HW3 Report

Computer Network
2013313217 조상연

1. Development Environments

1. Python Version: Python 3.4
2. IDE: VS Code
3. OS: Mac OS X 10.15.3

2. Design

3.1 사용한 모듈

```
import time
import socket
```

3.2 전체 흐름 및 설계

본 과제는 UDP 를 기반으로 하여 파일 전송을 하는 프로그램을 작성하는 것으로 TCP 의 주요 개념을 구현하는 것을 목표로 합니다. 이를 위해선 Sender 와 Receiver 가 서로 Scoket 통신을 하며 서로 Seq 와 ACK 를 주고 받아 전송 여부를 확인하고 다시 보내는 것을 반복해야 합니다.

```
def make_packet(_packet_raw):
    seq, fin, data = _packet_raw
    seq_byte = seq.to_bytes(4, byteorder='big')
    fin_byte = fin.to_bytes(1, byteorder='big')
    return seq_byte + fin_byte + data

def parse_packet(packet):
    seq = int.from_bytes(packet[:4], byteorder='big')
    fin = int.from_bytes(packet[4:5], byteorder='big')
    data = packet[5:]
    return seq, fin, data
```

서로의 패킷 구성을 통일시키기 위해 packet 을 만들고, 파싱하는 함수를 서로 같게 하였습니다. 이때 패킷 구성은 Seq (4B) + Fin (1B) + Data 로 하여 현재 Seq 와 끝 패킷인지 알 수 있도록 하였습니다. 이렇게 패킷을 주고 받으며 서로의 패킷이 모두 주고 받은 것이 확인이 되면 프로그램이 종료됩니다.

3.3 Sender

Sender 는 주요한 알고리즘이 구현되어있습니다. 그 중 특히 avgRTT 를 계산하고 이를 통해 timeout 기준 값을 계산해내는 것이 핵심적이며 이를 구현하기 위해 모든 seq 에 대해 보낸 시간을 dictonary 값에 담아 저장하고 ack 가 오면 해당 ack 를 보낸 시간을 탐색하고 여기서 rtt 를 구합니다. 그리고 아래 함수를 통해 새로운 avgRTT, devRTT 를 도출합니다.

```
def calculate_rtt(samplertt, avgRtt, devrtt):
    alpha = 0.125
    beta = 0.25
    avgRtt = (1-alpha) * avgRtt + (alpha * samplertt)
    devrtt = (1-beta) * devrtt + beta * abs(samplertt - avgRtt)
    return avgRtt, devrtt
avgRtt, devrtt = calculate_rtt(samplertt, avgRtt, devrtt)
TIMEOUT_VALUE = avgRtt + 4 * devrtt
```

이렇게 구한 TIMEOUT_VALUE 는 전역적으로 사용되며 특히 보낸시간을 저장하는 dictonary 에 함께 저장하여 추후 timeout 이 일어나는지 체크할 때 함께 사용합니다.

```
def send_packet(packet, context='sent'):
    global serverInfo, startTime, packetTimeDict, TIMEOUT_VALUE
    global SOCK

    seq, fin, _ = packet
    SOCK.sendto(make_packet(packet), serverInfo)
    packetTimeDict[seq] = [time.time(), TIMEOUT_VALUE]
    writePkt(seq, context)
```

packet 을 보내는 함수는 위 함수 하나로 통일하여 해당 함수 내에서 로그 파일 작성과 보낸 시간 및 당시 TIMEOUT_VALUE 저장을 하여 함수의 활용성을 높였습니다.

```
while prevAck < max_packets:
        # check timeout packet
        seq_timeout = checkTimeout()

        if seq_timeout:
            # print("Timeout!", seq_timeout)
            send_packet(packets[seq_timeout], "retransmitted")
            continue
```

```
        try:
            ack = get_recieved_pkt()
        except:
            continue
        # cumAck 확인


        if ack > prevAck:
            # 이전 ack와 같지 않은 경우
            # 이전 ack보다 큰 경우
            prevAck = ack
            start, _ = packetTimeDict.get(ack)
            samplertt = time.time() - start
            avgRtt, devrtt = calculate_rtt(samplertt, avgRtt, devrtt)
            TIMEOUT_VALUE = avgRtt + 4 * devrtt
            # print("SET TIMEOUT", TIMEOUT_VALUE)
            SOCK.settimeout(TIMEOUT_VALUE)
            ackDupCount = 0

            if nextSeq <= max_packets:
                # print("transfer", nextSeq)
                send_packet(packets[nextSeq])
                nextSeq += 1
        elif prevAck == ack:
            # 이전 ack랑 같은 경우 (중복)
            ackDupCount += 1
        else:
            # 이전 ack 보다 작은 경우
            # 일단 무시
            pass
```

