (2) : Topology

Below in Fig (3) are the flow rules inserted for switch S1, similarly the rules are defined for the other two switches S2, S3 as well.

```
table_set_default ipv4_lpm drop
table_set_default int_table add_int_header 1
table_add ipv4_lpm ipv4_forward 10.0.1.1/32 => 00:00:0a:00:01:01 1 1
table_add ipv4_lpm ipv4_forward 10.0.2.2/32 => 00:01:0a:00:02:02 2 1
table_add ipv4_lpm ipv4_forward 10.0.3.4/24 => 00:00:0a:00:03:04 3 1

mirroring_add 100 4
```

Fig (3) : Flow rules of switch S1

4.1 Capturing the packet

Packets are being generated using the message digest.

1. Ethernet Header
2. IPV4 Header

Each header field contains some attributes which are shown in table (1).

Header Fields	Values
Ethernet	MAC source and destination address, Ethernet type
IPV4	Version, IHL, TOS, Total Length, Identification, Flags, Flag Offset, TTL, Protocol, Options, Header checksum, Source and Destination address

Table (1) : Headers Values

4.2 Modify the basic p4 with int headers

Inband Network Telemetry (“INT”) is a framework designed to allow the collection and reporting of network state, by the data plane, without requiring intervention by the control plane in collecting and delivering the state from the data plane. In the INT architectural model, packets may contain header fields that are interpreted as “telemetry instructions” by network devices.

INT traffic sources can embed the instructions either in normal data packets, cloned copies of the data packets or in special probe packets. In this case we add INT headers to the packets that are generated using digest.

Now we define the int header as shown in the Fig (4)

```
header int_header_t {  
    switch_id_t switch_id;  
    queue_depth_t queue_depth;  
    output_port_t output_port;  
    |  
    ingress in_ts;  
}
```

Fig (4) : INT header

This included the fields such as source address, destination address, switch id, ingress and egress timestamp.

4.3 Getting metadata for ingress arrival time stamp

Metadata is the intermediate data generated during execution of a P4 program. In this case the metadata includes the fields shown in Fig (5)

```
struct learn_t {  
    bit<8> digest;  
    bit<32> srcIP;  
    bit<32> dstIP;  
    bit<64> arrival_time;  
    bit<8> sid;  
}  
  
struct learn_2 {  
    bit<64> egress_time  
}
```

Fig (5) : Meta data of Ingress and Egress related fields

After inserting the INT header, we do the ingress processing of the metadata and send the time stamp in the metadata to the controller via digest.

The ingress processing includes the following steps:

1. Setting the IPV4 source address as the source address of metadata.
2. Setting the IPV4 destination address as the destination address of metadata.
3. Assigning the ingress timestamp to the metadata as arrival_time.
4. Assigning the switch id as sid in the metadata.
5. Finally sending these parameters to the controller via digest using “*digest(1, meta.learn);*”

The egress processing includes the following steps:

1. Assigning the egress timestamp to the metadata as egress_time.
2. Finally this will also be sent to the controller via digest.

