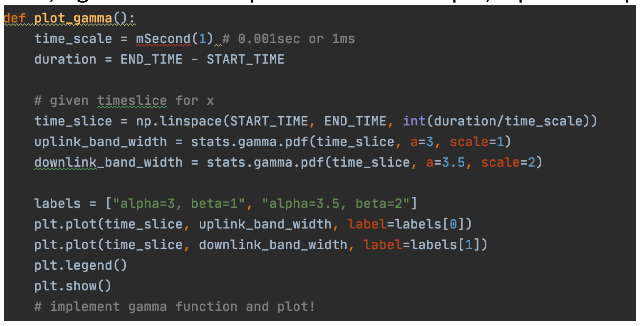
Project 2 보고서

2017-11621 전기정보공학부 배병욱

1. Reproduce

1-a. Gamma distribution

Function to plot gamma distribution is shown as below, it has been achieved by using scipy's library. Also, figure below is reproduced same as pdf, alpha is shape parameter and beta is scale parameter.



```python
def plot_gamma():
    time_scale = mSecond(1) # 0.001sec or 1ms
    duration = END_TIME - START_TIME

    # given timeslice for x
    time_slice = np.linspace(START_TIME, END_TIME, int(duration/time_scale))
    uplink_band_width = stats.gamma.pdf(time_slice, a=3, scale=1)
    downlink_band_width = stats.gamma.pdf(time_slice, a=3.5, scale=2)

    labels = ["alpha=3, beta=1", "alpha=3.5, beta=2"]
    plt.plot(time_slice, uplink_band_width, label=labels[0])
    plt.plot(time_slice, downlink_band_width, label=labels[1])
    plt.legend()
    plt.show()
    # implement gamma function and plot!
```

Fig 1. plot_gamma() function's code

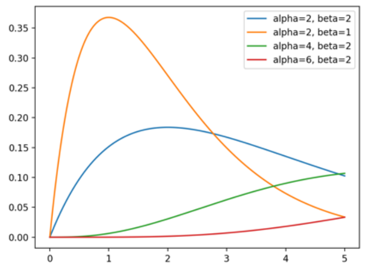
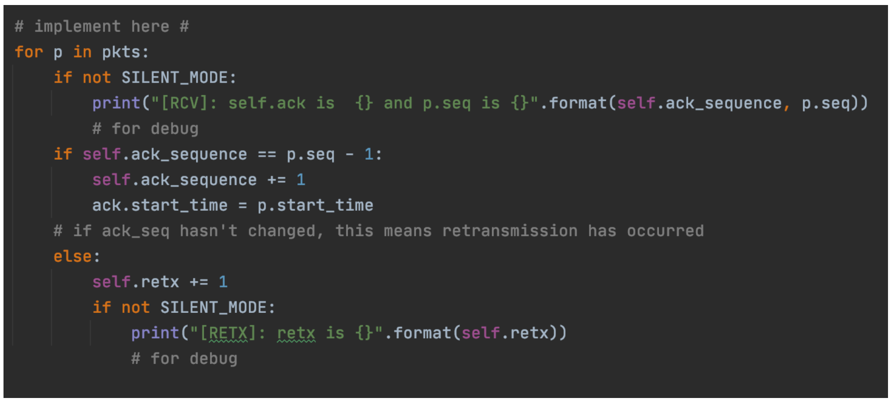


Fig 2. Reproduction of gamma function of pdf

1-b. case1

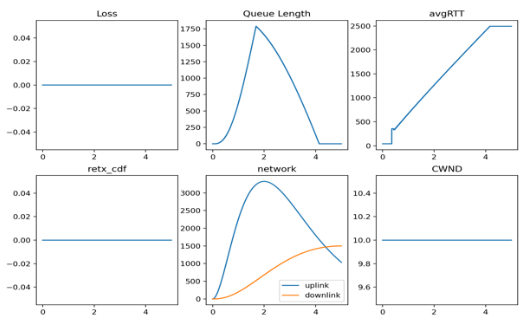
For case 1 all I had to do is implementation of receive function. And this receive function is same for all case

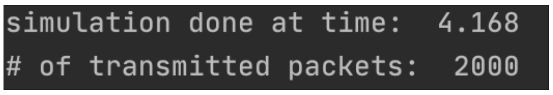


```python
# implement here #
for p in pkts:
    if not SILENT_MODE:
        print("[RCV]: self.ack is  {} and p.seq is {}".format(self.ack_sequence, p.seq))
        # for debug
    if self.ack_sequence == p.seq - 1:
        self.ack_sequence += 1
        ack.start_time = p.start_time
    # if ack_seq hasn't changed, this means retransmission has occurred
    else:
        self.retx += 1
        if not SILENT_MODE:
            print("[RETX]: retx is {}".format(self.retx))
            # for debug
```

Fig 3. Implementation of receive function

By checking sequence number of packet, receive function can prevent making duplicated Acks for packets that received. And by checking ack's sequence number, receiver can check retx number too.
Result of reproduction result of case 1 is as belows.



