**::::::::Desktop:UW CMYK Colour.eps**

|  |  |
| --- | --- |
| By: | Cynthia Deng & William Gao |
| Student Number: | 20782862 & 20780743 |
|  |  |
| Submission date: | Friday October 14th 2022 |
|  |  |
| Submitted to: | Prof Irene Huang |
|  |  |

**Department of Electrical and Computer**

**Engineering**

**ECE 358: Computer Networks**

**Project 1: M/M/1 and M/M/1/K Queue**

**Simulation**

Contents

[Question 1 3](#_Toc116567761)

[I. M/M/1 Queue 3](#_Toc116567762)

[Question 2 3](#_Toc116567763)

[Question 3 8](#_Toc116567764)

[Question 4 9](#_Toc116567765)

[M/M/1/K Queue 9](#_Toc116567766)

[Question 5 9](#_Toc116567767)

[Question 6 12](#_Toc116567768)

# Question 1

|  |  |  |
| --- | --- | --- |
| **Run** | **Mean** | **Variance** |
| 1 | 0.013858672478097408 | 0.00017418973917540548 |
| 2 | 0.012303461910498893 | 0.00016470504001987526 |
| 3 | 0.013300150856802513 | 0.00017009661987798348 |
| 4 | 0.014624846865467396 | 0.00022944861550141895 |
| 5 | 0.013492076623601867 | 0.0001816752818396449 |
| **avg** | 0.013515842 | 0.000184023 |

The average mean is 0.013515842 while the average variance is 0.000184023.

This agrees with the expected values since the mean of the function is given by 1/𝜆, which in our case, with 𝜆 = 75, 1/ 𝜆 = 0.0133.

The variance also agrees with expected value since the expected value is 1/ 𝜆­2, which in our case is equal to 1.778\*10­-4.

# I. M/M/1 Queue

## Question 2

In our code, we have 5 defined functions that will be performing all the calculations. In the main function, we will be defining the variables needed and printing the final E(N) and P\_idle graphs.

Our 5 defined functions are **generateExponentialVar**(lamb), **populateArrays**(lamb, lambL, C, T), **createDES**(p\_a, p\_d, o), **queueProcessing**(DES, T), and **oneSimulation**(lamb, avg\_l, C, T).

**generateExponentialVar**(lamb) shown in Fig. 1 generates a random exponential value with the inverse method. The input of this function is the 𝜆 found in the equation. This function is used to generate the various random exponential values that are needed throughout the code. This includes inter-arrival time, inter-observation time, and packet length

Text

Description automatically generated

Figure 1. generateExponentialVar function

**populateArrays**(lamb, lambL, C, T) generates the multiples arrays that contain packet arrival times, packet departure times, as well observation times. Since this is an infinite queue, all these values are pre-generated and thus, this function is able to return packet departure times as well as the other two (which is not the case for finite queue).

In order to generate inter-arrival times, inter-observation times, as well as packet length, we specify the necessary lambda and run this through the previous function, **generateExponentialVar**. The arrival time that is appended to the arrival time array will then be a cumulative value of all the previously generated inter-arrival times. This logic also holds for observation times, with the difference being that the rate will be 5 times faster for observation times. A sample of the code used to generate arrival times is shown in Fig. 2 below.

Graphical user interface, text

Description automatically generated

Figure 2. Sample code used to generate arrival times

Departure time is calculated differently, and since it uses information from both arrival times and packet length, it is calculated after the above while loops. Since there is one packet departure for every packet arrival, a for loop will iterate that many times. In the for loop, a service time will be calculated each iteration based on the correlated random packet length. For the first arrival time, the departure will be directly correlated with arrival time plus the service time. For every packet after that, a check needs to be performed. If the current packet arrival time is greater than the previous departure time, then there is no queue, and the departure will be a simple sum of packet arrival time and packet service time. If the current packet arrives before the previous packet departs, that means there is a queue, and the packet needs to wait. This means the departure time will actually be the sum of the previous departure time and the current packet service time. The value that is calculated is then appended onto an array that will contain all the final departure times. The code that does this can be found in Fig. 3.

Text

Description automatically generated

Figure 3. Sample code used to generate the departure times

**createDES**(p\_a, p\_d, o) is the function used to create a combined list of Arrivals, Departures, and Observations, sorted by the time of the event. The inputs this function take are the 3 arrays that contain packet arrival times, packet departure times, and observation times. The output of this function is an array of dictionaries, where each dictionary represents one time point.

