Computer aided simulations and performance evaluation lab N1

Matteo Latino S269317

1 Introduction: Queuing system

The aim of this laboratory is to simulate a queuing system and, whenever is possible, the obtained result will be compared with the theoretical ones. To simulate a queue is used a next event approach, so the system will jump forward through discrete time instant, in particular i decide to use an event scheduling approach. First is decided when in the future the event will happen and then is scheduled its execution, by adding the event on the Future Event Set (FES). As a starting point, I use the code that was provided to us during the course. To use an event scheduling must be identified:

- The entities in the system: in this case are the client, the server and the queue itself.
- The state variables: is only one, the number of clients in the queue.
- The events that change the value of the state variable: are the arrival (A) of a client in the queue and the departure (D) of a client from the queue.
- The results we are interested in: are the average number of clients in the queue, the average time to cross the queue, the utilization level of the servers and, if possible, the probability of losing a client.

2 Task 1: M/M/1 and M/G/1 queue

In the $M/M/1$ queue, there is only one server and the queue has an infinite capacity of the waiting line. Moreover, both the distribution of the arrival and of the service time are exponentially distributed.
Like the $M/M/1$ also the $M/G/1$ queue has only one server and an infinite capacity of the waiting line, the main difference consists of the distribution of the service time which can be anyone. In this lab I will assume an uniform distribution for the service time of the $M/G/1$ queue.
Since both queues have an infinity capacity, it's not possible to lose a client.

2.1 Input variables

The input variables of the simulator are:

- The type of distribution for the service time, with the parameter necessary to define it.
- The load on the system computed as λ/μ.
- The confidence level for the confidence interval.
- The initial seed, the number of runs and the duration of the simulation.

2.2 Output metrics

The output metrics of the simulator are:

- The average number of clients in the queue.
- The average time to cross the queue.
- The utilization level of the server.
- The confidence interval for the previous measures.

2.3 Structure of the simulator

2.3.1 Data structure

To manage the Future Event Set is used a *PriorityQueue* from the *queue* library, each entry will be a tuple: *(time of the event, type of the event)*, the time of the event indicates its priority with respect to the other events. When a tuple is extracted from the queue, the one with the smallest time is returned and it coincides with the next event that must happen. Then, for each entity identified before is built up an ad-hoc class that will implement the methods required to manage it, more in details:

- The class Client: is the smallest one, each client has only two parameters: the arrival and the departure time. The only required methods are the setter and getter methods to update and access these values.
- The class Server: its attributes are: the id, the type of service time distribution and the number of people served by that server. The main method of this class is the *serve*() method, it receives the actual time and the FES to schedule the departure for the next client in the queue, moreover, it returns the duration of the service.
- The class My_queue: its attributes are: the number of departures from the queue, the number of arrivals in the queue, the list of the clients in the queue (both in services and not) and other attributes that are required for the computation of some measures. The main methods of this class are:
  - The arrival method: when an arrival event occurs a client is added in the clients list, the state variable is updated, and a new arrival is scheduled in the FES.
  - The departure method: when a departure event occurs the first client from the clients list is extracted and the state variable together with all the measures is updated, this method returns the client that just left the system.

The script also contains another class called $Measure\_plot$ which is used to collect the data obtained from the different runs of the simulator. Like the Client class also this contains only the getter and setter methods.

2.3.2 Algorithm

The simulator mainly consists of a loop on which, at each iteration an event is extracted from the FES, and all the action related to that event take place.
For the arrival: first is invoked the arrival method from the My_queue class, then, if the client is the only one in the system is called the *serve*() method from the *Server* class to schedule its departure from the queue.
For the departure: first is called the *depart*() method of the My_queue class and a client leaves the queue. Then, if there are other clients in the queue, the server will start the service for the next one by scheduling its departure in the FES.

2.4 Results

The following graphs shows the obtained results for different values of the load. The service time distribution for the $M/M/1$ queue is an exponential with a parameter μ equal to $\frac{1}{30}$, while the $M/G/1$ queue uses an uniform distribution between 1 and 59, so in both cases the mean value for the service time is equal to $30s$.

