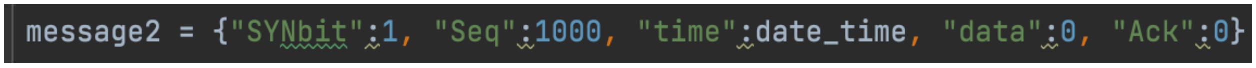
Report

1. PTP Protocol

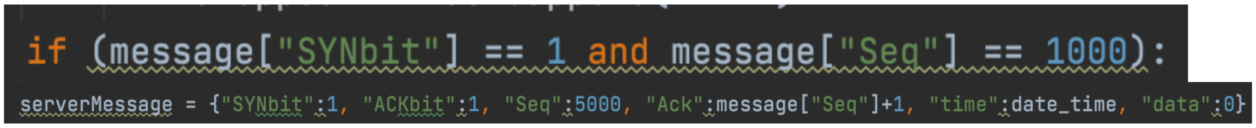
My UDPClient3.py and UDPServer3.py is trying to simulate the process of downloading file from the server to the client.

• 3-way handshake, connection establishment



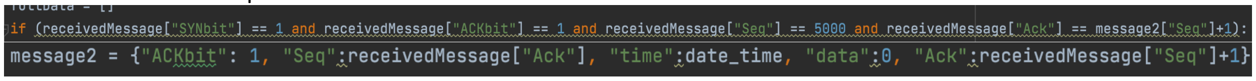
```
message2 = {"SYNbit":1, "Seq":1000, "time":date_time, "data":0, "Ack":0}
```

Due to decide to download the file, client send a request by using ‘SYNbit = 1, Seq = 1000’ from the client side to the server side.



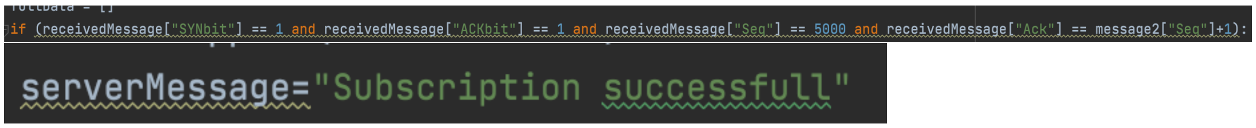
```
if (message["SYNbit"] == 1 and message["Seq"] == 1000):
serverMessage = {"SYNbit":1, "ACKbit":1, "Seq":5000, "Ack":message["Seq"]+1, "time":date_time, "data":0}
```

After the receiver in server checked ‘SYNbit’ value, the server sends the ‘SYNbit = 1, ACKbit = 1, Seq = 5000, Acknum = 1001’ as response.



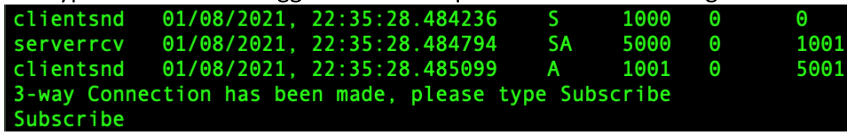
```
if (receivedMessage["SYNbit"] == 1 and receivedMessage["ACKbit"] == 1 and receivedMessage["Seq"] == 5000 and receivedMessage["Ack"] == message2["Seq"]+1):
message2 = {"ACKbit": 1, "Seq":receivedMessage["Ack"], "time":date_time, "data":0, "Ack":receivedMessage["Seq"]+1}
```

After the client checked ‘SYNbit’, ‘ACKbit’ values, the client reply the server’s receiver again with ‘ACKbit = 1’.



```
if (receivedMessage["SYNbit"] == 1 and receivedMessage["ACKbit"] == 1 and receivedMessage["Seq"] == 5000 and receivedMessage["Ack"] == message2["Seq"]+1):
serverMessage="Subscription successfull"
```

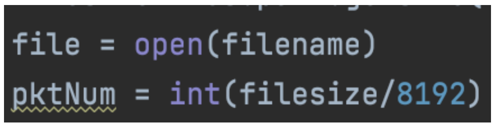
After the receiver in server checked ‘ACKbit’ value, the connection has been made successful. The client and type ‘Subscribe’ to trigger the ‘Subscription successful’ message to continuously download the file later.



