Xi'an Jiaotong-Liverpool University
西交利物浦大学

EEE413 DATA COMMUNICATION AND COMMUNICATIONS NETWORKS

Coursework 2: Traffic Shaping with Token Bucket Filter

Student Name	Xun Xu
Student ID	1926930

2019/12/20

1.Introduction

There are two commonly used current limiting algorithms: leaky bucket algorithm and token bucket algorithm.

The idea of the leaky bucket algorithm is very simple. The water (request) first enters the leaky bucket, and the leaky bucket outputs water at a certain speed. When the water inflow speed is too large, the leaky bucket algorithm can forcibly limit the data transmission rate. As shown in the figure 1, a is a fluid model, and b is a model of a computer processing package.

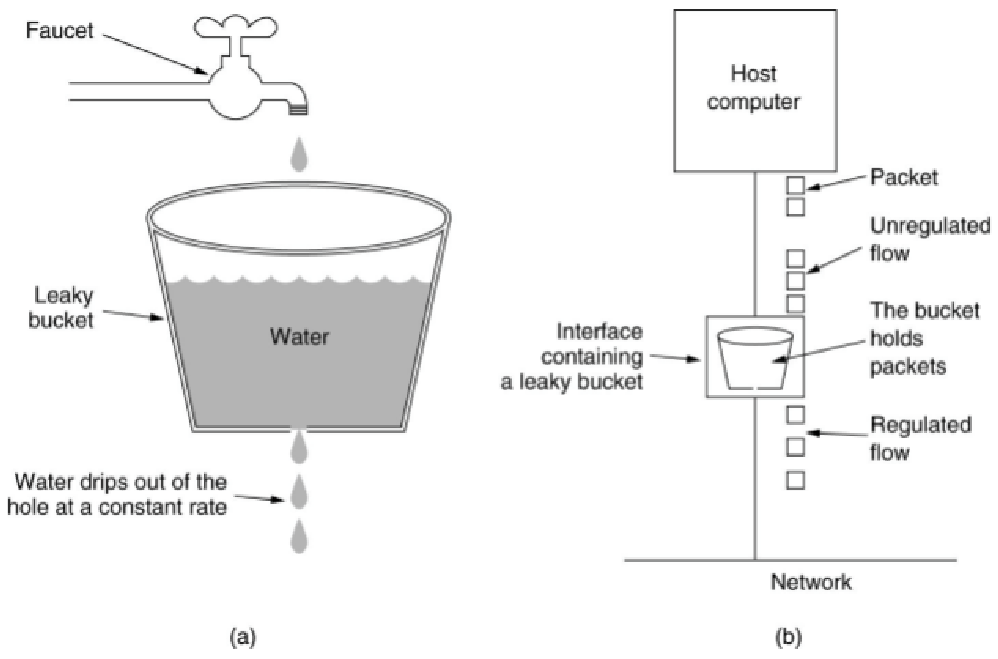


figure 1： Leaky bucket algorithm model

For many application scenarios, in addition to being able to limit the average data transmission rate, it is also required to allow some degree of burst transmission. At this time, the leaky bucket algorithm may not be suitable, and the token bucket algorithm is more suitable. As shown in Figure 2, the principle of the token bucket algorithm is that the system will put tokens into the bucket at a constant rate, and if the request needs to be processed, you need to obtain a token from the bucket first. When the token is desirable, the service is denied, as shown in figure 2.