위는 핵심 코드로 먼저 checkout 된 패킷이 있는지 검사한 후 그에 맞게 패킷을 보냅니다. 그런 후 패킷을 기다리게 되는데 이때 timeout 이 발생할 수 있으므로 try except 로 묵었습니다. 그런 후 받은 패킷이 이전 패킷보다 큰 경우 새로 RTT 를 갱신 하고 다음 보낼 패킷을 보내게 됩니다. 만약 이전과 같다면 중복값을 늘립니다. 또한 mini net 환경에선 딜레이가 발생하여서 인지 숫자가 작은 ACK 가 나중에 오는 경우가 생기는데 이 경우는 무시합니다. 이런 다음 받은 ack 보다 작거나 같은 값에 대하여 timeout 에 대한 후보 리스트에서 제외 합니다.

```
if ackDupCount == 3:
            # print("Duplicate!", ack)
            writePkt(ack, "3 duplicated ACKs")
            send_packet(packets[ack+1], "retransmitted")
            # print("retransmitted")
            while True:
                # seq_timeout = checkTimeout()

                # if seq_timeout:
```

```
                    # print("loop Timeout!", seq_timeout)
                #     send_packet(packets[seq_timeout])
                #     continue
                try:
                    ack = get_recieved_pkt()
                except:
                    send_packet(packets[ack+1], "retransmitted")
                    continue

                if ack > prevAck:
                    prevAck = ack
                    start, _ = packetTimeDict.get(ack)
                    samplertt = time.time() - start
                    avgRtt, devrtt = calculate_rtt(samplertt, avgRtt, devrtt)
                    TIMEOUT_VALUE = avgRtt + 4 * devrtt
                    # print("loop SET TIMEOUT", TIMEOUT_VALUE)
                    SOCK.settimeout(TIMEOUT_VALUE)

                    seq_arr = []
                    for kseq, vals in packetTimeDict.items():
                        if kseq <= ack:
                            seq_arr.append(kseq)

                    for dseq in seq_arr:
                        del packetTimeDict[dseq]

                    ackDupCount = 0
                    sent_as_window(packets, windowSize, offset=ack+1)
                    nextSeq = ack + windowSize + 1
                    break

                else:
                    # send_packet(packets[ack+1], "retransmitted")
                    continue
```

위 코드는 3 번 중복 ACK 가 발생했을 경우 어떻게 핸들링할지를 구현한 것으로 정상적인 패킷이 올 때까지 다른 패킷이 오는 것을 무시하고 정상적인 패킷이 온다면 그에 맞게 다음 패킷을 window size 에 맞게 전송합니다.

3.3 Receiver

```
while True:
        packet, cAddr = sock.recvfrom(MAX_SIZE)

        if startFlag:
```

```
            startTime = time.time()
            startFlag = False

        seq, fin, data = parse_packet(packet)
        # print("Seq ", seq, fin)

        if seq == 0:
            dest_filename = data.decode('utf-8')
            logFile = log_file_init(dest_filename)
        else:
            buffers[seq] = data

        writePkt(logFile, seq, "received")

        seq_nums[seq] = 1

        if seq - cumAck == 1:
            cumAck = max_cum_ack(seq, seq_nums)

        ack_packet = make_packet(cumAck,0,b'')
        sock.sendto(ack_packet,cAddr)
        writeAck(logFile, cumAck, "sent")

        if fin:
            last_seq = seq
            finFlag = True
            if last_seq+1 == sum(seq_nums[:last_seq+1]):
                break
            # fin 은 왔는데 아직 다 안도착한 경우
            continue

        if finFlag and last_seq+1 == sum(seq_nums[:last_seq+1]):
            break
```

Receiver 의 경우 Sender 보다는 단순하지만 중요한 점은 cumAck 를 잘 전달하는 가에 있습니다. 이는 선형적으로 찾는 방법을 택했으며 데이터가 매우 크지 않는 이상 (100MB 이상) 정상적으로 동작합니다.