4.4 Sending the data to controller via digest packet

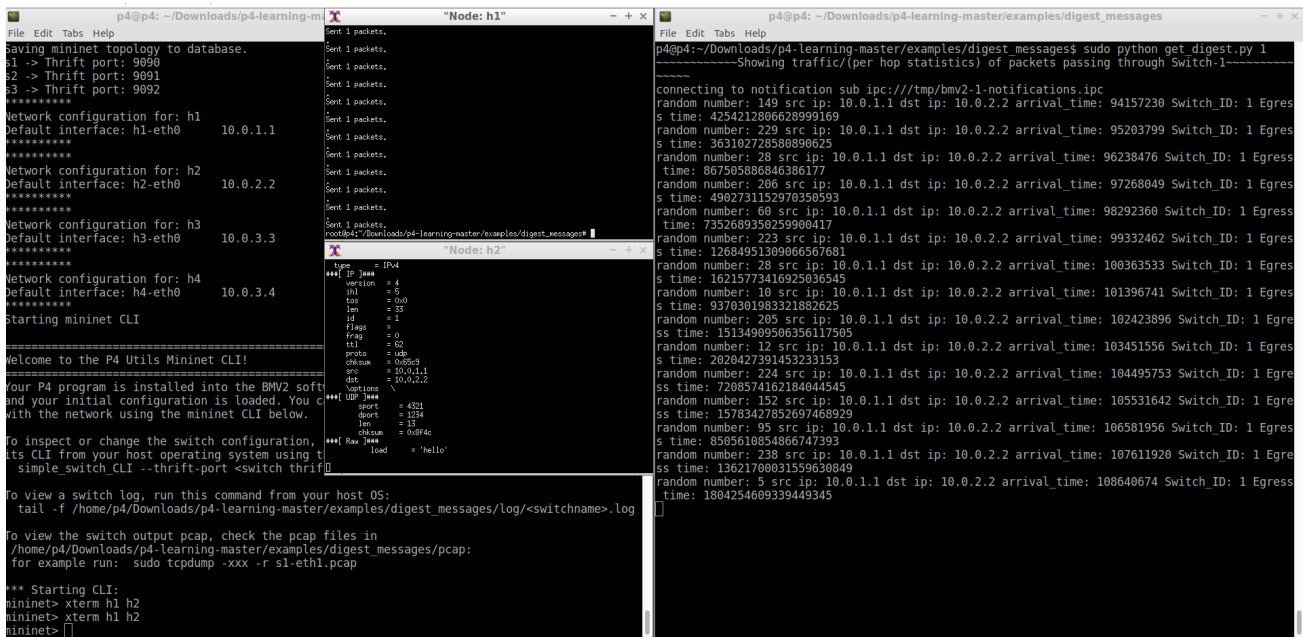
Instead of controller.py as the main python file we use digest.py to establish connection between the controller and the data plane. After sending the data onto the controller we compute the waiting time of each packet in the buffer. In this way we can monitor the network performance with help of digest.

4.5 Displaying network statistics on the controller side

The final network statistics will be obtained after running the digest.py call and saved in a separate text file in a well formatted manner. More detailed information will be discussed in the next section.

Simulation Results and Discussion

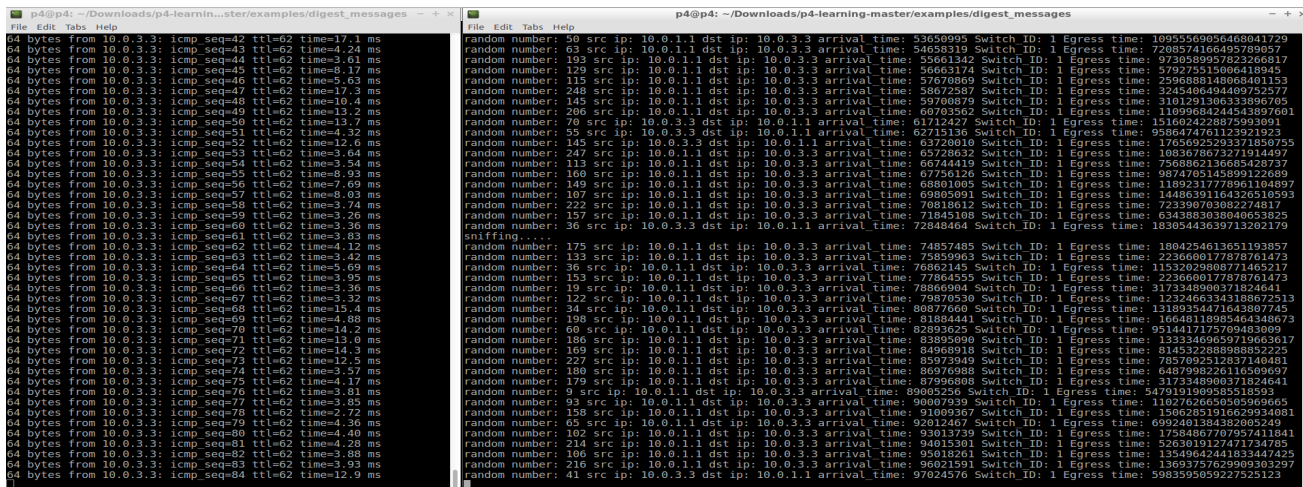
The below picture Fig (6) shows us the simulation on UDP traffic i.e when we use send and receive.py files for sending and receiving the packets, this traffic will be sent by the digest onto the controller which can be seen on the digest.py file's output. And we save the telemetry data which we obtained in a separate .txt file as shown in Fig (8).



The screenshot displays two terminal windows. The left window is the Mininet CLI, showing the setup of four hosts (h1, h2, h3, h4) with IP addresses 10.0.1.1, 10.0.2.2, 10.0.3.3, and 10.0.3.4 respectively. It also shows the start of a packet capture on the h1-eth0 interface. The right window shows the output of the 'get digest.py 1' command, which displays a list of captured UDP packets. Each entry includes the packet number, source IP, destination IP, arrival time, and the switch ID (1 or 2). The packets are numbered 149 through 238.

