Università di Genova | DIBRIS DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

Report - MMN Queue Simulator

Distributed Computing

Master course in Computer Science

Authors:
Cattaneo Kevin - 4944382
Foschi Lorenzo - 4989646

Index

I. Definition of MMN Queue Simulation

Since queuing systems play a crucial role in modeling various real-world scenarios, such as telecommunications networks and customer service processes, their simulation allows us to understand their behavior under different conditions, that we can influence through the modification of parameters in order to make decisions for system optimization and improvement.

II. Supermarket model

The supermarket model allows us to choose the minimum loaded queue between random sorted d queues, then the job will be scheduled to the one selected. Since it brings great performance, in all the plots of the report where d > 1, the supermarket model has been implicitly applied.

III. Analysis

To analyze the performance of the multi-server queuing system we observe the definition and behavior of various parameters:

- λ: rate of job arrivals in the queuing system. Possible optimal (from the point of view of the analysis) chosen values are: [0.5, 0.9, 0.95, 0.99]. It needs to be less than μ.
- μ: rate at which servers process jobs in the queuing system, impacting the system's efficiency in handling incoming tasks. Default value is set to one, allowing the usage of the specific formula $W = 1/(1 - \lambda)$, which can be generalized in $W = 1/(\mu - \lambda)$
- n: denotes the number of servers available for job processing in the queuing system.
  We'll see in the analysis how changes in n can vary the results. If n = 1 we have MM1 queue, with enough computational power we can try with way higher n, and with theoretically infinite power we'll see graphs that assume $n = +\infty$
- d: controls the number selected for job assignment, influencing the overall job distribution strategy. If d = 1 we obtain what is called "Random Model", in which we simply randomly select a single queue within the n available. If d > 1 a Supermarket Model is indeed obtained. In this more refined model the random selection happens with d queues, and then the least populated one will be chosen.
- max_t: Defines the upper limit for the simulation's runtime, providing a controlled environment for observing the system's behavior over time. Default max_t is set to one million, but we played with values and chose to use max_t = 1000 for most of the graphs produced: a trade-off between accuracy and performance is required.
- arrival_rate: Represents the aggregated rate of job arrivals, considering the combined effect of the arrival rate (λ) and the number of servers (n). It reflects the rate at which jobs enter the system and its to handle the incoming workload.

- completition_rate: Represents the rate of job competition.

- W: the average time spent by a job, in detail we distinguished:
    - Practical W (W): It is the average time a job spends in the system, including both the time spent waiting in the queue and the time spent being processed by the servers (in our case, this last one is very short). In the provided code, this value is computed based on the completion and arrival times of jobs, providing an *empirical* measure of the system's performance in handling job processing and wait times.
    - Theoretical W (Wt): It represents the theoretical expectation for the average time a job spends in the system. According to Little's Law, the average number of jobs in a stable system (L) is equal to the arrival rate (λ) multiplied by the average time a job spends in the system (Wt). Therefore, in this context, the theoretical W value is computed based on the average queue lengths and the job arrival rate, providing an analytical measure of the system's performance under theoretical assumptions. Given the Little's Law we have L = avg queue length = Wt * λ

- t: boolean value, if t = 1 we also plot theoretical graphs
- csv: denotes the file name for storing simulation results in CSV format, allowing for structured analysis and further processing of the simulation results.
- len_schedule: determines the frequency of the queue length event scheduling.
- shape: determines the usage of different distributions (expovariate (1) or Weibull)
- extension: boolean parameter to enable (1) the extension of the self-balancing system

IV. Extra Implementation

1. Usage of Weibull Distribution

We also have implemented the usage of Weibull Distribution, from *workload.py* file, that is a generalization of the exponential distribution. If shape = 1 we obtain the same behavior of expovariate, while if:

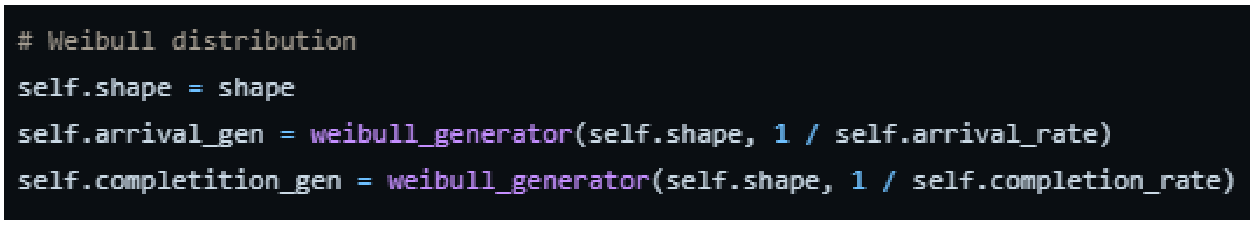
- shape > 1: we approach a bell curve (more uniform values)
- shape < 1: we have most of the work concentrated on few jobs (less uniform values), so called “heavy tailed”

The arrival_rate and completition_rate dictate how the distribution will operate, contributing to create the generators together with shape and mean parameter. By setting the mean to 1 / λ we ensure that the average value corresponds to the reciprocal of the scale parameter.

In theory we have the formula: mean = scale * Γ(1 + (1 / shape))

In practice we invert to calculate scale = mean / Γ(1 + (1 / shape))

This gamma function is an extension of the factorial function to real and complex numbers. When x is a positive integer, Γ(x) equals (x-1)! → so for our case of x = 2: Γ(2) = 1



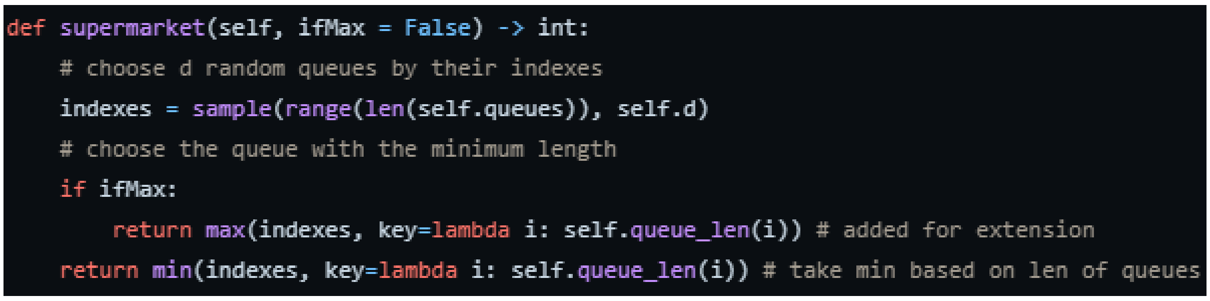
```
# Weibull distribution
self.shape = shape
self.arrival_gen = weibull_generator(self.shape, 1 / self.arrival_rate)
self.completition_gen = weibull_generator(self.shape, 1 / self.completion_rate)
```

