

EE5150: Communication Networks

Programming Assignment-2 Report

Ishaan Agarwal
EE20B046

May 15, 2023

1 Introduction

In this problem, we shall simulate a simplified version of a scheduler that has the essence of TCP. Using that we shall try to understand the interplay between schedulers like max-weight and TCP. We do not consider dupACKs for simplicity of the problem.

2 Interplay between the MAC layer and a TCP like transport layer

We start by defining the given binomial and bernoulli random variables corresponding to the service and some arrivals. This is done as follows:

```
1 def service(N, K):  
2     #This is a binomial probability distribution with N trials  
3     #and K successes  
4     #The probability of success is k/N  
5     return np.random.binomial(N, K/N)  
6 def arrival_bernoulli(p):  
7     #This is a Bernoulli distribution with probability p  
8     return np.random.choice([0,1], p=[1-p, p])
```

The service capacity is $\text{Binomial}(N, K/N)$, where $N > K$, this means that the expected number of packets served in one time slot is K , note that we only have K flows, and since the expected number of packets served is itself K , this hints to the fact that we might not have pretty large queues at any given time.

We are given K flows, R out of which are $\text{Bernoulli}(p)$, and the remaining $K - R$ are also independent Bernoulli, but their rates change with time.

2.1 Average Delay update

For each of these $K - R$ flows maintain average delay in the following way. For flow i , if the average delay up to packet n is $T_n^{(i)}$ and the delay (from arrival to service) of the $n + 1$ th packet is $D_{n+1}^{(i)}$ then update the average delay as $T_{n+1}^{(i)} = 0.9T_n^{(i)} + 0.1D_{n+1}^{(i)}$. This is done using this function:

```
1 def average_delay_update(i, R, Tn, Dnplusone, n):
2     #This is the update equation for the average delay
3     #Tn is the previous average delay
4     #Dnplusone is the new delay
5     if (Tn == 0):
6         return Dnplusone
7     if (i < R):
8         return (Tn*n+Dnplusone)/(n+1)
9     return 0.9*Tn + 0.1*Dnplusone
```

2.2 Arrival Rate update

For each of the K-R flows, the arrival rate is not constant, in fact the arrival rate changes dynamically with time, this is in close resemblance with the TCP protocols that we studied in class, where we performend "Window flow control" and varied the window size W , dynamically with time. The scheme is as follows:

Suppose the arrival rate for flow i at time t is $p^{(i)}(t)$ and $n^{(i)}(t)$ be the number of packets served so far. If at time t there is at least one un-served packet that arrived on or before $t - \lceil 1.2T_{n(t)}^{(i)} \rceil$, update $p^{(i)}(t+1) = \frac{2p^{(i)}(t)}{3}$.

If at time t , r packets are served and all of them arrived after $t - \lceil 1.2T_{n(t)}^{(i)} \rceil$, then update $p^{(i)}(t+1) = \min(0.9, p^{(i)}(t) + \delta)$ for $0 < \delta < 1$. δ has been used as 0.01 in this report. Generate an arrival in slot $t+1$ for this flow according to Bernoulli $p^{(i)}(t+1)$.

This is done using the following function:

```
1 def arrival_general_update(i, p, n, t, T, R, arrival_times,
2     service_times, delta):
3     #This is the update equation for the probability of arrival
4     #p is the previous probability of arrival
5     #n is the number of packets served so far for this flow
6     #T is the average delay for this flow
7     #arrival_times is the list of arrival times for this flow
8     #service_times is the list of service times for this flow
9     #This function returns the new probability of arrival
10    if n == 0:
11        return p
12    elif i < R:
```

```

12         return p
13     elif len(arrival_times) > len(service_times):
14         if (arrival_times[len(service_times)] <= t - np.ceil
15             (1.2*T)):
16             return 2*p/3
17         #for the 2nd case,
18         #the first packet served in this time slot should have
19         arrived
20         #at most 1.2*T time slots ago
21     else:
22         for i in range(len(service_times)):
23             if service_times[i] == t:
24                 if arrival_times[i] > t - np.ceil(1.2*T):
25                     return min(0.9, p+delta)
26                 else:
27                     break
28     return p

```

2.3 Initialiser Code

We initialise all the variable values. Most of the variable names are self-explanatory. Let us start by taking $p = 0.5$, and simulate for $T = 1000$ time slots.

```

1 N=30
2 K=20
3 R=15
4 T = 1000
5 p=0.5
6 #One dimensional array for the number of packets in the queue
7 queue_size = [0 for i in range(K)]
8 #Two dimensional arrays for the arrival times and service times
9 arrival_times = [[] for i in range(K)]
10 service_times = [[] for i in range(K)]
11 #One dimensional arrays for the current average delay and
12     current probability of arrival
13 average_delay = [0 for i in range(K)]
14 #probability_of_arrival is p for each flow in the beginning
15 probability_of_arrival = [p for i in range(K)]
16 #Two dimensional arrays for the number of packets served
17 number_of_packets_served = [[0 for i in range(T)] for j in
18     range(K)]
19 #One dimensional array for server utilization
20 server_utilization = [0 for i in range(T)]

```

We define throughput for a flow as the total number of packets served for that flow divided by the total time T .

