Lab 2: Simulation of Fair Queueing and Weighted Fair Queueing

SCHOOL OF ADVANCED TECHNOLOGY

INT405: DATA COMMUNICA TION AND COMMUNICA TIONS NETWORKS

Mengfan Li

ID number：2032048

2020 Fall

Abstract

In this Lab, you are to implement simulation models of a router running fair queueing (FQ) and weighted fair queueing (WFQ) algorithms based on SimPy and investigate the flow throughputs of the implemented models.

Declaration

Content

1 Objectives

1.1 Model description

Fig. 1 shows a router with four packets queued for a common output line **O**, where a box of each packet indicates a data unit and the service of packets starts from the input line **A** and down.

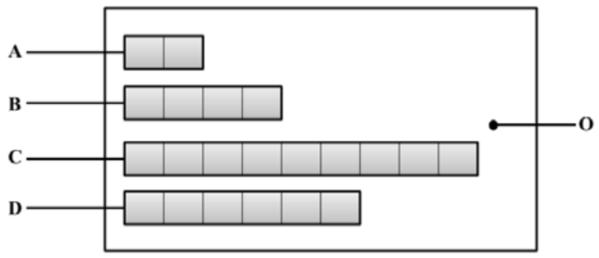
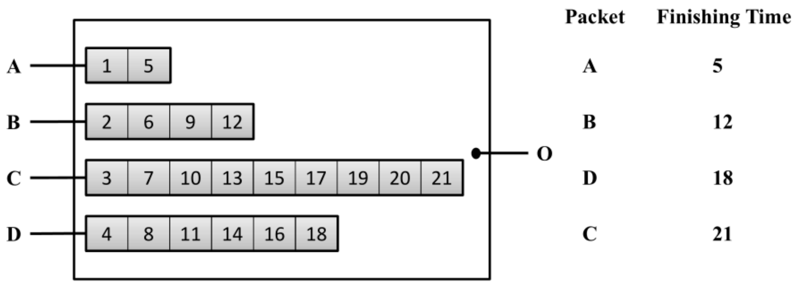


Fig. 1. A router with packets queued for a common output line.

Fig. 2 illustrates how to determine the order of packet transmissions under **FQ** algorithm, where the finishing times of packets are calculated based on unit-wise round robin; the order of packet transmissions is given by **A→B→D→C**.



Packet	Finishing Time
A	5
B	12
D	18
C	21

Fig. 2. Finishing times of packets under FQ.

If FQ is replaced by **WFQ** with the weights given in Table I, the calculation of the finishing times of packets are changed as shown in Fig. 3; the order of packet transmissions in this case is given by **A→C→D→B**.

TABLE I
WEIGHTS OF INPUT LINES.

Input Line	A	B	C	D
Weight	1	1	3	2

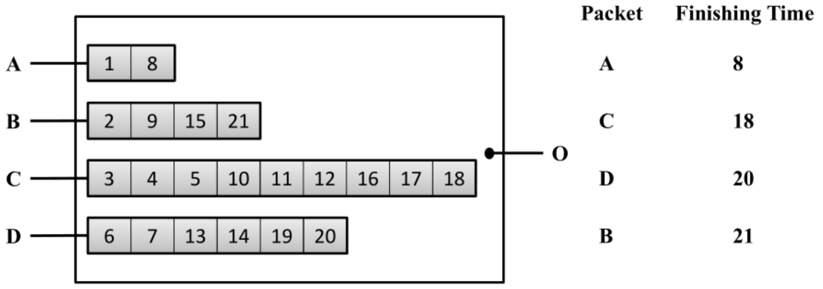


Fig. 3. Finishing times of packets under WFQ.

1.2 Mathematical description(WFQ)

The biggest disadvantage of the GSP (General Purpose Processor Sharing) algorithm is that it cannot handle variable-length packets. Assuming that Fp is the departure time of the data packet P in the GPS algorithm, then the WFQ algorithm is a job reservation algorithm that simulates the GPS algorithm and schedules data packets in ascending order. That is to say, the GSP algorithm always selects the data packet with the smallest Fp value for scheduling . His characteristics are as follows:

GPS is an ideal fluid service model; in GPS, an independent FIFO queue is established for each session sharing the same link; when there are N non-empty queues in any period of time, the server simultaneously responds to N queues Header package service; each session has a different service share, and each session provides services in proportion to their service share; the service shares of N sessions are represented by φ1, φ2, ... φN, we also call φi as the queue weight of i.

The server runs at a constant rate r and is continuous (Work-conserving); Wi(t1,t2) is used to represent the number of services for session i in time [t1,t2], then a GPS server satisfies the service of active session i, j within time [t1, t2]:

$$\frac{W_i(t_1,t_2)}{W_j(t_1,t_2)} = \frac{\emptyset_i}{\emptyset_j} = \frac{r_i t}{r_j t} = \frac{r_i}{r_j}$$

If the set B(t1) of backlog sessions remains unchanged at time [t1, t2], the rate at which session i is served during this time period is:

$$g_i(t_1, t_2) = \frac{\emptyset_i}{\sum_{j \in B(t_1)} \emptyset_j} r$$

Because B(t1) is a subset of the set B of all sessions in the server, it is easy to see (ri is the reserved rate, the minimum guaranteed speed of session i, that is, the minimum speed of queue i, and r is the maximum output link Capacity, which is the service speed of the server, is generally a fixed value):

$$g_i(t_1, t_2) \geq r_i = \frac{\emptyset_i}{\sum_{j \in B} \emptyset_j} r$$

The WFQ algorithm is an approximation of the GPS algorithm. Assuming that Fp is the departure time of the data packet P in the GPS algorithm, then the WFQ algorithm is a job reservation algorithm that simulates the GPS algorithm and schedules data packets in ascending order, that is, the WFQ algorithm always selects the data packet with the smallest Fp value for scheduling. The following is the virtual time implementation of the WFQ algorithm.

Define the arrival or departure of each data packet as an event, set the time when the jth event occurs as tj, and the scheduler sends the first data packet (for example, after a period of time when all queues are idle, a data packet arrives) Time t1 = 0. , The processing rate of the server is r. It can be seen that the data flow in the active state in the time interval (tj-1, tj) is fixed, and we denote it as the set Bj. Define the virtual time function V(t), V(t) is 0 when the scheduler is idle (all queues are empty). In any scheduler busy period, there is:

$$\begin{cases} V(0) = 0 \\ V(t_{j-1} + \tau) = V(t_{j-1}) + \dfrac{\tau}{\sum_{j \in B} \emptyset_i} \end{cases} \tag{1}$$

$\tau \in t_j - t_{j-1}, j = 2,3 \ldots,$ The rate of change of V is $\frac{\delta V \ (t_{j-1}+\tau)}{\delta r} = \frac{1}{\sum_{j \in B} \emptyset_i}$, and for the queue i of each backlog session, the service rate it gets is$\emptyset_i \frac{\delta V \ (t_{j-1}+\tau)}{\delta r}$. So also write

$$\begin{cases} V(0) = 0 \\ V(t_{j-1} + \tau) = V(t_{j-1}) + \dfrac{r\tau}{\sum_{j \in B} r_i} \end{cases} \tag{2}$$

