Assignment #5
(Introduction to Computer Networks)

학번: 2013311659

이름: 곽 창 근

1. 개발 환경

OS : Windows10

사용 언어 : Python 3.6.7 64bit

사용 프로그램 : Python 3.6.7 Shell

사용 라이브러리 : socket, threading, time, random, os

2. 설명 및 작동 방법

채점 기준을 모두 구현하였다. 설명은 채점 기준들을 중점으로 설명하겠다.

1) In the case of a single sender
   A. NEM limits FR as BLR

Receiver 에서는 NEM 에서 우선 패킷을 받고, BLR 에 따라서 패킷을 드랍시킨다.

본 코드에서는

def packetReceive() : #NEM

def fileReceive() : #RM

위와 같이 packetReceive 함수는 NEM 역할, fileReceive 함수는 RM 역할을 한다.

본론적으로, NEM 함수 내에서 “NEM limits FR as BLR”을 실행해주는 부분은 아래 부분이다.

```
#받은 seq번호 저장
for m in message.decode().split(sep=';') :
    if(m[0:3] == 'seq') :
        r_seq = int(m[4:])
    if(m[0:4] == 'port') :
        r_port = int(m[5:])
if forward_count - check_forward >= blr : # blr에 따른 제어
    if(q_fill < queue_size) :
        if q_fill == 0 :
            queue.append((senderAddress, r_seq, r_port))
            q_fill += 1
        elif random.random() < 1/(q_fill*q_fill) :
            queue.append((senderAddress, r_seq, r_port))
            q_fill += 1
else : # RM으로 보내지는 패킷들
    if q_fill > 0 :
        mailbox = mailbox + queue[:forward_count - check_forward]
        queue = queue[forward_count - check_forward:]
        q_fill = len(queue)
        forward_count += len(queue[:forward_count - check_forward])
        if(forward_count - check_forward < blr) :
            mailbox.append((senderAddress, r_seq, r_port))
            forward_count += 1
    else :
        mailbox.append((senderAddress, r_seq, r_port))
        forward_count += 1
```

처음 for 문에서 port 번호와 보내온 seq 번호를 확인한 후에 다음 if-else 문을 통하여 BLR 값에 따라서 제어를 해준다. 현재 지정된 BLR 만큼 이미 forward 를 해줘서 더 이상 보낼 수 없다면 queue 가 비었는지를 보고 queue 에 넣을지 안 넣을지를 선택해주고, BLR 보다 적은 수를 forward 해줬다면 RM 으로 패킷을 보내준다. 보낼 때는 mailbox 라는 리스트를 만들어서 여기에 패킷을 저장하면 RM 에서 읽는 방식이다.

그래서 밑에 “4)”의 FR 그래프를 봐도 알겠지만 multiple sender 상황 뿐만 아니라 single sender 에서도 항상 |1-FR/BLR|의 값을 0.1 이하로 유지하였다.

B. Keep | 1-G/BLR | below 0.1

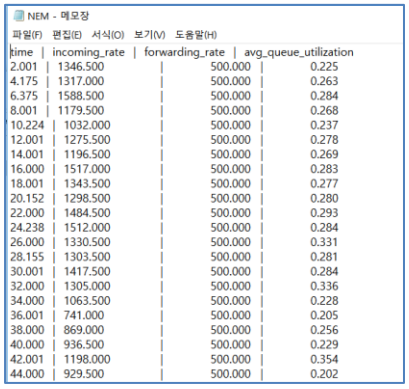
2 초마다 Goodput 을 Sender 에서 측정한다. Goodput 의 측정은 2 초 전에 받았던 ACK 과 현재의 ACK 을 비교해서 얼마나 진행됐는지를 측정하였다.

추가로 Sender 에서의 sending rate 은 매번 패킷을 전송할 때마다 count 를 세서 2 초 간 얼마나 보냈는지를 측정하였다. 아래 코드와 같이 패킷을 보낸 후에는 매번 p_count += 1 을 실행해서 패킷 보낸 횟수를 측정하였다.

```
s = "seq=%d;port=%d" % (seq, port_ack)
#print(s) #확인용
s_socket.sendto( s.encode(), (receiverIP, receiverPort) )
timeline[seq] = cur_time #timeline에 해당 패킷 보낸시간 저장
seq += 1
p_count += 1
```

C. Properly well-written log files

아래 사진은 Receiver 의 NEM.log 의 일부 캡처 사진이다. 아래와 같이 log 파일들을 잘 구성하였다.