2.4 Scheme 1: Processor Sharing

In this scheme, we try to distribute the load as uniformly as possible, and thus we find an l such that $S(t) > lK$ and $S(t) \leq l(K + 1)$. We then serve

l packets from each flow first and then serve the remaining capacity in a round robin fashion one packet at a time from each flow.

Consider $S = 11$, queue sizes: $Q = (7, 3, 2, 1)$, then after this time slot, the queue sizes would be: $Q = (2, 0, 0, 0)$.

- Step 1: $l = 2$. Serve l packets from each queue. (if possible) $Q = (5, 1, 0, 0)$ and $S=4$.
- Step 2: Serve one packet from each queue. (round robin) $Q = (4, 0, 0, 0)$ and $S=2$.
- Step 3: Round Robin continues: $Q = (2, 0, 0, 0)$ and $S=0$.

The simulation is as follows:

```

1 #Scheme 1: Processor Sharing
2 for t in range(T):
3     #defining arrivals
4     for i in range(K):
5         if(arrival_bernoulli(probability_of_arrival[i])):
6             arrival_times[i].append(t)
7             queue_size[i] += 1
8     #total service capacity
9     S = service(N, K)
10    S_copy = S
11    #finding 'l' value as per the scheme:
12    for l in range(0,100):
13        if(S > l*K and S <= (l+1)*K):
14            break
15
16    #servicing
17    #serve l packets from each flow first, then serve one
18    #additional packet
19    #from each flow in a round robin fashion until S packets
20    #are served
21    for i in range(K):
22        for j in range(l):
23            if(queue_size[i] > 0):
24                service_times[i].append(t)
25                queue_size[i] -= 1
26                number_of_packets_served[i][t] += 1
27                S -= 1
28    #round robin
29    i=0
30    while(S > 0):
31        if(queue_size[i] > 0):
32            service_times[i].append(t)
33            queue_size[i] -= 1
34            number_of_packets_served[i][t] += 1
35            S -= 1
36            i = (i+1)%K

```

```

35         if(i == 0):
36             break
37     #server utilization
38     if(S_copy == 0):
39         server_utilization[t] = 0
40     server_utilization[t] = (S_copy - S)/S_copy
41
42     #updating average delay and probability of arrival for each
    flow
43     for i in range(K):
44         if(number_of_packets_served[i][t] > 0):
45             #total packets served so far
46             total = 0
47             for j in range(t-1):
48                 total = total + number_of_packets_served[i][j]
49             for j in range(number_of_packets_served[i][t]):
50                 average_delay[i] = average_delay_update(i, R,
51                 average_delay[i], t - arrival_times[i][total + j], total+j)
52                 probability_of_arrival[i] = arrival_general_update(
53                 i, probability_of_arrival[i], total, t, average_delay[i], R
54                 , arrival_times[i], service_times[i], 0.01)
55
56     #plotting moving average of server utilization
57     moving_average = [0 for i in range(T)]
58     for i in range(T):
59         total = 0
60         for j in range(20):
61             if(i-j >= 0):
62                 total = total + server_utilization[i-j]
63         moving_average[i] = total/20
64     plt.plot(moving_average, 'r')
65     plt.xlabel('Time')
66     plt.ylabel('Server Utilization')
67     plt.title('Server Utilization vs Time for Processor Sharing')
68     plt.show()
69
70     #throughput and average delay for each flow
71     throughput = [0 for i in range(K)]
72     for i in range(K):
73         throughput[i] = len(service_times[i])/T
74
75     throughput_processor_sharing = throughput
76     average_delay_processor_sharing = average_delay
77     server_utilization_processor_sharing = server_utilization
78     moving_average_processor_sharing = moving_average

```

Here, we have plotted the moving average of server utilisation versus time to see how efficient the scheme is.

The obtained output is as follows:

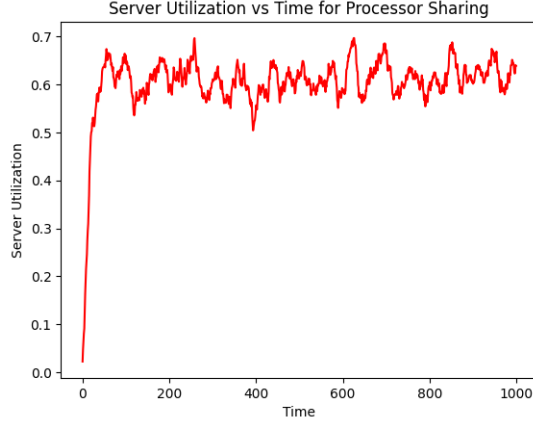


Figure 1: Processor sharing server utilisation

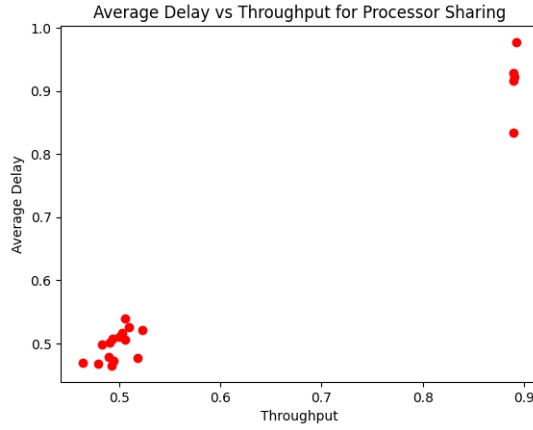


Figure 2: Average Delay vs Throughput

2.5 Scheme 2: Water Filling or minimise the max queue

The scheme is as follows: Let flow i have $Q_i(t)$ number of packets at time t . Serve the flows in such a way that at $t + 1$, the gap $\max_i Q_i(t + 1) - \min_i Q_i(t + 1)$ is minimized.

There is an iterative way of doing this, let's say my service capacity for this time slot is S . Then until $S=0$, I can go on decreasing S by 1 by serving 1 packet from the current maximum queue at every step. Since, at every step, I am making the greedy choice of minimising the gap $\max_i Q_i(t + 1) - \min_i Q_i(t + 1)$, then after all S capacity has been used, the current queues are the optimal queues and the packets served are optimal as per the water filling scheme.

Consider $S = 11$, queue sizes: $Q = (7, 3, 2, 1)$, then after this time slot, the queue sizes would be: $Q = (0, 0, 1, 1)$.

- Step 1: Queue 1 is max, keep serving it till it is max: $Q = (2, 3, 2, 1)$ and $S=6$.
- Step 2: Queue 2 is max: $Q = (2, 1, 2, 1)$ and $S=4$.
- Step 3: Continue the same: $Q = (0, 0, 1, 1)$ and $S=0$.

This is done in code as follows:

```

1 #Scheme 2: Water filling or minimising the max queue
2 for t in range(T):
3     #defining arrivals
4     for i in range(K):
5         if(arrival_bernoulli(probability_of_arrival[i])):
6             arrival_times[i].append(t)
7             queue_size[i] += 1
8     #total service capacity
9     S = service(N, K)
10    S_copy = S
11
12    #servicing
13    #at each time slot, serve the flow with the maximum queue
    size
14    while(S>0):
15        #finding the flow with the maximum queue size
16        max_queue = 0
17        for j in range(K):
18            if(queue_size[j] > max_queue):
19                max_queue = queue_size[j]
20                max_flow = j
21        #serving the flow with the maximum queue size
22        if(max_queue == 0):
23            break
24        service_times[max_flow].append(t)
25        queue_size[max_flow] -= 1
26        number_of_packets_served[max_flow][t] += 1
27        S -= 1
28
29    #server utilization
30    if(S_copy == 0):
31        server_utilization[t] = 0
32    server_utilization[t] = (S_copy - S)/S_copy
33
34    #updating average delay and probability of arrival for each
    flow
35    for i in range(K):
36        if(number_of_packets_served[i][t] > 0):
37            #total packets served so far
38            total = 0
39            for j in range(t-1):

```

```

40         total = total + number_of_packets_served[i][j]
41         for j in range(number_of_packets_served[i][t]):
42             average_delay[i] = average_delay_update(i, R,
43             average_delay[i], t - arrival_times[i][total + j], total+j)
44             probability_of_arrival[i] = arrival_general_update(
45             i, probability_of_arrival[i], total, t, average_delay[i], R
46             , arrival_times[i], service_times[i], 0.01)
47
48 #plotting moving average of server utilization
49 moving_average = [0 for i in range(T)]
50 for i in range(T):
51     total = 0
52     for j in range(20):
53         if(i-j >= 0):
54             total = total + server_utilization[i-j]
55     moving_average[i] = total/20
56 plt.plot(moving_average, 'b')
57 plt.xlabel('Time')
58 plt.ylabel('Server Utilization')
59 plt.title('Server Utilization vs Time for Water Filling')
60 plt.show()
61
62 #throughput and average delay for each flow
63 throughput = [0 for i in range(K)]
64 for i in range(K):
65     throughput[i] = len(service_times[i])/T
66
67 #plotting throughput and average delay for each flow
68 plt.plot(throughput, average_delay, 'bo')
69 plt.xlabel('Throughput')
70 plt.ylabel('Average Delay')
71 plt.title('Average Delay vs Throughput for Water Filling')
72 plt.show()
73
74 throughput_water_filling = throughput
75 average_delay_water_filling = average_delay
76 server_utilization_water_filling = server_utilization
77 moving_average_water_filling = moving_average

