University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

ECE 358 Project 1: M/M/1 and M/M/1/K Queue Simulation

Prepared by

Alex Chun Kei Chiu

ackchiu

20615465

Frank Liang

Table of Contents

[1 Exponential Random Variables 2](#_Toc536558058)

[2 M/M/1 E[N] QES Design and Discussion 2](#_Toc536558059)

[2.1 Average Number of Packets in Queue E[N] 2](#_Toc536558060)

[2.2 Proportion of Time System is Idle 2](#_Toc536558061)

[3 M/M/1 E[N] and as a Function of ρ 2](#_Toc536558062)

[4 M/M/1 E[N] and when ρ = 1.2 2](#_Toc536558063)

# 

# 1 Exponential Random Variables

A class was written to generate exponential random variables whenever the class instance calls a genValue method. In this method, a uniform random variable is generated and uses it along with an inputted lambda value to generate a single exponential random via the inverse method. Theoretically, exponential distributions should have an expected value of and a variance of . When 1000 exponential random variables were generated, their expected value and variance were very close to that of the theoretical value. Running lab1.py and selecting question one will produce the following data.

Table 1: Theoretical and Experiment Values of Generated Exponential Random Variables

|  |  |  |  |
| --- | --- | --- | --- |
| **Metric** | **Theoretical Value** | **Experimental Value** | **Error %** |
| Expected Value E[N] | 0.013333 | 0.013235 | 0.735 % |
| Variance | 0.000178 | 0.000169 | 5.056 |

# 2 M/M/1 E[N] QES Design and Discussion

In order to generate a clean and comprehensive simulator, several separations of concern were made. The *Packet* data structure is self-explanatory, created by giving a packet size. It contains one method for which it is to calculate the transmission time given the transmission rate. The *Event* data structure contains a timestamp and identifies as one of three types: arrival, observer and departure.

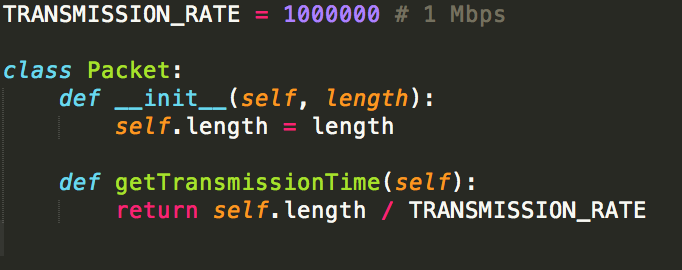
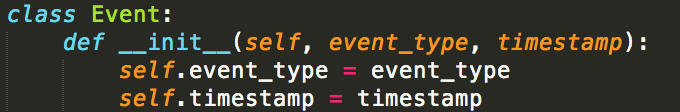


Figure 1: Event and Packet Class in Python 2.7

The simulator is designed as a class being initialized by input a value of rho. The queue will just be an array of packet objects and the simulation will loop through a sorted array of events, processing data depending on its type. The simulation will first generate eventsand packets, process the events, then print the results. The three steps are encapsulated into their own private methods.

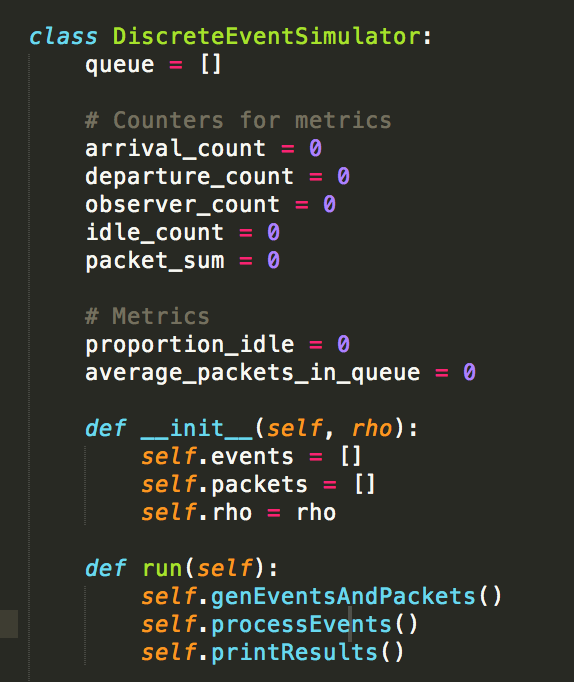


Figure 2: DiscreteEventSimulator Initial State and Run Method

Arrival event timestamps, observer event timestamps and packet lengths are generated using the exponential random variable generator in section 1. This is done so by inputting the correct lambda values based on the inputted rho, calculated as shown in figure 3.

