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

Yan Jiaqi A20321362
Liu Ping A20326564
CS555 Project
supervised by Edward Chlebus
Live Section

December 12, 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**

2.1 Does Numpy generate random variables? 5

2.2 How we initialize the seed of our RNG? 6

2.3 Are two generated random value sequences different? 6

3 Eliminate Warm-up Period **6**

4 System Properties at Stationary State **7**

5 Conclusion **11**

A Source Code Printout **13**

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

```
function SIMULATION_CORE(arrive_time_seq, serve_time_seq)
    let arrive_time_seq denote the arriving time of each packet
    let serve_time_seq denote the service time of each packet
    let N denote the number of total packets
    while pkt_served + pkt_dropped < N do
        if pkt_seen < N then                ▷ Insert new arrival event to event list
            ts ← arrive_time_seq[pkt_seen]
            Create new arrival event evt with time stamp ts and id pkt_seen
            Insert evt to event_list
        end if
        Sort event_list based on events' time stamp
        Pop up the next event evt_x we need to handle in event_list
        clock ← evt_x.time_stamp
        if evt_x is departure event then
            if queue buffer is empty then
                Set the status of the server evt_x is leaving to idle
            else
                − − waiting
            end if
            + + pkt_served
            evt_x.exit_time ← clock
            spending_time ← evt_x.exit_time − evt_x.enter_time
            Record the spending time of packet of event evt_x
        end if
        if evt_x is arrival event then
            if queue buffer is full then
                + + pkt_dropped
            else
                if There is available server then
                    id ← evt_x.pkt_id
                    ts ← clock+ exponential service time
                    Choose an available server s
                    Mark s as busy
                else
                    (ts, s) ←SCHEDULE_DEPARTURE( )
                    ts ← ts+ exponential service time
                    + + waiting
                end if
                Create new departure event evt with time stamp ts
                evt.enter_time ← clock
                evt.depart_srv ← s
                Insert evt to event_list
            end if
        end if
    end while
end function
```

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

function SCHEDULE_DEPARTURE
  Sort *event_list*
  for each server s in the system **do**
    Find the last *departure* event *evt* scheduled on this server s
    $earlier\_time\_stamp = \text{MIN}(earlier\_time\_stamp, evt.time\_stamp)$
  end for
  Return *earlier_time_stamp* and the corresponding *server_id*
end function

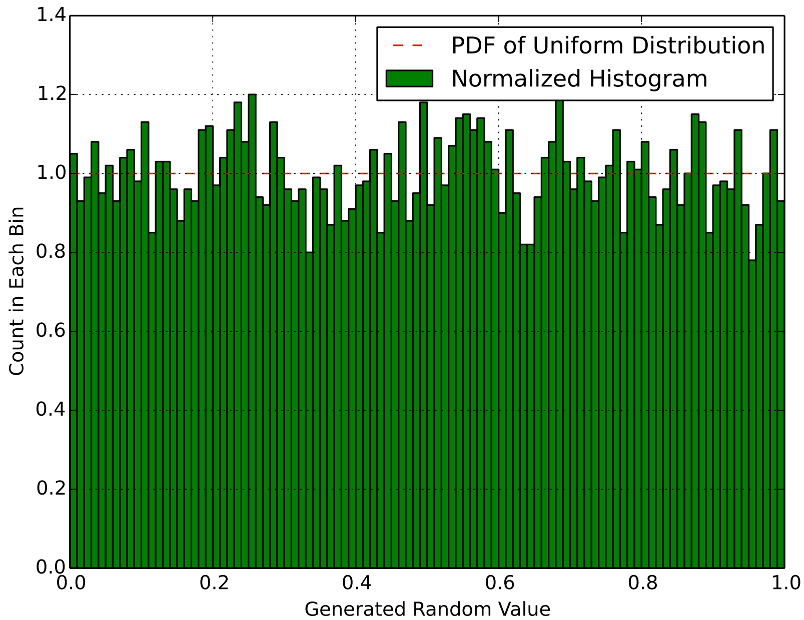


Figure 1: Histogram of Uniform Random Values Generated by Numpy

2.1 Does Numpy generate random variables?

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.

