Simulation of M/M/2/2+5 Queueing System

Jiaqi Yan, Ping Liu
CS555 Project
supervised by Edward Chlebus

November 23, 2015

Abstract

In this project, we simulate and analyze the M/M/2/2+5 queueing system. To generate generate packet arriving time and processing time, we use Python's built-in *random* module, which is validated carefully. Then we apply the **Welch graphical procedure** to eliminate the warm-up period in the simulation. With the stationary region, we then analyze the system's properties such as blocking probability and mean number of packet in the system. The 90% confidence interval for these properties are also given.

Contents

1 Discrete Event Simulation

We designed and implemented our own simple simulator for M/M/2/2+5 queueing system. The simulation workflow is shown in Algorithm 1. The whole process is embedded inside a while loop.

The while loop will keep consume event in the *event_list* until we handled all the packets. Whenever there is unseen packets, we create an arrival event and insert it to the list. This list thus should be maintained in order with respect to events' time stamp, which is determined as soon as we create a `event` object. Also based on this time stamp, we will pop up the next event we should handle from the list according to the event type, either *arrival* or *departure*.

If the type is *departure*, we first check number of arrived but waiting packets in the system. When no packets need to be served at this moment, set the server this packet is leaving from to *idle*; otherwise, just decrease this *waiting* counter. Anyway, we just served a packet. This means the counter for served packet should increase. We can also calculate the entire time this packet is in the system, from it entering until its exit, e.g. current clock.

For *arrival* event, we have more cases to deal with. Firstly, the system's queueing buffer may be already full when this arrival event happens; this means the arriving packet is dropped and we just need to increase the counter *pkt_dropped*. Notice that for this kind of arrival event, no corresponding *departure* event needs to be scheduled. However, as long as any system servers is available or the buffer can hold more packets, we need to properly create a corresponding *departure* event and put it into the global event list. The former case is easy to handle: we just find an available server, mark its status as *busy* and record server ID to the event. The server ID will be used to unmark the *busy* status when we handling this *departure* event. The time stamp of this newly created *departure* event can be calculated by increasing the current clock time by the time it ought to be served, an exponential random value generated by our random generator.

When all servers are busy, we need to calculate the server who will serve the packet in the future and the time stamp at the end of the service differently. Algorithm 2 shown the critical part of this task: find a server that will be available first. Then the newly created departure event should be served from this *earlier_time_stamp* and on this first available server. Therefore its time stamp should be this this returned time stamp plus the duration it will be served in the server.

2 Random Generator

2.1 Validation

In M/M/2/2+5 queueing system, the number of packet arriving in a fixed interval follows **Poisson Distribution** and the service time for each packet follows **Expo-**

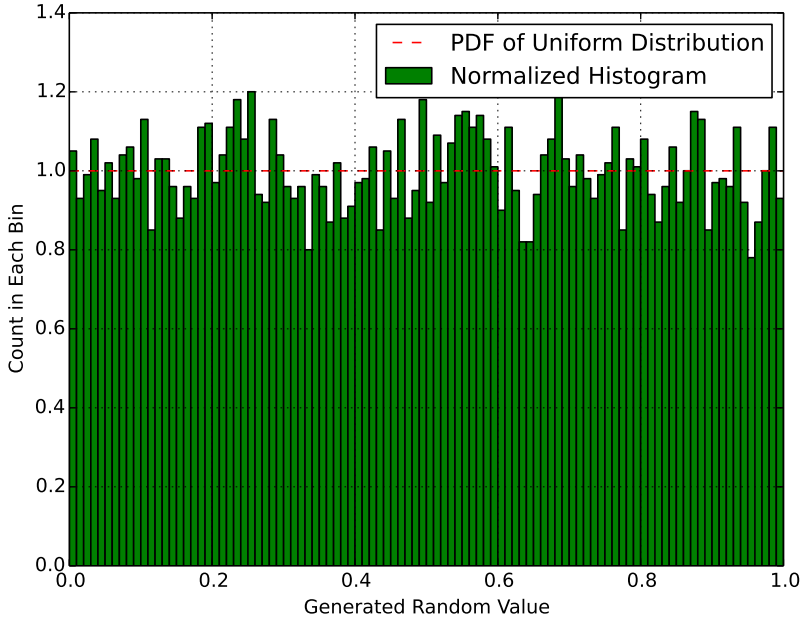

Figure 1: Histogram of Uniform Random Values Generated by Numpy

nential Distribution. These input data is generated by *Numpy*'s *random* module. Before running the simulator, it is important to test the wellness of this random generator.

