Assignment 3:

Pipelined Reliable Data Transfer over UDP

2017313008 김태은

1. Development environment

- Version of operating system: **Ubuntu 14.04.4 LTS mininet**
- Programming language: **Python**
- Interpreter version: **Python 3.4**

```
    popens = {}
    "If your code is python, uncomment this"

    popens[receiver] = receiver.popen('python3','receiver.py')
    popens[sender] = sender.popen('python3','sender.py',recvAddr, str(windowSize), srcFilename, dstF
ilename)
```

2. How to design

① sender.py

```
if __name__=='__main__':
    recvAddr = sys.argv[1]  #receiver IP address
    windowSize = int(sys.argv[2])   #window size
    srcFilename = sys.argv[3]   #source file name
    dstFilename = sys.argv[4]   #result file name
    recvPort=10080

    sendSocket=socket(AF_INET, SOCK_DGRAM)
    sendSocket.bind(('', 0))
    #print('sender program starts...')

    filePkt=[]
    f=open(srcFilename, "rb")
    data=f.read(1400)
    while True:
        filePkt.append(data)
        if len(data)<1400: break
        data=f.read(1400)

    pktNum=len(filePkt)
    pktNumstr='%20d'%(pktNum)

    log=open(srcFilename+"_sending_log.txt", "w")

    fileSender()

    log.close()
    sendSocket.close()
```

sender.py 가 실행되면 먼저 인자로 주어진 recvAddr, windowSize, srcFilename, dstFilename를 전달받는다. udp통신을 위한 소켓을 생성하고, 바인드한다. 여기서는 파일을 패킷으로 나누어 전송할 것이기 때문에 미리 filePkt 리스트에 순차적으로 저장해둔다. 또한 보내고 받은 기록을 하기 위한 log 파일을 생성하고, fileSender()호출을 통해 파일 전송을 시작한다.

```
def fileSender():


    seq='%20d'%(-100)
    sendSocket.sendto(seq.encode()+pktNumstr.encode()+dstFilename.encode(), (re
cvAddr, recvPort))
    sendSocket.sendto(seq.encode()+pktNumstr.encode()+dstFilename.encode(), (re
cvAddr, recvPort))
    sendSocket.sendto(seq.encode()+pktNumstr.encode()+dstFilename.encode(), (re
cvAddr, recvPort))


    sendThread=Thread(target=sendPkt, args=(sendSocket,))
    recvThread=Thread(target=recvAck, args=(sendSocket,))


    sendThread.start()
    recvThread.start()


    sendThread.join()
    recvThread.join()


    writeEnd(log,  pktNum/(time.time()-startTime), avgRTT*1000)
```

fileSender에서는 먼저 receiver에게 총 보낼 패킷이 몇 개인지, destination file name이 무엇인지 등을 알려준다. 그 후 sendThread, recvThread를 생성하고 시작한다. 마지막으로는 log파일에 goodput과 average RTT를 기록한다.

```
def sendPkt(sendSocket):

    global sending, startTime, sendTime, drop
    cnt=0
    sendTime=[0.0]*pktNum

    startTime=time.time()

    while True:
        if drop==True:
            continue
        while sending<windowSize:
            if drop==True: continue
            seq='%20d'%(cnt)
            sendSocket.sendto(seq.encode()+filePkt[cnt], (recvAddr, recvPort))

            sendTime[cnt]=time.time()
            writePkt(log, sendTime[cnt]-startTime, cnt, "sent")

            cnt+=1

            lock.acquire()
            sending+=1
            lock.release()


            if cnt==pktNum: return
```

