













MMN ASSIGNMENT REPORT: SCHEDULING SIMULATOR



INDEX

Introduction 3
MMN queue simulation definition 3
Supermarket model 3
MMN with Ubuntu scheduling?? 3
Analysis and definition of various parameters 4
Empirical vs theoretical graphs 5
Analysis of Temporal Performance and Queue Length in the Supermarket Model6
Weibull distribution 6
My RR -> Results and plots 7
Confronting Weibull and Round Robin graphs8
Appendix ... 10
... (or how to run the code) 11
Peer Review changes done 11
Conclusions  <small>END</small>12

Introduction

In the following document I will analyze and discuss the results obtained from my choices of implementation in this assignment.

MMN queue simulation definition

Nowadays queueing systems play a crucial role in modeling the majority of scenarios, e.g. servers, networks and customer services. Simulating them allows us to analyze and understand their behavior under different conditions and variables, so that we can change parameters on the go to optimize the system or improve it; i.e. the purpose of this assignment.

Supermarket model

It is self-explanatory, in the sense that it behaves like us when we are doing groceries: it chooses the minimum loaded queue (in our case, server) between *random* sorted d queues, then the job will be scheduled to it. As performant as it is, all the plots where $d > 1$, the supermarket model has been applied.

MMN with Ubuntu scheduling??

My initial idea for the extension was to implement the actual scheduling of Ubuntu. After researching it I discovered that it uses:

1. FIFO if the process is not critical;
2. RR for continuity critical processes;
3. Other scheduling in case of necessity.

I tried to implement them together with the following code:

```
def supermarket(self) -> int:
    # Choose d random queues by their indexes.
    indexes = sample(range(len(self.queues)), self.d)
    # Return the index of the queue with the minimum length.
    return min(indexes, key=lambda i: self.queue_len(i))

def fifo_schedule(self) -> int:
    return 0

def my_round_robin(self, job_id) -> int:
    # Choose the next queue in the round robin fashion.
    return job_id % self.n

def is_critical(self, job_id) -> bool:
    return job_id % 2 == 0
    # return True
```

In the end, the FIFO implementation does not really make sense in the context of this assignment, and, for the sake of simplicity, I adapted the code to run only **my_round_robin** if the running job was critical, choosing that priority with a 50% chance. Since there were not “slices of time”, my adaptation consisted in trying to schedule the upcoming job to a server x , the next job to the server $x+1$ and so on, like a loop. I ended up using a module operation by the number of servers:

```
def my_round_robin(self, job_id) -> int:
    # Choose the next queue in the round robin fashion.
    return job_id % self.n
```

So if there are 3 servers (e.g. 0, 1, 2), the only possible integer outputs for $\%3$ are 0, 1 and 2. In the following chapters I will analyze the resultant graphs.

Analysis and definition of various parameters 🤖

But first, let us go through the definition and behavior of all the parameters:

- **λ** = rate of job arrivals in the queueing system, Possible optimal values are (like we have seen during lectures) 0.5, 0.9, 0.95, 0.99. λ needs to be less than μ ;
- **μ** : service rate at which servers process jobs in the queueing system, crucial for defining capability and efficiency of each server of the system. An analogy could be to imagine μ as the average number of customers a cashier can serve per unit of time (e.g. customers/minute). Of course, a higher μ would represent a faster cashier;
- **n** : number of servers;
- **d** : controls the number selected for job assignment, influencing the overall job distribution strategy. If ($d==1$) then “Random Model” is obtained. It means that it will randomly select a single queue within the n available. If greater than 1, a supermarket model is obtained;
- **max_t** : represents the maximum threshold for the simulation execution. Obviously, there will be compromises between performance and accuracy;
- **$arrival_rate$** : it reflects the rate at which jobs enter the system and the handling of the incoming workload;
- **$completion_rate$** : as above, but for the completion timestamp of jobs;
- **W** is the actual empirical average time spent by a job in the system;
- **Wt** represents the theoretical expectation for the average time a job will spend in the system. Thanks to the Little’s law, we have that L equals the average queue length that equals Wt multiplied by λ ;
- **t** is a flag that, if present, tells the code to plot also the theoretical graphs, the final image could be a little overcrowded;
- **csv** is a parameter that stores the file names for saving the simulation results in, obviously, CSV format;
- **$queue-length-frequency$** : the frequency of the event scheduling;
- **e** is flag that, if present, uses the extension of the assignment (use a different scheduling than acsupermarket);