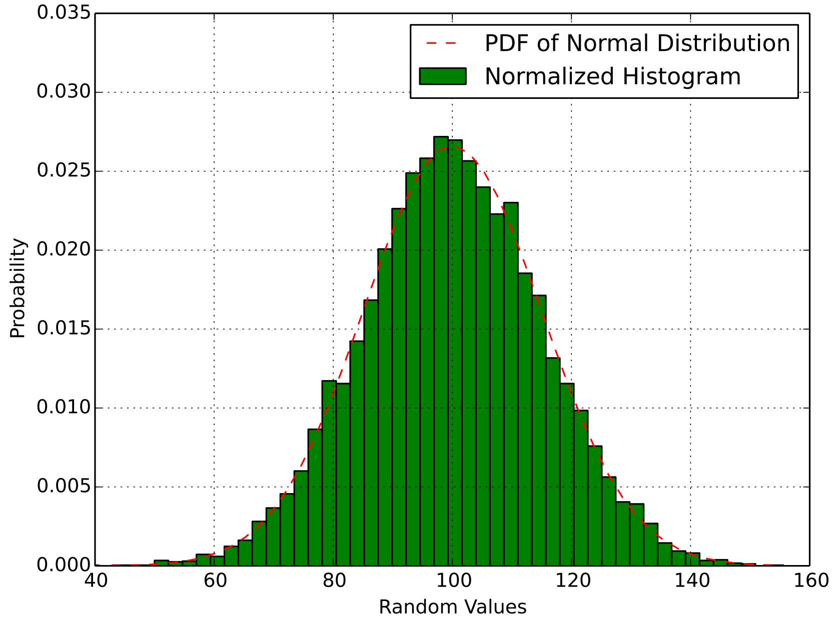


Figure 2: Comparison to Best Fit Curve for Normal Distribution

2.2 How we initialize the seed of our RNG?

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**.

2.3 Are two generated random value sequences different?

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. 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. Therefore, if λ is 10, the simulation will terminate at approximately 100 seconds. This termination time varies from simulation run to simulation run since arrival time and service time is randomly generated. 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 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

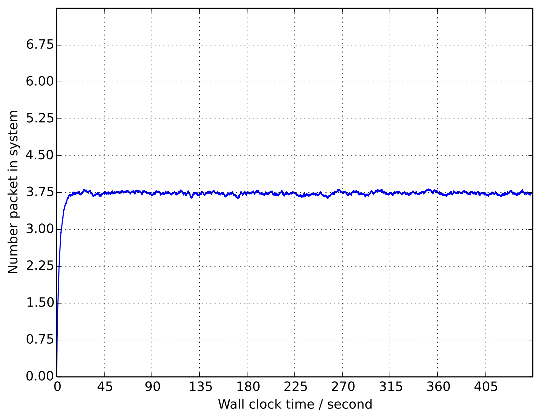
In the last section, we have applied Welch graphical procedure and show the average simulation results of 5000 runs for each system with different initial state. These figures clearly identifies the nonstationary and stationary states. For system A, the time threshold for stable state is around 45 second 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:

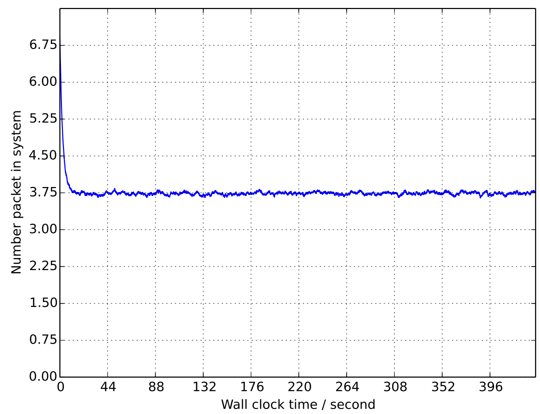
- **Blocking Probability** P_B for a particular simulation run is defined by the ratio of number of dropped packets to the number of arrived packets:

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

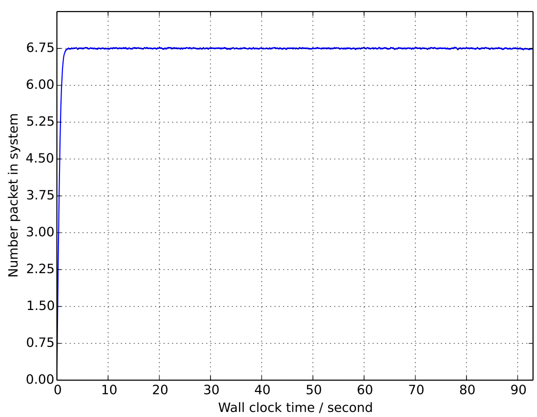
- **Mean Spending Time** S, assuming there are totally n served packet over