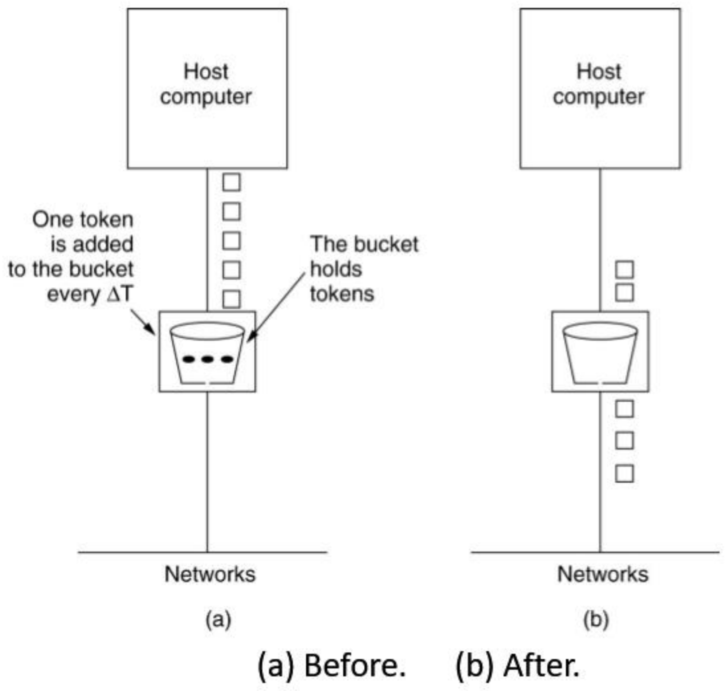


figure 2：Token Bucket Algorithm Model

In this coursework, what we have to do is to simulate the control process of the token bucket on the data packets in the network to achieve the purpose of controlling bursts. Before the simulation, we need to calculate two important parameters that need to be used: token bucket generation rate and token bucket size. The calculation method is as follows:

Here we use a TBF traffic shaper（the combined leaky bucket and token bucket）and it is designed for allowing the periodic on-off traffic shown in to pass through it without shaping. We assume that the on and off periods are both 0.1ms. The On-off Packet generation pattern is as follows:

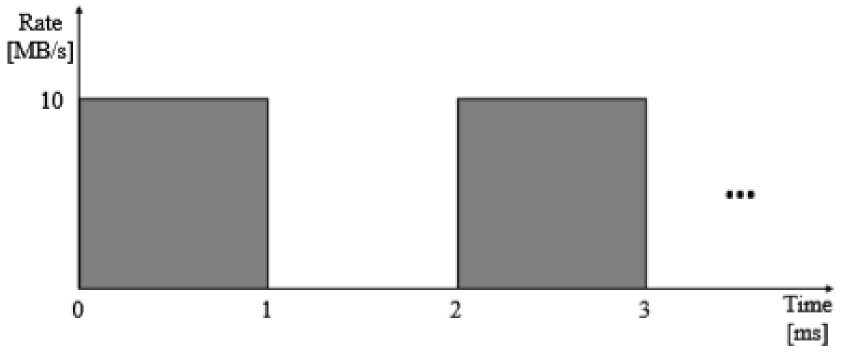


figure 3: A periodic on-off traffic pattern

Below are the TBF parameters (as shown in figure 4):

• Token generation rate: ρ MB/s.

• Token bucket capacity: C MB (full when the first burst arrives).

• Peak output rate: 10 MB/s (same as the input rate).

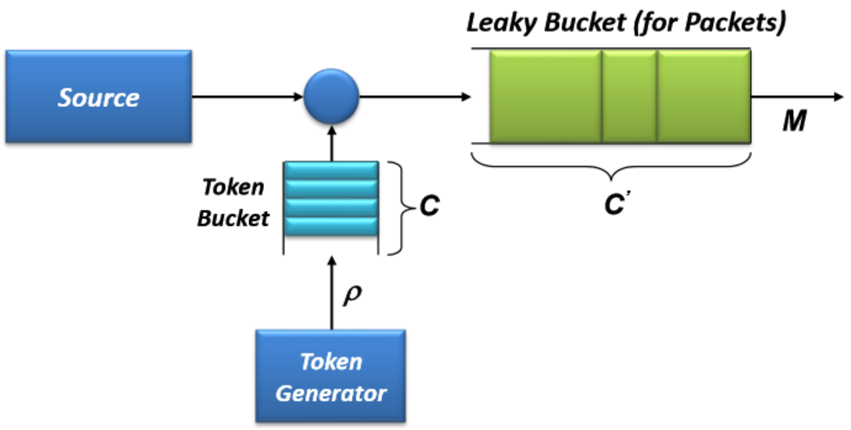


figure 4: Combined Use of Leaky Bucket and Token Bucket

The TBF design is done in the following three steps using a fluid model1: 1) First, we find the minimum value of the token bucket capacity C — in terms of ρ — for the first burst to pass through the TBF without shaping. Because the maximum burst length S in second is given by:

$$S = \frac{C}{10 - \rho}$$

The condition for C that allows the first burst with the duration of 0.001s (i.e., 1ms) to pass through is:

$$0.001 \leq S = \frac{C}{10 - \rho}$$

$$0.01 - 0.001\rho \leq C$$

Therefore the minimum token bucket capacity is (0.01−0.001ρ) MB.