NEM - 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

time	incoming_rate	forwarding_rate	avg_queue_utilization
2.001	1346.500	500.000	0.225
4.175	1317.000	500.000	0.263
6.375	1588.500	500.000	0.284
8.001	1179.500	500.000	0.268
10.224	1032.000	500.000	0.237
12.001	1275.500	500.000	0.278
14.001	1196.500	500.000	0.269
16.000	1517.000	500.000	0.283
18.001	1343.500	500.000	0.277
20.152	1298.500	500.000	0.280
22.000	1484.500	500.000	0.293
24.238	1512.000	500.000	0.284
26.000	1330.500	500.000	0.331
28.155	1303.500	500.000	0.281
30.001	1417.500	500.000	0.284
32.000	1305.000	500.000	0.336
34.000	1063.500	500.000	0.228
36.001	741.000	500.000	0.205
38.000	869.000	500.000	0.256
40.000	936.500	500.000	0.229
42.001	1198.000	500.000	0.354
44.000	929.500	500.000	0.202

2) In the case of multiple senders

A. Allow multiple senders fully utilize BLR

이는 “4)”의 FR 그래프를 보면 확실히 알 수 있다. BLR = 500 인 상황에서 FR = 500 으로 프로그램 시작시점부터 끝날 때까지 유지하였다.

B. Jain Fairness

이는 아래의 코드를 통해서 계산하였다. P_rec_array 에는 각 sender 주소마다의 2 초마다 받은 패킷 수의 정보가 들어있다. 이를 이용해서 Jain Fairness 식을 통해 값을 계산하였다.

추가적으로, 현재 통신하고 있는 sender 들의 주소도 p_rec_array 를 통해 확인할 수 있다. 이를 통해 2 초마다 각 주소마다의 receiving_rate 도 측정하였다.

```
if cur_time - start_time >= check_time:
    #print(s)
    rm_log = open(rm_name, 'a')
    _sum_pow = _sum = 0
    len_p_rec_array = 0
    for key in p_rec_array :
        if p_rec_array[key] != 0 :
            _sum_pow += (p_rec_array[key]/2) * (p_rec_array[key]/2)
            _sum += p_rec_array[key]/2
            len_p_rec_array += 1
    if _sum_pow == 0 or len_p_rec_array == 0 :
        jf = 0
    else :
        jf = (_sum * _sum)/(len_p_rec_array * _sum_pow)
    s = "%0.3f  |  " % (cur_time - start_time) + "%0.3f₩n" % jf
    rm_log.write(s)
    for key in p_rec_array :
        s = "₩t%s:%d₩t|₩t" % (key[0], key[1]) + "%0.3f₩n" % (p_rec_array
        rm_log.write(s)
        p_rec_array[key] = 0

    rm_log.close()

    check_time += 2
```

3) Bottleneck queue utilization is kept low (less than 30%)

Queue 의 utilization 은 Receiver 의 NEM 에서 관리하였다. 아래 코드는 “1)”에서 보여줬던 코드의 일부분으로, BLR 만큼 Forwarding 을 해버렸을 경우 Queue 에 공간이 있는지를 확인해서 만약 공간이 있으면 1/(queue 가 차있는 정도)^2 의 확률로 queue 에 넣었다. 이렇게 해서 queue utilization 을 30% 살짝 이하로 유지할 수 있었다.