2. Extension: Self-balancing distributed system

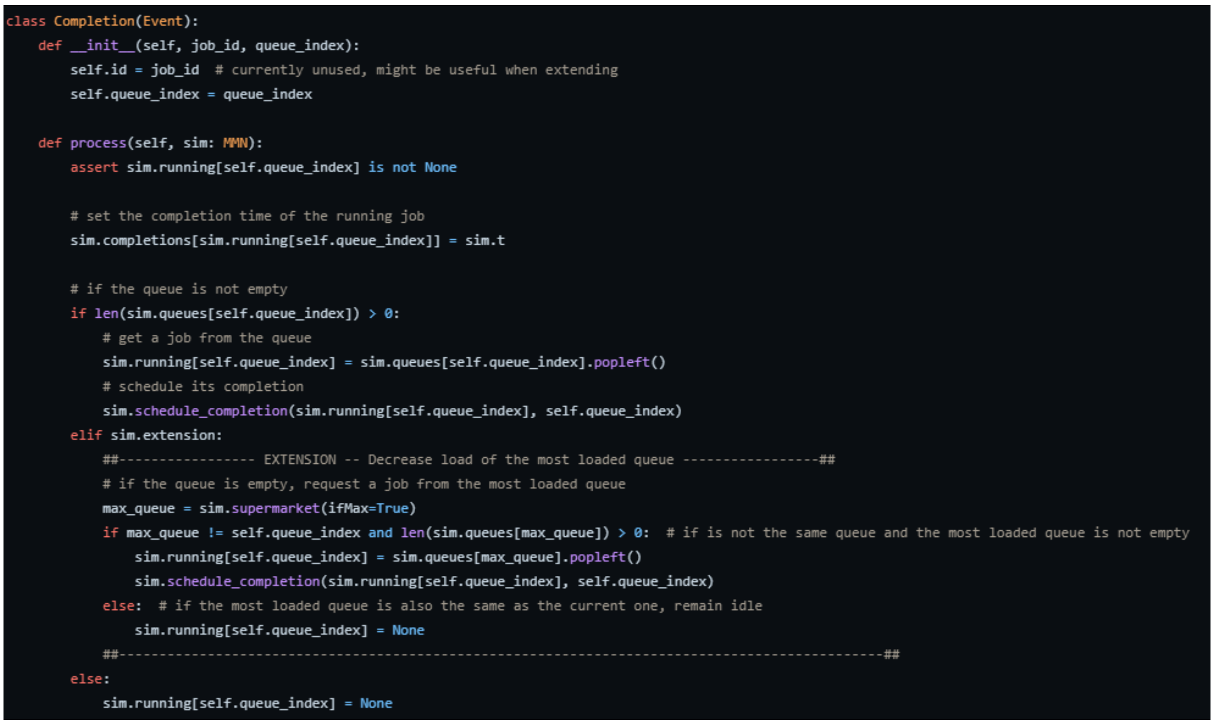
Our extension involves the observation of the performance of our system, under certain constraints. In particular we focus on the graphs.

The step we've taken to achieve a more self-balancing distributed system, that is a system in which machines attempt to decrease the load of their neighbors if they are unoccupied, is to add a new search after the completion of one job; we are re-using the supermarket function, but this time we search for the most loaded queue (machine) and freeing it from one job, that is taken from the unoccupied caller.

This application allows to automatically manage the distribution of workloads or resources among nodes in the network based on their current capacity and availability. Then we take in analysis the results with and without this extension.



```python
def supermarket(self, ifMax = False) -> int:
    # choose d random queues by their indexes
    indexes = sample(range(len(self.queues)), self.d)
    # choose the queue with the minimum length
    if ifMax:
        return max(indexes, key=lambda i: self.queue_len(i)) # added for extension
    return min(indexes, key=lambda i: self.queue_len(i)) # take min based on len of queues
```



