Simulating M/M/2/2+5 Queueing System

Jiaqi Yan, Ping Liu
CS555 Project
supervised by Edward Chlebus

December 5, 2015

Abstract

In this project, we simulate and analyze the **M/M/2/2+5 queueing system**. To generate packet arriving time and service 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 intervals** 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 this arrived packet should be served at this *earlier_time_stamp* time point and on this first available server. Therefore its departure time stamp should be this returned time stamp plus the duration it will be served in the server.

2 Random Generator 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 **Exponential 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.

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:             $ts \leftarrow arrive\_time\_seq[pkt\_seen]$
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:        $clock \leftarrow evt_x.time\_stamp$
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:            $evt_x.exit\_time \leftarrow clock$
22:            $spending\_time \leftarrow evt_x.exit\_time - evt_x.enter\_time$
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:                    $id \leftarrow evt_x.pkt\_id$
31:                    $ts \leftarrow clock+$ exponential service time
32:                    Choose an available server $s$
33:                    Mark $s$ as $busy$
34:                else
35:                    $(ts, s) \leftarrow$SCHEDULE_DEPARTURE( )
36:                    $ts \leftarrow ts+$ exponential service time
37:                    $++waiting$
38:                end if
39:                Create new departure event $evt$ with time stamp $ts$
40:                $evt.enter\_time \leftarrow clock$
41:                $evt.depart\_srv \leftarrow s$
42:                Insert $evt$ to $event\_list$
43:            end if
44:        end if
45:    end while
46: 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
```

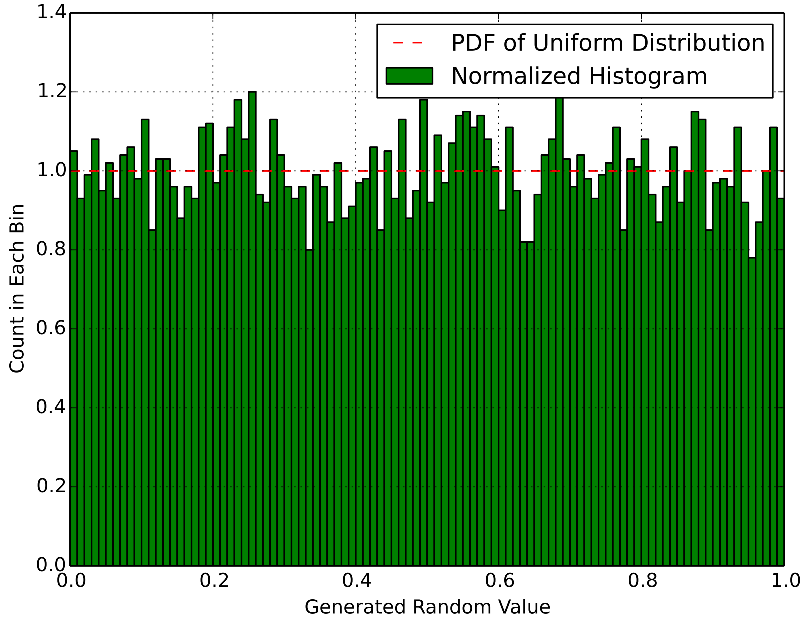


Figure 1: Histogram of Uniform Random Values Generated by Numpy

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**.

To demonstrate that different seed indeed generates different random value sequences, we generated two sequences of 1000000 numbers with 2 distinct seed. Then we combine them into a **set** object, which only hold distinct elements in Python.

