

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

Jiaqi Yan, Ping Liu  
CS555 Project  
supervised by Edward Chlebus

December 8, 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	3
2	Random Generator Validation	3
3	Eliminate Warm-up Period	6
4	System Properties at Stationary State	7

# 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 ← 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 evtx we need to handle in event_list
13:    clock ← evtx.time_stamp
14:    if evtx is departure event then
15:      if queue buffer is empty then
16:        Set the status of the server evtx is leaving to idle
17:      else
18:        – – waiting
19:      end if
20:      ++ pkt_served
21:      evtx.exit_time ← clock
22:      spending_time ← evtx.exit_time – evtx.enter_time
23:      Record the spending time of packet of event evtx
24:    end if
25:    if evtx 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 ← evtx.pkt_id
31:          ts ← clock + exponential service time
32:          Choose an available server s
33:          Mark s as busy
34:        else
35:          (ts, s) ← SCHEDULE_DEPARTURE( )
36:          ts ← ts + exponential service time
37:          ++ waiting
38:        end if
39:        Create new departure event evt with time stamp ts
40:        evt.enter_time ← clock
41:        evt.depart_srv ← s
42:        Insert evt to event_list
43:      end if
44:    end if 4
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
```

---

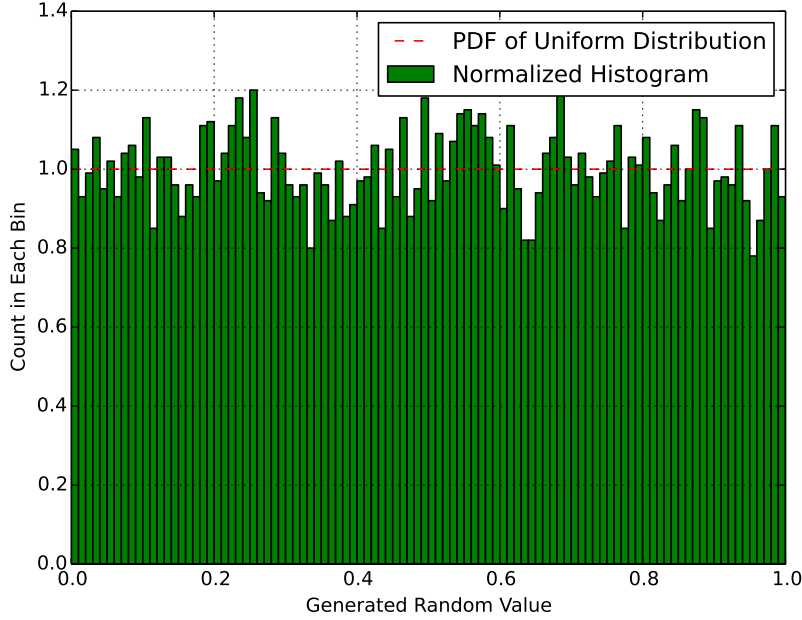


Figure 1: Histogram of Uniform Random Values Generated by Numpy

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.

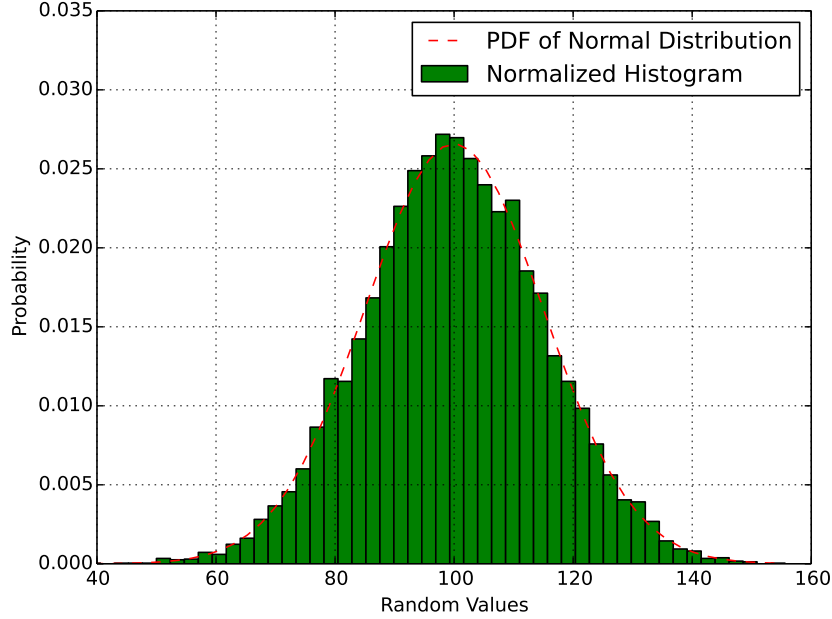


Figure 2: Comparison to Best Fit Curve for Normal Distribution

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 happening 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  $m = MIN(\text{length of all outputs})$ . 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