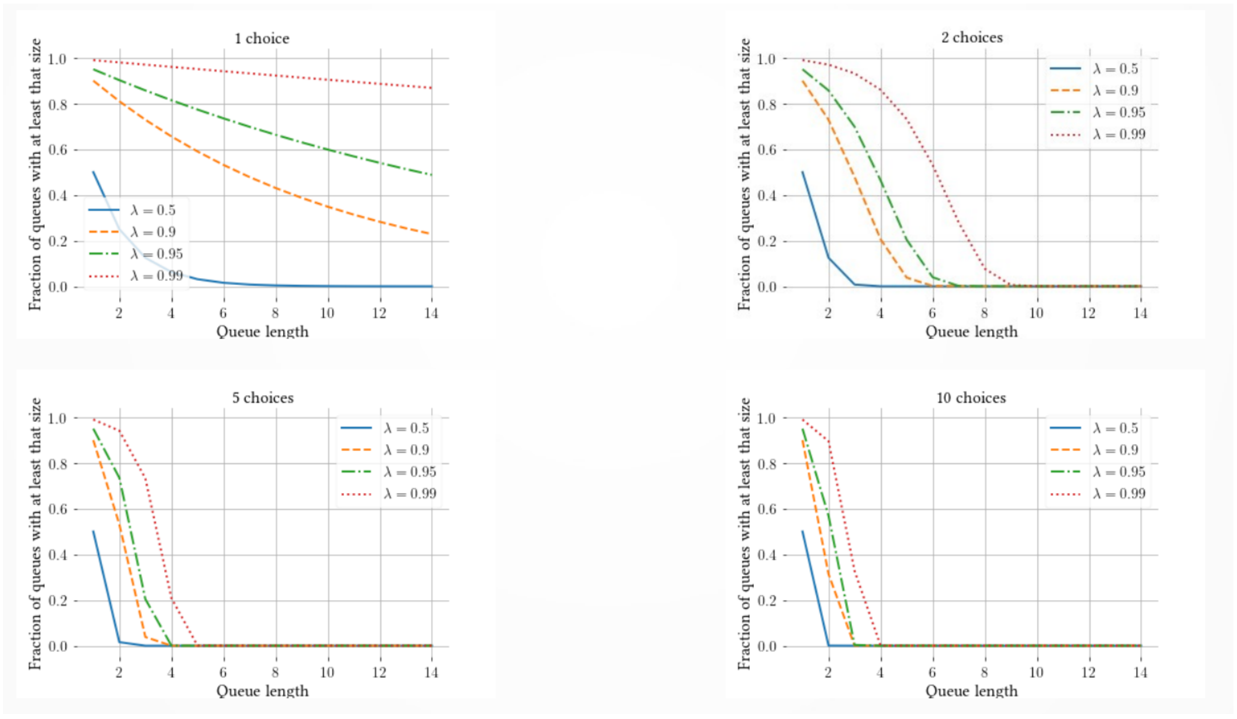
```python
class Completion(Event):
    def __init__(self, job_id, queue_index):
        self.id = job_id  # currently unused, might be useful when extending
        self.queue_index = queue_index

    def process(self, sim: MMN):
        assert sim.running[self.queue_index] is not None

        # set the completion time of the running job
        sim.completions[sim.running[self.queue_index]] = sim.t

        # if the queue is not empty
        if len(sim.queues[self.queue_index]) > 0:
            # get a job from the queue
            sim.running[self.queue_index] = sim.queues[self.queue_index].popleft()
            # schedule its completion
            sim.schedule_completion(sim.running[self.queue_index], self.queue_index)
        elif sim.extension:
            ##---------------- EXTENSION -- Decrease load of the most loaded queue ----------------##
            # if the queue is empty, request a job from the most loaded queue
            max_queue = sim.supermarket(ifMax=True)
            if max_queue != self.queue_index and len(sim.queues[max_queue]) > 0:  # if is not the same queue and the most loaded queue is not empty
                sim.running[self.queue_index] = sim.queues[max_queue].popleft()
                sim.schedule_completion(sim.running[self.queue_index], self.queue_index)
            else:  # if the most loaded queue is also the same as the current one, remain idle
                sim.running[self.queue_index] = None
            ##-----------------------------------------------------------------------------------------------##
        else:
            sim.running[self.queue_index] = None
```

V. Results and plots

Our analysis focused on understanding the influence of varying parameters on the system's behavior, enabling us to draw meaningful conclusions.

1. Ideal plots (Slides plots)

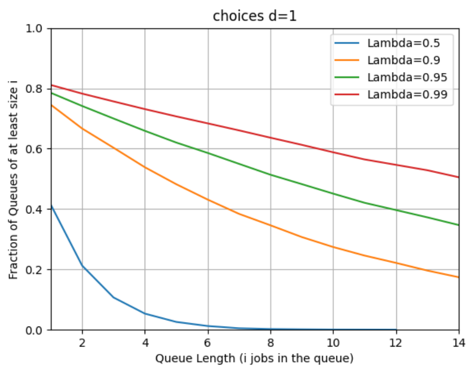
Firstly, we put as objective to reach as ideal plots the ones from professor's slides Note that in these graphs n is set to "virtually infinite", that can lead to differences between these ideal graphs and ours.

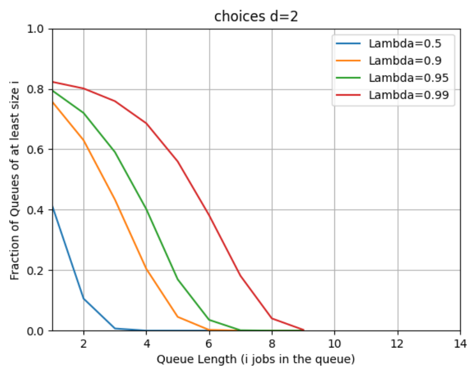


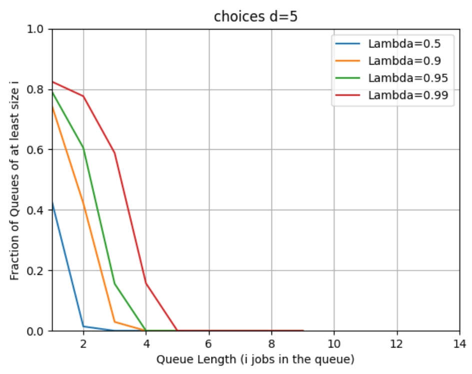
2. Our implementation (without extension)

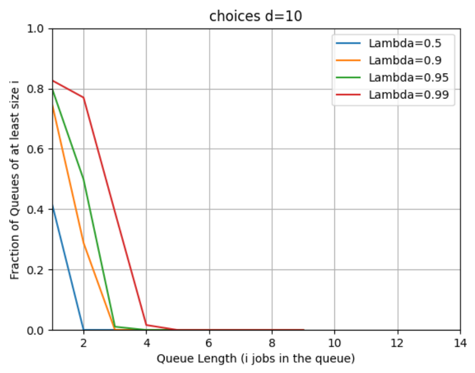
We set n = max_t = 1000; seed = 5 to make experiments repeatable.

2.1 Our implementation plots









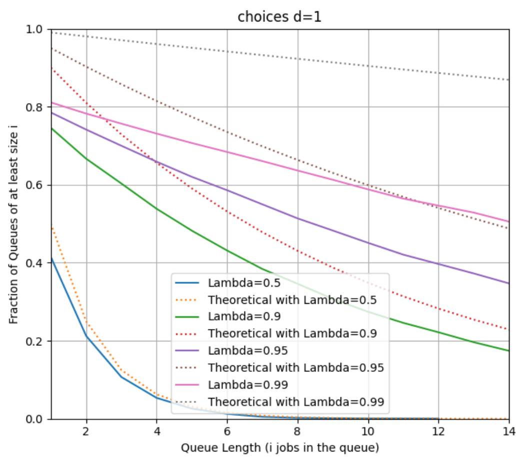
As we can clearly see from both slides and our graph, increasing d has the consequence of decreasing “Fraction of queues with at least that size”. However, the biggest decrease happens at the very first d increase (so after considering the supermarket model, with d = 2), the improvement grows with increasing d, in particular with d = 5 and then d = 10. We observe a change in the total time of the simulation when we have n big and d that grows.

