UniGe | DIBRIS

MMN Queue Simulation

Assignment Report

Distributed Computing

Professor Matteo Dell'Amico

Seyede Aida Atarzadeh Hosseini - 6936800

Delaram Doroudgarian - 5881909

Introduction:

This assignment emphasizes the practical investigation of system behavior through simulation, exploring the complexities of distributed systems. It challenges conventional simplifying assumptions, such as memoryless exponential distributions and single-server models, by adopting a more nuanced approach. The tasks involve completing a discrete event simulation (DES) framework, implementing an M/M/1 FIFO simulation, and extending it to an M/M/n model. These efforts aim to provide practical experience in analyzing system behavior under realistic conditions. This report outlines the process, highlighting the methods employed, obstacles encountered, and key findings obtained throughout the exploration.

Crucial parameters and formula:

lambda(λ): it represents the arrival rate of jobs into the system. It is typically measured as the number of arrivals per unit of time.

- Low λ: Jobs arrive infrequently, and queues are mostly empty.
- High λ: Jobs arrive quickly, and queues become congested.

mu(μ): It represents the service rate, which is the rate at which jobs are processed or completed by a server. It is a key parameter that helps determine how quickly the system can handle incoming jobs. It's typically measured as the number of jobs completed per unit of time.

Number of servers (n): The total number of servers available in the system to process entities concurrently. The number of servers is a fundamental distinction between M/M/1 and M/M/N queueing systems. In M/M/1 there is only one server. However, there is more than one servers in M/M/N.

Maximum simulation time (max_t): This value ensures that the simulation does not run indefinitely and provides a boundary for processing events. max_t limits the simulation to a finite duration and all events (arrivals, completions, and queue recordings) are processed up to this time.

Times: It records the lengths of all queues at specific intervals during the simulation. It serves as a log or history of queue states, used for analysis, such as plotting the queue length distribution.

d: d is the number of randomly chosen queues. When d = 1, it's a queue is randomly chosen from the available ones. For d > 1, a more advanced "Supermarket Model" is used. In this model, d queues are randomly selected, and the least busy queue is chosen for the job.

Implementation of M/M/N Queue:

- Initially, the M/M/1 queue was implemented by completing the provided code framework.
- Subsequently, the implementation was extended to an M/M/n queue. To accommodate n servers, the sim.queue variable was modified from a single collections.deque to an array of collections.deque, corresponding to the number of servers. Additionally, the sim.running variable was updated from a single variable to an array to account for multiple servers processing jobs simultaneously.
- It is important to note that modifications were also made to the Arrival and Completion functions to support the changes in the queue and server structure.

```
self.running = [None] * n  # if not None, the id of the running job (per queue)
self.queues = [collections.deque() for _ in range(n)]  # FIFO queues of the system1
```

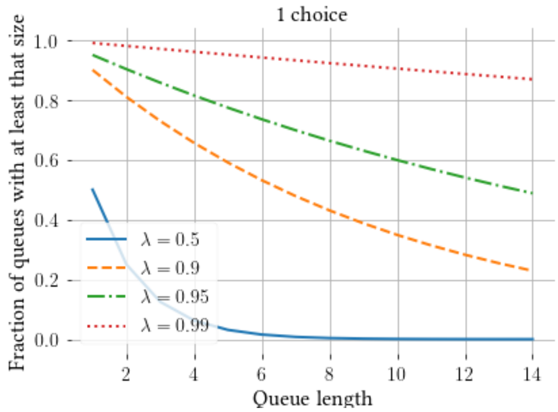
Arrival Function with the Supermarket Model Optimization:

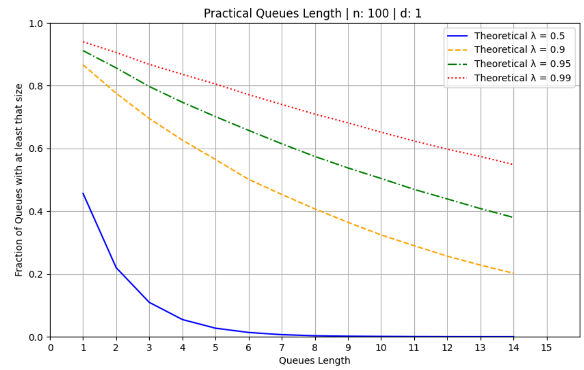
The Arrival function in the M/M/N queue simulation models the entry of new jobs into the system, ensuring optimal queue selection using the Supermarket Model optimization. When a job arrives.

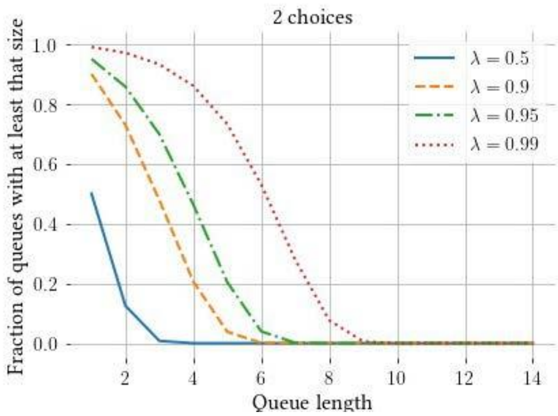
```
class Arrival(Event):
    def __init__(self, job_id):
        self.id = job_id
    def process(self, sim: MMN):
        sim.arrivals[self.id] = sim.t
        samples = sample(sim.queues, sim.d)
        shortest_lists = [lst for lst in samples if len(lst) == min(len(sublist) for sublist in samples)]
        selected_list = choice(shortest_lists)
        queue_index = sim.queues.index(selected_list)
        if sim.running[queue_index] is None:
            sim.running[queue_index] = self.id
            sim.schedule_completion(self.id, queue_index)
        else:
            sim.queues[queue_index].append(self.id)
        sim.schedule_arrival(self.id + 1)
```

