**ECE 358 Lab 1**

Jung Tae Kim, 20302725, jt2kim

Jay Mahendru, 20300785, kmahendr

Date: June 4th, 2012

1. The average service time received by a packet is L/C. The utilization of the queue can be expressed as (Arrival Rate)/(Service Rate) which is equal to (Arrival Rate) \* (Service Time) = (λ \* L)/C, where Lambda is the arrival rate in Hertz (1 / sec), L is the length of the packet in bits, and C is the bandwidth of the outgoing link in bits/sec.

**Variables:**

*queue<double> buffer:*  
The queue used to store the packets

*double t\_arrival :*

the rate at which packets arrive

*double t\_departure:*  
 the time left for a packet to finish getting served

*transmissionTime:*

length of the packet divided by the transmission rate (C), which represents the amount of time required to serve a packet

*bufferSize:*

specified by user via “K” of the user input section. Set to -1 to indicate infinite size when the user does not specify the size

*ticks:*   
 number to be multiplied by 100000 to indicate a runtime in microseconds

*runningQueueSizeSum, runningDelaySizeSum, runningIdleSizeSum:*these values hold the running sums of queue size, delay length, and idle time.

*queueSizeCtr, delaySizeCtr, idleSizeCtr:*keeps track of how many times each of quesize, delaySize, and idleSize have been summed so that correct average can be calculated when the simulation is over.

**Functions:**

To explain the logic behind the code, let us look at the code in terms of the individual methods:

*int main()*:

This function will receive input from the command line and set ticks, lambda, L, and C values and initialize t\_arrival and t\_departure

*void startSimulation (int ticks)*:  
This function will initiate a loop which simulates a tick at each iteration. Other than keeping count of the current tick, this function will also keep track of the running sum of the buffer size

*void arrival (double t)*:  
This function will decrement t\_arrival counter and check how many more ticks must pass until a new packet arrives.  
  
When it is time for a packet to arrive (t\_arrival = 0), it will check if the buffer is full and add the packet to the buffer if it is not full or if there is no limit to the buffer size.  
  
This function will also reset the idle time to zero whenever a packet arrives, and update the running sum of total idle time.

*void departure (double t)*:  
This function will remove a packet from the front of the buffer after a packet has been served. When a packet enters a server, the variable “currentlyServing” will be set to the packet’s arrival time, and t\_departure will be decremented. Once the t\_departure gets decremented down to zero, a packet fron the front of the buffer will be removed, and t\_departure will be reset to “transmissionTime” calculated in the main function. t\_departure will start to get decremented again when a packet is being served again, i.e currentlyServing is not set to -1 anymore.

*void computePerformances()*:

This function will use the total sums of delay time, average size of the buffer, and total idle time to calculate the average queue size, average delay time, and the average idle time if a M/D/1/K format is being used, this program will also output the % of packets dropped.

1. Based on the given inputs, our code generated the following outputs:
   * 1. E[N] was calculated by adding up the number of packets in the queue at each tick and dividing it by the total number of packets generated during the span of the simulation.  
          
        For the given inputs, out code generated the value of **E[N] = 0.229657**
     2. E[T] was calculated by taking the time difference between the arrival of a packet at the queue and the time at which it leaves the server.   
          
        For the given inputs, our code generated the value of **E[T] = 22.4953 µs**
     3. P­Idle was calculated by summing up how many microseconds the queue has been idle for and dividing it by the total number of times the summation had been done.  
          
        For the given inputs, our code generated an average value of **P­Idle = 8013.03 µs**
2. Since the input to our application is uses λ and not ρ, we can redefine the problem in terms of λ. We know that ρ = (λ \* L) / C. Therefore, λ = (ρ \* C) / L. Hence, we can express the problem with the three parts in terms of 100 < λ < 475 with a step size of 25 λ:

**E[N] as a function of ρ:** As the utilization rate ρ increases, it takes a longer to service each packet. As a result, the average number of packets in the queue increases. The graph below proves the same.

**E[T] as a function of ρ:** The sojourn time is the sum of the queuing delay and the service time for each packet. If the utilization rate is high, that is, there are more packets coming in, than going out, it would take longer to service the packets and the queuing delay will increases as well, or, in other words, the average sojourn time will increase as the utilization rate increases. The graph given below proves the same.

**Pidle as a function of ρ:** If the utilization rate is high, it takes longer to service each packet, which means that the queue will be idle much less since there are other packets waiting in the queue. The graph given below demonstrates the same.

**5.** E[N] as a function of ρ:

E[T] as a function of ρ:

**6.**  When p is 1.2, E[N] = 275521, E[T] = 4.58639e+08 μS and Pidle = 1.26436e-06 μS.

**7.**  In order to change the M/D/1 simulator to M/D/1/K queue simulator, only three very minor changes were required.

1. Additional input parsing code was added to set the buffer size to K
2. Add a simple if statement to check if the buffer is full before adding a packet
3. Adding a counter to keep track of number of lost packets

The new variables introduced were:

* *bufferSize*: used to keep track of the size of the buffer and check if the buffer is full or not
* *packetsDropped, packetsAdded*: used to keep track of total number of packets generated and number of dropped packets

**8.**

This graph shows the average number of packets in the buffer for 0.5 < ρ < 1.5, step size 0.1. The number of packets in the buffer increases as the ρ increases because larger ρ value means larger service time required to serve each packet.

This graph shows the average delay time of packets in the buffer for 0.5 < ρ < 1.5, step size 0.1. The average delay time of packets in the buffer increased as the ρ increases because larger ρ value means larger service time required to serve each packet. Because more time was require for packets to leave the server at higher ρ value, packets had to wait for longer time to be served. This result agrees with the results displayed in the graph which shows the number of packets in the buffer.

This graph shows the percentage of packets lost for 0.5 < ρ < 1.5, with step size 0.1. As mentioned before, larger ρ means more server time is required for packets to be served, which implies that the average size of the buffer goes up. Due to this effect, the buffer will be filled up more often at higher ρ values, and higher numbers of packets were lost at higher ρ values.

This graph shows the average idle time as a percent of total runtime for 0.5 < ρ < 1.5, with step size 0.1. In this graph it can be observed that the proportion of run time that the server was idle for is relatively high at low ρ values while the idle time eventually decreased to zero for all K values. This is because since there are more packets in the buffer at higher ρ values, there are more packets waiting to be served by the server, hence there is always a packet being pushed into the server.  
  
When ρ was increased to 5, the PLoss rate was increased to 80%. This is because the time required to serve an individual packet is very long, and the buffer gets full very quickly. Because the buffer is full for most of the execution time, majority of arriving packets were dropped.