Empirical vs theoretical graphs

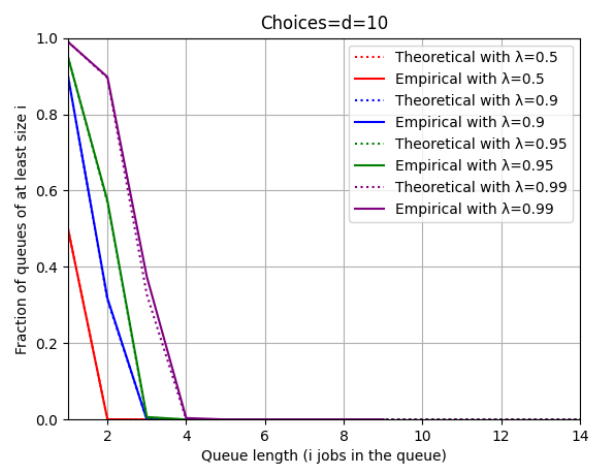
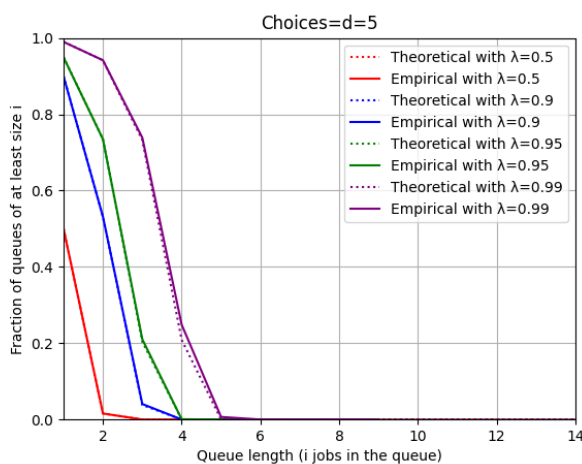
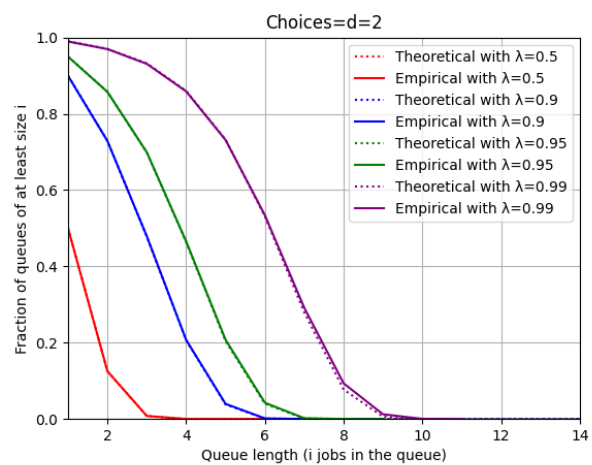
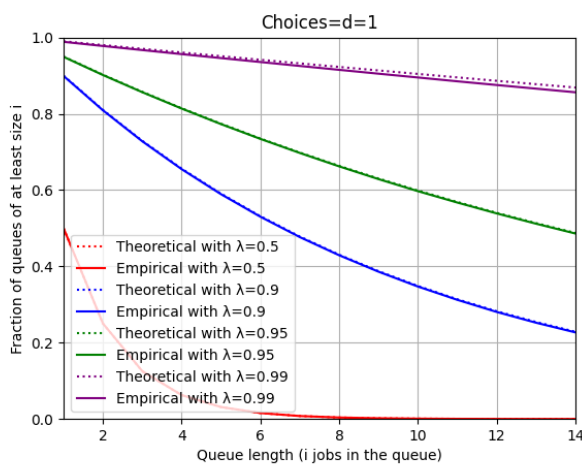
Subsequently, I wanted to examine the average queue lengths in the supermarket policy, graphing the various results (both empirical and theoretical) to visualize them effectively.

