ECE358: Computer Networks

Winter 2018

Project 1: Queue Simulation

Date of submission:

Submitted by: YuCheng Liu

Student ID: y698liu

Student name : Liu, YuCheng

Waterloo Email address: y698liu@uwaterloo.ca

Marks received:

Marked by:

Table of Contents

Distribution Simulation **3**

Question 1: How would you generate an exponential random variable with parameter from U (0,1)? **3**

M/M/1 Queue **4**

Question 2: Build your simulator for this queue and explain in words what you have done. **4**

Step 1(Generating the list of observers): 4

Step 2(Generating the list of packets): 5

Step 3 (Creating event list for the DES): 6

Step 4 (Initialize variables and dequeue the event list): 6

Question 3: Assume L=12000 bits, C=1 Mbits/second and give the following figures using the simulator you have programmed. **7**

1. E[N], the average number of packets in the system as a function of (for 0.25< 0.95, step size 0.1). 7

2. PIDLE, the proportion of time the system is idle as a function of ρ, (for 0.25 < 0.95, step size 0.1). 8

Question 4: For the same parameters, simulate for ρ = 1.2. What do you observe? **10**

M/M/1/K Queue **11**

Question 5: Build a simulator for an M/M/1/K queue. **11**

Change 1 (Generating a packet list without departure time): 11

Change 2(Calculating the departure time during the simulation): 12

Change 3(Optimization of event handler speed): 13

Question 6: Let L=12000 bits and C=1 Mbits/second. Use your simulator to obtain the following figures **14**

1. E[N] as a function of ρ (for 0.5 < ρ < 1.5, step size 0.1) for K=5, 10, 40 packets. 14

2. PLOSS as a function of (for 0.4 < ρ <10) for K=5, 10, 40 packets. (One curve per value of K on the same figure). 16

Distribution Simulation

Question 1: How would you generate an exponential random variable with parameter from U (0,1)?

In order to create a function that can generate exponential random variables with parameter from U(0,1), I derived an equation that takes in the rate parameter, lambda in our case, and returns the result.

Equation: $f^{-1}(x) = -\frac{\ln(1-x)}{\lambda}, x = U\ (0,1)$

Calculation:

$f(x) = 1 - e^{-\lambda * x}$

$e^{-\lambda * x} = 1 - f(x)$

$-\lambda * x = \ln(1 - f(x)$

$x = \ln(1 - f(x))\ /\lambda$

Thus, the inverse of f(x) is $f^{-1}(x) = -\frac{\ln(1-x)}{\lambda}, x = U\ (0,1)$

```
def checkMeanVariance(Lambda):
    randomTime = [nextTime(Lambda) for i in xrange(1000)]
    mean = sum(randomTime) / 1000
    variance = [x-mean for x in randomTime];
    variance = sum([x*x for x in variance])/1000
    expectedMean = 1/Lambda
    expectedVariance = expectedMean/Lambda
        print("Mean:" + str(mean) + " compare to " + str(expectedMean)+ "\n" + "Variance:" + str(variance)+ " compare to " +
str(expectedVariance))
def nextTime(rateParameter):
    return -math.log(1.0 - random.random()) / rateParameter
```

For Lambda = 75:

```
Lab1 $ python Lab1.py
Mean:0.0132434053893 compare to 0.0133333333333
Variance:0.000188891068463 compare to 0.000177777777778
Lab1 $ python Lab1.py
Mean:0.013032477334 compare to 0.0133333333333
Variance:0.000171351837958 compare to 0.000177777777778
Lab1 $ python Lab1.py
Mean:0.0132905588025 compare to 0.0133333333333
Variance:0.000173888385356 compare to 0.000177777777778
Lab1 $ python Lab1.py
Mean:0.0138144162272 compare to 0.0133333333333
Variance:0.000174434791397 compare to 0.000177777777778
Lab1 $ python Lab1.py
Mean:0.0132648395681 compare to 0.0133333333333
Variance:0.000161259948369 compare to 0.000177777777778
```

M/M/1 Queue

Question 2: Build your simulator for this queue and explain in words what you have done.

Before creating a DES, I have analyzed the system and created a few object class to make to system easier to understand and implement.

```
class packet:

    def __init__(self,arrivalTime,packetSize,serviceTime=0,new_dpTime=0):
        self.arrivalTime = arrivalTime
        self.packetSize = packetSize
        self.serviceTime = serviceTime
        self.departureTime = new_dpTime

class observer:
    def __init__(self,new_observeTime):
        self.observeTime = new_observeTime

class event:
    def __init__(self,new_type,new_time):
        self.type = new_type
        self.time = new_time
```

The packet class represents the packet in the system and has arrivalTime, packetSize, serviceTime, and departureTime for attributes.
The observer is an object used to check the state of the system and the observeTime will be generated according to a Poisson distribution.
The event class in a generalized class to keep track and trigger actions depending on the type of the events.

In order to create a DES for a simple queue with an infinite buffer, I have separated the work into different steps.