In this sense, the virtual time can be interpreted as a function of increasing the rate of service received by all the backlog of conversation queues as the edge rate. Suppose that the k-th data packet of the i-th session arrives in the queue at time aik, and its length is Lki. Then, the virtual time of the arrival time of this data packet is recorded as Ski, and the virtual time of the departure time of this data packet is recorded as Fki. For all i, define F0i = 0, then

$$\begin{cases} S_i^k = max\{F_i^{k-1}, V(a_i^k)\}, \\ F_i^k = S_i^k + \frac{L_i^k}{\emptyset_i} \end{cases} \tag{3}$$

So also write

$$\begin{cases} S_i^k = max\{F_i^{k-1}, V(a_i^k)\}, \\ F_i^k = S_i^k + \frac{L_i^k}{r_i} \end{cases} \tag{4}$$

2 Materials

Pycharm software was used to compile the code, and all the code versions were python3.8. The simulation used simpy to carry out the process-based discrete-event simulation

3 Methods, result and discussion

#1

Consider a router running WFQ with one common 1-kB/s output line and four incoming flows whose traffic patterns are as follows:

Input Line	A	B	C	D
Mean Packet Interarrival Time [s]	1	2	4	3
Packet Length [kB]	2	4	8	6

WFQ whose flow weights are given blow.

Input Line	A	B	C	D
Weight	1	2	3	4

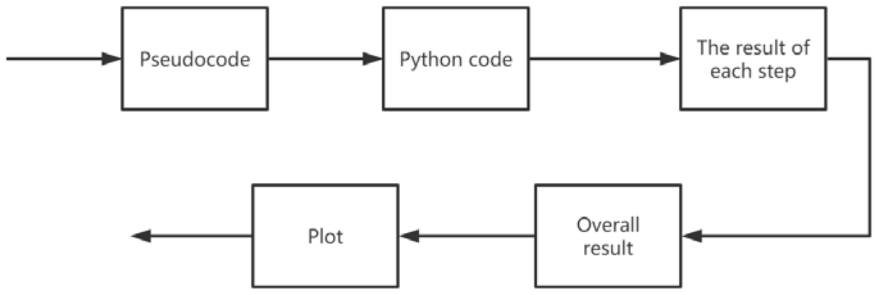
Packet interarrival times are exponentially distributed, and packet lengths are fixed.

Implement and carry out a simulation experiment based on the above as follows:

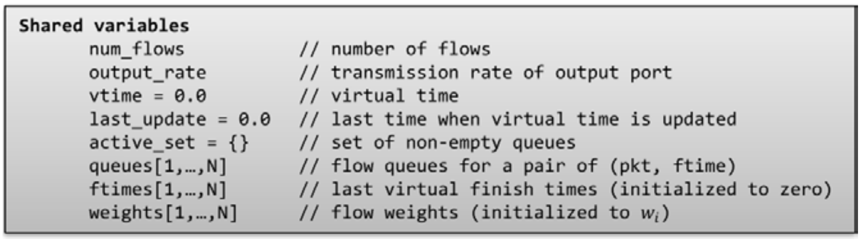
- [15 points] Run a simulation of a WFQ system consisting of packet generators, a router, and a packet sink for 1-hour in simulation time; provide the program source code and describe it.
- Measure and plot flow throughputs at the packet sink during the simulation for the following two cases and plot a time series of them (i.e., two time series plots in total):
  - [10 points] Every minute (i.e., total 60 measurements per flow).
  - [10 points] Every 10 minutes (i.e., total 6 measurements per flow).

Method

The overall idea is:



Part1: Shared variables



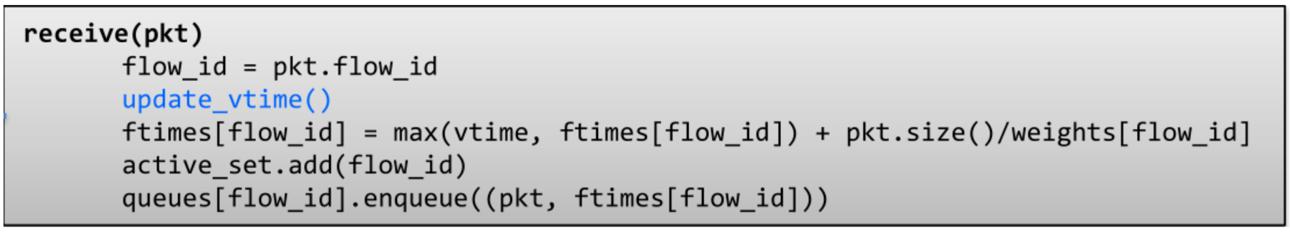
```
Shared variables
        num_flows            // number of flows
        output_rate          // transmission rate of output port
        vtime = 0.0          // virtual time
        last_update = 0.0    // last time when virtual time is updated
        active_set = {}      // set of non-empty queues
        queues[1,…,N]        // flow queues for a pair of (pkt, ftime)
        ftimes[1,…,N]        // last virtual finish times (initialized to zero)
        weights[1,…,N]       // flow weights (initialized to $w_i$)
```

Code:

```
def __init__(self, env, num_inputs, output_rate, trace=False):
    self.env = env
    self.num_inputs = num_inputs
    self.output_rate = output_rate
    self.trace = trace
    self.pkt_container = simpy.Container(env)  # as a counter for all packets
```

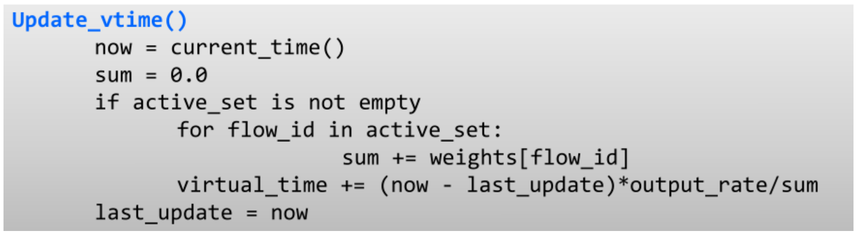
```python
        self.flow_queues = []
        for i in range(num_inputs):
            self.flow_queues.append(deque())
        self.out = None
        self.last_flow_id = 0  # flow ID served during the last round


        self.vtime = 0              # virtual time
        self.last_update = 0.0      # last time when virtual tim
        self.active_set = set()     # set of non-empty queues
        self.ftimes = [0, 0, 0, 0]
        self.finish_time_list = [float('inf'), float('inf'), float('inf'), float('inf')]

        self.weights = [1, 2, 3, 4]  # WFQ
        # self.weights = [1,1,1,1]   # FQ


        self.action = env.process(self.run())  # start the run process when an instance is created
```

Part2: receive



```
receive(pkt)
        flow_id = pkt.flow_id
        update_vtime()
        ftimes[flow_id] = max(vtime, ftimes[flow_id]) + pkt.size()/weights[flow_id]
        active_set.add(flow_id)
        queues[flow_id].enqueue((pkt, ftimes[flow_id]))
```

Update_vtime():

