CS-350 - Fundamentals of Computing Systems
Homework Assignment #2

Due on September 29, 2020 at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

Problem 1

You just bouth a fresh new server from a eBay auction from the user *noSnakeOil*. To your surprise, the server is affected by intermittent, random reboots. You have computed the MTBF as 5 minutes and you know that you can compute the reliability of the server (i.e. the probability that the server will be up for a continued time window of length t) as $R(t) = e^{-t/MTBF}$. The server takes 1 minute to reboot.
You want your server to process requests from users. Each request takes 2 minutes (on average) to complete. Because you know that the machine is unreliable, if the request arrives when the server is up, you log the request on disk before initiating processing. If the request arrives when the server is down, the request is lost. Moreover, if a reboot occurs in the middle of processing, the whole progress (but not what has been logged on the disk) is lost and the request needs to be re-computed from scratch. Whenever a request has been completed, the result is logged on disk so that it is preserved across reboots. You can assume that the data on the disk is always correct.
First off, figure out the following:

a) What is the probability that a new request will be lost?

Ans-1a P(a new request is lost) = P(a req arrives when system is unavailable) = 1- Availability of the server
Now $Availability = \frac{MTBF}{MTBF+MTTR} = 5/6$
Thus P(a req is lost) = 1 - 5/6 = 1/6

b) How many subsequent requests on average will be correctly received (but not necessarily processed!) before one is lost?

Ans-1b This can be represented as a Geometric Distribution of n subsequent requests. Let p = 5/6 = system is available = prob of success.And 1-p = 1/6 = prob of system not available.

Let μ = Mean Number of correctly received requests = $\frac{p}{1-p} = 5$

c) What is the probability that a correctly received request will complete processing the first time the system attempts to process it?

Ans-1c $R(2) = \exp^{-2/5} = 0.67$

To cope with the unreliability of your server, you devise a limited retrial strategy to process user requests. Specifically, you select an integer parameter $N \in 1, 2, \ldots$. Then, for each correctly received request, you attempt processing up to N times before rejecting the request. For instance, with $N = 2$, you try to process a given request once. If the server reboots in the middle, you try again to process the request after reboot. If it reboots again, you reject the request. With $N = 3$ you try processing up to three times and so on.
Now, answer the following:

d) What value should you set the parameter N to make sure that a correctly received request is correctly processed (i.e. not rejected) with 95% probability.

Ans-1d P(req completes processing within N attempts in total) = 0.95
1- P (req doesnt complete processing with N attempts) = 0.95.
Now the P(req doesnt complete processing 1 time (in 2min)) = 1- R(2) therefore
P(req doesnt complete processing N times) = $[1 - R(2)]^N \implies$
P(req completes processing within N attempts in total) = $1 - [1 - R(2)]^N = 0.95$. Solving for N with R(2)=0.67 gives us N = 3.

e) Assuming that you set $N = \infty$, how many times on average (including the last successful attempt) processing of will be attempted on a given request.

Ans-1e Mean of total trials for a Geometric distribution = $\mu = \frac{1}{1-p}$ where p= 1-R(2).
Thus $\mu = 1/R(2) = 1/0.67 = 1.49$.

Unhappy with the current strategy, you devise an additional check-pointing strategy instead. That is, once your server has processed half of the request — i.e. on average after 1 minute of processing — the partial progress is saved to disk (check-point). This way, if a reboot occur while processing any *half-request*, you only need to re-process the half that failed, if the first half was successfully processed. In this case, N is considered independently on the two halves of a given request. For instance, if $N = 2$, the first half will be attempted twice, and so will the second half.

f) Is it true that the two strategies are identical when $N = 1$? Motivate your answer.

Ans-1f Yes the two strategies are identical for N=1 as there are no retrials thus P(req processes correctly for strategy 1) = R(2) = 0.67 and P(req processes correctly for strategy 2)=R(1)*R(1)=0.67.

g) Is it possible to set $N = 2$ and still achieve the same quality of service obtained with the initial retrial strategy and $N = 3$ (case d))? Motivate your answer.