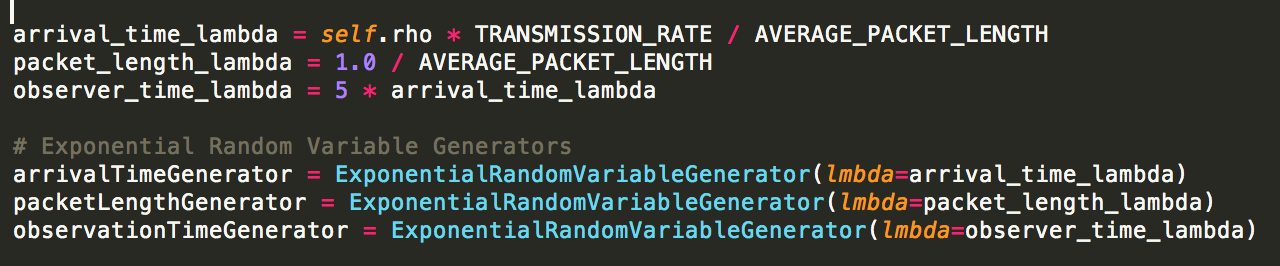


Figure 3: Initializing Lambda Values and Exponential Random Variable Generator

Each time genValue is called on a generator, a random variable will be outputted with the generator’s respective lambda. The generated value is added to a variable which tracks the current timestamp. This becomes the next event’s timestamp. This is done for arrival and observer events. Whenever arrival events are generated, a packet and departure event will also be generated. For departure events, a variable is needed to track the previous calculated departure time because the queue may be servicing a packet. If the queue is servicing a packet, the next departure time will be calculated as the previous calculated departure time plus the generated packet’s transmission time. If the queue is empty, then the departure time is calculated as the current timestamp plus the transmission time. Whether or not the queue is empty is determined by whether or not the current timestamp exceeds the previous calculated departure time. The logic here is that the previous calculated departure time always represents the departure time of the last packet in the queue. If the current time is greater, then that means the event is entering the queue where all its packets have been serviced. If the current calculated departure time is less than that of the simulation time, save the generated arrival event, departure event and packet.

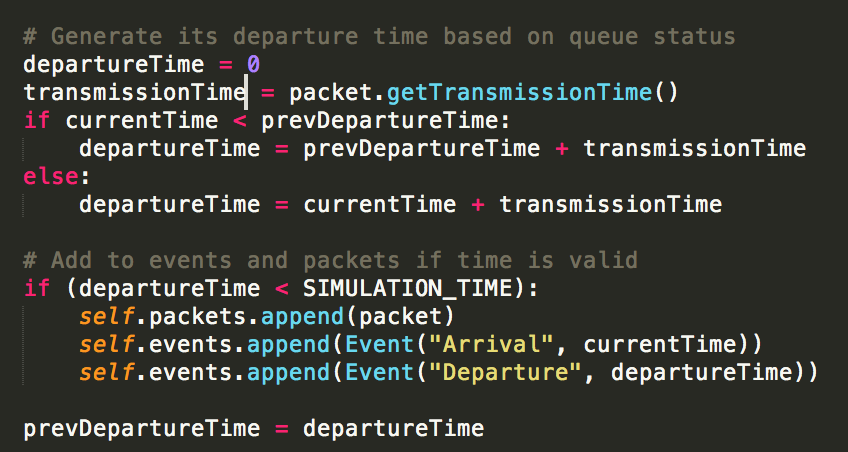


Figure 4: Departure Time Calculation and Event Saving

The event types are processed as follows. For arrivals, simply put the next packet that was saved into the back of the queue and increment the arrival count. For departures, remove the packet from the front of the queue and increment the departure count. For observer events, the observer count will be incremented, and the idle count will be increment if the queue is empty at this point in time. There are two metrics calculated in the observe event: the average number of packets in the queue, E[N] and idle proportion, . E[N] is calculated by keeping a variable to track the sum of the observer’s observed number of packets in the queue. This sum is then divided by the observer count. is calculated by taking the idle count divided by the observer count. The simulation time is chosen to be T

