EE5150: Communication Networks
Programming Assignment Report

Ishaan Agarwal
EE20B046

May 15, 2023

1 Introduction

The aim of these problems is to simulate some fairly known concepts taught in the course and make observations in concurrence with theory taught in the lectures.

2 The Geo/Geo/1 Queue

We aim to simulate a discrete time Geo/Geo/1 queue i.e a discrete time queue with Bernoulli arrivals and services. We start by defining the functions for arrivals and services in the following way:

```
def arrival(lamda):
    #This is a Bernoulli random variable with success
    probability lambda
    return np.random.choice([0,1], p=[1-lamda, lamda])

def service(mu):
    #This is a Bernoulli random variable with success
    probability mu
    return np.random.choice([0,1], p=[1-mu, mu])
```

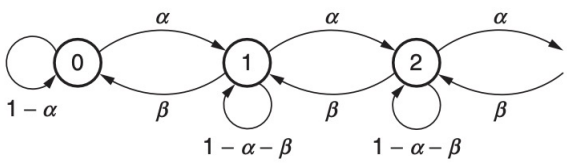

Figure 1: The Discrete time Geo/Geo/1 Queue

2.1 Simulating the Geo/Geo/1 Queue

We simulate the queue for $T = 10000$ time slots for $\lambda = 0.1, 0.2 \ldots 1$ for a constant $\mu = 0.9$. This is done in the following code block:

```
mu = 0.9
T = 10000 #total time
#queue length = number of packets in the queue at any time
#sojourn time = time spent in queue + time spent in service
lamda_range = np.arange(0.1, 1.1, 0.1)
average_q = []
average_sojourn_time = []

for lamda in lamda_range:
    ql = 0 #queue length
    q = [] #queue length at each time instant just before the arrival
    q.append(0)
    arrival_times = []
    service_times = []
    sojourn_times = []
    t = 1

    while t < T:
        if arrival(lamda):
            ql += 1
            arrival_times.append(t)
        if service(mu):
            if(ql!=0):
                ql-=1
                service_times.append(t)
        q.append(ql)
        t += 1

    average_q.append(np.mean(q))
    for i in range(len(service_times)): #computing only for those who have been served
        sojourn_times.append(service_times[i] - arrival_times[i])
    average_sojourn_time.append(np.mean(sojourn_times))
```

2.2 Average Queue Lengths

Now, we attempt to observe the average queue lengths versus the arrival rate λ. The corresponding code is:

```

print("Average values of queue length vs lamda:")
for i in range(1, len(average_q)+1):
    print("lamda = ", i/10, ":", average_q[i-1])

```

```
plt.plot(lamda_range, average_q, 'ro-')
plt.xlabel('lamda')
plt.ylabel('average queue length')
plt.grid()
plt.show()
```

The obtained output is as follows:

```
Average values of queue length vs lamda:
lamda =  0.1 : 0.0132
lamda =  0.2 : 0.0267
lamda =  0.3 : 0.0484
lamda =  0.4 : 0.0837
lamda =  0.5 : 0.1307
lamda =  0.6 : 0.2203
lamda =  0.7 : 0.3396
lamda =  0.8 : 0.824
lamda =  0.9 : 18.8372
lamda =  1.0 : 524.8979
```

Figure 2: Output

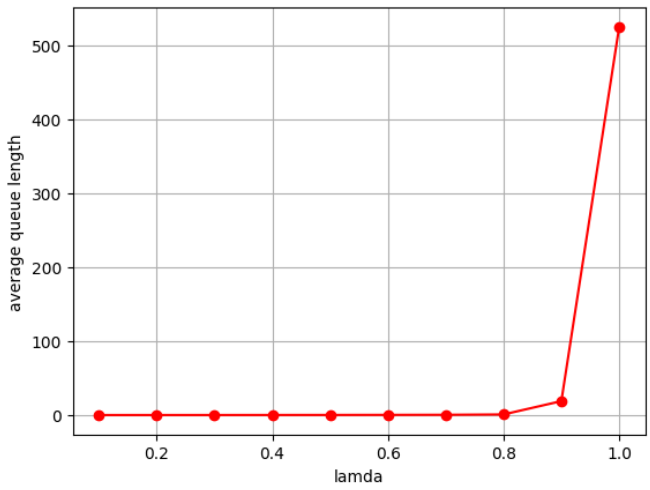

Figure 3: Average queue lengths vs λ

We see that the average values of queue lengths are very small for $\lambda < 0.9$, this is because in these cases the arrivals are slower than services and thus, the queue length does not blow up, whereas in the other two cases, $\lambda \geq \mu$, and thus the queue length is significantly high.