Fig (6) : Displaying result on UDP traffic

Pinging any two hosts gives us the source ip, destination ip which are the captured from INT headers, it also displays us the switch ID of the host which would be helpful in tracing the path of the packet and finally it gives us the arrival time of the packet as shown in Fig (7) which is one of the main objectives of our work.



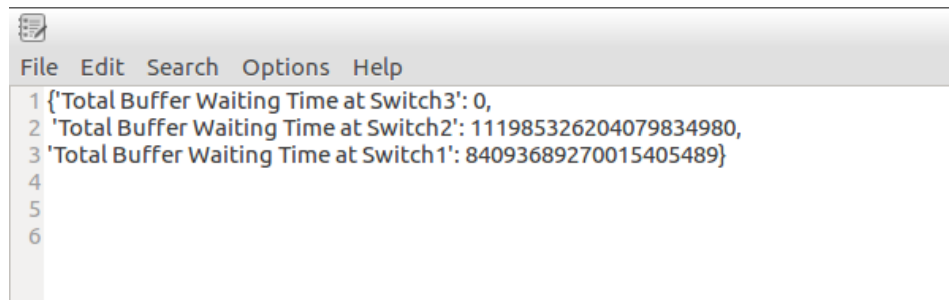
The screenshot displays two terminal windows. The left window shows the output of the 'tcpdump -xxx -r 01-eth1.pcap' command, displaying a list of captured ICMP packets. Each entry includes the packet number, source IP, destination IP, arrival time, and the switch ID (1 or 2). The packets are numbered 42 through 84. The right window shows the output of the 'get digest.py 1' command, which displays a list of captured ICMP packets. Each entry includes the packet number, source IP, destination IP, arrival time, and the switch ID (1 or 2). The packets are numbered 50 through 41.

Fig (7) : Displaying the result on ICMP traffic


```
p4@p4:~/Downloads/p4-learning-master/examples/digest_messages$ python stat.py
Total Buffer Waiting Time at Switch1 84093689270015405489 Total Buffer Waiting Time at Switch2 111985326204079834980 Total Buffer Waiting Time at Switch3 0
p4@p4:~/Downloads/p4-learning-master/examples/digest_messages$
```

Fig (8) : Running stat.py for the buffer time of switches

These were the statistics which we obtained when we generated the control plane traffic between h1, h2 via the command “h1 ping h2” and the results were saved in a separate text file with each switch's buffer waiting time as described in Fig (9) and the command line output as shown above in Fig (8).



```
File Edit Search Options Help
1 {'Total Buffer Waiting Time at Switch3': 0,
2  'Total Buffer Waiting Time at Switch2': 111985326204079834980,
3  'Total Buffer Waiting Time at Switch1': 84093689270015405489}
4
5
6
```

Fig (9) : Buffer Waiting time exported to txt file for 8 packets

Summary

In this project we have developed a prototype for calculating the packets waiting time in the buffer. For which we first created a simple switch functioning p4 code, after that added some changes with respect to the INT headers and the metadata that is to be collected (arrival times of the packets) to the code. Later we sent this collected data to the controller with the help of digest. In this way the complete architecture for sending the relevant information with the help of digest onto the controller was done.

Code

P4 program:

```
/* -*- P4_16 -*- */
#include <core.p4>
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;
const bit<5> IPV4_OPTION_INT = 31;
/*****
***** H E A D E R S *****/
```

```

*****/

#define MAX_INT_HEADERS 9

typedef bit<9> egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

typedef bit<13> switch_id_t;

typedef bit<13> queue_depth_t;

typedef bit<6> output_port_t;

//typedef bit<48> interval_t;

typedef bit<48> ingress;

//typedef bit<48> difference;

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

header ipv4_t {
    bit<4> version;
    bit<4> ihl;
    bit<8> tos;
    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;
}

header udp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<16> length_;
    bit<16> checksum;
}

```

```

struct parser_metadata_t {

    bit<16> num_headers_remaining;

}

struct learn_t {
    bit<8> digest;
    bit<32> srcIP;
    bit<32> dstIP;
    bit<64> arrival_time;
    bit<8> sid;
}

struct learn_2{
bit<64> egress_time;
bit<16> qd;

}

struct metadata {
    learn_t learn;
    learn_2 learn2;
    parser_metadata_t parser_metadata;
}

struct headers {
    ethernet_t ethernet;
    ipv4_t ipv4;
    //ipv4_option_t ipv4_option;
    udp_t udp;

    //int_count_t int_count;

    // int_header_t[MAX_INT_HEADERS] int_headers;
}

error { IPHeaderWithoutOptions }
/*****
***** 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 {