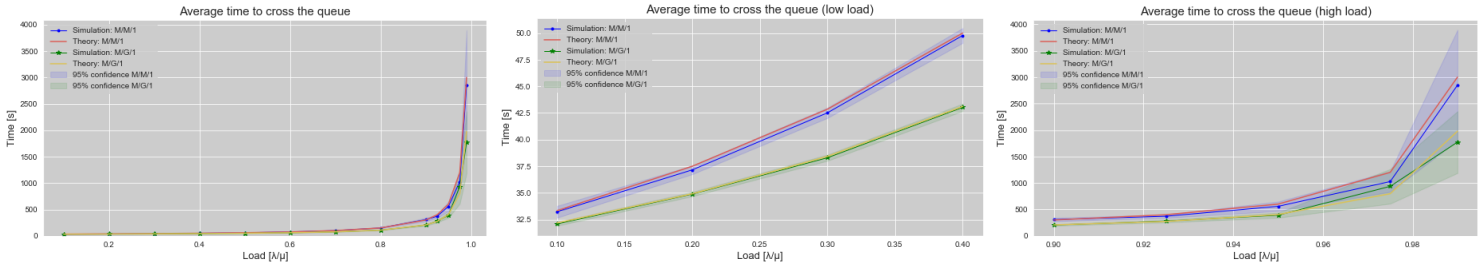
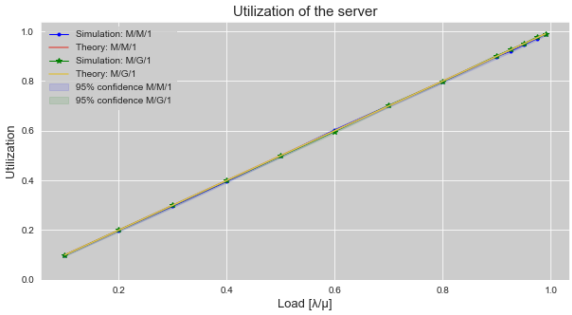


Figure 1: Time to cross the queue

To better understand the value in the graph, the low and the high load cases are shown on 2 different graphs. The results obtained from the simulator are very near to the theoretical one, in fact, as expected with the increase of the load on the queue the average time to cross the queue tends to increase. It's also possible to see that the $M/G/1$ queue tends always to perform better than the $M/M/1$ queue (with the same load, lower time to cross the queue). This result is not unexpected, thanks to the $Pollaczek - Khintchine$ formula it is known that the second moment of the distribution has an important impact on the queue performance, large values of the variance imply long queues and large times in the queue. In this case the exponential distribution has a variance equal to 900, while the uniform distribution has a variance equal to 280.33, so as consequence the last one will perform better.



Regarding instead the utilization of the server, it can be seen that the behaviour is almost the same for both queues, and is equal to the load in the queue. It was computed as 1 minus the division between the number of clients that found the server idle when they arrive on the queue and the total number of arrivals.

3 Task 2: M/M/1/B and M/G/1/B queue

The main difference with respect to the previous queue is that, the waiting line of the queues, now has a max capacity equal to B. If a new client arrives when there are already B clients in the system he will not be served, this event corresponds to the loss of a client.

3.1 Input variables

The input variables of the simulator are the same as the previous one, plus B which is the max capacity of the waiting line.

3.2 Output metrics

The output metrics of the simulator are the same as the previous one plus the probability of losing a customer, with the relative confidence interval.

3.3 Structure of the simulator

3.3.1 Data structure

All the class and data structure described for the previous simulator will be reused, the only difference is in the arrival method of the *My_queue* class. If the waiting line is already full, the arrival result on a loose of the client, otherwise the customer is added to the clients queue and the state variable is updated, in both cases this method will schedule the next arrival in the FES. The class now also has a counter for the number of clients lost.

3.3.2 Algorithm

The structure of the algorithm of the simulator is almost the same as before, the only difference consists of a check to manage the case on which the queue has 0 capacity, in that case in the system there is only 1 client which is in service, further arrival will result in a loss. To better see how the limited capacity of the queue affects the behaviour of the system, the load is fixed to 0.95 and at each iteration the capacity is increased.