Step 1(Generating the list of observers):

I created a function that generates a set of random observation times according to a Poisson distribution with input parameter α and T. The function will continuously generate observer until the arrival time of the observer is larger than the total simulation time T. In the end, the function will return a list of observers and the arrival time of those observers will be a Poisson distribution.

```
def generateObserverList(T,Alpha):
   arrivalTime = nextTime(Alpha) #generate an arrival time
   newObserver = observer(arrivalTime) # create a observer for the new arrival time
   observerList = [newObserver]
   while(arrivalTime < T): # check if this arrival is still in the time period
      nextarrival = nextTime(Alpha) #generate an arrival time
      arrivalTime += nextarrival #increment the time counter
      newObserver = observer(arrivalTime) #create a observer for the new arrivaltime
      observerList += [newObserver] #add the new observer to the observer list
   return observerList
```

Step 2(Generating the list of packets):

I created another function that generates a set of packet arrival times (according to a Poisson distribution with parameter λ) and their corresponding length (according to an exponential distribution with parameter 1/L, L is the average length of a packet in bits), and calculate their departure times based on the state of the system (the departure time of a packet of length Lp depends on how much it has to wait and on its service time Lp/C where C is the link rate). The function will continuously generate observer until the arrival time of the observer is larger than the total simulation time T. In the end, the function will return a list of packets that is generated based on the simulation time T and the given λ.

```
def generatePacketList(T,Lambda):
    arrivalTime = nextTime(Lambda) #generate an arrival time
    packetSize = nextTime(1.0/12000.0) #generate packet size, L =12000bits
    serviceTime = packetSize/1000000 #calculate the service time, C = 1Mbits/sec
    departureTime = arrivalTime+serviceTime #calculate the departure time
    sojournTime  = departureTime - arrivalTime # calculate the sojourn time
    sojournList = [sojournTime]
    newPacket = packet(arrivalTime,packetSize,serviceTime,departureTime)
    packetList = [newPacket] #add the new packet to the packet list
    while(arrivalTime < T):# check if this arrival is still in the time period
        nextarrival = nextTime(Lambda)
        arrivalTime += nextarrival
        packetSize = nextTime(1.0/12000.0)
        serviceTime = packetSize/1000000
        if(arrivalTime >= packetList[-1].departureTime):
            #check if the queue is empty
            departureTime = departureTime = arrivalTime+serviceTime
        else:
            #calculated the departure time base on the last packet departure time
            departureTime = packetList[-1].departureTime + serviceTime
        sojournTime  = departureTime - arrivalTime
        sojournList += [sojournTime]
        newPacket = packet(arrivalTime,packetSize,serviceTime,departureTime)
        packetList += [newPacket]
    resultList = [packetList,sojournList]
    return resultList
```

Step 3 (Creating event list for the DES):

After generating the observer and packet list, I created a function that combines the observer list and packet list into a single list of different type of events. All the events have a specific time and a type indicating when will they happen and what will be triggered.

```
def createDES(packetList,observerList):
    eventList = []
    for i in packetList:
        arrivalEvent = event("Arrival",i.arrivalTime)
        eventList.append(arrivalEvent)
        departureEvent = event("Departure",i.departureTime)
        eventList.append(departureEvent)
    print("packetlist done")
    for i in observerList:
        observerEvent = event("Observer",i.observeTime)
        eventList.append(observerEvent)

    return eventList
```

When the eventList is finished, I used merge sort to sort the eventList by time. All the initialization has finished when the eventList is sorted.

Step 4 (Initialize variables and dequeue the event list):

In order to start the simulation and record all the results from dequeueing the events, I created a function that first resets all the counters for the different measurements. Then, the event handler loop through the event list, and update different counters depending on the type of the event. If it is an observer event, the event handler will record the number of packet in queue by calculating the difference between number of packets arrived and number of packets departure.

```
def eventHandler(eventList):
    NofArrival = 0 # reset all the counters
    NofDeparture = 0
    NofObservation = 0
    NofIdle = 0
```

```
packetInQueueCount = []
for i in eventList:
    if(i.type == "Arrival"):# checking the type of the event
        NofArrival = NofArrival+1#updating the counter
    elif(i.type == "Departure"):
        NofDeparture = NofDeparture+1#updating the counter
    else:
        NofObservation = NofObservation + 1#updating the counter
        packetCount = NofArrival-NofDeparture#calculating the packet in queue
        if(packetCount>0):
            packetCount = packetCount
        packetInQueueCount.append(packetCount)
        if(packetCount == 0):
            NofIdle = NofIdle + 1
return [NofArrival,NofDeparture,NofObservation,NofIdle,packetInQueueCount]
```

All the functions are called by the main function which has a loop that can simulate the same system with different values of ρ.

Question 3: Assume L=12000 bits, C=1 Mbits/second and give the following figures using the simulator you have programmed.

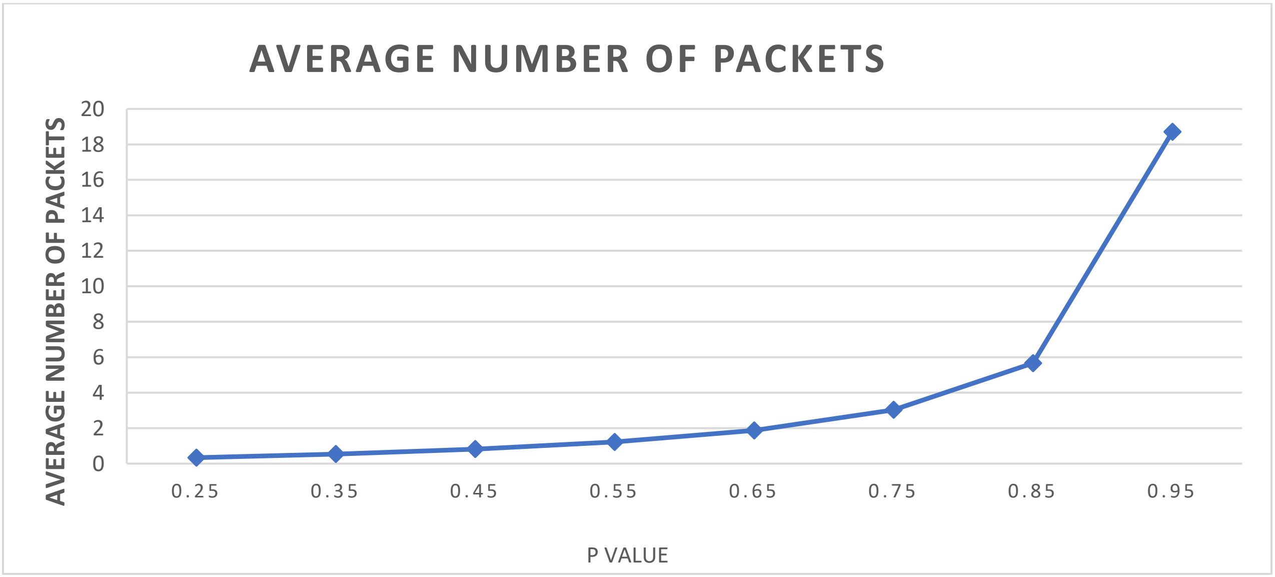
1. E[N], the average number of packets in the system as a function of (for 0.25< 0.95, step size 0.1).