<sendPkt 함수> 패킷을 send하는 thread에서 실행하는 함수이다. 처음으로 보내는 시간을 startTime으로 설정한다. drop변수는 현재 패킷이 drop되어서 packet recovery상태에 돌입했을 때는 더 이상의 in-order packet을 전송하면 안되므로 설정해준 변수이다. drop==true일 때는 지금 packet recovery상태이란 뜻이고, false는 in-order packet 전송가능한 상태이다. 여기서 보낼 때 windowSize만큼 보내야 하므로 global 변수 sending을 이용하여 전송을 조절하게 된다. 한 패킷을 보낼 때마다 sending은 1씩 증가하고 한 패킷이 ack될때마다 1씩 감소한다. 그러므로 window size에서 지금 추가적으로 더 보낼 수 있는 패킷 수를 알 수 있다. 패킷을 보낼 때는 sequence number와 파일 데이터를 함께 보낸다. 이 패킷의 처음 20개 자리는 sequence number 자리가 될 수 있도록 설정했다. 보낼 때마다 log 파일에 기록해두고, 만약 지금 보내는 패킷 (cnt)가 총 패킷 수(pktNum)만큼 다 보낸 상태이면 sendPkt함수를 종료한다.

```
def recvAck(sendSocket):

    global sending, avgRTT, drop, dropPkt, startTimer, timeout, ackSeq

    ackPkt=[0]*pktNum

    chk=0
    ackTime=0
    devRTT=0
    avgRTT=0
    timeout=1
    preSeq=-1


    singleTimer()

    while True:

        ackMsg, recvAddress=sendSocket.recvfrom(20)

        ackTime=time.time()
        ackSeq=int(ackMsg.decode())

        # dstFilename packet drop
        if ackSeq==-100:
            seq='%20d'%(-100)
            sendSocket.sendto(seq.encode()+pktNumstr.encode()+dstFilename.encod
e(), (recvAddr, recvPort))
            continue

        writeAck(log, ackTime-startTime, ackSeq, "received")
```

<recvPkt 함수> 패킷을 receive하는 thread에서 실행하는 함수이다. Timeout 감지를 위한 singleTimer()를 호출한다. 만약 ackSeq==-100이라는 뜻은 destination filename packet이 drop되었다는 것을 의미하므로 다시 한 번 보내주고 정상적인 패킷이 들어올때까지 기다린다.

```
if ackSeq==-1:
    seq='%20d'%(0)
    sendSocket.sendto(seq.encode()+filePkt[0], (recvAddr, recvPort))
    continue

ackPkt[ackSeq]+=1

if ackPkt[ackSeq]==1:
   if ackSeq>preSeq:
        lock.acquire()
        sending-=abs(ackSeq-preSeq)
        lock.release()


#print(preSeq, ackSeq)
#3 Duplicated ACKs
if ackPkt[ackSeq]==4:

    drop=True
    dropPkt=ackSeq+1

    writeDuplicate(log, time.time()-startTime, ackSeq)
    seq='%20d'%(ackSeq+1)
    sendSocket.sendto(seq.encode()+filePkt[ackSeq+1], (recvAddr, recvPort))
    sendTime[ackSeq+1]=time.time()
    writeRetransmit(log, sendTime[ackSeq+1]-startTime, ackSeq+1)


if ackPkt[ackSeq]>1:
    if ackSeq>preSeq:
        preSeq=ackSeq
    continue
```

ackSeq==-1이라는 뜻은 receiver side에서 가장 초기 패킷인 0번째 패킷이 드랍되었다는 것을 의미한다. 이 때는 드랍이 확실한 상황이므로 바로 0번째 패킷을 재전송하고 정상적인 패킷이 들어올 때까지 기다린다.

그 밑의 코드부터는 모두 정상적인 패킷이 들어왔을 경우에 실행되는 코드이다.
만약 어떤 sequence의 ack 패킷이 처음 들어온 패킷이면 sending을 감소시킨다. (이 때 ack가 순차적으로 들어오는 것이 아니므로 preSeq를 사용하여 이전패킷보다 sequence가 클 경우 그 차이만큼 빼준다.)