```

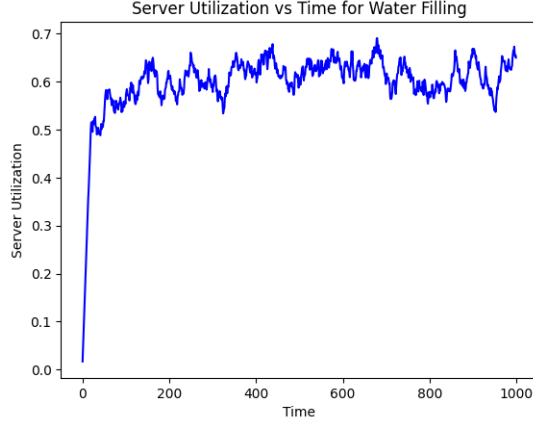



Figure 3: Water filling server utilisations

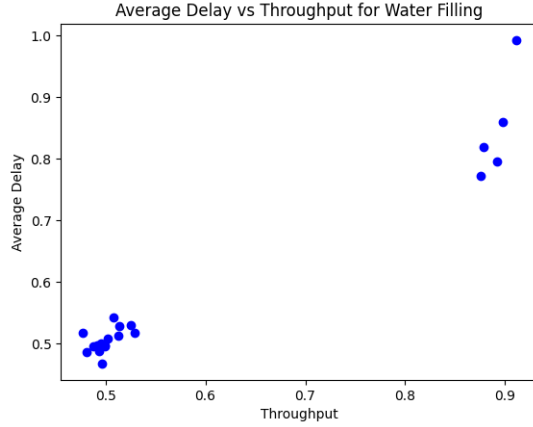


Figure 4: Average Delay vs Throughput

2.6 Scheme 3: MaxWeight

In this scheme, at every time step, we try to serve the maximum queue size first and then continue onto the smaller queues, if capacity is left. This is done in the following way, let's say at time step t , we have a service capacity S , we first find the maximum queue size and serve it entirely (or) as much as possible as per our capacity. If this queue is over, and we still have some capacity left, we serve the next maximum queue entirely (or) as much as possible and so on.

Consider $S = 11$, queue sizes: $Q = (7, 3, 2, 1)$, then after this time slot, the queue sizes would be: $Q = (0, 0, 1, 1)$.

- Step 1: Serve the max queue. $Q = (0, 3, 2, 1)$ and $S=4$.

- Step 2: Serve the max queue. $Q = (0, 0, 2, 1)$ and $S=1$.
- Step 3: Serve the max queue (as much as possible). $Q = (0, 0, 1, 1)$ and $S=0$.

This algorithm is executed in code as follows:

```

1 #Scheme 3: MaxWeight
2 for t in range(T):
3     #defining arrivals
4     for i in range(K):
5         if(arrival_bernoulli(probability_of_arrival[i])):
6             arrival_times[i].append(t)
7             queue_size[i] += 1
8     #total service capacity
9     S = service(N, K)
10    S_copy = S
11
12    #servicing
13    #at each time slot, give all S slots to the flow with the
14    #maximum weight (queue size)
15    while(S>0):
16        #finding the flow with the maximum queue size
17        max_queue = 0
18        for j in range(K):
19            if(queue_size[j] > max_queue):
20                max_queue = queue_size[j]
21                max_flow = j
22        if(max_queue == 0):
23            break
24        for i in range(max_queue):
25            if(S == 0):
26                break
27            service_times[max_flow].append(t)
28            queue_size[max_flow] -= 1
29            number_of_packets_served[max_flow][t] += 1
30            S -= 1
31
32    #server utilization
33    if(S_copy == 0):
34        server_utilization[t] = 0
35    server_utilization[t] = (S_copy - S)/S_copy
36
37    #updating average delay and probability of arrival for each
38    #flow
39    for i in range(K):
40        if(number_of_packets_served[i][t] > 0):
41            #total packets served so far
42            total = 0
43            for j in range(t-1):
44                total = total + number_of_packets_served[i][j]
45            for j in range(number_of_packets_served[i][t]):
46                average_delay[i] = average_delay_update(i, R,
47                average_delay[i], t - arrival_times[i][total + j], total+j)

```

```

45     probability_of_arrival[i] = arrival_general_update(
46         i, probability_of_arrival[i], total, t, average_delay[i], R
47         , arrival_times[i], service_times[i], 0.01)
48
49 #plotting moving average of server utilization
50 moving_average = [0 for i in range(T)]
51 for i in range(T):
52     total = 0
53     for j in range(20):
54         if(i-j >= 0):
55             total = total + server_utilization[i-j]
56     moving_average[i] = total/20
57 plt.plot(moving_average, 'g')
58 plt.xlabel('Time')
59 plt.ylabel('Server Utilization')
60 plt.title('Server Utilization vs Time for MaxWeight')
61 plt.show()
62
63 #throughput and average delay for each flow
64 throughput = [0 for i in range(K)]
65 for i in range(K):
66     throughput[i] = len(service_times[i])/T
67
68 #plotting throughput and average delay for each flow
69 plt.plot(throughput, average_delay, 'go')
70 plt.xlabel('Throughput')
71 plt.ylabel('Average Delay')
72 plt.title('Average Delay vs Throughput for MaxWeight')
73 plt.show()
74
75 throughput_max_weight = throughput
76 average_delay_max_weight = average_delay
77 server_utilization_max_weight = server_utilization
78 moving_average_max_weight = moving_average