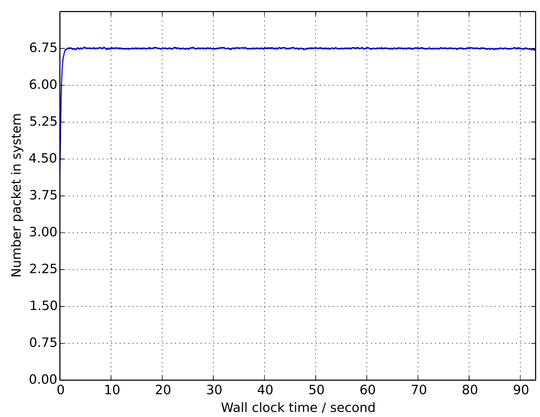
(a) **System A with 0 Initial Packets**



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

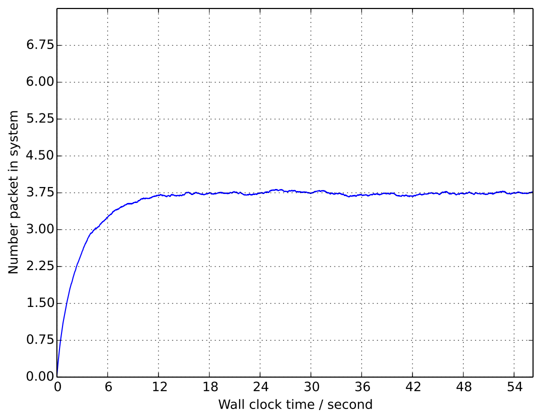


(c) **System B with 0 Initial Packets**

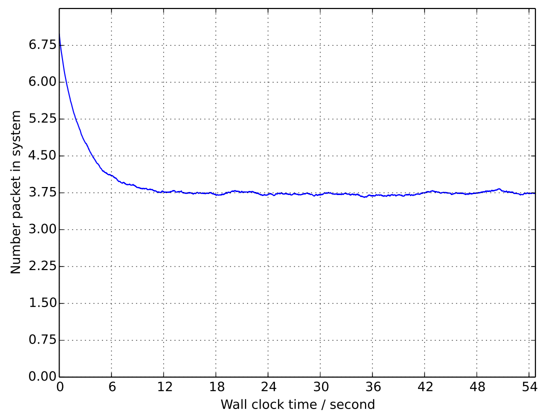


(d) **System B with 4 Initial Packets**

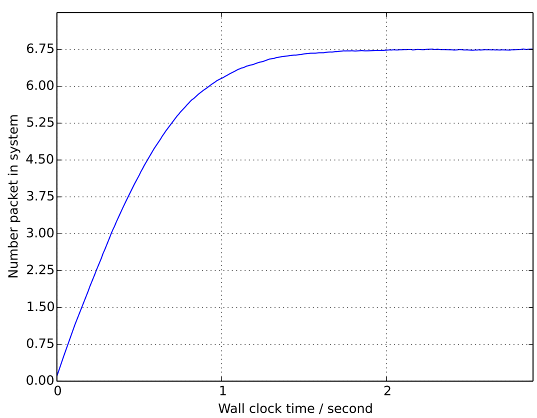
Figure 3: **Average 5000 Simulation Runs**



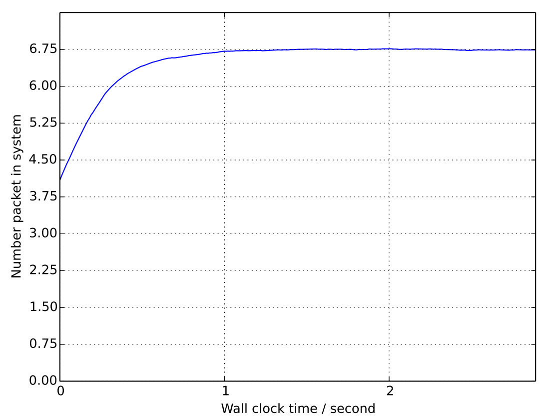
(a) **System A with 0 Initial Packets**



(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**

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 $\alpha = 0.1$.

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: **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

Table 2: **System B with $x_{t=0} = 0$**

	Our Simulator	Mathematica	Error
S_s	3.359	≈ 3.375	0.474%
N_s	6.739	≈ 6.750	0.163%

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.

To further validate our simulator, we compare the simulation result of system B under $x_{t=0} = 0$ initial state to the theoretic result from Mathematica. The output from Mathematica is shown in Figure 5.

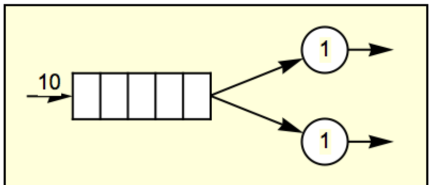
We summarize the results in Table 2. As indicated by the last column, our simulation gives very closing results to theoretic values. The error of system stable state measurement, mean spending time in system S_s and mean number of packets in system N_s, are less than 0.5%.

5 Conclusion

In this project, using discrete event simulation, we simulated 2 queueing system differentiated by packet arrival rate and initial state. Simulation results show that

- As long as the packet arrival rate λ and service rate $\mu$are fixed, systems with different initial state will eventually converge to the same stationary state. That is with same blocking probability, mean spending time in system etc.

- From Table 1, we can conclude that, assuming service rate is fixed, a system with larger λ will eventually have larger blocking probability, mean spending time in system and mean number of packets in the system.