```

```

        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType){

            TYPE_IPV4: parse_ipv4;
            default: accept;
        }

    }

state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition select(hdr.ipv4.protocol) {
        17 : parse_udp;
        default : accept;
    }
}

state parse_udp {
    packet.extract(hdr.udp);
    transition accept;
}

}

/***** CHECKSUM VERIFICATION *****/
control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
    apply { }
}

/***** INGRESS PROCESSING *****/
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, bit<8> swid) {

        //set the src mac address as the previous dst, this is not correct right?
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    }
}

```

```

        //set the destination mac address that we got from the match in the table
        hdr.ethernet.dstAddr = dstAddr;

        //set the output port that we also get from the table
        standard_metadata.egress_spec = port;

        //decrease ttl by 1
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
        random(meta.learn.digest, (bit<8>) 0, (bit<8>) 255);
        meta.learn.srcIP = hdr.ipv4.srcAddr;
        meta.learn.dstIP = hdr.ipv4.dstAddr;
        meta.learn.arrival_time = (bit<64>)standard_metadata.ingress_global_timestamp;
        meta.learn.sid = (bit<8>) swid;

        //digest packet
        digest(1, meta.learn);
    }

table ipv4_lpm {
    key = {
        hdr.ipv4.dstAddr: lpm;
    }
    actions = {

        ipv4_forward;
        drop;
        NoAction;
    }
    size = 1024;
    default_action = NoAction();
}

apply {
    //only if IPV4 the rule is applied. Therefore other packets will not be forwarded.
    if (hdr.ipv4.isValid()){
        ipv4_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) {

    action add_int_header(switch_id_t swid){
        meta.learn2.egress_time = (bit<64>)standard_metadata.egress_global_timestamp;
        meta.learn2.qd = (bit<16>)standard_metadata.deq_qdepth;

    }

    table int_table {

        actions = {

            add_int_header;

            NoAction;

        }

        default_action = add_int_header(1);

    }

    table dummy{

```

```

key = {

    meta.learn2.egress_time: exact;
    meta.learn2.qd: exact;

}

actions = {

}

}

apply {

    int_table.apply();

    dummy.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.tos,
              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);
    }
}

```

```

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

control MyDeparser(packet_out packet, in headers hdr) {
    apply {

        //parsed headers have to be added again into the packet.
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        // packet.emit(hdr.ipv4_option);

        // packet.emit(hdr.int_count);

        // packet.emit(hdr.int_headers);

        packet.emit(hdr.udp);
    }
}

/*****
***** S W I T C H *****
*****/

//switch architecture
V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;

```

Python program digest.py:

```

import numpy
import struct
import sys
import ipaddress
from p4utils.utils.topology import Topology
from p4utils.utils.sswitch_API import SimpleSwitchAPI
import time
import json

class DigestController():

```



```

def __init__(self, sw_name):

    self.sw_name = sw_name
    self.topo = Topology(db="topology.db")
    self.sw_name = sw_name
    self.thrift_port = self.topo.get_thrift_port(sw_name)
    self.controller = SimpleSwitchAPI(self.thrift_port)

def recv_msg_digest(self, msg):

    topic, device_id, ctx_id, list_id, buffer_id, num = struct.unpack("<iQ
iiQi",
                                                                    msg[:32])

    #print num, len(msg)
    offset, offset2 = 18, 26

    msg = msg[32:]
    i=0
    list1=[]
    sublist=[]

    buffer1, buffer2, buffer3=0, 0, 0
    for sub_message in range(num):
        random_num, src, dst, times, switch_id= struct.unpack("!BIIqb", msg[
0:offset])
        try:
            _, _, _, _, etime=struct.unpack("!BIIqbQ", msg[0:offset2])
        except:
            print "sniffing....."
            #random_num, time= struct.unpack("!BII", msg[0:offset])
            #sublist=[]
            if i%2:
                #print "random number:", random_num, "src ip:", str(ipaddress.
IPv4Address(src)), "dst ip:", str(ipaddress.IPv4Address(dst)), "arrival_time:",
times, "Switch_ID:", switch_id, "Egress time:", etime, "Buffer_Time:", abs(etime-ti
mes)

                print "Buffer_Time:", abs(etime-times), "Switch_ID:", switch_id
                '''if switch_id == 1:
                    buffer1+= abs(etime-times)
                elif switch_id == 2:
                    buffer2+=abs(etime-times)
                else:
                    buffer3+=abs(etime-times)'''
            # dict2 = {"Total buffer time at Switch 1":buffer1, "Total buffe
r time at Switch 2":buffer2, "Total buffer time at Switch 3":buffer3}