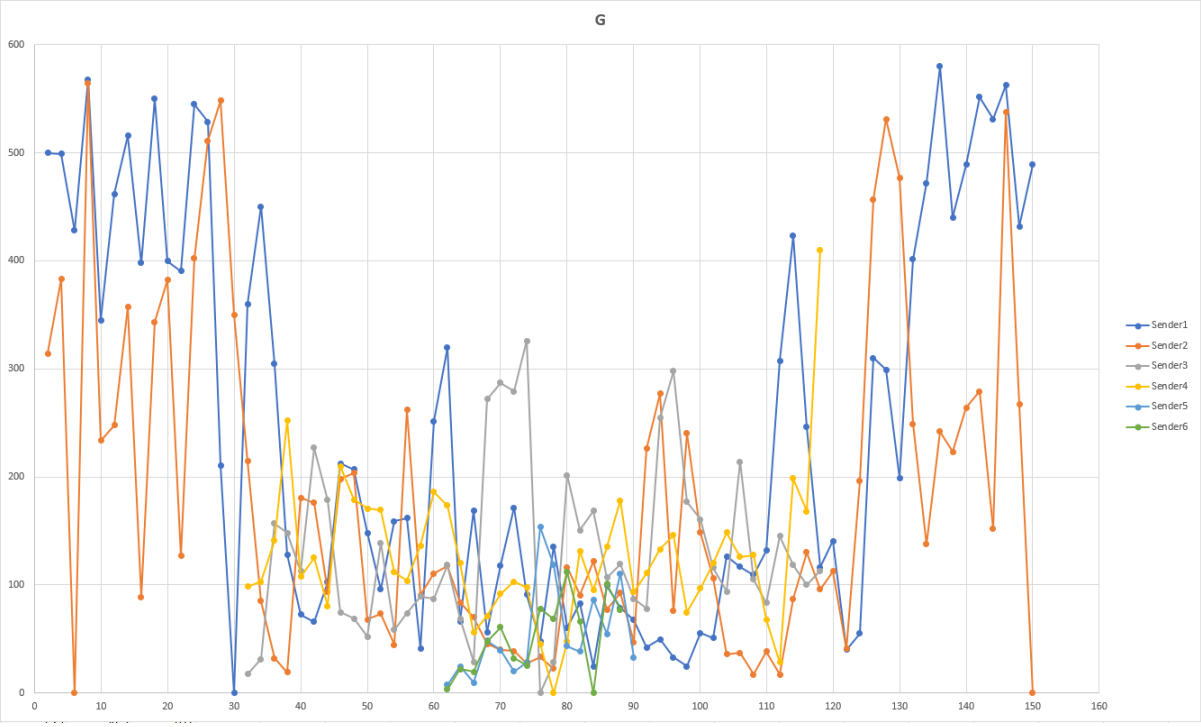
```
if forward_count - check_forward >= blr : # blr에 따른 제어
    if(q_fill < queue_size) :
        if q_fill == 0 :
            queue.append((senderAddress, r_seq, r_port))
            q_fill += 1
        elif random.random() < 1/(q_fill*q_fill) :
            queue.append((senderAddress, r_seq, r_port))
            q_fill += 1
```

4) The graphs of the experiment

과제에 나와있는대로 BLR = 500, queue_size = 50, initial_window_size = 5 의 상황에서 G, FR, queue utilization, Jain_fairness 를 측정하였다.