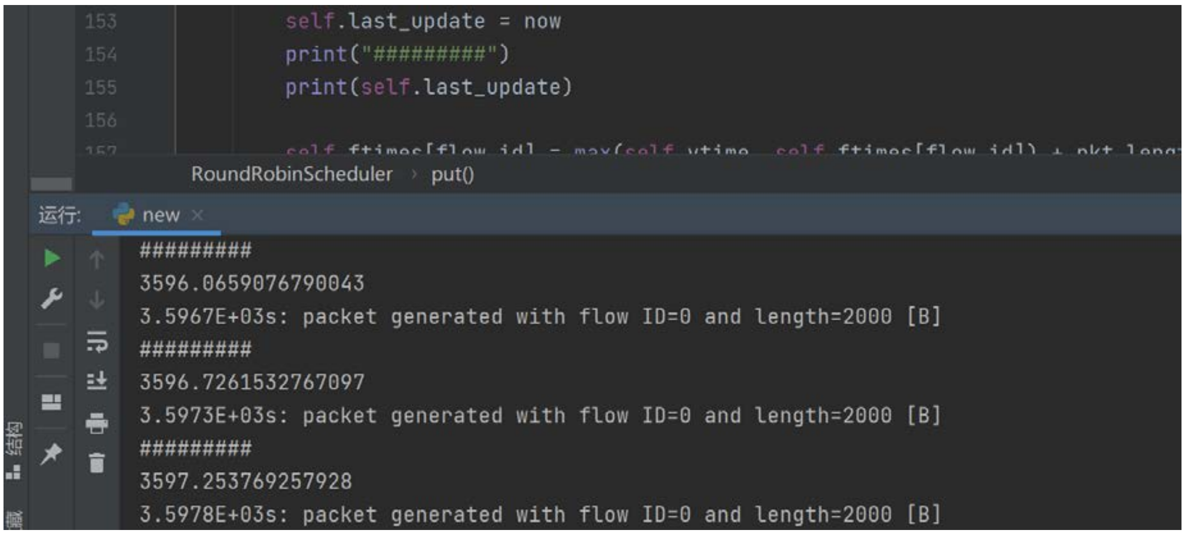


```
Update_vtime()
        now = current_time()
        sum = 0.0
        if active_set is not empty
                for flow_id in active_set:
                                sum += weights[flow_id]
                virtual_time += (now - last_update)*output_rate/sum
        last_update = now
```

Code：

```python
    #update virtual time
    now = self.env.now
    sum_temp = 0.0
    if self.active_set:
        for flow_id_temp in self.active_set:
            sum_temp += self.weights[flow_id_temp]
        self.vtime += (now - self.last_update) * self.output_rate / sum_temp
    self.last_update = now
```

Result：



```python
            self.last_update = now
            print("#########")
            print(self.last_update)

            self.ftimes[flow_id] = max(self.vtime, self.ftimes[flow_id]) + pkt.leng
```

```
#########
3596.0659076790043
3.5967E+03s: packet generated with flow ID=0 and length=2000 [B]
#########
3596.7261532767097
3.5973E+03s: packet generated with flow ID=0 and length=2000 [B]
#########
3597.253769257928
3.5978E+03s: packet generated with flow ID=0 and length=2000 [B]
```

At this time, what **self.last_update** contains is the time of each packet that entered the queue last

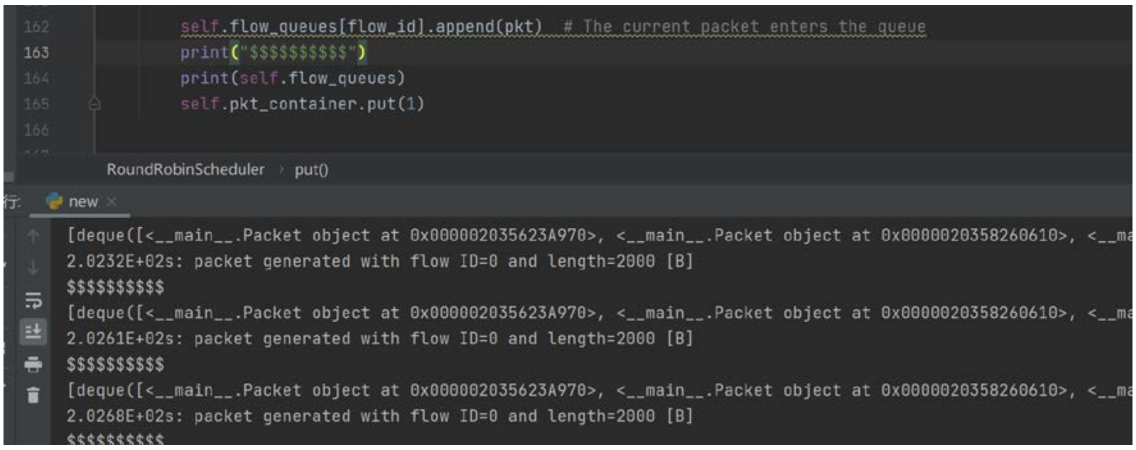
Receive part besides update virtual time

```
ftimes[flow_id] = max(vtime, ftimes[flow_id]) + pkt.size()/weights[flow_id]
active_set.add(flow_id)
queues[flow_id].enqueue((pkt, ftimes[flow_id]))
```

Code：

```python
    self.ftimes[flow_id] = max(self.vtime, self.ftimes[flow_id]) + pkt.length / self.weights[flow_id]
    pkt.v_finish = self.ftimes[flow_id]  # The virtual completion time of the current package
    self.active_set.add(flow_id)

    self.flow_queues[flow_id].append(pkt)  # The current packet enters the queue
    self.pkt_container.put(1)
```

Result:



```
        self.flow_queues[flow_id].append(pkt)  # The current packet enters the queue
        print("$$$$$$$$$$")
        print(self.flow_queues)
        self.pkt_container.put(1)
```

```
[deque([<__main__.Packet object at 0x000002035623A970>, <__main__.Packet object at 0x0000020358260610>, <__ma
2.0232E+02s: packet generated with flow ID=0 and length=2000 [B]
$$$$$$$$$$
[deque([<__main__.Packet object at 0x000002035623A970>, <__main__.Packet object at 0x0000020358260610>, <__ma
2.0261E+02s: packet generated with flow ID=0 and length=2000 [B]
$$$$$$$$$$
[deque([<__main__.Packet object at 0x000002035623A970>, <__main__.Packet object at 0x0000020358260610>, <__ma
2.0268E+02s: packet generated with flow ID=0 and length=2000 [B]
$$$$$$$$$$
```

You can see that at this time **self.flow_queues** contains the information of the incoming packet

Part3: Send

```
send()
        flow_id = select_queue()
        pkt = (queues[flow_id].dequeue())[0]
        if queues[flow_id] is empty
                active_set.remove(flow_id)

        // reset virtual time and reinitialize last
        // finish times when there is no active flow.
        if active_set is empty
                vtime = 0.0
                ftimes = [0.0,…,0.0]

        // send the packet
```

Select_queue:

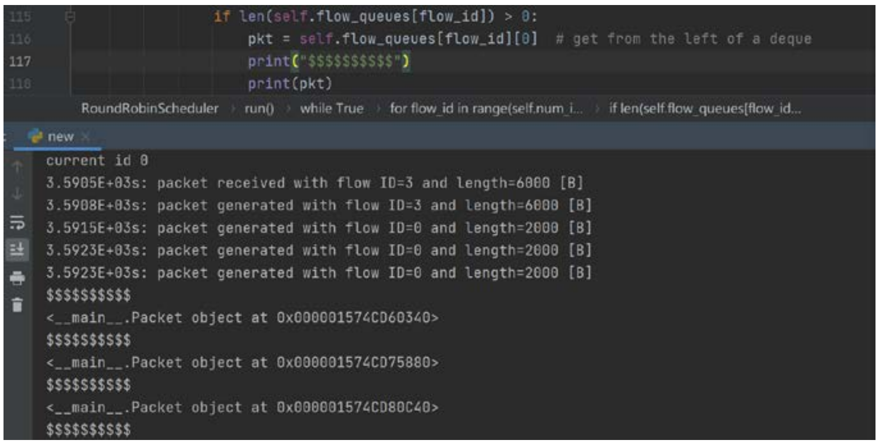
```
select_queue()
        // - check the virtual finish times of the packets at the
        //   head of all non-empty queues.
        // - select the minimum virtual finish time.
        // - return the ‘flow_id’ of the corresponding queue.
```

Code:

```
"""
# select queue
"""
for flow_id in range(self.num_inputs):
    if len(self.flow_queues[flow_id]) > 0:
        pkt = self.flow_queues[flow_id][0]  # get from the left of a deque
        self.packet_v_finish[flow_id] = pkt.v_finish

flow_id = self.packet_v_finish.index(min(self.packet_v_finish))
pkt = self.flow_queues[flow_id].popleft()  # Propose a package to send
```

Result:

- pkt:



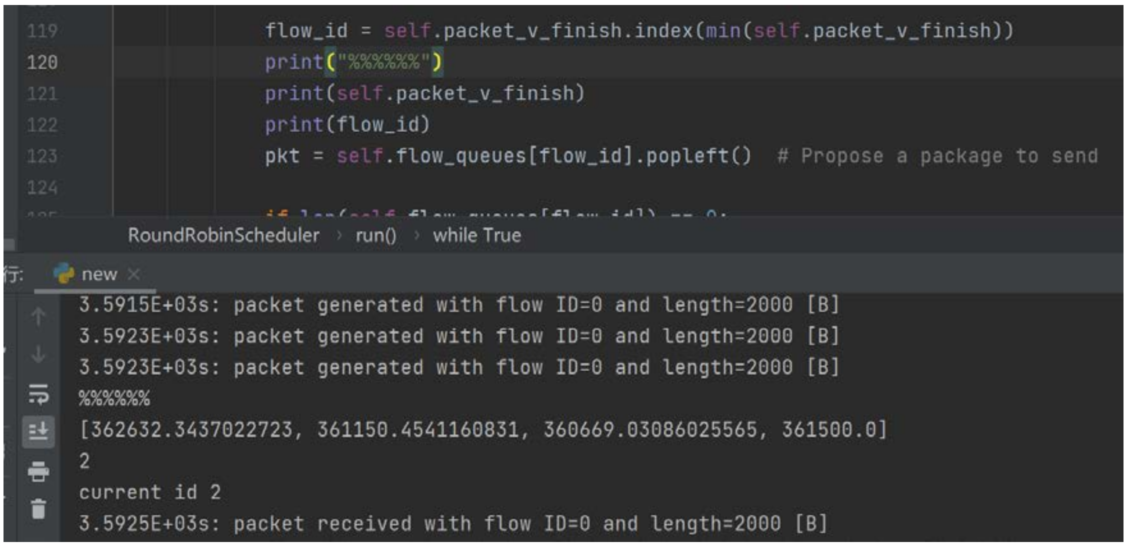
```
if len(self.flow_queues[flow_id]) > 0:
    pkt = self.flow_queues[flow_id][0]  # get from the left of a deque
    print("$$$$$$$$$$")
    print(pkt)
```

RoundRobinScheduler › run() › while True › for flow_id in range(self.num_i... › if len(self.flow_queues[flow_id...
new

```
current id 0
3.5905E+03s: packet received with flow ID=3 and length=6000 [B]
3.5908E+03s: packet generated with flow ID=3 and length=6000 [B]
3.5915E+03s: packet generated with flow ID=0 and length=2000 [B]
3.5923E+03s: packet generated with flow ID=0 and length=2000 [B]
3.5923E+03s: packet generated with flow ID=0 and length=2000 [B]
$$$$$$$$$$
<__main__.Packet object at 0x000001574CD60340>
$$$$$$$$$$
<__main__.Packet object at 0x000001574CD75880>
$$$$$$$$$$
<__main__.Packet object at 0x000001574CD80C40>
$$$$$$$$$$
```

You can clearly see the information of each package taken out in **pkt**

- flow id



```
flow_id = self.packet_v_finish.index(min(self.packet_v_finish))
print("%%%%%%")
print(self.packet_v_finish)
print(flow_id)
pkt = self.flow_queues[flow_id].popleft()  # Propose a package to send
```

```
3.5915E+03s: packet generated with flow ID=0 and length=2000 [B]
3.5923E+03s: packet generated with flow ID=0 and length=2000 [B]
3.5923E+03s: packet generated with flow ID=0 and length=2000 [B]
%%%%%%
[362632.3437022723, 361150.4541160831, 360669.03086025565, 361500.0]
2
current id 2
3.5925E+03s: packet received with flow ID=0 and length=2000 [B]
```

We can clearly see that in **self.packet_v_finish** is each virtual finish time in [0,1,2,3], we choose a smallest queue to be sent, that is, the earliest completed queue is the sending queue.

Then we use index sorting when selecting, and select the queue number with the earliest completion time.

Send part besides select queue

Code:

```
if len(self.flow_queues[flow_id]) == 0:
    self.active_set.remove(flow_id)

```

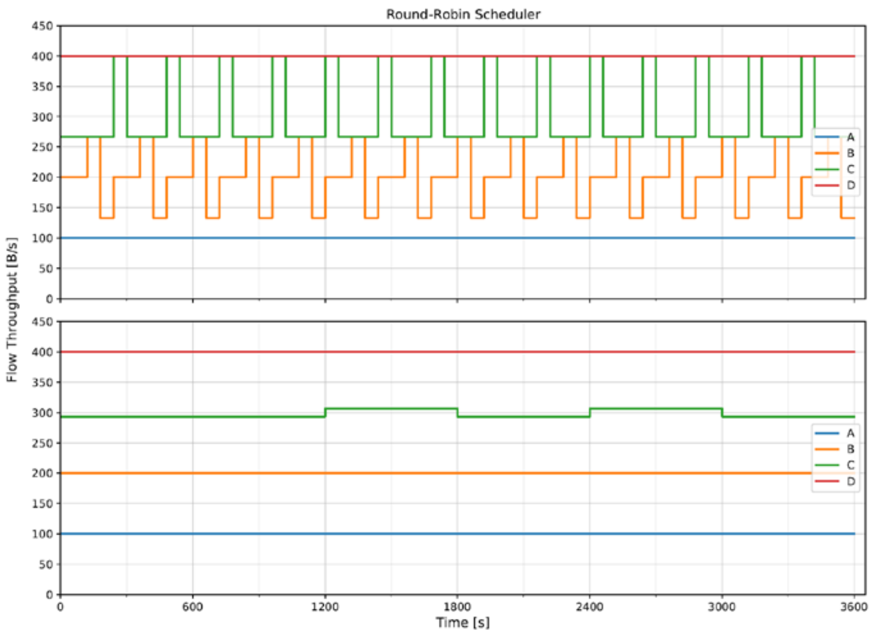
```
# reset the virtual time
if not self.active_set:
    self.vtime = 0
    self.ftimes = [0, 0, 0, 0]

for i in range(4):
    self.packet_v_finish[i] = float('inf')

print('select send package queue id is : ', flow_id)
```

