Xiaoyu Xia
xix90@pitt.edu

homework 5

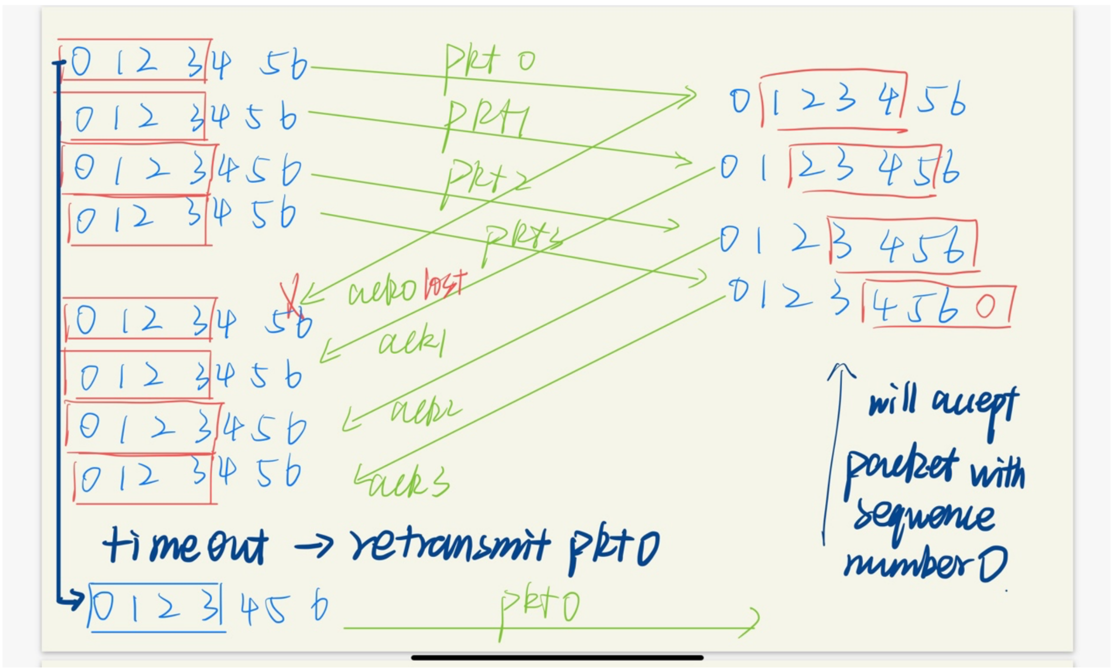
1.

(a)

Assume that the window size for both sender and receiver is 4, and the sequence space is less than 2*4=8(2N), which is 7.

Assume that the sender sends 4 packages successfully and the receiver has already received these four packages, so we can slide the window of the receiver forward and wait for new data. But when the receiver sends ack0 to the sender, it is lost, so the sender has to resend packet 0（timeout）so that the window for the sender can slide forward.

At this time, the receiver's receiving window has been updated, and the receiver cannot distinguish between new data frames and old data frames with sequence number 0, and ambiguity occurs.