만약 어떤 sequence의 패킷이 네번째 들어온다면 3 duplicate acks이다. 따라서 이 때 drop 변수를 true로 바꾸고 packet recovery 모드에 진입한다. 또한 drop된 패킷은 dropPkt변수에 저장해둔다. 그리고 drop된 해당 패킷을 retransmit한다.

만약 어떤 sequence의 패킷이 2번 이상 들어온 상태이면 duplicate이므로 정확한 sample RTT를 알 수 없다. 따라서 averageRTT, timeout 계산으로 넘어가지 않고 continue를 통해 다음 패킷을 받는다.

```
if drop==False:
    # first RTT measurement
    if chk==0:
        avgRTT=ackTime-sendTime[ackSeq]
        devRTT=avgRTT/2
        chk=1
    # subsequent RTT measurement
    else:
        avgRTT=0.875*avgRTT+0.125*(ackTime-sendTime[ackSeq])
        devRTT=0.75*devRTT+0.25*abs(ackTime-sendTime[ackSeq]-avgRTT)
    #print(avgRTT)

# all packet is successfully transmitted
if ackSeq==pktNum-1:
    return

timeout=avgRTT+4*devRTT

# max of timeout: 60 secods
if timeout>60:
    timeout=60

# set timeout start -> oldest unacked packet
startTimer=ackSeq+1

if ackSeq>preSeq:
    preSeq=ackSeq

if drop==True and ackSeq>=dropPkt:
    drop=False
```

retransmit RTT나 duplicated acks는 avgRTT 계산에 반영하지 않는다. 따라서 drop==False, 즉 packet recovery 상태가 아닐 때만 sample RTT로 avgRTT를 계산한다. 표준 문서를 참고하여 가장 초기인 경우, 아닌 경우를 따로 분리해서 계산해준다.

> Taking RTT Samples
>
> TCP MUST use Karn's algorithm [KP87] for taking RTT samples. That is, RTT samples MUST NOT be made using segments that were retransmitted (and thus for which it is ambiguous whether the reply was for the first instance of the packet or a later instance). The

> (2.2) When the first RTT measurement R is made, the host MUST set
>
> SRTT <- R
> RTTVAR <- R/2
> RTO <- SRTT + max (G, K*RTTVAR)
>
> where K = 4.
>
> (2.3) When a subsequent RTT measurement R' is made, a host MUST set
>
> RTTVAR <- (1 - beta) * RTTVAR + beta * |SRTT - R'|
> SRTT <- (1 - alpha) * SRTT + alpha * R'

또한 여기서 들어온 ack packet의 sequence가 마지막 file packet 번호라면 함수를 종료한다.

Timeout의 max는 60초로 한다. (구글시트 참고)

어떤 sequence의 패킷이 ack 된 상황에서 timer를 가장 오래된 unacked packet의 sendTime으로 설정할 것이다. cumulative ack이므로 oldest unacked packet은 ackSeq+1이다.

Global 변수 drop을 통해 패킷이 drop되어 packet recovery 상태인지 아닌지 알 수 있다. 만약 packet recovery 상태에서 새로 들어온 패킷이 현재 drop된 packet(dropPkt)보다 크거나 같다면 accumulative ack이므로 drop된 패킷 loss가 해결되었음을 의미한다. 따라서 packet recovery 상태를 해제한다.

```
def singleTimer():

    global startTimer, timeout, ackSeq, drop, dropPkt


    timeoutnow=timeout
    if sendTime[startTimer]!=0 and time.time()-sendTime[startTimer]>timeoutnow:
        nowTime=time.time()

        drop=True
        dropPkt=startTimer

        writeTimeout(log, nowTime-startTime, startTimer, sendTime[startTimer]-s
tartTime, timeoutnow)
        seq='%20d'%(startTimer)
        sendSocket.sendto(seq.encode()+filePkt[startTimer], (recvAddr, recvPort
))
        sendTime[startTimer]=nowTime
        writeRetransmit(log, sendTime[startTimer]-startTime, startTimer)


    timer=Timer(0.0000000001, singleTimer)


    if ackSeq<pktNum-1: timer.start()
    else: return
```