I used the above command to launch the system.

```
py u_mmn_queue.py -t --max-t 100000 --n 1000 --queue_length_frequency 50
--d <1, 2, 5, 10>
```

In particular, I used the flag -t for plotting also theoretical values and $n=1000$, $max-t=100000$ in order to have a sufficient number of servers to make sense of the experiment and to not have jobs in queue for more than a thousand seconds (a million milliseconds: around 16 minutes). The following figures show the results.

In this case, "Choices" represents the number of possible queue to choose from. On the y-axis, there is the percentage, expressed in tenths, of queues that have at least that length. For example, 10% (0.1) of all queues have at least 4 jobs.



Analysis of Temporal Performance and Queue Length in the Supermarket Model

Like we have seen during lectures, the supermarket policy brings significant improvements in temporal performance and queue length, but only if the number of servers n is sufficiently large relative to D , i.e., if $D > 1$. This implies that an $M/M/1$ queuing system is inherently inefficient. As the up-left plot demonstrates, the resulting graph traces the Random Model, where jobs are allocated to queues randomly without comparison or choice among multiple queues. This approach misses the load-balancing benefit that results from selection among multiple queues. The other three plots, however, demonstrate a proportional improvement in performance as D increases: more server selection leads to better-balanced queues and a more efficient system in general. As a consequence, when load balancing is a problem—especially in distributed systems—the Supermarket Model performs well in processing large numbers of jobs efficiently and quickly even under adverse conditions.

Another critical factor to analyze is the influence of simulation parameters. In a previous version of this report, the simulation was executed with very low parameter values, which resulted in unrealistically small data. For instance, there were too few simulated jobs or servers to reasonably test the Supermarket Model. To bridge the gap between theoretical and empirical results, I modified significant input parameters, specifically *queue_length_frequency*. The *queue_length_frequency* parameter determines the frequency of reporting queue lengths throughout the simulation. Higher levels (i.e., shorter reporting intervals) provide a full history of the system state over time, which assists in eliminating the impact of transient processes or random fluctuations that can distort empirical data. By setting *queue_length_frequency* to 100 or 50, I achieved more similarity between theoretical and empirical results, with the empirical plots closer to their theoretical counterparts.