2) Second, given the minimum value of C from 1), we find the minimum (numerical) token generation rate ρ for the second burst (and therefore all bursts) to pass through the TBF without shaping. With the minimum value of C from 1), the token bucket becomes empty at the end of the first burst. For the second burst to pass through it, therefore, the token bucket should be full again by the beginning of the second burst.

This means that the amount tokens generated during the off period (i.e., 1ms=0.001s) should be equal to or greater than the token bucket capacity, i.e.

$$0.001\rho \geq 0.01 - 0.001\rho$$

$$\rho \geq \frac{0.01}{0.002} = 5$$

Therefore the minimum token generation rate is 5MB/s.

3) Finally, from the results of 1) and 2), we obtain the minimum token bucket capacity of 5kB (= (0.01−0.001×5) MB).

2. Code writing and modification

2.1 Code execution process

After setting the initialization parameters related to the environment variable env of Simpy, the data packets generated from the on period of OnoffPacketGenerator() are put into the store's queue through store.put(), After passing into token bucket filter of FifoQueue(), the data packet is obtained through store.get(), and then passes out at token bucket algorithm, it enters PacketSink().

The main classes and variables used are as follows:

```
OnoffPacketGenerator() # Generate packets at on period
FifoQueue()                    # pass the token bucket filter
PacketSink()                  # pass the leaky bucket
store.get()                  # store the packet in a queue
store.put()                  #get packet from the queue once
Packet()                       #    Generate fixed-size packet
```

2.2 Parameters and variables

The main variables(global variables) used are as follows:

pkt_size: packet size, default is 1000B

token_grate: token_generator_rate, default is 5MB=5000000B bucket_size: token_bucket_size, default is 5KB=5000B

pkt_ia_time: packet interarrival time, default is 0.01ms= 0.00001s

on_period：on period, default is 0.1ms=0.0001s

off_period：off period, default is 0.1ms=0.0001s

sim_time: time to end the simulation, default is 1ms=0.01s

Set as a global variable so that we can use batch instructions to modify in batches later. Among them, pkt_ia_time is on_period/10 and the initialization codes of token_grate and token_bucket_size are as follows:

token_grate:

```
parser.add_argument(
    "-V",
    "--token_grate",
    help="token_generator_rate [byte/second]; default is 5000000",
    default=5000000,
    type=int)
```

token_bucket_size:

```
parser.add_argument(
    "-X",
    "--bucket_size",
    help="token_bucket_size size [byte]; default is 5000",
    default=5000,
    type=int)
```

Batch instructions to run code:

```
@echo off
for /L %S in (1000, 500, 10000) do python queue_onoff_traffic.py  -S %S --no-trace>>averagewaitingtime1(packet_size_change).out
for /L %T in (0.002, 0.002, 0.01) do python queue_onoff_traffic.py  -T %T --no-trace>>averagewaitingtime2(simulation_time_change).out
for /L %X in (1000, 500, 10000) do python queue_onoff_traffic.py  -X %X --no-trace>>averagewaitingtime3(token_bucket_size_change).out
```

2.3 Token bucket filter code

The main logical process of the token bucket algorithm is as follows:

In flow control, the size of the token bucket is the maximum value of one packet flow. The basic working process of the token bucket is as follows: Sets the maximum token value stored in the store. Then the token bucket size starts to increase at its rate, always keeping the size of the packets passing through the token bucket smaller than the token bucket. When the data packet arrives, the bucket will make a judgment. If the number of bucket tokens is greater than the packet, the packet-sized token is removed from the bucket and this message is sent. When the number of bucket tokens is less than the data packet, this data packet will be sent to the buffer and wait for enough tokens until the number of tokens grows to be consistent with the size of the data packet before transmitting, and then the token bucket value Set to 0.

The requirements for the coursework are as follows:

1. Update the amount of tokens based on – Amount of time passed since the last update– Token generation rate and bucket size

2. Compare the message size and the token amount.

– If the message size is larger than the token amount: • Wait until enough tokens are generated for the message. – Using yield self.env.timeout(…) • Then, set the token amount to zero.

– Otherwise, reduce the token amount by the message size.