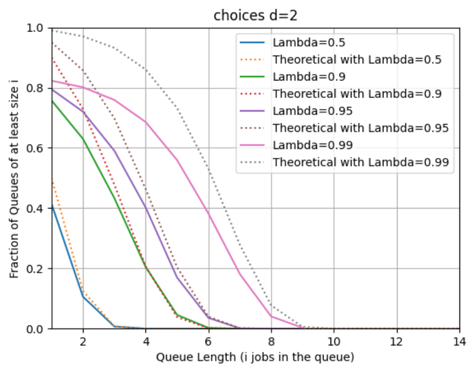
For example we can consider the λ = 0.99 case:

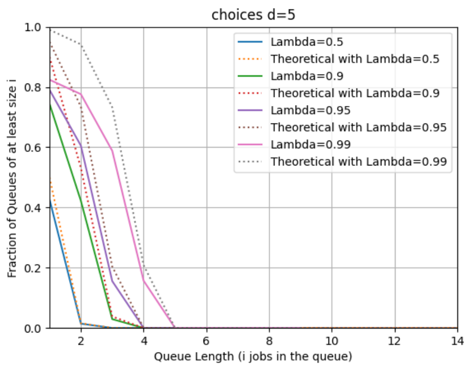
- with d = 1 (random model) we can’t even see at which value we reach a near-zero “Fraction of queues with at least that size”.
- with d = 2 we suddenly drop at around L = 9, with a big improvement
- with d = 5 the line “touches” the 0-axis at around L = 5
- with d = 10, so doubling it, we only get it at L = 4; here we can see the upper limitation of increasing d: even talking about great values of n, more splitting than this don’t bring to us a substantial improvement, instead we could slow the system when calling supermarket functions querying many queues, in according to the question that we have to pose ourselves about the right trade-off between precision and performance when dealing with

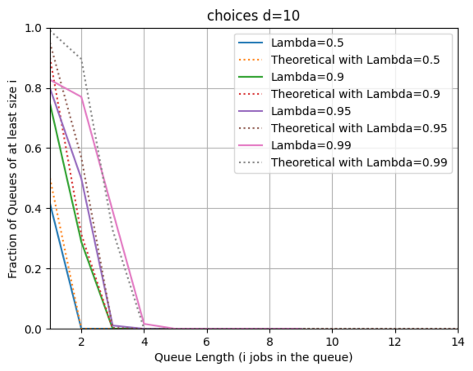
the choice of d. We can go on with d=15, d=20, but as we said, as we get some improvement, we furthermore have the risk of slowing the system.

2.2 Our implementation plots with theoretical values plotted









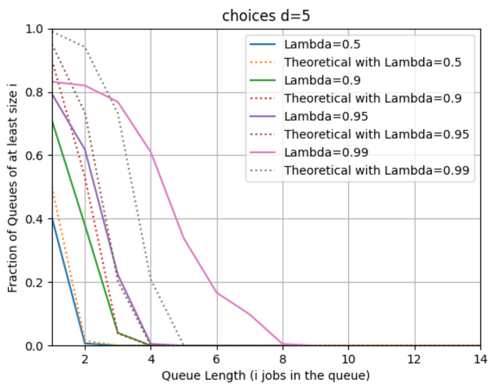
As we can see our plots follow the decreasing of the theoretical plots, in most of the cases the latter are the upper limit of fractions that our graph can reach.

Moreover we can observe how the system behave changing the other parameters, with d fixed (here d = 5), and as assumptions μ is also fixed, so:

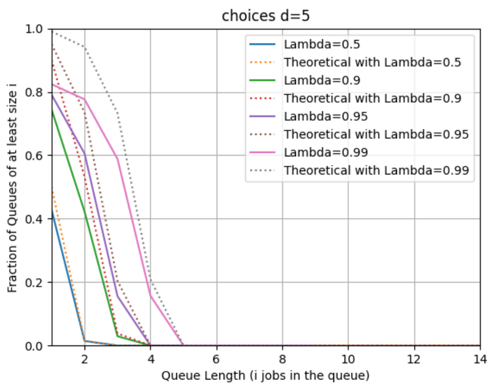
- varying n, the number of queues: if small it produces less precise results: since the simulation lasts a shorter time, statistically we have fractions on a few queues to plot, moreover we have queues with a bigger size; on the other hand since the simulation is faster with n small, changing that parameter can lead the trade-off between time and accuracy we want to achieve; *we decided to plot the difference of n in the following image*.
- varying max_t, the maximum simulation time: if small it produces less accurate results, since it manages the time of the simulation, so less time, less precision. However we obtained good results lowering max_t from 10^6 to 10^3.

- varying λ, as we can see in each graph, the arrival_rate of jobs change, and the system is much more occupied. For example if λ is equal to 0.99, it means that just the 1% of the system is, probabilistically, unoccupied. Since the rate is increased, also the load on the system is, so the length of the queues is bigger.