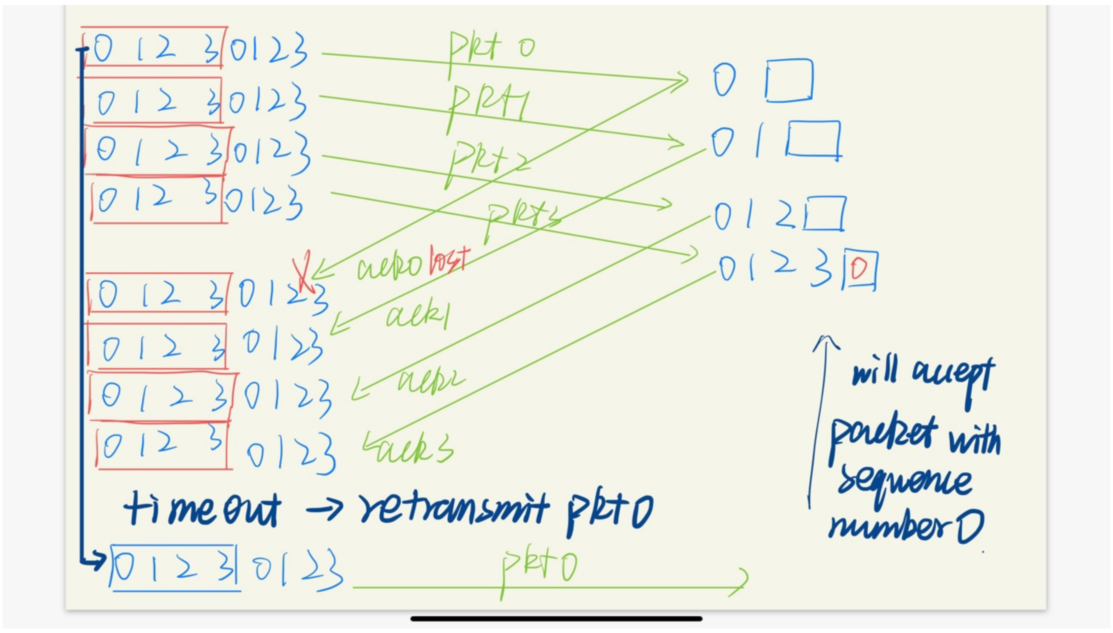

pkt 0
pkt1
pkt2
pkt3
ack0 lost
ack1
ack2
ack3
time out → retransmit pkt0
pkt0
will accept packet with sequence number 0

(b)

Assume that the window size for sender and receiver is 4, and the sequence space is less than 4+1=5(N+1), which is 4.

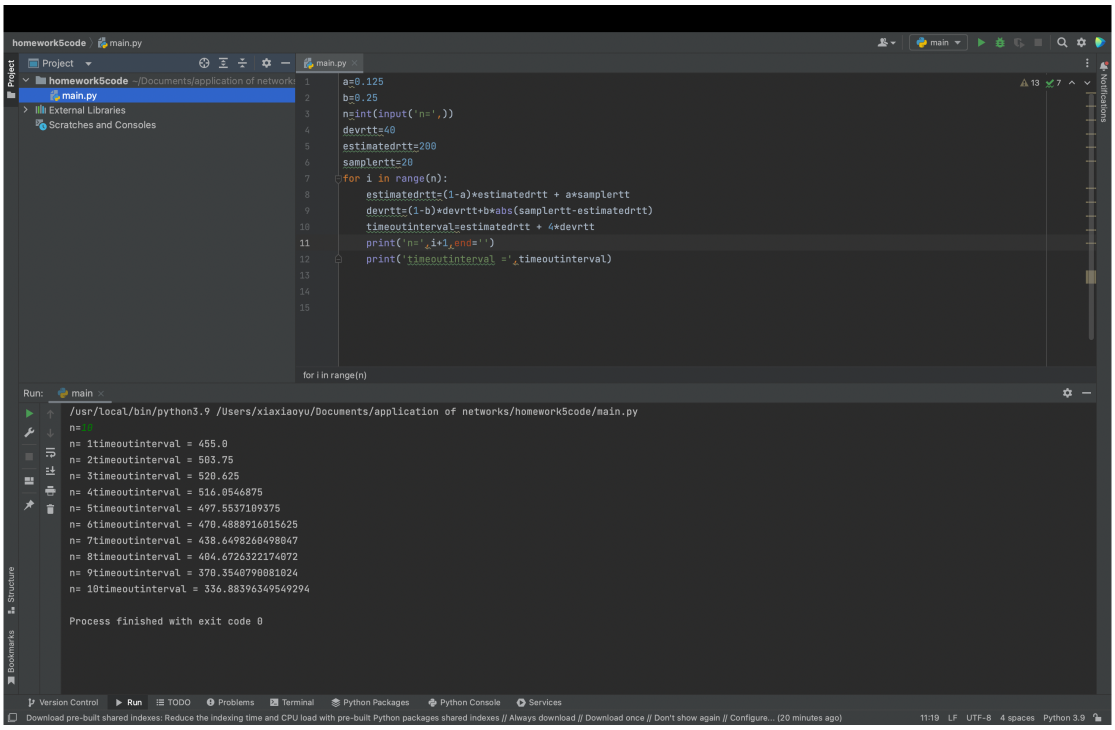
Assume that the sender sends 4 packages successfully and the receiver has already received these four packages one by one. But when the receiver sends ack0 to the sender, it is lost, so the sender has to resend packet 0（timeout） so that the window for the sender can slide forward.