"Basic Properties"	
"QueueNotation"	"M/M/2/7"
"ArrivalRate"	10
"ServiceRate"	1
"UtilizationFactor"	$\frac{195\,305}{195\,311}$
"Throughput"	$\frac{390\,610}{195\,311}$
"ServiceChannels"	2
"SystemCapacity"	7
"InitialState"	0
"Performance Measures"	
"MeanSystemSize"	$\frac{1\,318\,360}{195\,311}$
"MeanSystemTime"	$\frac{131\,836}{39\,061}$

Figure 5: Mathematica Result for M/M/2/7 system with $\lambda = 10$

A Source Code Printout

Listing 1: Source Code for Random Generator: mmrng.py

```
#!/usr/bin/python

import numpy as np

def generate_exp(length, mu, seed):
    """Generate exponential random sequence with seed"""
    randgen = np.random.RandomState(seed)
    return randgen.exponential(scale=mu, size=length)

def generate_exp_single(mu, seed):
    """Generate one exponential random value with seed"""
    randgen = np.random.RandomState(seed)
    random_values = randgen.exponential(scale=mu, size=1)
    return random_values[0]
```

Listing 2: Source Code for Event: mmevent.py

```
#!/usr/bin/python

class MMEvent(object):
    """Abstraction of packet arrival or departure"""
    def __init__(self, pkt_id, name, ts):
        """Create MMEvent object

        Attributes:
            pkt_id: which packet this event is about
            event_type(str): 'arrival' or 'departure'
            time_stamp: when should we handle this event;
                we need this value to put event into event_list
            enter_time: the moment it enter the system
            exit_time: the moment it exit the system
        """
        super(MMEvent, self).__init__()
        self.pkt_id = pkt_id
        self.evt_name = name
        self.time_stamp = ts
        self.enter_time = 0
```

```
        self.exit_time = 0
        self.depart_srv = None
```

Listing 3: Source Code for System: mmsystem.py

```
#!/usr/bin/python

class MMSystem(object):
    """Various system states and counters for M/M/2/2+5 system

    Attributes:
        num_srv: number of servers in this system
        capacity: queue size of the system, excluding buffer on server
        pkt_seen: number of packets arrived so far
        pkt_dropped: number of packets dropped so far
        pkt_dropped_id(list): who is dropped by system?
        pkt_waiting: number of packets in the system queue
        pkt_served: number of packets exited
        spending_time: how much each packet is spending in the system
        log_time: starting time stamp that pkt in system is changed
        log_num_pkt_inside: history of number of packets in system
        stable: if we passed warm-up period
        srv_status: idle or busy for a particular server
    """
    def __init__(self, num_srv, capacity):
        super(MMSystem, self).__init__()
        # immutable properties
        self.num_srv = num_srv
        self.capacity = capacity

        # couters
        self.pkt_dropped = 0
        self.pkt_dropped_id = []
        self.pkt_seen = 0
        self.pkt_waiting = 0
        self.pkt_served = 0

        # result log info
        self.spending_time = {}
        self.log_time = []
        self.log_num_pkt_inside = []
```

```python

        # state variables
        self.stable = False
        self.srv_status = {}
        for i in range(0, self.num_srv):
            self.srv_status[i] = 'idle'

    def full(self):
        """Test if the queue is full"""
        return self.pkt_waiting >= self.capacity

    def available(self):
        """Test if any server is idle"""
        return 'idle' in self.srv_status.values()

    def num_available_servers(self):
        """Return number of available servers"""
        counter = 0
        for key, value in self.srv_status.items():
            if value == 'idle':
                counter += 1
        return counter

    def available_server(self):
        """Find any 'idle' server in system"""
        for key, value in self.srv_status.items():
            if value == 'idle':
                return key

    def dump_num_pkt_inside(self, time):
        """Update pkt inside system and log it with current time stamp"""
        self.log_time.append(time)
        if self.pkt_waiting > 0:
            num_pkt_inside = self.pkt_waiting + 2
        else:
            num_pkt_inside = 2 - self.num_available_servers()

        self.log_num_pkt_inside.append(num_pkt_inside)

    def dump_pkt_spending_time(self, evt):
        """Calculate the duration a pkt spent in the system"""
```

```
        self.spending_time[evt.pkt_id] = evt.exit_time - evt.enter_time
```