```

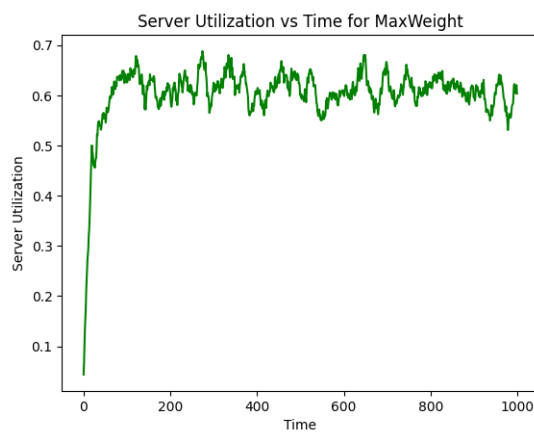


Figure 5: MaxWeight Server Utilisation

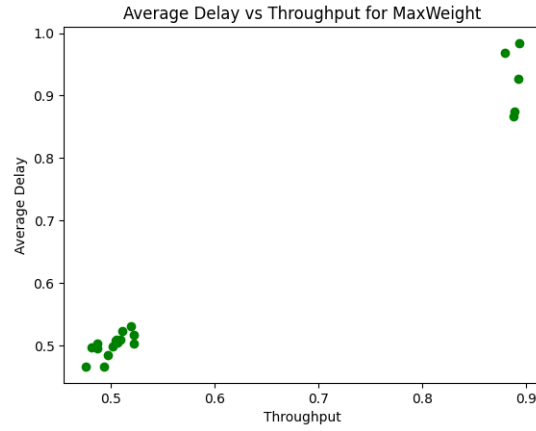


Figure 6: Average Delay vs Throughput

2.7 Comparing the three Schemes

Above, we have seen all the three schemes, the way they are executed, the algorithms and the simulation results. Now, we shall try to compare all these schemes and draw conclusions from our comparisons. Let us first plot the average delay vs throughputs for all the three schemes on the same plot.

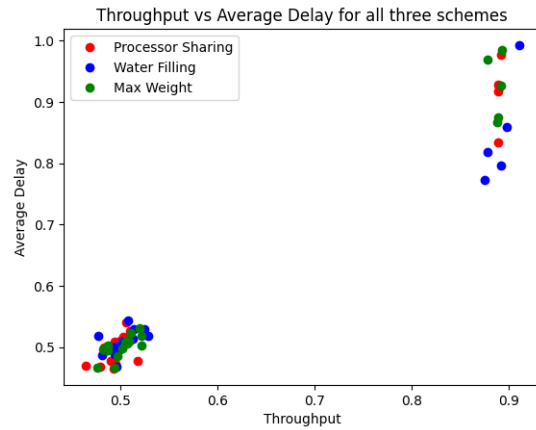


Figure 7: Average Delay vs Throughput for all three schemes

This does not give us any real distinctions between the three schemes in terms of average delay or throughput.

Let us take one particular flow, say flow 17 and try to observe the average delays and throughputs for that flow.

Consider the following codeblock:

```
1 #Comparing throughput and average delay for all three schemes
   for a particular flow
```

```

2 pipe = 17
3 print("Throughput for flow", pipe, "for Processor Sharing
    Scheme:", throughput_processor_sharing[pipe-1])
4 print("Average Delay for flow", pipe, "for Processor Sharing
    Scheme:", average_delay_processor_sharing[pipe-1])
5 print("Throughput for flow", pipe, "for Water Filling Scheme:",
    throughput_water_filling[pipe-1])
6 print("Average Delay for flow", pipe, "for Water Filling Scheme
    :", average_delay_water_filling[pipe-1])
7 print("Throughput for flow", pipe, "for Max Weight Scheme:",
    throughput_max_weight[pipe-1])
8 print("Average Delay for flow", pipe, "for Max Weight Scheme:",
    average_delay_max_weight[pipe-1])
9
10 #Plotting throughput vs average delay for all three schemes for
    a particular flow
11 plt.plot(throughput_processor_sharing[pipe-1],
    average_delay_processor_sharing[pipe-1], 'ro', label = '
    Processor Sharing')
12 plt.plot(throughput_water_filling[pipe-1],
    average_delay_water_filling[pipe-1], 'bo', label = 'Water
    Filling')
13 plt.plot(throughput_max_weight[pipe-1],
    average_delay_max_weight[pipe-1], 'go', label = 'Max Weight
    ')
14 plt.xlabel('Throughput')
15 plt.ylabel('Average Delay')
16 plt.title('Throughput vs Average Delay for flow ' + str(pipe))
17 plt.legend()
18 plt.show()