For each round of simulation, there is a list of values about the number of packet in the queue that the observer events have record. In order to find the average number of packets in the system, I summed up all the values recorded by the observers, then I divided by the total number of observers to find out the average. In order see the patterns and the relationships between the value of ρ and the E[N], the system is simulated for different values of ρ from0.25 to 0.95 with a step size of 0.1, and also for different time period T =5000 and T = 10000.

T=5000	
Average number of packets	ρ value
0.332014252	0.25
0.535566329	0.35
0.817422998	0.45
1.223118938	0.55
1.877010333	0.65
2.958039823	0.75
5.505208142	0.85
19.8608686	0.95

T = 10000	
Average number of packets	ρ value
0.335288527	0.25
0.539220749	0.35
0.818205743	0.45
1.221324952	0.55
1.870468723	0.65
3.028574219	0.75
5.662590768	0.85
18.71074199	0.95

From the table for T = 5000 and T = 10000, we can see that the relationship between the average number of packets and the ρ value is consistent. As the value of ρ increases, the average number of packets also increases, this is what we expected to happen, because $\rho = L\lambda/C$ and λ is the average number of packets generated per second, so there are more packet being generated in the same time period as ρ increase, so there average number of packets in the system also increase.

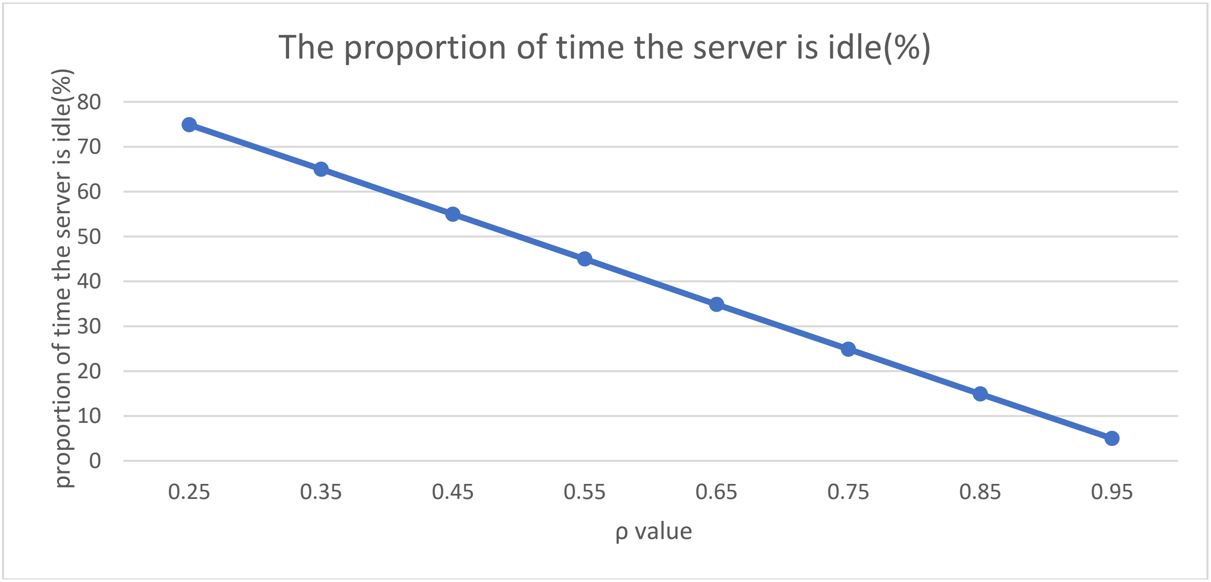


2. PIDLE, the proportion of time the system is idle as a function of ρ, (for 0.25 < 0.95, step size 0.1).

For each round of simulation, there is an idle counter that the observer events have being updating throughout the simulation. Whenever the observer event happens, it will check if the queue of the system is empty, if the queue is empty, it will increment the idle counter. In order to find the proportion of time the system is idle as a function of ρ, I used the idle counter and divided by the total number times of observer event happened to find proportion. In order see the patterns and the relationships between the value of ρ and the E[N], the system is simulated for different values of ρ from0.25 to 0.95 with a step size of 0.1, and also for different time period T =5000 and T = 10000.

T = 10000	
The proportion of time the server is idle(Unit %)	ρ value
74.88096038	0.25
64.97577024	0.35
54.96230015	0.45
44.98765556	0.55
34.84776015	0.65
24.86411717	0.75
14.89246664	0.85
4.997241171	0.95

From the table for T = 5000 and T = 10000, we can see that the relationship finds the proportion of time the system and the ρ value is consistent. As the value of ρ increases find the proportion of time the system decreases linearly, this is what we expected to happen, because ρ = L λ/C and λ is the average number of packets generated per second, so there are more packet being generated in the same time period as ρ increase, so the system have more packets to process, so the idle time will decrease.