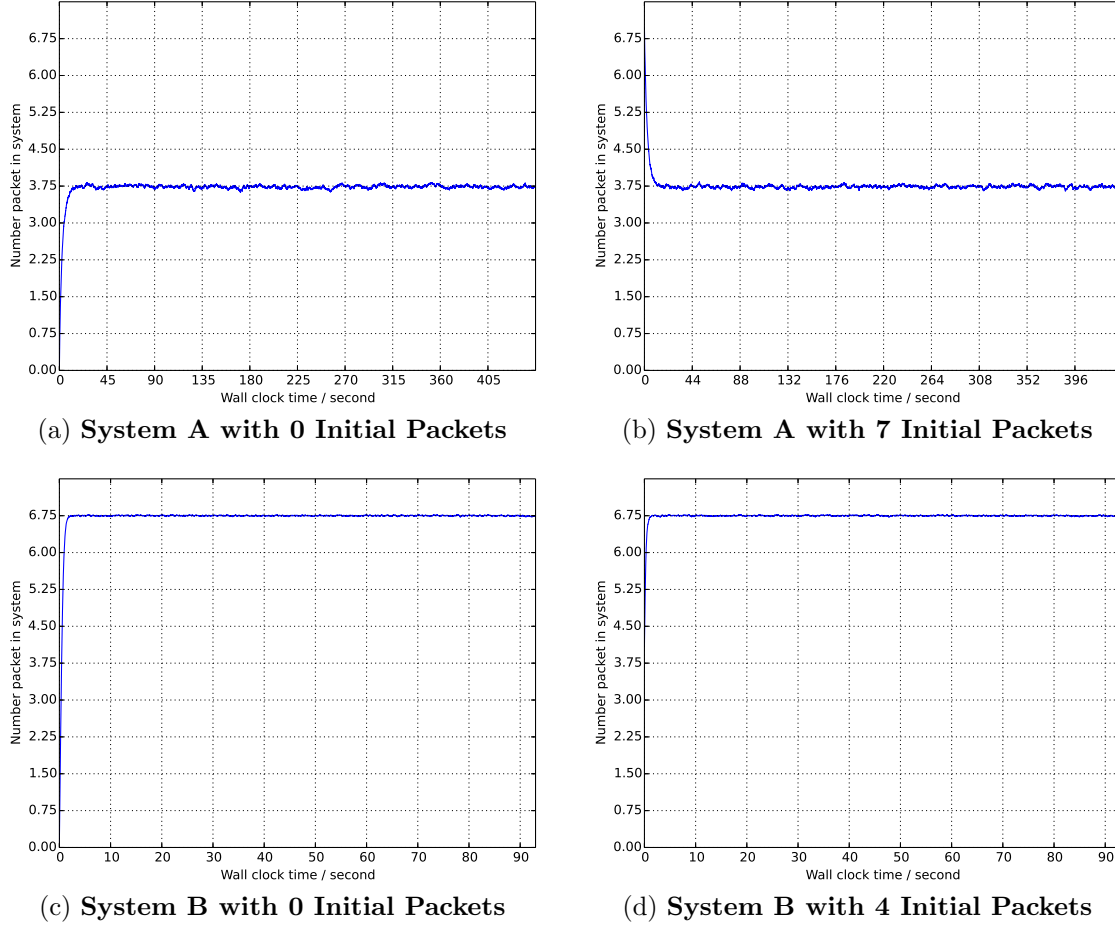


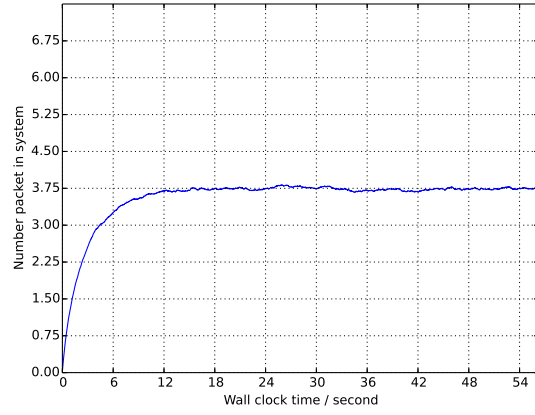
Figure 3: Average 5000 Simulation Runs

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 small fraction of the entire output, we further zoom out them in Figure 4a, Figure 4b, Figure 4c and Figure 4d.

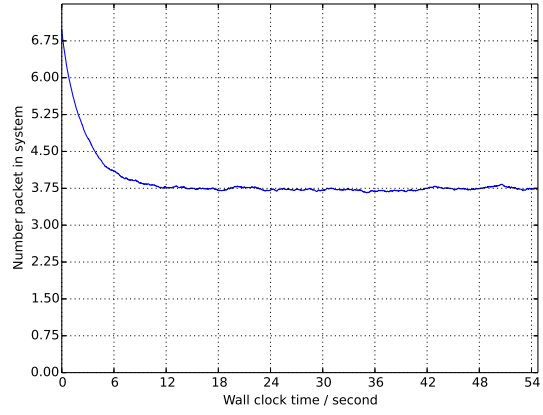
As we can see from both the figures for entire simulation output and the figures for only nonstationary output, system A as well as system B will enter their respective stable state **regardless of the initial state**. For system A, no matter we started from full buffer or empty buffer with empty server, 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

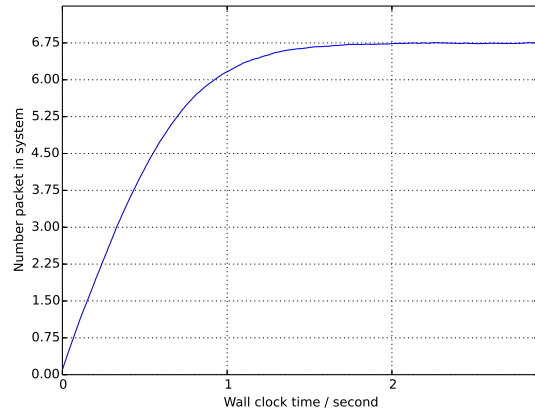
In the last section, we have applied Welch graphical procedure and show the average simulation results of 5000 runs for each system with different initial state. These



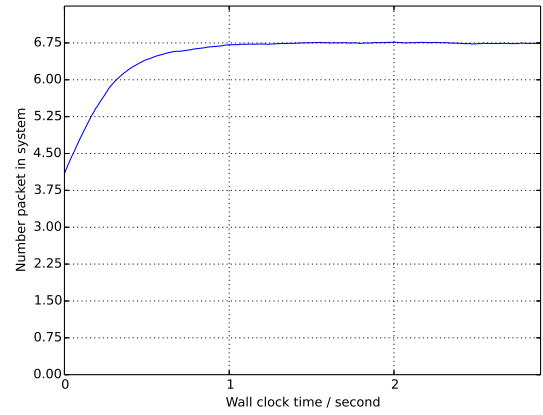
(a) System A with 0 Initial Packets



(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 properties 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:

$$P_B = \frac{\text{number of packets dropped}}{\text{number of packets arrived}} \quad (1)$$

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

$$S_s = \frac{\sum_{i=1}^n S_i}{n} \quad (2)$$

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:

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

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

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

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:

$$\overline{E(N_s)} = \frac{1}{N} \sum_1^N N_s[i] \quad (5)$$

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  $N_s$ :

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

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

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

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.

Listing 1: Source Code for Event: ../mmrng.py

