CS-350 - Fundamentals of Computing Systems
Homework Assignment #3

Due on October 5, 2022 — Late deadline: October 7, 2022 EoD at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

Problem 1

Your job is to dimension a network switch for ultra-high-speed communication. According to your evaluations, the network packets that will be transmitted on this network are exponentially distributed in size with an average of about 1.2 KB (1 KB = 1024 bytes). The switch will need to handle an average traffic of 8200 packets per second. Initially, you decide to go cheap on the design on the switch and use commercial-off-the-shelf (COTS) Ethernet technology, where the maximum bit transmission speed is 100 Mbps (10^8 bits per second).

a) Packets will be held in memory by the switch while queued and while being processed. How much memory (in bytes) does the switch need to operate such that *most of the time* no packets are dropped due to memory exhaustion?

b) If a timestamp is taken for each packet when they enter the system and another one when they begin service, what will be (on average) the difference between the two timestamps?

c) The COTS Ethernet chip you want to use requires to be cooled down with a fan if for every 10 seconds of operation, the chip is busy for 7 or more seconds. Do you need to add a fan in your design?

d) What assumptions have you made to solve the system in all the steps above?

Your colleagues have reviewed your design and are bringing something you did not consider to your attention. That is, the switch is too slow and will be the bottleneck of the entire network. So here is your plan. You propose to use Gigabit Ethernet technology instead, making your switch capable of processing packets at 1 Gbps (i.e. 10^9 bits per second). Respond to your colleagues by computing the following:

e) What will be the response time speedup compared to your previous proposal (i.e. standard Ethernet)?

f) Given how expensive are Gigabit Ethernet chips, can we lower the amount of memory to hold in-flight packets in the switch? And by how much?

Problem 2

At the Instakilo social network, queries for user profile bios are handled by a dedicated two-stage subsystem. Requests for bios first arrive at a user-facing authentication server. After authentication, they are sent to a backend server. Once they complete handling at the backend server, the response is provided to the users. Thanks to the well-behaved user-base all requests always pass authentication. The authentication server is capable of handling up to 10 requests per second. The backend server, instead, is capable of handling 12 requests per second. But unfortunately, a request at the backend requires a lot of memory and to prevent the server to fail due to memory exhaustion, the engineers have imposed a hard cap on the number of requests that can be admitted at the backend. The limit has been set to 5. Any request arriving while 5 are currently being handled at the backend is rejected and an error response is returned to the user.

a) What is the maximum traffic that the system can handle while still being able to reach steady-state?

b) From now on, consider an input traffic of 8 requests per second. What is the total average number of requests that is either queued or currently being processed in the entire system?

c) When a request is completed at the first stage and then rejected at the second stage, the computation time used to process the request at the first stage is wasted. Over an interval of time of 100 seconds, how much computation time is wasted on average in the system under analysis at steady-state?

d) What is the average response time for requests that are correctly handled (i.e., not rejected)?

e) What is the average response time for a generic request, rejected or not? Assume that the error response to the user is immediately produced if their request is rejected at the second stage.

Problem 3

Breaking news! Your multi-million grant to seed your *CheckMeOut* startup. The idea is the following. Your users upload a selfie and the clever CheckMeOut algorithm checks that everything looks in order. The modern-day version of a pocket mirror. But with an unforgiving computerized eye ready to (constructively) criticize you. For instance, forgot to shave? *CheckMeOut:* "Have you lost your razor?"; Pillow marks on your cheeks? *CheckMeOut:* "Them pillow marks make me yawn.". And so on.
With these funds at hand, now it is time to perform system tuning. After observing the system for a while you have noticed the average picture size submitted by the users is 3 MB (1 MB = 1024 KB; and 1 KB = 1024 bytes). Every day, you are receiving about 26,000 CheckMeOut requests that you process in order of arrival. Your want to know the following:

a) How long should your system spend processing each individual request (on average) so that the system ends up handling, on average, about 20 requests—either being currently processed or queued?

b) Assume that you were able to tune your system so that processing each request takes about 2.5 s, what is the time that each user will end up waiting on average before receiving their CheckMeOut response?

c) How would the average response time change if you decided to switch to SJN policy instead of serving requests in their order of arrival?

d) How would the average response time change if you determined that the processing time for each request follows a Normal distribution with mean 2.5 s and standard deviation 0.5 s?

From now on, consider the system tuned according to what mentioned in Question b). That includes exponentially distributed service times, with the server employing a FIFO dispath policy.

e) What is the slowdown of the system due to queuing?

f) What is the maximum number of requests that the system can sustain on a daily basis?

g) How much memory will the requests in the system occupy on average, considering that (1) queued requests occupy as much memory as the size of the input image; and (2) requests currently being processed occupy 3× as much until their processing is completed.

The federal agency providing the funds got back to you. They say that you must support way more users per day or the deal is off! Overnight, you devise the following optimization. You organize your system in two stages. At the first stage, pictures are compressed down to 450 KB. After compression, they are sent to the main processing server for CheckMeOut caption generation. This way, the final processing stage can take as little as 1.1 s. Now you worry about the following:

h) How long should compression take so that the system can sustain 70,000 requests per day?

i) Assume that compression takes 0.9 s, what is going to be the average time it takes your whole system (compression+processing) to take a new request in and produce the corresponding output when your system reaches 70,000 requests per day?