<singleTimer 함수> 패킷을 처음으로 받기 직전에 실행을 시작하는 타이머 함수이다. startTimer은 현재 상황에서 oldest unacked packet num이다. 해당 num의 패킷이 send된 상황이고 현재시간에서 보낸 시간이 timeout 을 넘어간다면 timeout을 발생시킨다. 여기서도 drop=True로 설정해서 packet recovery상태에 진입한다. 그리고 드랍된 해당 패킷을 dropPkt에 저장한다. 그 후 drop된 패킷을 재전송한다.

```
def writeTimeout(logFile, procTime, ackNum, sendTime, timeout):
    logFile.write('{:1.3f} pkt: {} | timeout since {:1.3f}(timeout value {:1.3f})\n'.form
at(procTime, ackNum, sendTime, timeout))

def writeRetransmit(logFile, procTime, ackNum):
    logFile.write('{:1.3f} pkt: {} | retransmitted\n'.format(procTime, ackNum))

def writeDuplicate(logFile, procTime, ackNum):
    logFile.write('{:1.3f} pkt: {} | 3 duplicated ACKs\n'.format(procTime, ackNum))
```

위의 세 개 함수를 새로 만들어서 사용하였다.

② receiver.py

```
def fileReceiver():

    recvSocket=socket(AF_INET, SOCK_DGRAM)
    recvSocket.setsockopt(SOL_SOCKET, SO_SNDBUF, 100000000)
    recvSocket.bind(('', 10080))
    #print('receiver program starts...')

    filename=""
    pktNumint=0

    log=open("log.txt", "w") # after rename


    cnt=0
    chk=0
    init=0
    ackNum=0
    startTime=0
    ackPkt=[]
    bufPkt=[]
```

receiver.py 가 실행되면 fileReceiver 함수를 호출함으로써 패킷 receiving이 시작된다. 먼저 소켓을 생성하고 버퍼의 크기를 늘려주었다. log 파일은 먼저 “log.txt”로 생성한 뒤에 dstFilename packet이 들어온다면 “dstFilename_receiving_log.txt”로 rename 해줄 것이다. ackPkt은 어떤 sequence의 패킷이 들어왔는지 안 들어왔는지를 저장해주는 리스트이다. bufPkt은 in-order packet data가 저장되는 리스트이다.

```
while True:
    message, sendAddress=recvSocket.recvfrom(2048)
    seq=int(message[:20].decode())

    if seq==-100:
        if init==1: continue
        print(int(message[20:40].decode()))
        print(message[40:].decode())
        filename=message[40:].decode()
        pktNumint=int(message[20:40].decode())

        ackPkt=[0]*(pktNumint+1)
        bufPkt=[""]*(pktNumint)
        os.rename("log.txt", filename+"_receiving_log.txt")

        init=1
        continue

    # dstFilename packet drop
    if init==0:
        seq='%20d'%(-100)
        recvSocket.sendto(seq.encode(), sendAddress)
        recvSocket.sendto(seq.encode(), sendAddress)
        recvSocket.sendto(seq.encode(), sendAddress)
        continue

    if chk==0:
        startTime=time.time()
        chk=1
```

If seq==-100:는 destination File name이 담긴 패킷이 들어온 것이다. Filename, pktNumint를 저장해서 ackPkt, bufPkt을 pktNumint+1, pktNumint 크기만큼 초기화를 시켜주고, log 파일 이름도 바꿔준다.

