1. How I have implemented the STP protocol

   The feature I implemented:

   1. A three-way-handshake:

      Before sending Data of files, sender sends a SYN to inform receiver and wait for response from receiver. There is a function called "BuildUpConnection" to achieve this.
   2. A four-segment connection termination:

      After sending all bytes of the file, sender is going to send a FIN to indicating a termination of this connection, after receiving a FIN from receiver, sender will wait 4 second before close the socket. I write this logic just in "SendFile" function.
   3. A single timmer for timeout operation:

      I define a "timmer" function and create a new thread to run it during sending file. There are four segments(sampleRTT, EstimatedRTT, DevRTT, gamma) will determine the "timeInterval" which is a crucial factor causing timeout operation. When waiting time for a segment larger than timeInterval, it will trigger a timeout operation, the sender will send all unacknowledged segments again.
   4. Fast retransmit:

      If sender receive a ack which has been antique, a counter will remember this duplicated ack. When counter reaches 3, sender will be acknowledge this via (self.fastRetransmit variable), and send the oldest unacknowledged segment again.
   5. MSS and MWS:

      Sender will going to send segments until segment's sequence number minus sender's sendbase(which is the oldest unacknowledged segment's sequence number) is larger than MWS. In this case, sender will resend the all segments in buffer. All of the segments sent will store in this buffer(a python dictionary whose key is segment's sequence number), segment in buffer will be removed after acknowledged by receiver.
   6. Re-order packet receive:

      For receiver side, if a segment's sequence number is larger than expected sequence number, it will store it in a buffer(which is also a python dictionary). Every time receiver updates its expected sequence number, it will look up the buffer to see if there is a packet has received previously. Once it finds out an appropriate segment in buffer, it will remove segment from buffer and send ack immediately.
   7. Send duplicated Ack:

      Same as the feature above, if receiver receives a segment whose sequence number is antique, it would also send back an Ack to sender immediately. In another word, whatever an segment is antique or advanced, receiver would send duplicate ack if it receives an unexpected segment.
   8. Simplified TCP sender:

      After three-way handshake with receiver, sender will split the file into bytes, and store them into a python array. Each packet is size of MSS, if it is allowed from entire file(the size of file may not be the multiple of MSS). Once sender receives an Ack from receiver, as long as Ack is not antique, sender will update its send base. Sender is allowed to send as many packets as MWS allowed simultaneously.

   Feature I fail to implement:

   1. Delay acknowledgment:

For my receiver, every segment it received will generate an Ack immediately. Which means, even if the Ack is cumulative, it has no difference on sender's efficiency.

2. An appropriate timeout:
   I has implemented the a single timmer listening on sender's sending behaviour. However, there might be some thing wrong with calculating timeInterval, because for test2.pdf, transportation time is fairly small, and I have no idea how to figure it out.

2 My STP header

I pack up the whole segment in a python class and create a "Segment.py" python package to store it.

source port	Int
Destination port	Int
Sequence number	Int
Signal(indicating it is a FIN or SYN or ACK)	Str
Ack number	Int
Checksum	Int (value <= 16bits)
Payload	List

3 My design trade-off:

1. As I list my header fields above, because I use python to implement STP, header part could be fairly large.
2. I create different threads for sending and receiving, as a result, I have to maintain a buffer to segment storage, which occupied a large memory space.

Extension I could realize in the future:

1. Delay Ack:
   On my receiver side, once it receives a segment, it immediately send a ack to sender. Which means the number of ack is more than the number of sender. For delay ack, receiver will send ack until it find a gap or wait for 500 msec. To achieve this, receiver side can implement a extra "timmer" listening on receive segments
2. Flow control:
   On sender side, it will continuously send packets until it across MWS. However, this way is inefficient when bandwidth is limited. For flow control, sender is suppose to implement different ratio to send packet when RTT is various.

4 Segments of code borrowed from web or other book:

1. Python lib:
   Pickle: I use pickle.loads, pickle.dumps to implement a serialization on my packet
   Struct: I use struct.unpack to convert a byte to integer
2. Some logic:
   Simplified TCP and fast retransmission is the concept of text book

5 Answer the following questions

a)

When pDrop equal to 0.1, drop only occurs twice where sequence number are 201 and 2701. When pDrop raise to 0.3, drop occurrence also comes to 9 times. Besides sequence 201, sender drops packets whose sequence number are 401, 501, 1101, 1501, 2501 and 3001. In both experiment, reactions of sender and receiver is similar. After drop packet 201, sender continue to send segments until sequence number reaches 501 without receiving any Ack. So the data length is beyond MWS, so sender retransmit the segments in its buffer including 201.

b)

Gamma	Number of packets	Running time
2	11863	44.35 second
4	12022	43.81 second
6	11405	43.75 second

For my STP, with the gamma raising, the running time is less and less. Most of my time is used for dealing with retransmission issue. And a fairly large part of my transmission issue is caused by timeout. So when timeInterval is larger, my retransmission is triggered less.

c)

Result:

The file has been transferred successfully. The overall transfer took 211.71 seconds(There might be a problem on my timmer…).

According to sender_log, there are 3250 packets which was dropped that are more than other PLD factors caused. So I guess the drop occurrence would be the critical contribution

Without drop, transfer duration is reduced 196.36 seconds

Without duplicated occurrence, transfer duration is reduced to 206.65 seconds

Without corrupted occurrence, transfer duration is reduced to 193.43 seconds

Without reorder occurrence, transfer duration is reduced to 198.64 seconds

So, the critical factor is corrupted occurrence