Part4: Result

Code:

```
# obtain flow throughputs for two different measurement intervals
mis = [60, 600]
times = [None] * 2
flow_throughputs = [None] * 2
for i in range(2):
    times[i], flow_throughputs[i] = run_simulation(sim_time=sim_time,
                                                   random_seed=random_seed,
                                                   mi=mis[i],
                                                   trace=trace)
```

Result:



#2

Consider a router running FQ with one common 1-kB/s output line and four incoming flows whose traffic patterns are as follows:

Input Line	A	B	C	D
Mean Packet Interarrival Time [s]	1	2	4	3
Packet Length [kB]	2	4	8	6

Packet interarrival times are exponentially distributed, and packet lengths are fixed.

Implement and carry out a simulation experiment based on the above as follows:

- [15 points] Run a simulation of an FQ system consisting of packet generators, a router, and a packet sink for 1-hour in simulation time; provide the program source code and describe it.
- Measure and plot flow throughputs at the packet sink during the simulation for the following two cases and plot a time series of them (i.e., two time series plots in total):
  - [10 points] Every minute (i.e., total 60 measurements per flow).
  - [10 points] Every 10 minutes (i.e., total 6 measurements per flow).

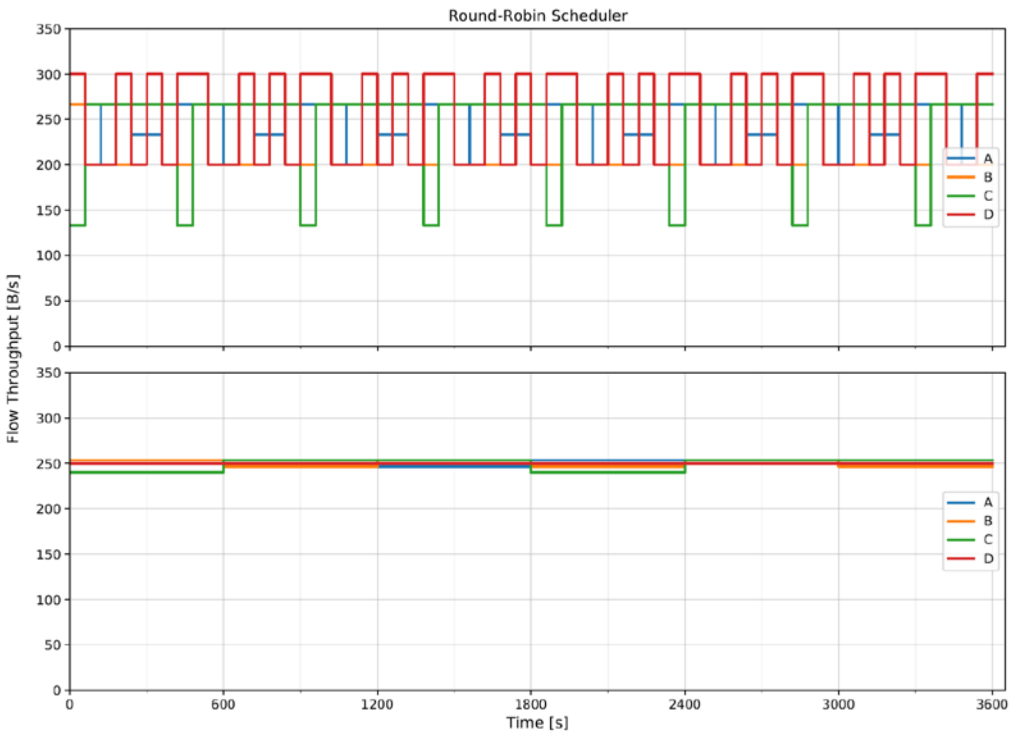
In fact, WFQ is a common situation, and FQ is a special WFQ situation, that is, each weight is 1, which constitutes an absolutely fair queue.

We only need to change **weight** from [1,2,3,4] to [1,1,1,1] to change WFQ to FQ. Other codes are the same as WFQ.

Code:

```
self.vtime = 0
self.last_update = 0.0
self.active_set = set()
self.ftimes = [0, 0, 0, 0]
self.packet_v_finish = [float('inf'), float('inf'), float('inf'), float('inf')]
# self.weights = [1, 2, 3, 4]  # WFQ
self.weights = [1,1,1,1]   # FQ
```

Result:

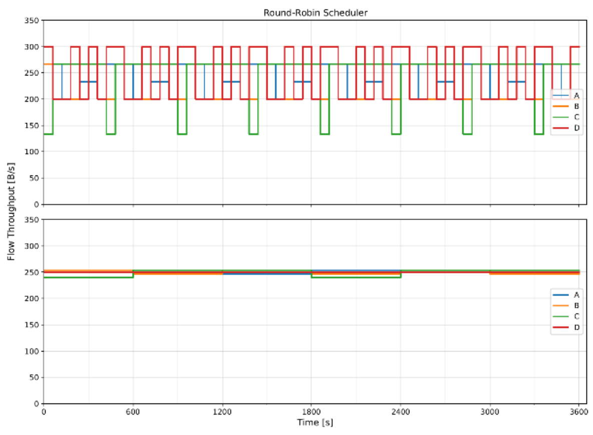


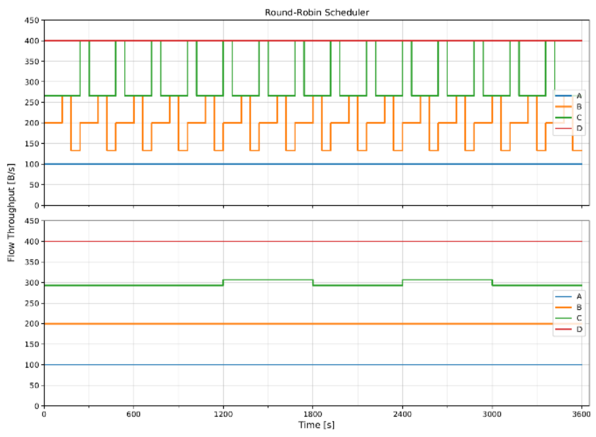
#3

Discuss the following based on the simulation results above:

• [15 points] If there are any differences between expected flow throughputs and measured ones during the simulation, discuss possible reasons for those differences (for both FQ and WFQ).