At this time, the receiver cannot distinguish between new data frames and old data frames with sequence number 0, and ambiguity occurs.



2.

(1) the timeout interval after 10 RTT measurements is 336.884ms.



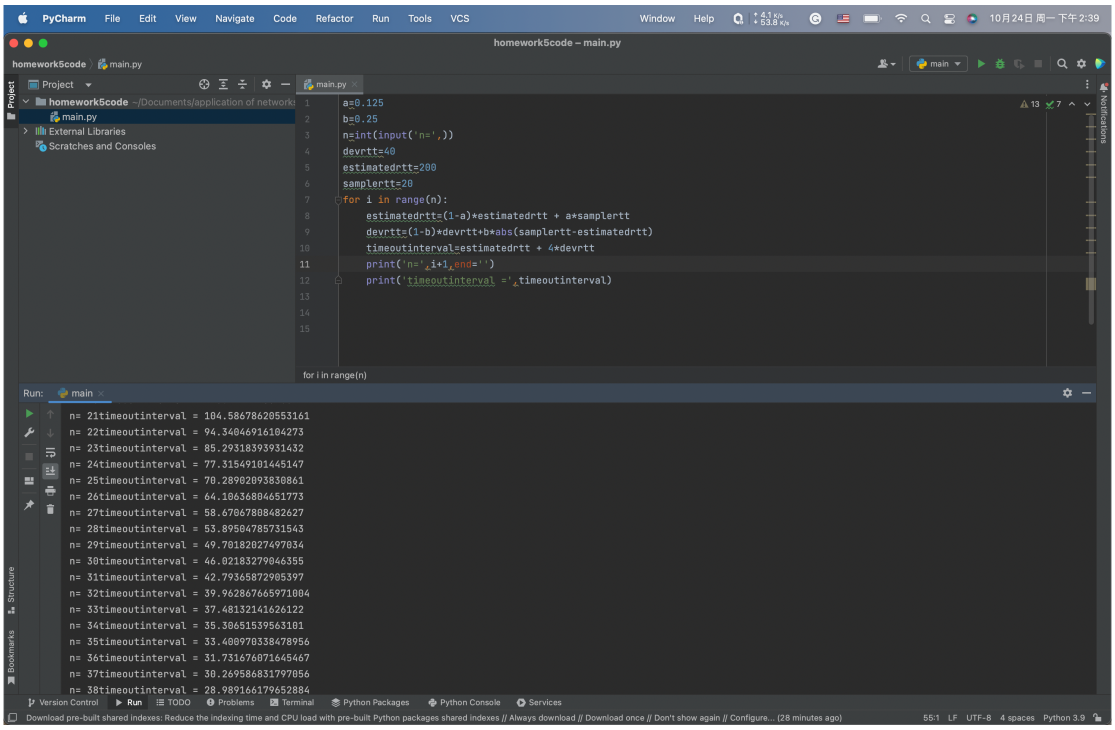
```python
a=0.125
b=0.25
n=int(input('n=',))
devrtt=40
estimatedrtt=200
samplertt=20
for i in range(n):
    estimatedrtt=(1-a)*estimatedrtt + a*samplertt
    devrtt=(1-b)*devrtt+b*abs(samplertt-estimatedrtt)
    timeoutinterval=estimatedrtt + 4*devrtt
    print('n=',i+1,end='')
    print('timeoutinterval =',timeoutinterval)
```

```
/usr/local/bin/python3.9 /Users/xiaxiaoyu/Documents/application of networks/homework5code/main.py
n=10
n= 1timeoutinterval = 455.0
n= 2timeoutinterval = 503.75
n= 3timeoutinterval = 520.625
n= 4timeoutinterval = 516.0546875
n= 5timeoutinterval = 497.5537109375
n= 6timeoutinterval = 470.4888916015625
n= 7timeoutinterval = 438.6498260498047
n= 8timeoutinterval = 404.6726322174072
n= 9timeoutinterval = 370.3540790081024
n= 10timeoutinterval = 336.88396349549294

Process finished with exit code 0
```

(2)

As the picture below shows: When the n is 29, the timeout interval falls below 50ms which is exactly 49.70ms.