```

The output is as follows:

```

Throughput for flow 17 for Processor Sharing Scheme: 0.898
Average Delay for flow 17 for Processor Sharing Scheme: 1.07396579102213
Throughput for flow 17 for Water Filling Scheme: 0.892
Average Delay for flow 17 for Water Filling Scheme: 0.8845909328159156
Throughput for flow 17 for Max Weight Scheme: 0.895
Average Delay for flow 17 for Max Weight Scheme: 0.5812663504261814

```

Figure 8: Output

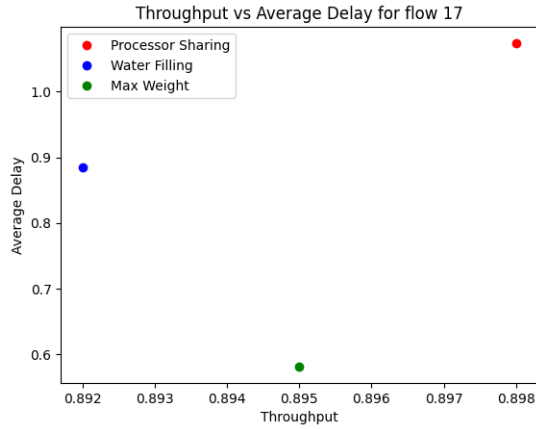


Figure 9: Throughput vs Average Delay for Flow 17

This shows us that the throughput is maximum for the Processor Sharing scheme, at the same time the average delay was minimum for the MaxWeight scheme. However, these results were not obtained constantly and hence no conclusions can be drawn between the choice of scheme and average delay and throughput.

Let us now try to compare the server utilisations for the three schemes. For this, we plot the server utilisations versus time for all the three schemes on the same plot, and also print their time averages.

This was done using the following codeblock:

```

1 #Plotting server utilizations of all three schemes on the same
  graph
2 plt.plot(moving_average_processor_sharing, 'r', label = '
  Processor Sharing')
3 plt.plot(moving_average_water_filling, 'b', label = 'Water
  Filling')
4 plt.plot(moving_average_max_weight, 'g', label = 'Max Weight')
5 plt.xlabel('Time')
6 plt.ylabel('Server Utilization')
7 plt.title('Server Utilization vs Time for all three schemes')
8 plt.legend()
9 plt.show()
10 #comparing average server utilization for all three schemes in
    percentage
11 print("Average server utilization for Processor Sharing Scheme
    in percentage:", np.mean(
    server_utilization_processor_sharing)*100)
12 print("Average server utilization for Water Filling Scheme in
    percentage:", np.mean(server_utilization_water_filling)
    *100)
13 print("Average server utilization for Max Weight Scheme in
    percentage:", np.mean(server_utilization_max_weight)*100)

```

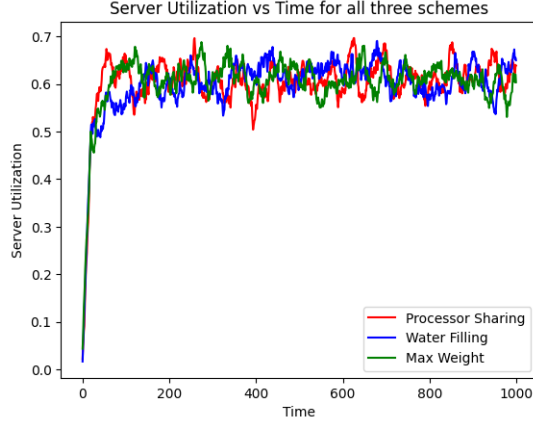


Figure 10: Output

```
Average server utilization for Processor Sharing Scheme in percentage: 60.97755607354238
Average server utilization for Water Filling Scheme in percentage: 60.57538243058945
Average server utilization for Max Weight Scheme in percentage: 60.93314124246527
```

Figure 11: Output

2.8 Server Utilisation versus arrival rate

Let us now try to see the correlation between server utilisation and arrival rate for all the three schemes. Let us first try for p very close to 0, say $p = 0.05$. Obtained results:

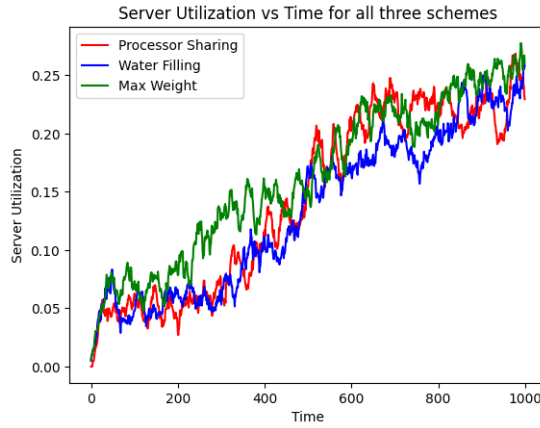


Figure 12: Server Utilisation for $p=0.05$

Clearly, for all three schemes, server utilisation seems to increase with time, this is because for all the $K - R$ flows, p is TCP like and dynamically

increases with time and thus server utilisation increases, since more packets come in with time. Average:

```
Average server utilization for Processor Sharing Scheme in percentage: 14.548646883479261
Average server utilization for Water Filling Scheme in percentage: 13.587500913177248
Average server utilization for Max Weight Scheme in percentage: 16.3797301174606
```

Figure 13: Output

Let us now simulate for a very high p value, say $p = 0.95$. Obtained results:



Figure 14: Server Utilisation for $p=0.05$

The server utilisations for all three schemes reach their highest values very quickly and thereby remain almost constant for the entire duration, since p can decay to 0.9 dynamically for the $K - R$ flows, even if it decays, there is not much change from 0.95 to 0.90 and that is why we don't see a great change in server utilisation.

```
Average server utilization for Processor Sharing Scheme in percentage: 90.95575920816692
Average server utilization for Water Filling Scheme in percentage: 94.97400942863939
Average server utilization for Max Weight Scheme in percentage: 94.38346197710797
```

Figure 15: Output

2.9 Changing (K,R)

All the results up until now were for the $(K, R) = (20, 15)$ pair, let us now try to simulate for $(K, R) = (5, 1)$ and $(K, R) = (50, 40)$ pairs, for $p = 0.5$.

2.9.1 $(\mathbf{K}, \mathbf{R}) = (5, 1)$

Taking $N = \lceil 1.1K \rceil = 6$, we get:

Average Server Utilisations:

- Processor Sharing: 81.08%
- Water Filling: 81.32%
- MaxWeight: 81.30%

Taking $N = \lceil 1.5K \rceil = 8$, we get:

Average Server Utilisations:

- Processor Sharing: 81.85%
- Water Filling: 84.83%
- MaxWeight: 82.88%

Taking $N = \lceil 3K \rceil = 15$, we get:

Average Server Utilisations:

- Processor Sharing: 77.98%
- Water Filling: 83.12%
- MaxWeight: 82.90%

2.9.2 $(\mathbf{K}, \mathbf{R}) = (50, 40)$

Taking $N = \lceil 1.1K \rceil = 55$, we get:

Average Server Utilisations:

- Processor Sharing: 58.08%
- Water Filling: 57.62%
- MaxWeight: 57.33%

Taking $N = \lceil 1.5K \rceil = 75$, we get:

Average Server Utilisations:

- Processor Sharing: 58.89%
- Water Filling: 58.23
- MaxWeight: 58.19

Taking $N = \lceil 3K \rceil = 150$, we get:

Average Server Utilisations:

- Processor Sharing: 58.73
- Water Filling: 58.22
- MaxWeight: 58.10

Here, we clearly note that if $\frac{R}{K}$ is higher, then server utilisation is low, this is because, these R flows do not dynamically update in a TCP like fashion and thus, lead to a lower average arrival rate and thus, lower server utilisations. Whereas, if $\frac{R}{K}$ is lower, then server utilisation is higher, because the $K - R$ flows dynamically update their arrival rates in order to maximise server utilisation for any scheme.

3 Finite Buffer Size

In the case of the finite buffer, we might have some packets which are "dropped", i.e any packet which is arriving to a full buffer will not be entertained, and will be dropped, thus will have to be sent again by the sender.

We simulate the case of finite buffers for all three schemes in a similar way with just some slight modifications to the code to handle the buffer.

The function `arrival_general_update` is modified as following to take into account dropped packets. This is because though a dropped packet will not contribute to the update of $T_n^{(i)}$, but it will contribute to the number of un-served packets that arrived on or before $t - \left\lceil 1.2T_{n(t)}^{(i)} \right\rceil$.

```

1 def arrival_general_update(i, p, n, t, T, R, arrival_times,
2   service_times, delta, dropped_times):
3     #This is the update equation for the probability of arrival
4     #p is the previous probability of arrival
5     #n is the number of packets served so far for this flow
6     #T is the average delay for this flow
7     #arrival_times is the list of arrival times for this flow
8     #service_times is the list of service times for this flow
9     #This function returns the new probability of arrival
10    if n == 0:
11        return p
12    elif i < R:
13        return p
14    elif len(arrival_times) > len(service_times):
15        if(arrival_times[len(service_times)] <= t - np.ceil
16           (1.2*T)):
17            return 2*p/3
18    elif len(dropped_times) > 0:
19        if dropped_times[0] <= t - np.ceil(1.2*T):
20            #remove first element of dropped_times
21            dropped_times= dropped_times.pop(0) #count one
22            dropped packet only once
23            return 2*p/3