Weibull distribution

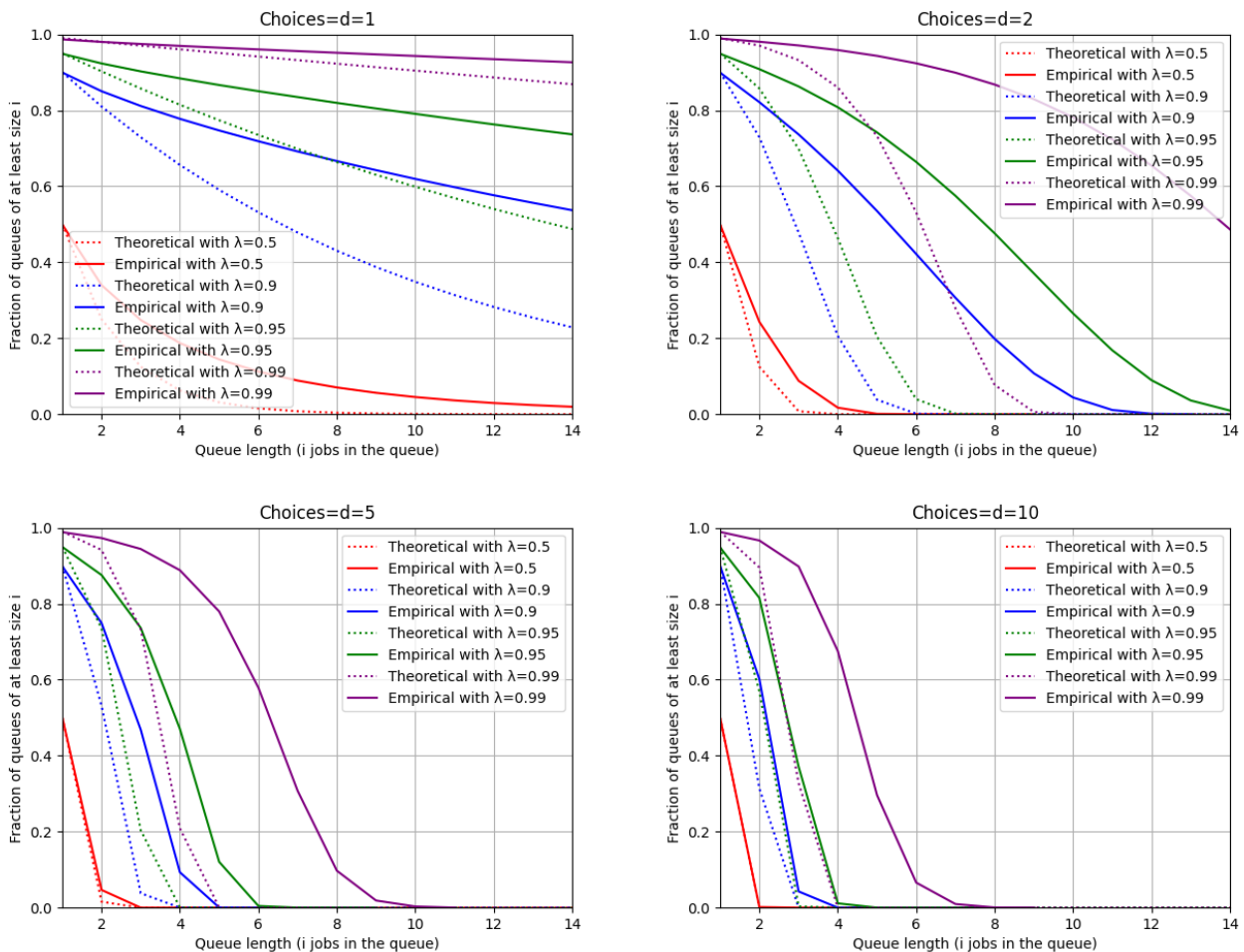
After the “peer” review, I decided to implement another type of arrival of the job scheduling. It was simply achieved by changing the original distribution to a fat-tail distribution, the Weibull one happens to be a so called “heavy” or “fat” tail distribution (when the shape parameter greater than 0 but less than 1); it means that the far-right end of the distribution is not exponentially bounded and has a much greater probability of producing extreme values, or outliers. Think about when we’re computing the mean salary of a group of friends and one of them happens to be Elon Musk: the average salary does not make sense anymore.

Having said that, I set the shape to 0.5 and launched the system with the command below.

```
py u_mmn_queue.py -t --max-t 100000 --n 1000 --queue_length_frequency 50
--d <1, 2, 5, 10> --shape 0.5
```

Note that, for *shape* > 1, instead, the results become more uniform (Bell curve) and, obviously, for *shape* = 1 we obtain the same results and behavior as expovariate.

The resultant plots and graphs are the following.