The logic of the **createDES** function is as follows. The code iterates through each of the 2 input arrays and adds a dictionary to the array with two keys. The two keys are “type” and “time”, where type corresponds with what type of event it was, and time is the numerical value found in the original array. The sample code for **createDES** can be found in Fig. 4.

Text

Description automatically generated

Figure 4. Sample code of createDES

The final function **queueProcessing**(eventTypes, DES, T) performs the calculations needed to find the E[N] and the PIDLE. The function requires the two arrays containing event types as well as event times, and the simulation time. It outputs the final E[N] and PIDLE.

**queueProcessing** will be iterating through eventTypes and DES simultaneously and if the DES value is greater than the simulation time, the process will stop. While iterating, depending on the event type value, NA, ND, and NO­ will be incrementing. When the event is an observation, an idle counter will also increment so that we can later calculate for PIDLE. During an observation, the number of packets in the queue will also be added to a total packet count which will be used to calculate E[N]. Finally, the function will actually calculate and return E[N] and PIDLE. Fig. 5 shows sample code of this function.

Text

Description automatically generated

Figure 5. Sample code of function queueProcessing

In terms of computing the performance metrics, E[N] was calculated as shown in Eq. 1 below, and PIDLE was calculated as shown by Eq. 2 below.

Equation 1. Formula used for calculating E[N] performance metric

Equation 2. Formula used for calculating PIDLE­ performance metric

In the **oneSimulation** function, it will be taking lamb, avg\_l, C and T as the inputs. This is where **populateArrays**, **createDES**, and **queueProcessing** are called. The final output is a printed value of E[N] and PIDLE.

Lastly, in the main function, the variables are initialized, and rho is then used to calculate the lambdas needed for the multiple iterations. The final outputs are 2 separate figures, one of E[N] vs rho and the other is of PIDLE vs rho. Fig. 6 below shows the for loop that runs through each iteration of rho

Text

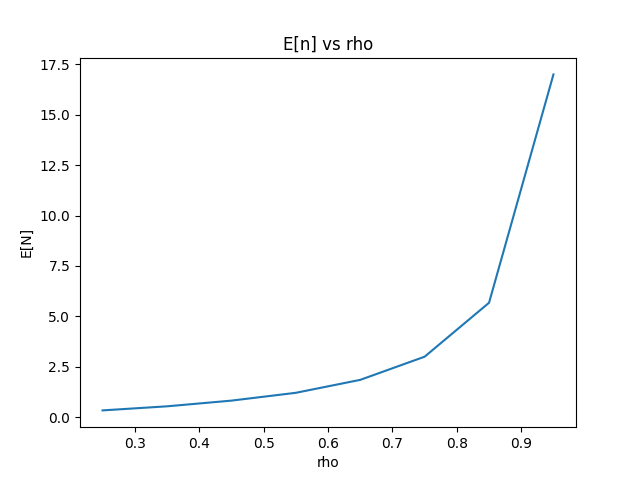
Description automatically generated

Figure 6. For loop that iterates through values of rho

The stability check was performed on T = 1000 and 2T = 2000, where lambda = 75. The variance in results between T and 2T are minimal enough that the time for simulation should be stable now.

|  |  |  |
| --- | --- | --- |
| **T** | **E[N]** | **PIDLE** |
| 1000 | 0.17313206296780262 | 0.852122932048538 |
| 2000 | 0.17571581703668546 | 0.8506730200039464 |

## Question 3

Chart, line chart

Description automatically generated

The E[N] vs rho graph is an exponential curve while the PIDLE vs rho is a linearly decreasing line.

The reason for the E[N] vs rho curve increasing is because as rho increases, there will be an increasing number of packets arriving per second. Since service time does not change, this means there will be a greater NA – ND the greater the value of rho. It is exponential as

The reason for the PIDLE vs rho line decreasing is because as rho increases, we can see that lambda also increases. As lambda increases, the number of packets arriving every second increases and leaves less time for the queue to be idle, thus resulting in decreasing PIDLE. The reason it is linear is because rho is the utilization of the queue, and this is directly correlated with idle time as 100% - rho (utilization) = PIDLE(idle time). Since rho is linear, PIDLE should also be linear.