We evaluate Numpy's random generator in two ways. First we generate random values follows uniform distribution and then plot their histogram. As shown in Figure 1, the number of random values falling into each interval is close to each other. This means that the generated values are very close to uniformly distributed. Besides, we compare the normalized histogram to the 'best fit' curve of both uniform distribution and normal distribution in Figure 1 and 2. From both figures we can say that the random generator generated random values of user-specified distribution.

To generate different random value sequences, we feed the random generator a seed, which is the integral value of system current time. That is the number of seconds since **Unix Epoch**. We generated two sequences of 1000000 numbers each and use **t-test** to test statistically are they significantly different from each other.

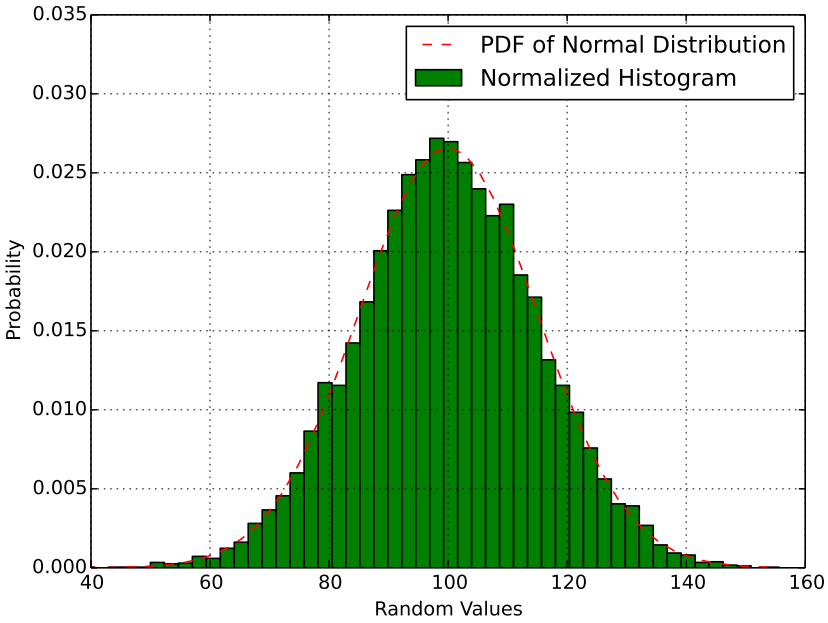

Figure 2: Comparison to Best Fit Curve for Normal Distribution

3 Eliminate Warm-up Period

4 System Properties at Stationary State

4.1 Blocking Probability

4.2 Mean Spending Time

4.3 Mean Number of Packet

Algorithm 1 Core of Discrete Event Simulation

```
1: function SIMULATION_CORE(arrive_time_seq, serve_time_seq)
2:     let arrive_time_seq denote the arriving time of each packet
3:     let serve_time_seq denote the service time of each packet
4:     let N denote the number of total packets
5:     while pkt_served + pkt_dropped < N do
6:         if pkt_seen < N then                ▷ Insert new arrival event to event list
7:             ts ← arrive_time_seq[pkt_seen]
8:             Create new arrival event evt with time stamp ts and id pkt_seen
9:             Insert evt to event_list
10:        end if
11:        Sort event_list based on events' time stamp
12:        Pop up the next event evt_x we need to handle in event_list
13:        clock ← evt_x.time_stamp
14:        if evt_x is departure event then
15:            if queue buffer is empty then
16:                Set the status of the server evt_x is leaving to idle
17:            else
18:                −− waiting
19:            end if
20:            ++ pkt_served
21:            evt_x.exit_time ← clock
22:            spending_time ← evt_x.exit_time − evt_x.enter_time
23:            Record the spending time of packet of event evt_x
24:        end if
25:        if evt_x is arrival event then
26:            if queue buffer is full then
27:                ++ pkt_dropped
28:            else
29:                if There is available server then
30:                    id ← evt_x.pkt_id
31:                    ts ← clock + serve_time_seq[id]
32:                    Choose an available server s
33:                    Mark s as busy
34:                else
35:                    (ts, s) ←SCHEDULE_DEPARTURE( )
36:                    ++ waiting
37:                end if
38:                Create new departure event evt with time stamp ts
39:                evt.enter_time ← clock
40:                evt.depart_srv ← s
41:                Insert evt to event_list
42:            end if
43:        end if
44:    end while
45: end function
```

Algorithm 2 Schedule Departure Event When Server is Unavailable but Buffer is Not Full

```
1: function SCHEDULE_DEPARTURE
2:     Sort event_list
3:     for each server s in the system do
4:         Find the last departure event evt scheduled on this server s
5:         earlier_time_stamp = MIN(earlier_time_stamp, evt.time_stamp)
6:     end for
7:     Return earlier_time_stamp and the corresponding server_id
8: end function
```