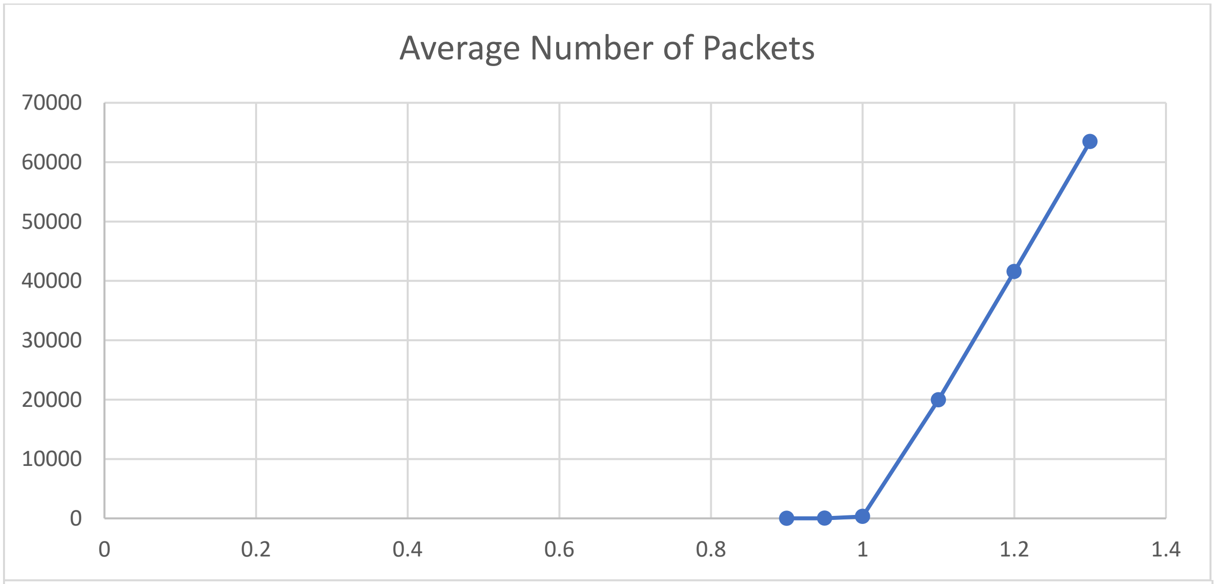


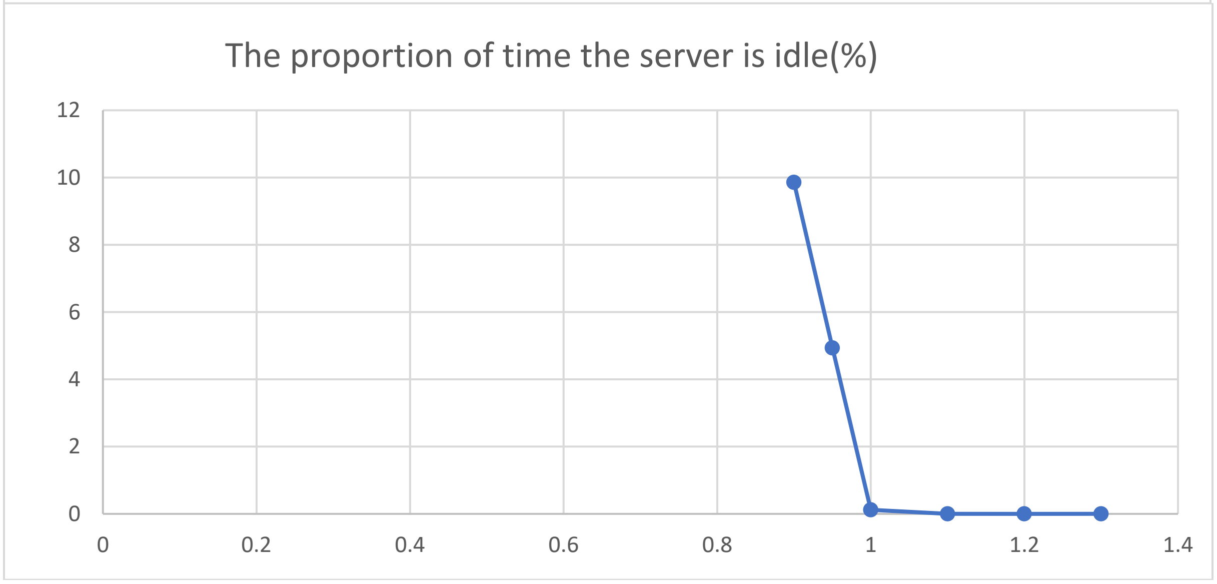
```
def infiniteBuffer(T):
    totalCSVResult = [['Average number of packets','The proportion of time the 
server is idle','Ro value']]
    for r in Ro:
        la = calculateLambda(r)
        checkMeanVariance(la)
        print(la)
        packetsList = generatePacketList(T,la)
        sojournList = packetsList[1]
        packetsList = packetsList[0]
        timeLength = len(sojournList)
        sojournTime = sum(sojournList)/timeLength
        observerList = generateObserverList(T,la*2)
        print("starting")
        eventList = createDES(packetsList,observerList)
        eventList = mergeSort(eventList)
        result = eventHandler(eventList)
        E = float(sum(result[4]))
        L = float(len(result[4]))
        meanOfPacket = E/L
        Pidle = result[3]/L*100
        print("Average number of packets " + str(meanOfPacket))
        print("Average sojourn time "+str(sojournTime))
        print("The proportion of time the server is idle "+str(Pidle))
        resultToCSV = [meanOfPacket,Pidle,r]
        totalCSVResult.append(resultToCSV)
        print("NofArrival: " + str(result[0])+ " NofDeparture: "+str(result[1])+
" NofObservation: "+str(result[2])+ " NofIdle: "+ str(result[3])+"\n")

    with open("Lab1Q2ResultT=10000.csv", "wb") as f:
        writer = csv.writer(f, delimiter = ',')
        for row in totalCSVResult:
             writer.writerow(row)
```

Question 4: For the same parameters, simulate for ρ = 1.2. What do you observe?