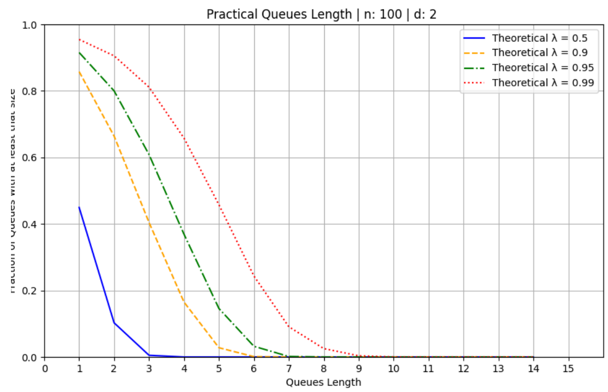
Theoretical vs Practical plot:

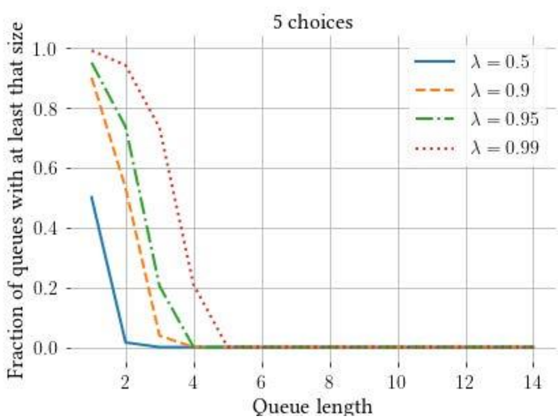
We began by plotting both the theoretical plot (on the left) and the practical plot (on the right), using the parameters n=100, max_t=1000. These parameters were consistently applied in all subsequent plots.

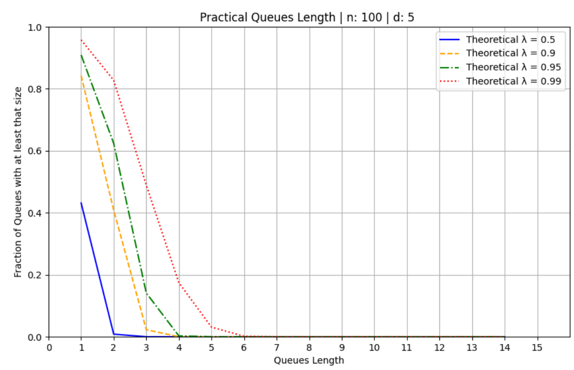

1 choice
Fraction of queues with at least that size
Queue length
$\lambda = 0.5$
$\lambda = 0.9$
$\lambda = 0.95$
$\lambda = 0.99$

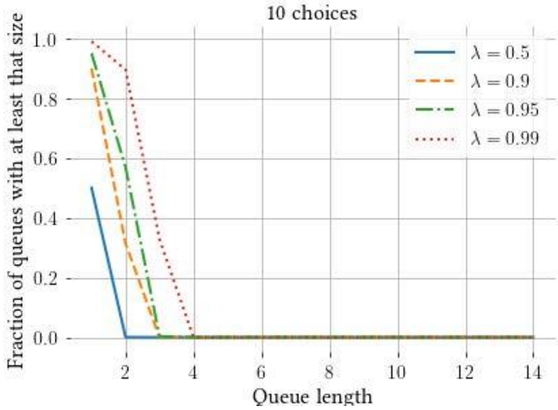

Practical Queues Length | n: 100 | d: 1
Fraction of Queues with at least that size
Queues Length
Theoretical λ = 0.5
Theoretical λ = 0.9
Theoretical λ = 0.95
Theoretical λ = 0.99

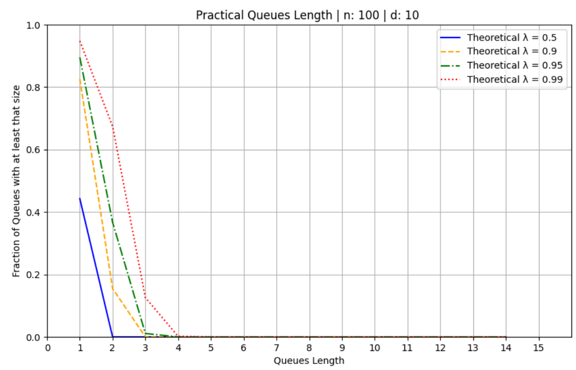

2 choices
Fraction of queues with at least that size
Queue length
$\lambda = 0.5$
$\lambda = 0.9$
$\lambda = 0.95$
$\lambda = 0.99$


Practical Queues Length | n: 100 | d: 2
Queues Length
Theoretical λ = 0.5
Theoretical λ = 0.9
Theoretical λ = 0.95
Theoretical λ = 0.99


5 choices
Fraction of queues with at least that size
Queue length
$\lambda = 0.5$
$\lambda = 0.9$
$\lambda = 0.95$
$\lambda = 0.99$


Practical Queues Length | n: 100 | d: 5
Fraction of Queues with at least that size
Queues Length
Theoretical λ = 0.5
Theoretical λ = 0.9
Theoretical λ = 0.95
Theoretical λ = 0.99





In advance of comparing the plots, let me explain the concept of "fraction of queues with at least X size":

This concept refers to the proportion of queues that have a length greater than or equal to a specific size.