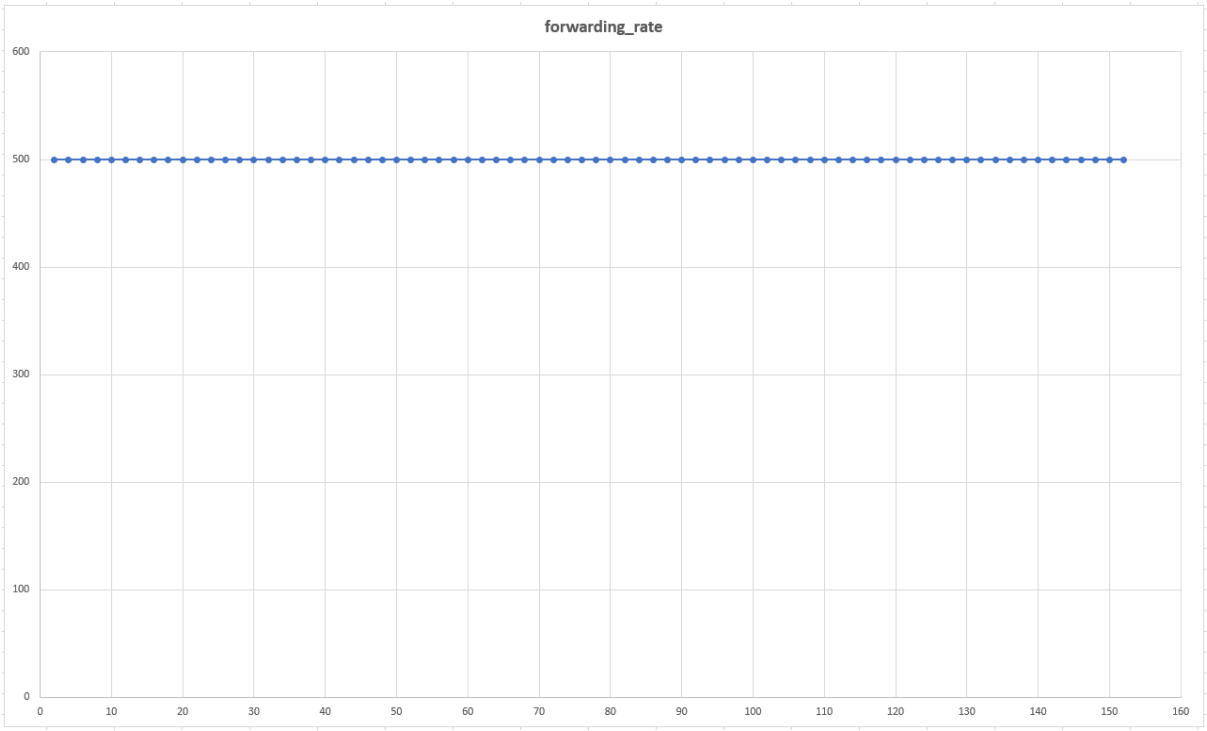
A. G

아래 그래프가 6 개의 sender 에 따른 G 그래프이다. Sender 가 추가될수록 각각의 Goodput 이 작아졌고, 6 개 모두 있을 때가 각각의 Goodput 이 가장 작은 것을 볼 수 있다. 기본적으로 Congestion control algorithm 을 AIMD 로 채택하였기 때문에 그래프가 톱날 모양으로 진행됨을 볼 수 있다.



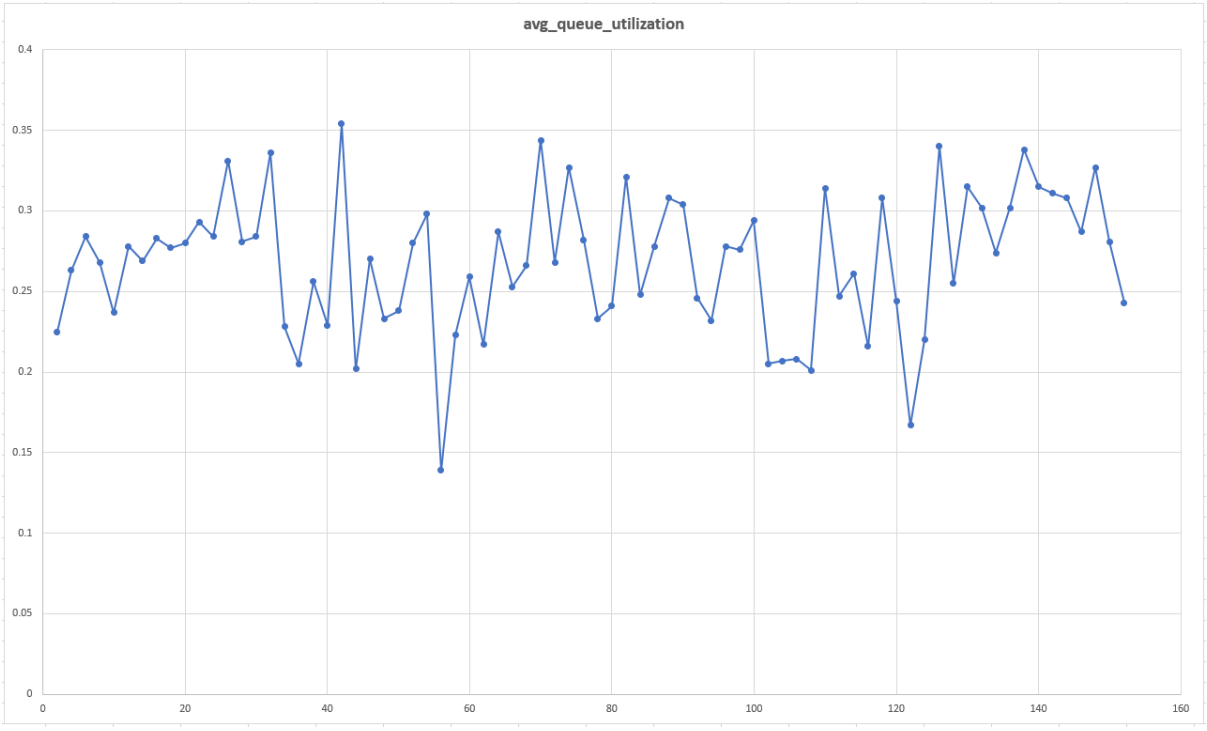
B. FR

아래 그래프는 FR 그래프이다. Forwarding rate 은 측정하는 내내 500 을 계속 유지하였다.