2.3 Simulations vs Theoretical

In the previous part, we have observed the simulated queue length vs λ. In this part, let us compare that with the theoretical average queue lengths, which is computed by taking the expectation under a stationary distribution for the same set of parameters.
We know that for a finite state Geo/Geo/1 queue, the stationary distribution is given by the following (for max n customers in the system)(since infinite is not realisable on a machine, we take very large n value): Local balance equations:

$$\pi(i)\lambda(1-\mu) = \pi(i+1)\mu(1-\lambda) \;\; \forall i = 0, 1, 2, \ldots, n-1$$

$$\implies \pi(i+1) = \pi(i)\rho \;\; \forall i = 0, 1, 2, ..., n-1$$

$$\implies Pi(i) = \frac{(1-p)\rho^i}{(1-p^{(n+1)})}) \;\; \forall i = 0, 1, 2, ..., n$$

$$\rho = \frac{\lambda(1-\mu)}{\mu(1-\lambda)}$$

The codeblock for the following part is:

```
#part (b):
n = 1000 #max queue length
mu = 0.9

theoretical_average_q = []

for lamda in lamda_range:
    p = lamda*(1-mu)/(mu*(1-lamda))
    Pi = np.zeros(n+1)
    #set Pi[0] = (1-p)(1-p**(n+1))
    Pi[0] = (1-p)/(1-p**(n+1))
    for i in range(1, n+1):
        Pi[i] = p**i*Pi[0]
    #Finding the expectation under the steady state
    distribution
    E = 0
    for i in range(n+1):
        E += i*Pi[i]
    theoretical_average_q.append(E)

print("Theoretical average queue length vs lamda:")
for i in range(1, len(theoretical_average_q)+1):
    print("lamda = ", i/10, ":", theoretical_average_q[i-1])

print("Simulated average queue length vs lamda:")
for i in range(1, len(average_q)+1):
    print("lamda = ", i/10, ":", average_q[i-1])

```

```
#plot theoretical average and simulated average on the same
    graph
plt.plot(lamda_range, theoretical_average_q, 'ro-', label='
    theoretical')
plt.plot(lamda_range, average_q, 'b--', label='simulated')
plt.xlabel('lamda')
plt.ylabel('average queue length')
plt.legend()
plt.grid()
plt.show()
```

The obtained output is:

```
Theoretical average queue length vs lamda:
lamda =  0.1 : 0.012499999999999997
lamda =  0.2 : 0.028571428571428564
lamda =  0.3 : 0.05
lamda =  0.4 : 0.07999999999999997
lamda =  0.5 : 0.12499999999999996
lamda =  0.6 : 0.1999999999999999
lamda =  0.7 : 0.3499999999999998
lamda =  0.8 : 0.8000000000000002
lamda =  0.9 : nan
lamda =  1.0 : nan
Simulated average queue length vs lamda:
lamda =  0.1 : 0.0132
lamda =  0.2 : 0.0267
lamda =  0.3 : 0.0484
lamda =  0.4 : 0.0837
lamda =  0.5 : 0.1307
lamda =  0.6 : 0.2203
lamda =  0.7 : 0.3396
lamda =  0.8 : 0.824
lamda =  0.9 : 18.8372
lamda =  1.0 : 524.8979
```

Figure 4: Output

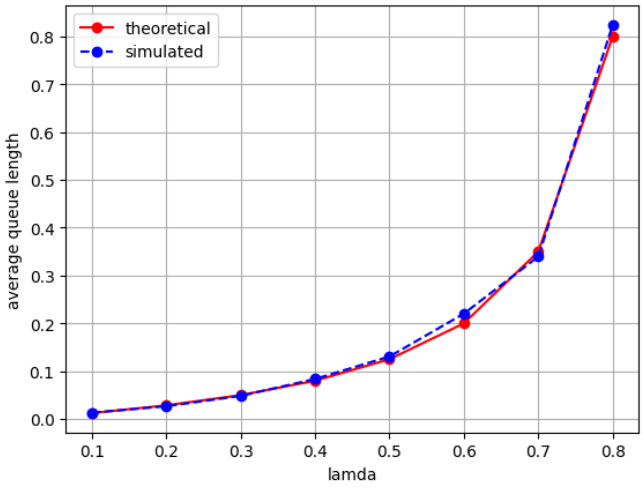

Figure 5: Theoretical average queue lengths and Simulated Average Queue lengths vs λ

Note that the theoretical averages are not defined for $\lambda \geq 0.9$ because the sum of the geometric series only converges if $\lambda < \mu$, which is also our necessary condition for positive recurrence of the discrete time Markov chain corresponding to this process.