If init==0:은 destination File name packet이 drop된 경우이다. 따라서 이 때는 sender에게 drop되었음을 알려준다.
If chk==0: 부터는 정상적인 패킷이 들어왔을 때 실행된다. 가장 처음으로 들어온 정상적인 패킷이라면 그 때를 startTime으로 설정한다.

```
    data=message[20:]
    writePkt(log, time.time()-startTime, seq, "received")

    if bufPkt[seq]=="":
        bufPkt[seq]=data


    if cnt!=seq:
        ackPkt[seq]=1
        for i in range(ackNum, pktNumint+1):
            if ackPkt[i]==0: break
        ackNum=i-1
        cnt=ackNum+1

    else:
        ackPkt[cnt]=1
        for i in range(ackNum, pktNumint+1):
            if ackPkt[i]==0: break
        ackNum=i-1
        cnt+=1


    ackNumstr='%20d'%(ackNum)
    recvSocket.sendto(ackNumstr.encode(), sendAddress)
    writeAck(log, time.time()-startTime, ackNum, "sent")
```

들어온 패킷의 20번째 바이트부터 파일 데이터가 들어있다. 만약 bufPkt의 seq번째 원소가 아직 비어있다면 데이터를 저장해준다.
cnt!=seq 일 때는 중간에 드랍된 패킷이 존재하여 while 문을 돌때마다 하나씩 증가하는 cnt와 들어온 패킷의 seq가 같지 않을 때이다. 이 때는 highest ack num을 for문을 통해 계산하여 ackNum에 저장한다.
cnt==seq일 때는 드랍된 패킷없이 순차적으로 잘 들어왔다는 뜻이고, 이 때도 앞과 비슷하게 highest ack num을 ackNum에 저장한다.

이 ackNum은 다시 sender로 보내져서 패킷이 드랍되었는지, 아닌지를 알려주게 된다.

```
    if ackNum==pktNumint-1:
        recvSocket.sendto(ackNumstr.encode(), sendAddress)
        recvSocket.sendto(ackNumstr.encode(), sendAddress)
        recvSocket.sendto(ackNumstr.encode(), sendAddress)

        f=open(filename, "wb")

        for i in range(0, pktNumint):
            f.write(bufPkt[i])
        break



writeEnd(log, pktNumint/(time.time()-startTime))

f.close()
log.close()
recvSocket.close()
```

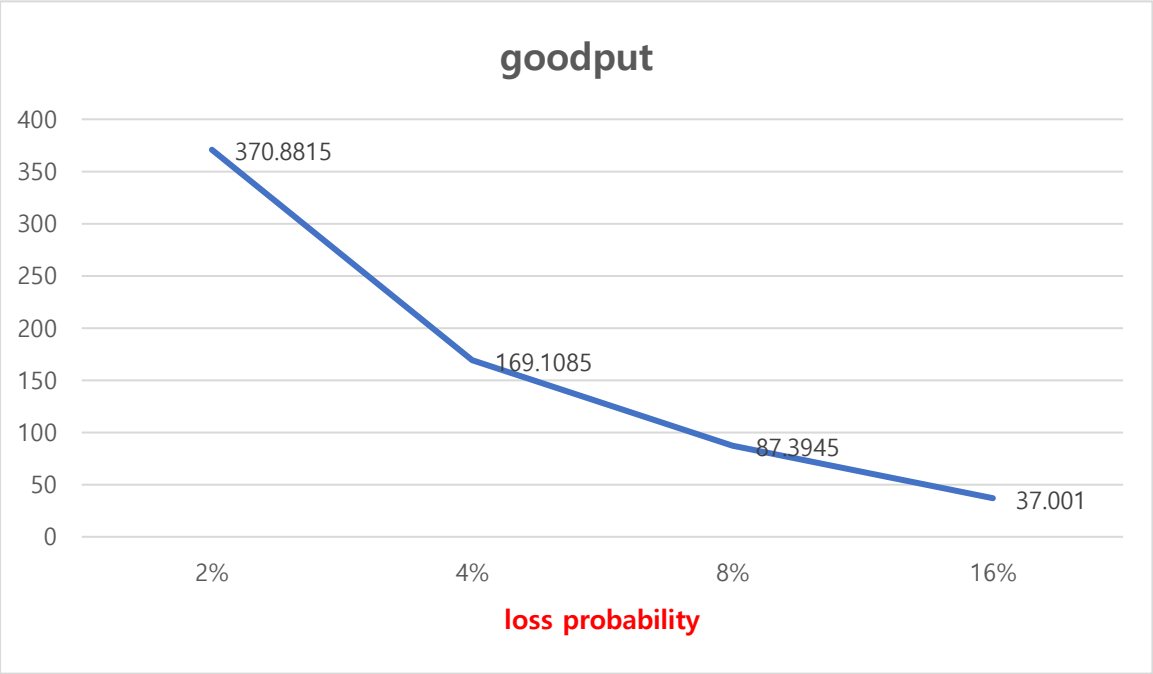
만약 모든 패킷이 다 정상적으로 들어온 상태라면 ackNum==pktNumint-1가 된다. 이 때 ackNum을 다시 sender로 전달함으로써 파일 전송이 끝난다는 것을 확실히 해주고, bufPkt에 저장된 데이터들을 destination file 에 저장한다.
마지막으로 goodput을 계산하여 log에 기록한다.