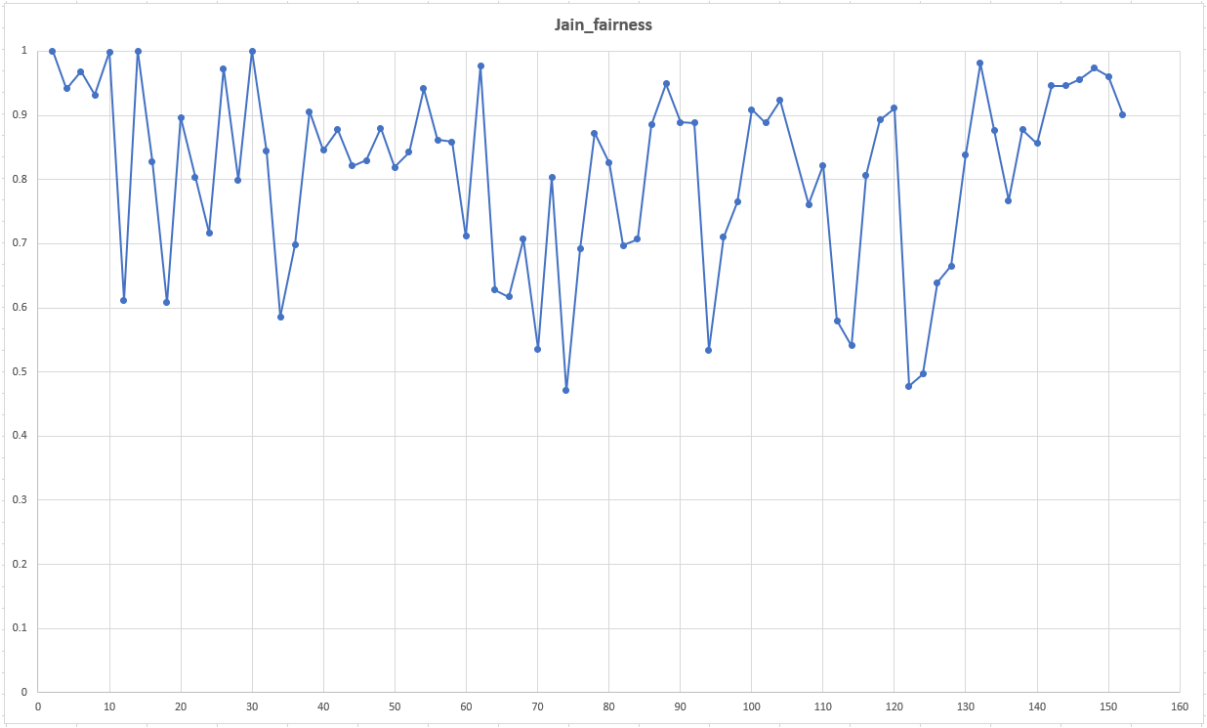
C. Queue utilization

아래 그래프는 Average queue utilization 그래프이다. 과제 조건 중에서 Queue utilization 을 30% 이하로 유지하라는 조건이 있었기에 그에 맞춰서 구현하였다. 30%를 넘기기도 하지만, 약 30%의 utilization 을 유지하기에 잘 구현하였다 할 수 있다.



D. Jain_Fairness

아래 그래프는 Jain Fariness 그래프이다. 150 초까지 측정 이후에 수작업으로 멈추는 과정에서 찍힌 점들은 제거하고 그래프를 만들었다. Jain fariness 값이 0.5 보다 크고 1 보다 작은 값을 유지하고 있다.



E. Data 값

그래프에 그릴 때 사용된 실제 데이터 값들이다.