Fraction on queues represents the ratio percentage of queues that satisfy a certain condition (in this case, having a length greater than or equal to a specified value) relative to the total number of queues.

$$Fraction = \frac{Number\ of\ queues\ with\ length \geq X}{Total\ number\ of\ queues}$$

- This metric helps assess how well the system distributes the load across queues.
- If the fraction for longer queue lengths (length ≥ 10) is low, it indicates that there are only a few long queues and the load is distributed effectively.
- If the fraction for shorter queue lengths (length ≥ 2) is high, it shows that most queues are short, which suggests the system is lightly loaded or well-optimized.

d=1

- When queues are chosen randomly (without considering their lengths), the system fails to distribute the load equally.
- At λ=0.9, the distribution is uneven, and there are more long queues compared to lower values of λ.
- For λ=0.5, the queues are generally shorter because the system experiences a lighter load.
- The practical model shows a similar trend but with a more uniform load distribution across the queues.

d=2

- When two queues are randomly selected, and their lengths are compared, the load distribution improves significantly, resulting in fewer long queues.
- For higher values of λ (λ ≥0.9), long queues still exist but are noticeably less frequent compared to the d=1 case.
- The practical outcomes closely match the theoretical predictions

d=5

- When d=5, the system distributes the load much better, significantly reducing the queue lengths.
- Long queues are almost nonexistent, even at high arrival rates (λ=0.99).
- The practical model also shows shorter queues and improved load distribution with d=5.
- The simulated results are very similar to the theoretical predictions.

d=10

- With d=10, there is a slight improvement in load distribution compared to d=5.
- However, the difference is minimal because the system has already reached a balanced state.
- The practical model also shows that increasing d to 10 doesn't provide significant improvements over d=5.
- The practical and theoretical results are almost alike.

Summary of observations:

- Increasing d improves load balancing and reduces queue lengths.
- The most noticeable improvement is seen at d=5, and increasing d beyond this point results in only minimal changes.
- The simulation and theoretical plots generally match well.
- d=5 appears to be the optimal choice, providing a good balance between complexity and improved load distribution.

Implementation of M/M/N queue with Weibull:

we have implemented two different service time distributions: exponential and Weibull. The left-side plots represent the system behavior under the exponential distribution, while the right-side of plots demonstrate the system's performance when using the Weibull distribution. The comparison of these plots provides insights into the impact of different service time distributions on the system's behavior.

It is characterized by a shape parameter (k) and a scale parameter (λ).

- When k=1: Weibull behaves like an Exponential Distribution
- When k<1: Some jobs are extremely long, causing queue congestion.
- When k>1: Most jobs have similar durations, but fewer long jobs.

The Weibull generator is defined in the workloads.py file, and it's used to produce interarrival times and service times in the simulator.

```
def weibull_generator(shape, mean):
    """Returns a callable that outputs random variables with a Weibull distribution having the given shape and mean."""

    return functools.partial(random.weibullvariate, mean / math.gamma(1 + 1 / shape), shape)
```

```
self.arrival_rate = lambd * n  # frequency of new jobs is proportional to the number of queues
self.completion_rate = mu
self.arrival_gen=weibull_generator(weibull_shape,1/self.arrival_rate)
self.service_gen=weibull_generator(weibull_shape,1/self.completion_rate)
self.schedule(self.arrival_gen(), Arrival(0))  # schedule the first arrival
```

Arrival_shape and service_shape are shape parameters for the Weibull distribution.
1/self.lambd and 1/self.mu are the mean service times for the Weibull distribution.
self.arrival_gen() is called whenever a new job needs to be scheduled for arrival, and self_service_gen() is called whenever a job's service time is required.

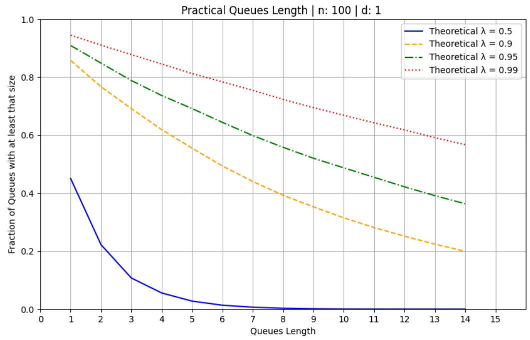
```
def schedule_arrival(self, job_id):
    self.schedule(self.arrival_gen(), Arrival(job_id))
```

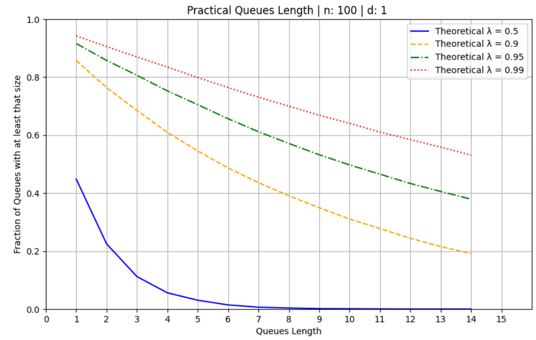
```
def schedule_completion(self, job_id, queue_index):
    self.schedule(self.service_gen(), Completion(job_id,queue_index))
```

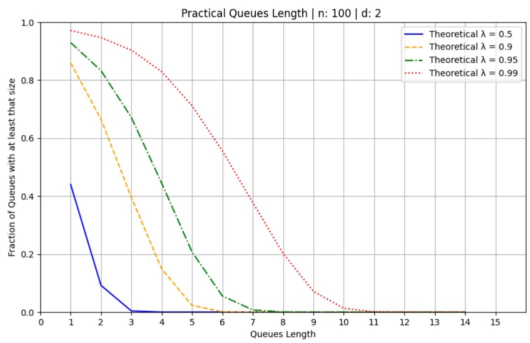
Analysing the Results of the Weibull Extention

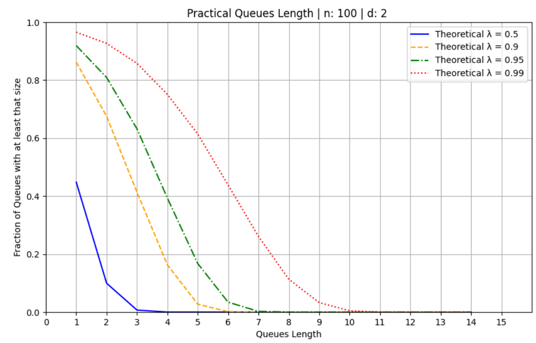
The lift-side plots represent the system's behavior before applying the Weibull distribution and the right-side ones are after.