Our expected result, according to the theoretical situation, the image should be a straight line,





but we can see that there are many fluctuations in the result of our simulation, as shown in the figure:

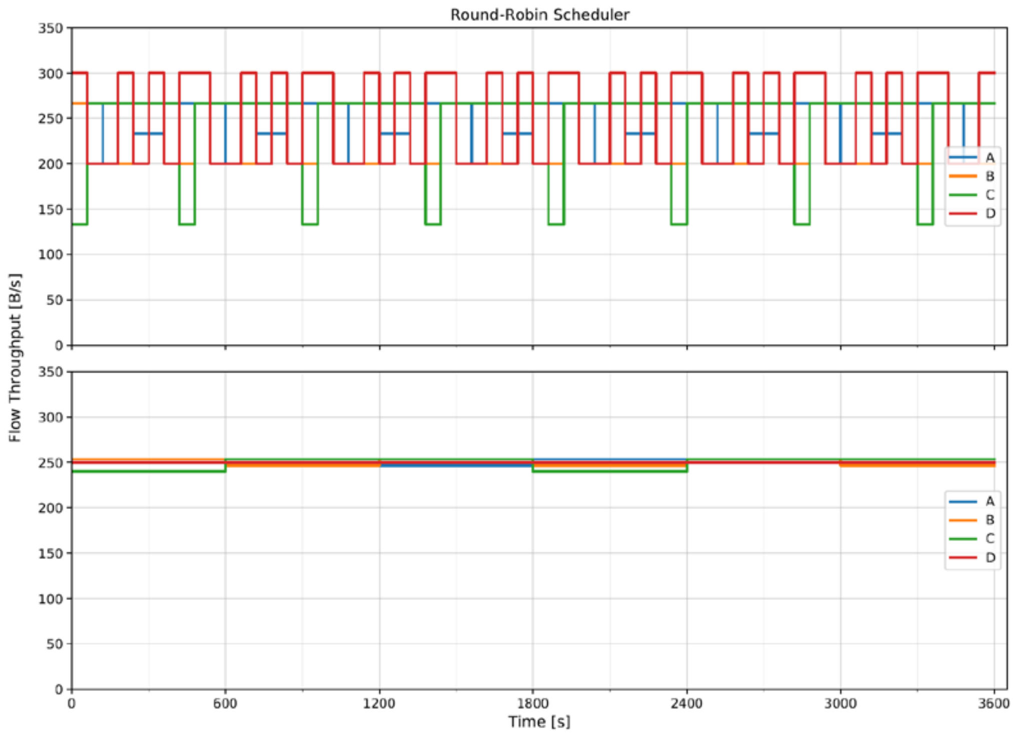
Then we can find some reasons for the fluctuations by reading Paper 《*Analysis and Simulation of a Fair Queueing Algorithm*》:

The most obvious flaw is its lack of con- sideration of packet lengths. A source using long packets gets more bandwidth than one using short packets, so bandwidth is not allocated fairly.

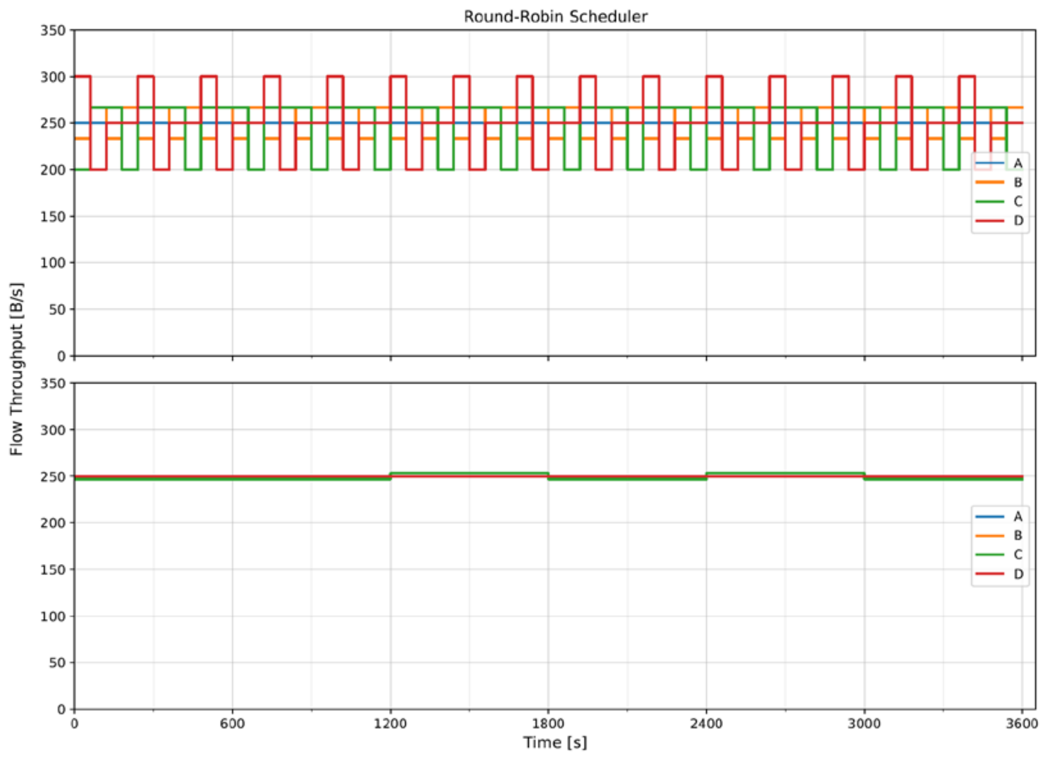
We can analyze that the reason that may affect the simulation results is the length of the packet. Let's test the length of the report below and analyze the results.

Here we are using FQ to test, and select *The Every 10 minutes (i.e., total 6 measurements per flow)* last measurement scale.

In the original program, the packet length is 1000(bytes), the simulation results are as follows:



packet length is 1000(bytes)



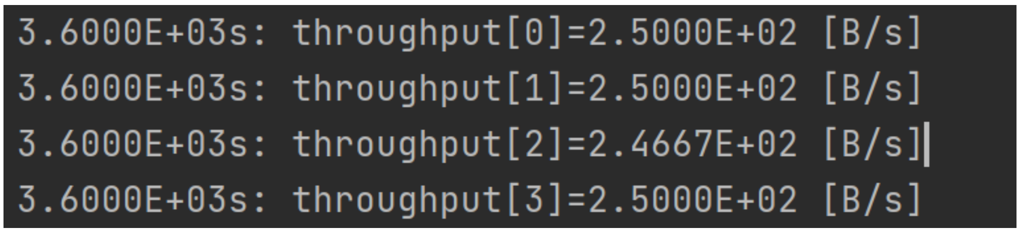
packet length is 500(bytes)

Now, we can be sure that **packet length** does affect the volatility, then we need to find an indicator to measure the volatility. In the expected situation, FQ we hope that the service time of each queue is fixed, that is, between the service time of the four queues The variance of is 0. In the program, we count the service time of each queue:

Code:

```
def update_stats(self):
    while True:
        yield self.env.timeout(self.mi)
        now = self.env.now
        self.times.extend([self.last_mt, now])
        throughput = self.bytes_rcvd / self.mi
        self.throughputs.extend([throughput] * 2)
        self.bytes_rcvd = 0
        self.last_mt = now
        if self.trace:
            print(f"{now:.4E}s: throughput[{self.flow_id:d}]={throughput:.4E} [B/s]"
)
```

Result:

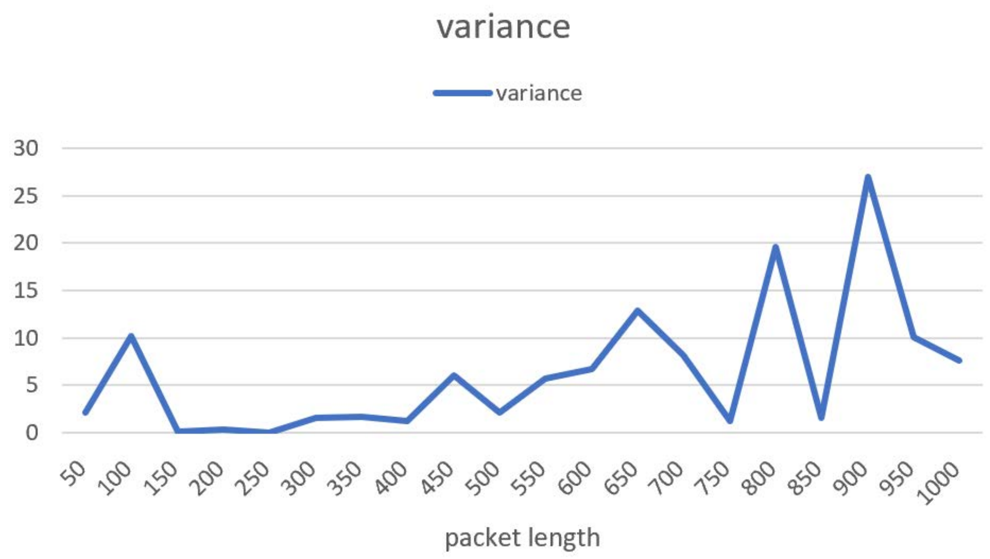


```
3.6000E+03s: throughput[0]=2.5000E+02 [B/s]
3.6000E+03s: throughput[1]=2.5000E+02 [B/s]
3.6000E+03s: throughput[2]=2.4667E+02 [B/s]
3.6000E+03s: throughput[3]=2.5000E+02 [B/s]
```

For more convenient calculation, we will change the output code to

```
print(f"{now:.4E}s: throughput[{self.flow_id:d}]={throughput:.4E} [B/s]")
```

Therefore, we can calculate the variance of the service time of the four queues corresponding to different **packet length** in the case of FQ.

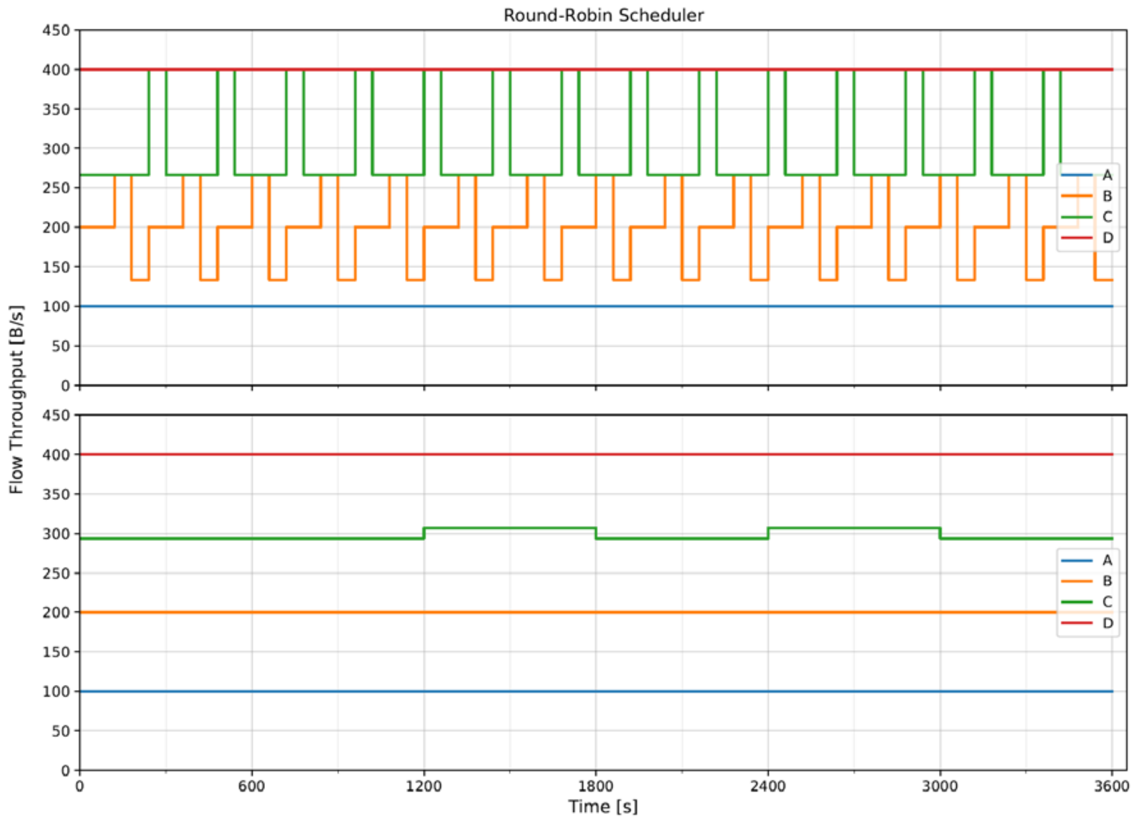


We can see that the influence of **packet length** on the simulated value is real, but through the line graph we can find that the influence of **packet length** that is too small or too large on the simulated value is large.

Analyzing the principle, the possible reason for this situation is that the continuous large-volume flow "monopolizes" the bandwidth, and the small-volume flow will cause a large delay. Therefore, choosing a suitable **packet length** is crucial for WFQ.

- [15 points] Discuss the effect of the measurement duration on the flow throughputs, if any.