```

```

        #dict1={"random number": random_num, "src ip": str(ipaddress.IPv4Address(src)), "dst ip": str(ipaddress.IPv4Address(dst)), "arrival_time": times, "Switch_ID": switch_id, "Egress time": etime, "Buffer_Time": abs(etime-times)}
        dict1={"Switch_ID": switch_id, "Buffer_Time": abs(etime-times)}
        #list1.append(dict1.copy())
        '''sublist.append(("random number:", random_num))
        sublist.append(("src ip:", str(ipaddress.IPv4Address(src))))
        sublist.append(("dst ip:", str(ipaddress.IPv4Address(dst))))

        sublist.append(("Switch_ID:", switch_id))
        sublist.append(("arrival_time:", times)) '''
        '''with open("file.txt", "w") as output:
        #output.write(str(times)+"\n")
            output.write("Hello \n")'''
        gk = open('file3.txt', 'a')
        gk.write(str(dict1)+"\n")
        gk.close()
        # json_object = json.dumps(dict1, indent=4)
        '''with open("file3.json", "a") as outfile:
            json_object = json.dump(dict1, outfile)'''

    i+=1
    #list1.append(sublist)

    #print "random number:", random_num, "time:", time
    # print list1

    '''for i in range(len(sublist)):
        with open("file.txt", "w") as output:
            output.write(str(sublist[i]))'''

    self.controller.client.bm_learning_ack_buffer(ctx_id, list_id, buffer_id)

def run_digest_loop(self):

    sub = npy.Socket(npy.AF_SP, npy.SUB)
    notifications_socket = self.controller.client.bm_mgmt_get_info().notifications_socket
    print "connecting to notification sub %s" % notifications_socket
    sub.connect(notifications_socket)
    sub.setsockopt(npy.SUB, npy.SUB_SUBSCRIBE, '')

    while True:
        msg = sub.recv()

```

```

        self.recv_msg_digest(msg)

def main():
    ''' print "~~~~~Showing traffic/(per hop statistics) of packets pas
sing through Switch-3~~~~~"

    DigestController("s3").run_digest_loop() '''
    if int(sys.argv[1])==3:

        print "~~~~~Showing traffic/(per hop statistics) of packets pas
sing through Switch-3~~~~~"

        DigestController("s3").run_digest_loop()
    if int(sys.argv[1])==2:
        print "~~~~~Showing traffic/(per hop statistics) of packets passing
through Switch-2~~~~~"
        DigestController("s2").run_digest_loop()
    if int(sys.argv[1])==1:
        print "~~~~~Showing traffic/(per hop statistics) of packets passing
through Switch-1~~~~~"
        DigestController("s1").run_digest_loop()
    else:
        print "Invalid switch id"
if __name__ == "__main__":
    main()

```

Python send.py

```

#!/usr/bin/env python

import argparse
import sys
import socket
import random
import struct

from scapy.all import sendp, send, hexdump, get_if_list, get_if_hwaddr
from scapy.all import Packet, IPOption
from scapy.all import Ether, IP, UDP
from scapy.all import IntField, FieldListField, FieldLenField, ShortField, Pac
ketListField, BitField
from scapy.layers.inet import _IPOption_HDR

from time import sleep

def get_if():
    ifs=get_if_list()
    iface=None # "h1-eth0"
    for i in get_if_list():