2.4 Sojourn times

Sojourn time is defined as the time from arrival till service completion of a packet, we want to observe the average sojourn times for each λ. This was done in the code block provided above by subtracting the arrival time from the service time for each packet and then taking the average.

```
print("Average sojourn time vs lamda:")
for i in range(1, len(average_sojourn_time)+1):
    print("lamda = ", i/10, ":", average_sojourn_time[i-1])
plt.plot(lamda_range, average_sojourn_time, 'ro-')
plt.xlabel('lamda')
plt.ylabel('average sojourn time')
plt.grid()
plt.show()
```

The obtained output is as follows:

```
Average sojourn time vs lamda:
lamda =  0.1 : 0.13510747185261002
lamda =  0.2 : 0.13303437967115098
lamda =  0.3 : 0.16340310600945307
lamda =  0.4 : 0.20987963891675024
lamda =  0.5 : 0.2595313741064337
lamda =  0.6 : 0.3596147567744042
lamda =  0.7 : 0.481838819523269
lamda =  0.8 : 1.031031031031031
lamda =  0.9 : 20.87391304347826
lamda =  1.0 : 525.4670164080812
```

Figure 6: Output

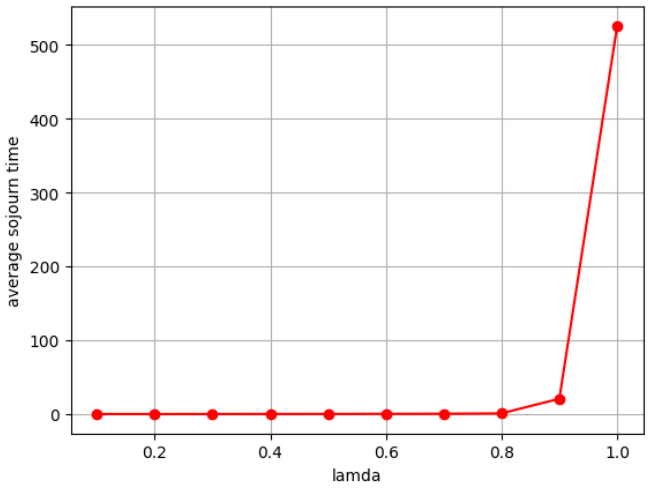

Figure 7: Average sojourn times vs λ

2.5 Little's Law

Little's Law states that the average queue length is equal to the product of the mean arrival rate and the mean sojourn times. Let us verify this law from our simulations.

```
#Part c: Little's Theorem
#plotting the ration of average queue length and average
    sojourn time vs lamda
ratio = np.divide(average_q, average_sojourn_time)
print("Ratio of average queue length and average sojourn time
    vs lamda:")
```

```
for i in range(1, len(ratio)+1):
    print("lamda = ", i/10, ":", ratio[i-1])
plt.plot(lamda_range, ratio, 'ro--')
plt.xlabel('lamda')
plt.ylabel('ratio of average queue length and average sojourn
    time')
plt.grid()
plt.show()
```

Now, to verify that this ratio is indeed equal to the mean arrival rate, let us plot this ratio against λ, and then fit a least squares line to this using `np.linalg.lstsq` function.

```
#Let us verify this by fitting a line to the plot
#We want to fit a least squares line to the data
#Fit a line y = mx + c to the data using numpy.linalg.lstsq
m, c = np.linalg.lstsq(np.vstack([lamda_range, np.ones(len(
    lamda_range))]).T, ratio, rcond = None)[0]
print("m = ", m, "c = ", c)
plt.plot(lamda_range, ratio, 'ro-', label='Original data',
    markersize=10)
plt.plot(lamda_range, m*lamda_range + c, 'b--', label='Fitted
    line')
plt.legend()
plt.grid()
plt.show()
#calculating mean squared error in percentage
mse = np.mean(np.square(np.subtract(ratio, m*lamda_range + c)))
print("Mean squared error in percentage = ", mse*100)
```

The graph obtained is as follows, with a slope of $m = 1.00394$ and an intercept of $c = -0.00067$, with a mean squared error of 0.001% which shows that the obtained line is very close to the line $y = x$.