Listing 4: Source Code for Welch Procedure: mmwelch.py

```
#!/usr/bin/python

import numpy as np
from matplotlib import pyplot as plt
import math

class MMWelch(object):
    """Welch graphic method to eliminate warm-up period"""
    def __init__(self, num_replicas, interval, run_length, prefix, \
                 mode='online', warmup=1):
        """Create object with

        Attributes:
            num_replicas: to average these number of replicas
            run_length: the min length over all replications
            interval & time_seq: to draw simulation time
            prefix: name for output .txt and .eps files
        """
        super(MMWelch, self).__init__()
        self.num_replicas = float(num_replicas)
        self.prefix = prefix
        self.mode = mode
        self.warmup = float(warmup)
        self.interval = float(interval)

        if self.mode == 'online':
            self.run_length = run_length
            self.time_seq = [self.interval * i for i in range(0, self.run_length)]
            self.avg_run = [0] * self.run_length
        if self.mode == 'offline':
            load_file_name = self.prefix + 'Avg%d.txt' % self.num_replicas
            self.avg_run = np.loadtxt(load_file_name)
            warmup_period = len(self.avg_run) / self.warmup
            self.avg_run = self.avg_run[:warmup_period]
            self.run_length = len(self.avg_run)
            self.time_seq = [self.interval * i for i in range(0, self.run_length)]
            self.prefix += 'Ofl'
```

```
        print "Use %d as common length " % self.run_length

    def average_all_runs(self):
        """Average all replica runs, store in self.avg_run"""
        for i in range(0, int(self.num_replicas)):
            # truncate only the common part of each runs
            load_file_name = '%sRun%d.txt' % (self.prefix, i)
            replica = np.loadtxt(load_file_name, dtype=int, comments='#')
            replica = replica[:self.run_length]
            self.avg_run = np.add(replica, self.avg_run)
        # average and save
        self.avg_run = [float(x) / self.num_replicas for x in self.avg_run]
        file_name = self.prefix + 'Avg%d.txt' % self.num_replicas
        np.savetxt(file_name, self.avg_run)

    def plot_avg_run(self):
        """Draw a figure, output to file"""
        #y_max = math.ceil(max(self.avg_run)) + 0.5
        y_max = 7.5 # empiriall value
        x_max = math.ceil(self.time_seq[-1] / 10) * 10

        figure_name = self.prefix + 'Avg%d.eps' % self.num_replicas
        plt.figure()
        plt.plot(self.time_seq, self.avg_run)
        plt.xlabel('Wall clock time / second')
        plt.ylabel('Number packet in system')
        plt.xticks(np.arange(0, x_max, x_max / 10.0))
        plt.yticks(np.arange(0, y_max, y_max / 10.0))
        plt.xlim(self.time_seq[0], self.time_seq[-1])
        plt.ylim(0, y_max)
        plt.grid()
        plt.savefig(figure_name, format='eps')
```

Listing 5: Source Code for Result Reporter: mmreporter.py

```
#!/usr/bin/python

import bisect

class MMReporter(object):
    """Calculate system measurements"""
```

```
    def __init__(self, system, ats):
        super(MMReporter, self).__init__()
        self.system = system
        self.ats = ats
        self.obsrv_pkt = []

    def blocking_prob(self):
        """Blocking probability"""
        num_dropped = 0
        num_seen = 0

        for i in range(self.system.pkt_seen):
            if self.ats[i] > (self.system.log_time[-1] / 3.0):
                if i in self.system.pkt_dropped_id:
                    num_dropped += 1
                num_seen += 1

        return num_dropped/float(num_seen)
        #return float(self.system.pkt_dropped) / float(self.system.pkt_seen)

    def mean_time_spending_in_system(self):
        """Mean time pkt spending in the system"""
        dur_sum = 0.0
        spending_time_seq = self.system.spending_time.values()
        start = int(len(spending_time_seq) / 2.0)
        end = int(len(spending_time_seq) / 3.0 * 2)
        for i in range(start, end):
            dur_sum += spending_time_seq[i]
        return dur_sum / (end - start)

    def mean_num_pkt_in_system(self):
        """Mean number of pkt in the system"""
        num_pkt_duration = {}
        entire_duration = 0.0
        product_sum = 0.0

        start = int(len(self.system.log_time) / 2.0) - 1
        end = int(len(self.system.log_time) / 3.0 * 2) - 1
        for i in range(start, end):
            dur = self.system.log_time[i+1] - self.system.log_time[i]
            num_pkt = self.system.log_num_pkt_inside[i]
```

```
            if num_pkt in num_pkt_duration.keys():
                num_pkt_duration[num_pkt] += dur
            else:
                num_pkt_duration[num_pkt] = dur
            entire_duration += dur

        for num_pkt, dur in num_pkt_duration.items():
            product_sum += num_pkt * dur
        return product_sum / entire_duration

    def warm_up_finding(self, interval_sequence):
        """Find #pkt for all moment in interval sequence"""

        for i in interval_sequence:
            index = bisect.bisect_left(self.system.log_time, i)
            index = index - 1
            self.obsrv_pkt.append(self.system.log_num_pkt_inside[index])

        return self.obsrv_pkt
```