For the same parameters, when I change ρ to 1.2, I saw that lambda became 100 and caused the number of packets to increase significantly. As what we observed from question 3, when ρ increases, the number of packets per second also increase, causing the system to be busier. At ρ = 1.2, the system is also working 100% of the time. In order to verify the results that I saw, I simulated ρ from 0.9 to 1.3 with a step size of 0.1 and with the extra ρ = 0.95 to compare to the result above. In the table below, we can see that the average number of packets increases significantly when ρ increases, and the proportion of time the server is busy decreases and approaches to zero.

Average number of packets	The proportion of time the server is idle	Ro value
9.157203951	9.856926997	0.9
18.3374261	4.932292456	0.95
313.7319055	0.120196489	1
19957.19055	0.000436191	1.1
41570.83897	0.000200053	1.2
63466.89028	0.000461773	1.3





M/M/1/K Queue

Question 5: Build a simulator for an M/M/1/K queue.

Compare to the infinite queue simulator, there were 3 major change in the system that I have to make in order to have a limited queue DES. All the changes all made to record the major problem of a limited queue, which is packet loss.

Change 1 (Generating a packet list without departure time):

Compare to the infinite queue's packet list generation, I will not be able to generate the departure time when creating the packets. Instead, I have to calculate each packets departure time on the fly during the simulation. Thus, the packet list generation became simpler with only the packet arrival time and packetSize.

```
def generatePacketListLimitK(T,Lambda):
    arrivalTime = nextTime(Lambda)
    packetSize = nextTime(1.0/12000.0)
    newPacket = packet(arrivalTime,packetSize)
    packetList = [newPacket]
    while(arrivalTime < T):
        nextarrival = nextTime(Lambda)
        arrivalTime += nextarrival
        packetSize = nextTime(1.0/12000.0)
        newPacket = packet(arrivalTime,packetSize)
        packetList += [newPacket]
    return packetList
```

Change 2(Calculating the departure time during the simulation):

In an infinite queue system, handling the event list only need to dequeue and update counters depending of the time, but in a finite queue system, the packets only have arrival time and missing the departure time. So, I changed my event handler function to calculate a departure time, when the event is arrival and the queue is not full. After calculating the departure time, the function will create a new departure event and append to the event list and sort the event list again. On the other hand, when the event is arrival and the queue is full, the event handler will drop the packet and increment the packet drop counter.

```
def eventHandlerLimitK(eventList,K):
    NofArrival = 0 #Reset the counters
    NofDeparture = 0
    NofObservation = 0
    NofIdle = 0
    NofDropPacket = 0
    NofPacketGenerate = 0
    packetInQueueCount = []
    NofPacketInQueue = 0
    mostRecentDpartTime = 0
    lastDpIndex = 0
    while(eventList):
        i = eventList[0]
        if(i.type == "Arrival"): #check the event type
            packetSize = nextTime(1.0/12000.0)
            serviceTime = packetSize/1000000
            if (NofPacketInQueue < K+1): #check if the queue is full
                if(NofPacketInQueue == 0):#caclulate the departure time
                    departureTime = i.time + serviceTime
                    mostRecentDpartTime = departureTime
                else:
```

```
                departureTime = mostRecentDpartTime + serviceTime
                mostRecentDpartTime = departureTime
            departureEvent = event("Departure",departureTime)
            #create a new departure event and insert to the event list.
            resultList = departureInsert(eventList,departureEvent,lastDpIndex)
            lastDpIndex = resultList[1]
            eventList = resultList[0]
            NofArrival = NofArrival+1
            NofPacketInQueue = NofPacketInQueue + 1
        else:
            NofDropPacket = NofDropPacket + 1
    elif(i.type == "Departure"):
        NofDeparture = NofDeparture+1
        NofPacketInQueue = NofPacketInQueue -1
    else:
        NofObservation = NofObservation + 1
        packetCount = NofPacketInQueue
        packetInQueueCount.append(packetCount)
        if(packetCount == 1):
            NofIdle = NofIdle + 1
    eventList.pop(0)
    if(lastDpIndex > 0):
        lastDpIndex = lastDpIndex - 1
return [NofArrival,NofDeparture,NofObservation,NofIdle,NofDropPacket,packetInQueueCount]
```

Change 3(Optimization of event handler speed):

For each of the new departure event, the original implementation was append the new departure event to the event list and sort the event list again. However, I found out that this process took too long, because sorting the list is O(nlog(n)) and n is the size of the event list, which is very large when T is large. So, I create a function that remembers the index of the last departure event and insert the new departure into the event list, which only have worst runtime of O(n).

```
def departureInsert(eventList,departureEvent,dpIndex):
    i = dpIndex
    if(not eventList):
        return

    while(eventList[i].time < departureEvent.time):
        j = len(eventList) -1
        if(len(eventList) == 1):
            break

        if(len(eventList)-i == 1):
            break
        i = i + 1
    eventList.insert(i+1,departureEvent)
    result = [eventList,i+1]
    return result
```