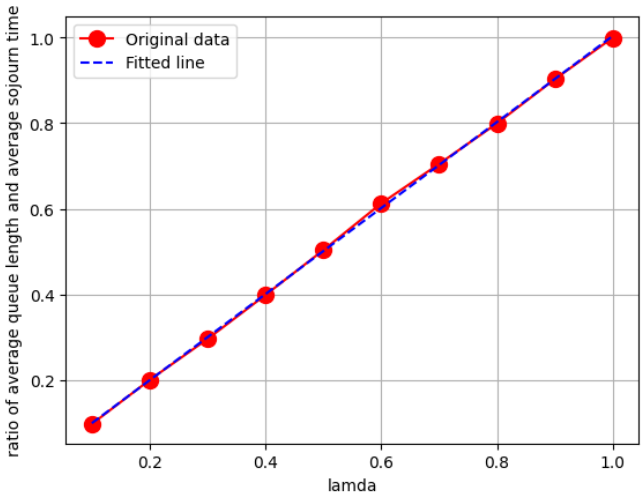

Figure 8: Little's Law

3 Conclusion

We thus have simulated the discrete time Geo/Geo/1 queue and have verified the following results.

- Average queue lengths increase with arrival rate.
- Average sojourn time increase with arrival rate.
- The simulated average queue length was found to be very close to the theoretical average queue length.
- Little's law was simulated and verified.

The code can be found here: Code

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:

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

The service capacity is $Binomial(N, K/N)$, 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 $Bernoulli(p)$, 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:

```
def average_delay_update(i, R, Tn, Dnplusone, n):
    #This is the update equation for the average delay
    #Tn is the previous average delay
    #Dnplusone is the new delay
    if(Tn == 0):
        return Dnplusone
    if(i<R):
        return (Tn*n+Dnplusone)/(n+1)
    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 $t - \left\lceil 1.2T_{n(t)}^{(i)} \right\rceil$, update $p^{(i)}(t+1) = \frac{2p^{(i)}(t)}{3}$. If at time t, r packets are served and all of them arrived after $t - \left\lceil 1.2T_{n(t)}^{(i)} \right\rceil$, then update $p^{(i)}(t+1) = \min\left(0.9, p^{(i)}(t) + \delta\right)$ 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:

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

```
        return p
    elif len(arrival_times) > len(service_times):
        if(arrival_times[len(service_times)] <= t - np.ceil
    (1.2*T)):
            return 2*p/3
    #for the 2nd case,
    #the first packet served in this time slot should have
    arrived
    #at most 1.2*T time slots ago
    else:
        for i in range(len(service_times)):
            if service_times[i] == t:
                if arrival_times[i] > t - np.ceil(1.2*T):
                    return min(0.9, p+delta)
                else:
                    break
    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.

```
N=30
K=20
R=15
T = 1000
p=0.5
#One dimensional array for the number of packets in the queue
queue_size = [0 for i in range(K)]
#Two dimensional arrays for the arrival times and service times
arrival_times = [[] for i in range(K)]
service_times = [[] for i in range(K)]
#One dimensional arrays for the current average delay and
    current probability of arrival
average_delay = [0 for i in range(K)]
#probability_of_arrival is p for each flow in the beginning
probability_of_arrival = [p for i in range(K)]
#Two dimensional arrays for the number of packets served
number_of_packets_served = [[0 for i in range(T)] for j in
    range(K)]
#One dimensional array for server utilization
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:

```
#Scheme 1: Processor Sharing
for t in range(T):
    #defining arrivals
    for i in range(K):
        if(arrival_bernoulli(probability_of_arrival[i])):
            arrival_times[i].append(t)
            queue_size[i] += 1
    #total service capacity
    S = service(N, K)
    S_copy = S
    #finding 'l' value as per the scheme:
    for l in range(0,100):
        if(S > l*K and S <= (l+1)*K):
            break

    #servicing
    #serve l packets from each flow first, then serve one
    additional packet
    #from each flow in a round robin fashion until S packets
    are served
    for i in range(K):
        for j in range(l):
            if(queue_size[i] > 0):
                service_times[i].append(t)
                queue_size[i] -= 1
                number_of_packets_served[i][t] += 1
                S -= 1
    #round robin
    i=0
    while(S > 0):
        if(queue_size[i] > 0):
            service_times[i].append(t)
            queue_size[i] -= 1
            number_of_packets_served[i][t] += 1
            S -= 1
        i = (i+1)%K
```