Listing 6: Source Code for Simulator: mmsimulator.py

```
#!/usr/bin/python

from mmevent import MMEvent

class MMSimulator(object):
    """Simulator for M/M/2/7 system"""
    def __init__(self, system, end_time):
        super(MMSimulator, self).__init__()
        self.system = system
        self.end_time = end_time
        self.clock = 0
        self.initialized = False

    def init_simulation(self, num_pkt_init):
        """Initialize simulator before its core"""
        self.init_system_status(num_pkt_init)
        self.init_event_list()
        self.initialized = True

```

```
    def init_system_status(self, num_pkt_init):
        """Set system initial status"""
        self.system.log_time.append(0)
        self.system.log_num_pkt_inside.append(num_pkt_init)
        # initial number of pkt may large than system capacity
        # num_busy_srv = num_pkt_init - self.system.capacity
        # if num_busy_srv > 0 and num_busy_srv <= self.system.num_srv:
            # for i in range(0, num_busy_srv):
                # self.system.srv_status[i] = 'busy'

    def init_event_list(self):
        """Do necessary initialization"""
        self.event_list = []

    def sort_event_list(self):
        """Sort event list on time stamp of every events"""
        self.event_list.sort(key=lambda event: event.time_stamp, reverse=False)

    def last_departure_srv(self, srv_id):
        """Search in event list the last depart event
        with specified depart server id"""
        self.sort_event_list()
        for event in reversed(self.event_list):
            if event.evt_name == 'departure' and event.depart_srv == srv_id:
                return event

    def schedule_departure(self):
        """Find server for new depart event with the depart time stamp"""
        earliest_ts = None
        self.sort_event_list()
        for s_id in self.system.srv_status.keys():
            evt = self.last_departure_srv(s_id)
            if earliest_ts == None or earliest_ts > evt.time_stamp:
                earliest_ts = evt.time_stamp
                earliest_srv_id = s_id
        return (earliest_ts, earliest_srv_id)

    def next_event(self):
        """Pop up the earliest event"""
        self.sort_event_list()
        return self.event_list.pop(0)
```

```

    def should_continue(self, N):
        """Test if seen all pkt and clock NOT exceed predefined end time"""
        return self.system.pkt_served + self.system.pkt_dropped < N \
            and self.clock < self.end_time

    def simulate_core(self, arrive_time_seq, depart_time_seq_server1, \
                      depart_time_seq_server2):
        """Discrete event simulation"""
        if not self.initialized:
            print "Simulator is not explicitly initialized"

        N = len(arrive_time_seq)
        flag_server1 = 0
        flag_server2 = 0

        while self.should_continue(N):
            # schedule/add a new pkt arrive event if still pkt unseen
            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_arrival_ts)
                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
                        # mark this server as 'busy'
                        new_depart_srv = self.system.available_server()
                        if new_depart_srv == 0:
                            new_depart_ts = self.clock + \
                                depart_time_seq_server1[flag_server1]
                            flag_server1 += 1
                        else:
                            new_depart_ts = self.clock + \
                                depart_time_seq_server2[flag_server2]
                            flag_server2 += 1

                        self.system.srv_status[new_depart_srv] = 'busy'
                    else:
                        # find the server pkt should go
                        earliest_ts, earliest_srv = self.schedule_departure()
                        if earliest_srv == 0:
                            new_depart_ts = earliest_ts + \
                                depart_time_seq_server1[flag_server1]
```

```
                            flag_server1 += 1
                        else:
                            new_depart_ts = earliest_ts + \
                                depart_time_seq_server2[flag_server2]
                            flag_server2 += 1

                        new_depart_srv = earliest_srv
                        self.system.pkt_waiting += 1

                    # either server became busy or waiting packet increases
                    # log num_pkt_inside the system
                    self.system.dump_num_pkt_inside(self.clock)
                    # actually insert new departure event
                    new_depart = MMEvent(evt_x.pkt_id, 'departure', new_depart_ts)
                    new_depart.enter_time = self.clock
                    new_depart.depart_srv = new_depart_srv
                    self.event_list.append(new_depart)
```