- Sender 6 개의 Goodput 값

time	Sender1	Sender2	Sender3	Sender4	Sender5	Sender6
2.001	500	314				
4	499	383				
6.001	428	0				
8.001	568	564				
10	345	233.5				
12.001	462	248				
14.001	515.5	357				
16	398	89				
18.001	550	343.5				
20.001	399.5	382				
22	390.5	127				
24.001	545	402				
26	528.5	511				
28	210.5	548				
30.001	0	350				
32	360	214.5	17.5	98.5		
34	450	85	31.5	103		
36.001	304.5	32	157	141		

38	128	19	148	252		
40	72.5	180.5	112.5	107.5		
42.001	66	176	227	125		
44	102.5	94	178.5	80.5		
46.001	212.5	198	74.5	210		
48.001	207	203.5	69	179		
50	148	68	52	170.5		
52.001	96	73.5	139	169.5		
54.001	159	44.5	58.5	112		
56	162	262	74	104		
58.001	41.5	90.5	89.5	136.5		
60.001	251.5	110.5	87	186		
62	319.5	117.5	118.5	173.5	7.5	3.5
64.001	66.5	84	69	120	24.5	22
66.001	168.5	70	29	56	9.5	19.5
68	56.5	45.5	272	71.5	48.5	47.5
70.001	118	40	287	92	39.5	61
72	171.5	39	279	103	20	32
74	91	27.5	326	97.5	28.5	25.5
76.001	48	33.5	0	45.5	153.5	78
78	135.5	22.5	28.5	0	118.5	68.5
80	60.5	116.5	201.5	47.5	43.5	112
82.001	83	90.5	150.5	131.5	38.5	66.5
84	24.5	122	168.5	95	86.5	0
86	99.5	77	107	135	54.5	101
88.001	79	92.5	119.5	178	110.5	77
90	67.5	47	87	93.5	33	
92.001	42	226	78	111		
94.001	49.5	277	254.5	133		
96	33	76	298	146		
98.001	24.5	240.5	177	74.5		
100.001	55.5	148.5	160.5	97		
102	51	106.5	116	120		
104.001	126.5	36	94	148.5		
106.001	117	37	214	126		
108	109.5	17	105	127.5		
110.001	132	38.5	83.5	68		
112.001	307.5	17	145	28.5		
114	423	87	118.5	198.5		
116.001	246	130	100	168		
118	116.5	96	113	409.5		
120	140.5	112.5				
122.001	40	41				
124	55	196				
126	310	457				
128.001	299	531				
130	199	477				
132	401.5	249				
134.001	472	137.5				
136	580	242				
138.001	440	223				
140.001	489	264				
142	551.5	279				
144.001	531	152				
146.001	562.5	537.5				
148	432	267				
150.001	489	0				

- NEM.log 값

time	incoming_rate	forwarding_rate	avg_queue_utilization
2.001	1346.500	500.000	0.225
4.175	1317.000	500.000	0.263
6.375	1588.500	500.000	0.284
8.001	1179.500	500.000	0.268
10.224	1032.000	500.000	0.237
12.001	1275.500	500.000	0.278
14.001	1196.500	500.000	0.269
16.000	1517.000	500.000	0.283
18.001	1343.500	500.000	0.277
20.152	1298.500	500.000	0.280
22.000	1484.500	500.000	0.293
24.238	1512.000	500.000	0.284
26.000	1330.500	500.000	0.331
28.155	1303.500	500.000	0.281
30.001	1417.500	500.000	0.284
32.000	1305.000	500.000	0.336
34.000	1063.500	500.000	0.228
36.001	741.000	500.000	0.205
38.000	869.000	500.000	0.256
40.000	936.500	500.000	0.229
42.001	1198.000	500.000	0.354
44.000	929.500	500.000	0.202
46.001	1168.000	500.000	0.270
48.001	1062.500	500.000	0.233
50.000	935.000	500.000	0.238
52.001	906.500	500.000	0.280
54.001	1235.500	500.000	0.298
56.000	688.500	500.000	0.139
58.001	906.500	500.000	0.223
60.001	1133.000	500.000	0.259
62.004	900.500	500.000	0.217
64.001	1097.000	500.000	0.287
66.001	1002.500	500.000	0.253
68.000	930.000	500.000	0.266
70.001	1338.500	500.000	0.344
72.000	1053.000	500.000	0.268
74.000	1313.500	500.000	0.327
76.001	1076.000	500.000	0.282
78.000	976.500	500.000	0.233
80.000	891.500	500.000	0.241
82.001	1310.500	500.000	0.321
84.050	1142.500	500.000	0.248
86.000	1327.500	500.000	0.278
88.001	1190.000	500.000	0.308
90.000	1314.000	500.000	0.304
92.001	957.500	500.000	0.246
94.001	1067.000	500.000	0.232
96.000	961.000	500.000	0.278
98.016	1094.000	500.000	0.276
100.001	1133.000	500.000	0.294
102.000	779.500	500.000	0.205
104.001	845.500	500.000	0.207
106.001	791.000	500.000	0.208
108.000	905.000	500.000	0.201
110.001	1176.500	500.000	0.314
112.001	1193.500	500.000	0.247
114.000	932.500	500.000	0.261
116.001	860.000	500.000	0.216
118.000	1174.500	500.000	0.308
120.000	949.500	500.000	0.244
122.001	782.000	500.000	0.167
124.076	854.500	500.000	0.220
126.294	1471.500	500.000	0.340
128.001	1389.000	500.000	0.255
130.000	1528.000	500.000	0.315
132.000	1363.500	500.000	0.302
134.213	1230.000	500.000	0.274
136.000	1475.500	500.000	0.302
138.203	1556.500	500.000	0.338
140.001	1437.000	500.000	0.315
142.126	1568.500	500.000	0.311
144.331	1456.000	500.000	0.308
146.001	1379.500	500.000	0.287
148.201	1604.500	500.000	0.327
150.368	1458.000	500.000	0.281
152.045	1373.000	500.000	0.243