```

1 __author__ = 'Ping'
2 import numpy as np
3
4 def generate_exp(length, mu, seed):
5     """Generate exponential random sequence with seed"""
6     randgen = np.random.RandomState(seed)
7     return randgen.exponential(scale=mu, size=length)
8
9 def generate_exp_single(mu, seed):
10    """Generate one exponential random value with seed"""
11    randgen = np.random.RandomState(seed)
12    random_values = randgen.exponential(scale=mu, size=1)
13    return random_values[0]

```

Listing 2: Source Code for Event: ../mmevent.py

```

1 #!/usr/bin/python
2
3 class MMEvent(object):
4     """Abstraction of packet arrival or departure"""
5     def __init__(self, pkt_id, name, ts):
6         """Create MMEvent object
7
8         Attributes:
9             pkt_id: which packet this event is about
10            event_type(str): 'arrival' or 'departure'
11            time_stamp: when should we handle this event;
12                we need this value to put event into event_list
13            enter_time: the moment it enter the system
14            exit_time: the moment it exit the system
15        """
16        super(MMEvent, self).__init__()
17        self.pkt_id = pkt_id
18        self.evt_name = name
19        self.time_stamp = ts
20        self.enter_time = 0
21        self.exit_time = 0
22        self.depart_srv = None

```

Listing 3: Source Code for Event: ../mmsystem.py

```

1 #!/usr/bin/python
2
3 class MMSystem(object):
4     """Various system states and counters for M/M/2/2+5 system
5
6     Attributes:
7         num_srv: number of servers in this system
8         capacity: queue size of the system, excluding buffer on server
9         pkt_seen: number of packets arrived so far
10        pkt_dropped: number of packets dropped so far
11        pkt_waiting: number of packets in the system queue
12        pkt_served: number of packets exited
13        spending_time: how much each packet is spending in the system
14        log_time: starting time stamp that pkt in system is changed
15        log_num_pkt_inside: history of number of packets in system
16        stable: if we passed warm-up period
17        srv_status: idle or busy for a particular server
18    """
19    def __init__(self, num_srv, capacity):
20        super(MMSystem, self).__init__()

```

