Week 5 Assignment - Data Structures

Overview

This week's assignment will focus on using one of the data structures from our reading, the Queue. In this week's reading, there was a discussion about writing a program to simulate a printer and its print queue. The simulation would determine when print tasks would be generated, placed them on the printer's queue to eventually be processed by the printer.

This week, we will do something similar. We will write a network simulator to simulate how network requests are processed by a single web server, and a group of web servers behind a *load balancer*. We will see how the average wait time, or *latency*, is related to the server processing rate.

Make sure to create a github repository for this assignment named **IS211_Assignment5**. All development should be done in this repository.

Useful Reminders

1. Read the assignment over a few times. At least twice. It always helps to have a clear picture of the overall assignment when understanding how to build a solution.
2. Think about the problem for a while, and even try writing or drawing a solution using pencil and paper or a whiteboard.
3. Before submitting the assignment, review the "Functional Requirements" section and make sure you hit all the points. This will not guarantee a perfect score, however.

Background

What exactly do we mean by request? In this case, a request is simply some user requesting a file from a web server, via a web browser. For example, when you go to www.google.com in your browser, you are making a network request to the web server hosting google.com. This request gets processed and eventually, the results are sent back to your browser to be displayed.

In this assignment, we will look at how we can simulate this process, representing the server as a queue of requests that need to be processed. In the printer example, the printer can print *x* number of pages per second. In our case, each request will have its own 'processing time', which is the amount of time that is needed for the server to process this request. Also, in the real world, a server can process more than one request at a time, since servers typically have more than one CPU. However, just like the printer simulation, we can assume for now that the server only processes on request at a time.

In essence, this assignment is very similar in structure to the printing simulation. Therefore, it is highly recommended to read over the printer simulation section again. This biggest difference between this and the printing simulation is that we will not be generating requests at random, but reading them from a file. This will help keep results consistent and easily testable.

Part I - Implement Simulation with One Server

To help outline a solution, you will have a ***Server*** class and a ***Request*** class. This are akin to the Printer and Task class in the reading. You will need to update these classes to represent the problem at hand.

The file that is used for input will be a CSV file in the following format (an example file can be located here **TODO**):

7, /images/test.gif, 1
7, /images/header.jpg, 1
8, /home, 2

So, the first line represents a request that came in for the /images/test.gif file at the 7th second of the simulation, and will take 1 second to process. This is an important point: each request tells you which second of the simulation its being generated for. This is unlike the printer simulation, where this was done randomly. You need to update the ***Request*** class accordingly. Your ***main()*** function should read this file in to get the list of requests being made, and to instantiate your Request objects. This means that you do not need an equivalent function for ***newPrintTask()***.

You will also need to implement the ***simulateOneServer()*** function, which should take in the input filename. The ***simulateOneServer()*** function is responsible for printing out the average wait time for a request (i.e., how long, on average, did a request stay in the server queue before being processed). The simulate function should return this average.

Your main function should accept only one parameter, *--file*, which points to the file to read for request inputs. This ***main()*** function should call your ***simulateOneServer()*** function. Make sure to put all this code in a file called *simulation.py*.

Part II - Extending the SImulation

The simulation should now be extended to include what is known as a load-balancer. In the real world, typically we have more than on server handling requests.

Update the ***main()*** function to take another parameter, *--servers*, which is an integer representing the number of servers. If the parameter is not there, you should run ***simulateOneServer()***; otherwise, you should run your new code in ***simulateManyServers()***, passing in the number of servers to simulate***.*** This new function will be responsible for maintaining a queue for each server, and sending requests to these many servers in a *round-robin* fashion. For example, if you have two servers, then the first request goes to the first server, the second request to the second server, the third request goes to the first server, etc. The ***simulateManyServers()*** function should still return the average latency for the requests.

Food For Thought: Can you come up with different schemes besides Round Robin on how to distribute the requests to many servers?

Part III - Results

After running both simulations, what are the results do you see? How does each simulation differ in its average latency?