Question 6: Let L=12000 bits and C=1 Mbits/second. Use your simulator to obtain the following figures

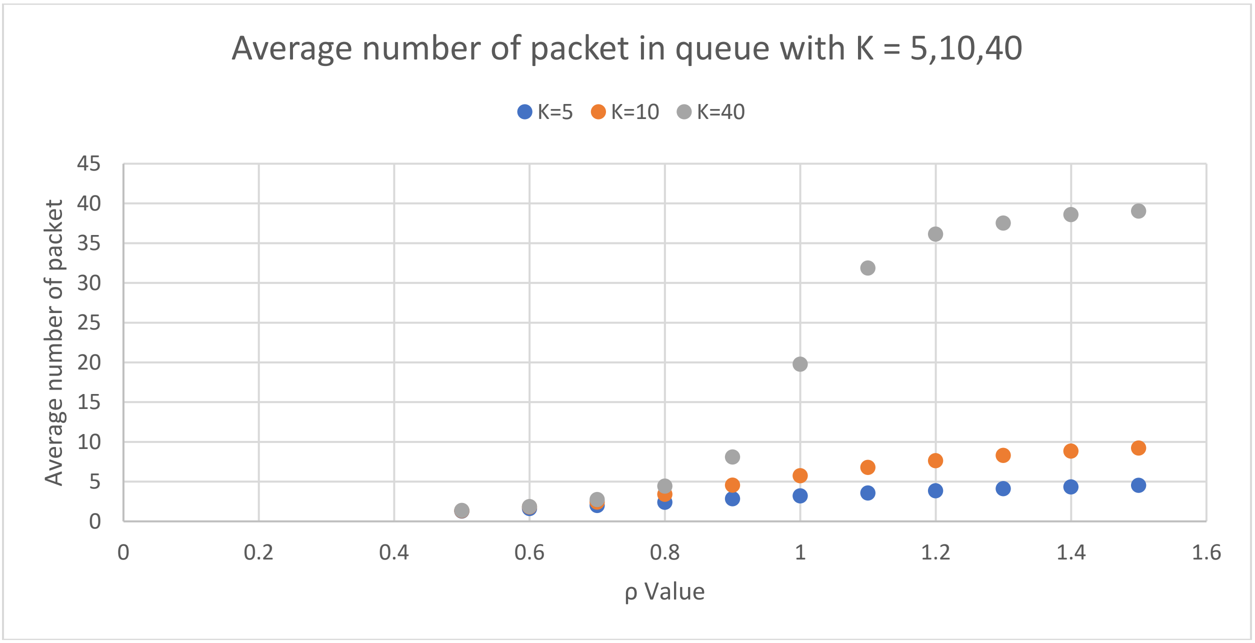
1. E[N] as a function of ρ (for 0.5 < ρ < 1.5, step size 0.1) for K=5, 10, 40 packets.

For each of the different size system queue, I designed the system to simulate over a range of ρ from 0.5 to 1.5 with a step size of 0.1. There is a list of values about the number of packets in the limited queue that each observer events have record. In order to find the average number of packets in the system, I summed up all the values recorded by the observers, then I divided by the total number of observers to find out the average. In order see the patterns and the relationships between the value of ρ, the E[N] and the size of the queue, the system is simulated for different values of ρ from 0.5 to 1.5 with a step size of 0.1 and with different queue size.

K=5	
Average number of packets	Ro value
1.27249612	0.5
1.597246316	0.6
1.981313222	0.7
2.37777678	0.8
2.816021162	0.9
3.178851675	1
3.534759824	1.1
3.830569456	1.2
4.090518217	1.3
4.303061556	1.4
4.51891548	1.5
K=10	
1.318152931	0.5
1.790433327	0.6
2.396060801	0.7
3.381342689	0.8
4.528836162	0.9
5.71804464	1
6.760556753	1.1
7.608829311	1.2
8.264669463	1.3
8.814000858	1.4
9.206309622	1.5