Problem 4

Coding Assignment: In this problem you will develop code to simulate a dual-class queuing system with traffic loopback. The server you simualate accepts requests from two classes of users that send requests with different parameters. Moreover, once a request leaves the server, it *might* leave the system or go back and ask for more service. The additional amount of service time is distributed in the same was as the originally requested amount. After that, the request might go out or loop back in and so on. Request service times at all the servers are modeled using an exponential distribution, while the arrival process of all 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) Extend your Simulator class to be able to handle two independent streams of requests, of class X and Y, respectively. The Server should still be used to enqueue and process arriving requests following a given policy—FIFO or Shortest Job Next (SJN). Once again, you are free to take any implementation choice that allows the simulator to correctly handle the simulated traffic and service policies. For this part, still assume that once a request has completed processing at the server, it leaves the system immediately.

   Like before, 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, a timeline and two load generators. For these, you can re-use/adapt the classes 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 6 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 of class X (λ_x);

   (c) average arrival rate of requests of class Y (λ_y);

   (d) average service time at the server for requests of class X (Ts_x);

   (e) average service time at the server for requests of class Y (Ts_y);

   (f) queuing policy which can either be "FIFO" or "SJN";

   (g) probability p_{out} of leaving the system after completion (set to 1 for this part).

   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 of class X or Y arrives at the system (`ARR`), initiates service (`START`), and completes service (`DONE`).

   To generate a sequence of `ARR` events with the right inter-timing, reuse the loadGenerator approach from HW1. In this case, you will have two LoadGenerator instances, for class X and Y requests, respectively. When their releaseRequest(...) is called, TWO things happen: (1) just like before, generate the next arrival using λ_x or λ_y; and (2) create a new Request object with length sampled from an exponential distribution with parameter $1/Ts_x$ or $1/Ts_y$ and pass it to the Server. Thus, the Server class should have some receiveRequest(...) method that can be called by the LoadGenerator for this purpose. At the end of the simulation, still print out global statistics for the utilization (UTIL), average system

population (QAVG), and average requests waiting (WAVG). Unlike before, compute average waiting time (TWAIT) and average response time (TRESP) separately for the two X and Y request classes. See output below.

Under the FIFO policy, the order of service is the same as the order of arrival. In this case, the output must look like this:

```
X0 ARR: <timestamp> LEN: <length>
X0 START: <timestamp>
Y0 ARR: <timestamp> LEN: <length>
X1 ARR: <timestamp> LEN: <length>
X0 DONE: <timestamp>
Y0 START: <timestamp>
Y0 DONE: <timestamp>
X1 START: <timestamp>
X1 DONE: <timestamp>
...
UTIL: <utilization>
QAVG: <avg. number of requests in the system>
WAVG: <avg. number of requests waiting for service>
TRESP X: <avg. response time of requests of class X>
TWAIT X: <avg. time spent in the queue by X-class requests>
TRESP Y: <avg. response time of requests of class Y>
TWAIT Y: <avg. time spent in the queue by Y-class requests>
```

where `<timestamp>` is the simulated time in milliseconds at which the event occurred printed in decimal format, and `<length>` is the service time of the request. **Be extremely careful to follow the format described above. CodeBuddy will be very sensitive to output formatting issues.**

Under the SJN policy, if two or more requests are queued, the one with the shortest service time is picked for execution. Thus, if SJN is selected and assuming that `<x1_length>` < `<y0_length>`, the correct output will be:

```
X0 ARR: <timestamp> LEN: <length>
X0 START: <timestamp>
Y0 ARR: <timestamp> LEN: <y0_length>
X1 ARR: <timestamp> LEN: <x1_length>
X0 DONE: <timestamp>
X1 START: <timestamp>
X1 DONE: <timestamp>
Y0 START: <timestamp>
Y0 DONE: <timestamp>
...
UTIL: <utilization>
QAVG: <avg. number of requests in the system>
WAVG: <avg. number of requests waiting for service>
TRESP X: <avg. response time of requests of class X>
TWAIT X: <avg. time spent in the queue by X-class requests>
TRESP Y: <avg. response time of requests of class Y>
TWAIT Y: <avg. time spent in the queue by Y-class requests>
```

b) Modify the code of the simulator written in the first part of this problem to handle possible loop-back of requests. The simulator remains the same in terms of interface, accepted parameters, and simulation logic. However, whenever a request completes service, you should flip a coin. With probability p_{out} (6-th parameter), the request actually completes at the system and therefore leaves for good. Otherwise, it goes back into the server queue. It keeps the same ID and requests an additional amount of service (length) distributed identically to new requests of the same class. The new output is similar to what you had before. There are only two differences. First, if a requests leaves the system for good, the you still print `YO DONE: <timestamp>`. But if it goes back in, you should print `YO LOOP: <timestamp>` instead. Second, you should keep track of how many times a generic request (regardless of the class) asks for service and compute the average number of times a request demands service. We call this a RUN. A brand new request always asks for a first RUN, then it might ask for one more RUN or exit and so on. Thus, statistics will need to be reported as:

```
UTIL: <utilization>
QAVG: <avg. number of requests in the system>
WAVG: <avg. number of requests waiting for service>
TRESP X: <avg. response time of requests of class X>
TWAIT X: <avg. time spent in the queue by X-class requests>
TRESP Y: <avg. response time of requests of class Y>
TWAIT Y: <avg. time spent in the queue by Y-class requests>
RUNS: <avg. number of times a request demands service>
```

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 `hw3.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 `hw3.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 `hw3.zip` archive at `https://cs-people.bu.edu/rmancuso/courses/cs350-fa22/codebuddy.php?hw=hw3`. 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.