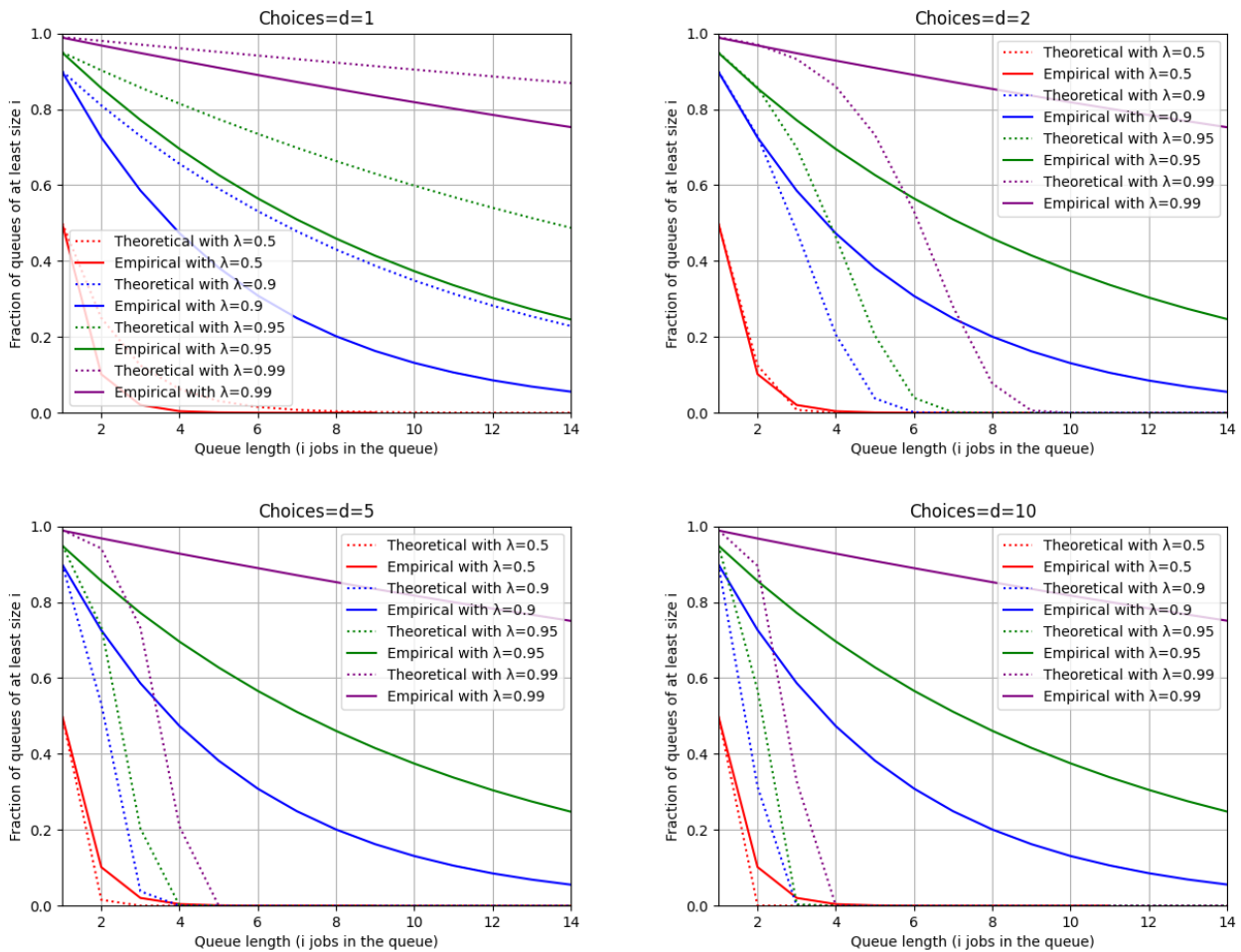


The dotted lines are the theoretical ones without the Weibull computation, while the continuous lines are the system results using the Weibull distribution.

Since I plotted them together, the difference between the two is very spottable. Indeed, the Weibull distribution makes the arrival of the jobs less homogeneous, as by definition. In particular, many jobs arrive at the beginning of the runtime, or the, opposite, very much later. The unpredictability can be seen on the figure above: the every plot has a bigger queue length (the Weibull graph is heavier to the right than the dotted theoretical lines) and analyze much more volume (**heavy-tail**). That explains the importance of the supermarket scheduling, especially with values of d much bigger than 10, so that the load between queues is more distributed.

My RR -> Results and plots 🧐

In the following figures, there are the resultant graphs of four execution with my interpretation of the round robin scheduling in the context of queueing systems and of this assignment, as explained before.