- RM.log 의 Jain Fairness 값

time	Jain_fairness
2.001	1
4	0.942
6.001	0.968
8.001	0.932
10	0.999
12.001	0.612
14.001	1
16	0.828
18.001	0.608
20.001	0.897
22	0.804
24.001	0.717
26	0.973
28	0.799
30.001	1
32	0.844
34	0.586
36.001	0.698
38	0.906
40	0.846
42.001	0.878
44	0.821
46.001	0.83
48.001	0.88
50	0.819
52.001	0.843
54.001	0.942
56	0.862
58.001	0.859
60.001	0.712
62	0.977
64.001	0.628
66.001	0.617
68	0.707
70.001	0.536
72	0.803
74	0.471
76.001	0.692
78	0.872
80	0.826
82.001	0.697
84	0.707
86	0.885
88.001	0.949
90	0.889
92.001	0.888
94.001	0.534
96	0.71
98.001	0.765
100.001	0.909
102	0.888
104.001	0.923
108	0.761
110.001	0.822
112.001	0.58
114	0.541
116.001	0.807
118	0.893
120	0.911
122.001	0.478
124	0.497
126	0.639
128.001	0.665
130	0.839
132	0.981
134.001	0.877
136	0.767
138.001	0.878
140.001	0.856
142	0.946
144.001	0.946
146.001	0.956
148	0.974
150.001	0.961
152.001	0.901

5) 코드 전반적인 설명

A. Sender

Sender 의 함수 정의를 제외한 실제 실행 부분은 아래와 같다. Receiver 로부터 ack 을 받는 getAck 을 먼저 쓰레드로 실행시키고, 이후 start 와 함께 window 크기를 입력해주면 메인 패킷 전송 부분인 fileSend 함수를 쓰레드로 실행시켜준다. 이는 이후 stop 을 입력해주기 전까지 계속 진행된다.

```
ack = 0
dup_ack = 0
stop = False
timeout = 1.00

receiverPort = 10080
packetsize = 1400
s_socket = socket(AF_INET, SOCK_DGRAM)
s_socket.setsockopt(SOL_SOCKET, SO_SNDBUF, 10000000)
s_socket.setsockopt(SOL_SOCKET, SO_RCVBUF, 10000000)
s_socket.bind(('',0))

receiverIP = input("Receiver IP address : ")

#getAck부터 먼저 시작
port_ack = 0
t2 = threading.Thread(target = getACK)
t2.deamon = True
t2.start()

while True :
    command = input("command>>")
    command = command.split(" ")
    if(command[0] == "start") :
        window = int(command[1])
        port_num = s_socket.getsockname()[1]
        t = threading.Thread(target = fileSend, args=("", receiverIP, receiverPo
        t.deamon = True
        t.start()
    elif command[0] == "stop" :
        stop = True

        #쓰레드 끝나기를 기다린다
        t.join()
        break
    else :
        print("Command error. Please check your command. ('stop' : exit)")

s_socket.close()
```