```
        if(i == 0):
            break
    #server utilization
    if(S_copy == 0):
        server_utilization[t] = 0
    server_utilization[t] = (S_copy - S)/S_copy

    #updating average delay and probability of arrival for each flow
    for i in range(K):
        if(number_of_packets_served[i][t] > 0):
            #total packets served so far
            total = 0
            for j in range(t-1):
                total = total + number_of_packets_served[i][j]
            for j in range(number_of_packets_served[i][t]):
                average_delay[i] = average_delay_update(i, R, average_delay[i], t - arrival_times[i][total + j], total+j)
            probability_of_arrival[i] = arrival_general_update(i, probability_of_arrival[i], total, t, average_delay[i], R, arrival_times[i], service_times[i], 0.01)

#plotting moving average of server utilization
moving_average = [0 for i in range(T)]
for i in range(T):
    total = 0
    for j in range(20):
        if(i-j >= 0):
            total = total + server_utilization[i-j]
    moving_average[i] = total/20
plt.plot(moving_average, 'r')
plt.xlabel('Time')
plt.ylabel('Server Utilization')
plt.title('Server Utilization vs Time for Processor Sharing')
plt.show()

#throughput and average delay for each flow
throughput = [0 for i in range(K)]
for i in range(K):
    throughput[i] = len(service_times[i])/T

throughput_processor_sharing = throughput
average_delay_processor_sharing = average_delay
server_utilization_processor_sharing = server_utilization
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:

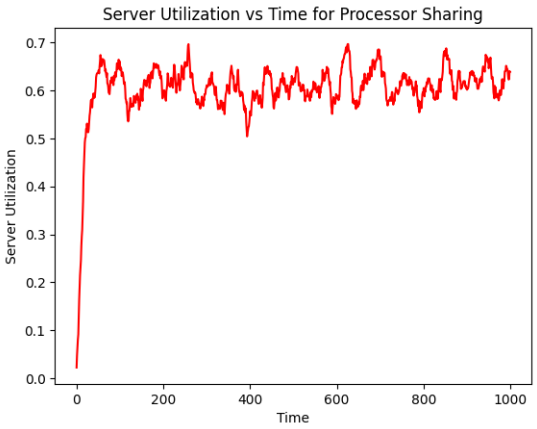

Figure 1: Processor sharing server utilisation

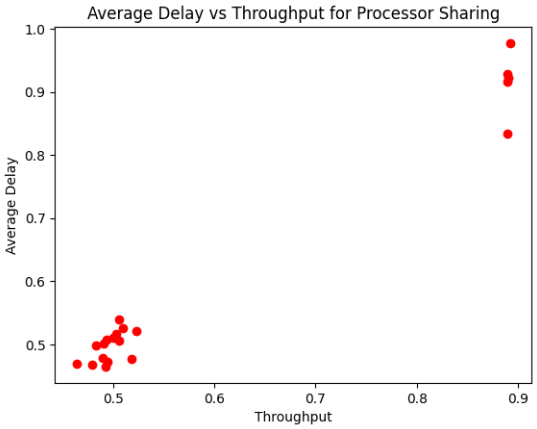

Figure 2: Average Delay vs Throughput

2.5 Scheme 2: Water Filling or minimise the max queue

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

```
#Scheme 2: Water filling or minimising the max queue
for t in range(T):
    #defining arrivals
    for i in range(K):
        if(arrival_bernoulli(probability_of_arrival[i])):
            arrival_times[i].append(t)
            queue_size[i] += 1
    #total service capacity
    S = service(N, K)
    S_copy = S

    #servicing
    #at each time slot, serve the flow with the maximum queue
    size
    while(S>0):
        #finding the flow with the maximum queue size
        max_queue = 0
        for j in range(K):
            if(queue_size[j] > max_queue):
                max_queue = queue_size[j]
                max_flow = j
        #serving the flow with the maximum queue size
        if(max_queue == 0):
            break
        service_times[max_flow].append(t)
        queue_size[max_flow] -= 1
        number_of_packets_served[max_flow][t] += 1
        S -= 1

    #server utilization
    if(S_copy == 0):
        server_utilization[t] = 0
    server_utilization[t] = (S_copy - S)/S_copy

    #updating average delay and probability of arrival for each
     flow
    for i in range(K):
        if(number_of_packets_served[i][t] > 0):
            #total packets served so far
            total = 0
            for j in range(t-1):
```

```
                total = total + number_of_packets_served[i][j]
            for j in range(number_of_packets_served[i][t]):
                average_delay[i] = average_delay_update(i, R,
    average_delay[i], t - arrival_times[i][total + j], total+j)
            probability_of_arrival[i] = arrival_general_update(
    i, probability_of_arrival[i], total, t, average_delay[i], R
    , arrival_times[i], service_times[i], 0.01)

#plotting moving average of server utilization
moving_average = [0 for i in range(T)]
for i in range(T):
    total = 0
    for j in range(20):
        if(i-j >= 0):
            total = total + server_utilization[i-j]
    moving_average[i] = total/20
plt.plot(moving_average, 'b')
plt.xlabel('Time')
plt.ylabel('Server Utilization')
plt.title('Server Utilization vs Time for Water Filling')
plt.show()

#throughput and average delay for each flow
throughput = [0 for i in range(K)]
for i in range(K):
    throughput[i] = len(service_times[i])/T

#plotting throughput and average delay for each flow
plt.plot(throughput, average_delay, 'bo')
plt.xlabel('Throughput')
plt.ylabel('Average Delay')
plt.title('Average Delay vs Throughput for Water Filling')
plt.show()

throughput_water_filling = throughput
average_delay_water_filling = average_delay
server_utilization_water_filling = server_utilization
moving_average_water_filling = moving_average
```