I immediately noted that the “round robin” graphs are flatter (because it spreads the load evenly across all queues without considering their current lengths). That leads to a more uniform distribution of queue lengths (flatter graphs).

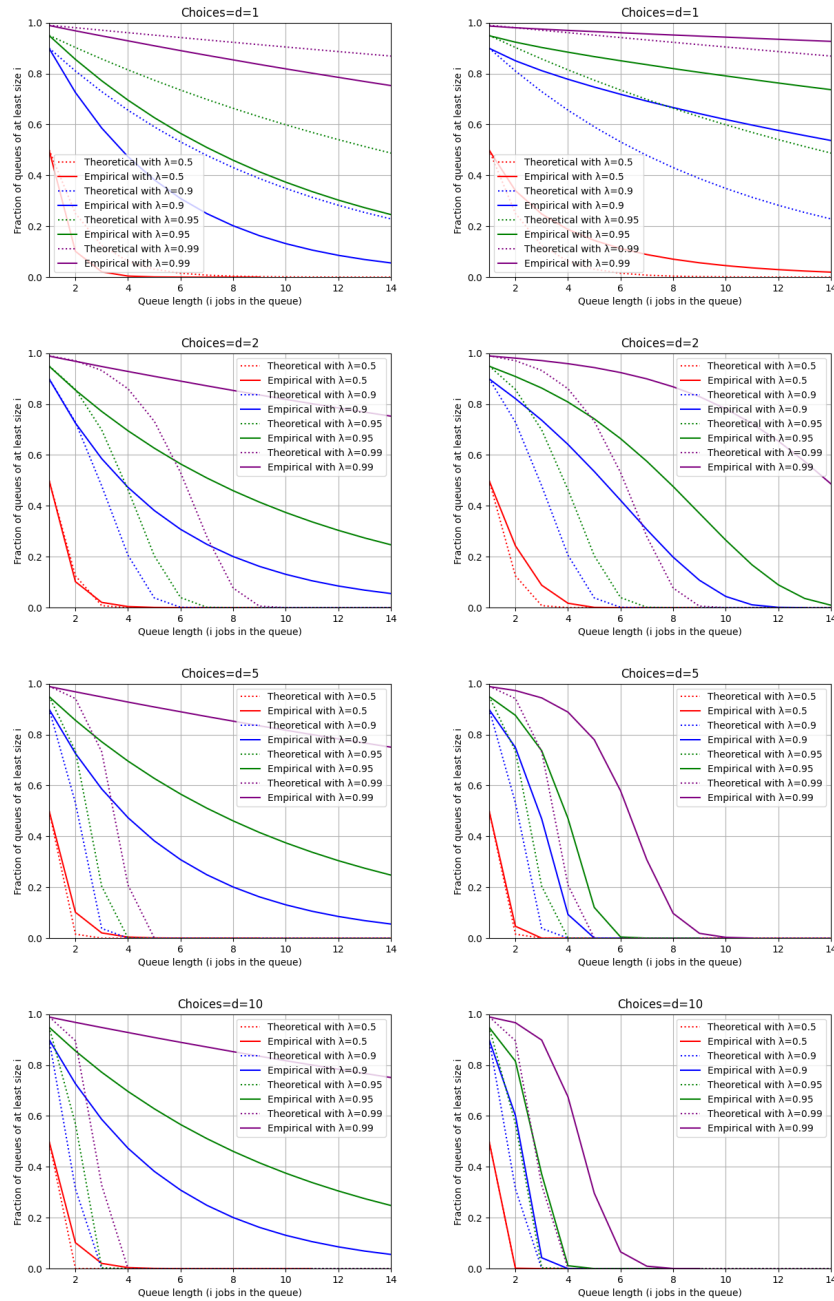
Secondly, since the Round Robin method doesn’t favor shorter queues, all queues tend to grow at a similar rate and the chance that queues have lengths closer to the average is higher. Hence why the fraction of queues decreases more gradually.

In the end, in scenarios where load balancing is critical to maintaining system performance, especially in distributed environments, round robin is obviously not the best choice.