The T in this simulation was 1000, since 2T holds stability, as shown in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **E[N]** | | **PIDLE** | |
| **Rho** | **T** | **2T** | **T** | **2T** |
|  | 0.3311 | 0.3325 | 0.7516 | 0.7507 |
|  | 0.5349 | 0.5384 | 0.6507 | 0.6498 |
|  | 0.8192 | 0.8168 | 0.5495 | 0.5499 |
|  | 1.2261 | 1.2221 | 0.4491 | 0.4506 |
|  | 1.8582 | 1.8513 | 0.3493 | 0.3505 |
|  | 2.9937 | 3.0302 | 0.2489 | 0.2492 |
|  | 5.7082 | 5.7180 | 0.1507 | 0.1490 |
|  | 20.0646 | 18.7293 | 0.0502 | 0.0493 |

## Question 4

For rho = 1.2, the E[N] is equal to 49483.3405 while the P\_IDLE is equal to 2.9995220761492e-06. While initially these numbers seem to be extreme, it follows the trend set by the graphs of an exponential growth of E[N] and a linearly decreasing trend of PIDLE. The queue at this point is essentially never idle, and as a result, the queue will accumulate more and more packages over time resulting in an E[N] that is exponentially growing towards infinity.

# M/M/1/K Queue

## Question 5

For the simulator built for the M/M/1/K Queue, it is very similar to simulator built for the M/M/1 Queue. It contains all the same functions that perform the same tasks. But the key difference appears in **populateArrays**. This is because, with a finite buffer size, the departure times for the packets cannot be pre-generated. Therefore, in the **populateArrays** function, the departure times are calculated with a queue and then appended to an array. The function takes in an additional argument, *k*, which is used to define the size of the queue buffer.

Text

Description automatically generated

Figure 7. Sample of code to generate and processes departure times in M/M/1/K simulator

All possible arrival times of a packet are generated through the same way as the M/M/1 simulator and stored into an array called*all\_packet\_arrivals.* Observation times are also generated the same way as the M/M/1 simulator. The function then iterates through all of the possible arrival times and generate a departure time through the service time. Once the departure time is generated, it will be added into the *departure\_queue*. The departure queue is processed and ran through if the arrival time of the packet is greater than the first element of the queue. It will then append the time to an array that keeps track of successful departure times. The function will also append the corresponding arrival time to an array called *actual\_packet\_arrivals* which keeps track of arrival times of packets that were not lost. Figure 7. Shows the code for the updated **populateArray** departure time calculations. A variable called *total\_lost* and *total\_created* are returned from the function additionally now for the purpose of calculating PLOSS.

Functions **queueProcessing** and **oneSimulation** were slightly adjusted from the M/M/1 simulator. **queueProcessing** now has additional arguments, *total\_lost* and *total\_created\_packets* which are arguments used for calculating the performance metric PLOSS­ and the function also returns it with the variable *P\_LOSS*. The PLOSS of the simulation is calculated by dividing the total amount of packets lost with the total packets generated as shown in figure 8. The other performance metrics were calculated the same was as the M/M/1 simulator. **oneSimulation** has an additional argument, *K*, which is used to define the queue buffer size for **populateArray** to use.

Text

Description automatically generated

Figure 8. Calculations for the performance metrics

The other functions, **generateExponentialVar** and **createDES** were not changed from the M/M/1 simulator and perform the same tasks.

Text

Description automatically generated

Figure 9. Initial variables used to test M/M/1/K simulator

The stability check for this simulator was performed with the variables shown in figure 8. T was increased from T=1000 to T=2000 and the results are stable and consistent as shown in the table below.

|  |  |  |
| --- | --- | --- |
| **T** | **E[N]** | **PLOSS** |
| 1000 | 6.725180043080399 | 0.19504449908807212 |
| 2000 | 6.7183604650368025 | 0.19296183132224523 |

## Question 6

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Figure 10. Graphs of E[n] and PLOSS for the M/M/1/K simulator

The following tables show the stability check for when T goes from 1000 to 2000. All the differences in the values when T increases are negligible.

Table below is outlining the performance metrics for K = 10

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rho** | **E[n]** | | **PLOSS** | |
|  | **T = 1000** | **T = 2000** | **T = 1000** | **T = 2000** |
| 0.5 | 0.996148766064 | 1.000798709905 | 0.000518636886096 | 0.0005263421223795 |
| 0.6 | 1.462591609541 | 1.475895346894 | 0.002593968855702 | 0.0024104287666694 |
| 0.7 | 2.120384719911 | 2.114989819331 | 0.008531268737813 | 0.0086573593982556 |
| 0.8 | 2.958124539010 | 2.970835215530 | 0.023287919618349 | 0.0239626483360349 |
| 0.9 | 3.942621381907 | 3.973827226545 | 0.049395972557546 | 0.0514628883758479 |
| 1.0 | 5.006034212330 | 4.998852564538 | 0.091325820868077 | 0.0914257494434219 |
| 1.1 | 5.933767721687 | 5.962419657107 | 0.139946656675781 | 0.1412683132577214 |
| 1.2 | 6.706252496537 | 6.700986850874 | 0.192388086932832 | 0.1909369624616349 |
| 1.3 | 7.309804234613 | 7.309515254800 | 0.243519043248470 | 0.2444113653183037 |
| 1.4 | 7.769015041824 | 7.777968256871 | 0.290985662146392 | 0.2925789744763649 |
| 1.5 | 8.121641704691 | 8.133312468951 | 0.337643991669619 | 0.3374924550890216 |