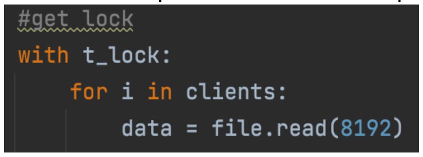
```
clientsnd   01/08/2021, 22:35:28.484236    S     1000   0      0
serverrcv   01/08/2021, 22:35:28.484794    SA    5000   0      1001
clientsnd   01/08/2021, 22:35:28.485099    A     1001   0      5001
3-way Connection has been made, please type Subscribe
Subscribe
```

• File Transfer

The full file has been divided into several packets or segments and convert each of them from the server’s sender to the client side one by one. After the client got the packet, it provides the feedback to the server’s receiver and the server’s sender sends the next packet to the client side. This process has been repeated until all the packets has been converted from the server to the client successfully and the client eventually has the full version of file. Then, the file transfer process has been completed.

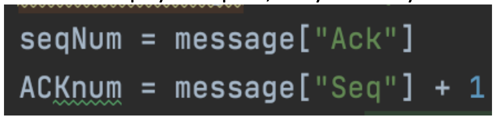


```
file = open(filename)
pktNum = int(filesize/8192)
```



```
#get lock
with t_lock:
    for i in clients:
        data = file.read(8192)
```

8192 is the payload part, only 8192 bytes of can be read and transferred in sender each time.



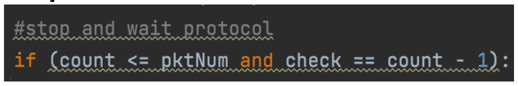
```
seqNum = message["Ack"]
ACKnum = message["Seq"] + 1
```

Initialize the first packet’s ‘seqNum’ and ‘ACKnum’ from the sender to the client, and the ACKnum should be incremented by 1 of the third-way handshake’s sequence number, which is 1002, because the packet is sent from the server side to the client, and it’s a downloading process not an uploading process in my simulation.

• Rule of setting values for ‘seqNum’, ‘ACKnum’, and data size in different modes

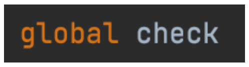
The rule between ‘serversnd’ and ‘packetdrop’
The sequence number is incremented by the data size (8192 bytes), ACK number doesn’t change if the packet is dropped.
The rule between ‘packetdrop’’ and ‘serverrcv’
The sequence number and ACK number just switch their positions, and after switching, ACK number of ‘receiver’ is not incremented by data size because the packet is dropped and the data hasn’t been transferred to the client side. The data size of the sending ‘ACK’ process should always be zero.
The rule between ‘serversnd’ and ‘serverrcv’
The sequence number and ACK number switch their positions, and after switching, ACK number of ‘receiver’ is incremented by data size because the data has arrived client side.
The data size of the sending ‘ACK’ process should always be zero.
The rule between ‘serverrcv’ and ‘serversnd’
The sequence number and ACK number just switch their positions. The data size of the sending ‘Data’ process should always be 8192 bytes.

• Stop-and-wait Protocol

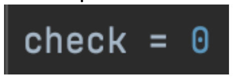


```
#stop and wait protocol
if (count <= pktNum and check == count - 1):
```

Try to imitate the rdt3.0 process, when the process of the previous packet has been transferred from the server’s sender to the client and the feedback of the client has reached the server’s receiver has been completely finished, the next packet will be considered to transfer.

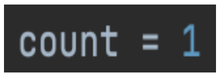


```
global check
```

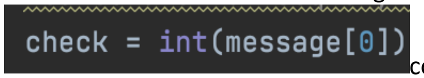


```
check = 0
```

check is a global variable and is initialize from 1

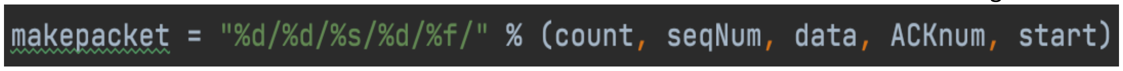


```
count = 1
```



```
check = int(message[0])
```

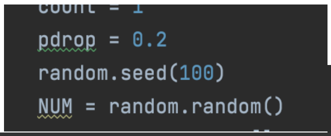
count starts from 1 in sender not global



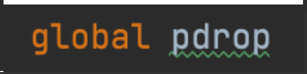
```
makepacket = "%d/%d/%s/%d/%f/" % (count, seqNum, data, ACKnum, start)
```

‘Count’ stands for the number of the packet to transfer, and it’s encapsulated in to the PTP header. The check value only get updated when ‘count’ reaches client and travels to receiver as ‘int(message[0])’. If the check value always keeps one less than count value, which means that last data segment transferred successful, you can start to send next one. When ‘count’ value is incremented to the same as ‘pktNum’, which stands for all the packets have been transferred, and the file is a full version downloaded.