Confronting Weibull and Round Robin graphs

Having analyzed both the Weibull distribution and the Round Robin scheduling graphs, we note an interesting contrast between the two: the Round Robin algorithm produces flat graphical plots as a result of its approach of uniformly distributing jobs across all queues regardless of their present lengths, leading to a more uniform growth in queue lengths and a slow decrease in the percentage of queues with larger lengths. In contrast, the Weibull distribution, especially when the shape parameter is less than 1, produces heavy-tailed job arrivals, leading to large variability in queue lengths over time. This effect causes a rightward shift in the graph, where long tails represent long

queue lengths and high volumes compared to the theoretical model. Although the basic mechanisms of how they operate differ fundamentally, both methods can produce visually similar plots under some conditions.*



Round Robin


Supermarket w. Weibull
Distribution

In conclusion, the supermarket model alone performs better than the Round Robin scheduling and the Supermarket model with Weibull distribution in managing queues. When $d \geq 10$, it balances loads effectively, minimizing queues and maximizing performance in heavy loads. Greater d values produce more balanced queues and performance. Weibull distribution causes heavy-tail job arrivals and fluctuating queue lengths. As explained before, a shape parameter less than 1 causes the curve to shift to the right and increase volumes

and queue lengths. Despite the variability, effective load balancing within the supermarket model minimizes the irregular workload.

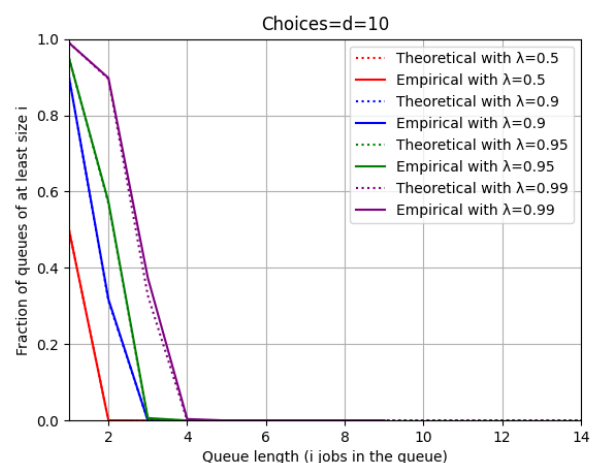
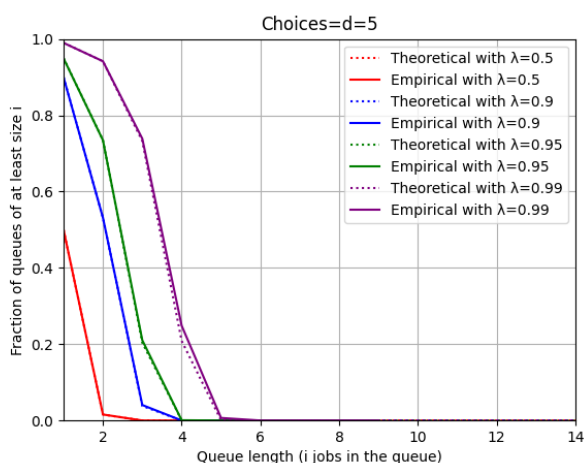
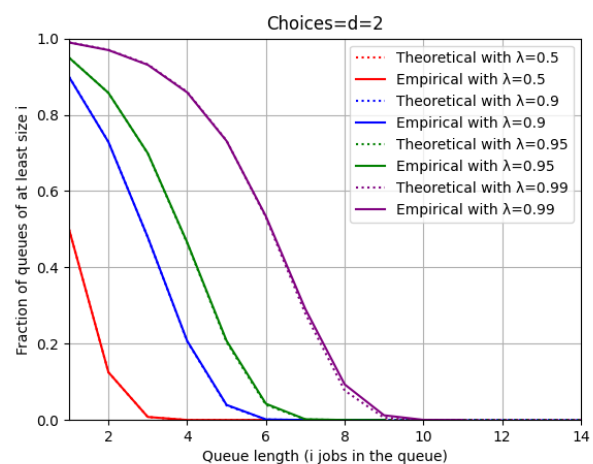
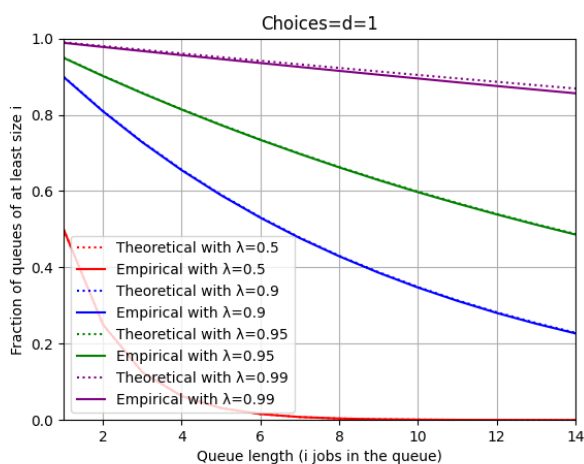
Round Robin scheduling provides jobs equally, maintaining growth balance. But, it does not adjust to dynamic workloads or decreasing queues. Even though it guarantees fairness, it is not the best for efficient load balancing. Weibull and Round Robin produce similar graphs for $d < 5$, even though the approach is different. Weibull's group formation emulates Round Robin's fairness, emphasizing the influence of each approach on load balancing and performance.