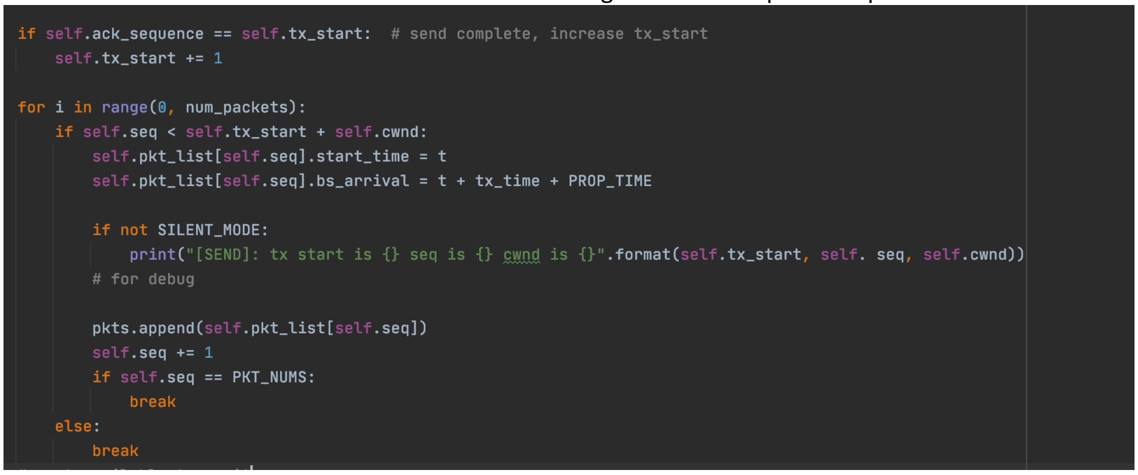


```
simulation done at time:   4.168
# of transmitted packets:  2000
```

Fig 4. Reproduction of case 1

1-c. case2 & case3

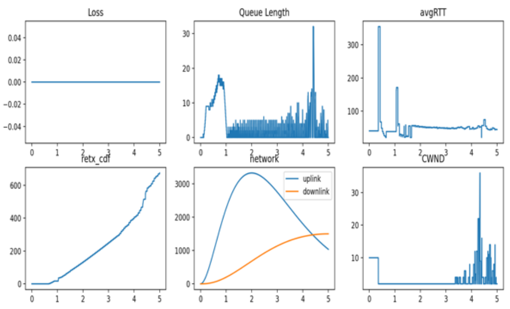
For case 2 and 3 I had to check CWND to restrict sliding window. This part is implemented at send function.

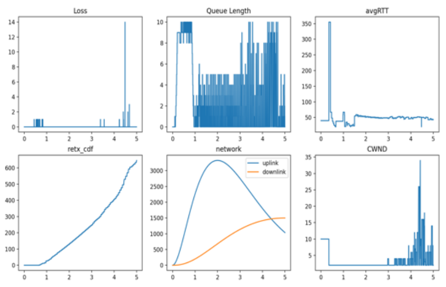


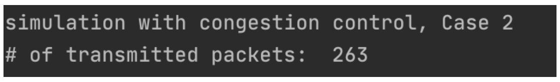
```
if self.ack_sequence == self.tx_start:  # send complete, increase tx_start
    self.tx_start += 1

for i in range(0, num_packets):
    if self.seq < self.tx_start + self.cwnd:
        self.pkt_list[self.seq].start_time = t
        self.pkt_list[self.seq].bs_arrival = t + tx_time + PROP_TIME

        if not SILENT_MODE:
            print("[SEND]: tx start is {} seq is {} cwnd is {}".format(self.tx_start, self. seq, self.cwnd))
        # for debug

        pkts.append(self.pkt_list[self.seq])
        self.seq += 1
        if self.seq == PKT_NUMS:
            break
    else:
        break
```

Fig 5. Implementation of send function

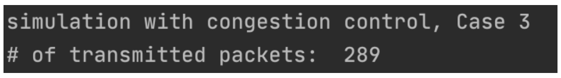
Almost everything is same with case1's send function except there's if statement to check whether sliding window is restricted by CWND size. Result of reproduction of case 2 is as belows.







```
simulation with congestion control, Case 2
# of transmitted packets:  263
```