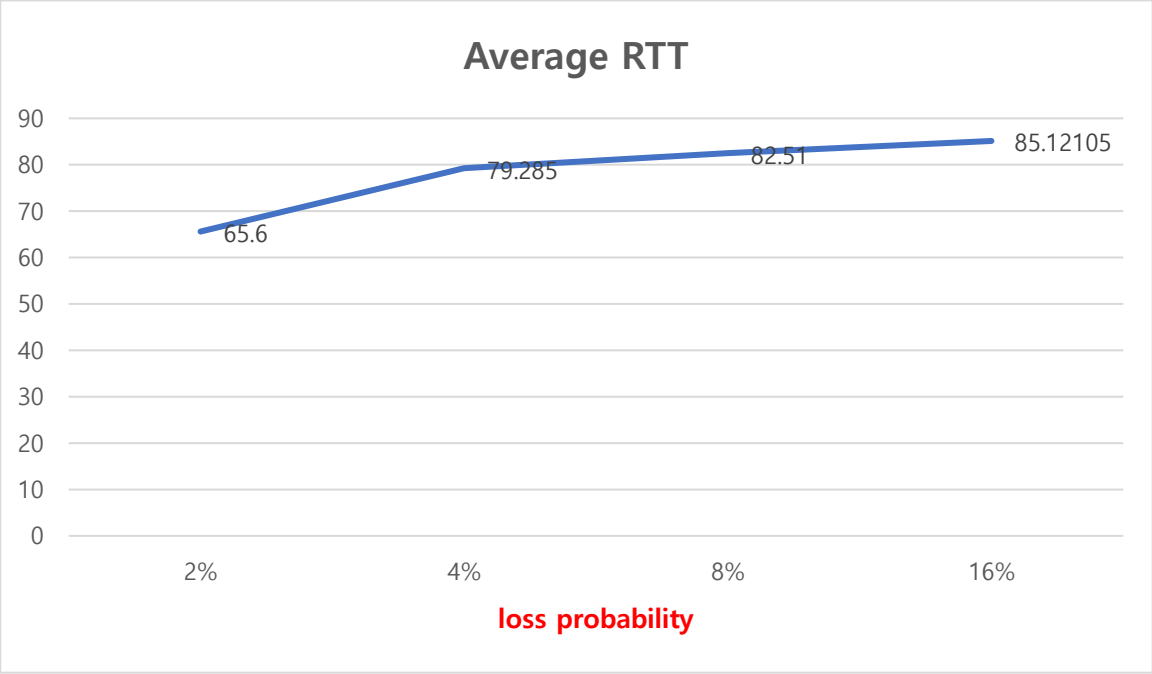
3. Experimentation Result

Testing file: image.jpg (13.1MB)

Experiment 1)

window size=40, bandwidth=10Mbps, one-way delay: 25ms





Experiment 2)

loss probability=2%, bandwidth=10Mbps, one-way delay: 25ms