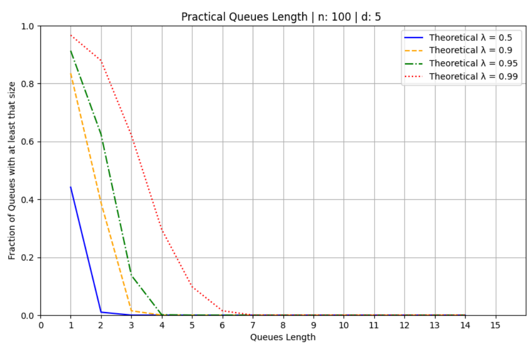
Shape = 1

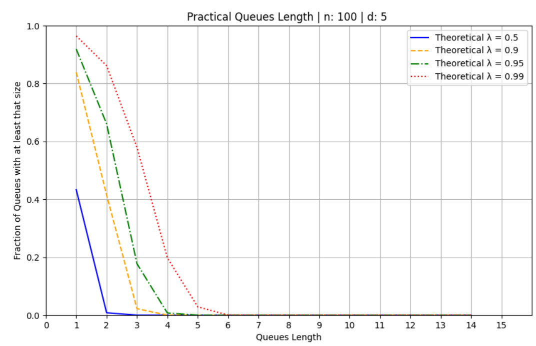

Practical Queues Length | n: 100 | d: 1
Theoretical λ = 0.5
Theoretical λ = 0.9
Theoretical λ = 0.95
Theoretical λ = 0.99
Fraction of Queues with at least that size
Queues Length

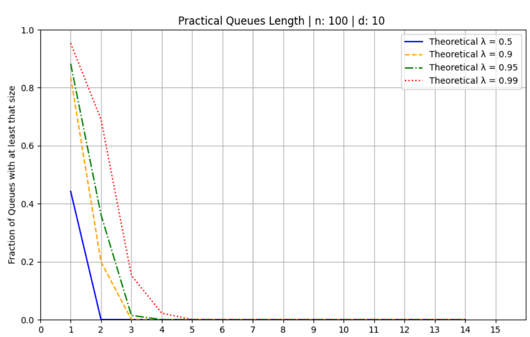

Practical Queues Length | n: 100 | d: 1
Theoretical λ = 0.5
Theoretical λ = 0.9
Theoretical λ = 0.95
Theoretical λ = 0.99
Fraction of Queues with at least that size
Queues Length

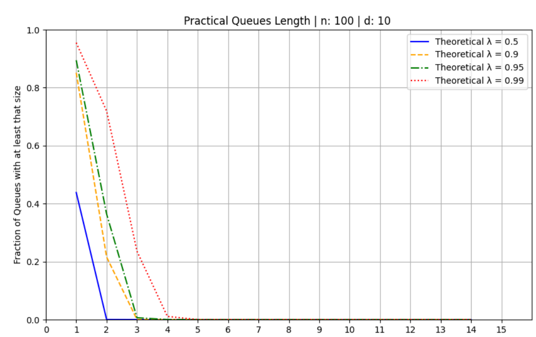

Practical Queues Length | n: 100 | d: 2
Theoretical λ = 0.5
Theoretical λ = 0.9
Theoretical λ = 0.95
Theoretical λ = 0.99
Fraction of Queues with at least that size
Queues Length


Practical Queues Length | n: 100 | d: 2
Theoretical λ = 0.5
Theoretical λ = 0.9
Theoretical λ = 0.95
Theoretical λ = 0.99
Fraction of Queues with at least that size
Queues Length


Practical Queues Length | n: 100 | d: 5
Theoretical λ = 0.5
Theoretical λ = 0.9
Theoretical λ = 0.95
Theoretical λ = 0.99
Fraction of Queues with at least that size
Queues Length


Practical Queues Length | n: 100 | d: 5
Theoretical λ = 0.5
Theoretical λ = 0.9
Theoretical λ = 0.95
Theoretical λ = 0.99
Fraction of Queues with at least that size
Queues Length


Practical Queues Length | n: 100 | d: 10
Theoretical λ = 0.5
Theoretical λ = 0.9
Theoretical λ = 0.95
Theoretical λ = 0.99
Fraction of Queues with at least that size
Queues Length