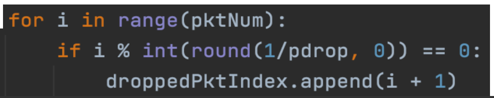
• The PL Module and Seed for random number generators



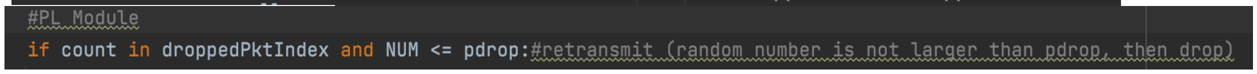
```
count = 1
pdrop = 0.2
random.seed(100)
NUM = random.random()
```



```
global pdrop
```

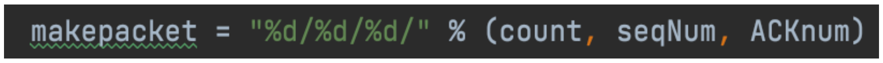


```
for i in range(pktNum):
    if i % int(round(1/pdrop, 0)) == 0:
        droppedPktIndex.append(i + 1)
```



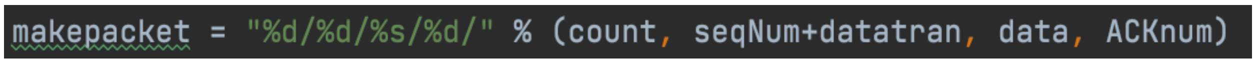
```
#PL Module
if count in droppedPktIndex and NUM <= pdrop:#retransmit (random number is not larger than pdrop, then drop)
```

Use seed to set random value for ‘NUM’. I choose ‘averagely dropping packets strategy’ and the third image is the algorithm designed by me to store the index of packet, which are decided to drop, into the ‘droppedPktIndex’ list. The length of ‘droppedPktIndex’ list is the number of packets dropped by PL module. If the No. of packet are going to be dropped and ‘NUM’ is not larger than ‘pdrop’, the data won’t reach the client side and it will be re-sent from the sender.



```
makepacket = "%d/%d/%d/" % (count, seqNum, ACKnum)
```

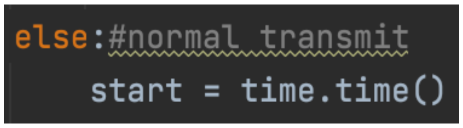
Before dropped, no data part in the segment.



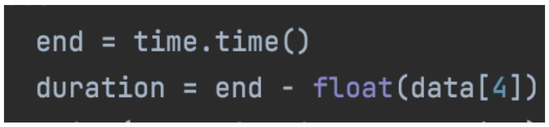
```
makepacket = "%d/%d/%s/%d/" % (count, seqNum+datatran, data, ACKnum)
```

After dropping and retransmission, the data part is back.

- **The timer and timeout**

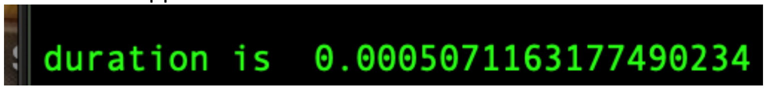


```
else:#normal transmit
    start = time.time()
```



```
end = time.time()
duration = end - float(data[4])
```

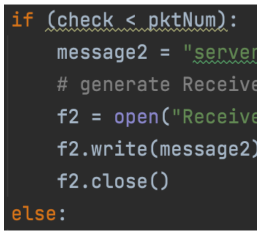
'start' is the time point in the server's sender before sending packets, 'end' is the time point in the client after receiving the data, the duration is the transferring time period in second. If the duration is less than the timeout value. Nothing change, the 8192 bytes data will be appended to the txt file. If it's larger than the timeout value, the 8192 bytes data won't be appended to the text file, the packet will be dropped. Same rules as PL Module, when the next packet is arrived or the same data retransmitted, this time the data will be appended to the text file.



```
duration is  0.0005071163177490234
```

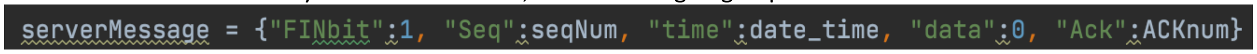
'Settimeout(0.0005)' to satisfy the most of packets get successfully transmission without dropping. And if the dropped packet happens, the No. of this packet should not be the same one dropped by PL module.