```

```

        if "eth0" in i:
            iface=i
            break;
    if not iface:
        print "Cannot find eth0 interface"
        exit(1)
    return iface

'''class SwitchTrace(Packet):
    fields_desc = [ BitField("swid", 0, 13),
                    BitField("qdepth", 0,13),
                    BitField("portid",0,6)]
    def extract_padding(self, p):
        return "", p

class IPOption_INT(IPOption):
    name = "INT"
    option = 31
    fields_desc = [ _IPOption_HDR,
                    FieldLenField("length", None, fmt="B",
                                  length_of="int_headers",
                                  adjust=lambda pkt,l:l*2+4),
                    ShortField("count", 0),
                    PacketListField("int_headers",
                                    [],
                                    SwitchTrace,
                                    count_from=lambda pkt:(pkt.count*1)) ]'''

def main():

    if len(sys.argv)<3:
        print 'pass 2 arguments: <destination> "<message>"'
        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) / UDP(
            dport=1234, sport=4321) / sys.argv[2]

    # pkt = Ether(src=get_if_hwaddr(iface), dst="ff:ff:ff:ff:ff:ff") / IP(
    #     dst=addr, options = IPOption_MRI(count=2,
    #     swtraces=[SwitchTrace(swid=0,qdepth=0), SwitchTrace(swid=1,qdepth
    # =0)])) / 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()

```

Python receive.py

```

#!/usr/bin/env python
import sys
import struct

from scapy.all import sniff, sendp, hexdump, get_if_list, get_if_hwaddr
from scapy.all import Packet, IPOption
from scapy.all import PacketListField, ShortField, IntField, LongField, BitField, FieldListField, FieldLenField
from scapy.all import IP, UDP, Raw
from scapy.layers.inet import _IPOption_HDR

def get_if():
    ifs=get_if_list()
    iface=None
    for i in get_if_list():
        if "eth0" in i:
            iface=i
            break;
    if not iface:
        print "Cannot find eth0 interface"
        exit(1)
    return iface

'''class SwitchTrace(Packet):
    fields_desc = [ BitField("swid", 0, 13),
                    BitField("qdepth", 0,13),
                    BitField("portid",0,6)]
    def extract_padding(self, p):
        return "", p

class IPOption_INT(IPOption):
    name = "INT"
    option = 31
    fields_desc = [ _IPOption_HDR,
                    FieldLenField("length", None, fmt="B",
                                  length_of="int_headers",
                                  adjust=lambda pkt,l:l*2+4),
                    ShortField("count", 0),
                    PacketListField("int_headers",

```

```

        [],
        SwitchTrace,
        count_from=lambda pkt:(pkt.count*1)) ]'''

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

def main():
    iface = 'h2-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()

```

Python stat.py

```

f = open('file3.txt', 'r')
data = f.read()
data=data.split('\n')
data.remove('')

## Making a good format of the txt file which is being read into a list..

dict1, results={},[]
for row in data:
    row=row[1:-1]
    row=row.split(':')
    row[1]=row[1].split(',')
    results.append([(row[0],int(row[1][0].strip()[:-1])) , (row[1][1].strip(), i
nt(row[2].lstrip())) ]])

## Collecting the TotalBuffer waiting time for each switch...

Totalbuffer1, Totalbuffer2, Totalbuffer3=0,0,0
for res in results:
    if res[1][1]==1: # for switch id 1
        Totalbuffer1+=res[0][1]
    elif res[1][1]==2:
        Totalbuffer2+=res[0][1]
    else:
        Totalbuffer3+=res[0][1]

```

```

## Saving the Statistics to a new text file..

temp_dict={'Total Buffer Waiting Time at Switch1': Totalbuffer1,
  'Total Buffer Waiting Time at Switch2': Totalbuffer2,
  'Total Buffer Waiting Time at Switch3': Totalbuffer3}
f=open('new_file.txt', 'a')
f.write(str(temp_dict.copy())+"\n")
f.close()

## Printing the collected statistics..

#print f'Total Buffer Waiting Time at Switch1 {Totalbuffer1}\n'
#, f'Total Buffer Waiting Time at Switch2 {Totalbuffer2}\n', f'Total Buffer Wa
iting Time at Switch3 {Totalbuffer3}\n'

print "Total Buffer Waiting Time at Switch1",Totalbuffer1,"Total Buffer Waitin
g Time at Switch2",Totalbuffer2,"Total Buffer Waiting Time at Switch3",Totalbu
ffer3

```

References

1. <https://p4.org/p4-spec/docs/PSA-v1.1.0.html#sec-timestamps>
2. https://github.com/nsg-ethz/p4-learning/tree/master/examples/simple_int
3. https://github.com/nsg-ethz/p4-learning/blob/master/examples/digest_messages/digest_messages.p4
4. https://github.com/nsg-ethz/p4-learning/blob/master/examples/simple_int/send.py
5. <https://www.netronome.com/blog/in-band-network-telemetry-its-not-rocket-science/>
6. https://www.researchgate.net/publication/344533358_In-band_Network_Telemetry_A_Survey