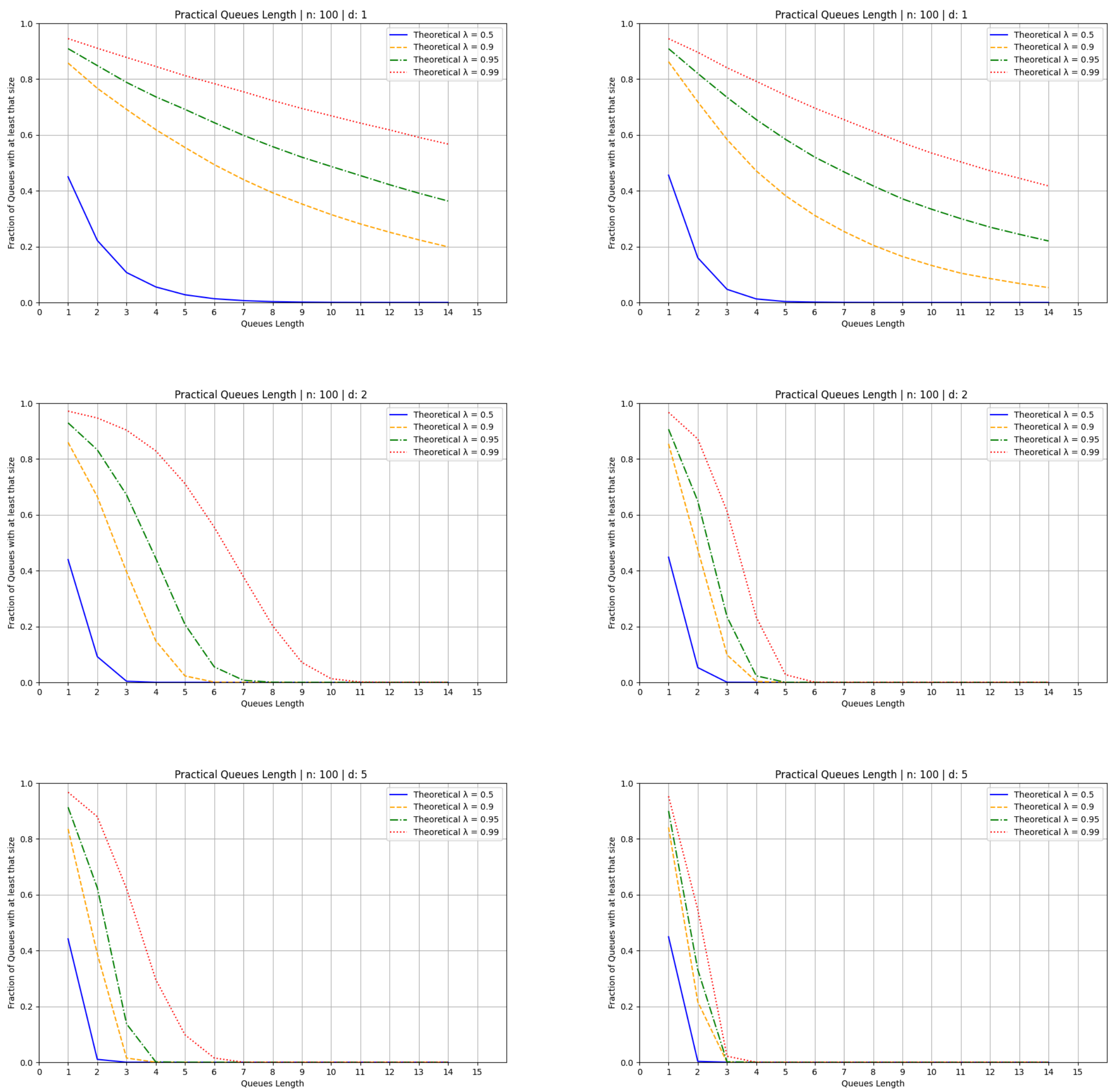

Practical Queues Length | n: 100 | d: 10
Theoretical λ = 0.5
Theoretical λ = 0.9
Theoretical λ = 0.95
Theoretical λ = 0.99
Fraction of Queues with at least that size
Queues Length

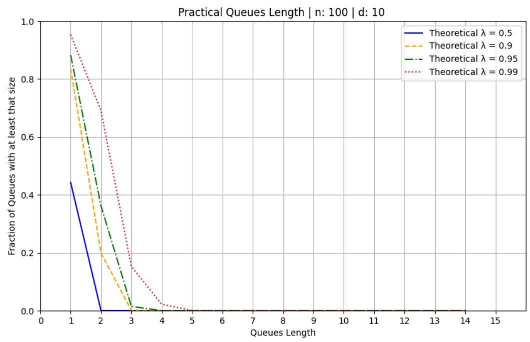
Since shape = 1 in the Weibull distribution corresponds to an exponential distribution, we expect similar behavior before and after applying Weibull.

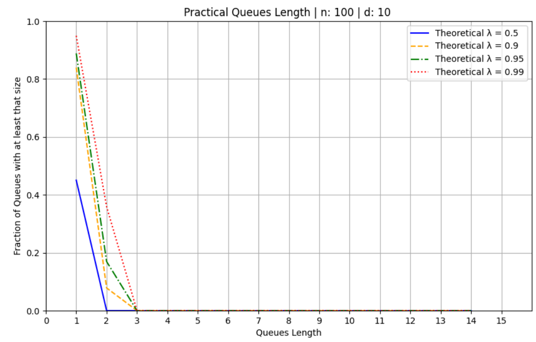
The overall trend remains nearly the same between both sets of plots.

No shift is observed between the left and right plots, which confirms that Weibull (shape = 1) does not change queue behavior.