n = 100

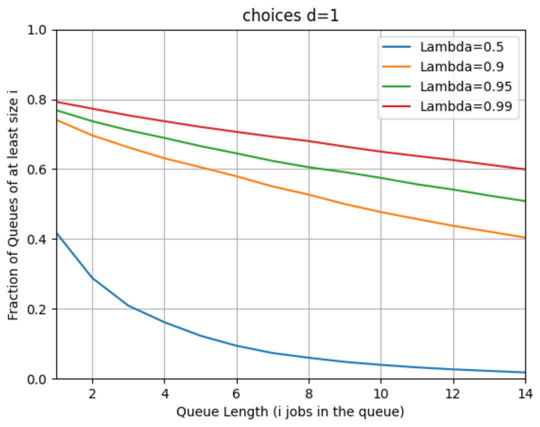


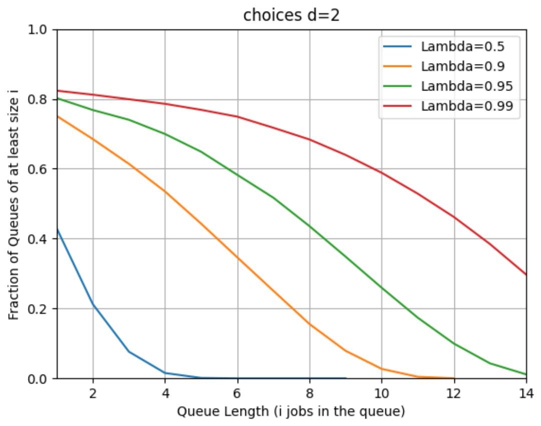
n = 1000

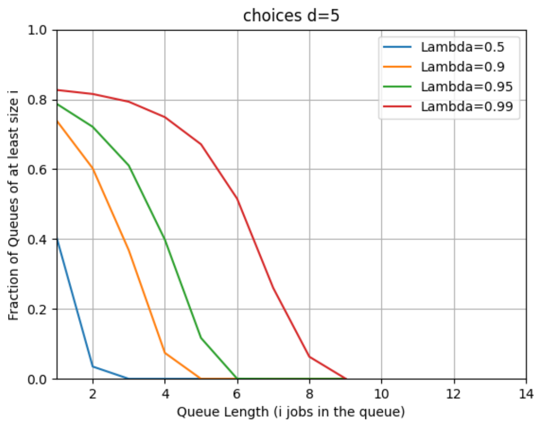


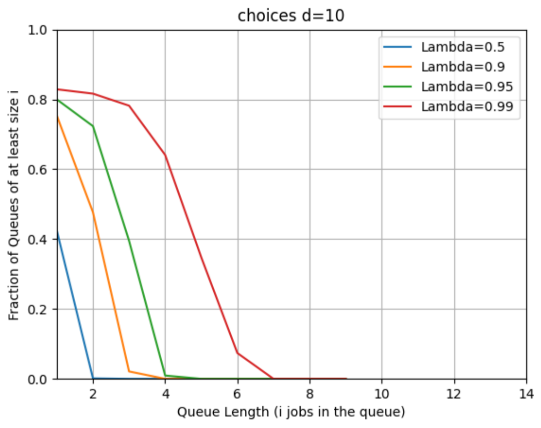
2.3 Our implementation plots with Weibull Distribution

We decided to observe how the system behaves with another type of arrival job scheduling, that has been obtained by changing the distribution with a “heavy tailed” Weibull one. According to this, we set the shape to 0.5.





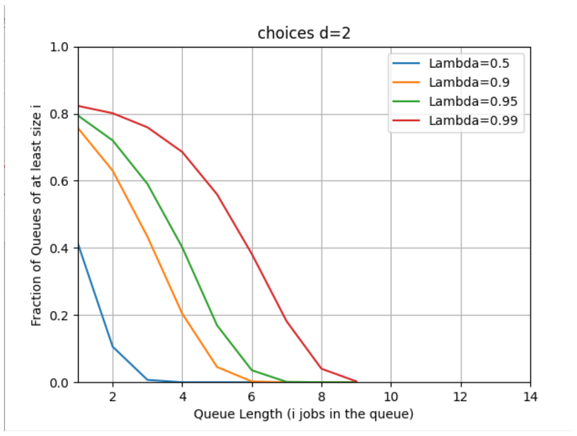


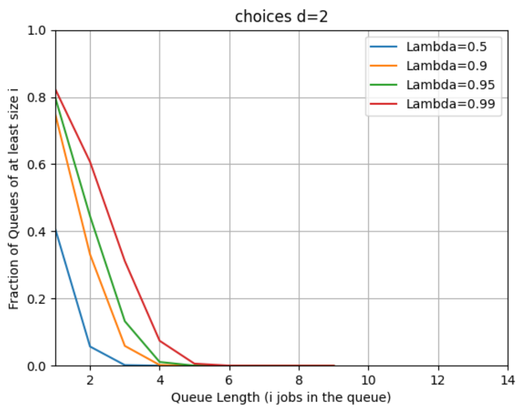


Here we can see a big difference when the distribution changes. The current Weibull Distribution we have selected makes the arrival of the jobs less “homogeneous”, in fact, as the definition of “heavy tailed”, many jobs arrive at the beginning of the timeline, or very late. That “unpredictability” can be seen on the plots as a bigger queue length, and a much more volume for each fraction analyzed. Here comes the importance to use the supermarket with values of d much more bigger than the previous ones used, so the “upper limit” seen before, can be increased, so that the load between queues is more distributed.

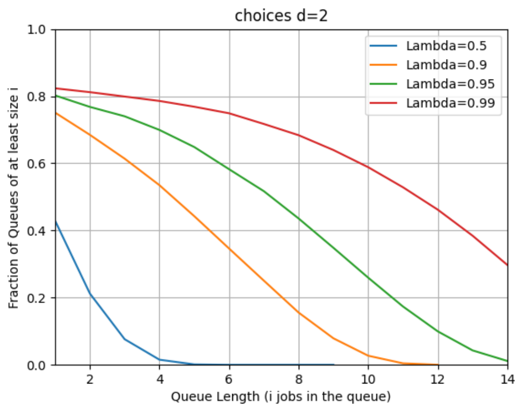
3. Our implementation (with extension)

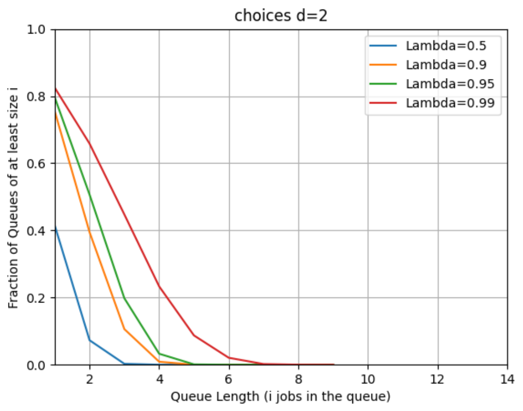
We set n = max_t = 1000; seed = 5 to make experiments repeatable.
As we said, our extension applies a new supermarket round if the current machine that has finished all its jobs is unoccupied; this time that supermarket will select the most loaded queue to transfer a job to the idle one.