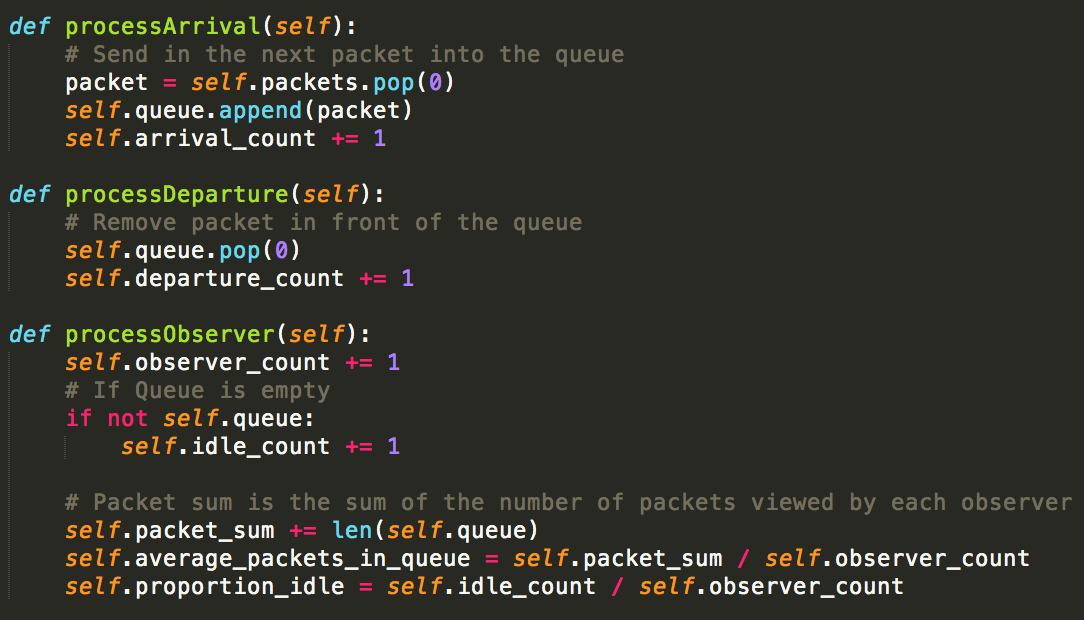


Figure 5: Logic for Processing Each Event Type

The simulation time was chosen to be 1000. To test whether or not this produces stable results, E[N] and for a rho of 0.55 was measured at simulation times 1000, 2000 and 3000. The results were within 5% error as indicated in the table below.

Table 2: E[N] and for Varying Simulation for ρ = 0.55

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Simulation Time** | **E[N]** | **Error %** |  | **Error %** |
| 1000 | 1.228 |  | 0.44986 |  |
| 2000 | 1.227 | 0.08 | 0.44979 | 0.16 |
| 3000 | 1.218 | 0.81 | 0.45017 | 0.07 |

# 3 M/M/1 E[N] and as a Function of ρ

The simulator has been designed such that the average number of packets for an inputted traffic intensity rho will be outputted after the simulation. Only a loop is needed to reinitialize the simulator multiple times for each distinct value of rho. As the traffic intensity is increased, the average number of packets E[N] increases exponentially. This makes sense because the departure time of an incoming packet depends on all the departure times prior to it. Therefore, as the input rate increases with a decreasing departure time, the number of packets in the queue will ultimately increase.

Figure : Average Number of Packets vs Traffic Intensity

Idle proportion is outputted at the same time E[N] is outputted. As the traffic intensity is increased, the idle proportion is decreased, indicating that the queue becomes less likely to be empty. With an exponentially increasing E[N], it is trivial to see that the number of packets to serve will almost never approach 0. As a result, the idle proportion will ultimately be 0 near high traffic intensities such as 0.95.

Figure : P\_Idle vs Traffic Intensity

# 4 M/M/1 E[N] and when ρ = 1.2

With a rho of 1.2, it represents an extremely high input rate. This can be shown by the increased number of event counts and the average number of packets in the queue. The results show that even with an unlimited buffer size, the input rate was significantly higher than the output rate, enough to reduce the idle time to almost 0 as shown in the table below.

Table : M/M/1 Simulation Results for ρ =1.2

|  |  |
| --- | --- |
| **Metrics** | **Value** |
| Arrival Count | 500193 |
| Departure Count | 500193 |
| Observer Count | 2999782 |
| Average Number of Packets E[N] | 41390.08 |
| Idle Proportion |  |

# Source Code: Lab1.py

# Source Code: ExponentialRandomVariableGenerator.py

|  |
| --- |
| import numpy as np  import math  class ExponentialRandomVariableGenerator:  def \_\_init\_\_(self, lmbda):  # input uniform distribution  self.lmbda = lmbda  def genValue(self):  if (self.lmbda > 0):  # Generate uniform random variable and then use inverse method to calcualte exponential  uniform\_random\_variable = np.random.uniform(0, 1)  return (-1.0/self.lmbda) \* math.log(1.0-uniform\_random\_variable)  else:  raise ValueError('Cannot generate exponential distribution with invalid lambda') |