Appendix ...

As anticipated before, at the beginning I tried to execute the assignment with a custom Ubuntu implementation, focussing only on FIFO for non-critical jobs and Round Robin for critical one (you can see the python implementation in the "MMN with Ubuntu scheduling?? " chapter).

To decide if a job was critical or not, i leveraged the *modulo* operator to have half of them critical and the other half non-critical.

The following figures show the resultant plots and graphs with the two scheduling combined (FIFO and Round Robin); all of them are obtained launching the system with the same parameters as before.



... (or how to run the code) 🐍

As explained during lectures, the files "discrete_event_sim.py" and "workloads.py" are necessary to the program, otherwise the code will not be executed.

Of course it can be executed directly from the command line using the python3 command and interpreter; e.g.

```
Desktop/dc/mmn_queue_sim via [ ] v3.12.3 took 6s
> py mmn_ubuntu_queue.py -t --max-t 100000 -e --n 10 --d 10
```

I suggest to run one of the several shell scripts attached to the compressed archive of the assignment that I implemented in order to be sure of having all the required pip/python packages. From inside there, it is possible to change the parameters as the user likes. Below there is a screenshot of one of them.

```
echo "\n📦 Installing the required packages..."
pip install matplotlib argparse > /dev/null
if [ $? -ne 0 ]; then
    echo "❌ Error: Failed to install the required python packages."
    exit
fi

brew ls imagemagick > /dev/null || brew install imagemagick
if [ $? -ne 0 ]; then
    echo "❌ Error: Failed to install imagemagick."
    exit
fi

for i in 1 2 5 10
do
    echo "\n🚀 Running mmn simulation with d equals to " $i " ..."
    /usr/bin/python3 mmn.py -t --max-t 100000 --n 1000 --queue_length_frequency 50 --d $i
    cp mmn.png tmp_${i}.png
done

magick tmp_1.png tmp_2.png +append tmp1.png
magick tmp_5.png tmp_10.png +append tmp2.png
magick tmp1.png tmp2.png -append result_mmn.png
rm tmp*.png
```

Peer Review changes done 🏖️

- Inverted legend lines (first Theoretical and then Empirical)
- Set the same color for theoretical and empirical graphs (given the same *lambda*)
- Adjusted empirical to theoretical graphs
- Deleted leftover comments
- Implemented Weibull distribution shape
- Added the commands executed to obtain the various figures

Conclusions END

At the end of the implementation of this assignment, I observed that a distributed system is very influenceable by the change of some parameters and especially to environment changes.

Additionally finding optimal values to run the mmn queue is not a simple task.

In the end, the benefits of the supermarket model in the context of distributed systems are abundantly clear.