Plotting the difference between the system without extension (left) and with extension (right), we chose d = 2 to show how, even with only the choice between two machines, this extension potentially reduces the load of the system.

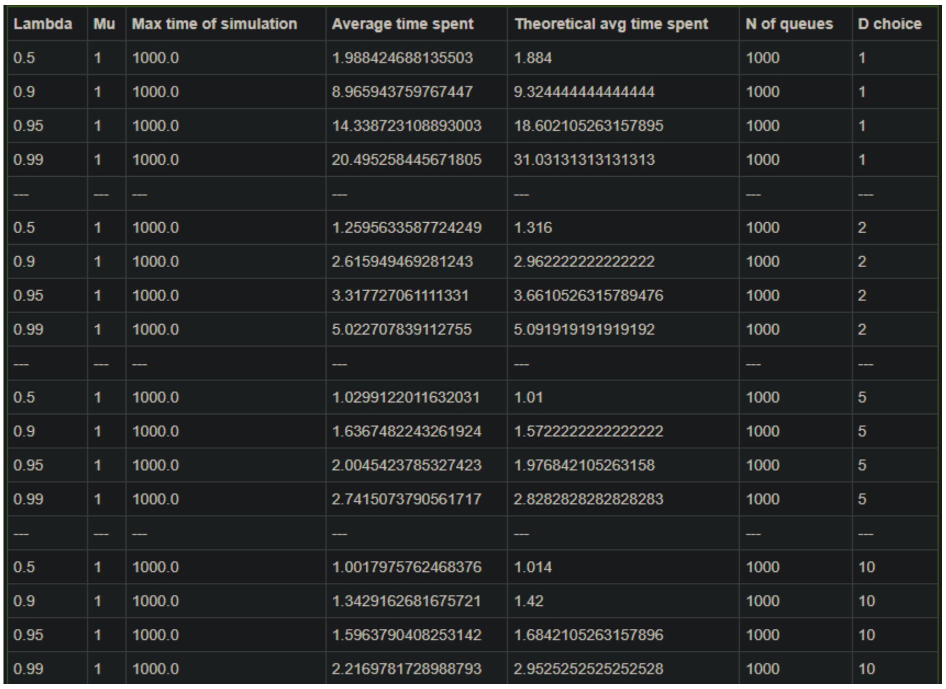




Plotted the difference between the system without extension (left) and with extension (right), using the Weibull Distribution with shape = 0.5 (“heavy tailed”)
In each choice of distribution we can see how this extension increases the performance of the system, greatly reducing the load of each queue so the majority of the fraction of queues with a certain size now have their length reduced.

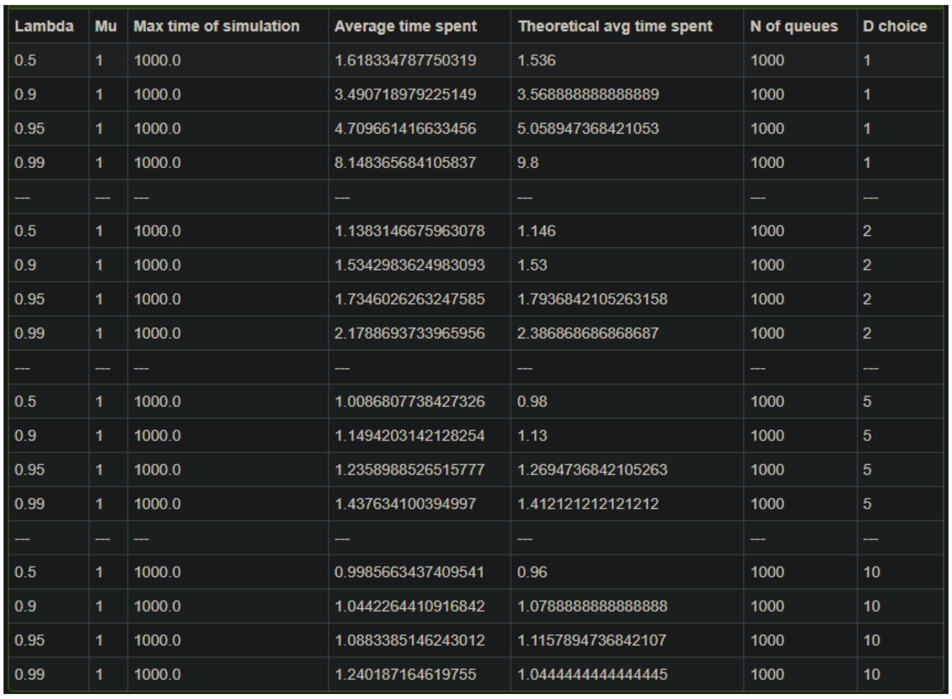
4. Our implementation on CSV

Without extension



Lambda	Mu	Max time of simulation	Average time spent	Theoretical avg time spent	N of queues	D choice
0.5	1	1000.0	1.988424688135503	1.884	1000	1
0.9	1	1000.0	8.965943759767447	9.324444444444444	1000	1
0.95	1	1000.0	14.338723108893003	18.602105263157895	1000	1
0.99	1	1000.0	20.495258445671805	31.03131313131313	1000	1
---	---	---	---	---	---	---
0.5	1	1000.0	1.2595633587724249	1.316	1000	2
0.9	1	1000.0	2.615949469281243	2.962222222222222	1000	2
0.95	1	1000.0	3.317727061111331	3.6610526315789476	1000	2
0.99	1	1000.0	5.022707839112755	5.091919191919192	1000	2
---	---	---	---	---	---	---
0.5	1	1000.0	1.0299122011632031	1.01	1000	5
0.9	1	1000.0	1.6367482243261924	1.5722222222222222	1000	5
0.95	1	1000.0	2.0045423785327423	1.976842105263158	1000	5
0.99	1	1000.0	2.7415073790561717	2.8282828282828283	1000	5
---	---	---	---	---	---	---
0.5	1	1000.0	1.0017975762468376	1.014	1000	10
0.9	1	1000.0	1.3429162681675721	1.42	1000	10
0.95	1	1000.0	1.5963790408253142	1.6842105263157896	1000	10
0.99	1	1000.0	2.2169781728988793	2.9525252525252528	1000	10

