# CS-350 - Fundamentals of Computing Systems
# Homework Assignment #6 - EVAL

Due on October 31st, 2024 — Late deadline: November 2nd, 2024 EoD at 11:59 pm

*Prof. Renato Mancuso*

**Renato Mancuso**

# EVAL Problem 1

In this EVAL problem, we will study the actual request lengths that are now unknowns that depend on the interaction between the workload created by the client and the server operating on the hardware at hand.

a) Alright folks, first off, congratulations on implementing your first REAL server that's doing an actual job: Image Processing. Welcome to the fun part: You get to test out and see what happens to your image after a few manipulations.
For this, you will need to add a single line right before you are about to RETRIEVE the image. When retrieving, right before you send your image back to the client, you can save your manipulated image to see what has happened to it. You have access to the following function:
**uint8_t saveBMP(const char* filename, const struct image* img)**
This function is defined in your imglib.c file and you will be able to call this directly in your server_img.c
To save the image, call this right before RETRIEVING and you'll be able to save the manipulated image. Save it and attach this image to your EVAL document and describe what has happened to the original image from your observation as well as the client requests, while running the following parameters:
run: ./build/server_img -q 100 2222 & ./client -a 30 -I images/ -n 5 2222 (this works provided that your server object file is in the build while the client that we provide to you is outside of the build as requested :) )

b) For this part, we are interested in the distribution of request lengths as measured by the server on your machine.
Here, it is immediately important to remember that your results might wildly vary when you compare them with those obtained by your colleagues, because you will be running your code on very different machines. The only good point is that now you can compete with your friends about who has the best machine!
To carry out these experiments, run the following command TWO times.
The first time, run: ./build/server_img -q 100 2222 & .//client -a 30 -I images_small/ -n 1000 2222 (NOTE: if your local machine (in case you are not using the SCC) is too weak—#noshame—and starts rejecting requests with the parameters above, you can reduce the arrival rate of requests sent by the client, but keep the same total number -n 1000).
In the images_small only keep the two smallest images, i.e. test1.bmp and test2.bmp. Collect the output of the server and client, and set them aside.
Next, run:
./build/server_img -q 100 2222 & ./client -a 30 -I images_all/ -n 1000 2222
In the images_all folder, keep all the images in the dataset. Once again, Collect the output of the server and client, and set them aside.
Post process the outputs from RUN1 and RUN2 and for each run, produce a plot of the CDF (with average and 99% tail latency, as we did in HW5) of the amount of time taken by EACH image operation, but exclude IMG_RETRIEVE—we do not trust the timing reporting of that one! So you should have 6 plots per run, one per operation for a total of 12 plots. Keep them small enough so that you can fit all of them in a single page, maybe 3 plots per row. Here I strongly encourage you not to generate each plot manually, but rather use some form of scripting.
Now look at the plots, and answer the following: (1) within the same run, what operations are similar in behavior and which ones behave differently? (2) Within the same run, which ones are the least predictable operations? (3) Across runs, by how much does the average response time increase for each operation? (3) Across runs, by how much does the 99% tail latency increase for each operation?

c) For this part, no need to run new commands! Yay! Instead, retrieve the output you generated in the previous part. The one for RUN2 in particular.
As you parse the output produced by your server, construct an online estimator for the length of each type of request. In other words, consider each type of image request (e.g., IMG_ROT90CLKW vs. IMG_HORIZEDGES etc.) as a separate task. Every time you observe an operation of a given type, consider the measured length as a new job length sample. Build an exponentially weighted moving average estimator (EWMA) with parameter $\alpha = 0.7$ for each request type. Use the estimator to predict the length of the next operation of the same type.
So, say you have observed n operations of type IMG_ROT90CLKW, you will construct the estimator $\bar{C}(n)_{\text{IMG ROT90CLKW}}$ for the $(n+1)^{th}$ operation of the same type.
For each request type, compute the misprediction error as the absolute value of the difference between your prediction and the observed $n^{th}$ job length. Take an average of that error and reason on how good your predictions are for each image operation (i.e., for each task).


Problem 1 continued on next page...

d) Okay, here we need to re-run some experiments. First off, set aside the output you produced for RUN2 in part (a). Now, take a look at the Makefile included in the template files and locate the following line: LDFLAGS = -lm -lpthread -O0. Focus on the -O0 (minus capital "o" zero).
First, set that to -O1, then run the command make clean (careful! this will remove the entire build folder), and then make again to recompile the server and all the libraries from scratch. Next, rerun the same command you used for RUN2 (the one with all the images) in part (a).
Now, do everything once again with -O2. At this point, you should have 3 outputs in total. Post-process those outputs to generate 1 plot per output depicting the CDF (with average and 99% tail latency) of the length of any image operation. In other words, do not differentiate by operation type, but treat all the operations from the same output as if they were all just generic image processing operations. Thus, you should just produce 3 plots, one for the -O0 case, one for the -O1 case, and one for the -O2 case.
First, try to figure out what the -O<value> flag is supposed to do in theory by looking up (or remembering from CS210) what that is. Next, with plotted data at hand, answer the question of whether or not that flag is behaving as it is supposed to.