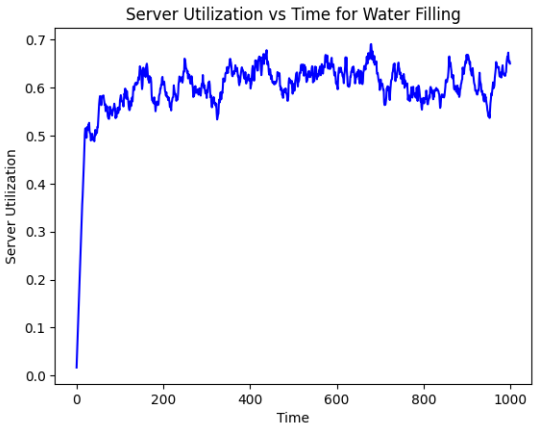

Figure 3: Water filling server utilisations

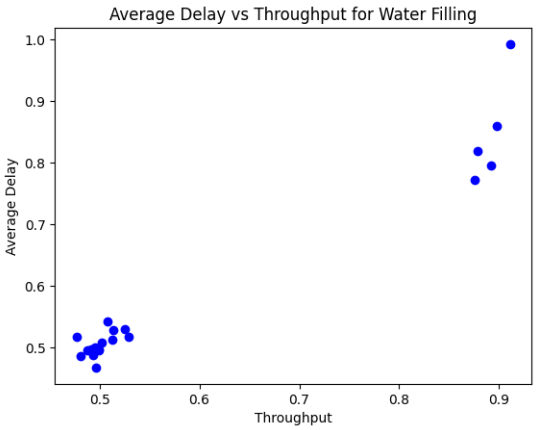

Figure 4: Average Delay vs Throughput

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:

```
#Scheme 3: MaxWeight
for t in range(T):
    #defining arrivals
    for i in range(K):
        if(arrival_bernoulli(probability_of_arrival[i])):
            arrival_times[i].append(t)
            queue_size[i] += 1
    #total service capacity
    S = service(N, K)
    S_copy = S

    #servicing
    #at each time slot, give all S slots to the flow with the
    maximum weight (queue size)
    while(S>0):
        #finding the flow with the maximum queue size
        max_queue = 0
        for j in range(K):
            if(queue_size[j] > max_queue):
                max_queue = queue_size[j]
                max_flow = j
        if(max_queue == 0):
            break
        for i in range(max_queue):
            if(S == 0):
                break
            service_times[max_flow].append(t)
            queue_size[max_flow] -= 1
            number_of_packets_served[max_flow][t] += 1
            S -= 1

    #server utilization
    if(S_copy == 0):
        server_utilization[t] = 0
    server_utilization[t] = (S_copy - S)/S_copy

    #updating average delay and probability of arrival for each
     flow
    for i in range(K):
        if(number_of_packets_served[i][t] > 0):
            #total packets served so far
            total = 0
            for j in range(t-1):
                total = total + number_of_packets_served[i][j]
            for j in range(number_of_packets_served[i][t]):
                average_delay[i] = average_delay_update(i, R,
average_delay[i], t - arrival_times[i][total + j], total+j)
```

```
            probability_of_arrival[i] = arrival_general_update(
    i, probability_of_arrival[i], total, t, average_delay[i], R
    , arrival_times[i], service_times[i], 0.01)

#plotting moving average of server utilization
moving_average = [0 for i in range(T)]
for i in range(T):
    total = 0
    for j in range(20):
        if(i-j >= 0):
            total = total + server_utilization[i-j]
    moving_average[i] = total/20
plt.plot(moving_average, 'g')
plt.xlabel('Time')
plt.ylabel('Server Utilization')
plt.title('Server Utilization vs Time for MaxWeight')
plt.show()

#throughput and average delay for each flow
throughput = [0 for i in range(K)]
for i in range(K):
    throughput[i] = len(service_times[i])/T

#plotting throughput and average delay for each flow
plt.plot(throughput, average_delay, 'go')
plt.xlabel('Throughput')
plt.ylabel('Average Delay')
plt.title('Average Delay vs Throughput for MaxWeight')
plt.show()

throughput_max_weight = throughput
average_delay_max_weight = average_delay
server_utilization_max_weight = server_utilization
moving_average_max_weight = moving_average
```