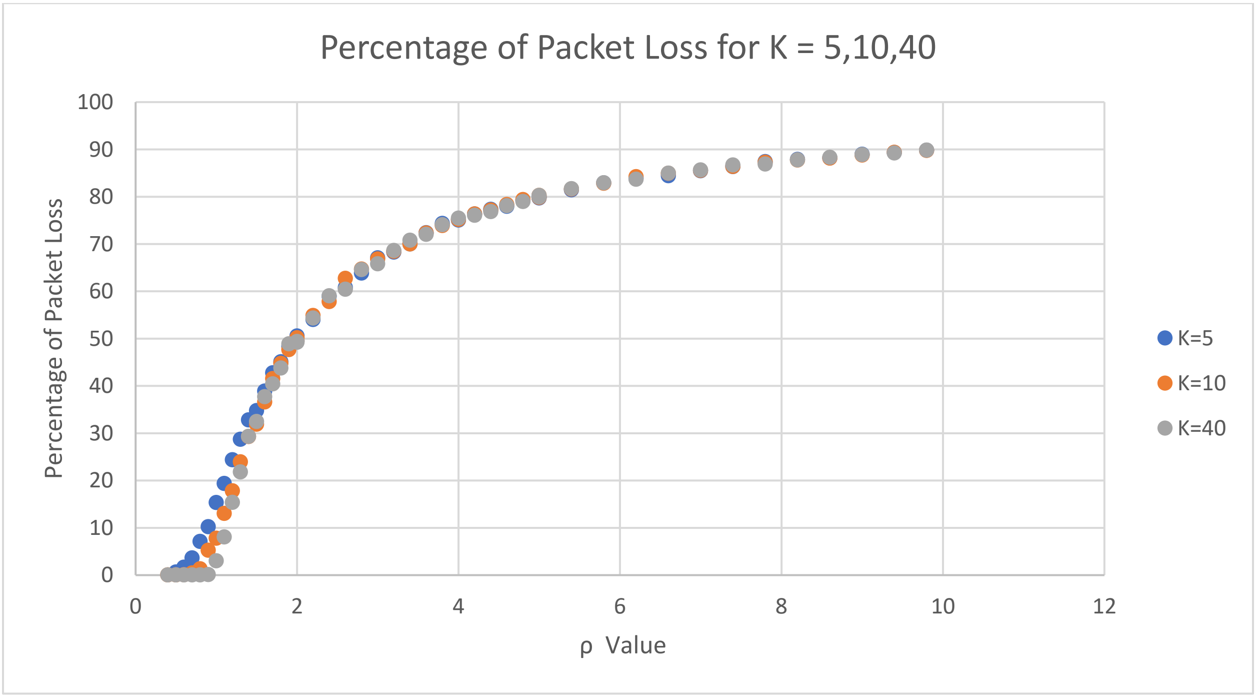
K=40	
1.350994732	0.5
1.830221511	0.6
2.708959624	0.7
4.417432849	0.8
8.069037992	0.9
19.74145256	1
31.85193458	1.1
36.11619335	1.2
37.49130647	1.3
38.56808651	1.4
39.01402324	1.5

From the simulation result, we can see that the relationship between the average number of packets in queue and the ρ value is consistent, when ρ increases, the average number of packets in queue also increases to it reaches the limit of the queue size. At the same time, we can see that as the queue size increases, the average number of packets in queue increases at a higher speed compare to the increase speed of the smaller queue size. This is what we expected to happen, because $\rho = L\lambda/C$ and λ is the average number of packets generated per second, so there are more packet being generated in the same time period as ρ increase, so the system have more packets to process, so there will be more packets in the queue waiting. On the other hand, the different increase speed is due to the upper limit of the queue size, when the upper limit is small and the queue size is small, more packet will be dropped when they arrive compare to a queue size that is larger, which allow more packets to be added to the queue in the early stage.