Shape > 1

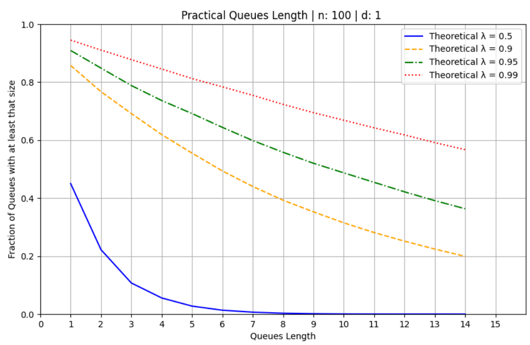


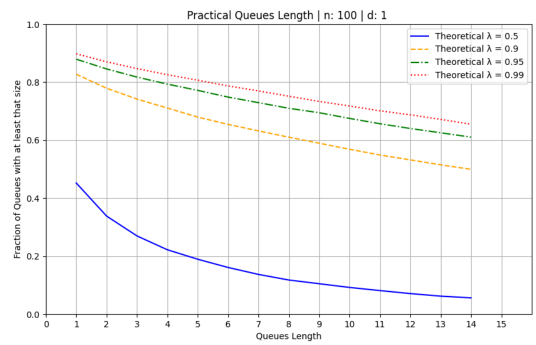


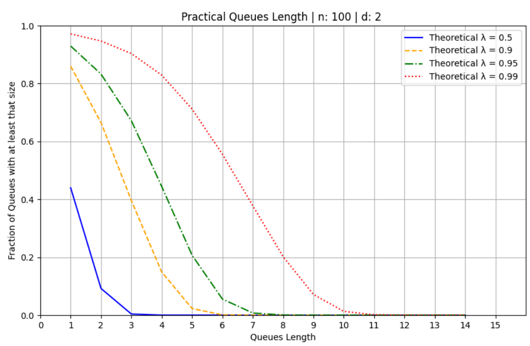


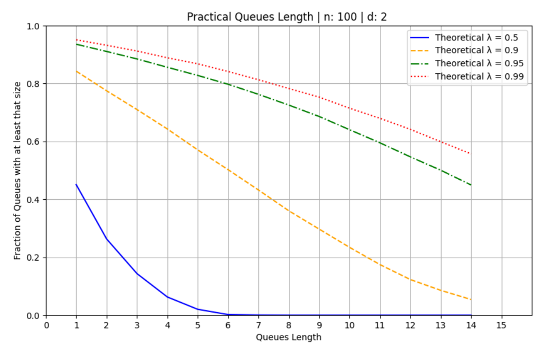
The curves drop off more sharply, indicating that the number of long queues is reduced faster than in the exponential case. This aligns with the behavior of a bell-shaped Weibull distribution, where extreme values are less frequent.

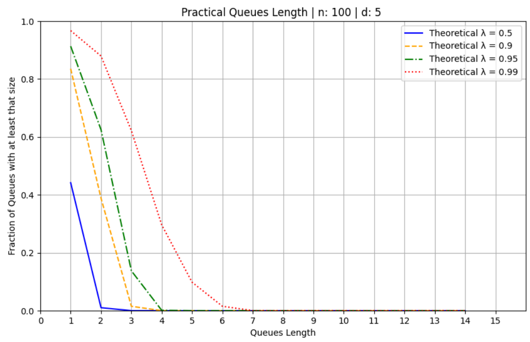
Shape < 1

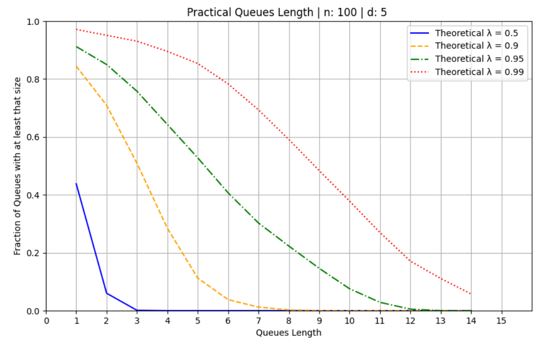


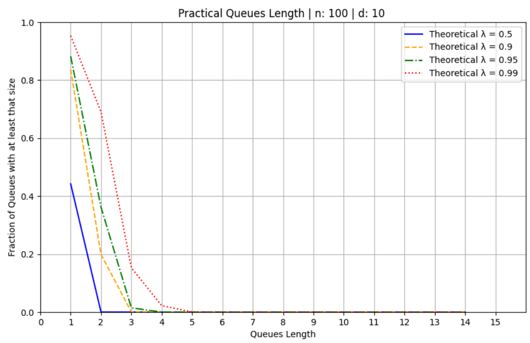


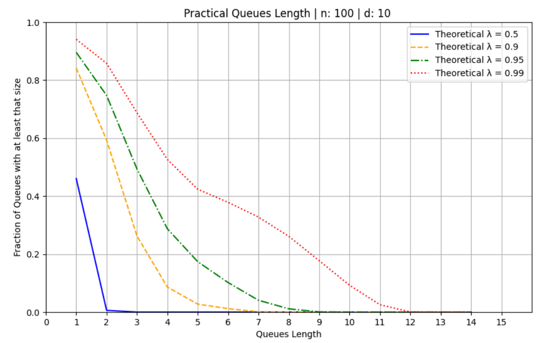












Using a Weibull distribution with shape < 1 negatively affects queue performance by introducing extremely long jobs, which increase congestion and slow down processing.
Even at high d, the system cannot fully compensate for the impact of long-tail jobs, meaning queues are more likely to grow.
The effect of increasing d is still beneficial, but it does not eliminate the inefficiency caused by heavy-tailed distributions.

Job's Priority Extention:

The current implementation lacks support for job priorities, preventing the prioritization of more urgent tasks. The goal is to ensure high-priority jobs are placed at the top of the heap queue. To evaluate the impact of the modification.

How job priority works:

1. ADD a priority Attribute to jobs:

```python
class Arrival(Event):
    def __init__(self, job_id, priority):
        self.id = job_id
        self.priority = priority
```

Every job has a priority value, which is stored when the job arrives.

2. Store jobs in queues with their priority:
   Each job has a priority (1-4) assigned at arrival.
   Lower values mean higher priority (priority 1 is the highest, 4 is the lowest).

```
sim.queues[queue_index].append((self.priority, self.id))
```

   Jobs are now stored in queues in the format (priority, job_id).