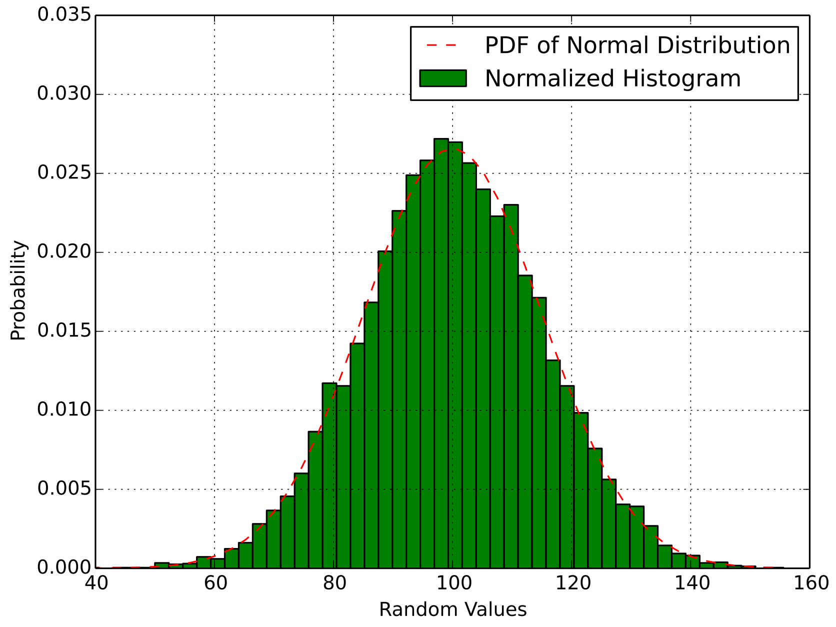


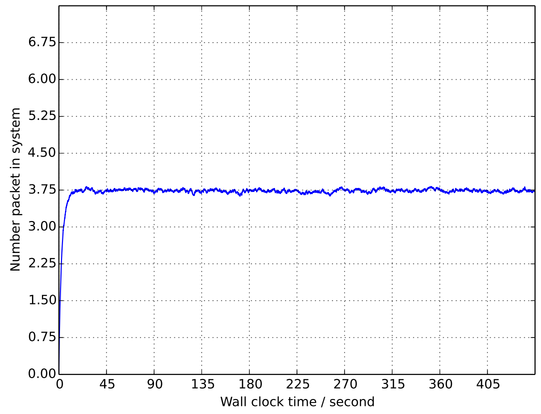
Figure 2: Comparison to Best Fit Curve for Normal Distribution

The number of different random values in 2 random sequence, of course, is the length of the set. Experiment result shows that the set's size is 2000000. This tell us that two random sequences are different.

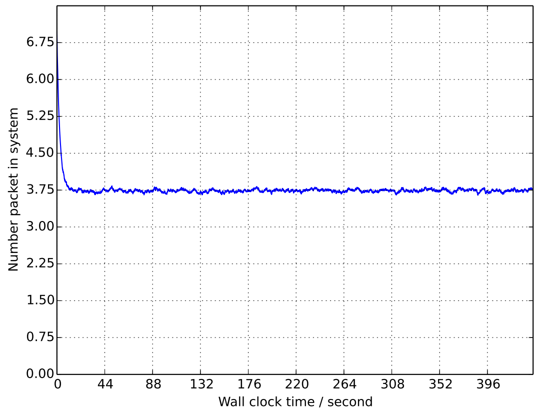
3 Eliminate Warm-up Period

Using Algorithm 1 as core, we run **5000 simulations** for system A and system B under different initial states respectively. For each simulation run, we **inject 1000 packets** into the system; the simulation terminates when these 1000 packets are all handled. We choose to observe number of packets in the system during simulation. Every time this system variable changes, we record its value and the happenning time moment. After simulation, we process this log as follows. First we create a observation time sequence with interval 1 magnitude less than arrival rate and ends at the finishing moment of the last departure event. This ensure that we will not lost any accuracy. For example, for system B where $\lambda = 10$, the observation interval is 0.01. Then based on the simulation log, we calculate the value of number of packet at every observation time point. In this way we get the output for one simulation run.