```

21     # immutable properties
22     self.num_srv = num_srv
23     self.capacity = capacity
24
25     # counters
26     self.pkt_dropped = 0      # dropped due to buffer full
27     self.pkt_dropped_id = []
28     self.pkt_seen = 0        # total arrived pkt
29     self.pkt_waiting = 0     # pkt in buffer
30     self.pkt_served = 0      # pkt departed from servers
31
32     # result log info
33     self.spending_time = {}
34     self.log_time = []
35     self.log_num_pkt_inside = []
36
37     # state variables
38     self.stable = False
39     self.srv_status = {}
40     for i in range(0, self.num_srv):
41         self.srv_status[i] = 'idle'
42
43 def full(self):
44     """Test if the queue is full"""
45     return self.pkt_waiting >= self.capacity
46
47 def available(self):
48     """Test if any server is idle"""
49     return 'idle' in self.srv_status.values()
50
51 def num_available_servers(self):
52     """Return number of available servers"""
53     counter = 0
54     for key, value in self.srv_status.items():
55         if value == 'idle':
56             counter += 1
57     return counter
58
59 def available_server(self):
60     """Find any 'idle' server in system"""
61     for key, value in self.srv_status.items():
62         if value == 'idle':
63             return key
64
65 def dump_num_pkt_inside(self, time):
66     """Update pkt inside system and log it with current time stamp"""

```

```

67         self.log_time.append(time)
68         if self.pkt_waiting > 0:
69             num_pkt_inside = self.pkt_waiting + 2
70         else:
71             num_pkt_inside = 2 - self.num_available_servers()
72
73         self.log_num_pkt_inside.append(num_pkt_inside)
74
75     def dump_pkt_spending_time(self, evt):
76         """Calculate the duration a pkt spent in the system"""
77         self.spending_time[evt.pkt_id] = evt.exit_time - evt.enter_time

```

Listing 4: Source Code for Event: ../mmwelch.py

```

1  #!/usr/bin/python
2
3  import numpy as np
4  from matplotlib import pyplot as plt
5  import math
6
7  class MMWelch(object):
8      """Welch graphic method to eliminate warm-up period"""
9      def __init__(self, num_replicas, interval, run_length, prefix, mode='online'):
10         """Create object with
11
12         Attributes:
13             num_replicas: to average these number of replicas
14             run_length: the min length over all replications
15             interval & time_seq: to draw simulation time
16             prefix: name for output .txt and .eps files
17         """
18         super(MMWelch, self).__init__()
19         self.num_replicas = float(num_replicas)
20         self.prefix = prefix
21         self.mode = mode
22         self.warmup = float(warmup)
23         self.interval = float(interval)
24
25         if self.mode == 'online':
26             self.run_length = run_length
27             self.time_seq = [self.interval * i for i in range(0, self.run_length)]
28             self.avg_run = [0] * self.run_length
29         if self.mode == 'offline':
30             load_file_name = self.prefix + 'Avg%d.txt' % self.num_replicas
31             self.avg_run = np.loadtxt(load_file_name)
32             warmup_period = len(self.avg_run) / self.warmup

```

```

33         self.avg_run = self.avg_run[:warmup_period]
34         self.run_length = len(self.avg_run)
35         self.time_seq = [self.interval * i for i in range(0, self.run_length)]
36         self.prefix += '0f1'
37         print "Use %d as common length " % self.run_length
38
39     def average_all_runs(self):
40         """Average all replica runs, store in self.avg_run"""
41         for i in range(0, int(self.num_replicas)):
42             # truncate only the common part of each runs
43             load_file_name = '%sRun%d.txt' % (self.prefix, i)
44             replica = np.loadtxt(load_file_name, dtype=int, comments='#')
45             replica = replica[:self.run_length]
46             self.avg_run = np.add(replica, self.avg_run)
47         # average and save
48         self.avg_run = [float(x) / self.num_replicas for x in self.avg_run]
49         file_name = self.prefix + 'Avg%d.txt' % self.num_replicas
50         np.savetxt(file_name, self.avg_run)
51
52     def plot_avg_run(self):
53         """Draw a figure, output to file"""
54         #y_max = math.ceil(max(self.avg_run)) + 0.5
55         y_max = 7.5 # empiriall value
56         x_max = math.ceil(self.time_seq[-1] / 10) * 10
57
58         figure_name = self.prefix + 'Avg%d.eps' % self.num_replicas
59         plt.figure()
60         plt.plot(self.time_seq, self.avg_run)
61         plt.xlabel('Wall clock time / second')
62         plt.ylabel('Number packet in system')
63         plt.xticks(np.arange(0, x_max, x_max / 10.0))
64         plt.yticks(np.arange(0, y_max, y_max / 10.0))
65         plt.xlim(self.time_seq[0], self.time_seq[-1])
66         plt.ylim(0, y_max)
67         plt.grid()
68         plt.savefig(figure_name, format='eps')

```

Listing 5: Source Code for Event: ../mmreporter.py