Ans-1g 2nd Strategy, N=2: we know that R(1) = $\exp^{-1/5} = 0.82$. Thus P(req completes within 2 attempts for each half) = 1 - P(req fails with 2 attempts for any one half) = $1 - [1 - R(1)]^2 = 1 - (1 - 0.82)^2 = 0.968 = 96.8\%$
1st Strategy, N=3: we know P(req completes within 3 attempts) = 1 - P(req fails with 3 attempts) = $1 - [1 - R(2)]^3 = 0.964 = 96.4\%$.
Thus Strategy 2 is slightly better with checkpoints as the interval is smaller for checking the reliability of any 1 half.

Problem 2

Take a look at how CodeBuddy works. New submissions are queued up in their order of arrival and then processed one at a time. Fast-forward to a future when Renato has implemented a feature that displays the average inter-arrival time between student submissions and that such value is 45 seconds. Figure out the following about the performance of CodeBuddy.

a) What is the throughput of the CodeBuddy system?

Ans-2a At steady-state: Rate of request arrival (λ) = Rate of request departure from the system = throughput of the system in steady-state. Therefore $\lambda = \frac{1}{inter-arrivaltime} = \frac{1}{45} = 0.022 req/s$

b) What is the formula for the probability distribution of the inter-arrival time of requests to the CodeBuddy system.

Ans-2b We assume the inter-arrival times follow exponential distribution. The PDF is therefore: $PDF = f(x) = \lambda \exp^{-\lambda x}$. Plugging in the value of λ: $f(x) = 0.022 \exp^{-0.022x}$

c) What is the probability that two subsequent submissions will be separated from one another by more than 1 minute?

Ans-2c $P(t > 60s) = 1 - P(t \leq 60s)$ = 1-F(60) = $1 - (1 - \exp^{-0.022*60}) = \exp^{-1.32} = 0.27$

d) What is the standard deviation of the inter-arrival time of requests to the CodeBuddy system?

Ans-2d Standard deviation of exponentially distributed inter-arrival times = $1/\lambda = 1/0.022 = 45.5$

e) What is the formula for the probability distribution of the number of requests arriving to the CodeBuddy system in a period of 120 seconds.

Ans-2e The number of requests are Poisson distributed. For an interval length of 120s, the parameter for such a distribution is $\lambda * 120$. Therefore the PMF of the number of requests is: $PMF = f(x) = \frac{(0.022*120)^x \exp^{(-0.022*120)}}{x!} = \frac{(2.64)^x \exp^{-2.64}}{x!}$

f) What is the probability that more than 3 submissions will be received within a 120-second period of time?

Ans-2f $P(x > 3) = 1 - P(x \leq 3) = 1 - [P(x = 3) + P(x = 2) + P(x = 1) + P(x = 0)]$
$= 1 - 0.728 = 0.272$

Finally, you notice that CodeBuddy reports (i) the average response time over all the completed submissions which is 30 seconds and (ii) the average service time for each request which turns out to be 20 seconds.

g) What is the utilization of CodeBuddy?

Ans-2g $\rho = \lambda * T_s = 0.022 * 20 = 0.44$

h) If you were to take a timed average of the number of submission queued and waiting to be run (i.e. excluding the one, if any, currently running), what would this number be?

Ans-2h $w = \lambda * T_w = \lambda * (T_q - T_s) = 0.022(30 - 20) = 0.22$ requests.

Problem 3

A multi-CPU processing cluster is dedicated to handling the user traffic on the TrickTruck social network. At peak popularity and at steady state the cluster is handling, on average, about 14,000 active requests. You have also measured that new requests are generated by the users at a rate of 4,200 requests per second, and that each one of them (once it starts processing) takes on average 3 milliseconds to complete.

a) What is the latency of the processing cluster perceived by the TrickTruck users?

Ans-3a $T_q = q/\lambda = 14000/4200 = 3.33s$

b) What is the slowdown of the server? That is, how much slower is the system because of the volume of traffic it is handling compared to the case where the system is under very light load?

Ans-3b Assuming an M/M/1 system: slowdown $= T_q/T_s = 3.33/0.003 = 1110$

c) Assuming that the requests are mostly CPU-intensive, and considering how busy is the server, how many CPUs the server must have at the very least?

Ans-3c $\rho = \lambda * T_s = 4200 * 0.003 = 12.6$. Since for an M/M/1 system $\rho \leq 1$ thus Number of CPUs $= 13$.

d) What is the utilization of each of the CPUs in server, assuming it has no more than the strictly required number of CPUs determined in Question c)?