3.4 Result

```
70.541 ACK: 7268 | received

File transfer is finished.
Throughput : 103.05 pkts/sec
Average RTT : 12.5 ms
```

위 처럼 Throughput 과 Average RTT 가 정상적으로 log 로 남겨지는 것을 확인할 수 있습니다.

```
0.057 ACK: 1 | received
0.057 ACK: 2 | received
0.057 pkt: 34 | sent
0.057 ACK: 2 | received
0.057 ACK: 2 | received
0.058 ACK: 2 | received
0.058 pkt: 2 | 3 duplicated ACKs
0.058 pkt: 3 | retransmitted
0.060 ACK: 2 | received
0.062 ACK: 2 | received
0.062 ACK: 2 | received
0.063 ACK: 2 | received
0.065 ACK: 2 | received
0.065 ACK: 2 | received
0.067 ACK: 2 | received
0.068 ACK: 2 | received
0.069 ACK: 2 | received
0.070 ACK: 2 | received
0.071 ACK: 2 | received
0.073 ACK: 2 | received
0.074 ACK: 2 | received
0.075 ACK: 2 | received
0.076 ACK: 2 | received
0.077 ACK: 2 | received
0.078 ACK: 2 | received
0.079 ACK: 2 | received
0.081 ACK: 2 | received
0.082 ACK: 2 | received
0.083 ACK: 2 | received
0.084 ACK: 2 | received
0.085 ACK: 2 | received
0.087 ACK: 2 | received
0.107 ACK: 2 | received
0.107 ACK: 2 | received
0.108 ACK: 2 | received
0.109 ACK: 34 | received
0.109 pkt: 35 | sent
0.109 pkt: 36 | sent
```

위 케이스는 34 까지 보내는 와중에 pkt 2 에서 ack 2 에서 duplicate 가 발생하였고 이에 따라 retransmitted 하는 과정을 보여주고 있습니다. 그 이후로 도착하는 ack2 를 무시하고 ack 34 가 정상적으로 도착하자 그 이후의 packet 을 보내는 것을 알 수 있습니다.

```
1.095 pkt: 229 | timeout since 1.058(timeout value 0.021)
1.095 pkt: 229 | retransmitted
1.095 pkt: 230 | timeout since 1.066(timeout value 0.023)
1.095 pkt: 230 | retransmitted
1.095 ACK: 215 | received
1.095 pkt: 215 | 3 duplicated ACKs
1.095 pkt: 216 | retransmitted
1.096 ACK: 215 | received
1.097 ACK: 215 | received
```

위 케이스는 pkt 229 가 1.095 초에 timeout 이 되었고, 이는 처음 보낸 1.058 초에 timeout value 0.021 를 합한 값보다 큼을 알 수 있습니다. 그리고 아래 230 에서도 timeout 발생했는데 이땐 timeout value 가 0.023 으로 약간 커져 Average RTT 를 통해 값이 조정되었음을 알 수 있습니다.

3. Experiment

Mininet 에서 Loss Rate, Window Size 2 가지 파라미터에 대해 여러 변화를 주고 이에 따른 Avg RTT 와 Throughput 값을 측정하였습니다. 이때 각 케이스 별로 10 번 정도의 테스트를 하였고 평균을 내어 아래와 같이 시각화를 하였습니다. 파일 용량은 10MB 이며 이미지 파일을 기준으로 하였고 전송 후 해당 파일이 정상적으로 열리는지 확인하였습니다.