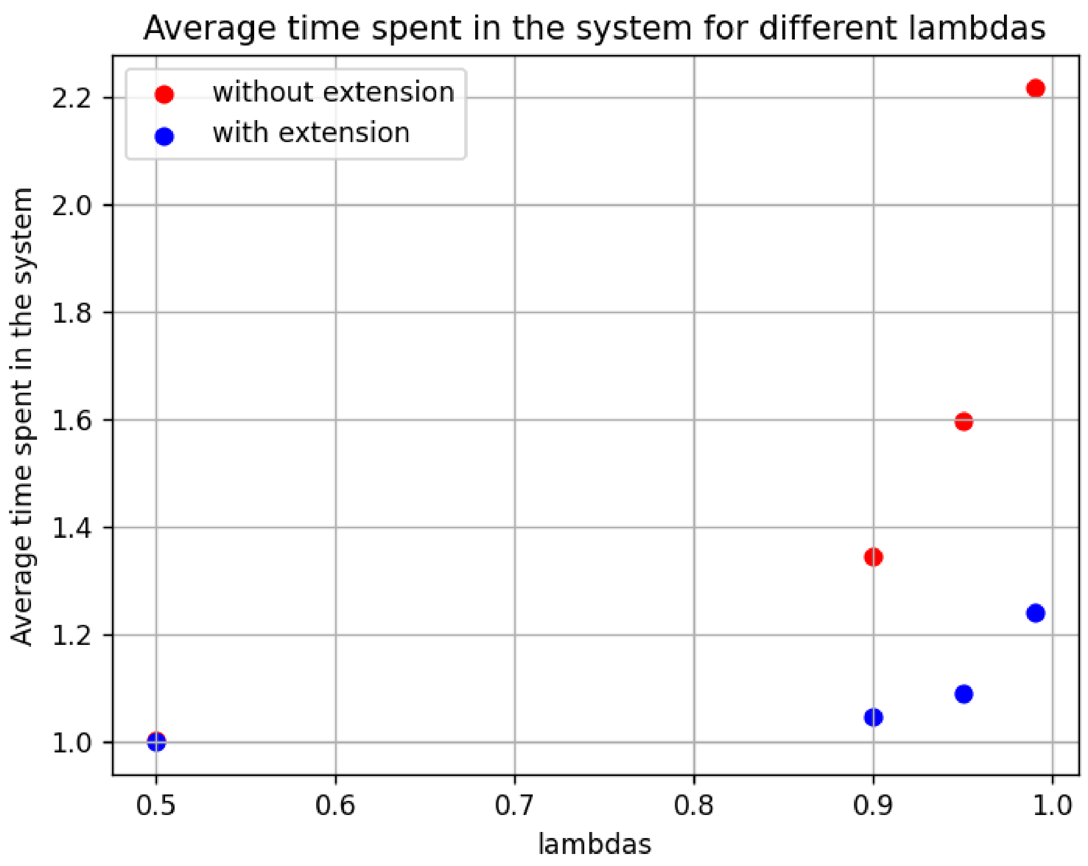
With extension



Lambda	Mu	Max time of simulation	Average time spent	Theoretical avg time spent	N of queues	D choice
0.5	1	1000.0	1.618334787750319	1.536	1000	1
0.9	1	1000.0	3.490718979225149	3.568888888888889	1000	1
0.95	1	1000.0	4.709661416633456	5.058947368421053	1000	1
0.99	1	1000.0	8.148365684105837	9.8	1000	1
---	---	---	---	---	---	---
0.5	1	1000.0	1.1383146675963078	1.146	1000	2
0.9	1	1000.0	1.5342983624983093	1.53	1000	2
0.95	1	1000.0	1.7346026263247585	1.7936842105263158	1000	2
0.99	1	1000.0	2.1788693733965956	2.386868686868687	1000	2
---	---	---	---	---	---	---
0.5	1	1000.0	1.0086807738427326	0.98	1000	5
0.9	1	1000.0	1.1494203142128254	1.13	1000	5
0.95	1	1000.0	1.2358988526515777	1.2694736842105263	1000	5
0.99	1	1000.0	1.437634100394997	1.412121212121212	1000	5
---	---	---	---	---	---	---
0.5	1	1000.0	0.9985663437409541	0.96	1000	10
0.9	1	1000.0	1.0442264410916842	1.0788888888888888	1000	10
0.95	1	1000.0	1.0883385146243012	1.1157894736842107	1000	10
0.99	1	1000.0	1.240187164619755	1.0444444444444445	1000	10

With CSV we can focus again on the two average values and see that the extension reduces the average time spent since unoccupied queues now take jobs from full queues, the most when the system is loaded at 99%.

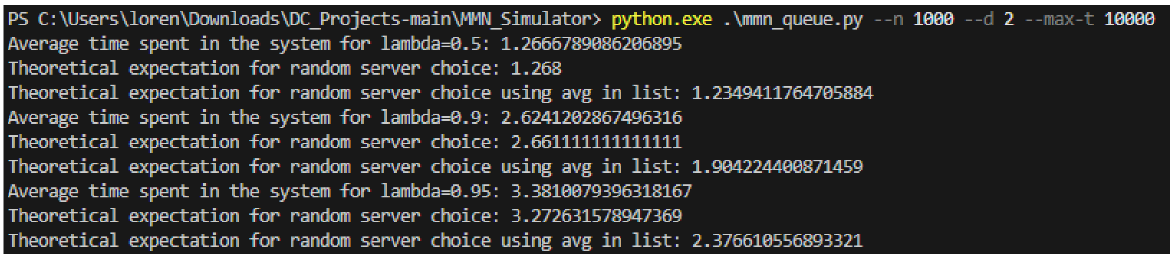
We decided to make an alternative visualization of those practical averages in the CSV between the version without and with extension, fixing d = 10.