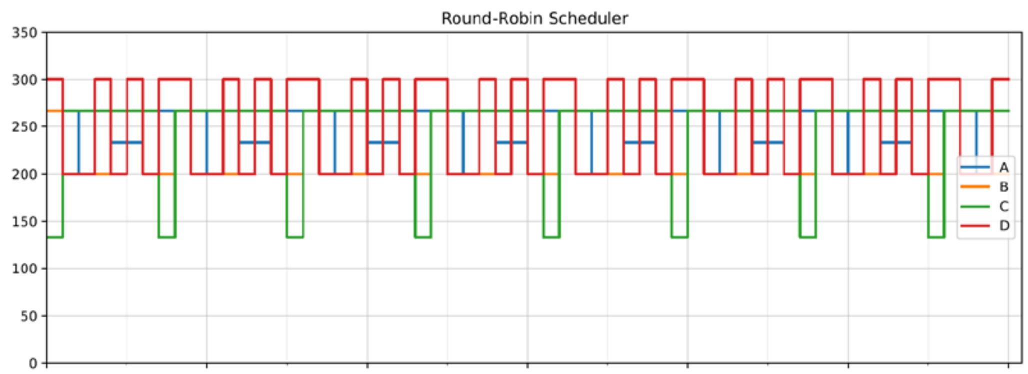
Let us first look at the most intuitive reflection of the measurement duration on the results. We use the WFQ model, weight control is [1,2,3,4], first use the *task3 #1* result



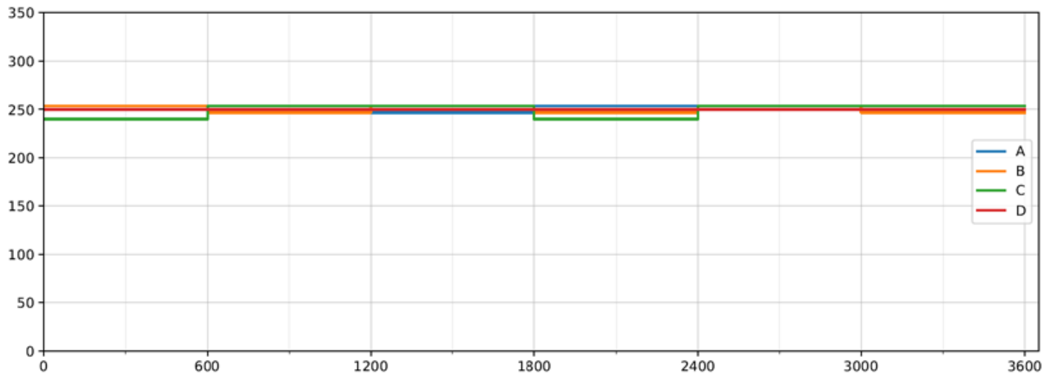
The measurement scale of the upper half is 60s, and the measurement scale of the lower half is 600s.

We found that the image corresponding to 600 seconds has fewer fluctuations, and there is no 60-second image fluctuation pattern.

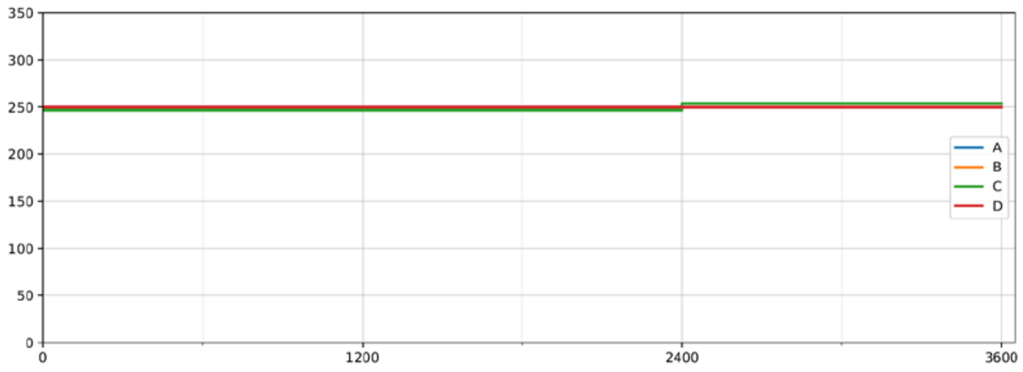
Below we use 60s as the basic group, and compare 600 seconds, 1200 seconds, and 3600 seconds respectively. Because the total time is 3600 seconds, we choose to show the duration of the complete process within 3600 seconds.



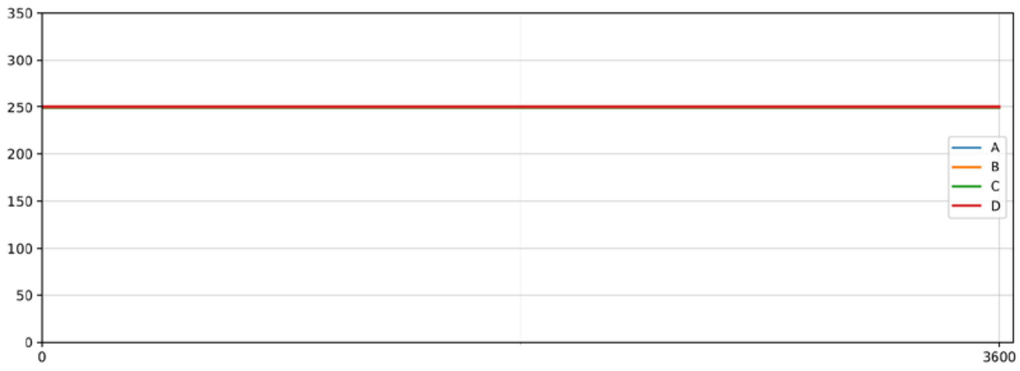
measurement duration = 60



measurement duration = 600



measurement duration = 1200



measurement duration = 3600

Let's look at the definition of throughputs:

```
def update_stats(self):
    while True:
        yield self.env.timeout(self.mi)
        now = self.env.now
        self.times.extend([self.last_mt, now])
        throughput = self.bytes_rcvd / self.mi
        self.throughputs.extend([throughput] * 2)
        self.bytes_rcvd = 0
        self.last_mt = now
        if self.trace:
            print(f"{now:.4E}s: throughput[{self.flow_id:d}]={throughput:} [B/s]")
```

It can be seen that the definition of A is that the flow through the measurement time is higher than the measurement time.

This is similar to an average. The more flows and the longer the time, the fluctuations in it may be more offset.

I personally feel that this problem is somewhat similar to the sampling theorem. The longer the sampling time, the fewer sampling points, so the situation reflected after sampling is not close to the real situation.

4 Discussion and improvement

1 active_set.remove

In FQ, because the weight is all 1, it can be omitted:

```
for flow_id in self.active_set:
    sum += self.weights[flow_id]
```

These two lines of code are only effective for the normal case of WFQ weight, and have no effect on FQ.

2 ‘inf’

In the code, when initializing, we are not initializing to 0, but to infinity, so that when looking for the minimum value, we can avoid the situation where 0 is the minimum due to no operation, which increases the robustness of the system .

5 Conclusion

In this experiment, we used the simpy generator to simulate WFQ and special case FQ. We observed the simulated results, but there are still differences from the ideal expectations. This is caused by the shortcomings of WFQ. In the experiment It is very important to choose a suitable packet length. And in order to better observe the simulation results, the duration of the measurement will also affect the observation.

REFERENCES

[1] A. Demers, S. Keshav, and S. Shenker, “Analysis and simulation of a fair queueing algorithm,” SIGCOMM Comput. Commun. Rev., vol. 19, no. 4, pp. 1–12, 1989.

[2] A. K. Parekh and R. G. Gallager, “A generalized processor sharing approach to flow control in integrated services networks: The single node case,” IEEE/ACM Trans. Netw., vol. 1, no. 3, pp. 344–357, Jun. 1993.

[3] Wikipedia. (2020) Fair queueing. [Online; accessed 26-October-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Fair_queuing

[4] ——. (2020) Weighted fair queueing. [Online; accessed 26-October-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Weighted_fair_queuing