3.4 Results

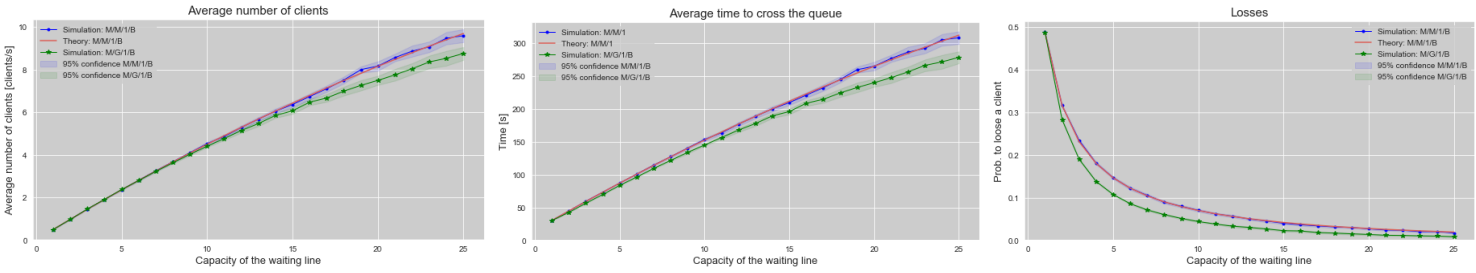


Figure 2: M/M/1/B and M/G/1/B

The obtained results are very near to the theoretical one, increasing the size of the waiting line leads to a lower probability to lose the client and a higher time to cross the queue. As before, the $M/G/1/B$ queue results to perform better than the $M/M/1/B$ queue. From these graphs it is also possible to see that, if we limit the size of the waiting line, the time to cross the queue tends to reduce at the price of losing some customers with a given probability that, however, could be very small.

4 Task 3: M/M/m/B queue

The simulator is now enhanced by adding the possibility of using more than one server, and also to use different policies to choose the server that will serve the next client. The cases that will be analyzed are the following:

- All the servers have the same capacity, and the next customers will be served by the first available server.
- The servers have different capacity, when a customer arrive in the queue he can choose the server by using one of the following policy:
  - The fastest one between the actual idle servers.
  - The least used between the actual idle servers.

4.1 Input variables

The simulator does not receive anymore the information relative to the service time distribution but it receives directly the list of servers that will serve the clients in the queue and the policy to choose the server.

4.2 Output metrics

The output metrics of the simulator are the same as the previous one.

4.3 Structure of the simulator

4.3.1 Data structure

Now it's necessary to manage more than one server, so both the FES and the different classes will be updated. In particular:

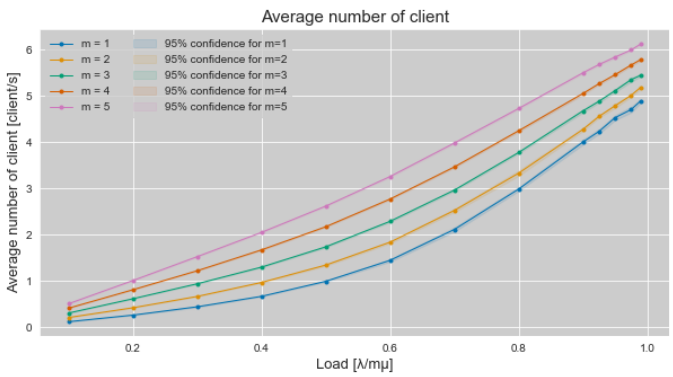
- The tuple in the FES, now contains also the Id of the server that schedules the departure and the Id of the client that will leave the queue. If the event is an arrival both Id are set to -1.
- To the server class is added the *state* attribute. It is a binary attribute, if it's 0 the server is idle, otherwise it's value is 1 and the server is busy. Now, when the serve method is called, the state attribute is set to 1, while to reset its value to 0 is used an ad-hoc method called $free\_server$. It is also added the *clean_server* method to reset the counters and the state of the server between the different runs of the simulator.
- The attribute *state* is added also to the client class to know if a given client is on the waiting line ($state = 0$) or is already in service ($state = 1$). To the client class is also added the Id attribute.
- The class My_queue has 2 new methods, one to find the Id of the next client to serve, and one to set a given client into the *in service* state. Moreover, the departure method also receives the Id of the client that will leave the queue.