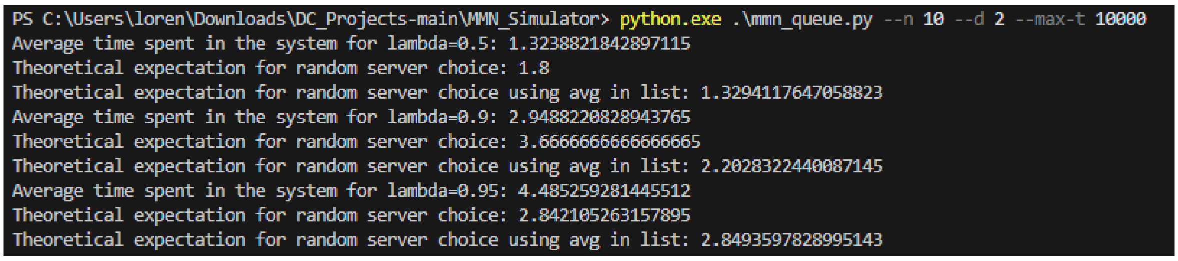
As we can see even in this visualization, the practical average time spent by the system is slightly smaller for λ = 0.5, while almost reduced by half for λ = 0.99.

5. More details about the theoretical average time W

In the following paragraph we assume W to be the theoretical average time, not the true average time made by the system.
As we pointed out in the first section of the report, we trace out some differences between calculating W once, by computing the mean of the length of the queue divided by the λ, that is the arrival rate of the jobs, and having an average of multiple W computations throughout the simulation, thanks to the GetLength Event; there we storage a list of intermediate W (called w_track) that contains the same previous computation of W, but done for each time the event is called. Then, in the main function we sum all the time elements and divide them by their quantity, obtaining another mean.



```
PS C:\Users\loren\Downloads\DC_Projects-main\MMN_Simulator> python.exe .\mmn_queue.py --n 1000 --d 2 --max-t 10000
Average time spent in the system for lambda=0.5: 1.2666789086206895
Theoretical expectation for random server choice: 1.268
Theoretical expectation for random server choice using avg in list: 1.2349411764705884
Average time spent in the system for lambda=0.9: 2.6241202867496316
Theoretical expectation for random server choice: 2.6611111111111111
Theoretical expectation for random server choice using avg in list: 1.904224400871459
Average time spent in the system for lambda=0.95: 3.3810079396318167
Theoretical expectation for random server choice: 3.272631578947369
Theoretical expectation for random server choice using avg in list: 2.376610556893321
```



```
PS C:\Users\loren\Downloads\DC_Projects-main\MMN_Simulator> python.exe .\mmn_queue.py --n 10 --d 2 --max-t 10000
Average time spent in the system for lambda=0.5: 1.3238821842897115
Theoretical expectation for random server choice: 1.8
Theoretical expectation for random server choice using avg in list: 1.3294117647058823
Average time spent in the system for lambda=0.9: 2.9488220828943765
Theoretical expectation for random server choice: 3.6666666666666665
Theoretical expectation for random server choice using avg in list: 2.2028322440087145
Average time spent in the system for lambda=0.95: 4.485259281445512
Theoretical expectation for random server choice: 2.842105263157895
Theoretical expectation for random server choice using avg in list: 2.8493597828995143
```

From those result (obtained without extension) we can observe that:

- for n big (up), the practical average time is much more similar to the computation of W done once, rather than the mean of the list of theoretical average time spent
- for n small (down), the viceversa applies, in fact there are more cases where practical average value is nearer the mean of the list of theoretical average time spent, while the one computed once may be even larger.
  In this case we expect that this value can be more reliable because the number of queues is small and so the mean computed every time is less influenced than the only one general mean, and in a certain point of view more precise; in fact the general mean would be influenced a lot if the load of the different queues is unequal (we can see it with the various λ in the plots), and there are a few queues. While if the number of queues is high (about to infinite), even if the load of a few queues is very distant from the general mean, this one would not be influenced so much ("the majority makes the mean", and so the first case).

  That behavior also applies to changing shape in the Weibull distribution: we observed that with smaller shape (heavy-tailed), the mean of the list is more reliable, since many jobs arrive at the beginning or at the end of the timeline, and so taking the average only at the end can lead to potential mistakes.

VI. Conclusions

At the end of our analysis we observed that a distributed system is very influenced by the change of some parameters and finding the optimal values is not an easy task.

For the first part the results of average time and the theoretical one showed us on which parameters we need to work on, and by increasing the number of queues we obtain better results. But still far from the theoretical ones.
Then both the supermarket model and plots had a key role: simply choosing supermarket instead of random queueing with a big value of d had a larger impact on the performance, allowing us to achieve the most similar graphs to the theoretical ones we could have, while the plots that pointed out what were the problems and the necessary tweaks on the parameters.

For the extension part, performance was our goal, and the possibility of reducing the load between the queues themselves automatically was a great idea. Since we had already implemented the supermarket to search for the minimum load between d chosen queues, we transformed it to get the maximum load, in order to lower it.
As the plots state out this has brought some performance in the system, in particular because that change scales well if we are talking of thousands of queues, that in the real world scenario could be machines that have been optimized.

VII. Appendix

1. Setup to run the code

File discrete_event_sim.py and workloads.py are necessary to run the program.

The code has been realized with Python version 3.12.0
To run the code simply call python.exe (or python3) on the mmn_queue.py script; to specify parameters use "--[name-of-parameter][space][value]", without quotes and brackets.

Example: "python.exe .\mmn_queue.py –max-t 1_000_000"
Note that the code may take a while depending on the value of arguments.
Please note also that the simulation has some randomness and statistics (if seed not fixed) within the scheduling of the events, so run the code multiple times to achieve a clear idea of the program behavior.

Please specify also:

- "--shape [value]" if you want to enable weibull distribution
- "--len-schedule [value]" if you want to control the scheduling of GetLength event to get personalized statistical values to be used in the plot
- "--csv [filename]" if you want to save results on a CSV file
- "--t 1" if you want to plot also theoretical lines in the plot
- "--lambd [value]" if you want to specify the lambda, by default we plot the result with four different lambda; note that it has to be less than μ.
- "--extension [0,1]" if you want to enable extension of self-balancing system; of course the better improvement is obtain with d > 1 (since supermarket applies)
- we suggest also to specify the n and max-t parameters equal to 1000, and **d**.

2. External references

We used the *weibull_generator* function to get the Weibull distribution, so you need to have the files specified at appendix #1.

VIII. Peer review

The following paragraph refers to the corrections and modifications suggested by the peer reviewer of our project. In particular we:

- added code snippets about our extension, in order to avoid to open the source code to have an overview of the implementation;
- improved the overall quality of the report, to make it more clear and readable.