3. Select the highest-priority job from each queue:

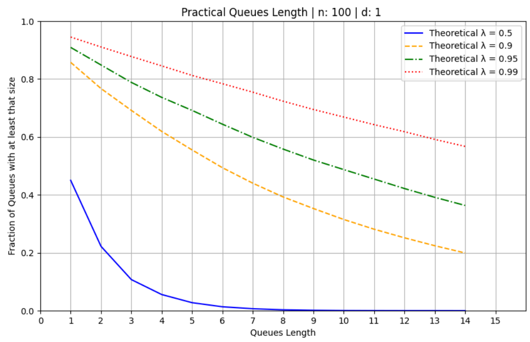
```
if sim.queues[queue_index]:
    priority, new_job = min(sim.queues[queue_index]) # Find the job with the highest priority
    sim.queues[queue_index].remove((priority, new_job)) # Remove the selected job
```

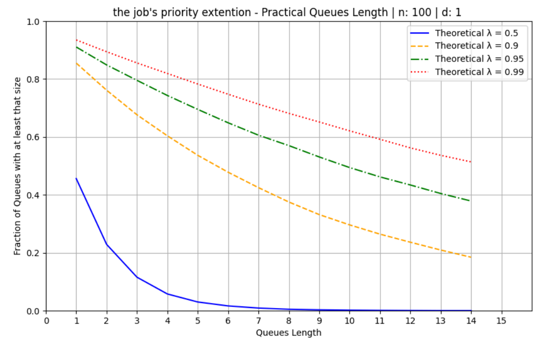
   Instead of popleft() (FIFO), min() selects the job with the lowest priority number (highest priority). The selected job is then removed from the queue before execution.

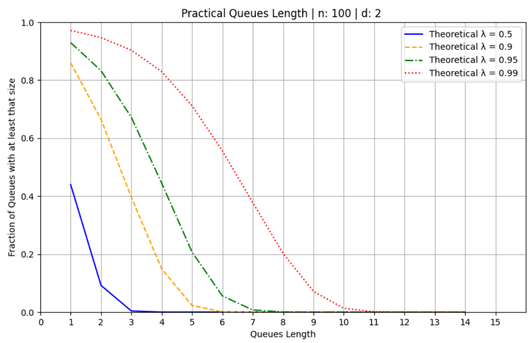
4. Pass job priority when scheduling completion and arrival:

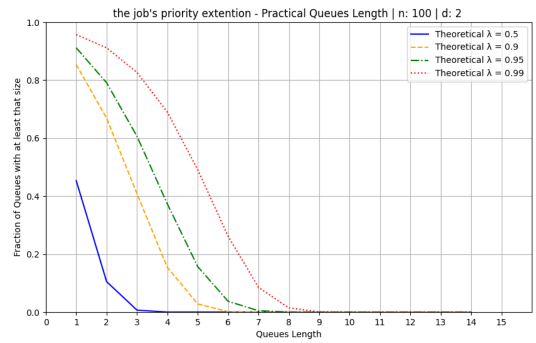
```
def schedule_arrival(self, job_id , priority):
    self.schedule(expovariate(self.arrival_rate), Arrival(job_id, priority), priority)

def schedule_completion(self, job_id, queue_index,priority):
    self.schedule(expovariate(self.mu), Completion(job_id, queue_index),priority)
```

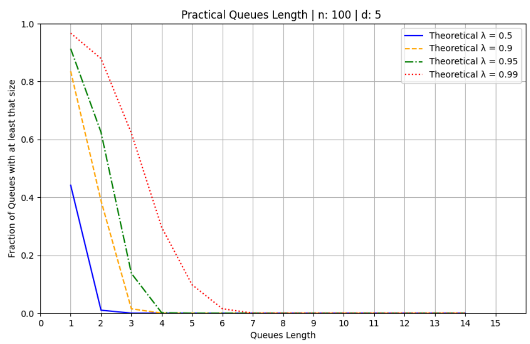
we compared the results from running the simulator both without and with the priority concept.

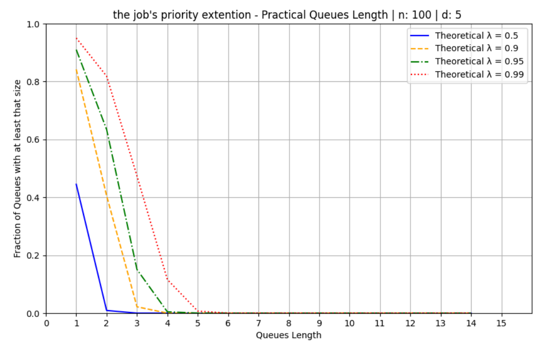


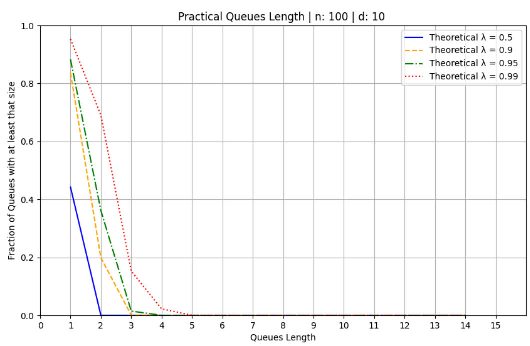


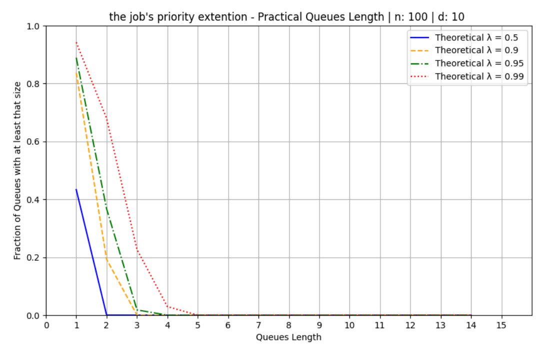












Even though priority values were assigned randomly, they still improved performance. The reason is that introducing variability into scheduling helps prevent worst-case scenarios caused by FIFO. let’s explore why:

1. **Random Priority Disrupts FIFO Bottlenecks:**
   Without Priority (FIFO Execution - The left-side plots):
   - Jobs always execute in strict arrival order.
   - If a long job enters first, all shorter jobs must wait behind it, leading to queue congestion.
   - This can increase waiting time unnecessarily, especially when d is small.