```

1 #!/usr/bin/python
2
3 import bisect
4
5 class MMReporter(object):
6     """Calculate system measurements"""
7     def __init__(self, system, ats):

```

```

8         super(MMReporter, self).__init__()
9         self.system = system
10        self.ats = ats
11        self.obsrv_pkt = []
12
13    def blocking_prob(self):
14        """Blocking probability"""
15        num_dropped = 0
16        num_seen = 0
17
18        for i in range(self.system.pkt_seen):
19            if self.ats[i] > (self.system.log_time[-1] / 3.0):
20                if i in self.system.pkt_dropped_id:
21                    num_dropped += 1
22                num_seen += 1
23
24        return num_dropped/float(num_seen)
25        #return float(self.system.pkt_dropped) / float(self.system.pkt_seen)
26
27    def mean_time_spending_in_system(self):
28        """Mean time pkt spending in the system"""
29        dur_sum = 0.0
30        spending_time_seq = self.system.spending_time.values()
31        start = int(len(spending_time_seq) / 2.0)
32        end = int(len(spending_time_seq) / 3.0 * 2)
33        for i in range(start, end):
34            dur_sum += spending_time_seq[i]
35        return dur_sum / (end - start)
36
37    def mean_num_pkt_in_system(self):
38        """Mean number of pkt in the system"""
39        num_pkt_duration = {}
40        entire_duration = 0.0
41        product_sum = 0.0
42
43        start = int(len(self.system.log_time) / 2.0) - 1
44        end = int(len(self.system.log_time) / 3.0 * 2) - 1
45        for i in range(start, end):
46            dur = self.system.log_time[i+1] - self.system.log_time[i]
47            num_pkt = self.system.log_num_pkt_inside[i]
48            if num_pkt in num_pkt_duration.keys():
49                num_pkt_duration[num_pkt] += dur
50            else:
51                num_pkt_duration[num_pkt] = dur
52            entire_duration += dur
53

```

```

54         for num_pkt, dur in num_pkt_duration.items():
55             product_sum += num_pkt * dur
56         return product_sum / entire_duration
57
58     def warm_up_finding(self, interval_sequence):
59         """Find #pkt for all moment in interval sequence"""
60
61         for i in interval_sequence:
62             index = bisect.bisect_left(self.system.log_time, i)
63             index = index - 1
64             self.observ_pkt.append(self.system.log_num_pkt_inside[index])
65
66         return self.observ_pkt

```

Listing 6: Source Code for Event: ../mmsimulator.py

```

1  #!/usr/bin/python
2
3  from mmevent import MMEvent
4
5  class MMSimulator(object):
6      """Simulator for M/M/2/7 system"""
7      def __init__(self, system, end_time):
8          super(MMSimulator, self).__init__()
9          self.system = system
10         self.end_time = end_time
11         self.clock = 0
12         self.initialized = False
13
14     def init_simulation(self, num_pkt_init):
15         """Initialize simulator before its core"""
16         self.init_system_status(num_pkt_init)
17         self.init_event_list()
18         self.initialized = True
19
20     def init_system_status(self, num_pkt_init):
21         """Set system initial status"""
22         self.system.log_time.append(0)
23         self.system.log_num_pkt_inside.append(num_pkt_init)
24         # initial number of pkt may large than system capacity
25         # num_busy_srv = num_pkt_init - self.system.capacity
26         # if num_busy_srv > 0 and num_busy_srv <= self.system.num_srv:
27             # for i in range(0, num_busy_srv):
28                 # self.system.srv_status[i] = 'busy'
29
30     def init_event_list(self):

```



```

31         """Do necessary initialization"""
32         self.event_list = []
33
34     def sort_event_list(self):
35         """Sort event list on time stamp of every events"""
36         self.event_list.sort(key=lambda event: event.time_stamp, reverse=False)
37
38     def last_departure_srv(self, srv_id):
39         """Search in event list the last depart event
40         with specified depart server id"""
41         self.sort_event_list()
42         for event in reversed(self.event_list):
43             if event.evt_name == 'departure' and event.depart_srv == srv_id:
44                 return event
45
46     def schedule_departure(self):
47         """Find server for new depart event with the depart time stamp"""
48         earliest_ts = None
49         self.sort_event_list()
50         for s_id in self.system.srv_status.keys():
51             evt = self.last_departure_srv(s_id)
52             if earliest_ts == None or earliest_ts > evt.time_stamp:
53                 earliest_ts = evt.time_stamp
54                 earliest_srv_id = s_id
55         return (earliest_ts, earliest_srv_id)
56
57     def next_event(self):
58         """Pop up the earliest event"""
59         self.sort_event_list()
60         return self.event_list.pop(0)
61
62     def should_continue(self, N):
63         """Test if seen all pkt and clock NOT exceed predefined end time"""
64         return self.system.pkt_served + self.system.pkt_dropped < N and self.cl
65
66     def simulate_core(self, arrive_time_seq, depart_time_seq_server1, depart_ti
67         """Discrete event simulation"""
68         if not self.initialized:
69             print "Simulator is not explicitly initialized"
70
71         N = len(arrive_time_seq)
72         flag_server1 = 0
73         flag_server2 = 0
74
75         while self.should_continue(N):
76             # schedule/add a new pkt arrive event if still pkt unseen

```

77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122