# Source Code: Event.py

|  |
| --- |
| class Event:  def \_\_init\_\_(self, event\_type, timestamp):  self.event\_type = event\_type  self.timestamp = timestamp |

# Source Code: Packet.py

|  |
| --- |
| TRANSMISSION\_RATE = 1000000 # 1 Mbps  class Packet:  def \_\_init\_\_(self, length):  self.length = length  def getTransmissionTime(self):  return self.length / TRANSMISSION\_RATE |

# Source Code: DiscreteEventSimulator.py

|  |
| --- |
| from \_\_future\_\_ import division  from Event import Event  from ExponentialRandomVariableGenerator import ExponentialRandomVariableGenerator  from Packet import Packet  AVERAGE\_PACKET\_LENGTH = 2000  SIMULATION\_TIME = 1000  TRANSMISSION\_RATE = 1000000 # 1 Mbps  class DiscreteEventSimulator:  queue = []  # Counters for metrics  arrival\_count = 0  departure\_count = 0  observer\_count = 0  idle\_count = 0  packet\_sum = 0  # Metrics  proportion\_idle = 0  average\_packets\_in\_queue = 0  def \_\_init\_\_(self, rho):  self.events = []  self.packets = []  self.rho = rho  def run(self):  self.genEventsAndPackets()  self.processEvents()  self.printResults()  def genEventsAndPackets(self):  arrival\_time\_lambda = self.rho \* TRANSMISSION\_RATE / AVERAGE\_PACKET\_LENGTH  packet\_length\_lambda = 1.0 / AVERAGE\_PACKET\_LENGTH  observer\_time\_lambda = 5 \* arrival\_time\_lambda  # Exponential Random Variable Generators  arrivalTimeGenerator = ExponentialRandomVariableGenerator(lmbda=arrival\_time\_lambda)  packetLengthGenerator = ExponentialRandomVariableGenerator(lmbda=packet\_length\_lambda)  observationTimeGenerator = ExponentialRandomVariableGenerator(lmbda=observer\_time\_lambda)  currentTime = 0  prevCalculatedDepartureTime = 0  # Generate Arrival, and if M/M/1, generate Departure  while currentTime < SIMULATION\_TIME:  # Add inter-arrival time to arrive at current timestamp  interArrivalTime = arrivalTimeGenerator.genValue()  currentTime += interArrivalTime  # Generate packet and its length  packet = Packet(length=packetLengthGenerator.genValue())  # Generate its departure time based on queue status  departureTime = 0  transmissionTime = packet.getTransmissionTime()  if currentTime < prevCalculatedDepartureTime:  departureTime = prevCalculatedDepartureTime + transmissionTime  else:  departureTime = currentTime + transmissionTime    # Add to events and packets if time is valid  if (departureTime < SIMULATION\_TIME):  self.packets.append(packet)  self.events.append(Event("Arrival", currentTime))  self.events.append(Event("Departure", departureTime))    prevCalculatedDepartureTime = departureTime  # Generate Observer Events  currentTime = 0  while currentTime < SIMULATION\_TIME:  interArrivalTime = observationTimeGenerator.genValue()  currentTime += interArrivalTime  if (currentTime < SIMULATION\_TIME):  self.events.append(Event("Observer", currentTime))  # Sort Events for processing chronologically  self.events.sort(key=lambda event: event.timestamp)  def processEvents(self):  for event in self.events:  if event.event\_type == "Arrival":  self.processArrival()  elif event.event\_type == "Departure":  self.processDeparture()  elif event.event\_type == "Observer":  self.processObserver()  def processArrival(self):  # Send in the next packet into the queue  packet = self.packets.pop(0)  self.queue.append(packet)  self.arrival\_count += 1  def processDeparture(self):  # Remove packet in front of the queue  self.queue.pop(0)  self.departure\_count += 1  def processObserver(self):  self.observer\_count += 1  # If Queue is empty  if not self.queue:  self.idle\_count += 1  # Packet sum is the sum of the number of packets viewed by each observer  self.packet\_sum += len(self.queue)  self.average\_packets\_in\_queue = self.packet\_sum / self.observer\_count  self.proportion\_idle = self.idle\_count / self.observer\_count  def getAveragePacketsInQueue(self):  return self.average\_packets\_in\_queue  def getIdleProportion(self):  return self.proportion\_idle  def printResults(self):  print("Counts", self.arrival\_count, self.departure\_count, self.observer\_count)  print("Average Packets In Queue ", self.average\_packets\_in\_queue)  print("Idle Proportion ", self.proportion\_idle) |

# Source Code: DiscreteEventSimulator.py