   With Random Priority (Even Without Intelligence - the right-side plots)
   - Jobs no longer execute strictly FIFO.
   - Some jobs jump ahead of long jobs randomly, which reduces the chance of excessive delays.
   - This prevents long jobs from blocking the queue entirely.

   So, even though priority is random, it breaks FIFO, preventing queue bottlenecks.

2. **Small d Benefits the Most from Random Priority:**
   When d is small, queue selection is less optimal, leading to unbalanced queue loads.
   - **Without priority**, bad queue assignments persist (long jobs block execution).
   - **With random priority**, jobs move through the system more dynamically, preventing long queues from getting stuck.

   Even without smart priority assignment, this added variability improves efficiency.

LWL Extention:

The current implementation follows the supermarket mode, where jobs are assigned to the queue with the least number of jobs among d randomly chosen queues. The goal now is to implement a least work left (LWL) model, where jobs are assigned to the queue with the least total processing time remaining, rather than just the smallest number of jobs.

Added self.service_times and self.start_times:

These two variables were introduced to accurately track the remaining workload in each queue.

Self.service_times stores the total service time (processing time) for each job. When a new job enters the system, we generate its total service time using a Weibull distribution (in Arrival class).

```
sim.service_times[self.id] = sim.service_gen()
```

Self.start_times stores the time when a job started execution. When a job begins execution, we record the simulation time in self.start_times (in completion class).

```
self.start_times[job_id] = self.t
```

Added a calculate_workload function to compare queue workload:

```
def calculate_workload(self, queue_index):
    if self.running[queue_index] is not None:
        job_id = self.running[queue_index]
        total_service_time = self.service_times[job_id]
        start_time = self.start_times[job_id]
        elapsed_time = self.t - start_time
        remaining_service_time=total_service_time - elapsed_time  #the remaining work of the currently running job
    else:
        remaining_service_time = 0
    # the total service time of jobs waiting in the queue
    queue_workload = sum(self.service_times[job] for job in self.queues[queue_index])
    adjusted_workload = queue_workload + remaining_service_time
    return adjusted_workload  #Returns the total remaining workload of the queue
```

Checks if there is a job currently running in the queue. If yes, it computes the remaining work of the currently running job. Then sums up the service time for all jobs waiting in the queue. Finally, it returns the total remaining workload of the queue.

Modified queue selection in Arrival class:

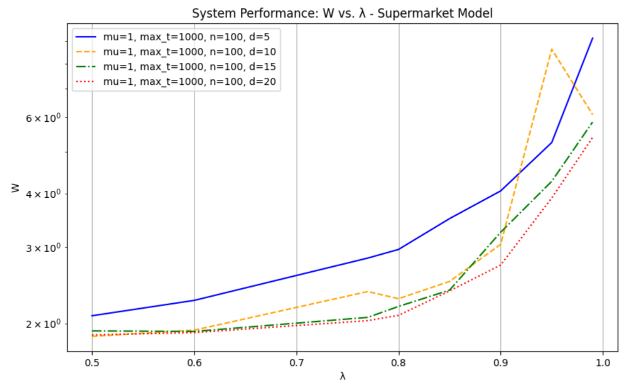
```
samples = sample(sim.queues, sim.d)
selected_list = min(samples, key=lambda lst: sim.calculate_workload(sim.queues.index(lst)))
```

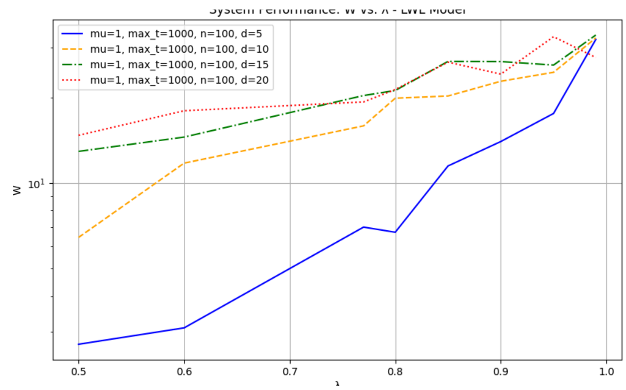
Instead of checking queue length, we now check the total work left so jobs will be assigned to the queue that can compete them fastest rather than the shortest queue.

Comparison Between Supermarket Model and LWL Model:

The two plots illustrate W (average time spent in the system) vs. λ (arrival rate) for different values of d:

The left plot is for the supermarket model and the right one is for the LWL model.





We expected the Least Workload First (LWL) model to perform better than the Supermarket (Shortest Queue First) model and to reduce the average waiting time W since LWL assigns jobs to the queue with the least remaining workload. However, the results from the plots contradict this expectation; in some cases, the Supermarket model performs better and results in lower waiting times compared to LWL.

This contradiction can be explained by the fact that while LWL makes an optimal decision at each moment, these local decisions can collectively lead to an imbalanced workload distribution over time.

- If a queue is repeatedly chosen as the least loaded queue multiple times in a row, its workload will increase rapidly, leading to congestion in that queue.
- Some queues may always have a high workload and, as a result, will rarely be selected, leading to workload imbalance in the system.
- In the Supermarket model, selecting the shortest queue ensures a more uniform load distribution among all queues. However, in LWL, increasing d is not always beneficial because it can lead to excessive workload concentration on specific queues.