```
if self.system.pkt_seen < N:
    new_arrival_ts = arrive_time_seq[self.system.pkt_seen]
    new_arrive = MMEvent(self.system.pkt_seen, 'arrival', new_arrive)
    self.event_list.append(new_arrive)
    self.system.pkt_seen += 1
    # if self.clock > float(self.end_time/3):
    #     self.system.pkt_seen_after_warm_up += 1

# pop up the next event
evt_x = self.next_event()
# advance simulation clock
self.clock = evt_x.time_stamp

if evt_x.evt_name == 'departure':
    # set the serving server to 'idle'
    # increase @pkt_served counter
    # calculate how long this pkt spend in @system
    if self.system.pkt_waiting == 0:
        self.system.srv_status[evt_x.depart_srv] = 'idle'
    else:
        self.system.pkt_waiting -= 1

    evt_x.exit_time = self.clock
    # either server became idle or waiting pkt decreased
    # log num_pkt_inside the system
    self.system.dump_num_pkt_inside(self.clock)
    # calculate the spending time of this packet
    self.system.dump_pkt_spending_time(evt_x)
    self.system.pkt_served += 1

if evt_x.evt_name == 'arrival':
    if self.system.full():
        # just drop pkt and increase counter
        self.system.pkt_dropped += 1
        self.system.pkt_dropped_id.append(evt_x.pkt_id)
        # if self.clock > float(self.end_time/3):
        #     self.system.pkt_dropped_after_warm_up += 1

        # no departure event for this pkt is created
        # but need to count its spending time/ not to count
        evt_x.exit_time = evt_x.enter_time = 0
        # self.system.dump_pkt_spending_time(evt_x)
    else:
        if self.system.available():
            # put pkt into one available server
            # calculate when it should exit the server
```

```

123         # mark this server as 'busy'
124         new_depart_srv = self.system.available_server()
125         if new_depart_srv == 0:
126             new_depart_ts = self.clock + depart_time_seq_server
127             flag_server1 += 1
128         else:
129             new_depart_ts = self.clock + depart_time_seq_server
130             flag_server2 += 1
131
132         self.system.srv_status[new_depart_srv] = 'busy'
133     else:
134         # find the server pkt should go
135         earliest_ts, earliest_srv = self.schedule_departure()
136         if earliest_srv == 0:
137             new_depart_ts = earliest_ts + depart_time_seq_server
138             flag_server1 += 1
139         else:
140             new_depart_ts = earliest_ts + depart_time_seq_server
141             flag_server2 += 1
142
143         new_depart_srv = earliest_srv
144         self.system.pkt_waiting += 1
145
146         # either server became busy or waiting packet increases
147         # log num_pkt_inside the system
148         self.system.dump_num_pkt_inside(self.clock)
149         # actually insert new departure event
150         new_depart = MMEvent(evt_x.pkt_id, 'departure', new_depart_ts)
151         new_depart.enter_time = self.clock
152         new_depart.depart_srv = new_depart_srv
153         self.event_list.append(new_depart)

```

Listing 7: Source Code for Event: ../mmconfidence.py

```

1 import numpy as np
2
3 def get_confidence_level():
4
5     Z = 1.645
6     # Need to run the other 2 tonight, then put them in this list
7     prefix_list = ["Lmda2Init7", "Lmda2Init0", "Lmda10Init4", "Lmda10Init0"]
8     file_name = 'confidence_level.txt'
9     output = open(file_name, 'a')
10
11     for prefix in prefix_list:
12         file_blocking = prefix + "_blocking.txt"

```