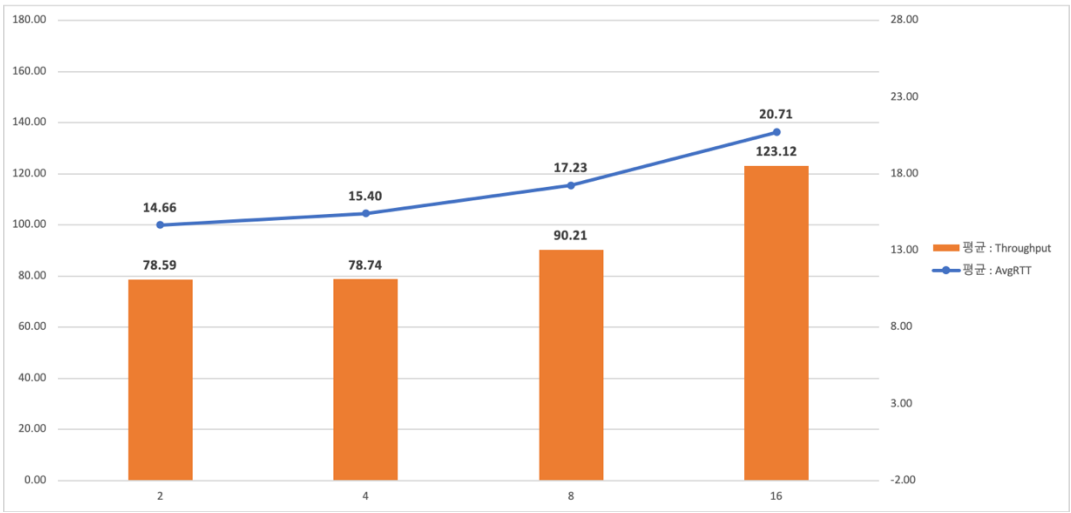

그림 1 Loss Rate 별 평균 Avg RTT, Throughput

위 그래프를 보면 Loss Rate 가 증가할 수록 평균 Avg RTT 가 확실히 증가함을 알 수 있습니다. 이는 중간에 유실되는 패킷이 증가함에 따른 결과로 보여집니다.

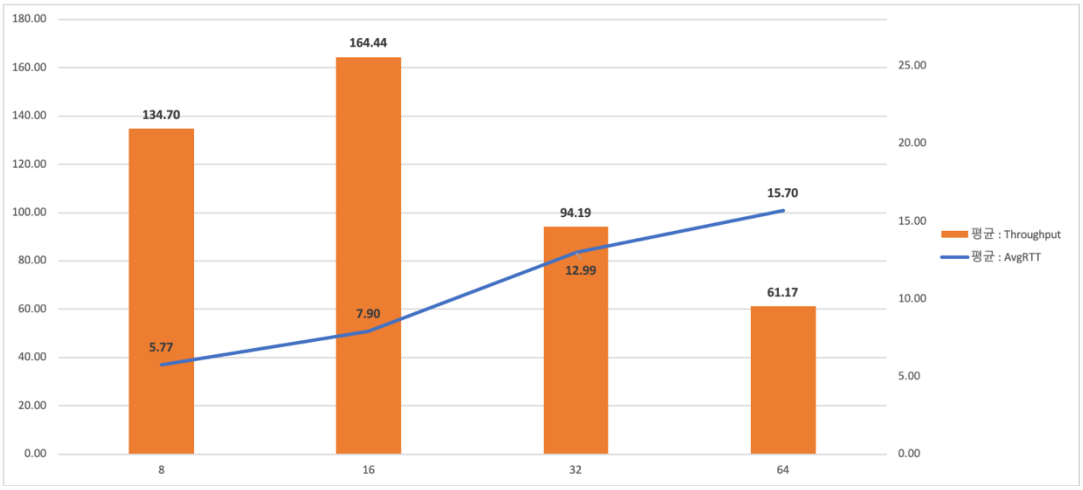

그림 2 Window Size 별 평균 Avg RTT, Throughput

위 그래프를 통해 window size 가 증가함에 따라 Avg RTT 는 증가하지만 Throughput 은 줄어드는 것을 확인할 수 있습니다.