Listing 7: Source Code for Confidence Interval: mmconfidence.py

```
import numpy as np

def get_confidence_level():

    Z = 1.645
    # Need to run the other 2 tonight, then put them in this list
    prefix_list = ["Lmda2Init7", "Lmda2Init0", "Lmda10Init4", "Lmda10Init0"]
    file_name = 'confidence_level.txt'
    output = open(file_name, 'a')

    for prefix in prefix_list:
        file_blocking = prefix + "_blocking.txt"
        file_spending = prefix + "_spending.txt"
        file_num_customers = prefix + "_num_customers.txt"

        output.write('####################################################\n')
        output.write('######### System configuration %s #########\n' % prefix)
        output.write('####################################################\n')

        #Blocking Probability
        blocking = np.loadtxt(open(file_blocking, 'rb'))
```

```
        length_blocking = len(blocking)
        mean_blocking = np.mean(blocking)
        # This is sample variance for estimator
        var_blocking = np.var(blocking) * length_blocking / (length_blocking-1)
        error_blocking = Z * np.sqrt(var_blocking/len(blocking))
        output.write('Mean blocking probability:    %.6f +- %.6f\n' \
                     % (mean_blocking, error_blocking))

        #Mean Spending time
        spending = np.loadtxt(open(file_spending, 'rb'))
        length_spending = len(spending)
        mean_spending = np.mean(spending)
        # This is sample variance for estimator
        var_spending = np.var(spending) * length_spending / (length_spending-1)
        error_spending = Z * np.sqrt(var_spending/len(spending))
        output.write('Mean spending time in system: %.6f +- %.6f\n' \
                     % (mean_spending, error_spending))

        #mean number of customers
        num_customers = np.loadtxt(open(file_num_customers, 'rb'))
        length_num_customers = len(num_customers)
        mean_num_customers = np.mean(num_customers)
        # This is sample variance for estimator
        var_num_customers = np.var(num_customers) * length_num_customers \
            / (length_num_customers-1)
        error_num_customers = Z * np.sqrt(var_num_customers/len(num_customers))
        output.write('Mean # of packets in system:  %.6f +- %.6f\n' \
                     % (mean_num_customers, error_num_customers))
        output.write('##################################################\n')
        output.write('\n')
```

Listing 8: Source Code for Main Program: mmmain.py

```
#!/usr/bin/python

from mmrng import generate_exp
from mmsystem import MMSystem
from mmreporter import MMReporter
from mmsimulator import MMSimulator
from mmwelch import MMWelch
from mmconfidence import get_confidence_level
```

```
import time, math
import numpy as np

def roundup_hundreds(num):
    """To round up ending time of the simulation"""
    return int(math.ceil(num / 100)) * 100

def simulator_driver(trial, l, u, num_pkt_init, num_pkts, obsrv_int, seed, prefix):
    """Reuse this function to get many replicas of simulation"""
    # ending time is dependent with arriving rate.
    end_time = 1000

    mm27 = MMSystem(num_srv, num_buffer)

    # arrival interval
    ats = generate_exp(num_pkts, 1.0 / l, seed)
    # arrival time stamp
    ats = np.cumsum(ats)
    # insert init pkt events
    init_ats = np.array([0] * num_pkt_init)
    ats = np.insert(ats, 0, init_ats)

    # departure time stamp
    dts_server1 = generate_exp(num_pkts + num_pkt_init, u, seed + 1)
    dts_server2 = generate_exp(num_pkts + num_pkt_init, u, seed + 2)

    simulator = MMSimulator(mm27, end_time)
    simulator.init_simulation(num_pkt_init)

    t0 = time.time()
    simulator.simulate_core(ats, dts_server1, dts_server2)

    reporter = MMReporter(mm27, ats)

    duration = time.time() - t0
    end_time = int(math.ceil(simulator.clock))
    ots = [obsrv_int for _ in range(0, int(end_time / obsrv_int) + 1)]
    ots = np.cumsum(ots) # obverse time stamp

    # output results to this file
    file_name = prefix + 'Run%d.txt' % trial
```

```
    # align results to observation intervals
    observations = reporter.warm_up_finding(ots)
    # output statistic results as header
    optional_output = ''
    optional_output += "##########################################################\n"
    optional_output += "################## Simulation results ##################\n"
    optional_output += "##########################################################\n"
    optional_output += "#Running time of NO.%d trial %.4fs\n" % (trial, duration)
    optional_output += "#Ending event time stamp %.4f\n" % end_time
    optional_output += "#Blocking probability %.4f\n" % reporter.blocking_prob()
    optional_output += "#Mean time spent in system %.4f\n" \
        % reporter.mean_time_spending_in_system()
    optional_output += "#Mean #pkt in system %.4f\n" \
        % reporter.mean_num_pkt_in_system()
    optional_output += "##########################################################\n"

    # for main to get min common simulation length
    return len(observations),reporter.blocking_prob(), \
        reporter.mean_time_spending_in_system(), reporter.mean_num_pkt_in_system()

def eliminate_warmup_period(l, u, num_pkt_init, seed):
    num_obsrv = 999999
    num_trials = 5000
    num_pkts = 1000
    obsrv_int = min(0.01, 1.0 / l / 10)
    blocking_list = []
    spending_list = []
    num_customers_list = []
    prefix = 'Lmda%dInit%d' % (l, num_pkt_init)
    for i in range(num_trials):
        new_obsrv, blocking, spending, num_customers = \
            simulator_driver(i, l, u, num_pkt_init, \
                             num_pkts, obsrv_int, seed, prefix)
        num_obsrv = min(num_obsrv, new_obsrv)
        blocking_list.append(blocking)
        spending_list.append(spending)
        num_customers_list.append(num_customers)
        seed += 10
    file_block = prefix + "_blocking.txt"
    file_spending = prefix + "_spending.txt"
    file_num_customers = prefix + "_num_customers.txt"
```

```
    #Save the history
    np.savetxt(file_block, blocking_list, fmt = '%1.5f')
    np.savetxt(file_spending, spending_list, fmt = '%1.5f')
    np.savetxt(file_num_customers, num_customers_list, fmt = '%1.5f')

    # welch = MMWelch(num_trials, obsrv_int, num_obsrv, prefix)
    # welch.average_all_runs()
    # Draw figure in offline mode
    # warmup = 1 if l == lambdaA else 32
    # welch = MMWelch(num_trials, obsrv_int, num_obsrv, prefix, 'offline', warmup)
    # welch.plot_avg_run()

def run_system(l, u, num_pkt_init):
    seed = int(time.time())
    eliminate_warmup_period(l, u, num_pkt_init, seed)

def main():
    run_system(lambdaA, u, 0)
    run_system(lambdaA, u, 7)
    run_system(lambdaB, u, 0)
    run_system(lambdaB, u, 4)
    get_confidence_level()

if __name__ == '__main__':
    num_srv = 2
    num_buffer = 5
    lambdaA = 2.0
    lambdaB = 10.0
    u = 1.0

    main()
```