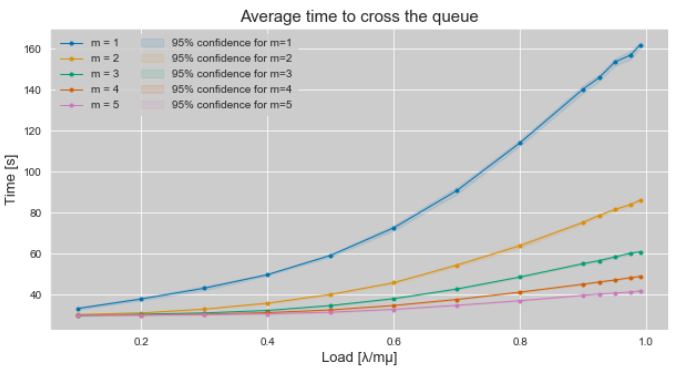
4.3.2 Algorithm

As in the previous simulator, the algorithm consists of looping through the FES until it reaches the time limit for the simulation, the main difference consist of:

- Before scheduling the next departure, a policy is applied to choose the server that must serve the next client and the Id of that client must also be retrieved. After the scheduling of the departure, the client state is set to *in service*.
- The utilization is computed by counting the average number of idle servers in the queue, in time.

4.4 Results





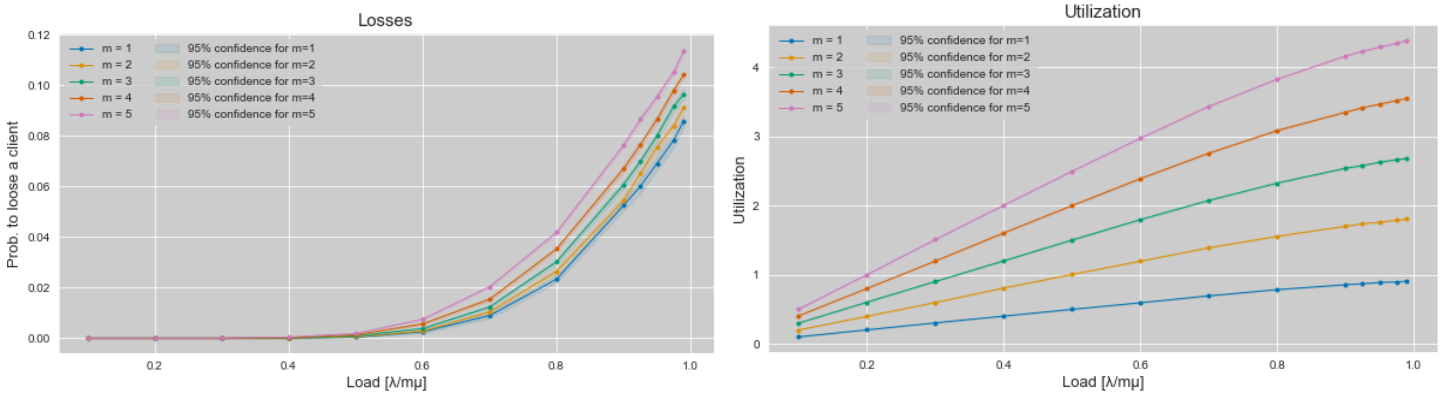


Figure 3: M/M/m/B queue with B=10

From the graphs above it is possible to see that:

- With the same load, a greater number of servers tends to reduce the time for the service.
- A queue with more servers is able to serve more clients in the same amount of time. Indeed, at parity of load, a queue with a greater number of servers also has a higher arrival rate of the clients.
- The probability to lose a client result to be lower with a lower number of servers. Probably the smaller arrival rate has a huge influence in this parameter.
- As expected, the utilization increases with the number of available servers.

For the case on which the servers have different capacity, I have assumed to have only 3 servers with average service time equals to: $15s$, $30s$ and $45s$ and a capacity of the queue equal to 10 clients. The following graph show the obtained results.