Sender 는 위에서 설명한 것 외에는 2 초마다 값들 측정하고 file write 해주는 것과 timeout, duplicated ack 체크해주는 것이 있다. Timeout, duplicated ack 을 관리하기 위해 여러 줄의 코드를 구성하였고, 이번 과제는 이에 큰 비중을 두지 않기에 설명은 생략하겠다. 사용자가 stop 을 입력 시에 프로그램을 끝내주기

위해서 각 함수는 “while stop != True :”라는 조건을 갖고서 루프를 계속 돌고 있다. 그러기에 사용자가 stop 을 입력해주는 stop = True 가 되고 프로그램은 종료될 수 있게 된다.

B. Receiver

Receiver 의 정의를 제외한 실제 실행 부분은 아래와 같다. NEM 역할을 하는 packetReceive 와 RM 역할을 하는 fileReceive 를 쓰레드로 실행시켜준다. Sender 와 마찬가지로 stop 을 입력하면 함수들을 루프를 종료시켜서 프로그램을 종료할 수 있게 하였다.

```
print("receiver program starts...")

#시간 측정 시작
start_time = time.time() + 100000 #NEM에서 받기 시작한 이후부터 시간 측정시작

#NEM
t1 = threading.Thread(target = packetReceive)
t1.deamon = True
t1.start()
#RM
t2 = threading.Thread(target = fileReceive)
t2.deamon = True
t2.start()

while True :
    command = input("프로그램을 끝내기 원한다면 stop을 입력해주세요.₩nconfigure>
    if command == "stop" :
        stop = True
        t2.join()
        break
```

NEM 역할을 하는 packetReceive() 함수에서는 incoming_rate, forwarding_rate, avg_queue_utilization 을 측정하는데, incoming_rate 은 아래 코드와 같이 패킷을 받을 때마다 p_count += 1 을 실행하여서 측정하였다.

```
#패킷 받는다.
message, senderAddress = r_socket.recvfrom( packetsize )
p_count += 1
```

Forwarding_rate 은 위에서 설명한 RM 으로 Forward 해주는 부분에서 카운트 해주는데, 보면 forward 해줄 때마다 Forward_count += 1 을 해주거나, queue 에 저장되있는 패킷들을 한 번에 forward 해줄 때에는 queue 의 특정 길이만큼 한 번에 forward_count 를 늘려준다.

```
else : # RM으로 보내지는 패킷들
    if q_fill > 0 :
        mailbox = mailbox + queue[:forward_count - check_forward]
        queue = queue[forward_count - check_forward:]
        q_fill = len(queue)
        forward_count += len(queue[:forward_count - check_forward])
        if(forward_count - check_forward < blr) :
            mailbox.append((senderAddress, r_seq, r_port))
            forward_count += 1
    else :
        mailbox.append((senderAddress, r_seq, r_port))
        forward_count += 1
```

Avg_queue_utilization 은 위에서 설명했던 queue 공간 관리방법 외에도 아래의 코드를 통해서 매번 100ms 마다 queue 가 얼마나 사용되고 있는지를 체크해서 리스트에 넣어준다. 이렇게 넣어준 값들을 2 초마다 평균을 내어서 측정하는 것이다.

```
if cur_time - ms100 >= 0.1 : # 100ms마다 실행
    if queue_size != 0 :
        queue_util.append((q_fill/queue_size))
    ms100 = time.time()
```

Sender 의 RM 역할을 하는 fileReceive() 함수는 크게 두 부분으로 구성 되어있다. NEM 으로부터 패킷을 받으면 패킷을 처리해서 ACK 을 보내주는 부분과 2 초마다 값들을 측정하는 부분으로 구성 되어있다. 패킷을 처리하는 부분에는 seq_array, p_rec_array 라는 두 개의 딕셔너리가 사용되고, seq_array 는 (senderIP, senderPort)를 키로 해서 받은 현재 받은 seq 상황을 저장해준다. 보내주는 port 는 sender 에서 패킷에 받는 port 번호도 저장해서 보내주는데, 이를 통해서 ack 을 받는 포트로 전송해줄 수 있다. p_rec_array 는 해당 주소로부터 받은 패킷 수를 저장해주는 것으로, 위에서 설명했던 것처럼 Jain Fairness 를 계산할 때 사용된다.