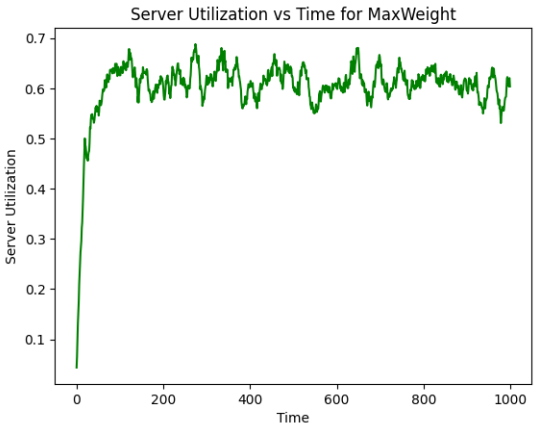

Figure 5: MaxWeight Server Utilisation

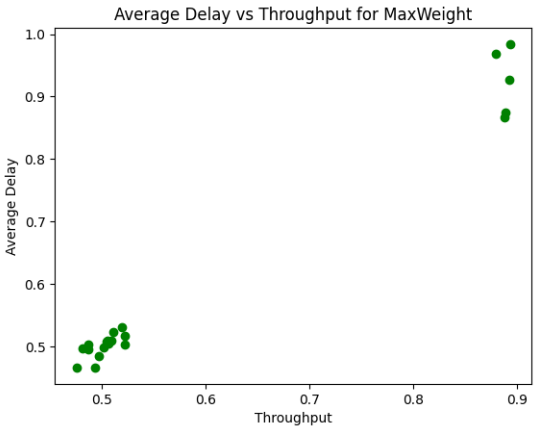

Figure 6: Average Delay vs Throughput

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.

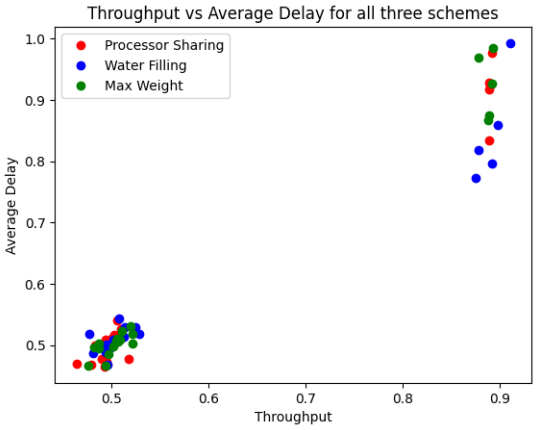

Figure 7: Average Delay vs Throughput for all three schemes

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:

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

```
pipe = 17
print("Throughput for flow", pipe, "for Processor Sharing Scheme:", throughput_processor_sharing[pipe-1])
print("Average Delay for flow", pipe, "for Processor Sharing Scheme:", average_delay_processor_sharing[pipe-1])
print("Throughput for flow", pipe, "for Water Filling Scheme:", throughput_water_filling[pipe-1])
print("Average Delay for flow", pipe, "for Water Filling Scheme:", average_delay_water_filling[pipe-1])
print("Throughput for flow", pipe, "for Max Weight Scheme:", throughput_max_weight[pipe-1])
print("Average Delay for flow", pipe, "for Max Weight Scheme:", average_delay_max_weight[pipe-1])

#Plotting throughput vs average delay for all three schemes for a particular flow
plt.plot(throughput_processor_sharing[pipe-1], average_delay_processor_sharing[pipe-1], 'ro', label = 'Processor Sharing')
plt.plot(throughput_water_filling[pipe-1], average_delay_water_filling[pipe-1], 'bo', label = 'Water Filling')
plt.plot(throughput_max_weight[pipe-1], average_delay_max_weight[pipe-1], 'go', label = 'Max Weight')
plt.xlabel('Throughput')
plt.ylabel('Average Delay')
plt.title('Throughput vs Average Delay for flow ' + str(pipe))
plt.legend()
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

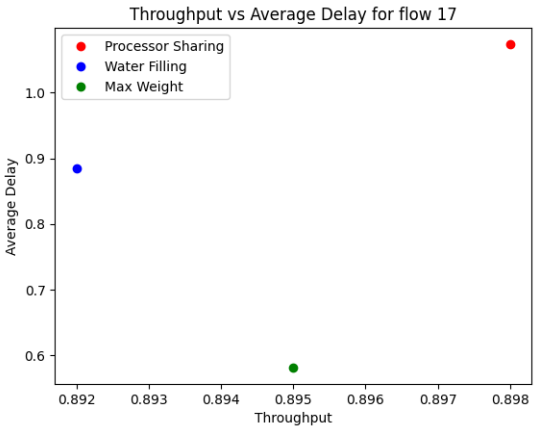

Figure 9: Throughput vs Average Delay for Flow 17

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:

```
#Plotting server utilizations of all three schemes on the same graph
plt.plot(moving_average_processor_sharing, 'r', label = 'Processor Sharing')
plt.plot(moving_average_water_filling, 'b', label = 'Water Filling')
plt.plot(moving_average_max_weight, 'g', label = 'Max Weight')
plt.xlabel('Time')
plt.ylabel('Server Utilization')
plt.title('Server Utilization vs Time for all three schemes')
plt.legend()
plt.show()
#comparing average server utilization for all three schemes in percentage
print("Average server utilization for Processor Sharing Scheme in percentage:", np.mean(server_utilization_processor_sharing)*100)
print("Average server utilization for Water Filling Scheme in percentage:", np.mean(server_utilization_water_filling)*100)
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:

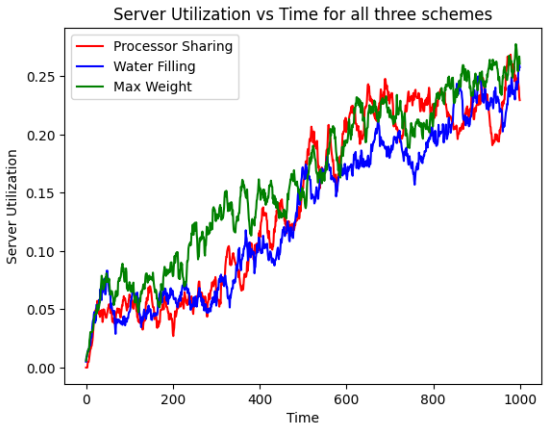

Figure 12: Server Utilisation for p=0.05

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 (K,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 (K,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 a 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 $t - \left\lceil 1.2T_{n(t)}^{(i)} \right\rceil$.

```
def arrival_general_update(i, p, n, t, T, R, arrival_times,
    service_times, delta, dropped_times):
    #This is the update equation for the probability of arrival
    #p is the previous probability of arrival
    #n is the number of packets served so far for this flow
    #T is the average delay for this flow
    #arrival_times is the list of arrival times for this flow
    #service_times is the list of service times for this flow
    #This function returns the new probability of arrival
    if n == 0:
        return p
    elif i<R:
        return p
    elif len(arrival_times) > len(service_times):
        if(arrival_times[len(service_times)] <= t - np.ceil
    (1.2*T)):
            return 2*p/3
    elif len(dropped_times) > 0:
        if dropped_times[0] <= t - np.ceil(1.2*T):
            #remove first element of dropped_times
            dropped_times= dropped_times.pop(0) #count one
    dropped packet only once
            return 2*p/3
```

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

```
#defining arrivals
    for i in range(K):
        if(arrival_bernoulli(probability_of_arrival[i])):
            if(queue_size[i] < q_max):
                queue_size[i] += 1 #increment queue size if
    buffer is not full
                arrival_times[i].append(t)
            else:
                dropped_packets[i] += 1 #increment dropped
    packets if buffer is full
                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,
Number of packets dropped for each flow for Water Filling Scheme: [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 $Binomial(N, K/N)$ 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