Table below is outlining the performance metrics for K = 25

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rho** | **E[n]** | | **PLOSS** | |
|  | **T = 1000** | **T = 2000** | **T = 1000** | **T = 2000** |
| 0.5 | 1.0044561977967534 | 0.99715345480884 | 0.0 | 0.0 |
| 0.6 | 1.492643943532954 | 1.51204570039031 | 0.0 | 0.0 |
| 0.7 | 2.334307246853928 | 2.34181259451455 | 3.85492902647e-05 | 3.00051437389e-05 |
| 0.8 | 3.9542593387526943 | 3.89940124639118 | 0.0007975149534053 | 0.000671660552512 |
| 0.9 | 7.243439740768061 | 7.19255150449022 | 0.0079703577331129 | 0.007521279718227 |
| 1.0 | 12.490589499180327 | 12.4758754109112 | 0.0385504595409088 | 0.038000146038992 |
| 1.1 | 17.33176969984388 | 17.2774282335467 | 0.0986232476497247 | 0.097967438936617 |
| 1.2 | 20.212399412691166 | 20.2451944380955 | 0.1683797374382670 | 0.168549108036345 |
| 1.3 | 21.723717308405156 | 21.7085014917327 | 0.2320411554693028 | 0.231258499633861 |
| 1.4 | 22.51070658841379 | 22.5293378196685 | 0.2863348739270865 | 0.286386658492843 |
| 1.5 | 22.988852571450497 | 23.0103302040426 | 0.3330721985460015 | 0.334130518390246 |

Table below is outlining the performance metrics for K = 50

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rho** | **E[n]** | | **PLOSS** | |
|  | **T = 1000** | **T = 2000** | **T = 1000** | **T = 2000** |
| 0.5 | 1.0005173396746756 | 1.00115651657687 | 0.0 | 0.0 |
| 0.6 | 1.488292532664094 | 1.51352062429729 | 0.0 | 0.0 |
| 0.7 | 2.359262475363478 | 2.35731015369139 | 0.0 | 0.0 |
| 0.8 | 3.9623881272799646 | 4.00965780996881 | 1.2513060506904082e-06 | 0.0 |
| 0.9 | 8.429612763052925 | 8.97072165956368 | 0.00047995652551564703 | 0.0007167550664 |
| 1.0 | 25.57334661174415 | 24.9286407680604 | 0.02035705638585461 | 0.0193486413869 |
| 1.1 | 40.17688001177438 | 40.3079659217776 | 0.08991470163805308 | 0.0909207538105 |
| 1.2 | 45.03651576346937 | 44.9535387135151 | 0.1660445864013643 | 0.1657787009989 |
| 1.3 | 46.677456722470154 | 46.6256808873909 | 0.23211153731803086 | 0.2299841543781 |
| 1.4 | 47.51144678111257 | 47.4960495082207 | 0.2859072132757028 | 0.2855645942959 |
| 1.5 | 48.0040840591172 | 47.9740095148468 | 0.33322695370953964 | 0.3328599928098 |

The results for the simulations are to be expected. In the first graph of figure 10, E[n] vs rho, when K = 10 the average number of packets in the buffer/queue is a smooth curve because the queue is only a size of 10 and therefore as rho increases the E[n] approaches 10. Once K is increased to 25, the average number of packets in the buffer/queue curve rises at a faster rate for larger values of rho and eventually approaches the max buffer size number of 25. And finally, when K = 50, the curve of E[n] rises exponentially for values of rho and approaches its max size. This is because, the average number of packets per second generated is derived from the rho value and as it rises, the packets generated per second rises while the service time is still the same. Therefore, at larger rho values the number of packets in the queue will rise too. And this also explains the curves for the PLOSS vs rho graph. As the rho value increases, the number of packets that are lost from the queue being full will increase exponentially as well.