```

13     file_spending = prefix + "_spending.txt"
14     file_num_customers = prefix + "_num_customers.txt"
15
16     output.write('#####\n')
17     output.write('##### System configuration %s #####\n' % prefix)
18     output.write('#####\n')
19
20     #Blocking Probability
21     blocking = np.loadtxt(open(file_blocking, 'rb'))
22     length_blocking = len(blocking)
23     mean_blocking = np.mean(blocking)
24     # This is sample variance for estimator
25     var_blocking = np.var(blocking) * length_blocking / (length_blocking-1)
26     error_blocking = Z * np.sqrt(var_blocking/len(blocking))
27     output.write('Mean blocking probability:      %.6f +- %.6f\n' % (mean_blocking, error_blocking))
28
29     #Mean Spending time
30     spending = np.loadtxt(open(file_spending, 'rb'))
31     length_spending = len(spending)
32     mean_spending = np.mean(spending)
33     # This is sample variance for estimator
34     var_spending = np.var(spending) * length_spending / (length_spending-1)
35     error_spending = Z * np.sqrt(var_spending/len(spending))
36     output.write('Mean spending time in system: %.6f +- %.6f\n' % (mean_spending, error_spending))
37
38     #mean number of customers
39     num_customers = np.loadtxt(open(file_num_customers, 'rb'))
40     length_num_customers = len(num_customers)
41     mean_num_customers = np.mean(num_customers)
42     # This is sample variance for estimator
43     var_num_customers = np.var(num_customers) * length_num_customers / (length_num_customers-1)
44     error_num_customers = Z * np.sqrt(var_num_customers/len(num_customers))
45     output.write('Mean # of packets in system:  %.6f +- %.6f\n' % (mean_num_customers, error_num_customers))
46     output.write('#####\n')
47     output.write('\n')

```

Listing 8: Source Code for Event: ../mmmain.py

```

1  #!/usr/bin/python
2
3  from mmrng import generate_exp
4  from mmsystem import MMSystem
5  from mmreporter import MMReporter
6  from mmsimulator import MMSimulator
7  from mmwelch import MMWelch
8  from mmconfidence import get_confidence_level

```

```

9 import time, math
10 import numpy as np
11
12 def roundup_hundreds(num):
13     """To round up ending time of the simulation"""
14     return int(math.ceil(num / 100)) * 100
15
16 def simulator_driver(trial, l, u, num_pkt_init, num_pkts, obsrv_int, seed, prefix):
17     """Reuse this function to get many replicas of simulation"""
18     # ending time is dependent with arriving rate.
19     end_time = 1000
20
21     mm27 = MMSystem(num_srv, num_buffer)
22
23     # arrival interval
24     ats = generate_exp(num_pkts, 1.0 / l, seed)
25     # arrival time stamp
26     ats = np.cumsum(ats)
27     # insert init pkt events
28     init_ats = np.array([0] * num_pkt_init)
29     ats = np.insert(ats, 0, init_ats)
30
31     # departure time stamp
32     dts_server1 = generate_exp(num_pkts + num_pkt_init, u, seed + 1)
33     dts_server2 = generate_exp(num_pkts + num_pkt_init, u, seed + 2)
34
35     simulator = MMSimulator(mm27, end_time)
36     simulator.init_simulation(num_pkt_init)
37
38     t0 = time.time()
39     simulator.simulate_core(ats, dts_server1, dts_server2)
40
41     reporter = MMReporter(mm27, ats)
42
43     duration = time.time() - t0
44     end_time = int(math.ceil(simulator.clock))
45     ots = [obsrv_int for _ in range(0, int(end_time / obsrv_int) + 1)]
46     ots = np.cumsum(ots) # observe time stamp
47
48     # output results to this file
49     file_name = prefix + 'Run%d.txt' % trial
50     # align results to observation intervals
51     observations = reporter.warm_up_finding(ots)
52     # output statistic results as header
53     optional_output = ''
54     optional_output += "#####\n"

```

```

55 optional_output += "##### Simulation results #####"
56 optional_output += "#####"
57 optional_output += "#Running time of NO.%d trial %.4fs\n" % (trial, duration)
58 optional_output += "#Ending event time stamp %.4f\n" % end_time
59 optional_output += "#Blocking probability %.4f\n" % reporter.blocking_prob()
60 optional_output += "#Mean time spent in system %.4f\n" % reporter.mean_time_spent()
61 optional_output += "#Mean #pkt in system %.4f\n" % reporter.mean_num_pkt_in_sys()
62 optional_output += "#####"
63 np.savetxt(file_name, observations, fmt='%i', header=optional_output, comments='')
64 # for main to get min common simulation length
65 return len(observations), reporter.blocking_prob(), reporter.mean_time_spent()
66
67 def eliminate_warmup_period(l, u, num_pkt_init, seed):
68     num_obsrv = 999999
69     num_trials = 5000
70     num_pkts = 1000
71     obsrv_int = min(0.01, 1.0 / l / 10)
72     blocking_list = []
73     spending_list = []
74     num_customers_list = []
75     prefix = 'Lmda%dInit%d' % (l, num_pkt_init)
76     for i in range(num_trials):
77         new_obsrv, blocking, spending, num_customers = simulator_driver(i, l, u, num_pkt_init, seed)
78         num_obsrv = min(num_obsrv, new_obsrv)
79         blocking_list.append(blocking)
80         spending_list.append(spending)
81         num_customers_list.append(num_customers)
82         seed += 10
83     file_block = prefix + "_blocking.txt"
84     file_spending = prefix + "_spending.txt"
85     file_num_customers = prefix + "_num_customers.txt"
86     #Save the history
87     np.savetxt(file_block, blocking_list, fmt = '%1.5f')
88     np.savetxt(file_spending, spending_list, fmt = '%1.5f')
89     np.savetxt(file_num_customers, num_customers_list, fmt = '%1.5f')
90
91     # welch = MMWelch(num_trials, obsrv_int, num_obsrv, prefix)
92     # welch.average_all_runs()
93     # Draw figure in offline mode
94     # warmup = 1 if l == lambdaA else 32
95     # welch = MMWelch(num_trials, obsrv_int, num_obsrv, prefix, 'offline', warmup)
96     # welch.plot_avg_run()
97
98 def run_system(l, u, num_pkt_init):
99     seed = int(time.time())
100     eliminate_warmup_period(l, u, num_pkt_init, seed)

```