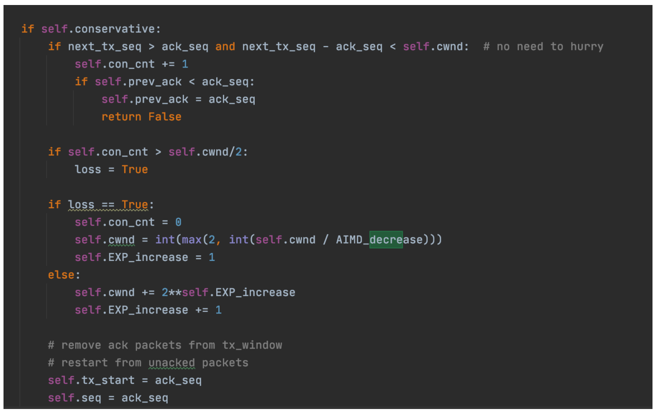


```
simulation with congestion control, Case 3
# of transmitted packets:  289
```

Fig 6& 7. Reproduction of case 2(left) case 3(right)

2. Version 2

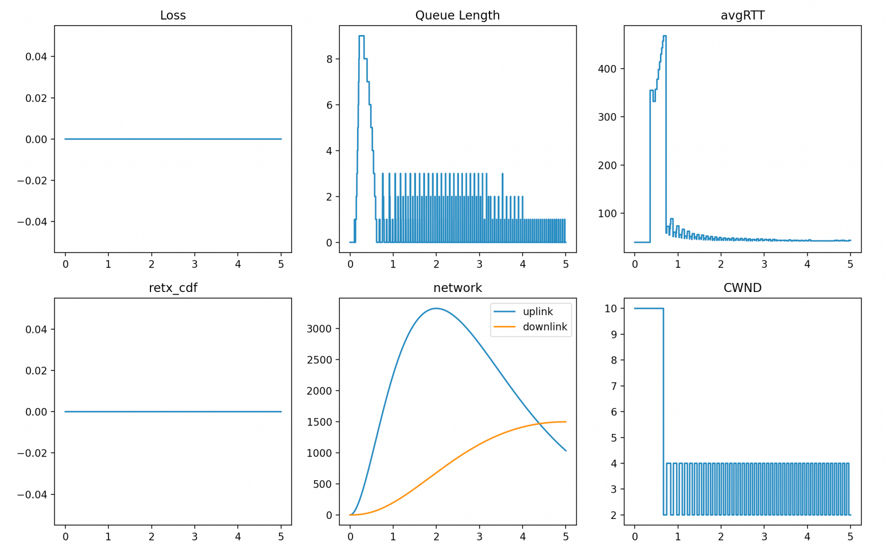
Retransmission of version 1 is due to naïve congestion control. For example, during naïve congestion control, sender sends 0 & 1 pkt(for cwnd 2), but due to bandwidth of downlink receiver receive pkt 0 and send ack1(as receiver sends ack sequence of next requested packet) and pkt1 is queued at base station. As sender received ack1, it will send pkt 1 & 2, and receiver will receive pkt 1 which was queued at base station. This will make vicious circle of every pkt is sent twice until down link BW is increased. Code that prevents retx is as below.

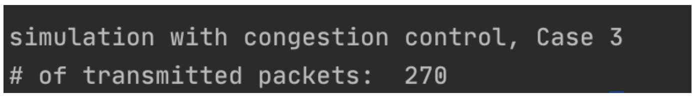


```
if self.conservative:
    if next_tx_seq > ack_seq and next_tx_seq - ack_seq < self.cwnd:  # no need to hurry
        self.con_cnt += 1
        if self.prev_ack < ack_seq:
            self.prev_ack = ack_seq
            return False

    if self.con_cnt > self.cwnd/2:
        loss = True

    if loss == True:
        self.con_cnt = 0
        self.cwnd = int(max(2, int(self.cwnd / AIMD_decrease)))
        self.EXP_increase = 1
    else:
        self.cwnd += 2**self.EXP_increase
        self.EXP_increase += 1

    # remove ack packets from tx_window
    # restart from unacked packets
    self.tx_start = ack_seq
    self.seq = ack_seq
```

Fig 7. Implementation of conservative congestion control

As retx is happened due to slow downlink rate, it can be prevented by waiting for ack sequence number until sequence number reaches cwnd window size. But this can be troublesome. Checking previous ack num is to check duplicated acks, which means loss. But as simulation situation does not consider packet loss, it is useless for this situation. To prevent CWND size does not change or changed too much, code check self.con_cnt, if it doesn't wait long enough, we will think there's need for increasement of cwnd size.