3. Store the current time in a member variable.

4. Delay packet transmission time.

5. Send the message through the output port.

The token bucket algorithm code is as follows. Read the comments for specific explanations.

```
    token_grate=token_grate
    bucket_size=bucket_size
    bucket_fixed=bucket_size
    lasttime = 0

    while True:
        msg = (yield self.store.get())

        # TODO: Implement packet processing here.
        # Calculate the generated token size based on the time difference between the two packets
        delay_global=(self.env.now-lasttime)
        bucket_size=bucket_size+delay_global*token_grate
        # If the token size exceeds the set maximum, modify it to a fixed size
        if(bucket_size>bucket_fixed):
            bucket_size=bucket_fixed
        # When the packet is larger than the token size, wait for the token to grow to
match the packet size
        if(msg.size>bucket_size):
            delay_add=(msg.size-bucket_size)/token_grate
            yield self.env.timeout(delay_add)
            self.out.put(msg)
            bucket_size=0
        # If the packet is smaller than the token size, the token size is reduced by one p
acket size
        if(msg.size<= bucket_size):
            bucket_size=bucket_size-msg.size
            self.out.put(msg)
        # Update the time when the last packet was sent
        lasttime=self.env.now
```

2.4 Code for plot the comparison

I wrote three plot codes, taking the example of changing the size of the token bucket as an example. The main method is to read out the previously generated out file, draw the result as the horizontal and vertical coordinates, and recalculate the coordinates and axis units for easy observation.

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

x1, y1 = np.loadtxt('averagewaitingtime3(token_bucket_size_change).out', delimiter='
\t', unpack=True)
x1=x1/1000
y1=10000*y1
#plt.plot(x2, y2, 'rx', label="Simulation")
plt.plot(x1, y1, 'rx')

# add labels, legend, and title

plt.xlabel(r'bucket size [E+3Byte]')
plt.ylabel(r'average delay [E-04s]')
plt.legend()
plt.title(r'Average delay for different bucket size')
plt.savefig('average_delay(token_bucket_size_change).pdf')
plt.axis([0,10,0,5])
plt.show()
```

2.5 Code running method

Run case.bat to produce the .out for plot.

Run plot_average_delay_for_bucket_change to plot the figure for bucket changing.

Run plot_average_delay_for_packet_change to plot the figure for packet changing.

Run plot_simulation_time_change to plot the figure for simulation time changing.

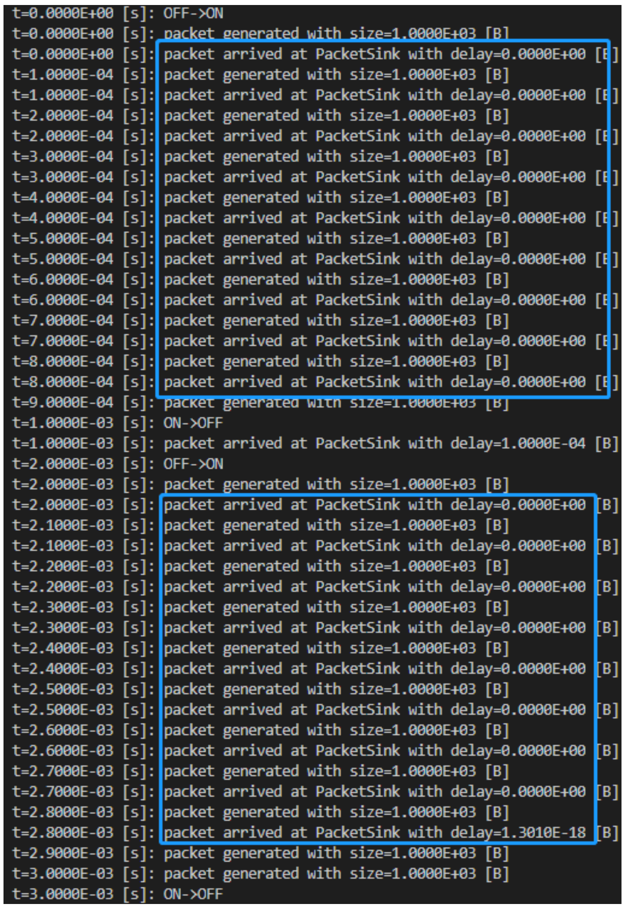
Because the batch instructions for small numbers (simulation time) will be executed as 0 and an error is reported, I put the results of different simulation times directly into the file change_simulation_time.

3. Task solving & Results analysis

3.1 Task solving

Task1

You can see 2.3 for the code of the token bucket algorithm. During the simulation, the delay of each packet can be displayed on the screen as follows:



```
t=0.0000E+00 [s]: OFF->ON
t=0.0000E+00 [s]: packet generated with size=1.0000E+03 [B]
t=0.0000E+00 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=1.0000E-04 [s]: packet generated with size=1.0000E+03 [B]
t=1.0000E-04 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=2.0000E-04 [s]: packet generated with size=1.0000E+03 [B]
t=2.0000E-04 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=3.0000E-04 [s]: packet generated with size=1.0000E+03 [B]
t=3.0000E-04 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=4.0000E-04 [s]: packet generated with size=1.0000E+03 [B]
t=4.0000E-04 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=5.0000E-04 [s]: packet generated with size=1.0000E+03 [B]
t=5.0000E-04 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=6.0000E-04 [s]: packet generated with size=1.0000E+03 [B]
t=6.0000E-04 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=7.0000E-04 [s]: packet generated with size=1.0000E+03 [B]
t=7.0000E-04 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=8.0000E-04 [s]: packet generated with size=1.0000E+03 [B]
t=8.0000E-04 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=9.0000E-04 [s]: packet generated with size=1.0000E+03 [B]
t=1.0000E-03 [s]: ON->OFF
t=1.0000E-03 [s]: packet arrived at PacketSink with delay=1.0000E-04 [B]
t=2.0000E-03 [s]: OFF->ON
t=2.0000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=2.0000E-03 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=2.1000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=2.1000E-03 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=2.2000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=2.2000E-03 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=2.3000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=2.3000E-03 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=2.4000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=2.4000E-03 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=2.5000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=2.5000E-03 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=2.6000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=2.6000E-03 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=2.7000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=2.7000E-03 [s]: packet arrived at PacketSink with delay=0.0000E+00 [B]
t=2.8000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=2.8000E-03 [s]: packet arrived at PacketSink with delay=1.3010E-18 [B]
t=2.9000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=3.0000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=3.0000E-03 [s]: ON->OFF
```

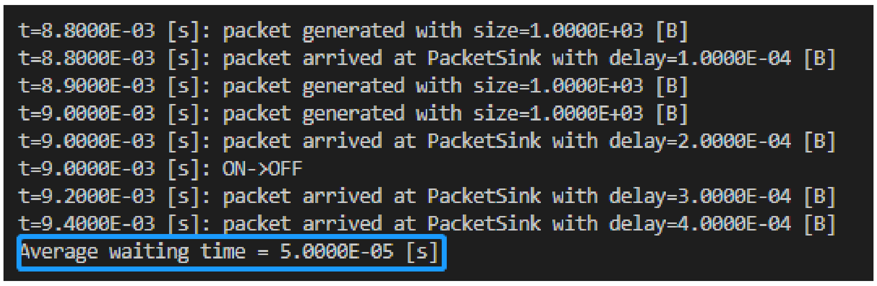
figure 5: delay of each packet

Here is its code:

```
    def run(self):
        # i=0
        # msg_lasttime=0
        # delay_msg_all=0
        while True:
            msg = (yield self.store.get())
            now = self.env.now
            delay_sink=now - msg.ctime
            self.wait_times.append(now - msg.ctime)
            if self.trace:
                print("t={0:.4E} [s]: packet arrived at PacketSink with 
delay={1:.4E} [B]".format(now, delay_sink))
```

We use the time of each packet creation and outgoing sink as the delay and print it to the screen.

The average delay is calculated at the end of the simulation:



```
t=8.8000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=8.8000E-03 [s]: packet arrived at PacketSink with delay=1.0000E-04 [B]
t=8.9000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=9.0000E-03 [s]: packet generated with size=1.0000E+03 [B]
t=9.0000E-03 [s]: packet arrived at PacketSink with delay=2.0000E-04 [B]
t=9.0000E-03 [s]: ON->OFF
t=9.2000E-03 [s]: packet arrived at PacketSink with delay=3.0000E-04 [B]
t=9.4000E-03 [s]: packet arrived at PacketSink with delay=4.0000E-04 [B]
Average waiting time = 5.0000E-05 [s]
```

figure 6: average delay

Here is its code:

```
    print("Average waiting time = {:.4E} [s]\n".format(np.mean(ps.wait_
times)))
```

Task2

As the average packet delay from the simulation is ignored, we can conclude that there has been no delay (equal to 0) at the TBF theoretically. However we can see through the previous task that there is a certain delay, which means that there must be some packets have been delayed at the TBF due to traffic shaping.

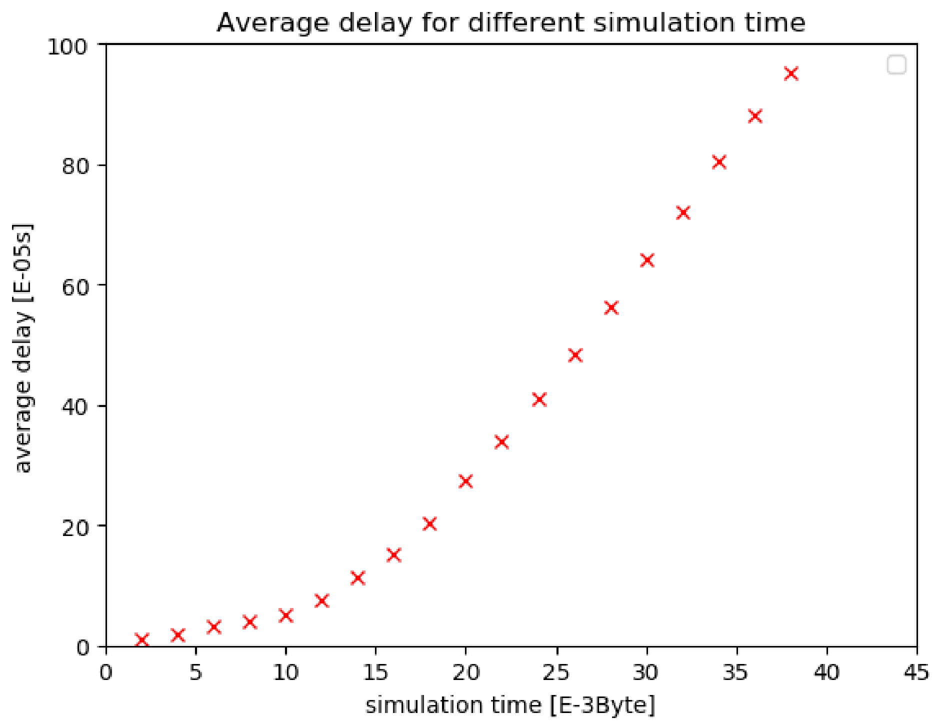


figure 7: Average delay for different simulation time

I draw the delays of different simulation times in the same picture, as shown in the figure 7. It can be seen that as time increases, the delay tends to increase linearly, that is, as the system enters dynamic equilibrium, the size of each packet that needs to be shaved is also stable, but a delay occurs. It is inconsistent with 0 in our theoretical results. I think this is related to our model in python. Our initial theoretical calculations were based on the fluid model, but in fact, the data packets we send cannot be completely regarded as fluid. It is generated and sent discretely in time. May be the cause of the delay, so we need shaping.