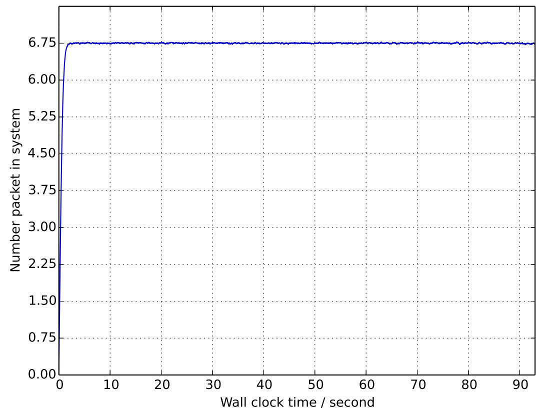
We process all 5000 simulation logs and then can obtain 5000 simulation outputs. Since simulation ends at different time, we only keep m observations where $m = MIN$(length of all outputs). Based on Welch graphical procedure, we average 5000 simulation outputs, observation by observation. There are totally 4 cases to investigate: system A with 0 initial packets, system A with 7 initial packets, system



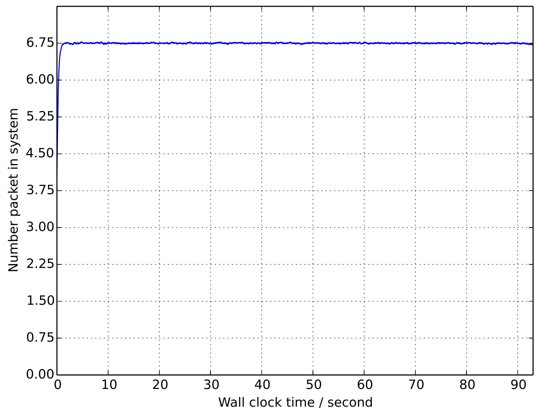
(a) **System A with 0 Initial Packets**



(b) **System A with 7 Initial Packets**



(c) **System B with 0 Initial Packets**



(d) **System B with 4 Initial Packets**

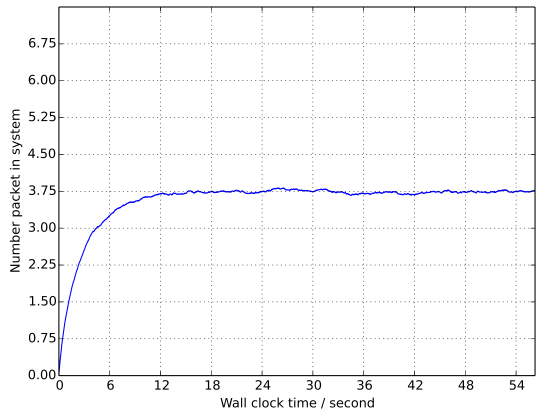
Figure 3: **Average 5000 Simulation Runs**

B with 0 initial packets, system B with 4 initial packets. All systems' Welch processing results are plotted in Figure 3a, Figure 3b, Figure 3c and Figure 3d respectively. Since the nonstationary states only take up a very smal fraction of the entire output, we further zoom out the them in Figure 4a, Figure 4b, Figure 4c and Figure 4d.

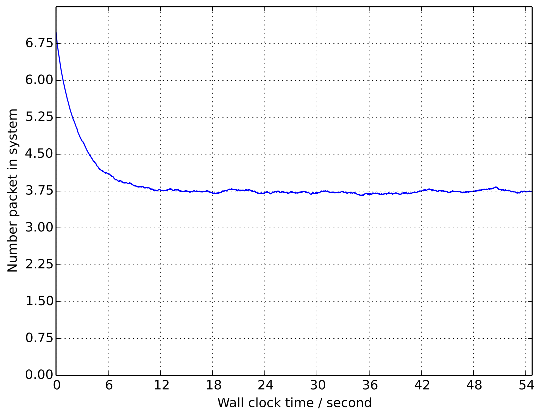
As we can see from both the figures for entire simulation output and the figures for only nonstationary output, system A as well as system B will enter their respective stable state **regardless of the inital state**. For system A, no matter we started from full buffer or empty buffer with empty server, the number of packet in the stable state is around **3.75**. For system B, the number of packet in system will increase from different initial states and end up with close to a value of **6.75**.