```

101
102 def main():
103     run_system(lambdaA, u, 0)
104     run_system(lambdaA, u, 7)
105     run_system(lambdaB, u, 0)
106     run_system(lambdaB, u, 4)
107     get_confidence_level()
108
109 if __name__ == '__main__':
110     num_srv = 2
111     num_buffer = 5
112     lambdaA = 2.0
113     lambdaB = 10.0
114     u = 1.0
115
116     main()

```

Listing 9: Source Code for Event: ../valid\_random\_generator.py

```

1 #!/usr/bin/python
2
3 import numpy as np
4 from time import time
5 from scipy import stats
6 import matplotlib.pyplot as plt
7 import matplotlib.mlab as mlab
8
9 def rg_fitness():
10     mu, sigma = 100, 15
11     x = mu + sigma * np.random.randn(10000)
12
13     fig = plt.figure()
14     ax = fig.add_subplot(111)
15
16     # the histogram of the data
17     n, bins, patches = ax.hist(x, 50, normed=1, facecolor='green', alpha=0.75,
18
19     # hist uses np.histogram under the hood to create 'n' and 'bins'.
20     # np.histogram returns the bin edges, so there will be 50 probability
21     # density values in n, 51 bin edges in bins and 50 patches.
22
23     To get
24     # everything lined up, we'll compute the bin centers
25     bincenters = 0.5*(bins[1:]+bins[:-1])
26     # add a 'best fit' line for the normal PDF
27     y = mlab.normpdf(bincenters, mu, sigma)
28     ax.plot(bincenters, y, 'r--', linewidth=1, label='PDF of Normal Distributio

```

```

27
28 ax.set_xlabel('Random Values')
29 ax.set_ylabel('Probability')
30
31 ax.set_xlim(40, 160)
32 ax.set_ylim(0, 0.035)
33 ax.grid(True)
34 plt.legend()
35 plt.savefig('rg_fitness.eps', format='eps')
36
37 def rg_histogram():
38     #1.1 Does your RNG generate random numbers?
39     randgen_1 = np.random.RandomState(1)
40     list_random = []
41     number = 10000
42     for i in range(0, number):
43         list_random.append(randgen_1.rand())
44
45     print "1.1 Does your RNG generate random numbers?"
46     print "Plot the histogram of generated random numbers"
47     fig = plt.figure()
48     ax = fig.add_subplot(111)
49     n, bins, patches = ax.hist(list_random, bins=100, normed=1, facecolor='green')
50     fit = [1 for x in bins]
51     ax.plot(bins, fit, 'r--', linewidth=1, label='PDF of Uniform Distribution')
52     ax.set_xlabel("Generated Random Value")
53     ax.set_ylabel("Count in Each Bin")
54     ax.grid(True)
55     plt.legend()
56     plt.savefig('rg_histogram.eps', format='eps')
57
58 def rg_seed():
59     """How do you initialize the seed of your RNG?"""
60     current_time = time()
61     np.random.RandomState(int(current_time))
62     print "1.2 How do you initialize the seed of your RNG?"
63     print "Use current time as int type: %s\n" % int(current_time)
64
65 def rg_diff():
66     """Generate two sequences of 1000000 numbers each
67     for every sequence use a different seed.
68     """
69     randgen1 = np.random.RandomState(1)
70     randgen2 = np.random.RandomState(2)
71
72     listA = randgen1.uniform(low=0.0, high=1.0, size=1000000)

```



```

73 listB = randgen2.uniform(low=0.0, high=1.0, size=1000000)
74
75 A, B = stats.mstats.ttest_ind(listA, listB)
76 print "The p-value is %s, the t-statistics is %s" % (B, A)
77
78 overall = np.append(listA, listB)
79
80 # Using set operation to differentiate 2 different lists
81 # We know that for set, every element is identical.
82 # So we can find the length of the set after we combine 2 lists.
83 # Method 1
84 temp3 = tuple(set(listA) - set(listB))
85 print len(temp3)
86 # Method 2
87 set_overall = set(overall)
88 print len(set_overall)
89
90
91
92
93 if __name__ == "__main__":
94     # rg_histogram()
95     # rg_fitness()
96     # rg_seed()
97     rg_diff()

```