- **Connection Teardown or Combined 4-way Termination**



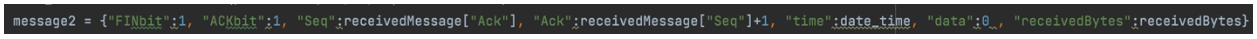
```
if (check < pktNum):
    message2 = "server
    # generate Receive
    f2 = open("Receive
    f2.write(message2)
    f2.close()
else:
```

When 'check == pktNum', the last packet's feedback got by the server's receiver. Same strategy as connection setup, when all the packets have been converted to the client side, which means the file has been fully downloaded. Then, the server is going to pass 'FINbit' to the client.



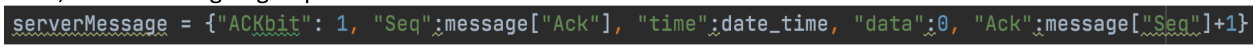
```
serverMessage = {"FINbit":1, "Seq":seqNum, "time":date_time, "data":0, "Ack":ACKnum}
```

Then, the client replies to the server with 'FINbit' and 'ACKbit'



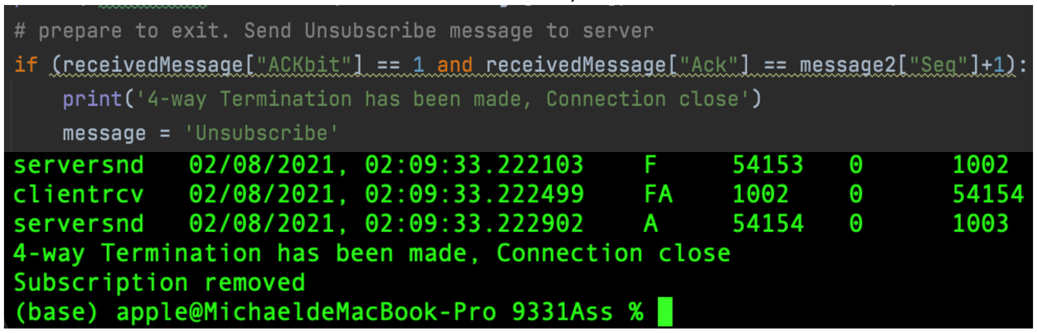
```
message2 = {"FINbit":1, "ACKbit":1, "Seq":receivedMessage["Ack"], "Ack":receivedMessage["Seq"]+1, "time":date_time, "data":0 , "receivedBytes":receivedBytes}
```

Then, the server is going to pass 'ACKbit' to the client.



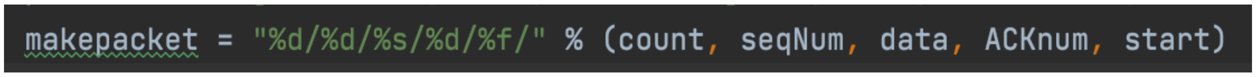
```
serverMessage = {"ACKbit": 1, "Seq":message["Ack"], "time":date_time, "data":0, "Ack":message["Seq"]+1}
```

When the client has checked received 'ACKbit' value, the connection is closed.



```
# prepare to exit. Send Unsubscribe message to server
if (receivedMessage["ACKbit"] == 1 and receivedMessage["Ack"] == message2["Seq"]+1):
    print('4-way Termination has been made, Connection close')
    message = 'Unsubscribe'
servers nd   02/08/2021, 02:09:33.222103    F     54153   0      1002
clientrcv    02/08/2021, 02:09:33.222499    FA    1002    0      54154
serversnd    02/08/2021, 02:09:33.222902    A     54154   0      1003
4-way Termination has been made, Connection close
Subscription removed
(base) apple@MichaeldeMacBook-Pro 9331Ass %
```

2. Diagram of PTP



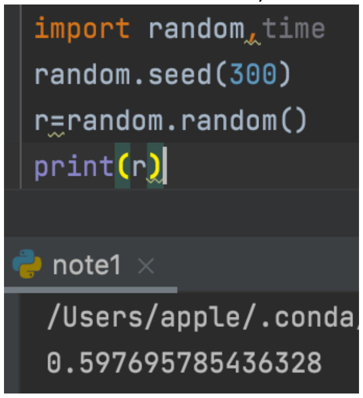
```
makepacket = "%d/%d/%s/%d/%f/" % (count, seqNum, data, ACKnum, start)
```

Source port					Destination port	
Sequence number						
Acknowledgement						
HdrLen	A	S	F	Count(Pkt Index)	Receive window	
Checksum					Start(sending time)	Options(variable length)
Data(Payload)						

Sequence number & Acknowledgement: counting by bytes of data
A: ACK # valid **S:** SYN **F:** FIN
Count: Packet Index (start from one, No.)
Receive window: # bytes 'rcv' will to be accepted
Start: Packet sending time
Checksum: same as UDP's, detect error.