4 System Properties at Stationary State

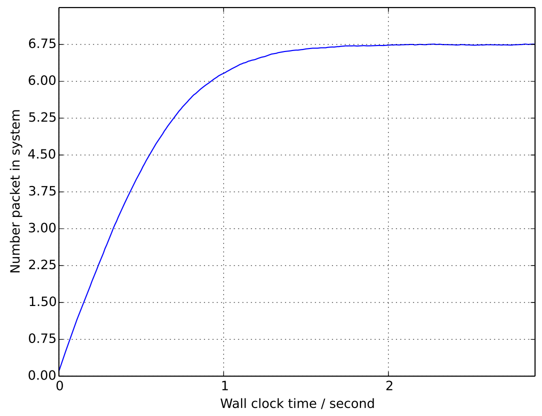
In the last section, we have applied Welch graphical procedure and show the average simulation results of 5000 runs for each system with different initial state. These



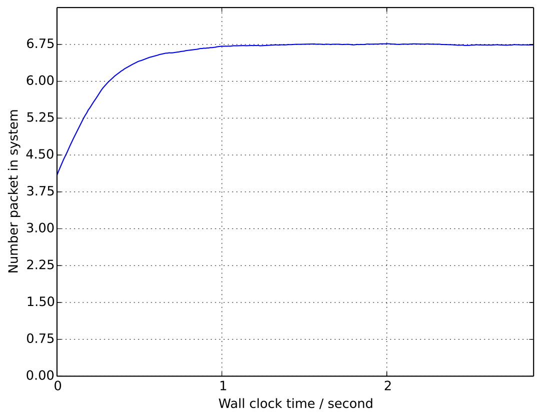
(a) **System A with 0 Initial Packets**



(b) **System A with 7 Initial Packets**



(c) **System B with 0 Initial Packets**



(d) **System B with 4 Initial Packets**

Figure 4: **Zoom out Nonstationary States**

figures clearly identifies the nonstationary and stationary states. For system A, the time threshold for stable state is around 45 second in real time. For system B, the time threshold for stable state is somewhere before 10 second. However, we can pick a much larger time threshold when analysis system properies at the stationary state. In the following statistic analysis, we treat the **first half** of the outputs as warmup peroid and just ignore them when calculating system properties.

For every simulation run, we first eliminate the data outputs belonging to warmup peroid, then we can calculate:

- **Blocking Probability** P_B for a particular simulation run is defined by the ratio of number of dropped packets to the number of arrived packets:

$$P_B = \frac{\textit{number of packets dropped}}{\textit{number of packets arrived}} \tag{1}$$

- **Mean Spending Time** S, assuming there are totally n served packet over time, is computed by:

$$S_s = \frac{\sum_{i=1}^{n} S_i}{n} \tag{2}$$

  Here S_i is the duration that packet i spent in the system. As shown in Algorithm 1, each accepted packet's spending time is recorded. Note that dropped packets are not counted in Equation 2

- **Mean Number of Packets** $E(N)$ of one run is calculated by:

$$N_s = \sum_{i=0}^{7} \frac{i \times T_i}{T} \tag{3}$$

  In Equation 3, T_i is defined as the summation of durations that there are i packets in the system. i can only range from 0 to 7, the capacity of our system. To get T_i, we need log more information in simulation: the moment t_i that the number of packets in system changes to i. When simulation finishes, we can calculate the summation of durations that there are i packets in the system. T is the entire duration of the simulated scenario, or the summation of all durations T_i.

With $N = 5000$ independent simulations, we can further get the confidence interval for these calculated system properties. The general formation of confidence interval for 90% confidence level is

$$\bar{X} \pm z_{1-\frac{\alpha}{2}} \sqrt{\frac{S^2}{N}} \tag{4}$$

where $z_{1-\frac{\alpha}{2}} = z_{0.95} = 1.645$ because $\alpha = 0.1$.

Table 1: **Confidence Interval for P_B, S_s and N_s**

	A with $x_{t=0} = 0$	A with $x_{t=0} = 7$	B with $x_{t=0} = 0$	B with $x_{t=0} = 4$
P_b	0.13264±0.00066	0.13318±0.00067	0.79902±0.00045	0.79958±0.00044
S_s	2.15510±0.00816	2.15168±0.00827	3.35867±0.01221	3.37481±0.01219
N_s	3.73037±0.01285	3.73350±0.01309	6.73854±0.00211	6.74043±0.00204

To use Equation 4, for example, to find the confidence level of mean number of packets in the system, we first need the estimator of the mean of N_s, which we treat as a random variable with unknown distribution:

$$\overline{E(N_s)} = \frac{1}{N} \sum_{1}^{N} N_s[i] \tag{5}$$

where $N_s[i]$ is the mean number of packets in system for the i^{th} simulation run and $N = 5000$. Then we need the estimator of the variance of N_s:

$$\overline{Var(N_s)} = \frac{\sum_1^N (N_s[i] - \overline{E(N_s)})^2}{N-1} \tag{6}$$

Based on Equation 4, we have the confidence interval for the mean number of packets in the system:

$$\overline{E(N_s)} \pm \sqrt{\frac{\overline{Var(N_s)}}{N}} \tag{7}$$

Table 1 summarizes the confidence intervals for blocking probability, mean time spending in the system and mean number of packets in the system regarding to both system A and B with different initial state.