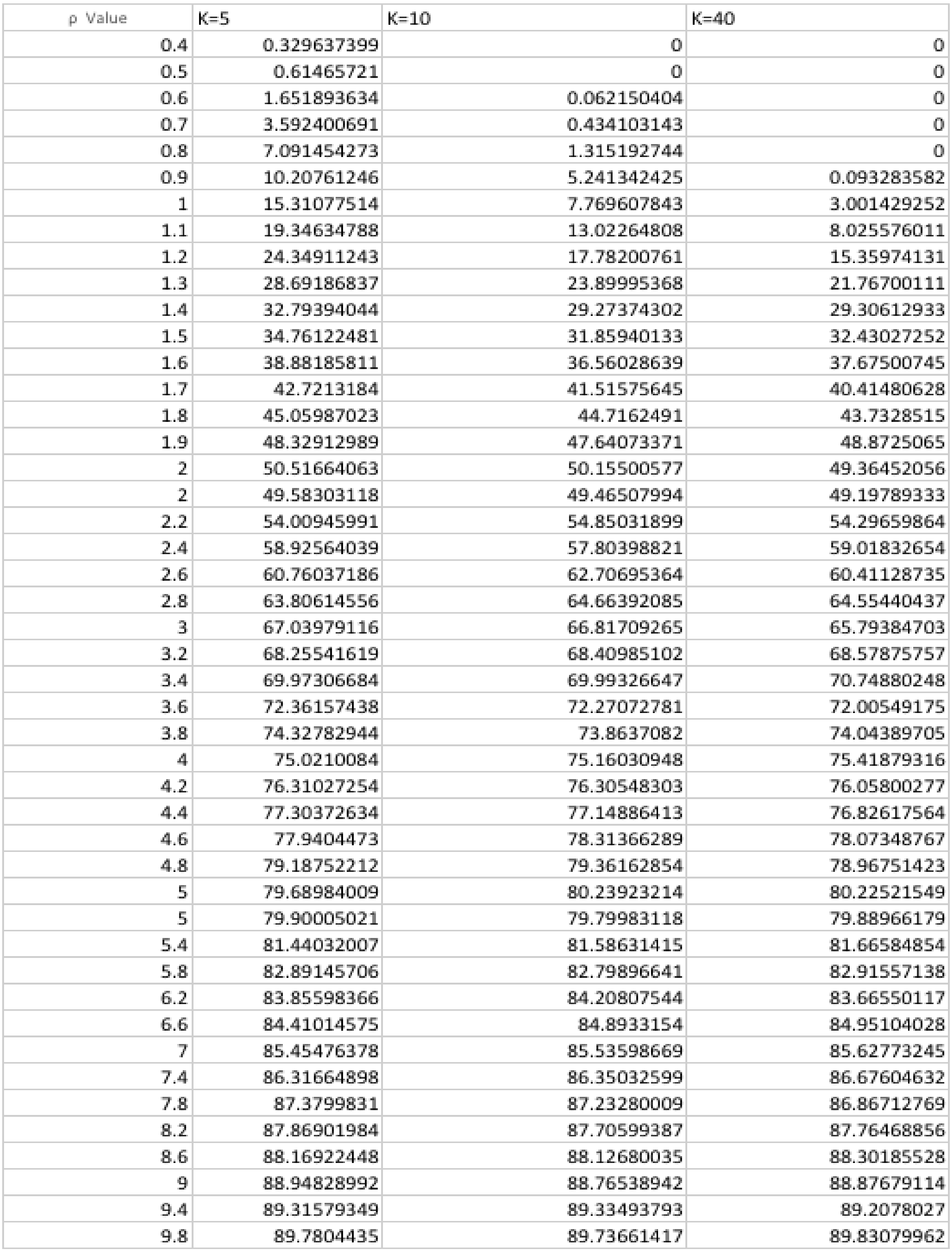


2. PLOSS as a function of (for 0.4 < ρ <10) for K=5, 10, 40 packets. (One curve per value of K on the same figure).

For each of the different size system queue, I designed the system to simulate over a range of ρ from 0.4 to 2 with a step size of 0.1, then from 2 to < 5 with a step size of 0.2, and from 5 to 10 with a step size of 0.4. Throughout the simulation, whether an arrival event happens and the queue is full, the simulator will drop the arrival packet and increment the packet loss counter. In the end of the one simulation, the function will use the packet loss counter divide by the total number of arrival events that got generated to find the percentage of the packet loos.



From the simulation result, we can see that the relationship between the percentage of packet loss and the ρ value is consistent, when ρ increases, the percentage of packet loss also increases to it. At the same time, we can see that as the queue size increases, the percentage of packet loss is increases slowly for ρ smaller than 2 and reach to the same value as ρ approaches 2. This is what we expected to happen, because ρ = L λ/C and λ is the average number of packets generated per second, so there is more packet being generated in the same time period as ρ increase, so the system has more packets to process, when the queue is full all the extra packets will be dropped.



ρ Value	K=5	K=10	K=40
0.4	0.329637399	0	0
0.5	0.61465721	0	0
0.6	1.651893634	0.062150404	0
0.7	3.592400691	0.434103143	0
0.8	7.091454273	1.315192744	0
0.9	10.20761246	5.241342425	0.093283582
1	15.31077514	7.769607843	3.001429252
1.1	19.34634788	13.02264808	8.025576011
1.2	24.34911243	17.78200761	15.35974131
1.3	28.69186837	23.89995368	21.76700111
1.4	32.79394044	29.27374302	29.30612933
1.5	34.76122481	31.85940133	32.43027252
1.6	38.88185811	36.56028639	37.67500745
1.7	42.7213184	41.51575645	40.41480628
1.8	45.05987023	44.7162491	43.7328515
1.9	48.32912989	47.64073371	48.8725065
2	50.51664063	50.15500577	49.36452056
2	49.58303118	49.46507994	49.19789333
2.2	54.00945991	54.85031899	54.29659864
2.4	58.92564039	57.80398821	59.01832654
2.6	60.76037186	62.70695364	60.41128735
2.8	63.80614556	64.66392085	64.55440437
3	67.03979116	66.81709265	65.79384703
3.2	68.25541619	68.40985102	68.57875757
3.4	69.97306684	69.99326647	70.74880248
3.6	72.36157438	72.27072781	72.00549175
3.8	74.32782944	73.8637082	74.04389705
4	75.0210084	75.16030948	75.41879316
4.2	76.31027254	76.30548303	76.05800277
4.4	77.30372634	77.14886413	76.82617564
4.6	77.9404473	78.31366289	78.07348767
4.8	79.18752212	79.36162854	78.96751423
5	79.68984009	80.23923214	80.22521549
5	79.90005021	79.79983118	79.88966179
5.4	81.44032007	81.58631415	81.66584854
5.8	82.89145706	82.79896641	82.91557138
6.2	83.85598366	84.20807544	83.66550117
6.6	84.41014575	84.8933154	84.95104028
7	85.45476378	85.53598669	85.62773245
7.4	86.31664898	86.35032599	86.67604632
7.8	87.3799831	87.23280009	86.86712769
8.2	87.86901984	87.70599387	87.76468856
8.6	88.16922448	88.12680035	88.30185528
9	88.94828992	88.76538942	88.87679114
9.4	89.31579349	89.33493793	89.2078027
9.8	89.7804435	89.73661417	89.83079962

```
def finiteBuffer(K):
    totalCSVResult = [['Average number of packets','The percentage of packet loss','Ro value']]
    for k in K:
        for r in RFinal:
            la = calculateLambda(r)
            checkMeanVariance(la)
            print(la)
            packetsList = generatePacketListLimitK(10000,la)
            packetListSize = len(packetsList)*1.0
            observerList = generateObserverList(10000,la*2)
            print("starting")
            eventList = createDESK(packetsList,observerList)
            print("sorting")
            eventList = mergeSort(eventList)
            result = eventHandlerLimitK(eventList,k)
            E = float(sum(result[5]))
            L = float(len(result[5]))
            print(sum(result[5]))
            meanOfPacket = E/L
            Pidle = result[3]/L*100
            print("Average number of packets " + str(meanOfPacket))
            print("The proportion of time the server is idle "+str(Pidle))
            packetLoss = result[4]/packetListSize*100
            print("The percentage of packet loss "+str(packetLoss))
            print("NofArrival: " + str(result[0])+ " NofDeparture: "+str(result[1])+ " NofObservation: "+str(result[2])+ "
NofIdle: "+ str(result[3])+ " NofPacketLoss: "+ str(result[4]))
            resultToCSV = [meanOfPacket,packetLoss,r]
            totalCSVResult.append(resultToCSV)
            gc.collect()


    with open("Lab1Q6ResultT=1000K=all.csv", "wb") as f:
        writer = csv.writer(f, delimiter = ',')
        for row in totalCSVResult:
             writer.writerow(row)
```