I think we have many ways to reduce this delay to 0, mainly as follows:

1.increase the size of the token bucket

2. reduce burst length

3.increase the period of off time

Among them, increasing the size of the token bucket is the most direct method. So I plotted the delays for different token bucket sizes, and the results are as follows:

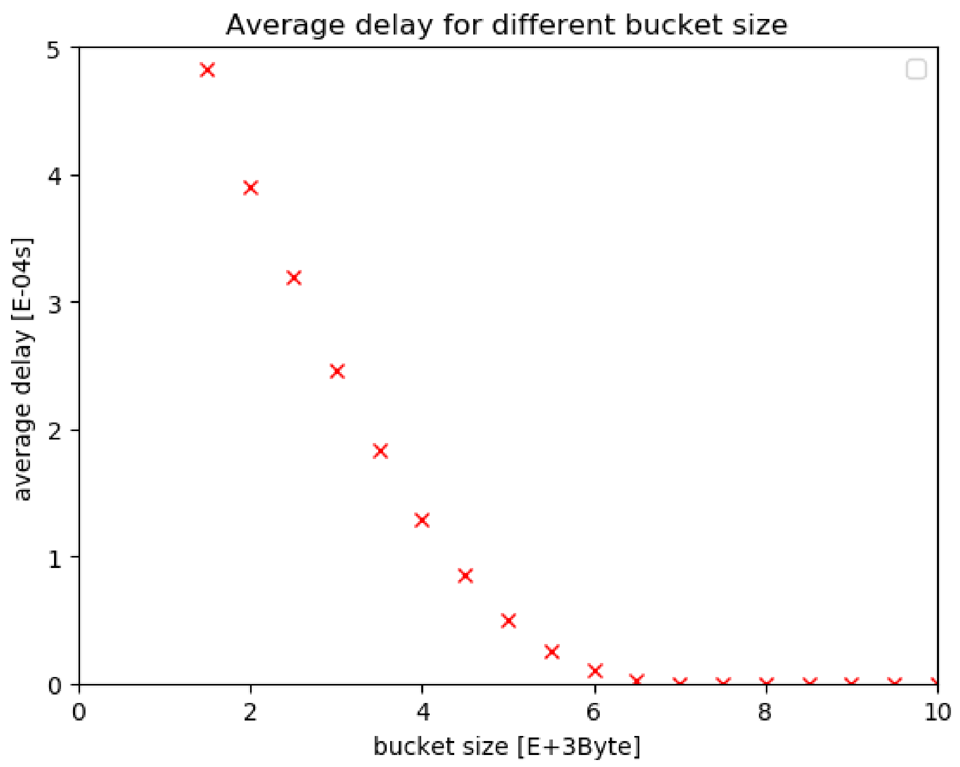
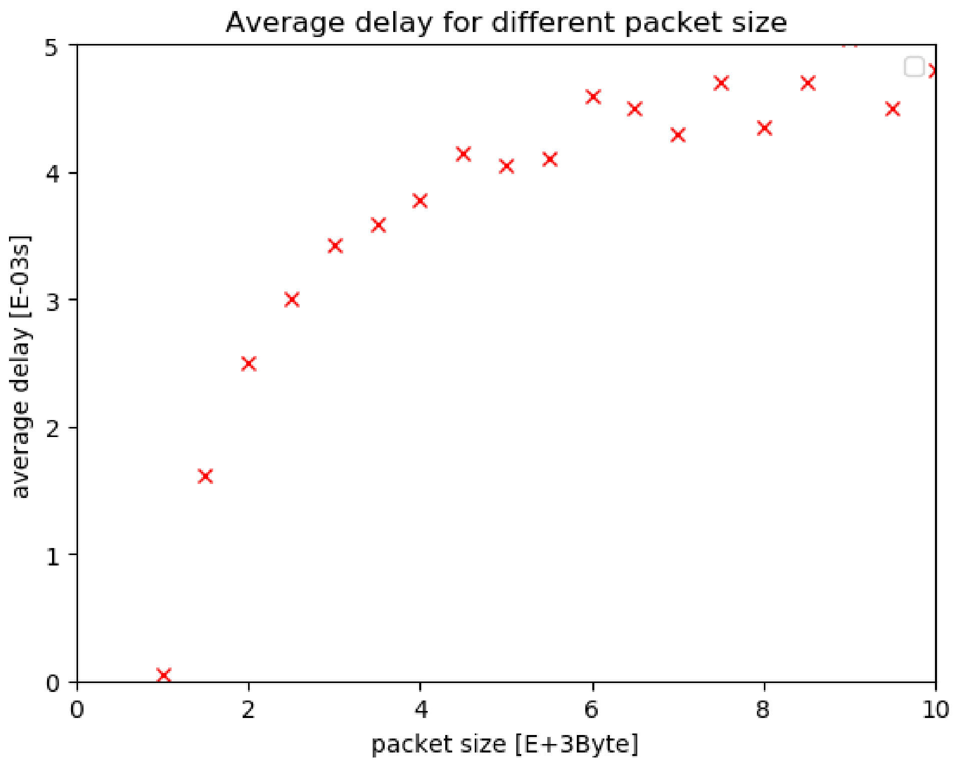


figure 8: Average delay for different bucket

According to the figure 8, when the token bucket size is greater than 6000B, the delay is close to 0, and when the token bucket size reaches 8000B, the delay is already 0, so we can increase the token bucket size to solve the shaping problem. problem. The process of other methods is similar and will not be repeated here.

In addition, I also found an interesting thing. As the number of packets increases, the delay actually approaches a stable value instead of increasing, as in figure 9.



4.Appendix: code

Two folders in compressed package