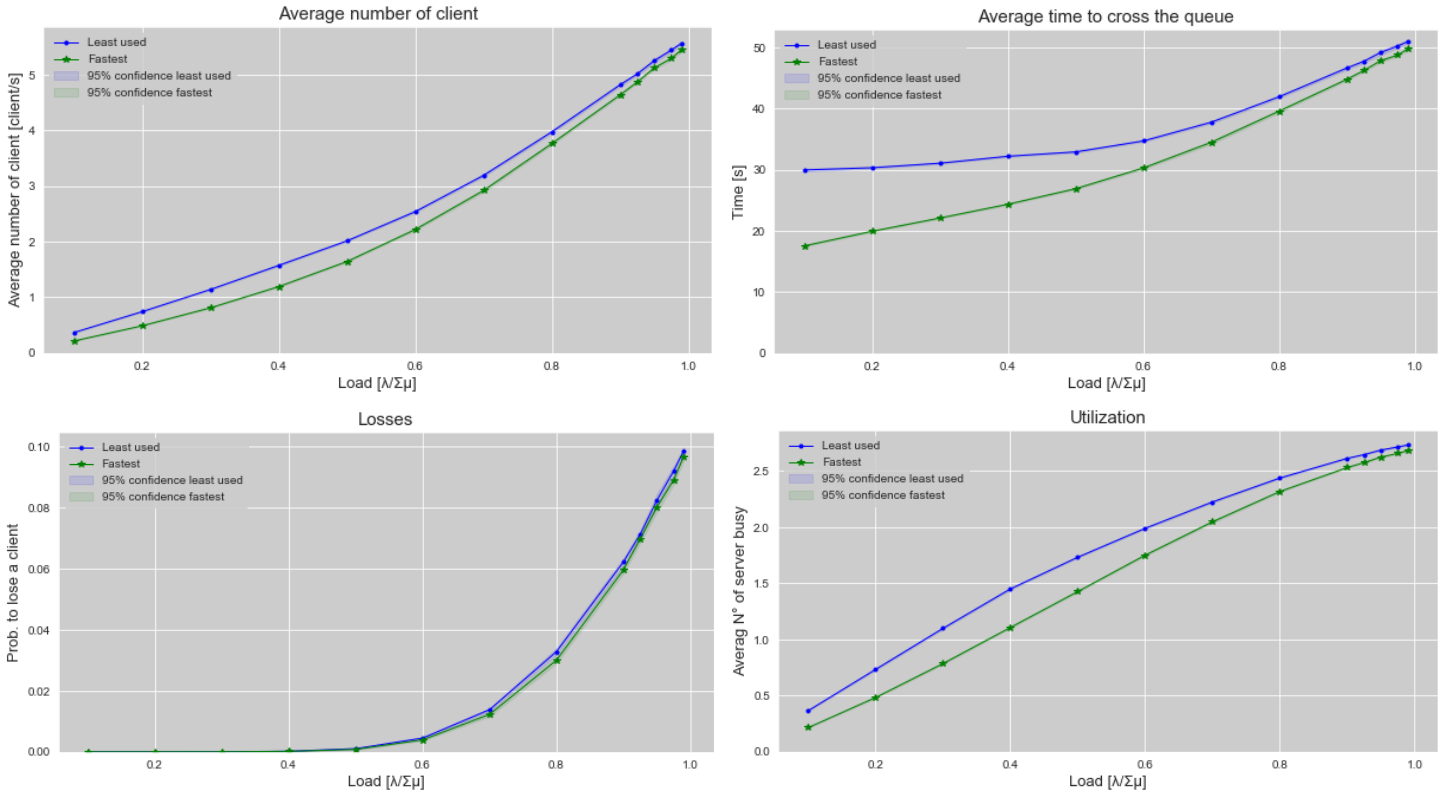


Figure 4: M/M/m/B queue. Comparing different policy

Is possible to see that the fastest policy works better than the least used policy, especially at low load. Indeed, with this policy we try to maximize the use of the fastest server and use the slowest one only if all the other servers are already busy, while with the least used policy may occur that a client is served by a slower server even if a faster one is idle.