```

```

21     #for the 2nd case,
22     #the first packet served in this time slot should have
    arrived
23     #at most 1.2*T time slots ago
24     else:
25         for i in range(len(service_times)):
26             if service_times[i] == t:
27                 if arrival_times[i] > t - np.ceil(1.2*T): #
checking if the served packet arrived in the last 1.2*T
time slots
28                     return min(0.9, p+delta)
29                 else:
30                     break
31     return p

```

The algorithms remain the same for all three schemes, which just a slight change in which input arrivals are handled. The change is:

```

1 #defining arrivals
2     for i in range(K):
3         if(arrival_bernoulli(probability_of_arrival[i])):
4             if(queue_size[i] < q_max):
5                 queue_size[i] += 1 #increment queue size if
buffer is not full
6                 arrival_times[i].append(t)
7             else:
8                 dropped_packets[i] += 1 #increment dropped
packets if buffer is full
9                 dropped_times[i].append(t)

```

This makes sure that only those packets which can still fit in the buffer are counted in `arrival_times` and those which are dropped are counted in `dropped_times` and increment `dropped_packets`.

We simulate for buffer sizes $\alpha \frac{Kp^2}{1-p}$ for $\alpha = 1.2, 1.5, 3$, for all the three schemes and try to observe the results.

Running the code for $\alpha = 1.2$ and observing throughput and delay for flow 17, we get:

```

Throughput for flow 17 for Processor Sharing Scheme: 0.886
Average Delay for flow 17 for Processor Sharing Scheme: 0.9743363097780845
Throughput for flow 17 for Water Filling Scheme: 0.875
Average Delay for flow 17 for Water Filling Scheme: 0.8288192515785472
Throughput for flow 17 for Max Weight Scheme: 0.881
Average Delay for flow 17 for Max Weight Scheme: 0.7497302370668733

```

Figure 16: Output

Not a very significant difference as compared to the infinite buffer case. Let us try to observe the server utilisations, we get:

```

Average server utilization for Processor Sharing Scheme in percentage: 60.70842387540095
Average server utilization for Water Filling Scheme in percentage: 59.944010266197914
Average server utilization for Max Weight Scheme in percentage: 60.88023901462351

```

Figure 17: Output

Yet again, these values do not sway too much as compared to the values for the infinite buffer case. Anyways, let's try to observe how many packets are dropped for each flow, maybe that can give us a hint about what is happening.

Printing the number of dropped packets for each flow, we get:

```

Number of packets dropped for each flow for Processor Sharing Scheme: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
Number of packets dropped for each flow for Water Filling Scheme: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
Number of packets dropped for each flow for Max Weight Scheme: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

```

Figure 18: Output

We get only 0's as the number of packets dropped for any flow, so something interesting is happening here, no packets are being dropped i.e the finite buffer case is as good as the infinite buffer case for $\alpha = 1.2$, if the buffer size corresponding to $\alpha = 1.2$ is able to handle all packets, then obviously $\alpha = 1.5, 3$ can handle all packets

An intuitive reason of why this happens is the following, the server has a $\text{Binomial}(N, K/N)$ distribution, which means that the expected capacity at any time slot is $N * K/N = K$, since the expected number of arriving packets is only $p_{avg} * K$, the server is efficiently able to handle all these arrivals and thus the queue sizes never get even remotely big, which we had seen in the infinite buffer case, all our average queue sizes were always less than 1.

Thinking on parallel lines, what if we increase p to 0.99, i.e as close to 1 as possible, on simulating this case, we observe that the number of packets dropped for this case is also 0 for all flows, thus indicating that increasing p did not cause packets to drop for a fixed α .

Similarly, decreasing K did not cause packets to drop either, and thus we can conclude that among the three $\alpha = 1.2, 1.5, 3$, $\alpha = 1.2$ is sufficient enough to handle this system.

4 Conclusion

We have simulated the given caricature TCP model and have studied three scheduling algorithms namely processor sharing, water filling and MaxWeight scheduling, we noticed that for small arrival rates the server utilisation tends

to increase with time as the arrival rates dynamically increase in a TCP like fashion, whereas for higher arrival rates, server utilisation is more or less constant.

An interesting point to note was that server utilisation was higher for lower values of $\frac{R}{K}$, this is because more number of flows were dynamic since $K - R$ was larger. Also, server utilisation is higher for higher arrival rates. It would be an interesting topic of research to study how the interplay of static and dynamic flows, along with arrival rates affects the server utilisation, while at the same time ensuring that queue lengths do not get large enough that a small finite buffer is unable to handle it, and packets start dropping. The interplay of all these factors with each other should be an interesting direction to research in.

Another interesting problem could be with respect to buffer sizes, in the simulations we noticed that we were able to get 0 packets dropped for the given buffer sizes, an interesting problem could be to study about how small our buffer could be so that even after dropping a few packets, we're able to maximise server utilisation and throughput.

The code can be found here: [Code](#)