COMP9334 Project
Session 1, 2017
Getting the best response time out of your power bill

REPORT

Student Name:	Boshen Hu
ZID:	z5034054
Lecturer:	**Chun Tung Chou**

How did I do the simulation:

For doing the project, I use Matlab to both generate the required probability distributions and simulate the PS server (in "**ps_server.m**"). The structure of my code draws on the lecture program code in Week 6. However, only the structure is similar. The method of computing each parameter is quite different.

1. Generating the required probability distributions:
   For we need to maintain a list of jobs for the servers, there are two kinds of the required attributes for a job, the arrival time and the service time. In the script of mine, the job would be generated dynamically by assigning two variables: next_arrival_time and service_time_next_arrival.
   1) next_arrival_time:
      From the project description, the inter-arrival time is the product of an exponential a1k and a uniform a2k. The information in the project description would be enough to help us compute the inter-arrival time = (-log(1-rand(1))/lambda) * ((1.17-0.75) * rand()+0.75), where lambda is 1/(mean arrival rate) == 1/7.2. Then, we assume that the PS server our simulated is the first server among the ten, the received requests are Request 1, Request (s + 1), Request (2s + 1) … Request (ms + 1), where m is an integer. Therefore, we initial the next_arrival_time as (-log(1-rand(1))/lambda) * ((1.17-0.75) * rand()+0.75), then we jump the job lists to (s+1), and so on, using the code:
      ```
      % generate a new job and schedule its arrival
      jump_job_list = 0;
      for i=1:s
          jump_job_list = jump_job_list + (-log(1-rand(1))/lambda)* ((1.17-0.75)*rand()+0.75);
      end
      next_arrival_time = master_clock + jump_job_list;
      ```

   2) service_time_next_arrival:
      Similarly, we can also generate the service time by utilising the probability density function g(t) when a server operates at 1GHz. The service time of an arrival would be t/f, where t can be generated by g(t) and f can be computed when knowing the number of open servers. Hence, $g(t) = \frac{\gamma}{t^{\beta}}(0.43 < t < 0.98)$, where γ ≈ 1.289, β = 0.86, => $t = \sqrt[0.86]{1.289/g(t)}$, f = 1.25 + 0.31(p/200 - 1), where p = 2000/s (s is the number of open servers),

      $$a \text{ service time} = \frac{\mathrm{t}}{\mathrm{f}} = \frac{\sqrt[0.86]{\frac{1.289}{g(t)}}}{1.25 + 0.31\left(\frac{\frac{2000}{s}}{200} - 1\right)},$$ where g(t) is between 0 and 1, s is between 1 and 10, theoretically (actually between 3 and 10).

2. Simulating the PS server

As the hint on the project specification, only we could simulate only one of the s servers. Therefore, the `mean response time` we want is T/(N*s), where T is the cumulative response time, N is number of completed customers at the end of the simulation of one server and s is the number of open servers. Simulation time (Tend) has been fixed to 2000s. If master_lock > Tend, the simulation would stop.

We push on the simulation by updating two types of events: arrivals and departures.

```
% Find out whether the next event is an arrival or depature
%
% We use next_event_type = 1 for arrival and 0 for departure
%
if (next_arrival_time < next_departure_time)
    next_event_time = next_arrival_time;
    next_event_type = 1;
else
    next_event_time = next_departure_time;
    next_event_type = 0;
end
```

As the code shows, we judge whether an event is an arrival or a departure by comparing next arrival time and next departure time, if the next arrival time is smaller, it should come first, therefore the event is an arrival; otherwise, it is a departure.

1) An arrival
   If the event is an arrival, we need to update:
   i) next arrival time (has been discussed in 1.1))

   ii) job list
   We perform job list update by using these codes:

```
%
% update job list
%
if queue_length
    partial_completion = (master_clock - former_event_time) / queue_length;
    job_list(:,2) = job_list(:,2) - partial_completion;
end

job_list = [job_list ; next_arrival_time service_time_next_arrival];
queue_length = queue_length + 1;
```

   where partial_completion means, at the time of the master clock, how many units of service have been achieved. We first subtract the achieved units of service and then, we add new job information to the list.

   iii) next departure time

```
%
% update next departure time
%
if ~(queue_length == 1)
    minimum_workload = Inf;
    for i=1:queue_length
        if minimum_workload > job_list(i,2)
            % disp(job_list(i,2))
            minimum_workload = job_list(i,2);
        end
        %disp(minimum_workload)
    end
    next_departure_time = master_clock + minimum_workload/((master_clock-former_event_time)/queue_length-1);
else % the former is a departure with empty job list
    next_departure_time = master_clock + job_list(1,2);
    rest_time = master_clock - former_event_time; % no job time
end
```

   In this code, first, we find the minimum of the workload of a single job compared with jobs, then we use it to compute the next departure time.

2) A departure
   If there is a departure, we need to update
   i) Cumulative response time T
      T = master_clock - rest_time, where rest time means the cumulative time with no jobs.

   ii) **Job list** as well as number of completed customers at the end of the simulation **N** (detailed information is in the annotation)

```
% update job list
% update N
%
if queue_length
    partial_completion = (master_clock - former_event_time) / queue_length;
    job_list(:,2) = job_list(:,2) - partial_completion;
    job_list_ = [];
    queue_length_ = 0;
    for i=1:queue_length
        %
        % ~(job_list(i,2) == 0) -> ~(job_list(i,2) < 0.0001)
        % because different degree of accuracy may cause infinite
        % loop, therefore, we assume a job which
        % left service time < 0.0001 can be regarded as a finished
        % job.
        % this part of the code would ensure validity when two or
        % more jobs finished at the same time
        %
        if ~(job_list(i,2) < 0.0001)
            job_list_ = [job_list_;job_list(i,:)];
            queue_length_ = queue_length_ + 1;
        end
    end
    job_list = job_list_;
    N = N + (queue_length - queue_length_);
    queue_length = queue_length_;
end
```

iii) Next departure time

```
    %
    % update next_departure_time
    %
if queue_length
    almost_gone_job = Inf;
    for i=1:queue_length
        if almost_gone_job > job_list(i,2)
            almost_gone_job = job_list(i,2);
        end
    end
    next_departure_time = master_clock + almost_gone_job/(1/queue_length);
else
    next_departure_time = Inf;
end
```

In this code piece, almost_gone_job is the smallest left workload of a job among jobs, therefore, almost_gone_job/(1/queue_length) is the estimated time which still be needed to finished this job.

The result of the simulation

One of the detailed results of the simulation has been provided in "mean_response_time_log.pdf" and "job_list_log.pdf", when using random setting saved in "saved_rand_setting.mat". With respect to this simulation, a conclusion can be given that when the job routine follows the job list in "job_list_log.pdf", mean response time will be smallest when 9 of 10 servers are operating and the minimum is 6.3232. Therefore, for this job routine, 9 servers should be switched on to achieve the greatest efficiency.