3. Question

(a) The number of data transferred should be MWS/MSS=100 per time.
When seed = 300, random generate NUM is 0.597695785436328.

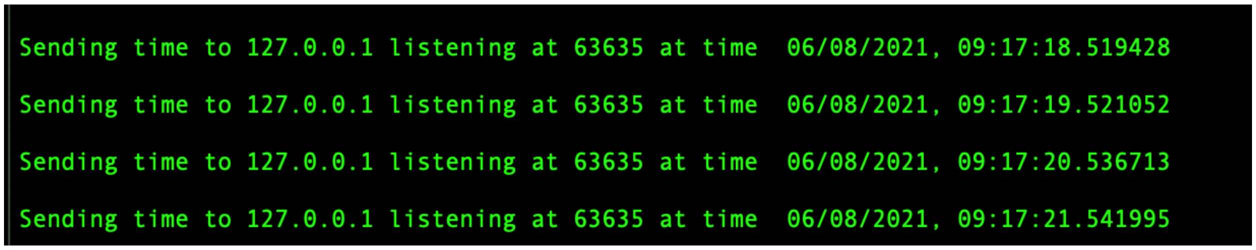


```
import random,time
random.seed(300)
r=random.random()
print(r)
```

note1

```
/Users/apple/.conda
0.597695785436328
```

when pdrop = 0.1 or 0.3, they're all smaller than random number. So, the packets are only dropped by timeout value not PL Module.

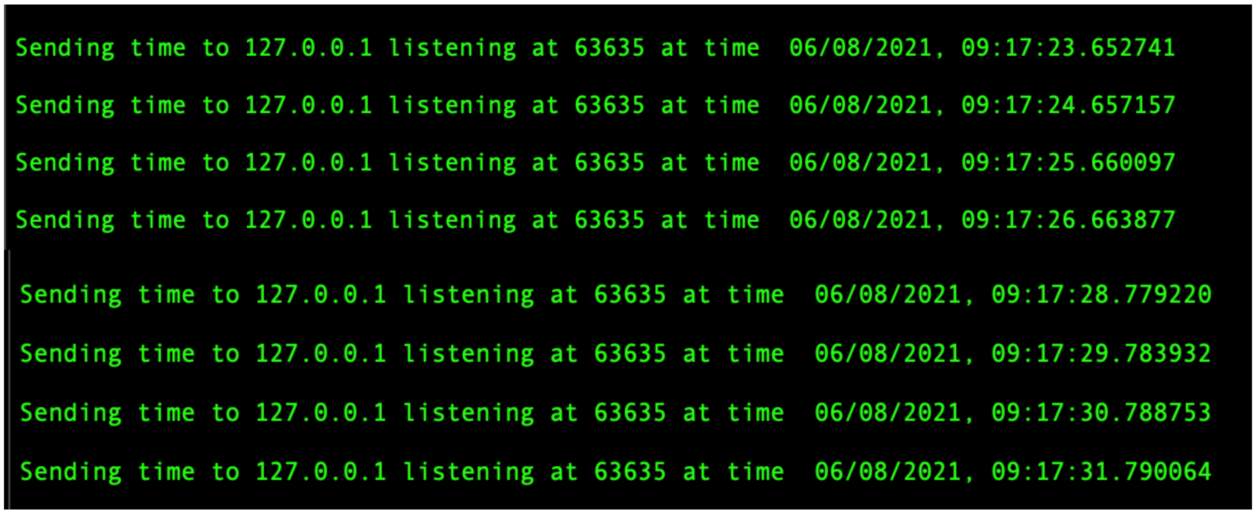


```
Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:18.519428

Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:19.521052

Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:20.536713

Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:21.541995
```


```
Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:23.652741
Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:24.657157
Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:25.660097
Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:26.663877
Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:28.779220
Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:29.783932
Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:30.788753
Sending time to 127.0.0.1 listening at 63635 at time  06/08/2021, 09:17:31.790064
```

The interval between each sending time is round 1-5ms, so the timeout value should not be set up for too small (enough large) in order to guarantee most the ACK can be sent back. If timeout value is too small, some packets are actually not lost, they just arrived at late, but they are regarded as 'lost' because of the small timeout value.