Ans-3d $\rho_{cpu} = \rho_{server}/13 = 12.6/13 = 0.969$

e) Sticking to the same number of CPUs, how many requests per seconds must each CPU be processing?

Ans-3e Requests per sec for each CPU $= \lambda/13 = 4200/13 = 323.08$

f) What assumptions did you have to make to answer the above questions?

Ans-3f System in steady-state, System is M/M/1, Load is equally balanced between CPUS, CPUs are identical

Problem 4

Coding Assignment: In this problem you will develop code to simulate a single-server queue, where the time to process a request at the server is modeled using an exponential distribution, while the arrival process of the requests follows a Poisson distribution (Hint: *recall that if an arrival process is Poissonian, then the inter-arrival time between requests is exponentially distributed*).

a) Write a Java class called "Simulator" that implements a discrete event simulator. The simulator should have a method with the following prototype: `void simulate(double time)`, that simulates the arrival and execution of requests at a generic server for `time` milliseconds, where `time` is passed as a parameter to the method. The class should be implemented in its own java file, named `Simulator.java`. The simulator class will internally use a exponentially-distributed random number generator. For this, you can re-use the Exp class you wrote as part of the previous assignment. **Please be mindful of the capitalization in all the files you submit.**

   Apart from implementing the `simulate(...)` method, the class should also include a `main(...)` function. The `main(...)` function should accept 3 parameters from the calling environment (in the following order):

   (a) length of simulation time in milliseconds. This should be passed directly as the `time` parameter to the `simulate(...)` function.

   (b) average arrival rate of requests;

   (c) average service time at the server;

   If you are not familiar with how to pass parameters (a.k.a. command-line arguments) from the calling environment to a Java program, take a look at this: `https://www.journaldev.com/12552/public-static-void-main-string-args-java-main-method`.

   It is responsibility of the `main(...)` function to internally invoke the implemented `simulate(...)` function **only once**. In this first version, the `simulate(...)` function will need to print in the console the simulated time at which each request arrives at the system (`ARR`), initiates service (`START`), and completes service (`DONE`). The output must look like this:

```
R0 ARR: <timestamp>
R0 START: <timestamp>
R1 ARR: <timestamp>
R2 ARR: <timestamp>
R0 DONE: <timestamp>
R1 START: <timestamp>
R1 DONE: <timestamp>
R2 START: <timestamp>
R2 DONE: <timestamp>
```

   where `<timestamp>` is the simulated time in milliseconds at which the event occurred printed in decimal format. This is also called the **trace** of the simulation. **Be extremely careful to follow the format described above. CodeBuddy will be very sensitive to output formatting issues.**

b) Modify the code of the simulator written in the first part of this problem to measure utilization, average queue length, and average response time of requests. The simulator remains the same in terms of interface, accepted parameters, and simulation logic. However, in addition to printing out the simulation trace just like before, the simulator should add three lines at the end of its output, i.e. after the trace. This should be formatted as:

Problem 4 continued on next page...

```
UTIL: <utilization>
QLEN: <avg. queue length>
TRESP: <avg. response time of requests>
```

where `<utilization>` is a decimal number between 0 and 1, `<avg. queue length>` is a decimal number, and `<avg. response time of requests>` is a decimal number expressed in milliseconds.

Submission Instructions: in order to submit this homework, please follow the instructions below for exercises and code.
The solutions for Problem 1-3 should be provided in PDF format, placed inside a single PDF file named `hw2`.pdf and submitted via Gradescope. Follow the instructions on the class syllabus if you have not received an invitation to join the Gradescope page for this class. You can perform a partial submission before the deadline, and a second late submission before the late submission deadline.

The solution for Problem 4 should be provided in the form of Java source code. To submit your code, place all the `.java` files inside a compressed folder named `hw2`.zip. Make sure they compile and run correctly according to the provided instructions. The first round of grading will be done by running your code. Use CodeBuddy to submit the entire `hw2`.zip archive at `https://cs-people.bu.edu/rmancuso/courses/cs350-fa20/scores.php?hw=hw2`. You can submit your homework multiple times until the deadline. Only your most recently updated version will be graded. You will be given instructions on Piazza on how to interpret the feedback on the correctness of your code before the deadline.