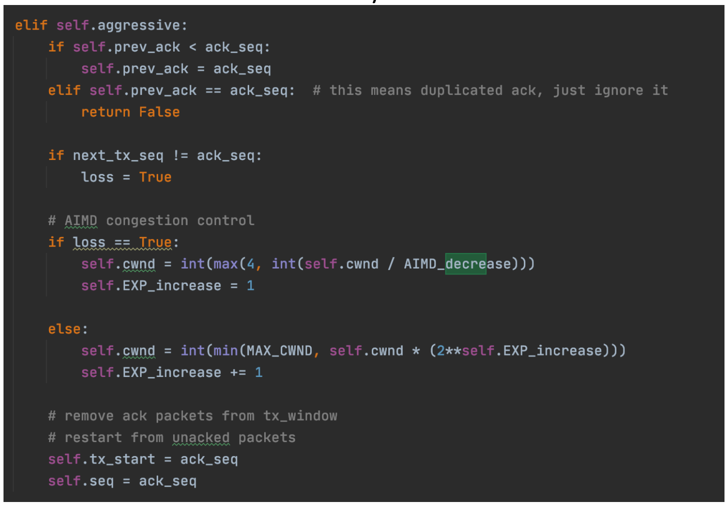
```
simulation with congestion control, Case 3
# of transmitted packets:  270
```

Fig 8. Result of case 3 of version 2

Result is completed as implemented, but by checking CWND it shows that size of CWND size is fluctuated, and transmitted packets have been decreased due to super conservative method. This can be shown by queue length to as it doesn't make queue length longer than 10. So This means it needs to be much greedy to use in reality. Result can be checked by setting self.conservative of Clinet() true.

3. Version 3

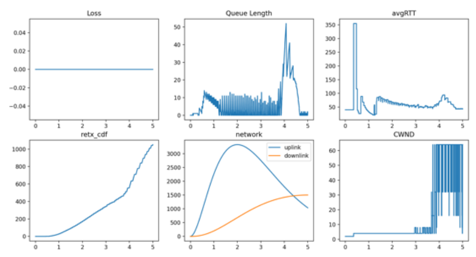
To make transmitted packets over 600. I implement exponential increase of CWND window size. This will make sure queue is always full and use entire bandwidth every times. Code is as belows.

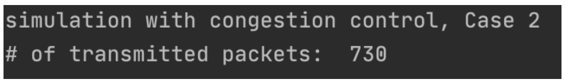


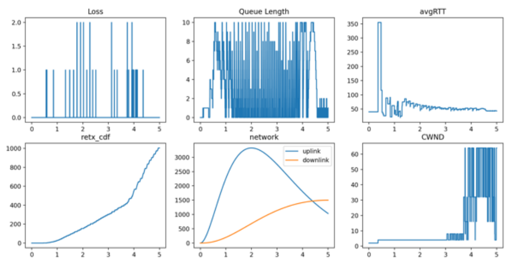
```
elif self.aggressive:
    if self.prev_ack < ack_seq:
        self.prev_ack = ack_seq
    elif self.prev_ack == ack_seq:  # this means duplicated ack, just ignore it
        return False

    if next_tx_seq != ack_seq:
        loss = True

    # AIMD congestion control
    if loss == True:
        self.cwnd = int(max(4, int(self.cwnd / AIMD_decrease)))
        self.EXP_increase = 1

    else:
        self.cwnd = int(min(MAX_CWND, self.cwnd * (2**self.EXP_increase)))
        self.EXP_increase += 1

    # remove ack packets from tx_window
    # restart from unacked packets
    self.tx_start = ack_seq
    self.seq = ack_seq
```

Fig 9. Result of case 3 of version 3

And to prevent duplicated resend of same packets, it checks previous ack sequence number and current ack sequence number. And if it doesn't changed neglect it. This can be done by condition of simulation as it doesn't have packet loss.




```
simulation with congestion control, Case 2
# of transmitted packets:  730
```



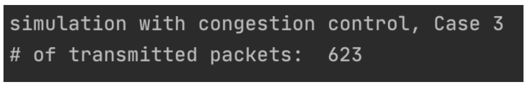

```
simulation with congestion control, Case 3
# of transmitted packets:  623
```

Fig 10 & 11. Result of case 2 of version 3(left) and result of case 3 of version 3(right)

As it can be checked on figure this method will overwhelm base station. For case 2, as it doesn't have limitation of base station's queue, loss had not occurred. But for case 3 loss occurred occasionally due to greedy choice of CWND window size and exceed base station's buffer size. In contrast to version 2, version 3 will resends packets again and again due to aggressive CWND size. In reality this will make unfair and inefficient use of bandwidth, which is not desirable. Result can be checked by settting Client()'s self.conservative false and self.aggressive true.

4. Conclusion

Through the comparison of naïve congestion control, and my implemented conservative/aggressive congestion control, pros and cons of each method could be seen. For naïve congestion control, this model does not have neither decent number of transmitted packets nor advantages of retransmission. For conservative model, it has advantage of no retransmission by super conservative attitude. But has disadvantage of transmitted packets. For aggressive model, it has advantage of transmitted packet but disadvantage of fairness of bandwidth. It will have troubles when it comes to reality, as it will degrade other client's bandwidth, and also hadn't consider loss of packets.