homework5code – main.py

```python
a=0.125
b=0.25
n=int(input('n=',))
devrtt=40
estimatedrtt=200
samplertt=20
for i in range(n):
    estimatedrtt=(1-a)*estimatedrtt + a*samplertt
    devrtt=(1-b)*devrtt+b*abs(samplertt-estimatedrtt)
    timeoutinterval=estimatedrtt + 4*devrtt
    print('n=',i+1,end='')
    print('timeoutinterval =',timeoutinterval)
```

```
n= 21timeoutinterval = 104.58678620553161
n= 22timeoutinterval = 94.34046916104273
n= 23timeoutinterval = 85.29318393931432
n= 24timeoutinterval = 77.31549101445147
n= 25timeoutinterval = 70.28902093830861
n= 26timeoutinterval = 64.10636804651773
n= 27timeoutinterval = 58.67067808482627
n= 28timeoutinterval = 53.89504785731543
n= 29timeoutinterval = 49.70182027497034
n= 30timeoutinterval = 46.02183279046355
n= 31timeoutinterval = 42.79365872905397
n= 32timeoutinterval = 39.962867665971004
n= 33timeoutinterval = 37.48132141626122
n= 34timeoutinterval = 35.30651539563101
n= 35timeoutinterval = 33.400970338478956
n= 36timeoutinterval = 31.731676071645467
n= 37timeoutinterval = 30.269586831797056
n= 38timeoutinterval = 28.989166179652884
```

Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Python packages shared indexes // Always download // Download once // Don't show again // Configure... (28 minutes ago)

3.

(a)

If the situation is lossless transmission: the total time for receiving an ack is an RTT, and then we can slide its window forward and transmit new data.

So the total time is $T_1 = RTT = 300ms$.

When we lose a segment, first the sender sends this package and receives ack: same time as $T_1 = RTT = 300ms$.

Secondly, upon receipt of the third duplicate ACK, the fast retransmit algorithm detects the loss and retransmits the lost segment. So after the sender sends segments, it will receive 3 duplicate acks which take 30ms. So $T_2 = T_1 + 3 \times T_{segments} = 300 + 30(ms) = 330\ ms$

When the sender detects the loss, it will retransmit the lost segment and receive the ack from the receiver, which costs one RTT.

So the total time $T_3 = T_2 + RTT = 330 + 300(ms) = 630\ ms$

Finally, the sender loses 330ms (630-300ms=330ms) compared with the lossless transmission.

(b).

If we use the timeout principle, so when timeout the sender retransmits the lost segment and receives the ack from the receiver, which costs one RTT.

So the total time under this situation is: $T_4 = Timeout + RTT = 400 + 300(ms) = 700\ ms$

We can compare both situations：$T_4 - T_3 = 700 - 630(ms) = 70\ ms$

We can save 70ms if we use fast retransmit.

4.

(a)

If our congestion window at some point in time is W bytes, then we are allowed to send W bytes per RTT.

So max throughput at that time is $\frac{W_{max}}{RTT}$.

$$throughtput = 100Mbps = \frac{max\ window\ size \times TCP\ segment\ size\ \times 8}{RTT}$$

So, $$= \frac{max\ window\ size\ \times 1460 \times 8}{0.05}$$

$$max\ window\ size = \frac{0.05 \times 100 \times 10^6}{1460 \times 8} = 428.08 \approx 428$$

(b)

Ignore slow start, assume we are always in congestion avoidance:

Window increases linearly from $\frac{W_{max}}{2}$ to W_{max} until loss occurs and process repeats.

So the average window size is ($\frac{W_{max}}{2}+W_{max}$)/2=0.75$W_{max}$

Average throughput is $\frac{0.75W_{max}}{RTT} = \frac{0.75 \times 428 \times 1460 \times 8}{0.05} = 74.9856Mbps$

5.

(a)

Parallel connection is separating TCP connection per object but runs them in parallel.

And the TCP fairness goal is: if K TCP sessions share the same bottleneck link of bandwidth R, each should have an average rate of R/K.

When we have a parallel connection, we can get a larger bandwidth compared to one TCP connection.

(b)

1. Allows the client to open multiple connections and execute multiple HTTP tasks in parallel.
2. Ability to increase page loading speed, overcoming dead time and bandwidth limitations of single-hop connections.