Listing 9: Source Code for Validating Random Generator: valid_random_generator.py

```
#!/usr/bin/python

import numpy as np
from time import time
from scipy import stats
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
```

```

def rg_fitness():
    mu, sigma = 100, 15
    x = mu + sigma * np.random.randn(10000)

    fig = plt.figure()
    ax = fig.add_subplot(111)

    # the histogram of the data
    n, bins, patches = ax.hist(x, 50, normed=1, facecolor='green', \
                               alpha=0.75, label="Normalized Histogram")

    # hist uses np.histogram under the hood to create 'n' and 'bins'.
    # np.histogram returns the bin edges, so there will be 50 probability
    # density values in n, 51 bin edges in bins and 50 patches.  To get
    # everything lined up, we'll compute the bin centers
    bincenters = 0.5*(bins[1:]+bins[:-1])
    # add a 'best fit' line for the normal PDF
    y = mlab.normpdf( bincenters, mu, sigma)
    ax.plot(bincenters, y, 'r--', linewidth=1, label='PDF of Normal Distribution')

    ax.set_xlabel('Random Values')
    ax.set_ylabel('Probability')

    ax.set_xlim(40, 160)
    ax.set_ylim(0, 0.035)
    ax.grid(True)
    plt.legend()
    plt.savefig('rg_fitness.eps', format='eps')

def rg_histogram():
    #1.1 Does your RNG generate random numbers?
    randgen_1 = np.random.RandomState(1)
    list_random = []
    number = 10000
    for i in range(0, number):
        list_random.append(randgen_1.rand())

    print "1.1 Does your RNG generate random numbers?"
    print "Plot the histogram of generated random numbers"
    fig = plt.figure()
```

```
    ax = fig.add_subplot(111)
    n, bins, patches = ax.hist(list_random, bins=100, normed=1, facecolor='green', \
                               alpha=0.75, label='Normalized Histogram')
    fit = [1 for x in bins]
    ax.plot(bins, fit, 'r--', linewidth=1, label='PDF of Uniform Distribution')
    ax.set_xlabel("Generated Random Value")
    ax.set_ylabel("Count in Each Bin")
    ax.grid(True)
    plt.legend()
    plt.savefig('rg_histogram.eps', format='eps')

def rg_seed():
    """How do you initialize the seed of your RNG?"""
    current_time = time()
    np.random.RandomState(int(current_time))
    print "1.2 How do you initialize the seed of your RNG?"
    print "Use current time as int type: %s\n" % int(current_time)

def rg_diff():
    """Generate two sequences of 1000000 numbers each
    for every sequence use a different seed.
    """
    randgen1 = np.random.RandomState(1)
    randgen2 = np.random.RandomState(2)

    listA = randgen1.uniform(low=0.0, high=1.0, size=1000000)
    listB = randgen2.uniform(low=0.0, high=1.0, size=1000000)

    A, B = stats.mstats.ttest_ind(listA, listB)
    print "The p-value is %s, the t-statistics is %s" % (B, A)

    overall = np.append(listA,listB)

    # Using set operation to differentiate 2 different lists
    # We know that for set, every element is identical.
    # So we can find the length of the set after we combine 2 lists.
    # Method 1
    temp3 = tuple(set(listA) - set(listB))
    print len(temp3)
    # Method 2
    set_overall = set(overall)
```

```
    print len(set_overall)

if __name__ == "__main__":
    rg_histogram()
    rg_fitness()
    rg_seed()
    rg_diff()
```